Nagyteljesítményű mikrokontrollerek és interfészek

Elektronikus jegyzet

I. rész

Nagyteljesítményű mikrokontrollerek

1–4. fejezet

Tárgyfelelős: **Dr. Tevesz Gábor**
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117 Budapest, Magyar tudósok körútja 2.
Q ép. B szárny II. em. B216.
Tel: 463-2881
E-mail: tevesz@aut.bme.hu

Szerző(k): **Dr. Gál Tibor, Dr. Tevesz Gábor, Kiss Domokos**
BME Automatizálási és Alkalmazott Informatikai Tanszék

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki mesterszak (MSc)
Számítógép alapú rendszerek specializáció hallgatói

Utolsó módosítás: 2021. 04. 06.

Verzió: 2.5

Copyright © 2021 / BME AUT

COPYRIGHT

Jelen dokumentum a BME Villamosmérnöki és Informatikai Kar hallgatói számára készített elektronikus jegyzet. A dokumentumot a Nagyteljesítményű mikrokontrollerek és interfészek c. tantárgyat (BMEVIAUMA07) felvevő hallgatók jogosultak használni, és saját céljukra 1 példányban kinyomtatni. A dokumentum módosítása, bármilyen eljárással részben vagy egészben történő másolása tilos, illetve csak a szerző(k) előzetes engedélyével történhet.

Copyright © 2021 / BME AUT

Tartalom

Bevezetés 5

1 Számítógép-architektúrák 7

1.1 Számítógép-generációk 7

1.2 A számítógép-architektúra fogalma 8

1.2.1 Architektúra, implementáció és realizáció 10
1.2.2 Utasításkészlet-architektúra 11
1.2.3 Mikroarchitektúra 13

1.3 A jó architektúra jellemzői 15

1.4 Számítógépcsaládok 19

1.5 A hardver és szoftver kapcsolata 20

1.5.1 A dinamikus-statikus interfész 21
1.5.2 Fordítás és értelmezés 22

Felhasznált és ajánlott irodalom 24

2 Mikroarchitektúrák 25

2.1 Processzorok teljesítőképessége 25

2.1.1 Iron Law 27
2.1.2 A teljesítőképesség mérése 27
2.1.3 A teljesítőképesség növelése 31

2.2 Csővezetékes processzorok 40

2.2.1 A csővezeték működési elve 41
2.2.2 Aritmetikai csővezeték 45
2.2.3 Utasítás-csővezeték 51
2.2.4 Függőségek és hazárdok a csővezetékben 56
2.2.5 Valós utasítás-csővezeték példák 66

2.3 Szuperskalár processzorok 69

2.3.1 Térbeli utasításszintű párhuzamosítás 69
2.3.2 A szuperskalár csővezetékek fajtái 74
2.3.3 A szuperskalár csővezetékek dinamikus viselkedése 78
2.3.4 Függőségek gyakorlati hatása a szuperskalár csővezetékben 85

2.4 VLIW processzorok 100

2.4.1 Történet 101
2.4.2 Előnyök és hátrányok 102

2.5 Többszálúság egy processzoron belül 104

2.5.1 Processzoron belüli többszálúság típusai 104

Copyright © 2021 / BME AUT

2.5.2 Az Intel Hyper-Threading Technology architektúra ... 108
2.6 Esettanulmány: Intel x86 mikroarchitektúrák ... 111
2.6.1 Az Intel P6 mikroarchitektúra ... 112
2.6.2 Az Intel Nehalem mikroarchitektúra ... 119
Felhasznált és ajánlott irodalom ... 125

3 ARM processzorok ... 128

3.1 Az ARM architektúra története és fejlődése ... 128
3.1.1 ISA generációk áttekintése ... 131
3.2 Az ARM utasításkészlet-architektúra ... 134
3.2.1 Az ARM ISA általános jellemzői ... 134
3.2.2 Az ARM ISA kiegészítései ... 143
3.2.3 Az ARMv8-A ISA sajátosságai ... 157
3.3 ARM Cortex-A mikroarchitektúrák ... 170
3.3.1 ARM Cortex-A8 ... 170
3.3.2 ARM Cortex A9 ... 171
3.3.3 ARM Cortex A72 ... 172
Felhasznált és ajánlott irodalom ... 174

4 Többprocesszoros rendszerek ... 176

4.1 Sebességnövekedés ... 176
4.2 Osztályozás ... 177
4.2.1 Feladat-hozzárendelési mód szerinti osztályozás ... 177
4.2.2 Processzorok közötti kapcsolat szerinti osztályozás ... 178
4.2.3 Memóriamegosztottság szerinti osztályozás ... 179
4.2.4 Processzorok típusa és kezelése szerinti osztályozás ... 182
4.3 Gyorsítótárak multiprocesszoros környezetben ... 183
4.3.1 Gyorsítótárak ... 183
4.3.2 Többszintű gyorsítótárak ... 188
4.3.3 Gyorsítótár-koherencia ... 191
4.3.4 A MESI protokoll ... 196
4.4 Az Intel Nehalem mikroarchitektúra multiprocesszoros tulajdonságai ... 200
4.4.1 Memóriaszervezés ... 200
4.4.2 Cache-koherencia ... 202
4.5 Többmagos ARM processzorok ... 205
4.5.1 Cache-koherencia ... 206
4.5.2 Az ARM big.LITTLE technológia ... 207
Felhasznált és ajánlott irodalom ... 210

Copyright © 2021 / BME AUT

Bevezetés

Ez a tantárgy a Budapesti Műszaki Egyetem Villamosmérnöki és Informatikai Kar nappali tagozatán kerül előadásra azon villamosmérnöki mesterképzésben résztvevő hallgatók számára, akik képzési szakterületként a Számítógép alapú rendszerek főspecializációt választották.

A tantárgy célja, hogy széleskörű ismereteket nyújtson a számítógépes rendszerek és a nagyteljesítményű mikrokontrollerek architektúráiról, ill. építőelemeiről. A hagyományos architektúrák elemzését követően bemutatja a teljesítőképesség növelése céljából megalkotott modern megoldásokat, amelyek a végrehajtás párhuzamosításának különféle lehetőségeit foglalják magukban. Speciális architektúrák (ARM, GPGPU) jellemzőit is tárgyalja, s összeveti ezeket a szoft- és hardprocesszoros SoC eszközökkel. A tárgy hallgatói megismerkednek a teljesítményt és megbízhatóságot növelő, s a fogyasztást csökkentő módszerekkel. Részletesen foglalkoznak az irányítórendszer részeit összekapcsoló nagysebességű buszrendszerek (USB, SATA, PCI-Express) jellemzőivel, működésével.

A tantárgy három fő részből, ezeken belül pedig 15 fejezetből áll a következők szerint:

I. rész – Nagyteljesítményű mikrokontrollerek

Az **1. fejezetben** áttekintjük a számítógép-architektúrák alapvető fogalmait és definícióit. Különbséget teszünk az utasításkészlet-architektúra és a mikroarchitektúra fogalma között, és a továbbiakban mindkettőt szem előtt tartva tárgyaljuk a különböző bemutatott architektúrák tulajdonságait.

A **2. fejezetben** a processzorok mikroarchitektúráival foglalkozunk. Áttekintjük a teljesítőképesség növelését célzó megoldásokat, amelyek a végrehajtás különböző szintű párhuzamosítására fókuszálnak. Ezek közé tartozik a csővezetékes végrehajtás alkalmazása, a szuperskalár szervezés, a sorrenden kívüli végrehajtás és a szálszintű párhuzamosság hardveres támogatása egy processzormagon belül.

A **3. fejezetben** az ARM processzorokat mutatjuk be. Áttekintjük az ARM utasításkészlet-architektúra általános jellemzőit és kiegészítéseit, valamint a legújabb generációkban bevezetett újdonságait. Áttekintjük az ARM processzorok mikroarchitektúráit, különös tekintettel a nagyteljesítményű alkalmazásokra szánt ARM Cortex-A család tagjaira.

A **4. fejezetben** a többprocesszoros rendszerek és többmagos processzorok felépítését és tulajdonságait tárgyaljuk. Áttekintjük a többprocesszoros rendszerek csoportjait, és reprezentatív rendszerek példáin keresztül megismerjük az alkalmazási lehetőségeiket. Külön kitérünk a közös memóriát használó, szorosan csatolt rendszerekben felmerülő adatkonzisztencia kérdéskörére és megoldási lehetőségeire.

Az **5. fejezetben** a processzorok egy speciális csoportjával, a grafikus processzorokkal ismerkedünk meg. Áttekintjük a grafikai alkalmazások fejlődése mentén a grafikus processzorok generációit, valamint az architektúrák fejlődését. Kitérünk a grafikus

Copyright © 2021 / BME AUT

processzorok általános alkalmazásának lehetőségeire (GPGPU) és az ezeket kiemelten támogató GPU architektúrák tulajdonságaira.

II. rész – Buszok és interfészek

A **6. fejezetben** az interfészek és buszok alapfogalmait, valamint mechanikai, elektromos és logikai jellemzőit tekintjük át. Általános szempontrendszert állítunk fel, amely alapján a specifikus interfészek vizsgálhatók és jellemezhetők.

A **7. fejezetben** a számítógépek nagy adatmennyiséget forgalmazó háttértárolóinál alkalmazott SATA interfésszel foglalkozunk.

A **8. fejezet** bemutatja a külső perifériák illesztésére széles körben alkalmazott USB interfész felépítését, tulajdonságait, működését.

A **9. fejezet** a PCI buszcsalád jellemzőit tekinti át, és részletesen foglalkozik a PCI-Express busz jellemzésével, különböző rétegeiben alkalmazott protokollok bemutatásával.

A **10. fejezet** a korszerű soros buszok kapcsolódását mutatja be, amelynek körében röviden bemutatjuk a DisplayPort, HDMI és Thunderbolt interfészeket, valamint ezek kapcsolódását egymáshoz és az előző fejezetekben tárgyalt USB és PCI-Express buszokhoz.

III. rész – System-on-chip rendszerek

A **11. fejezet** az egylapkás rendszerek általános tulajdonságait, típusait, felépítését mutatja be. Tárgyalja az ezen rendszerek létrehozásában központi szerepet játszó IP (intellectual property) blokkok leírási módjait és típusait.

A **12. fejezet** az egylapkás rendszerek megvalósításában elterjedten használt FPGA-kat ismerteti. A fejezet részletesen bemutatja a Xilinx cég 7-es sorozatú FPGA családját.

A **13. fejezet** a SoC rendszerek komponensei közötti kommunikációs buszokat mutatja be. Elsősorban az ARM AMBA buszcsaládhoz tartozó megoldásokat részletezzük a fejezetben.

A **14. fejezet** az egylapkás rendszerekben megvalósítható processzormagokat tekinti át. Bemutatunk néhány fontosabb soft és hard processzoros SoC rendszert a fejezet során.

A **15. fejezet** egy konkrét programozható egylapkás rendszer (PSoC), a Xilinx Zynq-7000 család architektúrájának és alkalmazási lehetőségeinek bemutatásával zárja a tantárgy tematikáját.

A tantárgy anyaga előadásokon és gyakorlatokon kerül ismertetésre. Az előadások és a gyakorlatok az anyag ütemében váltogatják egymást, a gyakorlatokon példák és esettanulmányok formájában kerül elmélyítésre az előadásokon elhangzott elméleti tananyag.

A tantárgyat felvevő hallgatóknak egy zárthelyi feladatot kell teljesíteniük a félév során, majd a félév végén írásbeli vizsgát kell tenniük.

Copyright © 2021 / BME AUT

1 Számítógép-architektúrák

A számítástechnika viszonylag fiatal és rendkívül gyorsan fejlődő szakterület. Az eredményei gyorsan változnak, ami elsősorban a nagymértékű technológiai fejlődésnek köszönhető. Az első elektronikus számítógép (ENIAC, azaz Elecronic Numerical Integrator and Computer) 1945-ben készült el az amerikai hadsereg ballisztikai kutatólaboratóriuma számára, és forradalmasította a tudományos kutatásokban (itt konkrétan lövedékek ballisztikus röppályaadatainak meghatározásában) használatos numerikus számítások elvégzését. Ugyanis a mérések szerint 2400-szor gyorsabban volt képes a számítások elvégzésére, mint az akkor külön erre foglalkoztatott matematikusok. Egy komplett termet elfoglalt, közel 30 tonnát nyomott és 150kW volt a felvett elektromos teljesítménye. Ezzel szemben napjainkban a kezünkben tartott okos eszközök processzorai sok nagyságrenddel meghaladják ezt a számítási kapacitást.

1.1 Számítógép-generációk

A számítógépek fejlődése napjainkig öt generációra osztható, ahol az új generációk mindig valamilyen jelentős technológiai fejlesztést jeleznek. A felosztás nem egységes, különböző szerzők máshová teszik a határokat, elsősorban a negyedik és ötödik generáció között.

1. generáció – Elektroncsövek (~1945–1956)

Az első elektronikus számítógépek (pl. ENIAC, IBM 701, Princeton IAS) áramkörei elektroncsöveken alapultak, a memóriát relékkel valósították meg, az alkatrészeket szigetelt vezetékekkel kötötték össze, és a programozás közvetlen gépi nyelven történt. A programot lyukkártyákon vagy lyukszalagon lehetett beadni, a kimenetet is ilyenekre nyomtatva szolgáltatták. Méretük és fogyasztásuk hatalmas volt, és a nagy disszipáció miatt rendszeresen meghibásodtak. Fixpontos műveletvégzésre voltak képesek, és utasításkészletükben ismert volt már az ugró utasítások és az akkumulátor (regiszter) fogalma is.

2. generáció – Tranzisztorok (~1956–1964)

A tranzisztor megalkotása (1947) után közel egy évtizeddel a diszkrét félvezető elemek megjelentek a számítógépekben is. Az összeköttetéseket nyomtatott áramkörök biztosították, így sokkal kisebb méret és fogyasztás volt elérhető. A relék helyett megjelentek a ferritmagos memóriák. A második generációs gépek már alkalmasak voltak lebegőpontos műveletvégzésre, a gépi kód helyett már assembly nyelvet használtak a programok írására, és ekkortájt kezdődött el a magas szintű nyelvek (Fortran, Algol, Cobol) és fordítóprogramok alkalmazása is. A be- és kimeneteket még itt is lyukkártyák szolgáltatták. Az első tranzisztor alapú számítógép az MIT-n fejlesztett TX-0 volt 1956-ban. További reprezentatív második generációs rendszerek az IBM 7090, az Univac LARC és a CDC 1604.

Copyright © 2021 / BME AUT

3. generáció – Integrált áramkörök (~1964–1971)

A harmadik generáció már integrált áramköröket használt logikai és memória funkciók megvalósítására, kis és közepes integráltsági fokú gyártástechnológiával (SSI – small-scale integration, MSI – medium-scale integration, több tíz vagy száz tranzisztor egy lapkán). Az alkatrészeket többrétegű nyomtatott áramkörökön helyezték el. Ekkor vált elterjedtté a mikroprogramozás a vezérlőművek tervezésében, és ekkor kezdték bevezetni a csővezetékes végrehajtás (pipelining) és a gyorsítótárak alkalmazását. Ennél a generációnál jelent meg a billentyűzet és monitor, mint felhasználói felület, valamint az operációs rendszerek és a virtuális memória használata. A generáció híres képviselői az 1964-ben megjelent IBM System/360-370 sorozatú, valamint a CDC 6600/7600 sorozatú és a DEC PDP-8 sorozatú mainframe számítógépek.

4. generáció – Mikroprocesszorok (~1971–2010)

Az Intel 1971-ben mutatta be a 4004-es mikroprocesszort, amely egy integrált áramköri lapkán tartalmazta az összes lényegi processzorfunkciót. A félvezető alapú memóriák az 1970-es évek végére kiszorították a ferritmagos memóriákat, ami az integrált áramköri gyártástechnológia fejlődésével együtt elindította a számítógépek azóta is megfigyelhető hihetetlen mértékű miniatürizálását és teljesítménynövekedését. Egyes szerzők a negyedik generációba a nagyfokú integrációval gyártott mikroprocesszorokat sorolja az 1980-as évekig (LSI és VLSI, azaz large-scale és very-large-scale integration, több tíz, ill. százezer tranzisztor egy lapkán). Mások ide sorolják az ULSI (ultra-large-scale integration, 1 milliónál több tranzisztor) technológiával gyártott processzorokat is kb. 2010-ig. A nagyfokú miniatürizálásnak köszönhetően ebben a korszakban jelentek meg az otthoni felhasználásra szánt asztali és hordozható számítógépek és a grafikus felhasználói felület.

5. generáció – Mesterséges intelligencia (~2010-től napjainkig)

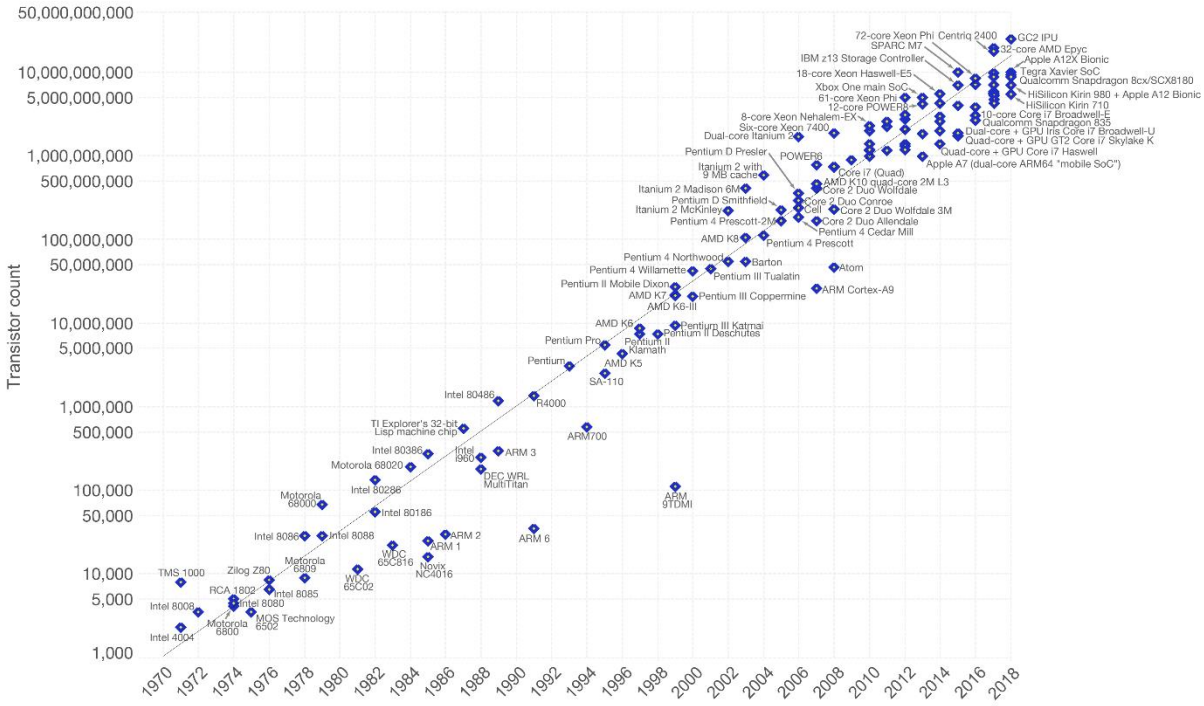
A gyártástechnológia-alapú besorolás az 1990-es évek elejétől datálja a számítógépek ötödik generációját, amelyek mikroprocesszorai már milliónál is több tranzisztort tartalmaznak. Egy másik megközelítés a mesterséges intelligencia szélesebb körben való használatát lehetővé tévő architektúrákat sorolja az ötödik generációba. A mély tanuló algoritmusok (deep learning) és más számításigényes MI alkalmazások a 2010-es évektől váltak népszerűvé, elég, ha csak a televíziós vetélkedőben is sikerrel szereplő IBM Watson-ra, az emberi beszéddel vezérelhető virtuális asszisztensekre (Apple Siri, Microsoft Cortana, Amazon Alexa stb.) és az önvezető járművek fejlesztésére gondolunk.

1.2 A számítógép-architektúra fogalma

Az integrált áramköri gyártástechnológia fejlődésével növekszik az egy lapkán megvalósítható áramkörök sűrűsége, sebessége és mérete is. A fejlődésnek egy szemléletes mérőszáma a processzorokban található tranzisztorok mennyisége. Míg az 1971-ben bemutatott első mikroprocesszor (Intel 4004) csak kb. 2300 tranzisztort tartalmazott, a 2019-ben megjelent AMD Epyc Rome már közel 40 000 000 000 tranzisztorból áll. A fejlődés mértéke nem lineáris,

Copyright © 2021 / BME AUT

hanem exponenciális. A Moore-törvény fogalmazza meg ezt a megfigyelést, amely szerint a processzorok tranzisztorainak száma kb. kétévente megduplázódik (ld. az 1.1. ábrán).



1.1. ábra – Moore-törvény: Processzorok tranzisztorszámának növekedése[1]

Ez a rendkívül gyors fejlődés az egyik oka annak, hogy a számítógép-architektúrákkal foglalkozó szakterület nem tekinthető egzakt tudománynak. Amit korábban megállapított, kimondott, az ma már nem biztos, hogy helytálló, érvényes. Sok megállapítása közelítő érvénnyel bír, ökölszabály jellegű, és nagyon gyakran a tapasztalaton alapul.

Például ilyen közelítő, ökölszabály jellegű megállapítást fogalmaz meg a számítástechnikában a közelségi elv (principle of locality), amely formális módszerekkel nem bizonyítható, de a tapasztalatok alapján igaznak bizonyult, és például a gyorsítótárak hatékony alkalmazásának alapjául szolgál:

- *Időbeli lokalitás:* Ha a program futás közben hozzáfér egy adott utasításhoz vagy adathoz, akkor a közeli időpontokban is nagy valószínűséggel hozzá fog férni azokhoz (pl. programhurkok, ciklikus feldolgozás).
- *Térbeli lokalitás:* Ha a program futás közben hozzáfér egy adott utasításhoz vagy adathoz, akkor annak környezetében lévő utasításokhoz és adatokhoz is hozzá fog férni a közeli időpontokban.

[1] Forrás: Our World in Data – Technological progress, https://ourworldindata.org/technological-progress

Copyright © 2021 / BME AUT

A számítógépek nagyon összetett rendszerek. A tervezés, az analízis vagy a megértés esetén egyaránt a hierarchikus megközelítési módot kell használnunk. Egy hierarchikus rendszer alrendszerek egymással kapcsolatban lévő halmaza, de maguk az alrendszerek is további al-alrendszerek hierarchikus struktúrái mindaddig, amíg el nem jutunk az elemi alrendszerekig.

Az összetett rendszerek hierarchikus jellege alapvető akár tervezésük, akár leírásuk vonatkozásában. A tervezőnek egy adott időben a rendszernek csak egy meghatározott részével, a hierarchia egy adott szintjén kell foglalkoznia a feladatokkal. Az egyes szinteken a rendszer komponensek és kapcsolatuk halmazából áll. Egy adott szinten a viselkedés csak az alatta lévő szint egyszerűsített, absztrakt módú jellemzésétől, leírásától függ. A tervező minden szinten struktúrával és funkcióval foglalkozik:

- Struktúra: Az egyes komponensek egymással való kapcsolata
- Funkció: A struktúrát alkotó egyes komponensek viselkedése.

E tantárgy keretében is a hierarchikus megközelítést fogjuk alkalmazni a különböző számítógép-architektúrák vizsgálatakor. Egy-egy esetben mindig csak egy adott absztrakciós szinten lévő komponensek viselkedésének és kapcsolatainak vizsgálatára szorítkozunk anélkül, hogy az alacsonyabb szintű megvalósítást részletesen vizsgálnánk.

1.2.1 Architektúra, implementáció és realizáció

A számítógépek tervezésével kapcsolatban három elkülönülő tartományt kell megkülönböztetnünk: az architektúrát, az implementációt – más nevén mikroarchitektúrát – és a realizációt.

Egy rendszer architektúrája alatt annak funkcionális megjelenését értjük felhasználói felé, más szavakkal a fenomenológiáját (fenomenológia = a közvetlenül észlelhető dolgok). Az architektúra fogalma nem újkeletű, hanem az emberek által létrehozott mechanizmusokkal egyidős. Amikor egy kisgyereknek az órát tanítjuk, az óra architektúráját tanítjuk. A kis- és nagymutató helyzetének megfigyelésére és azoknak az órákhoz és percekhez való viszonyára mutatunk rá. Amint a kisgyerek meg tudja különböztetni az architektúrát a vizuális megjelenéstől, akár a karóráról, akár a toronyóráról le tudja olvasni az időt.

Egy rendszer belső struktúrája nem tartozik az architektúrához: az idő meghatározásához nem szükséges tudnunk, hogy mitől és hogyan jár az óra belső szerkezete. Logikai nézőpontból a belső struktúrát implementációnak – informatikai rendszerek esetén más szóval mikroarchitektúrának –, a tényleges fizikai megvalósítást pedig realizációnak nevezzük.

Példaként tekintsük a számítógép ALU alrendszerét. Az, hogy a számítógép ALU alrendszere képes-e a szorzás utasítás végrehajtására, s ha igen, akkor azt milyen szóhosszúsággal stb. hajtja végre, ez az architektúra része. De az már az implementációhoz tartozik, hogy az ALU a szorzást egy külön szorzó egységgel vagy az összeadó egység ismételt alkalmazásával hajtja-e végre. A realizáció pedig a tényleges fizikai megvalósítást jelenti: a logikai áramköröket megvalósító integrált áramköröket, azok egymáshoz kapcsolódását stb.

Copyright © 2021 / BME AUT

Az a korábban már említett megállapítás, miszerint a számítástechnika nem egzakt tudomány, magára a számítógép-architektúra definíciójára is vonatkozik.

Az első definíciót először az IBM System/360 tervezői (Amdahl, Blaauw és Brooks) adták meg 1964-ben: egy számítógép architektúrája alatt annak funkcionális megjelenése értendő a gépi kódú programozók számára. Más szavakkal az architektúra specifikációja a számítógép összes olyan funkcióját tartalmazza, melyek a programok által „megfigyelhetők”.

Napjainkban a legszélesebb körben elfogadott definíció: a számítógép-architektúrát az **utasításkészlet-architektúra** és a **mikroarchitektúra** együtt képezi. Ebben a megközelítésben az előbb látott IBM-féle klasszikus definíció az utasításkészlet-architektúrának felel meg.

1.2.2 Utasításkészlet-architektúra

Az utasításkészlet-architektúra (Instruction Set Architecture, ISA) a számítógép programozásához kapcsolódva tartalmazza az utasításokat, az adattípusokat, regisztereket, címzési módokat, a memória architektúrát, a megszakítások és kivételek, valamint a külső be- és kivitel kezelését. A szakirodalomban ezt *programozói modell*nek is nevezik.

Az utasításkészlet-architektúra a hardver és a szoftver közötti interfésznek is tekinthető, elválasztja a statikus feladatokat (fordítási idő) a dinamikus feladatoktól (futási idő).

Az ISA mintegy „megállapodásként” szerepel a szoftver és a hardver között, s lehetővé teszi a programok és a gépek egymástól független fejlesztését. Másrészt az ISA mintegy specifikációul szolgál a számítógép implementációjának, azaz mikroarchitektúrájának megtervezéséhez.

Egy bevált, pontosabban széles körben elterjedt utasításkészlet-architektúrát a technológiai lehetőségek fejlődésével az architektúra fejlesztője időről-időre új utasításokkal bővíti. A technológiai fejlődés mellett a *kompatibilitás* igénye is alátámasztja az ISA bővítésére való törekvést. Az új funkciókat, a programozói lehetőségek kiterjesztését ilyenkor nem egy teljesen új utasításkészlet megalkotásával érik el, hanem azzal, hogy a korábbi utasítások megtartásával újabb utasításokat (és hozzájuk kapcsolódva adattípusokat, regisztereket stb.) vezetnek be. Így az új architektúrájú számítógép a korábbi családtagokra írt gépi kódú programok futtatására módosítás nélkül képes. Jellemző példa erre az Intel által megalkotott x86 utasításkészlet-architektúra család, amely először 1978-ban jelent meg a 16 bites 8086-os processzorban. Néhány évvel később, 1985-ben a 80386-os processzortól kezdve 32 bitesre bővült (IA-32, azaz Intel Architecture 32 bit), 2003 óta pedig elérhető a 64 bites változata is (x86-64 vagy x64 néven), amit mind a mai napig fejlesztenek. Az adathossz bővülése mellett újabb és újabb funkciók is megjelentek az évtizedek során, például az Intel Pentium processzornál vezették be a multimédia-alkalmazásokban hasznos SIMD (single instruction multiple data) utasításosztályt MMX néven, amelyet később jelentősen továbbfejlesztettek SSE (Intel Pentium III), SSE2, SSE3 (Intel Pentium 4) és SSE4 (Intel Core 2) néven. A mai processzorok által megvalósított x64 utasításkészlet jelentősen különbözik a több, mint 40 évvel ezelőtti változattól, de mind a mai napig visszafelé kompatibilis vele.

Copyright © 2021 / BME AUT

1.2.2.1 CISC vs. RISC

Az utasításkészlet-architektúrák említésekor nem mehetünk el szó nélkül a két nagy tervezési irányzat képviselői, a CISC és a RISC elvű számítógépek mellett. Ez a felosztás az 1980-as években jelent meg, és valójában nem csak az utasításkészlet-architektúrát, hanem a mikroarchitektúra számos aspektusát is meghatározta, hogy egy adott gép CISC vagy RISC elvek mentén került megtervezésre.

Mindkét számítógéptervezési elv a teljesítőképesség növelését tűzte ki célul, csak különböző korokban, különböző technológiai adottságok mellett. A CISC (Complex Instruction Set Computer) számítógépek jelentették a klasszikus tervezési elvek megtestesítőit az 1960-as, 70-es, években. Eleinte a programokat gépi kódban vagy assembly nyelven írták, így nagy volt a fogalmi távolság a megoldandó problémák és a rendelkezésre álló gépi utasítások között. Emiatt, illetve a magasszintű programozási nyelvek megjelenése után is a programozás megkönnyítésére igyekeztek egyre inkább olyan gépi utasításokat létrehozni, amelyek minél közelebb állnak a magas szintű nyelvek által biztosított funkciókhoz. Úgy tűnt, hogy a magas szintű nyelveken írt programok hatékonyabban fordíthatók ilyen utasításkészletre, és minél kevesebb utasítás szükséges egy magas szinten leírt probléma megoldásához, annál nagyobb lesz a számítógép teljesítőképessége. Fontos tudni, hogy ekkortájt még igen lassú volt a memóriák sebessége, így, ha a programfutás során kevesebb utasítás felhozatalára volt szükség, az valóban növelni tudta a végrehajtási sebességet. Ennek hatására nagyon komplex és igen sokféle utasítást tartalmazó architektúrák alakultak ki.

Ahogy azonban a memóriák egyre gyorsabbak lettek (megjelentek a félvezető alapú memóriák a korábbi ferritgyűrűs technológia után), és fejlődtek a fordítóprogramok is, felmerült, hogy a kevesebb és egyszerűbb utasítások alkalmazása elérheti, sőt meghaladhatja a komplex utasításokkal működő számítógépek teljesítményét. Már nem volt akkora probléma az, ha csak jóval több utasítással lehetett megvalósítani egy adott magas szintű műveletet, hiszen a fejlett fordítóprogramoknak hála, már egyre kevésbé a programozót terhelte a fogalmi távolság áthidalása, és a több utasítás felhozatala is egyre kisebb problémát jelentett a végrehajtás sebessége szempontjából. Ezen kívül a komplex utasításkészletek vizsgálata rámutatott, hogy a bonyolult, speciális utasításokat a fordítóprogramok jellemzően csak kis mértékben tudják kihasználni, és ezek elhagyása nem jelent nagy veszteséget a teljesítőképességben. Ugyanakkor kevesebb és egyszerűbb utasítás használatával a mikroarchitektúra és a realizáció egyszerűsíthető, így például nagyobb órajel alkalmazható. Tehát míg a CISC elv a programban lévő utasítások számának minimalizálására törekedett, a RISC elv az egyes utasítások végrehajtási idejének csökkentésére. A számítógépek teljesítőképességét ennek a két mennyiségnek a szorzata határozza meg, tehát a cél ugyanaz volt mindkét tábor képviselői számára. Az 1980-as években és a 90-es évek elején komoly vita zajlott szakmai körökben arról, hogy melyik megközelítés lehet az eredményesebb.

Bár a két tábor azóta is megmaradt, az utasításkészletek számos dologban konvergáltak egymáshoz, így a határok bizonyos szempontból elmosódtak. A RISC architektúrák is átvettek hasznosnak bizonyult komplexebb utasításokat, és a CISC architektúrákból is kikoptak azok, melyek az előfordulási gyakoriságukhoz képest fölöslegesen bonyolulttá tették az implementációt.

Copyright © 2021 / BME AUT

A CISC és RISC architektúrák megkülönböztetésében világos segítséget nyújt, ha megnézzük a memória-hozzáférések lehetőségeit. A RISC számítógépek *load-store architektúrát* valósítanak meg, azaz csak a töltő és tároló utasítások érik el a memóriát. Minden más utasítás operandusai csak regiszterek lehetnek. CISC architektúrákban nincs ilyen megkötés, mindenféle művelet változatos címzési módokat használhat, akár memóriában lévő operandusokkal.

Egy utasításkészlet-architektúra jellemzője az is, hogy az utasítások hossza fix-e vagy változó. Fix hosszúságú utasításkészlet esetén minden utasítás azonos számú biten van kódolva, következésképpen a dekódoló logika egyszerűbbé, így gyorsabbá válik, és a következő utasítás címe is egyszerűen kiszámítható már a dekódolás eredménye előtt. Ez, mint a későbbiekben látni fogjuk, komoly lehetőségeket biztosít a mikroarchitektúra hatékony kialakítására. Változó utasításhossz esetén az utasításokat eltérő számú biten (illetve bájton) kódolják attól függően, hogy hány operandusuk van, milyen címzési módot használnak stb. Ez a kialakítás nagyban növeli a programtár kihasználtságát, ha az utasítások komplexitása nagy ingadozást mutat az utasításkészleten belül. Ugyanakkor bonyolultabb dekódoló logikára van szükség, és a soron következő utasítás címe sem generálható addig, míg a dekódolás meg nem történik. Nem nehéz kitalálni, hogy a fix utasításhossz a RISC, a változó hosszúságú utasítások alkalmazása pedig a CISC elvű számítógép-architektúrákra jellemző leginkább.

A fenti szempontok alapján jól látható, hogy az utasításkészlet-architektúra szintjén meghozott döntések nagy mértékben kihatnak a mikroarchitektúra tervezésére. Ugyanez fordítva is igaz: a realizáció számára rendelkezésre álló technológiai lehetőségek meghatározzák a mikroarchitektúra lehetőségeit, és ez komoly kihatással van az utasításkészlet-architektúra kialakításánál meghozott döntésekre.

1.2.3 Mikroarchitektúra

A mikroarchitektúra a szilíciumlapkán megvalósítandó vagy megvalósított hardver működésének teljes mértékű logikai leírása. Tartalmazza a processzor egyes blokkjainak, közöttük a magoknak, végrehajtó egységeknek (mint pl. lebegőpontos- és egész típusú egységeknek), gyorsítótáraknak, periféria-támogatásnak a leírását.

A mikroarchitektúra meghatározza, hogyan valósul meg az egyes utasítások végrehajtása (pl. mikroprogramozott vagy huzalozott a végrehajtómű, hány gépi ciklust igényel egy utasítás végrehajtása), hány utasítás hajtható végre egyszerre, milyen csővezetékeket, elágazás prediktorokat vagy más hasonló megoldásokat alkalmazunk.

Gyakran, főleg tudományos körökben a mikroarchitektúrát *számítógép szervezésnek* is nevezik.

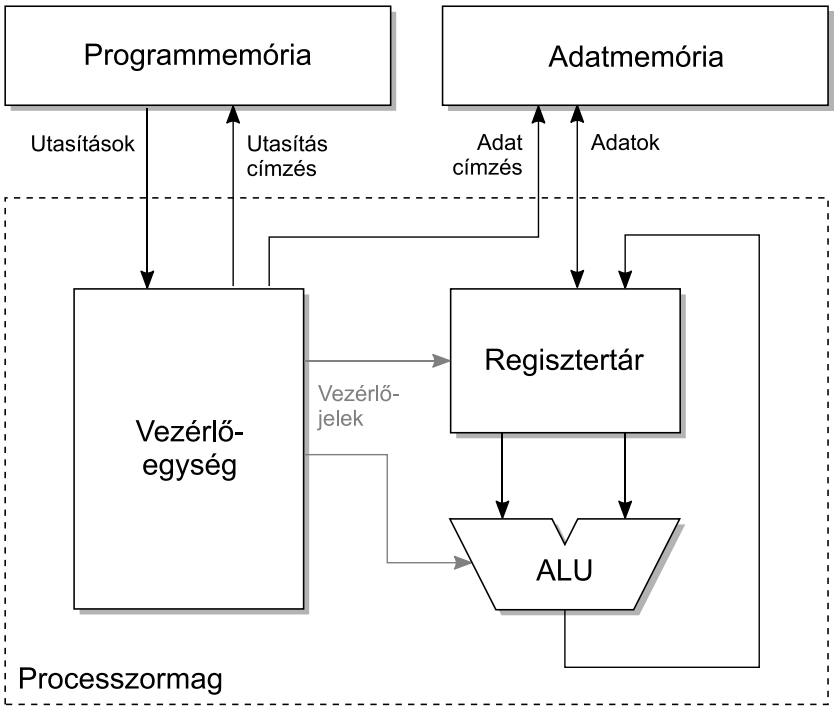
Az utasításkészlet-architektúra és a mikroarchitektúra határai elmosódhatnak: a számítógép-architektúra egy-egy aspektusáról nehéz lehet eldönteni, hogy az még a specifikáció (ISA) vagy az implementáció (mikroarchitektúra) része. Hasznos definíció lehet a megkülönböztetésre, hogy

Copyright © 2021 / BME AUT

- minden, amiről az assembly programozónak tudnia kell ahhoz, hogy helyesen működő programokat írjon, az ISA része, míg
- minden, ami a processzor teljesítményével van összefüggésben, és nem befolyásolja a programvégrehajtás *helyességét*, a mikroarchitektúra része.

1.2.3.1 Utasításvégrehajtás a processzorban

Egy processzor általános (egyszerűsített) felépítését láthatjuk az 1.2. ábrán. Három fő részre osztható működési szempontból: a *vezérlőegységre* (control unit), a *műveletvégző egységre* (az ábrán ALU, azaz arithmetic/logic unit) és a *regisztertárra* (register file).



1.2. ábra – Processzor általános felépítése

A vezérlőegység feladata az utasítások és adatok felhozatala a memóriából, az utasítások dekódolása, és ez alapján a regisztertár és a műveletvégző egység összehangolt vezérlése. A műveletvégző egység hajtja végre a dekódolt utasításokat a regisztertárban előkészített operandusokon, és szintén a regisztertárnak adja át az eredményt (load-store architektúrát feltételezve). Műveletvégző egységből nem csak egy lehet, a legkézenfekvőbb esetben pl. külön egység hajtja végre az egész és a lebegőpontos műveleteket.

A vezérlőegység lehet *huzalozott* vagy *mikroprogramozott*. Huzalozott megvalósítás esetén az utasítás dekódolásának eredménye közvetlen vezérlőbiteket jelent a végrehajtóegység és a regisztertár számára. Mikroprogramozott esetben az utasításkészletben előforduló műveletek nem közvetlenül hajtódnak végre, hanem elemi műveleteket reprezentáló mikroutasítások vagy más néven mikroműveletek („µops”) sorozatának felelnek meg. A dekódolás ez esetben azt

Copyright © 2021 / BME AUT

jelenti, hogy meghatározzuk az adott gépi utasításhoz tartozó mikroműveleteket, amelyek már tényleges hardver vezérlőjeleket jelentenek. Gyakorlatilag az ISA utasításait a vezérlőmű kis (mikro)programrészletekké konvertálja, amelynek (mikro)utasításai szekvenciálisan vezérlik a végrehajtó hardveregységeket. A mikroprogram (másnéven mikrokód) tehát egy hardverben megvalósított plusz „szoftveres” absztrakciós réteget jelent a gépi kódú utasítások és a hardveres végrehajtóegységek között. Ez a megoldás nagyban leegyszerűsíti a komplex vagy változó hosszúságú gépi utasítások dekódolását és végrehajtását, illetve a végrehajtómű tervezését. Bonyolultabb utasítások megvalósítása huzalozott logikával sokkal összetettebb hálózatot jelentene, ugyanakkor a szekvenciális végrehajtás ára, hogy az utasítások végrehajtási ideje az őket megvalósító mikroműveletek számától függ. A CISC processzorok jellemzően mikroprogramozott, a RISC processzorok pedig huzalozott vezérlőegységgel rendelkeznek.

A mikrokód hagyományosan egy ROM memóriában tárolódik a processzoron belül, de a modern processzorokban megjelentek az SRAM és Flash alapú írható mikroprogramtárak is. Ez utóbbinak előnye, hogy a mikrokód utólag módosíthatóvá válik, ami lehetővé teszi később felfedezett hibák javítását. Ez történt nemrég az Intel esetében is, amikor a híres „Spectre” és „Meltdown” biztonsági rések 2018-as napvilágra kerülése után mikrokód frissítést adtak ki évekkel korábban piacra került processzoraikhoz is. Ellenpéldaként említhetjük a korai Intel Pentium processzorok lebegőpontos osztási hibáját, ami miatt 1994-ben az Intel visszahívta az érintett processzorokat, és új hardver revíziót adott ki, mert akkoriban nem használtak még módosítható mikroprogramtárat.

1.3 A jó architektúra jellemzői

Amikor a jó architektúra jellemzőiről beszélünk, ismét nem egzakt dologról van szó. Sokkal inkább a korábbi tapasztalatokból leszűrt következtetések ezek. A legfontosabbak:

- Következetesség (consistency)
- Ortogonalitás (orthogonality)
- Megfelelőség (propriety)
- Takarékosság (parsimony)
- Átlátszóság (transparency)
- Általánosság (generality)
- Nyitottság (open-endedness)
- Teljesség (completeness)

Következetesség

A tiszta architektúra fontos jellemzője a konzisztencia. Ez azt jelenti, hogy a rendszer egy részének ismeretében következtethetünk a többi rész architektúrális jellemzőire. Például, ha nem ismerjük, hogy egy logaritmust számító utasítást hogyan hajt végre a számítógép, konzisztens architektúra esetén feltehetjük, hogy a többi aritmetikai utasítás adatábrázolási

Copyright © 2021 / BME AUT

formáját, kerekítési rendszerét, pontosságát, kivételkezelését, jelzőbit-beállító funkcióit stb. használja. Nulla logaritmusa számításának hasonló kivételt kell kiváltania, mint például a nullával való osztásnak.

Egy példa az inkonzisztens tervezésre: A Motorola MC6820 processzorának különböző utasítások esetén különböző hosszúságúak a közvetlen (immediate) operandusai:

- ADDQ (add quick): közvetlen operandusa 3 bites
- MOVEQ (move quick): közvetlen operandusa 8 bites

Ortogonalitás

A jó architektúrában a koncepcionálisan független funkciókat elkülönítjük specifikációjukban. Ezt az elvet ortogonalitásnak nevezik. Analógia a geometriából: egy koordináta menti változás nem változtatja meg a többi koordináta értékét. A korábbi óra példánkat ismét elővéve, világító számlappal és ébresztési funkcióval is elláthatjuk azt. Ezek egymástól független funkciók, s az ortogonalitást megsértené, ha az ébresztési funkció csak világító számlap esetén érvényesülne.

Általánosan: az ortogonalitás megköveteli, hogy egy jellemzőosztályon belüli változás ne hasson más jellemzőosztály értékeire. Esetünkben: a koncepcionálisan független funkciók specifikálása független legyen egymástól.

Processzorok esetében az ortogonális utasításkészlet-architektúra a címzési módok és az utasítások típusainak függetlenségét jelenti. Nézzünk erre két ellenpéldát, ahol a művelet (egyik koordináta) és operandus (másik koordináta) specifikációja nem ortogonális. Ha megváltoztatjuk az utasítás típusát jelentő „koordinátát mentén”, megváltozik a címzési lehetőségeket jelentő „koordináta” értéke.

Az egyik példánk ismét a Motorola MC6820 processzora. Utasításkészletében a MOVE utasítás mindkét operandusa megadható effektív címmel, ezzel szemben az ADD utasításnak csak egyik operandusa adható meg effektív címmel.

A másik példa legyen az Intel 8051 mikrokontroller utasításkészlete. Ezen belül az artimetikai és a logikai utasítások számára elérhető címzési módok különböznek. Aritmetikai utasításra példaként lássuk az összeadást, logikaira pedig a bitenkénti ÉS műveletet (ld. 1.1. táblázat).

ADD A, Rn	A = A + regiszter
ADD A, direct	A = A + direkt bájt (memóriából)
ADD A, @Ri	A = A + indirekt bájt (cím az Ri-ben)
ADD A, #data	A = A + közvetlen konstans

ANL A, Rn	A = A & regiszter
ANL A, direct	A = A & direkt bájt (memóriából)
ANL A, @Ri	A = A & indirekt bájt (cím az Ri-ben)
ANL A, #data	A = A & közvetlen konstans
ANL direct, A	direkt bájt = direkt bájt & A
ANL direct, #data	direkt bájt = direkt bájt & közvetlen konstans

1.1. táblázat – Intel 8051 aritmetikai és logikai utasításainak címzési módjai

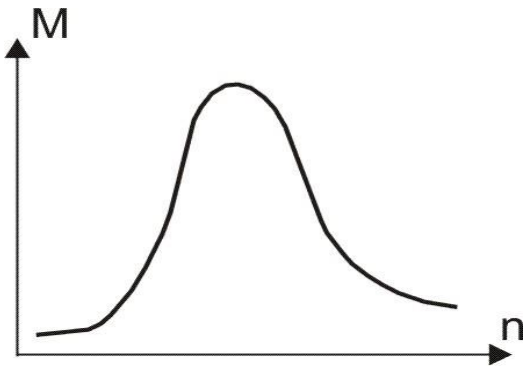
Copyright © 2021 / BME AUT

Az artimetikai műveleteknél az egyik forrásoperandus és a céloperandus mindig az akkumulátor. A másik forrásoperandus lehet regiszter, direkt című memória tartalom, indirekt memóriatartalom (cím egy regiszterben megadva), vagy közvetlen (immediate) konstans. Ugyanezeket a címzési módokat megtaláljuk a logikai műveleteknél is, de ott a céloperandus lehet direkt memóriatartalom is. Viszont ekkor a másik forrásoperandus csak az akkumulátor vagy immediate konstans lehet. Egyrészt eltér az aritmetikai és logikai műveletek számára elérhető címzési módok száma, másrészt a logikai műveleteknél a céloperandustól függ, hogy milyen címzés használható a forrásoperandushoz.

Megfelelőség

A megfelelőség követelményének akkor teszünk eleget, ha a specifikált funkciók logikailag hozzátartoznak a rendszer alapvető, lényeges követelményeihez. Ellentéte a feleslegesség (extraneousness), ami nem kell a lényegi cél eléréséhez.

Például az autók sebességváltója funkcionálisan felesleges, hiszen logikailag elegendő lenne egyetlen kezelőszerv, pl. a gázpedál a sebességváltoztatáshoz. Mégis miért van? Ennek oka a belsőégésű motorok nyomaték-fordulatszám jelleggörbéje. Amint az 1.3. ábrán láthatjuk, a belső égésű motorok csak bizonyos fordulatszám-tartományban képesek nagy nyomatékot leadni. Automata sebességváltós autókban ez ki van küszöbölve, azaz a sebességváltó funkció, mint a vezető szempontjából nem lényeges elem, el van rejtve.



1.3. ábra – A belsőégésű motorok nyomaték-fordulatszám jelleggörbéje

Példák a processzorok területéről:

- Az MC68000 mikroprocesszor CLR (törlés) utasítása először olvas, csak azután töröl. Így egyszerűbb volt megcsinálni. De ez nemcsak felesleges, de csak írható regisztereknél hibát is okozhat. Később ki is javították.
- A 2-es komplemens kódban csak egyetlen NULLA érték van, az 1-es komplemens kódban viszont kettő, a +NULLA és a –NULLA is létezik. A két nulla létezése a gyakorlatban olyan felesleges összehasonlításokat követel, ami a matematikában nem értelmezett.

Copyright © 2021 / BME AUT

Takarékosság

A jó architektúra gazdaságos is. A rendszer szempontjából inkorrekt funkciókat nem tartalmaz. Például az MC68020 mikroprocesszorban a mikroprogram lehetővé tenné, hogy egy 32 bites lebegőpontos szám 1-es komplemensét is képezzük. De minek? Mivel semmire nem használnánk, megvalósítása pazarlás lenne, megszegnénk a takarékosságra való törekvést.

Átlátszóság

Átlátszó a rendszer, ha az implementáció vagy realizáció szintjén meghozott döntés nem hat vissza az utasítás architektúra szintjére (information hiding).

Súlyosan sértette az átlátszóságot a RISC processzorok kezdeti csővezeték implementációja. A csővezeték szervezés ugyanis megkövetelte az ún. késleltetett elágazások alkalmazását, azaz akár létrejött az ugrás, akár nem, a processzor végrehajtotta az őt követő utasítást. Ezt assembly szinten a programozónak vagy a fordítónak kezelnie kell NOP-ok beszúrásával vagy az utasítások felcserélésével.

1. példa: NOP beszúrása

```
ADD A,B
JMP cím,Z
SUB A,C    // ezt az utasítást mindenképpen végrehajtotta a processzor
           // ezért a fordítóprogram beszúrt egy NOP-ot
```

Beszúrás után:

```
ADD A,B
JMP cím,Z
NOP
SUB A,C    // eddigre már kiértékelődik az elágazás
```

2. példa: Ugrás előtti utasítás áthelyezése

```
MOV D,( )                    ADD A,B
ADD A,B                      JPM cím,Z
JMP cím,Z                    MOV D,( )
SUB A,C                      SUB A,C
```

Általánosság

Ez a tulajdonság egy funkció sokféle használatának lehetőségére utal. A tervező egy funkciót ne korlátozzon a saját használati tapasztalatai alapján. Számítania kell a sok kreatív felhasználóra.

Például kezdetben a számítógépeket kizárólag számítási feladatokra használták és szánták. S a be-kiviteli egységeket csak adatok átvitelére képezték ki az akkor elérhető perifériákkal (lyukkártya olvasó, nyomtató). De mikor a számítógépeket folyamatirányításra is elkezdték

Copyright © 2021 / BME AUT

használni, ezek a be-kiviteli egységek már alkalmatlanok voltak, s rajtuk lényeges módosításokat kellett végrehajtani. Tehát a be-kiviteli egységek eredetileg nem eléggé általános funkcionalitással lettek megtervezve.

Másik példa lehet az érintőkijelző alkalmazása az okostelefonokon. Korábban a telefonokon funkcionálisan elkülönült egymástól a beviteli eszköz (gombok) és a kijelző. Aztán az érintőkijelzők fejlődésével rájöttek, hogy ha a billentyűzetet a kijelzőn jelenítik meg, akkor sokkal általánosabban lehet kihasználni a rendelkezésre álló felhasználói felületet.

Nyitottság

Egy architektúra nyitott, ha például a jövőbeli fejlesztésekre fenntartott műveleti kód- és címterülettel rendelkezik. Más példa: HTML fix elemkészletű, XML bővíthető elemkészletű dokumentációs nyelv. Az előbbi nem, az utóbbi viszont nyitott a további bővítésekre.

Teljesség

Ha egy adott funkcióosztály összes funkciója jelen van az architektúrában, akkor az architektúra ezen funkcióosztály szempontjából teljes. Ha például az utasításkészlet tartalmaz szorzó utasítást, akkor a teljességhez tartalmaznia kell osztó utasítást is.

Például az MC6809 volt az első 8 bites mikroprocesszor, amelynek utasításkészlete már tartalmazott szorzást, persze még csak egész típusú operandusok esetén. De az osztást nem tartalmazta, ezért utasításkészlet architektúrája nem volt teljes. Nyilván a teljességre törekedni kell, de biztosítását a rendelkezésre álló technikai színvonal, technológia korlátozhatja. A Motorola fejlesztői nyilván beletették volna az osztó utasítást is az akkori csúcsnak számító processzorukba, de a kezdetleges integráltsági fok miatt nem tehették.

Példa a teljességre a DEC System-10 számítógépének utasításkészlete, amely a logikai műveletek tekintetében mind a 16 lehetséges logikai műveletet tartalmazta. Ugyanakkor tapasztalat szerint a legtöbbször használt műveletek az AND, OR és NOT, és ezekkel a többi művelet is megvalósítható. Elhagyva a többi 13-at, utasításkódokat takaríthatunk meg. Tehát nem mindig célszerű a teljességet biztosítani.

1.4 Számítógépcsaládok

Egy jól bevált és elterjedt utasításkészlet-architektúránál fontos, hogy az arra írt szoftverek nagy mennyisége miatt sokáig elérhető és használható legyen. Ezért a gyártók az utasításkészlet-architektúrákat nem szokták gyökeresen megváltoztatni, inkább csak bővíteni úgy, hogy a korábbi utasításokat továbbra is támogassák. Így a korábbi változatokra írt (fordított) kódok az újakon is futtathatók maradnak. Az utasításkészlet-architektúrák köré így a gyártók tipikusan számítógépcsaládokat szerveznek. Egy családba az egymás után következő újabb és újabb utasításkészlet-változatok szerint működő, de visszafelé kódkompatibilis processzorok tartoznak. Továbbá ugyanazon utasításkészlet-változat esetén is teljesen bevett szokás, hogy

Copyright © 2021 / BME AUT

mikroarchitektúrájukban vagy realizációjukban eltérő, emiatt különböző teljesítményű és árú gépeket forgalmaz a gyártó.

Klasszikus példák:

- Az IBM System/360 mainframe számítógépcsaládja az 1960-as években.
- A Digital Equipment Corporation (DEC) PDP-8 és PDP-11 családjai az 1960-70-es, valamint VAX-11 és VAX-8000 családjai az 1970-80-as években.

Mai példák:

- Az Intel által bevezetett x86 (IA-32, majd x86-64) alapú processzorok. Ide nem csak Intel gyártmányok tartoznak, hiszen elterjedsége miatt számos más processzorgyártó (Cyrix, AMD, VIA) is fejlesztett és fejleszt ma is x86 alapú mikroarchitektúrákat és processzorokat. Érdekesség, hogy az x86 ISA 64 bites, de visszafelé kompatibilis kibővítését az AMD fejlesztette ki, és később az Intel is tőlük vette át a saját processzoraiba.
- Intel IA-64 (Itanium Architecture) utasításkészlet alapú processzorok. Ezt a 64 bites utasításkészletet szánta eredetileg az Intel a korábbi IA-32 utódjának. Szakítva a hagyományokkal, ez nem kompatibilis az elődjével, hanem egy teljesen új alapokra helyezett VLIW (Very Long Instruction Word) jellegű architektúra lett, amely elsősorban nagyteljesítményű vállalati és szerveralkalmazásokban terjedt el. Az architektúra nem lett olyan népszerű, mint az x86, és az Intel 2019-ben bejelentette, hogy nem folytatják a fejlesztését.
- ARM utasításkészlet- és processzorcsalád. Ez egy 1980-as évek végén indult RISC architektúra, amely eredetileg 32 bites volt, ma már 64 bitessé bővült. A kisteljesítményű beágyazott alkalmazásoktól a nagyteljesítményű többmagos processzorokig számtalan variációja elérhető, napjainkban a mobil eszközökben leggyakrabban előforduló architektúra.

1.5 A hardver és szoftver kapcsolata

A számítógép feladata, hogy a különböző szakterületek által absztrakt módon megfogalmazható problémákat bináris logika szerint működő digitális áramkörök segítségével oldja meg. A kettő közötti koncepcionális szakadék áthidalása a programozó, a rendelkezésre álló szoftver eszközök, programnyelvek és a hardver feladata. Ezen szereplők között különböző szintű kapcsolódási felületek, interfészek találhatók. A programozás napjainkban túlnyomórészt magas szintű programnyelveken zajlik. A programnyelvek fejlődésének célja, hogy az emberi és a gépi gondolkodásmód közötti fogalmi távolságot minél inkább lecsökkentse. A magas szintű nyelven írt programokat szoftveres fordító eszközök (compiler, interpreter stb.) gépi nyelvre alakítják, ami már a processzor által közvetlenül végrehajtható.

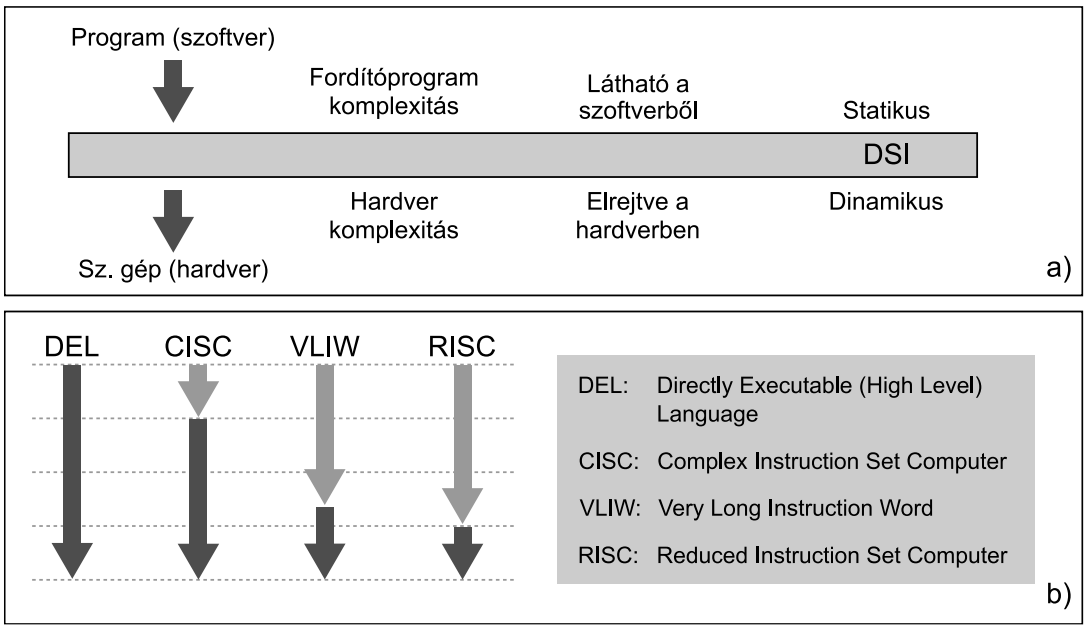
Copyright © 2021 / BME AUT

1.5.1 A dinamikus-statikus interfész

Az utasításkészlet által reprezentált hardver és szoftver közötti kapcsolódási felületet dinamikus-statikus interfésznek (DSI) is nevezzük. Ez elválasztja a fordítási időben végzett statikus tevékenységet a futási időben végzett dinamikus tevékenységtől. A különböző utasításkészlet-architektúrákban számos javaslat született arra, hogy a DSI milyen absztrakciós szintre kerüljön (ld. az 1.4. ábrán).

Ahogy korábban láttuk, a számítástechnika első évtizedeiben főleg gépi kódban és assembly nyelven folyt a programozás, ezért az utasításkészletek tervezői arra törekedtek, hogy minél magasabb szintű, absztrakt funkciókat valósítson meg egy-egy gépi utasítás. A mikroprogramozás egyre nagyobb fokú alkalmazása odáig jutott, hogy az 1970-es években megjelentek magas szintű programnyelven írt kódok közvetlen futtatására kifejlesztett „Directly Executable High Level Language” számítógép-architektúrák, vagyis a DSI a magas szintű programozási nyelvek absztrakciós szintjére került. Ezt követik a sorban a már említett CISC gépek, és a lista másik végén helyezkednek el a csupán egyszerű utasításokat megvalósító RISC architektúrák. Az ábrán megjelenik még a VLIW (Very Long Instruction Word) architektúra, amelynek gépi utasításai a RISC-hez hasonló alacsony absztrakciós szintet képviselnek. Ezek az utasítások annyiban tekinthetők magasabb szintűnek, hogy egyetlen hosszú utasításszót számos egyszerű, de párhuzamosan végrehajtható elemi utasítás alkot, vagyis a VLIW utasítások kvázi összetett műveleteket reprezentálnak. Azonban az összetett utasítások itt elsősorban a végrehajtás párhuzamosítását, és nem a fogalmi szint emelését célozzák.

A számítógépek fejlődésével a dinamikus-statikus interfész egyre alacsonyabb fogalmi szintre került, mivel a programnyelvek és fordítóprogramok fejlődésével egyre gazdaságosabb lett a hardver komplexitás csökkentése, s ezáltal gyorsabb gépek megvalósítása.



1.4. ábra – A dinamikus-statikus interfész szerepe (a) és absztrakciós szintje különböző architektúrákban (b)

Copyright © 2021 / BME AUT

1.5.2 Fordítás és értelmezés

A DSI fölötti „statikus” világban szükség van arra, hogy a magas szintű nyelveken megírt programokat a gépi utasítások szintjére leképezzük. Erre két alapvetően különböző megoldás született, a fordítás és az értelmezés (interpretálás).

Fordítás

Fordítás esetében a fordítóprogram (compiler) az egész forráskódot off-line módon elemzi, és igyekszik valamilyen szempontból (kódméret, futásidő stb.) optimális módon előállítani egy alacsonyabb szintű reprezentációt. Ez az alacsonyabb szintű nyelv lehet közvetlenül futtatható tárgykód (gépi utasítások), assembly kód vagy magasabb szintű közbülső nyelv. Assembly kódból az assembler állítja elő a tárgykódot (itt már nincs szó optimalizálásról, csak egyértelmű megfeleltetésekről és behelyettesítésekről), más közbülső nyelv esetén pedig további fordításra vagy interpretálásra van szükség a futtatás előtt. A módszer előnye a globális optimalizálás lehetősége, így hatékony programkód előállítása, hátránya viszont az offline jelleg, és az ezzel járó nagyobb fordítási idő, ami gyakran változó forráskód esetén (pl. fejlesztés alatt) kényelmetlen lehet. A széleskörben használt C/C++ nyelvű programfejlesztés jól ismert példája a fordítók használatának.

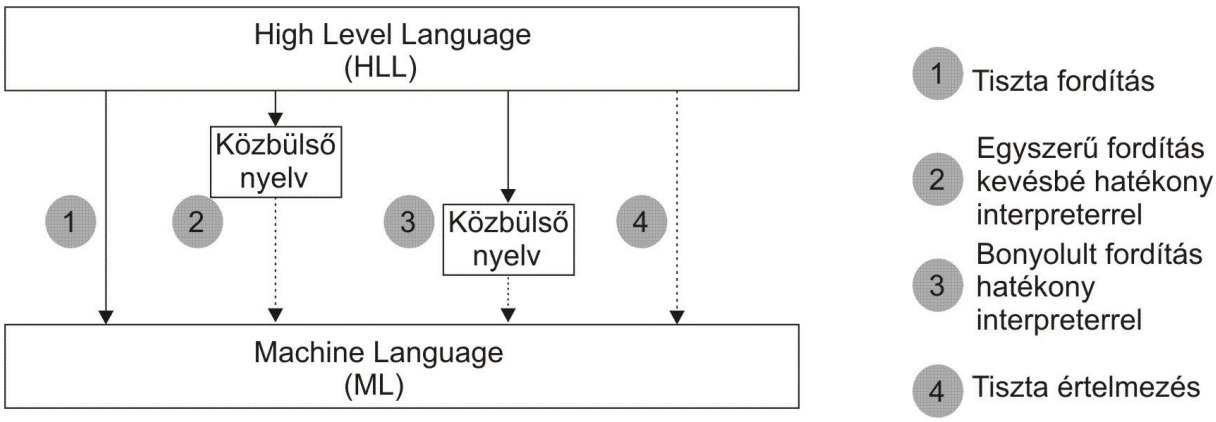
Értelmezés

Az értelmezés folyamata online módon történik az interpreter program segítségével. Ez a teljes forráskód elemzése nélkül, futásidőben alakítja át a magas szintű programozási, ill. szkriptnyelven megírt programokat gépi utasításokká. Mindig csak a következő utasítást figyeli, abból előállítja az alacsony szintű reprezentációt, és le is futtatja. Előnye ennek a megközelítésnek, hogy interaktív, valós idejű programozás valósítható meg vele, ahol például egy terminálba beírt parancsok azonnal végrehajtódnak. Hátránya a sokkal kevésbé hatékony végrehajtás, mivel egyrészt nincs meg az optimalizálás lehetősége, másrészt ismétlődő kódrészletek (pl. egy sokszor lefutó ciklustörzs parancsai) minden egyes alkalommal újra és újra értelmezésre kerülnek A parancsok egy forrásfájlba is összefoghatók (ezt nevezik szkriptelésnek), és így offline programozás is megvalósítható, de a végrehajtás a háttérben ugyanúgy sorról-sorra történik. A MATLAB vagy a különböző operációs rendszerekben elterjedt szöveges parancsértelmezők (shell-ek) tipikus példái az interpretált végrehajtásnak. De ha belegondolunk, tulajdonképpen a mikroprogramozott processzorokban maga a mikrokód és az azt futtató vezérlőegység is logikailag egy interpreternek tekinthető a gépi utasítások és a hardver által megvalósítható funkciók között.

Copyright © 2021 / BME AUT

Köztes nyelvek, bájtkód, virtuális gépek

A fordítás és értelmezés nem mindig önállóan jelenik meg. Gyakran alkalmazzák azt a megközelítést, hogy a forráskód valamilyen szintű analízisen átesik, és egy köztes nyelvre (amit gyakran bájtkódnak neveznek) fordítódik. Ez a bájtkód egy virtuális gép számára értelmezhető gépi kódnak tekinthető. A virtuális gépet[2] szoftverben valósítja meg és futtatja a bájtkód interpreter. A megoldás előnye, hogy platformfüggetlen, ezért ugyanaz a kód mindenféle gépen futtatható, amelyre létezik virtuális gép implementáció. Jó példa erre a Java bájtkód és az azt futtatni képes Java VM (virtual machine) alkalmazása.



1.5. ábra – Fordítás és interpretálás együttes alkalmazása

Futásidejű fordítás

A fordítás és interpretálás közti határokat még inkább elmossa a futásidejű fordítás (just-in-time compilation, JIT) módszere. Itt az offline módon korábban lefordított köztes bájtkódot futásidőben egy újabb fordítási fázis alakítja gépi kóddá. Ez a klasszikus interpretáláshoz képest lehetőséget biztosít a köztes kód analízisére és optimalizációjára, emiatt a közvetlen fordításhoz hasonló hatékonyságú futtatható kódot eredményez. Előnye a bájtkód interpretáláshoz képest a megnövekedett hatékonyság. A „hagyományos" fordításhoz képest pedig lehetőséget biztosít arra, hogy futás közben dinamikus optimalizálást végezzen a kódon, és bizonyos programrészeket szükség esetén újra lefordítson, ami akár a statikusan fordított kód hatékonyságát is meghaladhatja. Hátránya, hogy induláskor a bájtkód betöltése és kezdeti fordítása késleltetést okoz a végrehajtásban. A JIT fordítás során a programok egy futtatókörnyezetben (runtime environment) futnak, aminek célja hasonló a bájtkódot futtató virtuális gépekéhez. Manapság a legtöbb Java VM implementáció JIT fordítást alkalmaz, de közismert példa erre a .NET keretrendszer is.

[2] A virtuális gépek alkalmazásának ezt a formáját *folyamatszintű virtualizációnak* nevezzük. Ez esetben egyetlen program (folyamat, processz) futtatható a virtuális környezetben, amely a programfutás végeztével meg is szűnik. Ne tévesszük össze a *rendszerszintű virtualizációval*, amellyel egész számítógépek emulálhatók, operációs rendszerrel együtt.

Copyright © 2021 / BME AUT

Felhasznált és ajánlott irodalom

[1] Nicholas Carter. Schaum's Outline of Computer Architecture. McGraw-Hill, 2002. ISBN 0-07-136207-X

[2] Computer Hope. How many generations of computers are there? https://www.computerhope.com/issues/ch001921.htm (Hozzáférés: 2021.03.12)

[3] Business to Business. The five generations of computers. https://btob.co.nz/business-news/five-generations-computers/ (Hozzáférés: 2021.03.12)

[4] Wikipedia. Computer architecture. https://en.wikipedia.org/wiki/Computer_architecture (Hozzáférés: 2021.03.12)

[5] Wikipedia. Microarchitecture. https://en.wikipedia.org/wiki/Microarchitecture (Hozzáférés: 2021.03.12)

[6] Wikipedia. Instruction set architecture. https://en.wikipedia.org/wiki/Instruction_set_architecture (Hozzáférés: 2021.03.12)

[7] Wikipedia. High-level programming language. https://en.wikipedia.org/wiki/High-level_programming_language (Hozzáférés: 2021.03.12)

[8] Wikipedia. Compiler. https://en.wikipedia.org/wiki/Compiler (Hozzáférés: 2021.03.12)

[9] Wikipedia. Interpreter (computing). https://en.wikipedia.org/wiki/Interpreter_(computing) (Hozzáférés: 2021.03.12)

[10] Wikipedia. Just-in-time compilation. https://en.wikipedia.org/wiki/Just-in-time_compilation (Hozzáférés: 2021.03.12)

[11] Wikipedia. Bytecode. https://en.wikipedia.org/wiki/Bytecode (Hozzáférés: 2021.03.12)

[12] Wikipedia. Virtual machine. https://en.wikipedia.org/wiki/Virtual_machine (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

2 Mikroarchitektúrák

A számítógépek fejlődésének történetében a cél minden időben az egyre nagyobb teljesítőképességű architektúrák kialakítása volt. Ebben a fejezetben a teljesítőképesség növelésére alkalmazott különböző architekturális megoldásokat tekintjük át. Mindenekelőtt meghatározzuk a teljesítőképesség fogalmát és mérésének módszereit. Ezt követően a különböző párhuzamosítási megoldásokat vizsgáljuk meg, kezdve a csővezetékes végrehajtástól (pipelining) a különböző utasításszintű párhuzamossági megközelítéseken át (szuperskalár és VLIW architektúrák) a szálszintű párhuzamosság hardveres támogatásáig.

2.1 Processzorok teljesítőképessége

A processzorok összehasonlításához elengedhetetlen annak meghatározása, hogy mit értünk a teljesítőképesség fogalmán. Általánosságban úgy fogalmazhatjuk ezt meg, hogy mennyi idő alatt képes egy számítógép végrehajtani egy adott programot vagy programokat. Ennek számszerűsítésére különböző mérőszámokat alkottak meg az idők során.

MIPS

A számítástechnika korai évtizedeiben elterjedten használták a MIPS (millions of instructions per second) fogalmát egy számítógép számítási teljesítményének mérésére. Ezt a mérőszámot, amely azt adja meg, hogy átlagosan hány millió utasítást képes másodpercenként feldolgozni egy számítógép, úgy kapjuk meg, hogy osztjuk egy program futása során végrehajtott utasítások számát a teljes futási idővel (μs-ban kifejezve).

$$P_{MIPS} = \frac{\text{utasítások száma}}{\text{végrehajtás ideje } [\mu s]}, \text{ ahol } 1\ MIPS = \frac{1 \text{ utasítás}}{1\ \mu s} \tag{2.1}$$

Például, ha egy 10^6 db utasításból álló programot 1 másodperc alatt hajt végre egy processzor, akkor a MIPS-ben kifejezett teljesítőképessége $P_{MIPS} = \frac{10^6 \text{ utasítás}}{10^6 \mu s} = 1\ MIPS$.

Könnyen beláthatjuk, hogy különböző architektúrák összehasonlítására nem alkalmas ez a mérőszám, hiszen architektúrafüggő, hogy egy adott feladatot hány gépi utasítással lehet leírni (ld. a CISC és RISC utasításkészletek közötti alapvető különbségeket).

CPI, IPC

Egy másik mérőszám azt adja meg, hogy átlagosan hány órajelciklus alatt végez el egy utasítást a processzor. Ez egy program végrehajtásához szükséges órajelciklusok számának és a programfutás során végrehajtott utasítások számának a hányadosa (cycles per instrutcion, CPI).

Copyright © 2021 / BME AUT

$$CPI = \frac{\text{órajelciklusok száma}}{\text{utasítások száma}} \tag{2.2}$$

Léteznek olyan rendszerek, amelyeknél egy órajelciklus alatt több utasítás végrehajtása történik meg (a modern processzorok nagy része ilyen), ilyenkor a CPI helyett annak reciprokát, az IPC (instructions executed per cycle) mértékét adják meg.

$$IPC = \frac{\text{utasítások száma}}{\text{órajelciklusok száma}} = \frac{1}{CPI} \tag{2.3}$$

Fontos látnunk, hogy ez egy külső szemszögből értelmezett átlagos érték, vagyis ne gondoljuk, hogy pl. $CPI = 1$ azt jelentené, hogy egy utasítás végrehajtási ideje önmagában egy órajel lenne. Olyan architekturális megoldások, mint például a csővezetékes (pipeline) végrehajtás teszik lehetővé, hogy ideális esetben minden órajelperiódusban *befejeződjön* egy utasítás akkor is, ha egyébként maga a végrehajtás folyamata több órajelciklust igényel.

Lássunk erre egy példát:

Egy program egy 100 utasításból álló ciklust tartalmaz, amely 42-szer fut le. A végrehajtás 16000 órajelciklust vesz igénybe. Mennyi a rendszer CPI, illetve IPC értéke?

Megoldás: A végrehajtott utasítások száma $42 \cdot 100 = 4200$, ebből következik

$CPI = \frac{16000}{4200} = 3{,}81, \quad IPC = \frac{4200}{16000} = 0{,}2625.$ ■

Önmagukban a CPI és IPC értékek még kevésbé alkalmasak a teljesítőképesség mérésére, mint a MIPS, mert nem tartalmaznak információt a rendszerben alkalmazott órajelfrekvenciáról, sem arról, hogy hány utasítás szükséges egy adott feladat leírásához. Észrevehetjük viszont, hogy az órajelfrekvencia ismeretében a MIPS érték kifejezhető $P_{MIPS} = IPC \cdot f_{\text{clk}}$ formában, ahol a frekvenciát MHz-ben adjuk meg. Ezt illusztrálja a következő példa is:

Számítsuk ki az előző példában szereplő processzor MIPS értékét 10 MHz-es órajel alkalmazása mellett!

Megoldás: A 10 MHz-es órajel 0,1 µs-os órajelperiódust jelent. Mivel a program végrehajtása 16000 órajelciklust vesz igénybe, a végrehajtási idő 1600 µs, azaz 0,0016 s lesz. Ez alatt 4200 utasítás hajtódik végre, ami 1 másodperc alatt $4200/0{,}0016 = 2625000$ utasítást jelentene. Ha ennek vesszük az egymilliomod részét, akkor megkapjuk, hogy a MIPS-ben kifejezett teljesítőképesség 2,625. A közvetlen képlet alkalmazásával ugyanerre az eredményre jutunk:

$P_{MIPS} = IPC \cdot f_{\text{clk}} = 0{,}2625 \cdot 10MHz = 2{,}625\ MIPS.$ ■

Copyright © 2021 / BME AUT

2.1.1 Iron Law

Ahogy korábban is megállapítottuk, a teljesítőképességet az adott feladat végrehajtásának időszükségletével jellemezhetjük. Az előbb látott MIPS, CPI, IPC értékek egyike sem volt alkalmas arra, hogy ezt teljes mértékben kifejezze. Szükség van annak ismeretére is, hogy a végrehajtandó feladat hány utasítással írható le (jelöljük I_c-vel), ami szintén az adott architektúra jellemzője. Így tehát egy feladat (program) végrehajtásának T időszükséglete három tényező szorzataként adható meg:

$$T = I_c \cdot CPI \cdot T_{\text{clk}} \tag{2.4}$$

A 80-as években több független kutató felismerte ezt az összefüggést, amely világosan jellemzi azokat az alapvető tényezőket, melyek hozzájárulnak a processzorok teljesítőképességéhez. A teljesítőképességet, vagy más szóval áteresztőképességet (CPU throughput) W_p-vel jelöljük, és az időszükséglet reciprokaként határozhatjuk meg, mértékegysége feladat/másodperc:

$$W_p = \frac{1}{T} = \frac{f_{\text{clk}}}{I_c \cdot CPI} \tag{2.5}$$

Vagyis a teljesítőképesség fordítottan arányos

- a feladat leírásához szükséges utasítások számával (I_c), amely az utasítások komplexitásával van összefüggésben,
- az egy utasítás végrehajtásához átlagosan szükséges órajelciklusok számával (CPI),
- és egy ciklus időtartamával (órajelfrekvencia).

A (2.4) vagy (2.5) alakban megadott összefüggés *Iron Law of Processor Performance* néven vált ismertté a szakirodalomban.

Úgy tűnik, hogy az áteresztőképesség növelése a (2.5) képlet alapján triviális, az egyes tényezők növelésével, illetve csökkentésével elérhető. Sajnos a tényezők nem függetlenek egymástól, az egyik tényező változtatása ellenkező irányban hathat a másikra, ezért közel sem triviális feladat a teljesítőképesség növelése.

2.1.2 A teljesítőképesség mérése

További nehézséget okoz, hogy milyen programot használva vizsgáljuk a teljesítőképességet. Egy helyfoglalási rendszer programja például primitív utasításokat és gyakori adatbázis-hozzáférést végez, míg egy tudományos vagy mérnöki alkalmazásban a lebegőpontos aritmetikai műveletek a gyakoriak.

A számítógép-architektúrák fejlődésével egyre nehezebbé vált a különböző számítógépes rendszerek összehasonlítása specifikációik alapján. Ezért olyan teljesítménymérésre alkalmas teszteket fejlesztettek ki, melyek egy számítógépes rendszert vagy annak egy komponensét (pl. a processzort) bizonyos alkalmazási területen tipikus terhelésnek teszik ki. Ezek az ún.

Copyright © 2021 / BME AUT

benchmark programok. Közös jellemzőjük, hogy jól értelmezhető és összehasonlítható eredményt szolgáltatnak a tesztelt rendszer teljesítményéről.

A *szintetikus benchmarkok* sok alkalmazás tipikus műveleteit tartalmazó, statisztikai mérések alapján kifejlesztett egyszerű programok. Ezek bizonyos alkalmazási igényeket imitálnak, és leginkább komponens szintű (pl. CPU, hard diszk, hálózat stb.) tesztelésre alkalmasak.

Az *alkalmazási benchmarkok* viszont valóságos, tipikus alkalmazások programjait használják, és ezek teljesítményét mérik. Rendszerszintű tesztelésre alkalmasabbak a szintetikus benchmarkoknál.

A benchmarkok különösen fontosak a CPU-k tervezésében, mivel lehetőséget nyújtanak annak mérésére, hogy a mikroarchitekturális sajátosságok milyen hatást gyakorolnak a teljesítőképességre. Sokféle benchmark létezik, s a gyártók gyakran olyan benchmarkkal reklámozzák terméküket, amely szempontjából az kiemelkedő, ezért ezekkel óvatosan kell bánnunk. Ha a teljesítőképesség kritikus szempont, akkor csak az a benchmark lehet irányadó, amely a célalkalmazásra vonatkozik. A következőkben felsorolunk néhány elterjedten használt benchmark programot, programkészletet.

SPEC CPU Suite

A Standard Performance Evaluation Corporation (SPEC) egy nonprofit szervezet, amely sokféle szabványosított alkalmazási benchmark készletet hozott létre és gondoz. Ezek a „valós élethelyzet” mérését célozzák meg, közülük a *SPEC CPU Suite* benchmark készlet a processzorok teljesítőképességének mérésére fókuszál. Két részből áll:

- A *SPECint* csomag egész aritmetikát igénylő teszteket tartalmaz, amelyek compiler és interpreter programok, fájl- és videótömörítés, valamint mesterséges intelligencia algoritmusok (sakk, go, pályatervezés) futtatásán alapul.
- A *SPECfp* csomag a lebegőpontos számítási teljesítmény mérésére fókuszál. Fizikai szimulációkat, 3D grafikai és képfeldolgozási algoritmusokat, vegyészeti és meteorológiai számításokat, beszédfelismerést futtat és értékel ki.

Ez a benchmark készlet az egyes csomagokban a feladatok eredményét egyformán súlyozza, és az összesített eredmény meghatározásánál *mértani középértéket* használ. Felmerülhet bennünk a kérdés, hogy miért alkalmasabb a mértani közép a számtani középhez képest az átlagos eredmény számítására. Ennek megválaszolásához elevenítsük fel a kétféle középérték számítási módját. A számtani közép az elemek összegét osztja az elemek számával:

$$A = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.6}$$

A mértani közép pedig az n darab elem szorzatának az n-edik gyökét veszi:

Copyright © 2021 / BME AUT

$$A = \sqrt[n]{\prod_{i=1}^{n} x_i} \tag{2.7}$$

Az figyelhetjük meg, hogy egy-egy kiugró érték a mértani közepet kevésbé „húzza el”, emiatt ajánlott az utóbbit alkalmazni. Lássuk ezt egy egyszerű példán keresztül:

Egy benchmark program négyféle tesztet hajtott végre egy adott processzoron, melyeknek eredménye rendre 4, 2, 4 és 82. Határozzuk meg számtani és mértani középértékkel is az eredő pontszámot!

Megoldás:

Számtani középértékkel számolva: $A = \frac{1}{4}(4 + 2 + 4 + 82) = 23$

Mértani középértékkel számolva: $A = \sqrt[4]{4 \cdot 2 \cdot 4 \cdot 82} = 7{,}16$

Láthatjuk, hogy a mértani középérték jobban képviseli a tesztek többségén elért eredményt. ■

LINPACK benchmarkok

A szintetikus benchmarkok közül elterjedten használják a LINPACK benchmarkokat, melyek a lebegőpontos számítási teljesítmény (FLOPS, FLoating point Operation Per Second) mérésére szolgálnak. Ezek a mérnöki feladatokban gyakori problémával, lineáris algebrai egyenletek megoldásával tesztelik a számítógép teljesítőképességét, 64 bites ábrázolási pontosságot használva. A LINPACK benchmarkokat használják a világ 500 leggyorsabb szuperszámítógépét tartalmazó rangsor összeállítására.

Dhrystone

A Dhrystone egy szintetikus benchmark CPU-k általános (integer) teljesítőképességének mérésére, amely egy, sokféle tipikus szoftver analízise alapján kialakított mesterséges, reprezentatív összetételű, ciklikus lefutású program. Leginkább beágyazott processzorok, mikrokontrollerek teljesítménymérésében terjedt el a használata.

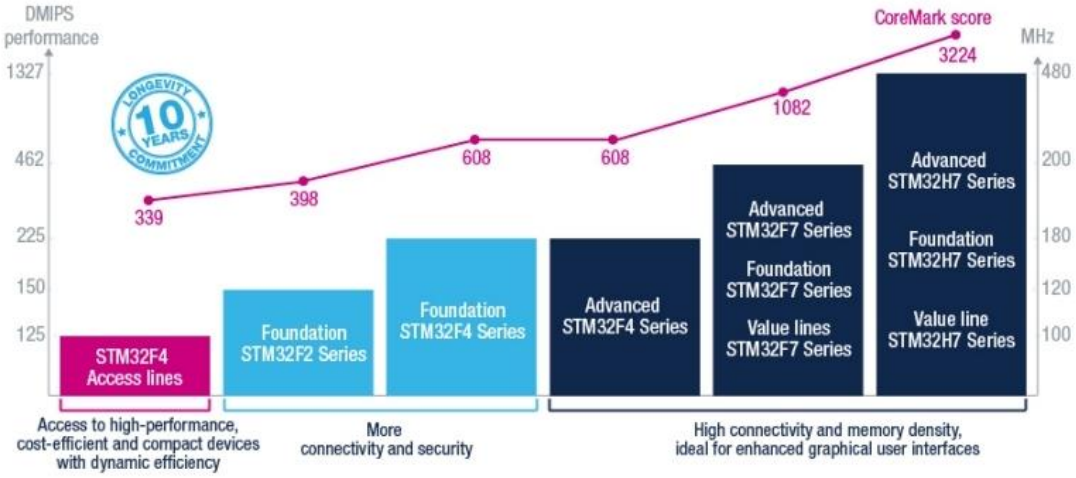
Többféle mérőszámot lehet származtatni a tesztalgoritmusból, ezek alapja a DPS (Dhrystones per second) érték, amely a tesztprogram főciklus lefutásainak számát jelenti másodpercenként. Gyakran találkozhatunk még a DMIPS értékkel, amely a DPS érték 1757-ed része. Hogy miért éppen ennyi? Ehhez érdemes tudnunk, hogy az 1980-as években igen népszerű VAX 11/780 CISC számítógép névleges utasításvégrehajtási sebessége 1 MIPS volt. Ez a gép 1757 DPS eredményt ért el a Dhrystone teszten. Így született az arányszám, amelyet alkalmazva kijelenthető, hogy ezen az architektúrán 1 MIPS számítási teljesítmény 1 DMIPS-nek felel meg. Természetesen más architektúrák esetén ez ebben a formában nem lesz igaz, de viszonyítási

Copyright © 2021 / BME AUT

alapnak megteszi. A processzorok órajel-független összehasonlításához elterjedt még az 1 MHz-re vonatkoztatott DMIPS érték (DMIPS/MHz) is.

CoreMark

A beágyazott processzorok, mikrokontrollerek teljesítménymérésére manapság elterjedt a CoreMark szintetikus benchmark használata is. A következő algoritmusok implementációját tartalmazza: listakezelés (keresés és rendezés), mátrixszorzás, bemeneti adatfolyam ellenőrzése véges állapotú automatával és CRC számítás. Kifejlesztésének egyik célja volt a Dhrystone hátrányos tulajdonságainak kiküszöbölése. Ezek között említhetjük, hogy a Dhrystone érzékeny arra, hogy algoritmusának bizonyos részét a fordítóprogramok kioptimalizálják, ezzel meghamisítva az eredményt. Másrészt a Dhrystone külső függvénykönyvtárakat is felhasznál, és tipikusan ezek a külső függvényhívások nagy részét teszik ki a mért időnek. Ezzel szemben a CoreMark nem használ külső hívásokat, és úgy van megalkotva, hogy minden művelet futási időben származtat olyan bemeneti adatokat, amelyek fordítási időben még nem állnak rendelkezésre, ezzel is „nehezítve” a fordító számára a kód optimalizálását.



2.1. ábra – Az ST Microelectronics ARM Cortex-M alapú mikrokontrollereinek DMIPS és CoreMark eredményei[3]

A beágyazott rendszerekbe szánt mikrokontrollereknél ezen két benchmark eredményét általában feltüntetik a gyártók. Például az ST Microelectronics STM32F446-os, ARM Cortex-M4 magra épülő, 180MHz-es mikrokontrollere 225 DMIPS és 608 CoreMark értéket ért el az adatlapja szerint. A 2.1. ábrán pedig az ST ARM Cortex-M alapú mikrokontroller-családjai DMIPS és CoreMark eredményeinek összehasonlítását figyelhetjük meg.

[3] Forrás: https://www.st.com/content/st_com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus.html (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

2.1.3 A teljesítőképesség növelése

Miután megismertük azokat az eszközöket, amelyekkel mérhető, számszerűsíthető a processzorok teljesítőképessége, felmerül a kérdés, hogy hogyan lehet növelni a teljesítőképességet. Az Iron law (2.5) alapján egyszerűnek tűnik a képlet, növelni kell az órajelfrekvenciát, csökkenteni az egy feladat leírásához szükséges utasítások számát és a CPI értéket. Ahogy korábban is említettük, mindez korántsem triviális.

A számítástechnika első évtizedeiben a gyártástechnológia rohamos fejlődésével az órajelfrekvencia növelése volt az egyik legkézenfekvőbb irány. Az integrált áramkörök bevezetésével és az integráltság fokának exponenciális növekedésével (ld. Moore-törvény) együtt járt az egyre nagyobb frekvenciájú órajelek alkalmazásának lehetősége is. Az 1970-es évek végén látott néhány MHz-es órajelfrekvenciák az 1980-as évek végére elérték a 100MHz-et, majd a következő néhány évben meg sem álltak a GHz-es nagyságrendig. Azonban körülbelül a 2000-es évek elején a nagyívű frekvencianövekedés megtorpanását figyelhetjük meg. Ekkorra érték el a processzorok a néhány (kb. 2–4) GHz-es órajelfrekvenciát, ami néhány kivételtől eltekintve a mai processzorokra is jellemző. A jelenség hátterét a CMOS technológiájú áramkörök dinamikus disszipált teljesítményének összefüggése világítja meg:

$$P_D = f \cdot C_{pd} \cdot V_{CC}^2 \tag{2.8}$$

A disszipált teljesítmény egyenesen arányos a kapcsolási frekvenciával és a CMOS tranzisztorok dinamikus kapacitásával, valamint négyzetesen arányos a tápfeszültséggel. Az integráltság növekedésével az egy lapkán elhelyezett tranzisztorok száma exponenciálisan növekszik, ami az egységnyi felületen disszipált teljesítményt is többszörözi. Ezt egyrészt a tranzisztorok méretének csökkenésével együtt járó C_{pd} csökkenés, másrészt az egyre alacsonyabb magfeszültségek alkalmazása csökkenti. Azonban egy bizonyos integráltsági fok fölött felett ezek a hatások épphogy arra elegendők, hogy a növekvő tranzisztorszám okozta teljesítménynövekedést ellensúlyozzák, és nem marad „tartalék” arra, hogy a frekvenciát érdemben növelni lehessen. Márpedig a termikus viszonyok kézben tartása érdekében a disszipált teljesítményt korlátok közt kell tartani.

Az egy adott programot megvalósító utasítások száma (I_c) függ az (utasításkészlet) architektúra megválasztásától, vagyis az utasítások komplexitásától (ld. CISC, RISC). A számítógépcsaládokon belüli kompatibilitás fenntartása érdekében viszont egy bevált utasításkészlet-architektúrán már nem szoktak alapvető változtatásokat végrehajtani, inkább csak bővítik azokat. Így az utasítások komplexitása lényegében nem változik. Emellett az alkalmazott fordítóprogram is befolyásolja, hogy milyen terjedelmes gépi kóddal sikerül megvalósítani egy magasszintű nyelven leírt algoritmust.

A harmadik komponens, a CPI csökkentése (illetve az IPC növelése) maradt, mint kézenfekvő fejlesztési irány. Ezt sok esetben különböző párhuzamosítási megoldásokkal érik el a (mikro)architektúra-fejlesztők.

A következő, mikroarchitektúrát érintő párhuzamosítási megoldások terjedtek el a teljesítőképesség növelésére:

Copyright © 2021 / BME AUT

- *Párhuzamos processzorok*, többmagos processzorok alkalmazása. Erre részletesebben a 4. fejezetben térünk majd vissza.
- *Utasításszintű párhuzamosság* (instruction-level parallelism, ILP), ezen belül:
  - *Csővezetékes végrehajtás* (pipelining), melynek segítségével a CPI érték 1 közelébe szorítható le.
  - *Szuperskalár csővezeték*, azaz több, párhuzamosan működő csővezeték alkalmazása a processzormagban. Ezzel lehetővé válik az 1 alatti CPI érték (IPC > 1) elérése is.
  - *Sorrenden kívüli végrehajtás* a szuperskalár csővezetékben, amellyel tovább növelhető a ténylegesen elérhető párhuzamosítás mértéke.
  - *Very Long Instruction Word (VLIW) architektúra*, amely szintén párhuzamos csővezetékeket tartalmaz.
- *Szálszintű párhuzamosság* (thread-level parallelism, TLP) megvalósítása egyetlen processzormagban, ezen belül:
  - *Időosztásos többszálúság* (temporal multithreading)
  - *Szimultán többszálúság* (simultaneous multithreading)

A fejezet hátralévő részében az utasításszintű és szálszintű párhuzamosítás különböző megvalósításait vesszük szemügyre.

2.1.3.1 Az Amdahl-törvény

A teljesítőképesség növekedésének jellemzésére az architektúra-fejlesztők gyakran a *sebességnövekedés* (speedup) fogalmát használják, amely azt adja meg, hogy egy architekturális változás vagy más fejlesztés hogyan változtatja meg a teljesítőképességet (ill. egy feladat végrehajtásának idejét). A sebességnövekedést egy hányadossal adhatjuk meg:

$$S = \frac{\text{időszükséglet}_{\text{előtte}}}{\text{időszükséglet}_{\text{utána}}} = \frac{\text{teljesítőképesség}_{\text{utána}}}{\text{teljesítőképesség}_{\text{előtte}}} \qquad (2.9)$$

Például, ha egy program 25 másodperc alatt fut le egy architektúrán, aminek egy továbbfejlesztett változatán már csak 15 másodpercet vesz igénybe ugyanez, akkor a fejlesztéssel elért sebességnövekedés $25\ s/15\ s = 1{,}67$.

Nagyteljesítményű számítógép-architektúrák tervezésekor, új teljesítőképesség-növelő funkciók kialakításakor az egyik legfontosabb irányelv, hogy a gyakran előforduló működési eseteket kell gyorsítani. Vagyis nem csak az a kérdés, hogy egy adott fejlesztés mennyit csökkent a futásidőn, amikor használják, hanem az is, hogy milyen gyakran fordul elő a kérdéses funkció használata. Ezt a szabályt az Amdahl-törvény fogalmazza meg:

$$\text{időszükséglet}_{\text{utána}} = \text{időszükséglet}_{\text{előtte}} \cdot \left[(1-p) + \frac{p}{S_{\text{funkció}}}\right], \qquad (2.10)$$

Copyright © 2021 / BME AUT

ahol $S_{\text{funkció}}$ egy adott új, teljesítménynövelést célzó funkció által önmagában okozott sebességnövekedés, *ha a funkció mindig használatban lenne*. A funkció használatának időhányadát $p \in [0,1]$ jelöli. Fontos, hogy a p használati időhányad az új funkció alkalmazása előtti állapotot tükrözi, ellenkező esetben téves eredményt kapnánk.

Az Amdahl-törvény sebességnövekedést kifejező alakja a következő:

$$S = \frac{\text{időszükséglet}_{\text{előtte}}}{\text{időszükséglet}_{\text{utána}}} = \frac{1}{(1-p) + \frac{p}{S_{\text{funkció}}}} \tag{2.11}$$

Fontos észrevennünk, hogy az elérhető sebességnövekedés elvi maximuma $\frac{1}{1-p}$, amelyet $S_{\text{funkció}} \to \infty$ esetén érnénk el. Vagyis az egy funkció fejlesztésével elérhető maximális sebességnövekedést a funkció használatának időaránya határozza meg. Illusztráljuk ez egy példán keresztül!

Tegyük fel, hogy egy architektúrában nincs hardveres támogatás a szorzás elvégzésére, ezért ezt a műveletet összeadásokra kell visszavezetni egy szoftveres szubrutinnal. A szoftveres megvalósítás 200 órajelciklust vesz igénybe. A hardveres szorzás bevezetésével ugyanez 4 órajelciklus alatt elvégezhető lenne.

a) Mekkora sebességnövekedést jelentene a hardveres szorzás bevezetése, ha a program jelenleg a futási idejének 10 százalékát tölti szorzás műveletek elvégzésével?

b) Mekkora a sebességnövekedés egy olyan program esetében, amelyben a szorzással töltött idő aránya 40 százalék?

Megoldás:

A hardveres szorzás miatti sebességnövekedés szorzó műveletek végrehajtásakor:

$$S_{\text{HW szorzás}} = \frac{200}{4} = 50$$

a) A szorzó műveletek 10%-os aránya esetén $p = 0{,}1$, ekkor az Amdahl-törvény alapján a teljes sebességnövekedés

$$S = \frac{1}{0{,}9 + \frac{0{,}1}{50}} = 1{,}109\,,$$

vagyis 10,9%-kal nőtt a rendszer teljesítőképessége.

b) A szorzó műveletek 40%-os aránya esetén $p = 0{,}4$, ekkor

Copyright © 2021 / BME AUT

$$S = \frac{1}{0{,}6 + \frac{0{,}4}{50}} = 1{,}645\,,$$

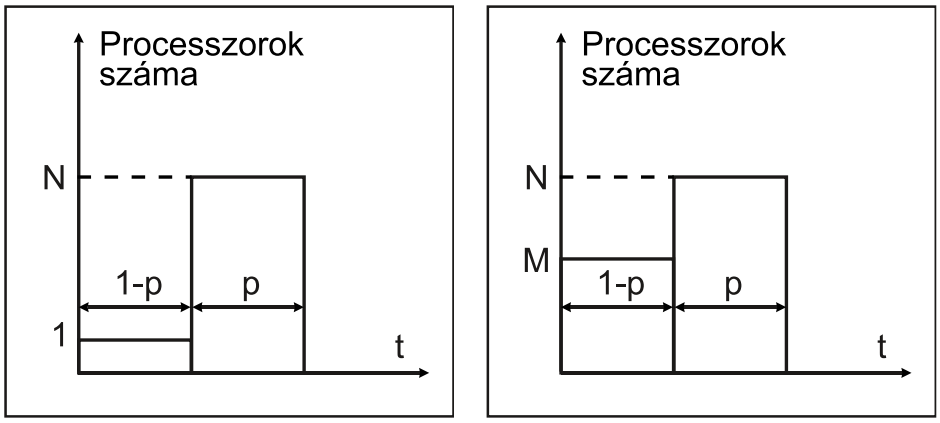
azaz ilyenkor 64,5%-os javulást tapasztalunk. ■

2.1.3.2 Az Amdahl-törvény alkalmazása különböző párhuzamosítási megoldások esetén

Egy rendszer áteresztőképességét vagy sávszélességét az időegység alatt elvégzett feladatok számával jellemezhetjük, amint azt a (2.5) egyenletben is láttuk. Ha egy rendszer egyidejűleg csak egyetlen feladaton dolgozik, akkor a sávszélessége egyenlő $1/D$-vel, ahol D a feladat által a rendszerben eltöltött idő, másszóval késleltetés. Ha a rendszert egyidejűleg több feladat ellátására alkalmassá tesszük, akkor a sávszélesség megnövekedhet a D csökkenése nélkül. A különböző párhuzamosítási törekvések erre az alapvető felismerésre épülnek.

Párhuzamos processzorok sebességnövekedése

A párhuzamosítás alapvető megközelítése, ha több processzort alkalmazunk a rendszerben, amelyek különböző programszálakat futtatnak egy időben.



a) b)

2.2. ábra – Párhuzamos processzorok sebességnövekedése a) csak teljes párhuzamosítást feltételezve, b) részleges párhuzamosítást is lehetővé téve.

Ha N db processzort alkalmazunk, és a programot teljesen (N-szeresen) párhuzamosítható és tisztán szekvenciális programrészekből állónak feltételezzük (ld. 2.2a ábra), akkor az Amdahl-törvényt alkalmazva az elvi sebességnövekedés

$$S = \frac{1}{(1-p) + \frac{p}{N}} \tag{2.12}$$

Copyright © 2021 / BME AUT

ahol p a teljesen párhuzamosítható programrészek időaránya. Ha N nagyon nagy, akkor a sebességnövekedést a soros rész határozza meg:

$$\lim_{N\to\infty} S = \frac{1}{1-p} \tag{2.13}$$

Ezt a szakirodalomban *soros útszűkületnek* (sequential bottleneck) nevezik.

Ha azt feltételezzük, hogy az $1-p$ időhányadban is elérhető egy részleges, átlagosan $M < N$ mértékű párhuzamosítás (ld. 2.2b ábra), akkor a sebességnövekedés

$$S = \frac{1}{\frac{1-p}{M} + \frac{p}{N}} \tag{2.14}$$

Tegyük hozzá, hogy az elvi sebességnövekedésnél a valóságos érték mindig kisebb. Hiszen szükség van kommunikációra, szinkronizációra is a közös feladaton dolgozó processzorok között, amely időveszteséggel jár.

Tekintsünk egy példát a processzorok száma és az elérhető sebességnövekedés közötti összefüggésre:

Egy többprocesszoros rendszerben legyen a párhuzamosítható programrészek aránya 75%, és az elérni kívánt sebességnövekedés 4-szeres.

a) Hány processzorra van szükségünk, ha a nem teljesen párhuzamosítható részeknél is átlagosan elérhető egy 2-szeres párhuzamosítás?

b) Hány processzorra van szükségünk, ha a teljesen párhuzamosítható részeken kívül csak 1 processzor tud dolgozni?

Megoldás:

a) Adataink: $p = 0{,}75$; $S = 4$; $M = 2$. Az Amdahl-törvényt átrendezve megkapjuk a szükséges processzorok számát:

$$S = \frac{1}{\frac{1-p}{M} + \frac{p}{N}} \quad \rightarrow \quad N = \frac{SMp}{M - S(1-p)}$$

$$N = \frac{4 \cdot 2 \cdot 0{,}75}{2 - 4 \cdot 0{,}25} = 6$$

b) Adataink: $p = 0{,}75$; $S = 4$; $M = 1$. Ezeket helyettesítsük be az előbb kapott képletbe:

$$N = \frac{4 \cdot 1 \cdot 0{,}75}{1 - 4 \cdot 0{,}25} = \frac{3}{0} \rightarrow \infty,$$

Copyright © 2021 / BME AUT

vagyis ilyen mértékű szekvenciális rész esetén nem érhető el 4-szeres sebességnövekedés, hiszen ebben az esetben az elvi maximum

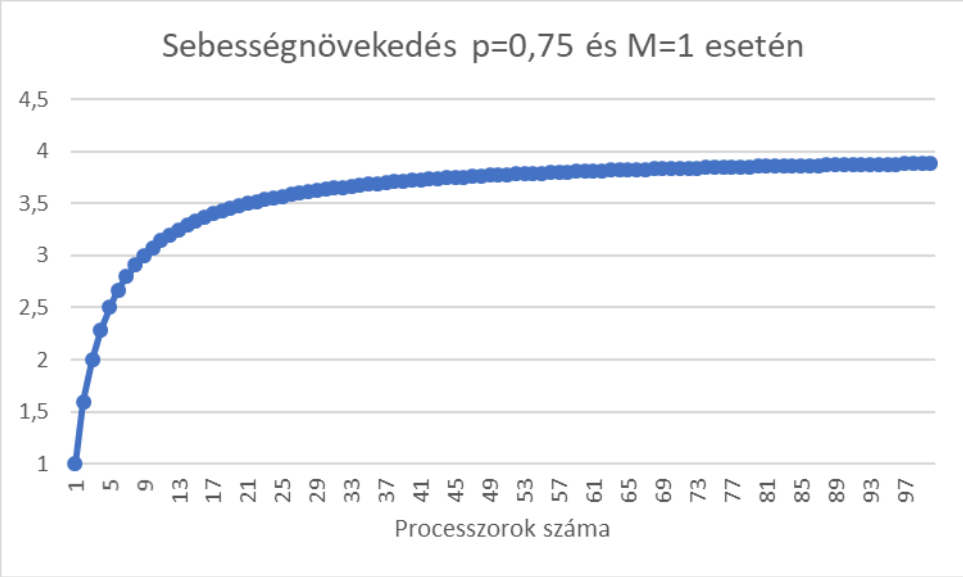
$$\lim_{N\to\infty} S = \frac{1}{1-0{,}75} = 4$$

Nézzük meg, hogy mekkora lenne az elméleti sebességnövekedés értéke 100 párhuzamos processzor esetén:

$$S = \frac{1}{\frac{1-p}{M} + \frac{p}{N}} = \frac{1}{\frac{0{,}25}{1} + \frac{0{,}75}{100}} = 3{,}88$$

A 2.3. ábrán láthatjuk, hogyan alakul S értéke egyre több párhuzamos processzor esetén. Megfigyelhetjük a soros útszűkület korlátozó hatását.

Példánk alapján is jól látható, hogy a programok részleges párhuzamosíthatósága domináns hatással lehet a sebességnövekedésre, illetve jelentős csökkenést okozhat a szükséges processzorszámban. ■



2.3. ábra – A párhuzamos processzorok számának hatása az elméleti sebességnövekedésre

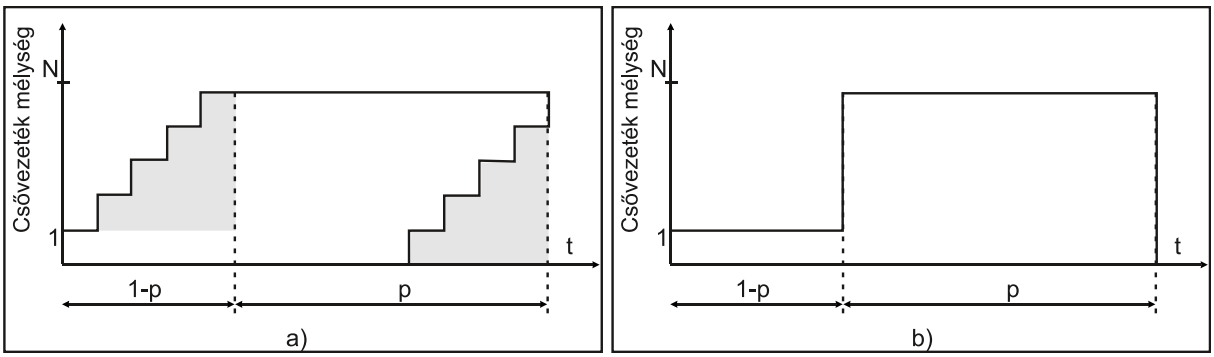
Sebességnövekedés csővezetékes végrehajtás esetén

Csővezetékes feldolgozás esetén az egymást követő ciklusokban (órajelekben) újabb és újabb utasítások lépnek be a feldolgozási folyamatba. Egyidejű feldolgozás alatt annyi utasítás áll, amennyi a csővezeték fázisainak száma. A feldolgozás befejezésekor órajelenként eggyel csökken a feldolgozás alatt álló utasítások száma (2.4a ábra). Átrajzolva a 2.4a ábrát, a 2.4b ábra szerint megkaphatjuk a sebességnövekedésre vonatkozó összefüggést:

Copyright © 2021 / BME AUT

$$S = \frac{1}{(1-p) + \frac{p}{N}} \tag{2.15}$$

A (2.15) egyenletben p jelentése: a teljes feldolgozási idő p hányadában N utasítás végrehajtása folyik, $1-p$ hányadban pedig egyetlené. A maximális sebességnövekedés $p = 1$ esetén lép fel, értéke N.



2.4. ábra – Csővezeték-szervezés analógiája a párhuzamos processzorokkal

A valóságban vannak tényezők, amelyek miatt $p < 1$, azaz a csővezeték „bedugul”, vagy „buborékok” keletkeznek benne. A programfutás elején és végén a csővezeték feltöltése, ill. kiürítése miatt mindenképpen megvan ez a hatás N órajelen keresztül. De más hatások is okozhatják a „bedugulás” létrejöttét (feltételes elágazások, adatfüggőségek stb.), amint azt a későbbiekben részletesen is látni fogjuk. Ilyenkor a „bedugulást” az okozza, hogy a csővezetéket részben vagy teljesen ki kell üríteni a hibamentes programvégrehajtás érdekében. A részleges bedugulást is figyelembe vevő pontosabb összefüggés:

$$S = \frac{1}{\frac{p_1}{1} + \frac{p_2}{2} + \cdots + \frac{p_N}{N}} \tag{2.16}$$

ahol p_i az időnek azt a hányadát jelöli, melyben i utasítás van a csővezetékben.

2.1.3.3 Az Amdahl-törvény alkalmazása többféle teljesítőképesség-növelő hatás együttes vizsgálata esetén

Az Amdahl-törvény akkor is alkalmazható, ha nem egy, hanem többféle architekturális fejlesztés együttes hatását szeretnénk megvizsgálni. Ennek illusztrálására nézzünk egy olyan példát, amikor két teljesítőképesség-növelő funkcióból a program egy részére mindkettő, más részére csak az egyik, a program többi részére pedig egyik funkció sincs hatással.

Copyright © 2021 / BME AUT

Egy matematikai program futtatásához keresünk processzort. Az általunk preferált gyártó többféle változatban is kínál egy konkrét processzortípust:

- Egymagos vagy kétmagos változatban
- SIMD (single instruction multiple data) műveletvégzéssel vagy anélkül

A program 70%-át több szálon futtatható, párhuzamos számítások teszik ki. Ezek a számítások 25%-ban tartalmaznak vektorműveleteket, amelyek végrehajtása átlagosan 5-szörösére gyorsítható az SIMD motor használatával.

A processzor legegyszerűbb képességű (egymagos, SIMD nélküli) változatához képest mekkora elvi sebességnövekedés érhető el:

a) a kétmagos változattal?

b) az SIMD motor alkalmazásával?

c) a kétmagos, SIMD motorral rendelkező változattal?

Megoldás:

a) Kétmagos processzorral elérhető sebességnövekedés SIMD motor nélkül

A párhuzamosítható programrészek aránya és a két processzormag miatti sebességnövekedése:

$$p_{par} = 0{,}7$$
$$S_{par} = N = 2$$

A két processzormag miatti eredő sebességnövekedés a teljes programra nézve:

$$S = \frac{1}{(1 - p_{par}) + \frac{p_{par}}{N}} = \frac{1}{0{,}3 + \frac{0{,}7}{2}} = \mathbf{1{,}538}$$

b) Egymagos processzor SIMD motorral elérhető sebességnövekedése

A vektorműveletek sebességnövekedése az SIMD motor használatával, valamint eredeti arányuk a párhuzamosítható részben:

$$S_{vec} = 5$$
$$p_{par,vec} = 0{,}25$$

A vektorműveletek aránya a teljes programban:

$$p_{vec} = p_{par,vec} \cdot p_{par} = 0{,}25 \cdot 0{,}7 = 0{,}175$$

Copyright © 2021 / BME AUT

Az SIMD motor által okozott eredő sebességnövekedés a teljes programra nézve:

$$S = \frac{1}{(1 - p_{vec}) + \frac{p_{vec}}{S_{vec}}} = \frac{1}{0{,}825 + \frac{0{,}175}{5}} = \mathbf{1{,}163}$$

c) Kétmagos processzor SIMD motorral elérhető sebességnövekedése

A párhuzamosítható programrész sebességnövekedése az SIMD motor miatt (de még csak egyetlen processzormagot feltételezve):

$$S_{par,SIMD} = \frac{1}{\left(1 - p_{par,vec}\right) + \frac{p_{par,vec}}{S_{vec}}} = \frac{1}{0{,}75 + \frac{0{,}25}{5}} = \mathbf{1{,}25}$$

A párhuzamosítható programrész kombinált sebességnövekedése a két processzormag és az SIMD motor együttes hatása miatt:

$$S_{kombi} = N \cdot S_{par,SIMD} = 2 \cdot 1{,}25 = 2{,}5$$

Az eredő sebességnövekedés a teljes programra nézve:

$$S = \frac{1}{\left(1 - p_{par}\right) + \frac{p_{par}}{S_{kombi}}} = \frac{1}{0{,}3 + \frac{0{,}7}{2{,}5}} = \mathbf{1{,}724}$$

Ezt az eredményt egy lépésben is megkaphatjuk, ha különválasztjuk azokat a programrészeket, amelyekre mindkét funkció, csak az egyik funkció, illetve egyik funkció sincs hatással. A párhuzamosítható vektorműveletek aránya a teljes programban és elemi sebességnövekedése a két funkció együttes hatása miatt:

$$p_{dual,SIMD} = p_{par,vec} \cdot p_{par} = 0{,}25 \cdot 0{,}7 = 0{,}175$$
$$S_{dual,SIMD} = N \cdot S_{vec} = 2 \cdot 5 = 10$$

A párhuzamosítható, de SIMD motorral nem gyorsítható műveletek aránya a teljes programban és elemi sebességnövekedése a két processzormag miatt:

$$p_{dual} = \left(1 - p_{par,vec}\right) \cdot p_{par} = 0{,}75 \cdot 0{,}7 = 0{,}525$$
$$S_{dual} = N = 2$$

Az eredő sebességnövekedés a teljes programra nézve:

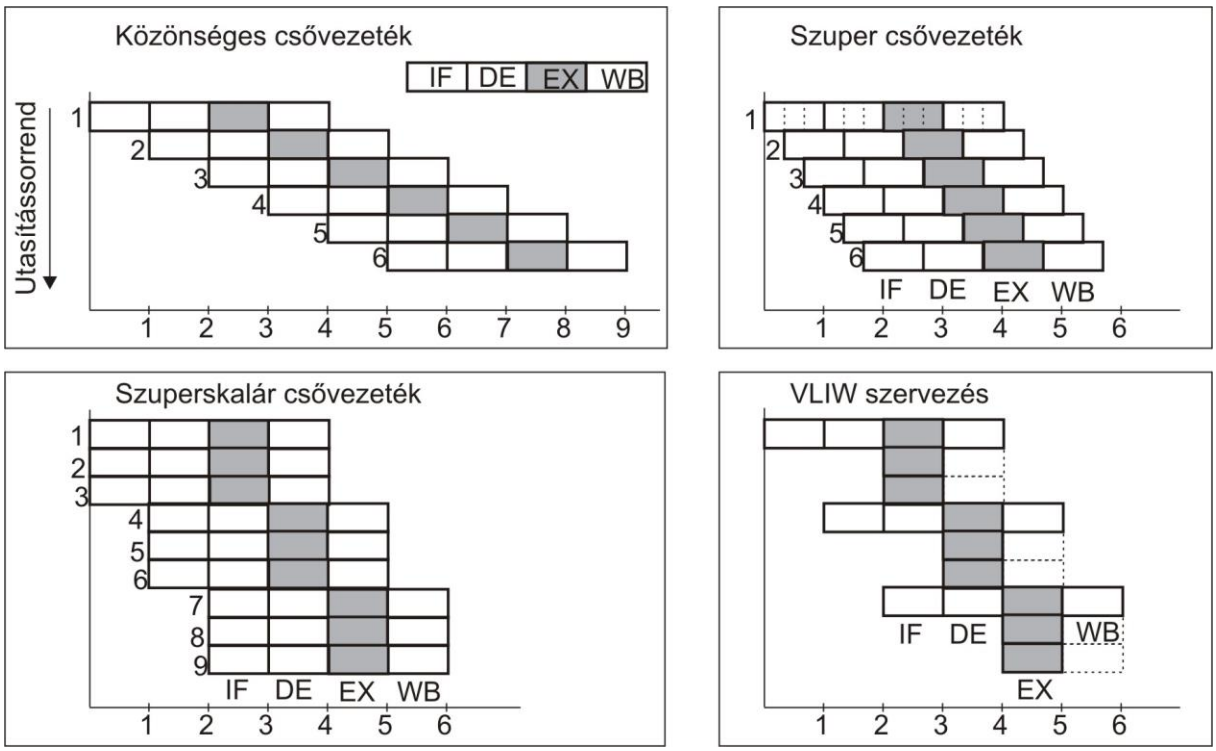
$$S = \frac{1}{\underbrace{\left(1 - p_{dual} - p_{dual,SIMD}\right)}_{0{,}3} + \underbrace{\frac{p_{dual}}{N}}_{0{,}2625} + \underbrace{\frac{p_{dual,SIMD}}{N \cdot S_{vec}}}_{0{,}0175}} = \frac{1}{0{,}58} = \mathbf{1{,}724}$$

■

Copyright © 2021 / BME AUT

2.2 Csővezetékes processzorok

A csővezetékes szervezés (pipelining) bevezetésének elsődleges célja a teljesítőképesség növelése a hardver kismértékű növelésével, szemben a multiprocesszoros rendszerekkel. A különböző csővezeték kialakítások az *utasításszintű párhuzamosság* (instruction-level parallelism, ILP) különböző megvalósítási formái.



2.5. ábra – Utasításszintű párhuzamosság megvalósítása különböző csővezeték-megoldásokkal

A 2.5. ábra négy különböző megvalósítást szemléltet:

- A közönséges utasítás-csővezetékben az utasítások végrehajtását 3–6 db alapvető fázisra bontják (az ábrán 4 db), és az egyes fázisokat különálló hardverfokozatok végzik el. Így, a végrehajtás átlapolódásával minden órajelciklusban elkezdődhet, ill. befejeződhet egy újabb utasítás.
- A szuper csővezetékes szervezés (superpipelining) célja az előbbi fázisok mind apróbb részekre bontása, azaz a csővezeték mélyítése (vertikális mélyítés). Minél kisebb egy elemi feldolgozási lépés ideje, annál rövidebb lehet a végrehajtást ütemező órajel periódusideje. Egy utasítás teljes végrehajtási ideje ugyanannyi, mint az előző esetben, de a megnövelt párhuzamossági fok miatt gyakrabban lépnek be, ill. fejeződnek be az utasítások.
- A szuperskalár szervezés több, párhuzamos csővezetéket, vagyis az egyes fázisok végrehajtóegységeinek többszörözését jelenti a végrehajtóműben (horizontális

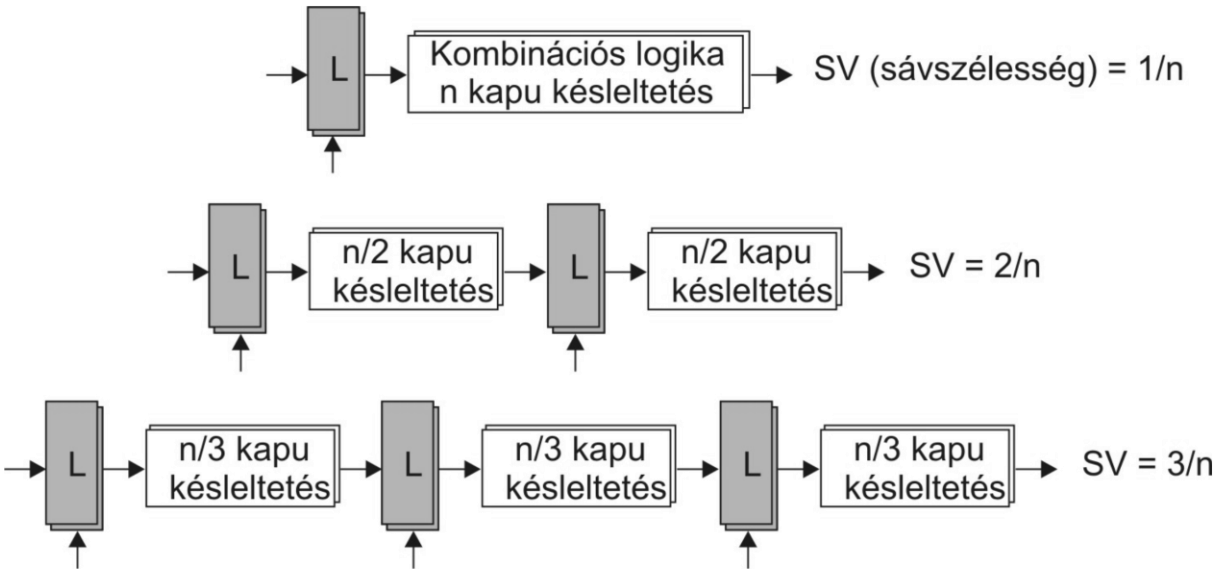
Copyright © 2021 / BME AUT

mélyítés). Mivel egyszerre több utasítás lehet a párhuzamos csővezetékek azonos fázisaiban is, ez a megoldás az átlapolódás hatásán túl további áteresztőképesség-növekedést okoz.

- A VLIW (very long instruction word) architektúra a szuperskalár processzorokhoz hasonlóan több, párhuzamos csővezetéket használ. Viszont míg a szuperskalár processzoroknál a párhuzamosításnak ezt a formáját teljes mértékben elfedi a hardver, addig a VLIW processzoroknál az utasítások szintjén a programozó adja meg, hogy melyik elemi utasítás melyik végrehajtóegységhez kerüljön. A párhuzamosan végrehajtandó utasítások alkotják a hosszú utasításszót.

2.2.1 A csővezeték működési elve

A csővezetékes szervezés a rendszert fázisokra bontja, a fázisok között átmeneti tárolókat használva. Ezek a fázisok és az átmeneti tárolók alkotják a csővezetéket. Az eredeti rendszer által elvégzendő számítási feladat végrehajtási idejét (késleltetését) jelöljük D-vel. Ha a számítási feladatot k részfeladatra bontjuk, amelyeket a csővezeték egyes fázisai végeznek el, akkor ahelyett, hogy D időnként indítanánk el egy új feladat feldolgozását, most ezt D/k időnként tehetjük meg. Ehhez feltesszük, hogy az egyes fázisokat úgy alakítottuk ki, hogy azok késleltetése D/k, azaz az eredeti késleltetés egyenletesen van elosztva a fázisok között, s a beiktatott átmeneti tárolók nem okoznak járulékos késleltetést. Ekkor a rendszer sávszélessége k-szorosra növekedhet. A 2.6. ábra illusztrálja a k-szoros sávszélesség-növekedést a k-fokozatú csővezetékben.



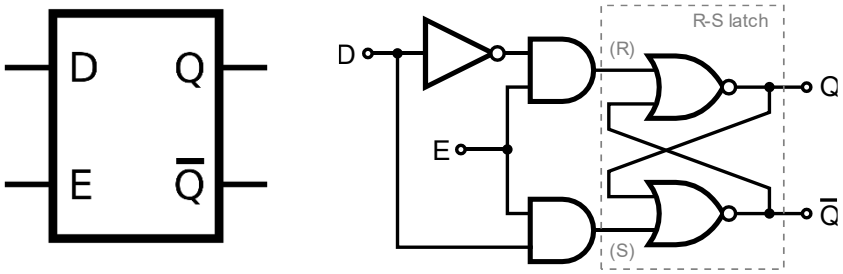
2.6. ábra – A csővezeték működési elve

Ismételjük át, hogy milyen eszközöket alkalmazhatunk átmeneti tárolóként. A digitális technikában egy bitnyi adat vagy állapot tárolására latch és flip-flop áramköröket alkalmaznak. A két fogalom gyakran összemosódik még a szakirodalomban is, de a legelterjedtebb

Copyright © 2021 / BME AUT

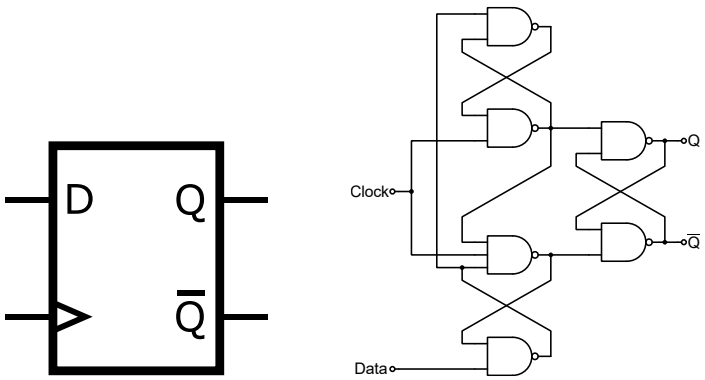
megállapodás szerint a szintvezérelt (aszikron) tárolókat nevezzük latch-nek, az élvezérelt (szinkron) tárolókat pedig flip-flopnak. Felhasználói szempontból a legegyszerűbb esetben egy adatbemenettel (D) és egy engedélyező (E) vagy órajel (Clk) bemenettel rendelkeznek. Szintvezérelt esetben az engedélyező bemenet aktív állapotában a latch „átlátszó", vagyis az adatbemenet állapota folyamatosan a kimenetre másolódik. Élvezérelt esetben az órajel valamelyik irányú szintváltozása váltja ki az adatbemenet átmásolását. Inaktív engedélyező bemenet, illetve élváltás hiánya esetén a kimenet megtartja az aktuális értékét.

A 2.7. ábrán egy szintvezérelt D-G latch (data-gate latch, gated D latch) szimbóluma és megvalósítása látható. Belsejében felismerhetjük a két visszacsatolt NOR kapuból álló R-S (reset-set) tárolót, amelynek bemeneteit a D bemenőjel ponált és negált változata adja, az engedélyező jellel kapuzva.



2.7. ábra – D-G latch logikai szimbóluma és megvalósítása kapuáramkörökkel[4]

Az élvezérelt D (data) flip-flop szimbólumát és megvalósítását mutatja a 2.8. ábra. Látható, hogy jóval több kapuból építhető fel, mint a D-G latch.



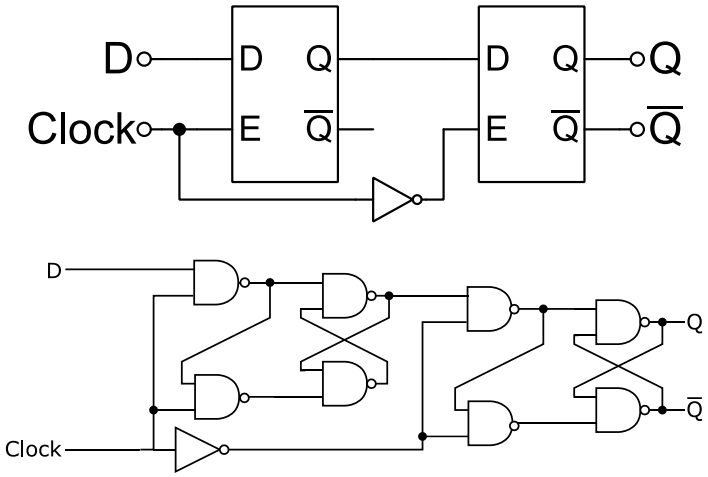
2.8. ábra – D flip-flop logikai szimbóluma és megvalósítása kapuáramkörökkel[4]

A D flip-flopnak létezik master-slave megvalósítása is, amely nem más, mint két D-G latch egymás után kötve, ellenfázisú órajelet kapcsolva a két fokozat engedélyező bemenetére (2.9. ábra). A D bemenet értéke két ütemben kerül át a kimenetre, hiszen a két D-G latch az órajel ellentétes szintjénél lesz átlátszó. A két fokozat közötti adatátadás az ábrán látott

[4] Forrás: https://en.wikipedia.org/wiki/Flip-flop_(electronics) (Hozzáférés: 2021.03.12)

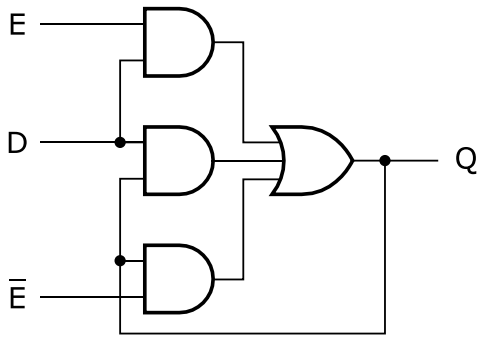
Copyright © 2021 / BME AUT

kiépítésnél az órajel lefutó élénél történik meg. Így lehet szintvezérelt tárolókat élvezérelt módon működtetni. Ehhez hasonló megoldással egy nagyobb kombinációs hálózat működését is részekre lehet osztani. Ha az ellenütemű órajelekkel vezérelt D-G latch-ek közé más kapukból felépített kombinációs hálózatrészeket teszünk, az ezek által végzett részműveletek eredményei szintén az órajel élváltásainak időpontjában terjednek tovább.



2.9. ábra – Master-slave D flip-flop megvalósítása 2 db D-G latch-ből[4]

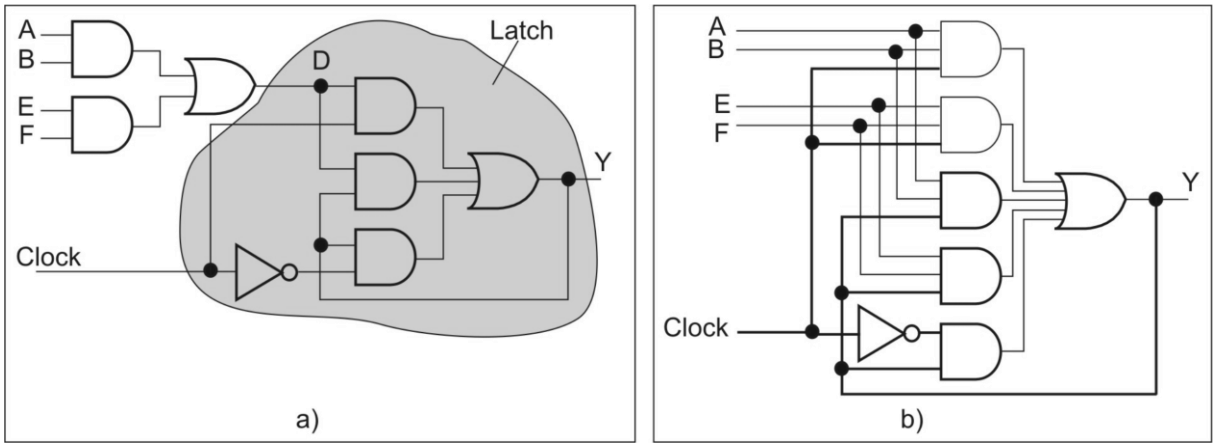
Az előzőekben azzal a feltevéssel éltünk, hogy a fokozatok közötti átmeneti tárolók nem okoznak járulékos késleltetéseket. Ennek van reális alapja. A 2.10. ábrán látható ún. Earle latch, mint a fázisok közötti átmeneti tároló először az IBM System/360 Model 91 számítógépben került alkalmazásra a szorzó egység összeadó részegységei között (John G. Earle, 1966).



2.10. ábra – Earle latch

Az Earle latch alkalmazása több szempontból is előnyösebb D-G latch egyszerű megvalósításánál. A D-G latch 2.7. ábrán látható egyszerű megvalósításának hátránya, hogy 2 vagy 3 kapuidőnyi a késleltetése, ami ráadásul a bemenet és az aktuális állapot értékétől függ. Ezzel szemben az Earle latch fixen 2 kapuidőnyi késleltetéssel rendelkezik. Egy másik rendkívül előnyös tulajdonsága, hogy egybeépíthető a megelőző kombinációs logika utolsó két kapuszintjével, így valóban nem okoz járulékos késleltetést.

Copyright © 2021 / BME AUT



2.11. ábra – Az Earle-latch alkalmazása. a) Hagyományos alkalmazás, b) beolvasztás a megvalóstandó kombinációs logika utolsó két fázisába

A 2.11a ábra az AB+EF összefüggéssel megadott funkciót megvalósító kétszintű logikát és egy hagyományosan illesztett átmeneti tárolót (Earle latch) tartalmazó csővezeték fázist szemléltet. Itt a két kapuáramkörnyi késleltetés néggyé változott a latch beiktatásával. A 2.11b ábrán a latch a megvalósítandó funkcióba lett beolvasztva, így az eredő késleltetés továbbra is két kapuáramkörnyi. Megjegyzés: A 2.10. és a 2.11a ábrán az Earle latch-ben látható középső ÉS kapu a hazárdmentesítést szolgálja. Hasonló a feladata a 2.11b ábrán az alulról számított második és harmadik ÉS kapunak.

Mivel a sávszélesség növekedése a csővezeték mélységével növelhető, úgy tűnik, a tervezésben ennek maximalizálására kell törekedni. A csővezetéket ütemező órajel periódusidejének minimumát, ill. maximális frekvenciáját a következőképpen fejezhetjük ki:

$$\tau_{min} \geq \max(\tau_i) + \tau_L,$$
$$f_{max} = \frac{1}{\tau_{min}} \tag{2.17}$$

ahol τ_i az egyes csővezeték-fázisok késleltetése, τ_L pedig a latch késleltetése. Itt láthatjuk, hogy a fenti Earle-latch megoldás milyen módon járul hozzá a maximális alkalmazható frekvencia növeléséhez: gyakorlatilag $\tau_L = 0$ lesz. Ilyenkor a leglassabb csővezeték-fázis határozza meg az alkalmazható maximális órajelfrekvenciát.

A gyakorlatban tovább csökkenti az órajel frekvenciájának felső határát a jelelcsúszás, aminek következtében az órajel nem egy időben érkezik meg az egyes fázisokhoz.

A csővezetékes processzorok esetén további komoly nehézséget okoz (lásd később), hogy a csővezeték egyes fázisai között bizonyos utasításoknál (pl. elágazás, azonos regiszterek használata stb.) egymásra hatás léphet fel, ami a mélység bizonyos határon túli növelését gazdaságtalanná teszi.

Copyright © 2021 / BME AUT

2.2.2 Aritmetikai csővezeték

Csővezetékes szervezést nem csak az utasításvégrehajtás esetében lehet alkalmazni, hanem általánosságban minden szekvenciális jellegű, fázisokra bontható műveletvégzésnél. Például bizonyos aritmetikai műveletek megvalósításában hatékony az aritmetikai csővezetékek használata. Ebbe a kategóriába tartoznak a lebegőpontos számokkal végzett műveletek (pl. összeadás, szorzás), de bizonyos fixpontos műveletek is, amelyek szekvenciális algoritmussal hajthatók végre (pl. szorzás, osztás). Jelprocesszorok, aritmetikai társprocesszorok esetében előnyös az aritmetikai csővezetékek használata, hiszen, ha sok egymás utáni hasonló művelet elvégzésére van szükség, az egyes fázisok időbeli átlapolásával lerövidíthető a végrehajtás ideje.

Az aritmetikai csővezeték működését egy lebegőpontos összeadó példáján mutatjuk be. Emlékezzünk vissza, hogy a lebegőpontos számábrázolás matematikai alapja a normálalak, ahol egy $[1,2)$ intervallumba normált mantissza és 2 valahányadik hatványának szorzataként adjuk meg a kérdéses számot:

$$x = \pm m \cdot 2^k \tag{2.18}$$

A legelterjedtebb lebegőpontos számábrázolási formátumot az IEEE 754 szabvány határozza meg. Ennek részletes tárgyalása nem célunk, csupán néhány alapvető tulajdonságát ismételjük át, amelyek a példánk megértéséhez szükségesek.

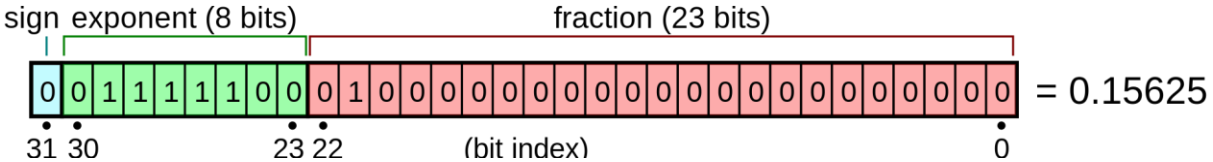
Ahogy fent már említettük, $m \in [1,2)$, azaz a mantissza egészrésze mindig 1. Ebből következik, hogy elegendő a törtrészt ábrázolni, vagyis az egészrész 1-esének elhagyásával egy bitet „spórolunk”. Ezt nevezzük rejtett 1-esnek, amit mindig oda kell érteni, de nem kell ábrázolni[5]. A törtrészt (fraction) előjeles abszolútértékes kódolásban (signed magnitude) tároljuk, vagyis a törtrész abszolútértéke mellett egy külön bit jelzi az előjelét (0: pozitív, 1: negatív).

A k kitevőt *karakterisztikának* nevezzük, és ofszet kódolással ábrázoljuk. Ez azt jelenti, hogy a kitevő előjeles értékéhez hozzáadjuk a bitmező méretéből adódó ábrázolási tartomány félértékének 1-gyel csökkentett változatát (pl. 8 bit esetén 127-et), és így kapjuk az ábrázolandó *exponens* értéket, amely így mindig pozitív lesz.

A szabvány többféle ábrázolási pontosságot ismer, ezek közül az *egyszeres pontosságú* (single precision) alak 32 biten ábrázolja a számokat 1 bit előjel, 8 bit exponens és 23 bit törtrész felosztásban. Ezeket a bitmezőket láthatjuk a 2.12. ábrán egy példán keresztül.

[5] Ez a szabvány szerinti *normalizált* alak. Speciális esetként létezik az ún. *denormalizált* alak is, ahol az egészrész 0, de ezzel itt terjedelmi okokból nem foglalkozunk.

Copyright © 2021 / BME AUT



2.12. ábra – Egyszeres pontosságú lebegőpontos számformátum (IEEE 754)[6]

A 2.12. ábrán látható egyszeres pontosságú lebegőpontos számban az egyes mezők értéke és értelmezése következő:

Előjel:	0 (pozitív)
Exponens:	$e = 01111100_b = 64 + 32 + 16 + 8 + 4 = 124$
Karakterisztika:	$k = e - 127 = -3$
Törtrész:	$f = 0.01_b = 0.25$
Mantissza:	$m = 1.f = 1.01_b = 1.25$
Az ábrázolt szám:	$+1.01_b \cdot 2^{-3} = 0.00101_b = 0.15625$ ■

Tekintsük ezek után a lebegőpontos összeadás, ill. kivonás műveletet, és azt, hogy miként lehet ezt csővezetékesen megvalósítani. Az összeadó aritmetikai csővezeték blokkvázlatát a 2.13. ábrán láthatjuk. Normálalakban megadott számok összeadása, ill. kivonása a következő formában írható fel:

$$x_3 = x_1 \pm x_2 = (\pm\, m_1 \cdot 2^{k_1}) \pm (\pm\, m_2 \cdot 2^{k_2}) = \pm\, m_3 \cdot 2^{k_3} \tag{2.19}$$

Az eredmény úgy számítható ki, ha a két szám kitevőjét összehangoljuk, mert az így átskálázott mantisszák illesztetté válnak, ezért közvetlenül összeadhatók, ill. kivonhatók:

$$\begin{aligned} x_1 &= \pm\, m_1 \cdot 2^{k_1} = \pm\, m_1' \cdot 2^{k_3'} \\ x_2 &= \pm\, m_2 \cdot 2^{k_2} = \pm\, m_2' \cdot 2^{k_3'} \\ x_1 \pm x_2 &= \big((\pm\, m_1') \pm (\pm\, m_2')\big) \cdot 2^{k_3'} = \pm\, m_3' \cdot 2^{k_3'} \end{aligned} \tag{2.20}$$

Előfordulhat, hogy az összeg mantisszája nem az $[1,2)$ tartományba adódik, ekkor szükség van egy utólagos normalizálási lépésre, azaz a kitevő igazítására:

$$\pm m_3' \cdot 2^{k_3'} \rightarrow \pm m_3 \cdot 2^{k_3},\ m_3 \in [1,2) \tag{2.21}$$

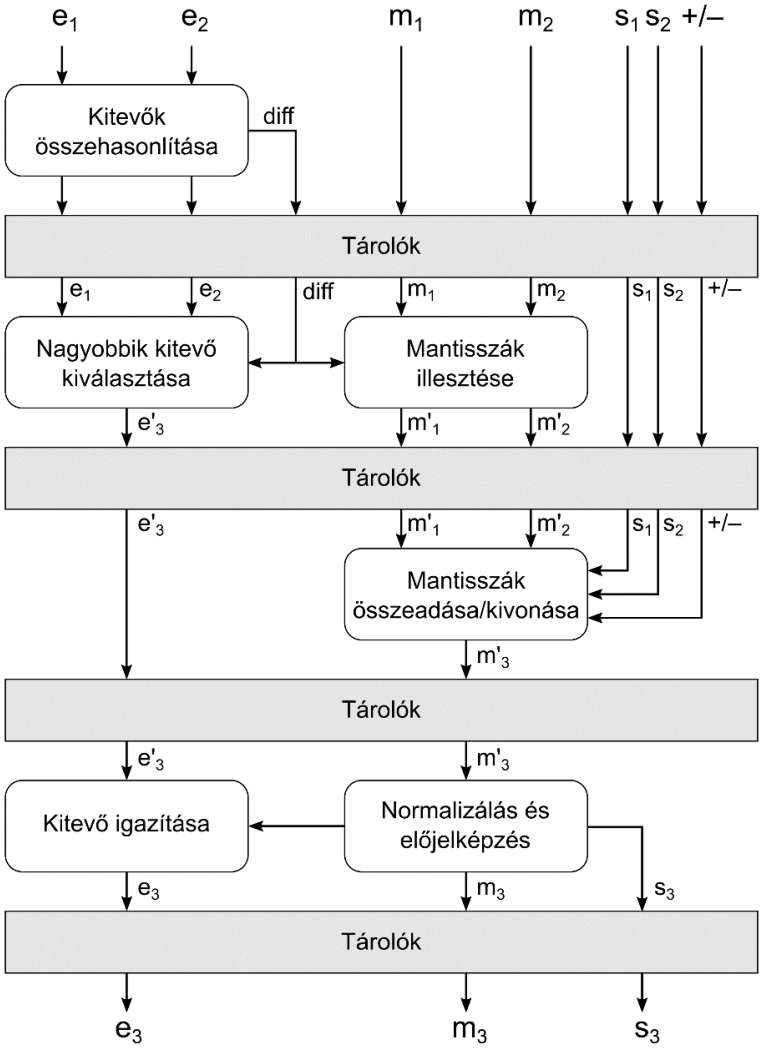
A műveletet elvégző csővezeték négy egymás utáni fázisból áll, ahogyan azt a 2.13. ábrán is nyomon követhetjük:

1) Kitevők összehasonlítása (különbségképzés). Vegyük észre, hogy a kivonásnál ugyanazt az eredményt kapjuk, ha közvetlenül az ábrázolt (eltolt) exponenst használjuk:

$$diff = k_1 - k_2 = (e_1 - offs) - (e_2 - offs) = e_1 - e_2 \tag{2.22}$$

[6] Forrás: https://en.wikipedia.org/wiki/IEEE_754 (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT



2.13. ábra – Lebegőpontos összeadást, ill. kivonást megvalósító aritmetikai csővezeték

2) A nagyobbik kitevő kiválasztása,

$$\begin{aligned} i &= \arg\min_{l\in\{1,2\}}(e_l) \\ j &= \arg\max_{l\in\{1,2\}}(e_l) \\ e'_3 &= e_j = \max_{l\in\{1,2\}}(e_l) \end{aligned} \tag{2.23}$$

és a mantisszák illesztése (a kisebbik kitevőhöz tartozó mantissza eltolása jobbra).

$$\begin{aligned} m'_i &= m_i \gg diff \\ m'_j &= m_j \end{aligned} \tag{2.24}$$

Ne tévesszük szem elől, hogy a szabvány szerinti formátumban csak a mantissza törtrészét ábrázoljuk. Emiatt az eltolás (és a későbbi összeadás/kivonás) előtt a rejtett egyest vissza kell állítani.

Copyright © 2021 / BME AUT

3) A mantisszák összeadása, ill. kivonása előjelhelyesen. (A műveletvégzés előtt a mantisszákat kettes komplemens alakra kell hozni az előjelbiteknek megfelelően.)

$$m_3' = (\pm m_1') \pm (\pm m_2') \tag{2.25}$$

4) Az eredményül kapott mantissza normalizálása, az előjel leválasztása, a kitevő igazítása a (2.21)-nek megfelelően.

Az összeadó/kivonó aritmetikai csővezeték elvi működésének megismerése után lássunk egy konkrét számpéldát az elvégzendő műveletek illusztrálása céljából!

Hogyan végzi el a lebegőpontos aritmetikai csővezeték a következő műveletet?

$$0x43700000 - 0x40B00000 = ?$$

Megoldás:

Először határozzuk meg az operandusok decimális alakját, és számítsuk ki az eredményt, hogy lássuk, mire kell jutnunk a bináris műveletek során.

1. operandus:

$$0x43700000 = 0100|0011|0111|0000\ldots_b = 0\ 10000110\ 1110000\ldots_b$$

$$s_1 = 0\ (+)$$

$$e_1 = 10000110_b = 128 + 4 + 2 = 134$$

$$k_1 = 134 - 127 = 7$$

$$f_1 = 0.111_b$$

$$m_1 = 1.111_b = 1.875$$

Decimális alakban:

$$x_1 = +1.875 \cdot 2^7 = 240$$

2. operandus:

$$0x40B00000 = 0100|0000|1011|0000\ldots_b = 0\ 10000001\ 0110000\ldots_b$$

$$s_2 = 0\ (+)$$

$$e_2 = 10000001_b = 128 + 1 = 129$$

$$k_2 = 129 - 127 = 2$$

$$f_2 = 0.011_b$$

Copyright © 2021 / BME AUT

$$m_2 = 1.011_b = 1.375$$

Decimális alakban:

$$x_2 = +1.375 \cdot 2^2 = 5.5$$

Az eredmény decimálisan kiszámítva:

$$240 - 5.5 \; = \; 234.5$$

Lássuk most az egyes csővezeték-fázisokban elvégzett részműveleteket!

1. Kitevők összehasonlítása: $diff = e_1 - e_2$

$$e_1 = 10000110_b$$

$$e_2 = 10000001_b$$

Képezzük az e_2 exponens ellentettjét (kettes komplemens):

$$-e_2 = 01111111_b$$

Végezzük el a kivonást:

```
  10000110    (e1)
+ 01111111    (-e2)
----------
  00000101  = +5d = diff
```

2a. Nagyobbik kitevő kiválasztása

A kitevők különbségének előjele alapján tudjuk kiválasztani a nagyobb kitevőt.

$$diff = e_1 - e_2 > 0 \quad \rightarrow \quad e_3' = e_1$$

2b. Mantisszák illesztése

Mivel e_2 volt a kisebb, m_2-t toljuk el jobbra a különbség (5) értékével. Figyeljünk a rejtett egyesre!

$$m_2 = 1.011_b \rightarrow \; m_2' = 0.00001011_b$$

$$m_1' = m_1 = 1.111_b$$

3. Mantisszák kivonása: $m_3' = m_1' - m_2'$

Képezzük m_2' kettes komplemensét a kivonás miatt (két egész bitet ábrázolunk az előjeles formátum miatt):

Copyright © 2021 / BME AUT

$$m_2' = 00.0000101100\ ..._b$$

$$-m_2' = 11.1111010100\ ..._b$$

Végezzük el a kivonást:

```
  01.1110000000...    (m1')
+ 11.1111010100...    (-m2')
------------------
  01.1101010100... = m3'
```

4. Mantissza normalizálás és előjelképzés, kitevő igazítása

Az eredményül kapott mantissza pozitív, így az előjelbit:

$$s_3 = 0$$

A mantissza az $[1,2)$ tartományban van, ezért nem szükséges további normalizálás (eltolás):

$$m_3 = m_3' = 1.1101010100_b$$

$$f_3 = 0.1101010100_b$$

Mivel nem kellett eltolni a mantisszát, ezért a végleges kitevő:

$$e_3 = e_3' = e_1 = 10000110_b = 134$$

$$k_3 = 134 - 127 = 7$$

Mindezekből már fel tudjuk írni az eredményt decimális és bináris (hexadecimális) alakban is.

Decimálisan:

$$x_3 = +\,1.110101010_b \cdot 2^7 = 1.110101010_b \ll 7 = 11101010.10_b$$

$$= 128 + 64 + 32 + 8 + 2 + 0.5 = \mathbf{234.5}$$

Binárisan és hexadecimálisan:

```
   0 100|0011|0 110|1010|1000|0000|0000|0000
0x     4    3     6    A    8    0    0    0
```

■

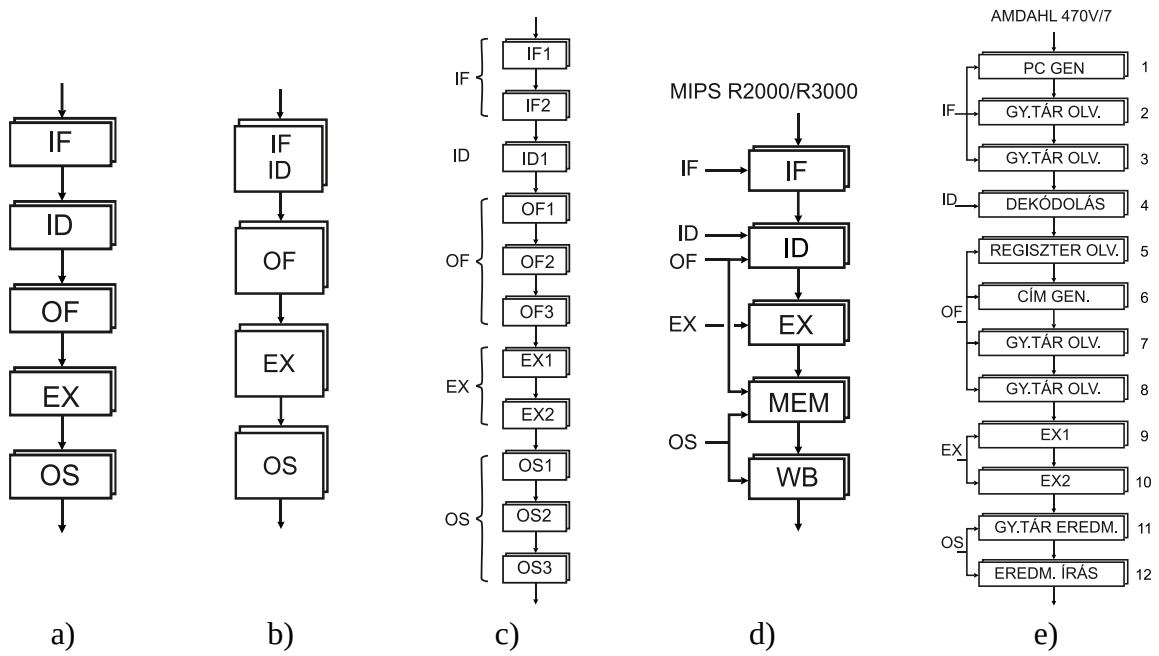
Copyright © 2021 / BME AUT

2.2.3 Utasítás-csővezeték

Egy mikroprocesszorban a gépi utasítások végrehajtása is fázisokra bontható, ez képezi a csővezetékes utasításvégrehajtás alapját. A végrehajtás tipikus fázisai:

1. IF – Instruction fetch (utasításbetöltés)
2. ID – Instruction decode (utasításdekódolás)
3. OF – Operand fetch (operandusok betöltése)
4. EX – Execution (végrehajtás)
5. OS – Operand store (eredmény tárolása)

Fontos megjegyeznünk, hogy ezek a részműveletek az utasításvégrehajtás logikai felosztásának felelnek meg. Egy adott architektúrában a ténylegesen megvalósított csővezeték-fázisok ettől eltérhetnek, ugyanis a fázisokra bontás fontos szempontja, hogy az egyes részműveletek végrehajtási ideje közel azonos legyen. Valós processzor mikroarchitektúrák tervezésénél előfordul, hogy a fenti tipikus fázisok közül némelyeket összevonnak vagy további részekre osztanak az időkiegyenlítés (vagy nagyobb órajelfrekvencia alkalmazása) céljából. A 2.14. ábrán erre látunk példákat. A 2.14a ábra a fenti tipikus felbontást mutatja. A 2.14b ábrán azt az esetet láthatjuk, amikor az utasításfelhozás és -dekódolás egy fázisba vonható össze, ami pl. egy utasítás-gyorsítótárral ellátott RISC processzor esetén elképzelhető. A 2.14c ábra egy CISC processzor 11-fázisú csővezetékét illusztrálja, amelyben a legtöbb részművelet több fázisra van bontva, elsősorban az összetett utasítások (IF és EX), valamint a változatos operandus címzési módok (OF, OS) miatt.



2.14. ábra – Különböző utasítás-csővezeték megvalósítások. a) Az utasításvégrehajtás tipikus fázisai. b) Az utasításbetöltés és -dekódolás összevonása egy fázisba (hipotetikus RISC processzor, utasítás-gyorsítótárral). c) 11-fázisú csővezeték, ahol a legtöbb részművelet több fázist igényel (hipotetikus CISC processzor). d) MIPS R2000/R3000 RISC processzorok csővezetéke. e) Amdahl 470V/7 CISC számítógép csővezetéke.

Copyright © 2021 / BME AUT

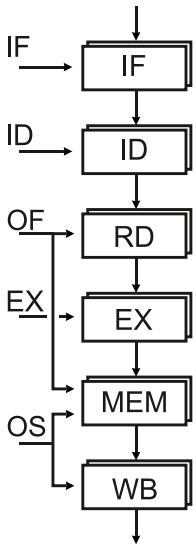
A 2.14d és 2.14e ábrákon két valóságos utasítás-csővezetéket láthatunk. A MIPS R2000/R3000 processzorok csővezeték megvalósítását mutatja a 2.14d ábra (MIPS = Microprocessor without Interlocked Pipeline Stages). Ezek a modellek a klasszikus RISC filozófiát megvalósító első processzorok közé tartoztak az 1980-as években. Load/store architektúrát valósítanak meg, azaz az aritmetikai és logikai műveletekhez csak a belső regiszterekből kell olvasni az operandusokat, így ez egy fázisba összevonható az utasítás dekódolásával. A memóriával való kommunikáció csak regiszterbe töltés és regiszterből tárolás lehet, amelyet az EX utáni MEM fázisban hajt végre a processzor. Betöltés (memóriaolvasás) esetén nem az ALU művelet eredménye íródik be a WB (write-back) fázisban az aktuális regiszterbe, hanem a memóriából érkező adat. Ezek a processzorok elkülönített utasítás és adat gyorsítótárral rendelkeznek, minden ciklusban mindkettőhöz hozzáférés történhet. Ha egyesített lenne a gyorsítótár, akkor kétkapusnak kellene lennie, így viszont nincs szükség többkapus megoldásra. A regiszter-tömbnek azonban egyidejűleg két olvasási és egy írási hozzáférést is biztosítania kell.

A 2.14e ábrán az Amdahl 470V/7 CISC elvű, tehát nem load/store architektúrájú mainframe számítógép utasítás csővezetéke látható (1970-es évek vége). Az első három fázis az utasításbetöltést valósítja meg, a negyedik fázis az utasításdekódolást. Mivel bonyolult címzési üzemmódokat használ a processzor, az OF fázis négy ciklust igényel (5,6,7,8). Az 5. fázisban két regiszter olvasása valósul meg, melyek címregiszterekként részt vesznek a címképzésben. A hatodik fázis a memória operandus címét számítja ki. A 7. fázis elindítja az adatgyorsítótár hozzáférését. A 8. fázisban fejeződik be a gyorsítótár hozzáférés, s ekkor történik meg a másik operandus olvasása a regisztertömbből. A 9. és 10. fázis a végrehajtás. A 11. és 12. fázisban az eredmény tárolódik a gyorsítótárban, illetve a specifikált célregiszterben. Ennek a csővezetéknek minden ciklusban két konkurens regiszter olvasást (5. fázis) és egy regiszter írást (12. fázis) kell támogatnia, továbbá négy konkurens gyorsítótár olvasást is a 2., 3., 7. és 8. fázisban. Az ehhez szükséges memória alrendszer tehát sokkal bonyolultabb, mint a MIPS R2000/R3000 processzoroké.

2.2.3.1 Egy tipikus csővezeték működésének vizsgálata

A következőkben egy tipikus felépítésű, hipotetikus RISC csővezeték (a továbbiakban TYP) példáján keresztül mutatjuk be a csővezetékes utasításvégrehajtás sajátosságait. Egy load/store architektúrájú, 6-fázisú pipeline megvalósítást feltételezünk, amely leginkább a MIPS R2000/R3000 csővezetékére hasonlít, annyi különbséggel, hogy az utasításdekódolást és a regiszter operandus olvasást külön fázisnak tekintjük, a 2.14a ábrán látható egyszerű logikai felosztáshoz hasonlóan. Ugyanakkor a load/store jelleg miatt itt is megtalálható az elkülönített MEM fázis a végrehajtási és regiszter-visszaírási fázisok között. A TYP csővezeték fázisai tehát:

1. IF – Instruction fetch (utasításbetöltés)
2. ID – Instruction decode (utasításdekódolás)
3. RD – Register read (regiszterolvasás)



2.15. ábra – TYP csővezeték

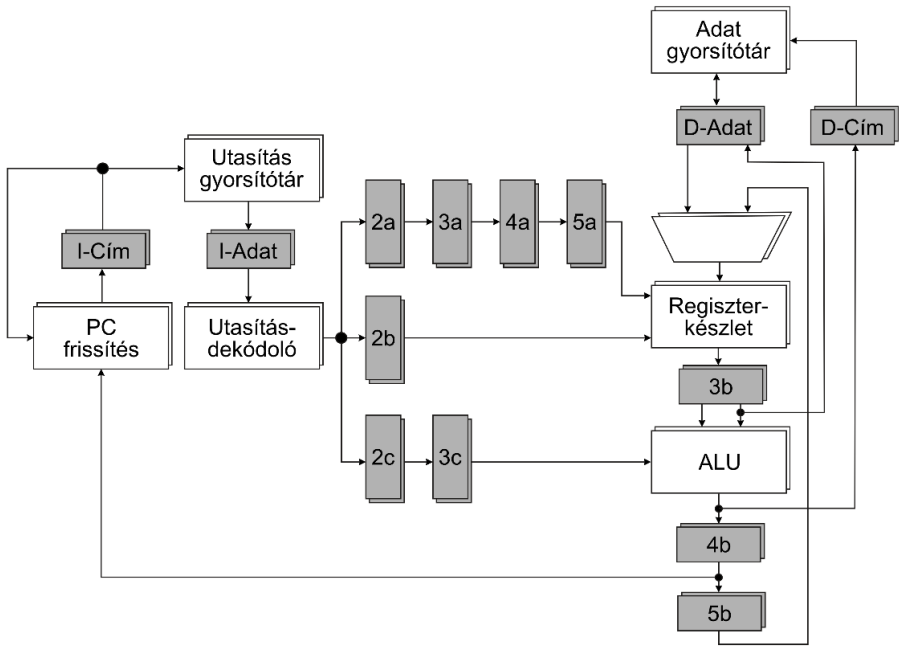
Copyright © 2021 / BME AUT

4. EX – Execution (végrehajtás)
5. MEM – Memory (memóriaművelet)
6. WB – Write back (regiszter-visszaírás)

Működésének részletesebb vázlata a 2.16. ábrán vehető szemügyre. Elkülönített utasítás- és adatgyorsítótárat feltételezünk (Harvard architektúra; RISC processzoroknál általánosan elterjedt megoldás, de a mai CISC processzorok is szinte kivétel nélkül ezt alkalmazzák az első szintű gyorsítótárak esetén). Az ábra bal oldalán a vezérlőegység blokkjait láthatjuk, ezek végzik a programszámláló (PC, program counter) frissítését, vagyis a következő beolvasandó utasítás címének (I-cím) előállítását, valamint a felhozott utasítás dekódolását. Az ábra jobb oldalán láthatjuk a műveletvégzéshez szükséges regiszterkészletet és a műveletvégző egységet (ALU). A szürke színű blokkok jelölik a csővezeték-szervezéshez elengedhetetlen átmeneti tárolókat. Ezek között vannak adat- és címregiszterek, de láthatunk olyanokat is, amelyeknek szerepe csupán az adatok, illetve vezérlőjelek késleltetése annak érdekében, hogy a megfelelő ciklusban érjenek oda a soron következő feldolgozó blokkhoz. Kísérjük figyelemmel két konkrét utasítástípus példáján keresztül, hogy mi történik az egyes fázisokban!

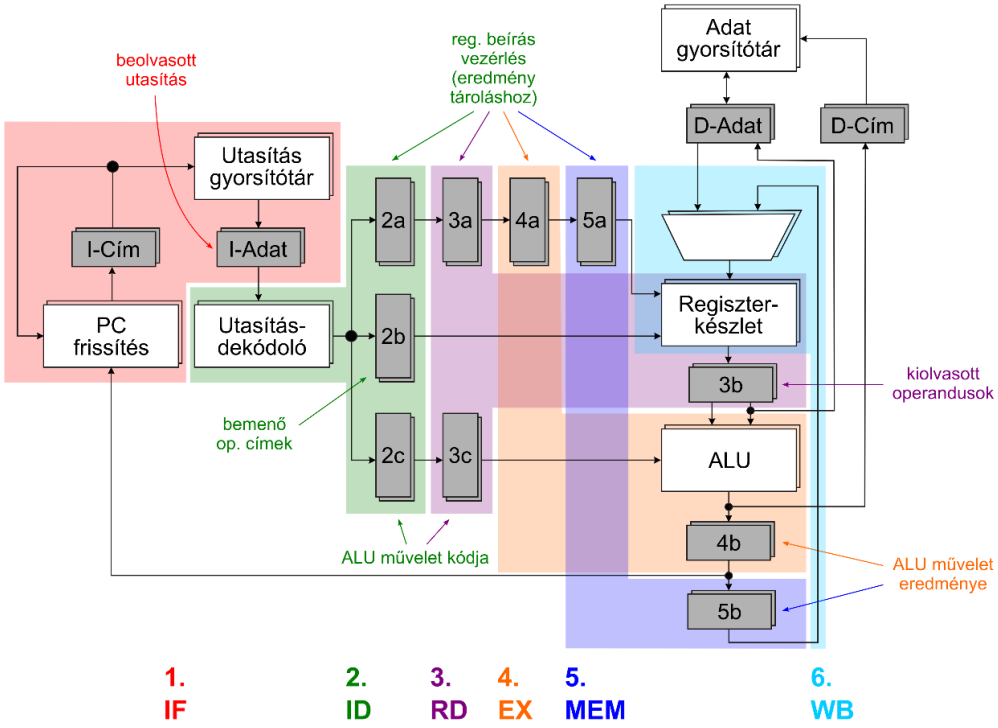
ALU utasítás végrehajtása

Az első vizsgált utasítástípus egy egyszerű adatfeldolgozó (ALU) művelet. Mivel load/store architektúrájú processzorunk van, a forrás- és a céloperandusok is regiszterekben tárolódnak. A feldolgozás szekvenciáját a 2.17. ábra szemlélteti.



2.16. ábra – A TYP csővezeték fizikai megvalósításának blokkvázlata

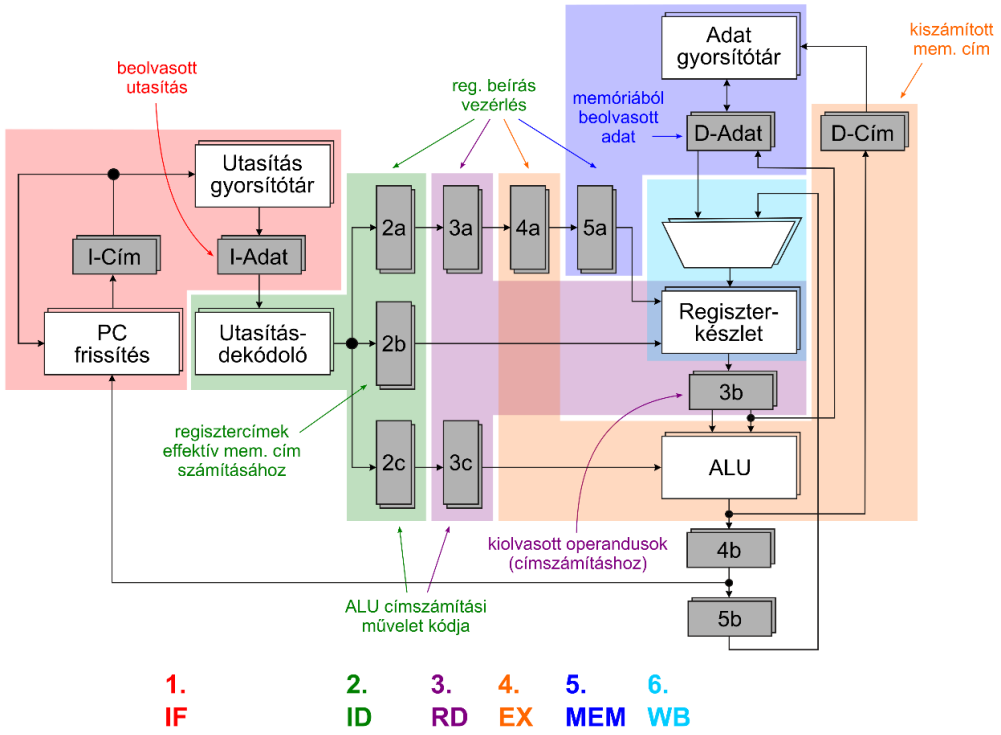
Copyright © 2021 / BME AUT



2.17. ábra – ALU művelet végrehajtása a TYP csővezetékben

1. IF: A friss programszámláló-tartalom (I-cím) alapján megcímzett utasítás cache-ből felhozott utasításkódot az [I-adat] regiszter tartalmazza.
2. ID: Az utasításdekóder kimenete három átmeneti tárolóba kerül bele. [2c] tárolja az elvégzendő ALU művelet kódját, [2b] a forrásoperandus regiszterek címeit, [2a] pedig a regiszterbeírás-vezérlést az eredmény eltárolásához.
3. RD: A [2b] alapján megcímzett regisztertárból kiolvasott operandusok [3b]-be kerülnek bele. A [3a] és [3c] tárolókba pedig egyszerűen továbblép a regiszterbeírás-vezérlés és az elvégzendő ALU művelet kódja.
4. EX: A műveleti kód és a [3b]-ben lévő operandusok felhasználásával az ALU elvégzi a műveletet, amelynek eredménye a [4b] tárolóba kerül. A regiszterbeírás-vezérlés a [4a] tárolóba lép tovább.
5. MEM: Mivel ALU művelet esetén nincs memóriahozzáférés, ezért ebben a fázisban csak az eredmény és annak eltárolásához szükséges vezérlőjelek továbbléptetése történik az [5b] és az [5a] tárolókba.
6. WB: Megtörténik az eredmény eltárolása a regiszterkészletbe az [5a] és [5b] tárolók tartalma alapján.

Copyright © 2021 / BME AUT



2.18. ábra – Memóriaolvasás végrehajtása a TYP csővezetékben

Memóriaolvasás végrehajtása

A második példánk egy memóriaolvasási (load) művelet. A címképzés miatt itt is szükség lesz az ALU egységre. A feldolgozás szekvenciáját a 2.18. ábrán követhetjük nyomon.

1. IF: Mivel a dekódolás során derül ki az utasítás típusa, ez a fázis megegyezik az ALU műveletnél látottakkal. Az [I-adat] tartalmazza a beolvasott utasításkódot.
2. ID: Itt derül ki, hogy memóriaolvasásról van szó. Az utasításdekóder kimenete három átmeneti tárolóba kerül bele. [2a] tárolja a regiszterbeírás-vezérlést a beolvasandó adat eltárolásához, [2b] a dekódolt utasításban lévő, az effektív cím kiszámítására szolgáló regiszter címeket, [2c] pedig az ALU-val elvégzendő címszámítás műveleti kódját.
3. RD: A [3b] tároló tartalmazza a címszámításhoz használandó regiszterekből kiolvasott operandusokat. A [3a] és [3c] tárolókba egyszerűen továbblép a regiszterbeírás-vezérlés és az elvégzendő címszámítás ALU művelet kódja.
4. EX: Az ALU által kiszámított effektív cím a [D-cím] regiszterbe íródik be. A regiszterbeírás-vezérlés a [4a] tárolóba lép tovább.
5. MEM: Az adat cache-ből beolvasott adat a [D-adat] regiszterbe kerül be. A regiszterbeírás-vezérlés az [5a] tárolóba lép tovább.
6. WB: Megtörténik a beolvasott adat eltárolása a regiszterkészletbe az [5a] és a [D-adat] tárolók tartalma alapján.

Copyright © 2021 / BME AUT

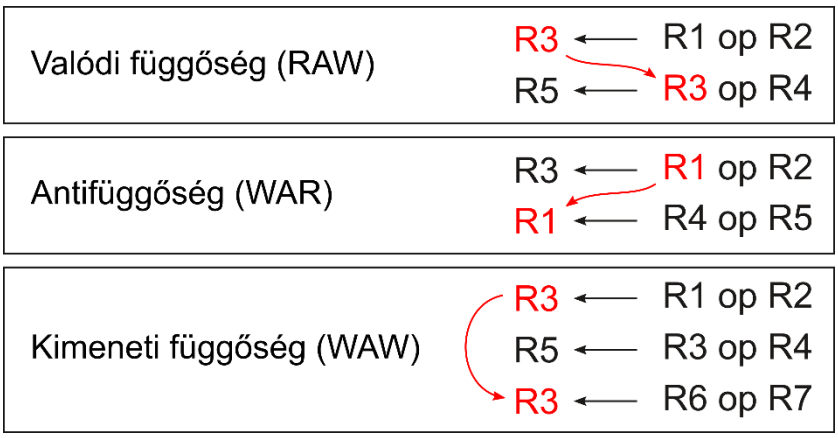
2.2.4 Függőségek és hazárdok a csővezetékben

A csővezetékes végrehajtás nagy előnye, hogy időbeli átlapolással egyszerre több utasítás feldolgozása folyik, ami nagyban növeli a hatékonyságot. Ugyanakkor a pipeline szervezés olyan új problémákat is felvet, amelyeket az utasításkészlet-architektúra (ISA) absztrakciós szintjén gondolkodó programozó (ill. fordítóprogram) elől a mikroarchitektúrának el kell rejtenie. Ezek a problémák abból erednek, hogy a csővezetékben konkurens módon feldolgozás alatt álló utasítások egymástól nem függetlenek, ezért ún. utasítás egymásra hatás léphet fel.

A helyes működés elégséges, de nem szükséges feltétele a teljesen sorrendi, másnéven **totálisan szekvenciális végrehajtás**. Ez azt jelenti, hogy egy utasítás végrehajtásának minden fázisa befejeződik a következő utasítás elkezdése előtt. Az ISA szintjén tevékenykedő assembly programozó ezzel a feltételezéssel él, de a pipeline szervezés a mikroarchitektúrában (szándékosan) megsérti a teljes sorrendiséget az időbeli átlapolással, és ezzel felszínre kerülnek az utasítások függőségeiből adódó problémák.

Előfordulhat ugyanis, hogy egy utasítás végrehajtásának aktuális fázisa olyan erőforrást vagy adatot igényelne (pl. regisztertartalom, gyorsítótár-tartalom), amely még nem áll rendelkezésre, mert egy olyan előző utasítás állítja elő, aminek a végrehajtása szintén folyamatban van még. Az ilyet fel kell ismerni, és a legkisebb teljesítménycsökkenést lehetővé tévő megoldással ki kell küszöbölni.

A függőségek megsértését hazárdnak nevezzük. A hazárd hibás működést okoz, amit mindenképpen el kell kerülni. Ha az utasítások közötti függőségek nem sérülnek, akkor a totálisan szekvenciális végrehajtás hiánya esetén is helyes marad a program működése. Az ISA absztrakciós szintjén a függőségeknek két alapvető típusát különböztetjük meg, ezek az *adatfüggőségek* és a *vezérlési függőségek*. Adatfüggőség akkor lép fel, ha különböző utasítások ugyanahhoz a regiszterhez vagy memóriarekeszhez férnek hozzá. Vezérlési függőségről pedig olyan ugró utasítások esetén beszélhetünk, amelyek feltételhez kötöttek, vagy amelyeknél az ugrási célcím egy művelet eredménye.



2.19. ábra – Adatfüggőségek típusai

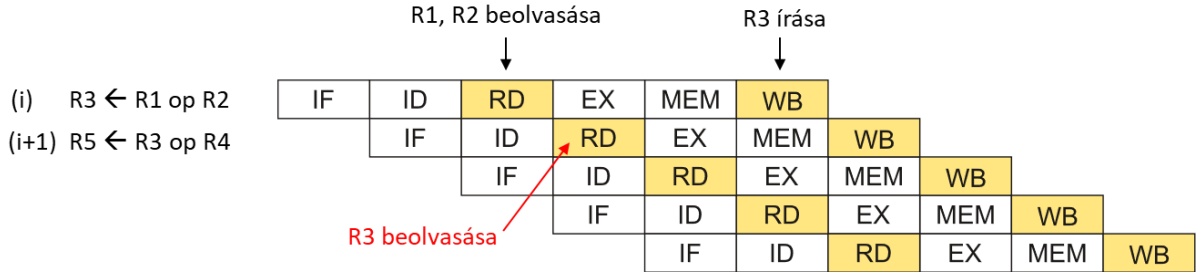
Copyright © 2021 / BME AUT

2.2.4.1 Adatfüggőség

Az adatfüggőségek között megkülönböztetünk RAW (read after write), WAR (write after read) és WAW (write after write) típusú függőségeket. Előfordulhatnak regiszterreferens és memóriareferens utasítások között is. A RAW típusút valódi függőségnek, a WAR típusút antifüggőségnek, a WAW típusút pedig kimeneti függőségnek is nevezzük. Mindháromra láthatunk egy-egy példát a 2.19. ábrán, regiszterreferens utasítások esetén.

Valódi függőség (RAW)

Read after write típusú függőségről akkor beszélünk, ha egy utasítás olyan adatot olvas, amelyet egy őt megelőző utasítás ír. Ha a két utasítás elég szorosan követi egymást a programban, csővezetékes végrehajtás esetén előfordulhat, hogy a kimeneti adat írása még nem történt meg, amikor a következő utasítás már olvasná azt. A RAW függőség által potenciálisan előidézett hazárdjelenséget a 2.20. ábra szemlélteti a TYP csővezetékben regiszterreferens utasítások esetén. A regiszter-visszaírás (WB) két fázis közbeiktatása után követi a regiszterolvasási (RD) fázist. Mikor az (i+1) utasítás a kérdéses regisztert olvasná, az (i) utasítás által oda beírandó adat még csak akkor keletkezik az EX fázisban.



2.20. ábra – RAW típusú adatfüggőség hatása a TYP csővezetékben

Antifüggőség (WAR)

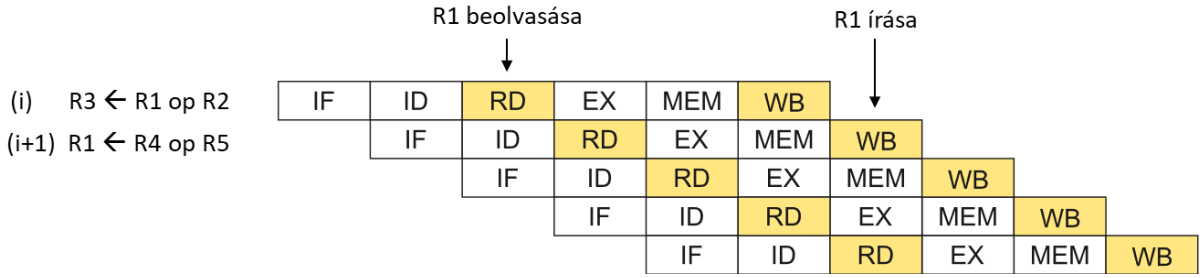
A write after read típusú (anti)függőség azt jelenti, hogy egy olvasott adatot egy következő utasítás felülír. Ez regiszterírás és -olvasás esetén látszólag semmi gondot nem okozhat, hiszen egy utasításnál az írási fázis később következik be, mint az olvasás, és ez időben egymást követő utasítások esetén nem fordulhat meg (ld. 2.21. ábra). A TYP csővezetékben ezért a WAR függőség nem okoz hazárdot. Később látni fogjuk, hogy szuperskalár csővezeték esetén nem ilyen egyszerű a helyzet, mert ott már lehetőség van az utasítások programsorrendtől eltérő végrehajtására, ezért ott a WAR függőségekre is oda kell figyelni.

Kimeneti függőség (WAW)

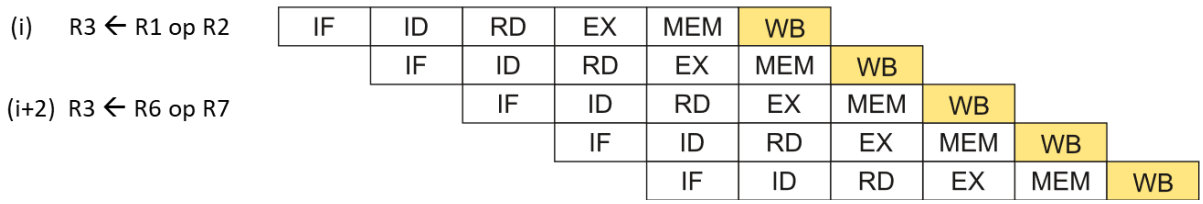
Write after write függőség esetén két egymást követő utasítás ugyanazt a regisztert írja. Hasonlóan a WAR függőségnél látottakhoz, a TYP csővezetékben ez sem okoz hazárdot, mivel az egyes utasítások szigorúan programsorrendben lépnek be a WB fázisba (ld. 2.22. ábra). A

Copyright © 2021 / BME AUT

szuperskalár csővezetékeknél viszont látni fogjuk, hogy a sorrenden kívüli végrehajtás miatt itt is figyelni kell a hazárdok kiküszöbölésére.



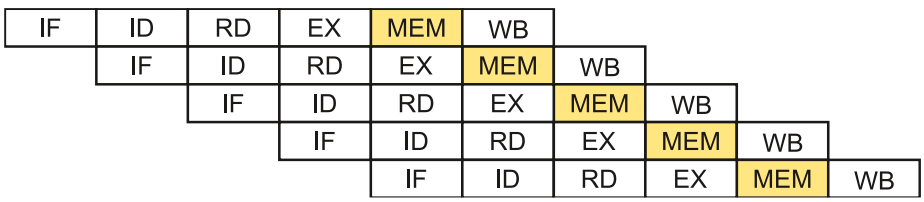
2.21. ábra – WAR típusú adatfüggőség hatása a TYP csővezetékben



2.22. ábra – WAW típusú adatfüggőség hatása a TYP csővezetékben

Memória-adatfüggőségek

Az előző példákban az utasítások között regiszter-adatfüggőség állt fenn. Ugyanakkor a felsorolt RAW, WAR, WAW típusú függőségek memóriában tárolt adatok kapcsán is felmerülhetnek. Az általunk vizsgált TYP csővezetéknél load/store architektúrát feltételeztünk, ahol memória-adatfüggőség csak a betöltő és tároló utasítások között léphet fel. Mivel a hipotetikus processzorunk osztott utasítás- és adatgyorsítótárral rendelkezik, adatokat érintő memóriahozzáférés csak a MEM fázisban történik. Ezért minden adathozzáférés szigorúan programsorrendben hajtódik végre (ld. 2.23. ábra), így a betöltő/tároló utasítások kapcsán memória típusú adathazárd nem fordulhat elő. A szuperskalár processzoroknál látni fogjuk, hogy a sorrenden kívüli végrehajtás miatt a memória-adatfüggőségek helyes kezelésére külön alrendszereket alkalmaznak a mikroarchitektúrákban.



2.23. ábra – Memória-adatfüggőség hatása a TYP csővezetékben

Copyright © 2021 / BME AUT

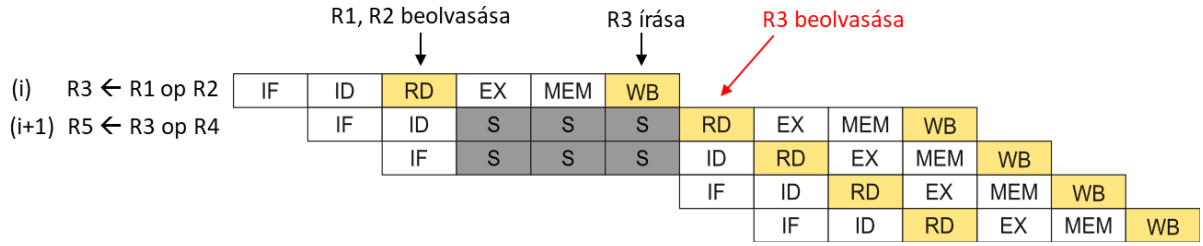
2.2.4.2 A RAW típusú hazárdok kiküszöbölése

Láthattuk, hogy az egyszerű, skalár csővezetékben a RAW típusú adatfüggőségek okoznak problémát. A következőkben két lehetséges módot mutatunk be a RAW típusú hazárdok kiküszöbölésére.

Feldolgozás felfüggesztése

A legegyszerűbb megoldás a függő utasítás végrehajtásának felfüggesztése (stall) addig, amíg a szükséges adat rendelkezésre nem áll a kérdéses regiszterben. Maga az ötlet nem csak regisztertartalomra vonatkozó adatfüggőségek esetén használatos, hanem bármilyen esetben, amikor egy adott végrehajtási fázishoz szükséges erőforrás nem áll még rendelkezésre. Sorrendi végrehajtás esetén a várakozás szükségképpen „buborékokat” képez a csővezetékben, ezért csökkenti a hatékonyságot.

Erre láthatunk egy példát a 2.24. ábrán. Az (i) utasítás írja az R3 regisztert, amelyet az (i+1) utasítás olvasna. De mikor az (i) utasítás még csak éppen befejezte a neki szükséges regiszterek olvasását az RD fázisban, hogy azokat feldolgozásra átadja az EX fázisnak, az (i+1) utasítás már az EX fázis által létrehozandó és az R3 regiszterbe visszaírandó eredményt olvasná az R3 regiszterből. Mivel ez az adat még nem áll rendelkezésre, a processzor felfüggeszti az (i+1) és az azt követő utasítások továbbléptetését a csővezetékben mindaddig, amíg az R3 regiszterben az (i) utasítás által generált adat rendelkezésre nem áll. Ez az (i) utasítás WB fázisának végéig így lesz. Ekkor az eredmény beíródik az R3 regiszterbe, és megszüntethető az (i+1) és az azt követő utasítások felfüggesztése.



2.24. ábra – RAW adathazárd kiküszöbölése a feldolgozás felfüggesztésével

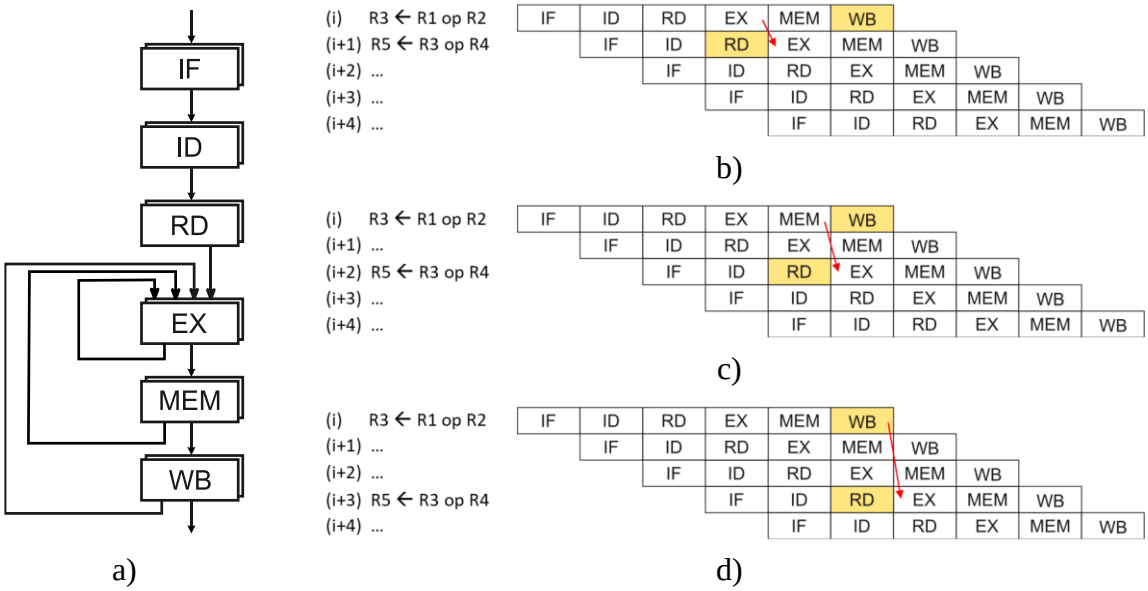
A csővezeték felfüggesztésének hardveres megvalósítását pipeline interlock-nak nevezik. Regiszterek esetén elterjedt megoldás a pontozótábla (scoreboard) használata, amely minden regisztert plusz egy bittel egészít ki (presence bit). Amikor egy utasítás az RD fázisba kerül, a hardver megvizsgálja a bementi regiszterek presence bitjének értékét. Ha a bemenetek rendelkezésre állnak, akkor betölti a tartalmukat, a kimeneti regiszter presence bitjét törli, és az utasítás továbblép az EX fázisba. Ha egy utasítás az RD fázisban azzal szembesül, hogy bármelyik bemeneti regiszterének presence bitje nincs beállítva, mindaddig az RD fázisban várakozik, amíg ez az állapot fennáll. Egy utasítás végrehajtásának végén, a WB fázisban történik a kimeneti regiszter presence bitjének (újra) beállítása. Ezt érzékelve a felfüggesztett utasítás végrehajtása folytatódhat.

Copyright © 2021 / BME AUT

Előrecsatolás

A RAW típusú hazárdok az ún. előrecsatolással (operand forwarding, register bypass) küszöbölhetők ki a hatékonyság csökkenése nélkül. Ennek lényege, hogy az EX fázis bemenetére visszacsatolják az EX, MEM és WB fázisok kimenetét[7]. Így a visszaírás megtörténte előtt a kiszámolt eredmény már rendelkezésre áll egy következő utasítás EX fázisa számára.

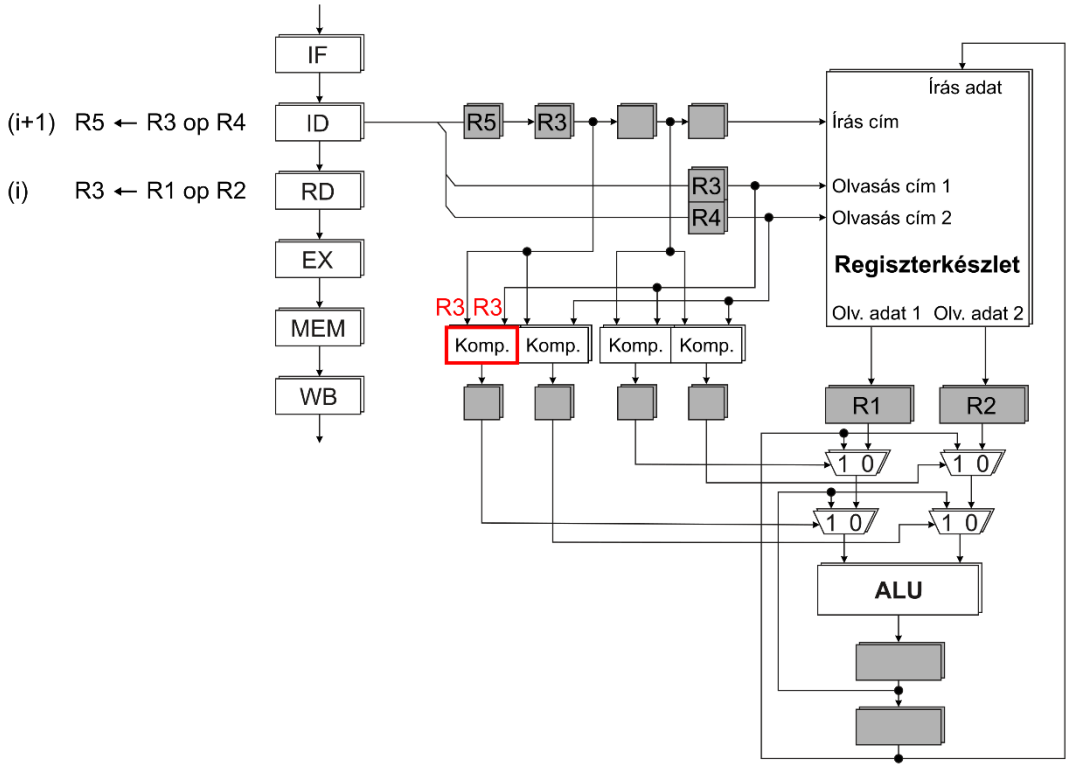
A 2.25. ábra az előrecsatolás alkalmazását szemlélteti a TYP csővezetékben. A visszacsatoló ágakat 2.25a ábrán láthatjuk, az egyes ágak működését pedig a 2.25b–d ábrákon figyelhetjük meg. Ha a függő utasítások közvetlenül követik egymást, akkor az (i+1) utasítás nem olvassa az (i) utasítás kimeneti regiszterét, hanem ehelyett az EX fázis kimenetéről visszacsatolt adattal hajtja végre a szükséges műveletet, hiszen az EX kimenetén már rendelkezésre áll az az adat, amit az R3 regiszterből olvasott volna. Ha az R3 regisztert olvasó utasítás nem közvetlenül, hanem egy utasítás közbeiktatásával követné az R3 írását végző utasítást, akkor a MEM fázis kimenetéről visszacsatolt adatott használja. Végül két utasítás közbeiktatása esetén a WB kimenetéről visszacsatolt adat használandó. Ha három vagy több utasítás iktatódik be a regisztert író, majd olvasó utasítások közé, akkor nem lép fel hazárd, nincs szükség előrecsatolásra.



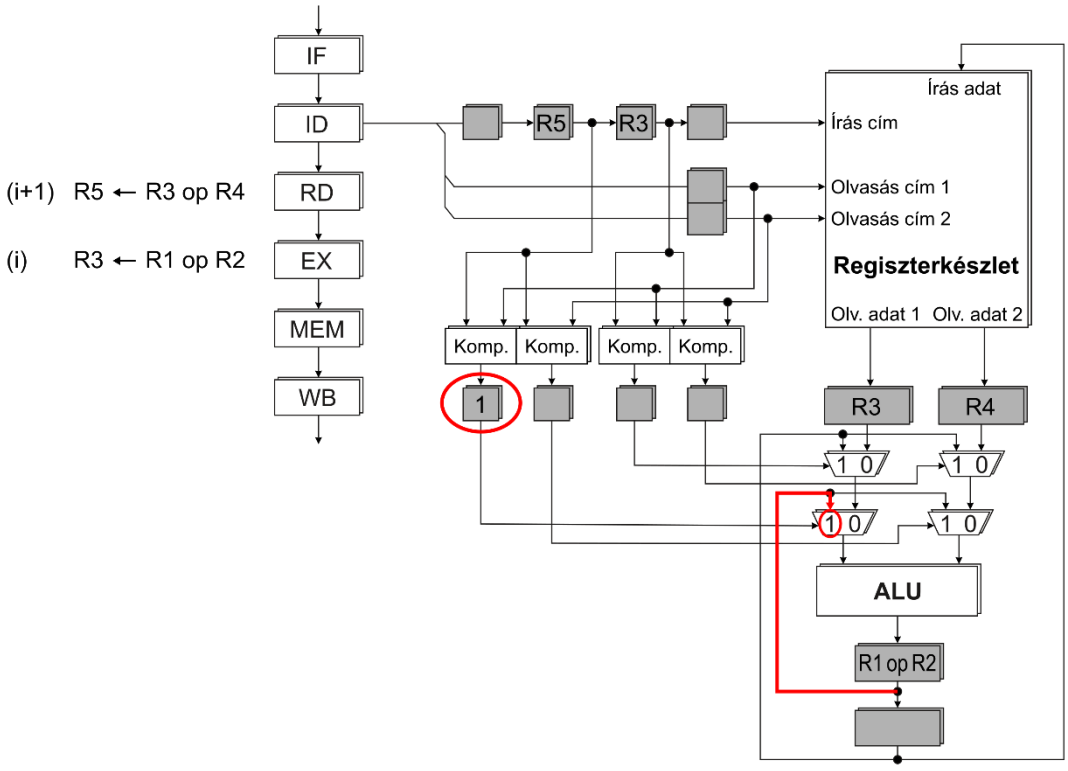
2.25. ábra – RAW adathazárd kiküszöbölése operandus előrecsatolással. a) Az előrecsatolás elve. b) Előrecsatolás az EX fázis kimenetéről. c) Előrecsatolás a MEM fázis kimenetéről. d) Előrecsatolás a WB fázis kimenetéről.

[7] Ne tévesszen meg minket az „előrecsatolás” és a „visszacsatolás” kifejezések egyidejű használata. Logikailag előrecsatoljuk a már kiszámolt, de még nem visszaírt eredményt egy következő utasítás EX fázisának bementére. Fizikailag ez viszont az EX, MEM és WB fázisok kimenetéről történő visszacsatolással valósítható meg, hiszen ugyanazok a funkcionális egységek hajtják végre az egyes utasítások ugyanazon fázisait.

Copyright © 2021 / BME AUT



a)



b)

2.26. ábra – Az előrecsatolás folyamata a TYP csővezetékben

Copyright © 2021 / BME AUT

Az előrecsatolás fizikai megvalósítását a 2.26. ábra szemlélteti a TYP csővezetékben. Az ábrán az egyszerűség kedvéért csak az EX és a MEM fázisok kimenetéről történő visszacsatolások szerepelnek. A RAW hazárdot komparátorok detektálják, melyek egymást követő utasítások regiszterhivatkozásait hasonlítják össze. Ha az (i+1) utasítás dekódolása megtörtént, és az ezt követő RD fázisban két regiszter operandus (a 2.26a ábrán R3 és R4) olvasására készül, akkor a baloldali két komparátor ellenőrzést végez az (i+1) és az (i) utasítás között a lehetséges RAW függőségre. Ez a két komparátor az (i+1) utasítás két forrásregiszterének hivatkozását hasonlítja össze az EX fázis előtt álló (i) utasítás célregiszterének hivatkozásával. Egyidejűleg a másik két komparátor az (i+1) utasítás és a MEM fázis előtt lévő (i–1) utasítás közötti RAW függőséget ellenőrzi. A négy komparátor kimenete, mint vezérlő jel szolgál a következő ciklusban (2.26b ábra) a megfelelő előrecsatoló út kiválasztására, ha függőség lépett fel.

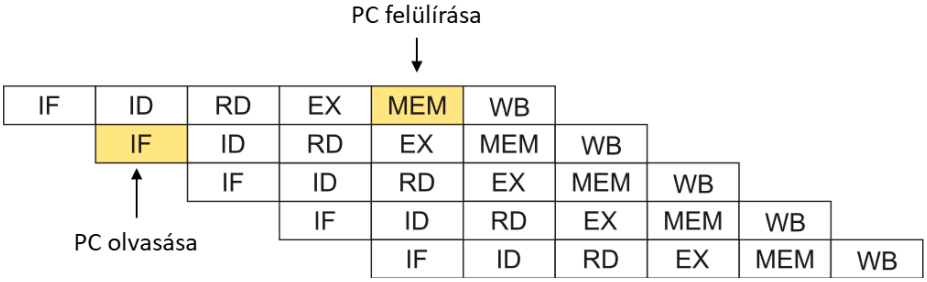
2.2.4.3 Vezérlési függőség

Vezérlési függőséget a vezérlésátadó utasítások okozhatnak, legyenek azok akár feltételes, akár feltétel nélküli ugró utasítások. Feltételes ugró utasítás esetén a feltétel határozza meg, hogy a sorrendben következő, vagy egy másik utasítással kell-e folytatni a program végrehajtását. Alapesetben az utasítások betöltése a gépi ciklusonként megnövelt programszámláló értéknek megfelelően történik. Feltételes ugró utasítás esetén a szekvenciálisan következő utasítás betöltése akkor helyes, ha a feltétel nem teljesül. Ellenkező esetben viszont egy másik címről kell felhozni a következő utasítást. A nehézséget az adja, hogy a kétértelműség mindaddig fennáll, amíg a feltétel ki nem értékelődik.

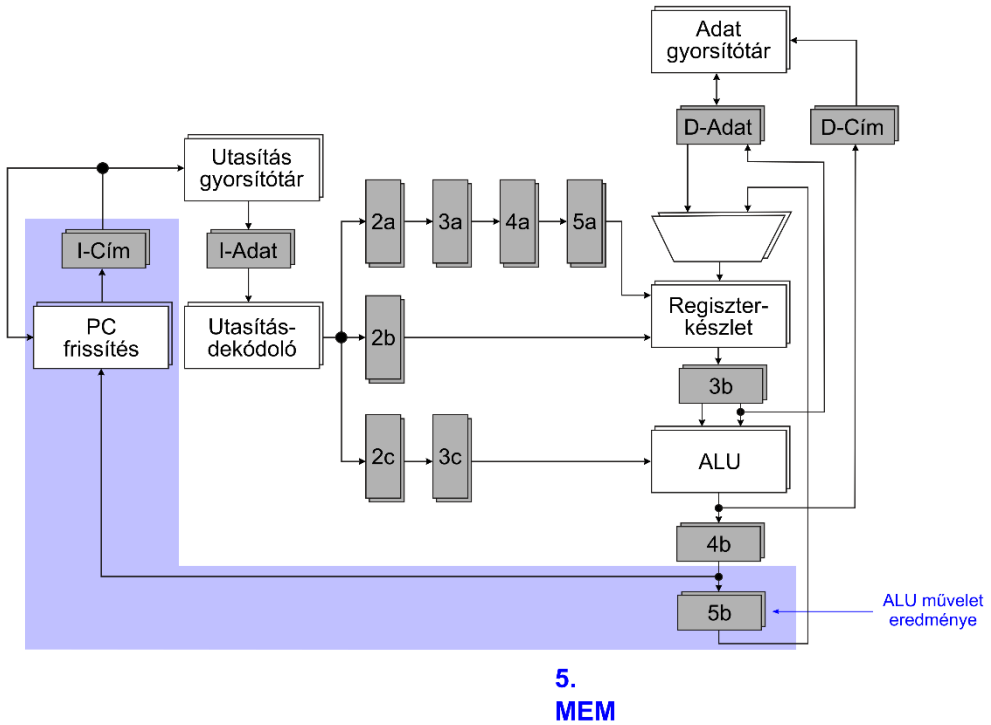
Feltétel nélküli ugró utasítások esetén is fennállhat a vezérlési függőség, ha az ugrás célcíme nem egy direkt cím, hanem valamilyen címszámítást igényel. Ekkor ugyanis az utasítás betöltésének befejeződésekor még nem áll rendelkezésre az ugrási cím, amivel felül kellene írni az egyszerűen inkrementált címet az utasításszámláló regiszterben. Ugyanis ez majd csak az egyszerűbb-összetettebb címaritmetikai műveletek eredményeként fog előállni. Emiatt a címszámítást igénylő utasításokat is „feltételesnek” nevezhetjük, hiszen az ugrási cím értéke regiszterek értékétől függ. Ez esetben nem az ugrás ténye, hanem az ugrás célcíme feltételes, azaz dinamikusan a program végrehajtása során alakul ki.

A vezérlési függőség RAW adatfüggőségnek is tekinthető a programszámláló (PC), mint regiszter vonatkozásában, ahogy ezt az 2.27. ábra is szemlélteti. A feltételes ugrás írja a PC-t, az utasításlehívás viszont olvassa. A feltételes ugró utasítás az ugráscímmel módosítja a PC-t, ha a feltétel teljesül, egyébként a PC tartalma a sorrendben következő utasítás címére frissül. A TYP csővezetékben a PC írása az ugráscímmel a MEM fázisban történik (ld. 2.28. ábra), ugyanakkor a PC olvasására már az IF fázisban sor kerül. Tehát RAW hazárd léphet fel a PC vonatkozásában feltételes ugró utasítások vagy címszámítást igénylő feltétel nélküli ugró utasítások esetén.

Copyright © 2021 / BME AUT



2.27. ábra – A vezérlési függőség RAW adatfüggőségként is tekinthető a PC regiszter vonatkozásában



2.28. ábra – Feltételes vagy címszámítást igénylő ugrás esetén a PC frissítése a MEM fázisban történik

2.2.4.4 Vezérlési hazárdok kiküszöbölése

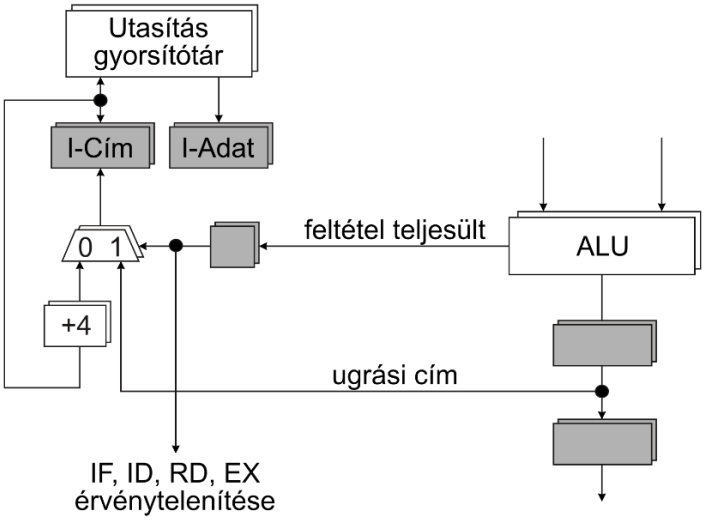
Az ugró utasítások következtében fellépő vezérlési hazárdok kiküszöbölésére két alapvető eljárást is alkalmazhatunk, melyeknek működését a 2.29. ábrán látható blokkvázlat szemlélteti. Alapesetben az IF fázis hozzáfér az utasítás gyorsítótárhoz, betölti a soron következő utasítást, és egyidejűleg inkrementálja a programszámlálót, felkészülve a következő utasítás betöltésére.

A kétféle megoldás onnan kezdve tér el, amikor egy utasításról a dekódolási fázisban kiderül, hogy ugró utasítás. Az egyik megoldásnál az IF fázist felfüggesztjük mindaddig, amíg az ugró utasítás át nem halad az EX fázison. Ekkorra kiértékelődik az ugrás feltétele és célcíme. Ha az ugrás feltétele teljesült (vagy ha feltétel nélküli volt), akkor az ezt követő MEM fázisban az

Copyright © 2021 / BME AUT

ugrási cím betöltődik a PC-be a multiplexer jobb bemenetén keresztül. Ez akár 4 ciklus kiesését is jelenti a feldolgozásból.

A másik megoldás esetén eleinte mindig azt feltételezzük, hogy az ugrás feltétele nem teljesül, és az IF fázis szekvenciálisan folytatja az utasítások betöltését. Viszont, ha az ugró utasítás EX fázisában kiderül, hogy az ugrás feltétele teljesült, akkor a MEM fázisban a PC tartalma felülíródik az ugrási címmel, s az ugró utasítás után betöltött utasításokat, melyek az IF, ID, RD és EX fázisban vannak, érvénytelenítjük, más szóval „kiöblítjük” (flush) a csővezetékből. Ebben az esetben a 4 ciklus csak akkor esik ki, ha az ugrás feltétele teljesült. Ezt a módszert statikus elágazásjövendölésnek (static branch prediction) is nevezik. Léteznek ennél sokkal kifinomultabb elágazásbecslési módszerek is, melyekkel részletesebben a szuperskalár csővezetékek tárgyalásánál foglalkozunk majd.



2.29. ábra – Vezérlési hazárd kiküszöbölése a TYP csővezetékben

2.2.4.5 Példa különböző függőségek kezelésére a TYP csővezetékben

A TYP csővezetékben fellépő különböző függőségek hatásának és kezelésének illusztrálására lássunk egy egyszerű gyakorlati példát. A csővezeték működését a következő assembly programrészlet[8] segítségével vizsgáljuk:

```
        LB   R1, src1     ; Load Byte (with sign-extend)
        LB   R2, src2     ; Load Byte (with sign-extend)
        SUB  R1, R1, R2   ; Subtract: R1 = R1–R2
        BGEZ R1, end      ; Branch if Greater Than or Equal Zero
        NEG  R1, R1       ; Negate: R1 = -R1
  end:  SB   dst, R1      ; Store Low Byte
        ...
```

[8] A példában a MIPS R2000 processzor utasításai szerepelnek, ugyanis ennek a processzornak az utasítás-csővezetéke hasonlít leginkább a hipotetikus TYP csővezetékre.

Copyright © 2021 / BME AUT

Az utasítás mnemonikok jelentését megjegyzésként minden sorhoz feltüntettük. Tegyük fel, hogy az `src1` és `src2` memóriacímeken az `src1 = 0x01` és `src2 = 0x02` adatok találhatók.

Azzal a feltételezéssel élünk, hogy a csővezetékben nincs kiépítve az operandus-előrecsatoló hálózat, valamint az ugró utasítások esetén nem alkalmazunk statikus elágazásjövendölést. Tehát mind RAW adatfüggőség, mind vezérlési függőség esetén az érintett csővezetékfázisok felfüggesztésével küszöböljük ki a hazárdok fellépését. Maga a program egy kivonást és egy abszolútérték-képzést hajt végre:

$$dst = |src_1 - src_2|,$$

ahol az abszolútérték kiszámítása úgy történik, hogy a kivonás eredményének előjelét egy feltételes elágazás (BGEZ) feltételében vizsgáljuk meg. Pozitív előjel vagy nulla esetén átugrunk egy utasítást (NEG), ami a szám negálását végezné el. Negatív előjel esetén nincs ugrás, tehát az előjelet pozitívra változtatjuk a NEG művelet segítségével.

A 2.1. táblázatban nyomon követhetjük az egyes utasítások végrehajtásának folyamatát. Az egymás melletti oszlopok az egymást követő gépi ciklusokat jelentik. Minden utasításnál láthatjuk, hogy mikor melyik csővezeték-fázisban vannak éppen. A MEM fázist M-el rövidítettük, az S a felfüggesztést (stall) jelöli. Azt alsó két sorban az R1 és R2 regiszterek értékének változásait is megfigyelhetjük.

Gépi ciklus száma	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
`LB R1, src1`	IF	ID	RD	EX	M	WB																
`LB R2, src2`		IF	ID	RD	EX	M	WB															
`SUB R1, R1, R2`			IF	ID	S	S	S	RD	EX	M	WB											
`BGEZ R1, end`				IF	S	S	S	ID	S	S	S	RD	EX	M	WB							
`NEG R1, R1`								IF	S	S	S	S	S	ID	RD	EX	M	WB				
`end: SB dst, R1`														IF	ID	S	S	S	RD	EX	M	WB
R1 értéke	?					1					-1							1				
R2 értéke	?						2															

2.1. táblázat – A TYP csővezeték végrehajtási fázisai a vizsgált példában

A program vizsgálata során a következő függőségeket fedezhetjük fel:

- Az első két betöltő (LB) utasítás írja az R1 és R2 regisztereket, amiket a harmadik (SUB) utasítás felhasznál. Tehát a „`SUB R1, R1, R2`” utasításnak RAW típusú adatfüggősége van az előző kettővel. Emiatt három stall fázis beiktatására van szükség (5–7. gépi ciklus), hogy az RD fázisban már elérhető legyen mindkét bemeneti regiszter új értéke. Vegyük észre, hogy a SUB utasítás felfüggesztése az őt követő utasítást is várakozásra kényszeríti.
- A „`BGEZ R1, end`” feltételes ugró utasítás kétféle függőségben is érintett. Egyrészt az ugrási feltételben az R1 regiszter értéke szerepel, amit az előző (SUB) utasítás ír, tehát

Copyright © 2021 / BME AUT

egy újabb RAW adatfüggőséggel van dolgunk. Emiatt a BGEZ utasítás másodszor is várakozni kényszerül (9–11. gépi ciklusok). Másrészt a feltételes ugrás vezérlési függőséget okoz, ezért a következő utasítás (NEG) végrehajtását fel kell függeszteni mindaddig, amíg kiderül, hogy valóban azt kell-e betölteni, vagy az ugrási címen lévő másik (SB) utasítást (vegyük észre, hogy a BGEZ utasítás dekódolásával párhuzamosan a NEG utasítás felhozatala már megtörtént). A BGEZ utasítás EX fázisában (13. gépi ciklus) kiderül, hogy nincs ugrás, vagyis a MEM fázisban nem fogjuk felülírni a programszámlálót, ezért a MEM fázissal (14. gépi ciklus) párhuzamosan már folytatódhat a NEG utasítás feldolgozása.

- Abban az esetben, ha nincs ugrás, vagyis az eredmény tárolását végző SB utasítást a NEG utasítás előzi meg, akkor az R1 regiszter módosítása miatt egy újabb RAW adatfüggőség lép fel, és ez újabb három stall fázist okoz (16–18. gépi ciklusok).

2.2.5 Valós utasítás-csővezeték példák

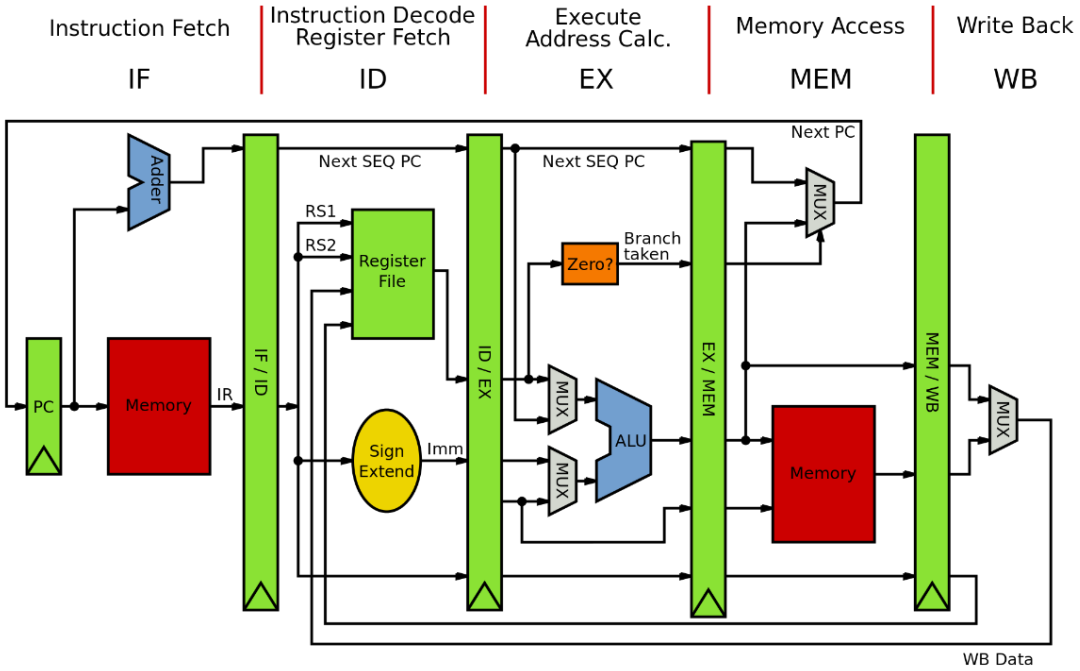
A következőkben két valós processzor utasítás-csővezetékére láthatunk példát. A működésük részletes tárgyalásától eltekintünk, a célunk csupán az előzőekben látott koncepciók illusztrálása. Az egyik processzor a korábban is említett, RISC architektúrájú MIPS R2000/R3000 1985-ből, illetve 1988-ból[9]. A csővezeték felépítése nagyban hasonlít a TYP csővezetékre, de itt az ID és RD fázisok össze vannak vonva. Ezt az teszi lehetővé, hogy a processzor kétfázisú órajellel működik, és minden CPU ciklusban mindkét órajel-fázis aktív éléhez lehet az eseményeket szinkronizálni. Így tehát minden CPU ciklus két alfázisra (Φ_1, Φ_2) bontható (ld. 2.2. táblázat). A MIPS R2000 a maga korában az első processzorok egyike volt, amely lehetővé tette azt, hogy minden gépi ciklusban befejeződjön egy utasítás.

Fázis	Végrehajtott művelet
1. IF	Φ_1: Virtuális cím transzformálása. Φ_2: Utasítás cache hozzáférés a fizikai címmel.
2. ID	Φ_1: Utasítás beolvasása és dekódolása. Φ_2: Regiszterek olvasása.
3. EX	Φ_1: ALU művelet elindítása, ha elágazás, akkor feltétel vizsgálat. Φ_2: ALU művelet befejezése, ha load/store, akkor a virtuális cím transzformálása.
4. MEM	Φ_1: Adat cache címzés. Φ_2: Adat betöltése a cache-ből.
5. WB	Φ_1: Regiszterek írása. Φ_2: –

2.2. táblázat – A MIPS R2000/R3000 csővezeték fázisai

[9] A MIPS R3000 processzor különböző szerver és workstation alkalmazások mellett például a Sony PlayStation és PlayStation 2 játékkonzolokban került alkalmazásra.

Copyright © 2021 / BME AUT



2.30. ábra – A MIPS R2000/R3000 processzor csővezetéke[10]

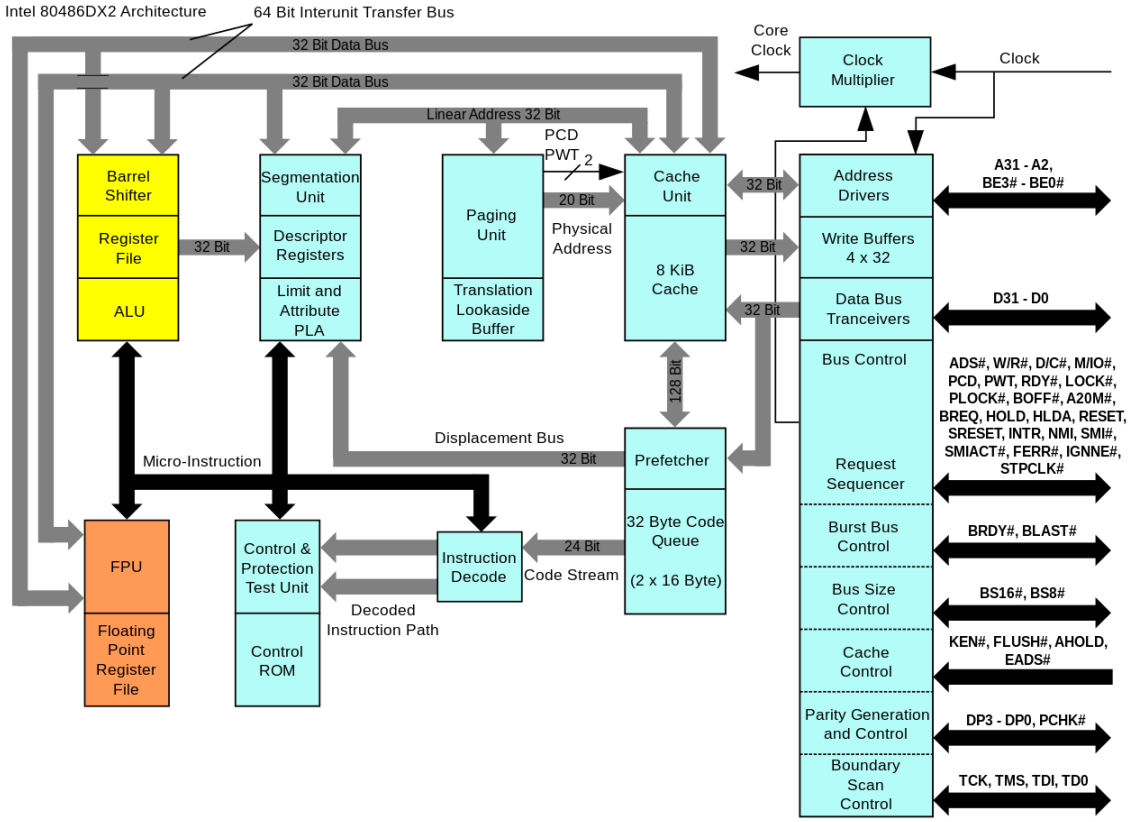
A másik példánk az Intel 80486-os CISC processzora (i486) 1989-ből, amely személyi számítógépek elterjedt x86 alapú processzora volt az 1990-es évek első felében. Ez is ötfázisú csővezetéket tartalmazott, de a komplex utasításkészletből következő mikroprogramozott végrehajtómű miatt a csővezeték időzítése kevésbé szabályos. A 2.3. táblázatban láthatjuk, hogy a dekódolás folyamatát két csővezeték-fázisra bontották. Ezek közül az ID-1 fázis szolgál az utasítás típusának meghatározására, ez egy vagy két ciklus hosszú is lehet az utasításkód hosszától függően. Az ID-2 fázisban fejeződik be az utasítás dekódolás, és itt történik az operandus memóriacímek előállítása és a közvetlen konstansok dekódolása is. Ez a fázis szintén egy vagy két ciklusos lehet a címzési módoktól függően. Az EX fázis működését mikroprogram vezérli, itt történik a regiszter operandusok olvasása, az ALU művelet végrehajtása és az adat cache hozzáférés is. Emiatt az EX fázis is több ciklusos lehet attól függően, hogy milyen utasításról van szó éppen. Például egy olyan aritmetikai művelet esetén, amelynél a forrás- és céloperandusok is a memóriában vannak, három ciklus hosszú a végrehajtási fázis.

[10] Forrás: https://hu.wikipedia.org/wiki/MIPS_architekt%C3%BAra (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

Fázis	Végrehajtott művelet
1. IF	Utasításbetöltés a 32 bájtos előbetöltő sorból.
2. ID-1	Az utasítás transzformálása vezérlőjelekbe vagy mikrokód címbe.
3. ID-2	Hozzáférés a mikrokód memóriához. Címgenerálás és memória-hozzáférés elindítása.
4. EX	Regiszterek olvasása, ALU és memóriaművelet végrehajtása.
5. WB	Eredmény visszaírása regiszterbe.

2.3. táblázat – Az Intel i486 csővezeték fázisai



2.31. ábra – Az Intel i486 processzor blokkvázlata[11]

[11] Forrás: https://en.wikipedia.org/wiki/Intel_80486 (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

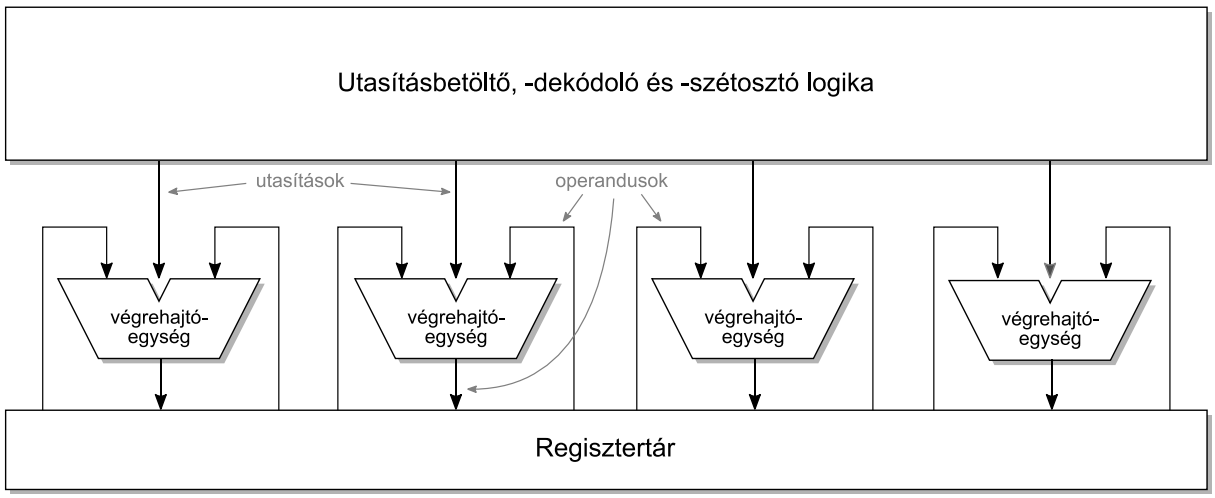
2.3 Szuperskalár processzorok

Az egyszerű (skalár) csővezetékes processzorok alkalmazásának nagy eredménye, hogy k-fázisú csővezetékkel k-szoros elvi sebességnövekedést lehet elérni az utasításvégrehajtásban. Ugyanakkor ennek a megoldásnak is vannak korlátai, ami további fejlesztésekre sarkallta a processzortervezőket.

A skalár csővezeték maximális átbocsátóképessége ciklusonként egy utasítás ($IPC = 1$). A végrehajtási sebesség növelése ezért a ciklusidő csökkentésével érhető el, amihez a csővezeték mélységét (a fázisok számát) kell növelni. Ugyanakkor a többfokozatú csővezetékben az utasítások egymásra hatása miatt kialakuló „bedugulások” hosszabb időkiesést okoznak, és ez a csővezeték mélyítésének határt szab. Ráadásul eltérő végrehajtási idejű utasítások esetén a ciklusidőt a leghosszabbhoz kell megválasztani, hogy az EX fázis minden esetben befejeződhessen a ciklusidő végéig. Emiatt az összes utasítástípus egyetlen csővezetékben való „egységesítése” a hatékonyságnak korlátot szab. Az egyetlen csővezeték természetes velejárója az is, hogy „bedugulás” esetén nem csak az időkiesést okozó függő utasításnak kell várakoznia, hanem a programban később szereplő összes többinek is.

2.3.1 Térbeli utasításszintű párhuzamosítás

A csővezetékes utasításvégrehajtás által megvalósított időbeli (vertikális) párhuzamosítás mellett további teljesítőképesség-növekedést lehet elérni a térbeli (horizontális) párhuzamosítással, vagyis az utasítás-csővezeték többszörözésével egy processzoron belül. Ezt a megoldást *szuperskalár* szervezésnek nevezzük. Ha a processzoron belül több párhuzamos végrehajtóegység található (ld. 2.32. ábra) egyidejűleg több utasítás (azonos fázisának) végrehajtása válik lehetővé, így az átbocsátóképesség elvi maximuma átlépheti az $IPC = 1$ értéket.



2.32. ábra – Párhuzamos végrehajtóművek egy processzoron belül

Copyright © 2021 / BME AUT

Az ötlet nagyon hasonló a többprocesszoros rendszerekhez. A párhuzamos, ill. többmagos processzorokhoz képest ez azonban sokkal szorosabb viszonyt jelent a végrehajtóegységek között: míg ott különálló programszálak végrehajtása lehetséges egy időben, a szuperskalár processzorokban egyetlen programszál utasításainak párhuzamosításáról van szó.

A térbeli utasításszintű párhuzamosság (instruction-level parallelism, ILP) alkalmazása azon alapul, hogy egy szekvenciális programban az egymást követő utasítások gyakran függetlenek egymástól, ezért párhuzamosan is végrehajthatók. Lássunk erre egy példát az alábbi programrészlet segítségével:

```
Programrészlet:                 Párhuzamosan végrehajtható részek:

LD  R1, (R2)                    1.  LD  R1, (R2)    ||  ADD R5, R6, R7
ADD R5, R6, R7                  2.  SUB R4, R1, R4  ||  MUL R8, R9, R10
SUB R4, R1, R4                  3.  ST  (R11), R4
MUL R8, R9, R10
ST  (R11), R4
```

Az öt utasítás vizsgálatakor szembetűnik, hogy az első és a harmadik, illetve a harmadik és az ötödik utasítás között RAW típusú regiszter-adatfüggőség áll fenn az R1, illetve az R4 regiszter vonatkozásában. A második és a negyedik utasítás viszont független az őket megelőző műveletektől, ezért semmi sem akadályozza meg, hogy a végrehajtásuk párhuzamosan történjen, ha van egy második csővezeték a processzorban. Ezt láthatjuk a jobb oldali programrészletben: öt helyett három gépi ciklus is elég lehet az utasítások befejezéséhez. Ehhez azt feltételezzük, hogy voltak megelőző utasítások is a programban, ezért a csővezeték feltöltésének idejét elhanyagoltuk. Továbbá azt is feltételezzük, hogy ki van építve az operandus-előrecsatoló hálózat a csővezeték-fázisok között, így a RAW adatfüggőség nem okoz stall fázisokat a végrehajtás során.

2.3.1.1 Szuperskalár processzorok alapvető tulajdonságai

A szuperskalár processzorokkal tehát lehetővé válik több utasítás kibocsátása is gépi ciklusonként ($IPC > 1$). Ezeknek a processzoroknak fontos jellemzője, hogy *hardveresen* támogatják az utasítások párhuzamosíthatóságának kiaknázását, ami azt jelenti, hogy az utasításkészlet-architektúra szintjén azonosak a skalár processzorokkal. Vagyis a programozónak, ill. fordítóprogramnak nem kell tudnia a belső párhuzamosításról, az rejtve marad a mikroarchitektúrában. Ennek nagy előnye, hogy egy adott számítógépcsaládban a szuperskalár szervezés bevezetésével úgy érhető el teljesítőképesség-növelés, hogy közben nem sérül az utasításszintű kompatibilitás a korábbi családtagokkal.

Ettől függetlenül a fordítóprogram *támogathatja* a kód hatékony végrehajtását megfelelő utasításelhelyezéssel a programon belül. Ugyanis a processzor végrehajtás közben egy adott méretű ablakban keresi a párhuzamosítható utasításokat. A párhuzamosítás nélküli esethez képest a hatékonyságnövekedést nagyban befolyásolja a pillanatnyilag vizsgált utasításablakon

Copyright © 2021 / BME AUT

belül talált független utasítások száma. Míg a hardver csak lokálisan képes elemezni a kódot, addig a fordítónak minden lehetősége megvan a globális optimalizálásra is.

2.3.1.2 Sorrenden kívüli végrehajtás

Korábban említettük, hogy a skalár csővezeték egyik hátránya, hogy egy utasítás végrehajtásának felfüggesztése esetén az összes soron következő utasítás is várakozni kényszerül. Például egy memóriaolvasásnál, ha a hivatkozott adat nincs benne a gyorsítótárban, akkor meg kell várni, míg megérkezik a főmemóriából, ami nagyon sok ciklusnyi felfüggesztést jelent. Ezalatt egy szuperskalár processzorban folytatódhat a következő utasítások feldolgozása, amíg van szabad műveletvégző egység és elegendő független utasítás. Ugyanakkor általában a független utasítások száma jóval nagyobb, mint ahány párhuzamos végrehajtóegység rendelkezésre áll. Ahhoz, hogy az utasítások az eredeti programsorrendben fejeződhessenek be, a végrehajtás alatt lévő utasításoknak meg kell várniuk a felfüggesztést okozó művelet végét, mielőtt újabb utasításokat lehetne betölteni a csővezetékbe. Például, ha négy párhuzamos végrehajtóegység található a processzorban, és a memóriaolvasás több tíz ciklusnyi késleltetést okoz, akkor három további utasítás feldolgozása megkezdődhet, de a befejezésük nem előzheti meg az olvasás befejezését. Így hiába lennének további független utasítások, a várakozás idejéből csak három ciklusnyit sikerül megspórolni.

A szuperskalár processzorok hatékonyságát nagyban növeli, ha megengedjük az utasítások programsorrenden kívüli végrehajtását. Ebben az esetben a fent vázolt példánkban mindaddig folytatódhat a soron következő utasítások feldolgozása, amíg elegendő független utasítás rendelkezésre áll. Természetesen ez komplexebb hardvert igényel, mert ahhoz, hogy az ISA szintjén elhelyezkedő programozó számára továbbra is átlátszó maradjon a működés, meg kell oldani a sorrenden kívül végrehajtott utasítások visszarendezését is. A sorrenden kívüli végrehajtás nem csak a komolyabb felfüggesztést okozó utasítások esetén növeli a feldolgozás hatékonyságát, hanem alapesetben is lehetőséget ad az utasításszintű párhuzamosság nagyobb mértékű kiaknázására. Szemléltessük ezt a következő egyszerű példa segítségével:

```
LD  R1, (R2)
ADD R3, R1, R4
SUB R5, R6, R7
MUL R8, R9, R10
```

Az egymás után következő négy utasítás közül az első kettő között RAW típusú regiszter-adatfüggőség áll fenn, a többi két utasítás viszont független. Tegyük fel, hogy processzorunk két párhuzamos végrehajtóegységet tartalmaz. Sorrendi végrehajtás esetén a következő történik:

```
1.  LD  R1, (R2)     ||   (NOP)
2.  ADD R3, R1, R4   ||   SUB R5, R6, R7
3.  MUL R8, R9, R10  ||   ...
```

Copyright © 2021 / BME AUT

Mivel a második (ADD) utasítás függ az előző (LD) eredményétől, ezért a végrehajtása nem kezdődhet meg azzal egyidejűleg, csak időben utána (akkor is, ha meg van oldva az operandus-előrecsatolás). Így egy ciklusnyi „buborék” keletkezik a második csővezetékben. A harmadik (SUB) utasítás a másodikkal együtt már betölthető, hiszen független tőle, utána a negyedik, és így tovább.

Sorrenden kívüli végrehajtás esetén meg tudjuk spórolni az üres ciklust, hiszen a független utasítások megelőzhetik egymást. Így a példában harmadik (SUB) utasítás már a második (ADD) előtt betölthető, és a két csővezeték maximális kihasználtsággal működhet:

```
1.  LD  R1, (R2)     ||   SUB R5, R6, R7
2.  ADD R3, R1, R4   ||   MUL R8, R9, R10
```

2.3.1.3 A térbeli utasításszintű párhuzamosság hatékonyságának korlátai

A térbeli utasításszintű párhuzamosságon alapuló teljesítőképesség-növekedésnek is vannak korlátai. Az egyik ilyen korlátozó tényező a párhuzamos végrehajtóegységek száma és fajtája. *Strukturális hazárdnak* nevezzük, ha kevesebb a szabad végrehajtóegység, mint ahány független utasítás rendelkezésre áll, és emiatt kell várakoznia a végrehajtásnak. Szintén korlátozó tényező a lokális utasításablak véges mérete, amelyben a párhuzamosítható műveleteket keressük. Ezeken túl a párhuzamosítási lehetőségeket jelentősen befolyásolják az ablakon belüli utasítások függőségei. A skalár csővezetéknél tárgyalt vezérlési és adat-függőségek hatása itt is jelentkezik.

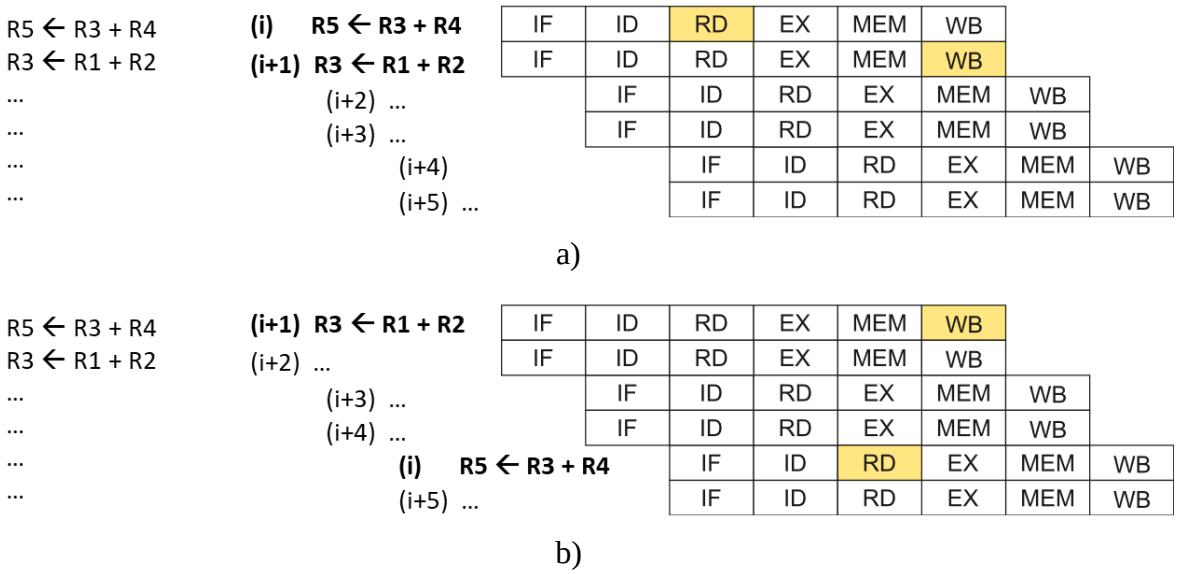
Az elágazások okozta vezérlési függőség miatti időkiesés a szuperskalár csővezetékeket sokkal jobban sújtja, hiszen az elágazási feltétel, illetve az ugrási cím kiszámításáig nem csak egy, hanem az összes párhuzamos csővezetéket fel kell függeszteni. Ellenkező esetben a téves ágról felhozott utasításokat az összes csővezetékből ki kell üríteni. Ezért szuperskalár processzorokban nagyon felértékelődik a hatékony elágazásjövendölés szerepe.

A RAW típusú (valódi) adatfüggőségek sorrendi végrehajtást követelnek meg, hiszen a szekvenciában későbbi utasítás nem tud addig továbblépni, amíg a bemenő adatai rendelkezésre nem állnak. A skalár csővezetéknél már bemutatott operandus előrecsatolás elve itt is alkalmazható, csak az előrecsatoló hálózat lesz bonyolultabb, hogy a párhuzamos csővezetékek között is lehetővé tegye a paraméterátadást. Ez lehetővé teszi a stall ciklusok kiküszöbölését, de a függő utasítások sorrendjét nem lehet felcserélni.

A WAW típusú (kimeneti) adatfüggőségek szintén sorrendben történő végrehajtást követelnek meg, hogy a későbbi írás eredménye maradjon a regiszterben.

A WAR típusú (anti)függőség esetén egyidejűleg elkezdhető a függő utasítások végrehajtása, sőt akár a sorrendjük is felcserélhető, de vigyázni kell, hogy az RD fázis mindenképpen hamarabb befejeződjön a következő utasítás WB fázisánál, ahogy azt a 2.33. ábra is szemlélteti.

Copyright © 2021 / BME AUT



2.33. ábra – Utasítások végrehajtása WAR regiszter adatfüggőség esetén szuperskalár csővezetékben. a) Egyidejű végrehajtás. b) Sorrenden kívüli végrehajtás.

Vizsgáljuk meg egy példán keresztül az adatfüggőségek hatását az utasítások párhuzamos és sorrenden kívüli végrehajtására:

Hogyan ütemezné az alábbi programrészlet végrehajtását egy olyan szuperskalár processzor, amelynek két azonos végrehajtóegysége van, és sorrenden kívüli végrehajtásra is képes? Az egyszerűség kedvéért tegyük fel, hogy WAR függőség esetén megtiltjuk az utasítások felcserélését. Továbbá tegyük fel azt is, hogy ki van építve az operandus-előrecsatoló hálózat a processzorban.

```
ADD R1, R2, R3    \                \
ADD R4, R5, R6     RAW  \           \
SUB R7, R1, R9    /      RAW  \      WAW
MUL R5, R4, R4          /      WAR  /
SUB R1, R12, R10              /    /
ST  (R13), R14
OR  R15, R14, R12
```

A függőségek azonosítása után könnyen belátható, hogy például a következő végrehajtási sorrend lehetséges. Láthatjuk, hogy elegendő független utasítás áll rendelkezésre ahhoz, hogy mindkét végrehajtóegység folyamatosan dolgozzon:

```
1.  ADD R1, R2, R3     ||   ADD R4, R5, R6
2.  SUB R7, R1, R9     ||   MUL R5, R4, R4
3.  SUB R1, R12, R10   ||   ST  (R13), R14
4.  OR  R15, R14, R12  ||   ...
```

Copyright © 2021 / BME AUT

Most nézzük meg, hogyan ütemezné ugyanezt a programrészletet egy négy párhuzamos végrehajtóegységgel rendelkező processzor:

```
1.  ADD R1,R2,R3 || ADD R4,R5,R6 || ST (R13), R14  || OR R15,R14,R12
2.  SUB R7,R1,R9 || MUL R5,R4,R4 || SUB R1,R12,R10 || ...
```

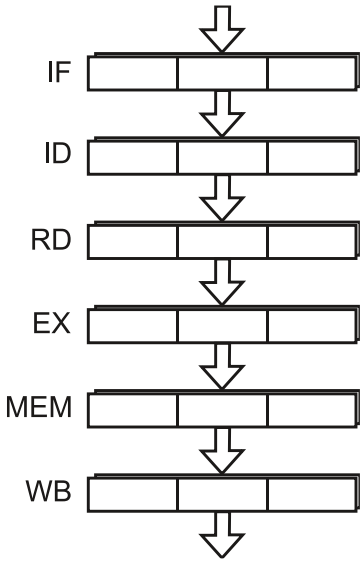
A programban az első (ADD) és a harmadik (SUB), illetve a második (ADD) és a negyedik (MUL) utasítások között fennálló RAW függőségek miatt ezeknek az utasításpároknak programsorrendben kell követniük egymást. Ezt láthatjuk az első és a második műveletvégző egység oszlopában. Ugyanakkor még van két szabad végrehajtóegységünk, ezért az első gépi ciklusban a hatodik (ST) és a hetedik (OR) utasítás is beütemezhető, mert minden előzőtől függetlenek. A második gépi ciklusban a már beütemezett SUB és MUL műveletek mellé az ötödik (SUB) utasítás is beütemezhető, hiszen bár a két SUB utasítás között WAR függőség áll fenn, ez nem akadályozza meg az egyidejű végrehajtást. ■

2.3.2 A szuperskalár csővezetékek fajtái

Szuperskalár csővezetékek többféle megvalósításban is napvilágot láttak. Három lényeges változatot különböztetünk meg, ezek az *egységes szerkezetű*, a *diverzifikált* és a *dinamikus csővezetékek*. A mai modern szuperskalár processzorok körében a dinamikus csővezeték alkalmazása a jellemző, ez az a változat, amely lehetőséget ad a sorrenden kívüli utasításvégrehajtásra is.

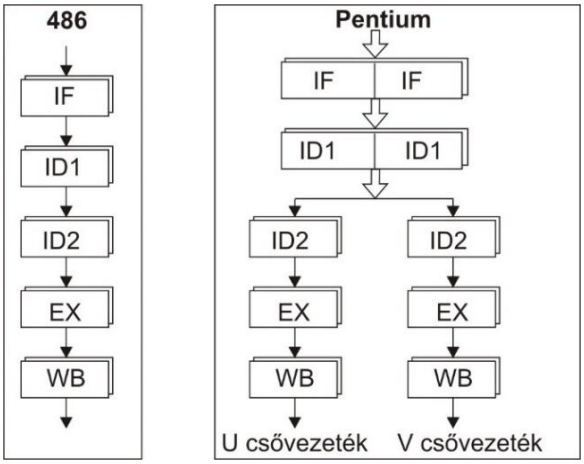
2.3.2.1 Egységes szerkezetű csővezeték

A szuperskalár csővezeték legkézenfekvőbb megvalósítása a hagyományos utasítás-csővezeték egyszerű többszörözését jelenti. N db párhuzamos csővezetékből áll, amelyek teljesen azonos fázisokat tartalmaznak, ahogyan azt a 2.34. ábra is érzékelteti. Az időbeli és térbeli utasításszintű párhuzamosság tiszta megvalósítását képviseli, amelyhez az egyes fázisokat megvalósító hardveregységek N-szeres többszörözésén túl a párhuzamos fázisok közötti kommunikációhoz nagyszámú (általános esetben N^2 nagyságrendű) kapcsolat kiépítésére is szükség van. Ezen kívül a regiszterblokk és az adatgyorsítótár kapuit is N-szeresen többszörözni kell, hogy az operandusok betöltése és visszaírása konkurensen működhessen a párhuzamos csővezetékekben. A végrehajtás teljesen programsorrendben történik.



2.34. ábra – Egységes szerkezetű csővezeték

Copyright © 2021 / BME AUT



2.35. ábra – Az Intel Pentium processzor csővezetéke

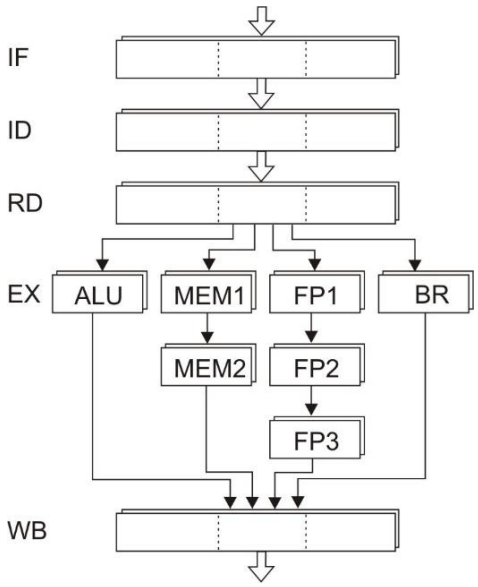
Ilyen egységes szerkezetű szuperskalár processzor például az Intel Pentium (1993), az i486-os processzor utódja, amely lényegében két i486-os csővezetéket valósított meg (ld. 2.35. ábra). Az első két fázisban gépi ciklusonként két utasítás betöltését és dekódolását hajtja végre, és egy-egy utasítást tud kibocsátani az U és V jelű csővezetékekbe, ami jelentős teljesítőképesség-növekedést jelentett az elődhöz képest. Természetesen mindez járulékos hardver erőforrásokat, például az öt csővezeték-fázist megvalósító hardveregységek megduplázását igényli.[12] Ahogy az i486-os processzor esetében már láttuk, az EX fázisban ALU művelet és adat cache hozzáférés is lehetséges. A két csővezeték miatt meg kellett oldani a regisztertár és a gyorsítótár vonatkozásában is a konkurens hozzáférést. A regisztertár esetén ez járulékos portok kiépítését jelentette. Viszont a dualport cache megvalósítása igen költséges, ezért egykapus, 8-utas asszociatív gyorsítótárat alkalmaztak. Különböző gyorsítótár-blokkokhoz konkurens módon mindkét csővezeték hozzá tudott férni, de azonos blokkhoz való egyidejű hozzáférési igényeket a hardvernek sorosítania kellett.

2.3.2.2 Diverzifikált csővezeték

A különböző típusú utasítások különböző végrehajtási időket és műveletvégző egységeket igényelnek. Ha ezek végrehajtását egyetlen univerzális csővezetékben szeretnénk megvalósítani, akkor a különböző típusú végrehajtási fázisokat sorosítanunk kell egymás után. Ilyenkor a legtöbb esetben lesznek „tétlen” fázisok, amelyek nem érintik az adott utasítás végrehajtását, viszont attól még át kell haladnia rajtuk. Erre láttunk egyszerű példát a TYP csővezeték vizsgálatakor, amelyben különálló MEM fázis van fenntartva a memória-hozzáférések végrehajtásához. Egy regiszterreferens ALU utasításnál a MEM fázisban semmi nem történik az egyszerű továbbléptetésen kívül, ami egy ciklusnyi fölösleges késleltetést jelent az adott utasítás szempontjából (ld. 2.17. ábra). A 2.34. ábrán bemutatott egységes szerkezetű szuperskalár csővezetéknél is ugyanezt látjuk, párhuzamosan több példányban.

[12] A teljes igazsághoz hozzátartozik, hogy az U és V csővezetékek a korlátozott tranzisztorszám miatt nem voltak teljesen egyenértékűek. A V csővezeték csak egyszerűbb utasításokat tudott feldolgozni, amelyek nem igényeltek mikrokód támogatást. Ha az ID1 fázisban kiderült, hogy a V ág utasítása nem minősül egyszerűnek, akkor az U ágnak adódott át, és a V ág üres maradt mindaddig, amíg nem érkezett általa is végrehajtható utasítás.

Copyright © 2021 / BME AUT



2.36. ábra – Diverzifikált csővezeték

Ha elképzeljük, hogy az utasításkészlet tartalmaz egész és lebegőpontos aritmetikai műveleteket, ezekből is egyszerűbbeket (pl. összeadás, kivonás) és komplexebbeket (pl. szorzás, osztás), emellett memóriaműveleteket és különböző vezérlésátadó utasításokat, akkor el tudjuk képzelni, hogy a különböző műveletvégző fokozatok egymás után állítása mennyi fölösleges fázist eredményez, főleg az egyszerűbb utasításokra nézve. Emiatt szuperskalár szervezésnél célszerű az egyes csővezetékek végrehajtó (EX) fokozatait diverzifikáltan, azaz különböző típusú és számú fázisokból kialakítani, ahogy ez a 2.36. ábrán is látható. Így minden utasításnak csak a számára szükséges fázisokon kell áthaladnia, megspórolva fölösleges végrehajtási ciklusokat.

Vegyük észre, hogy a diverzifikált csővezeték esetén az EX fázis(ok)at megvalósító párhuzamos funkcionális egységek száma eltérhet az utasításbetöltő, dekódoló és visszaíró csővezeték-szakaszok számától. Érezhetjük, hogy ez a csővezeték-szervezés bár hatékonyabb az előző változatnál, de csak akkor lesz teljesen kihasználva, ha az utasítások típusa és eloszlása olyan a programon belül, amely jól illeszkedik a csővezetékek típusaihoz és késleltetéseihez. Ugyanakkor a sorrendi végrehajtás érdekében, ha egy hosszabb csővezeték még nem végzett egy utasítással, akkor a következő – párhuzamos csővezetékekben esetleg már végrehajtott – utasítások visszaírása várakozni kényszerül.

A különböző utasítástípusok párhuzamos funkcionális egységekkel történő végrehajtására a számítógépek több generációjában is találunk példákat. Már a CDC 6600 mainframe CISC számítógép (1964) is például 10 párhuzamos végrehajtó funkcionális egységgel rendelkezett. Ezek között volt fixpontos és lebegőpontos összeadó, két szorzó, egy osztó, egy eltoló, egy logikai műveletvégző, két inkrementáló és egy elágazásokat kezelő egység, amelyek a különböző utasításokat 3–29 órajelciklus alatt hajtották végre. Másik példa lehet a Motorola 88110 RISC mikroprocesszor (1992), szintén 10 párhuzamos műveletvégző egységgel.

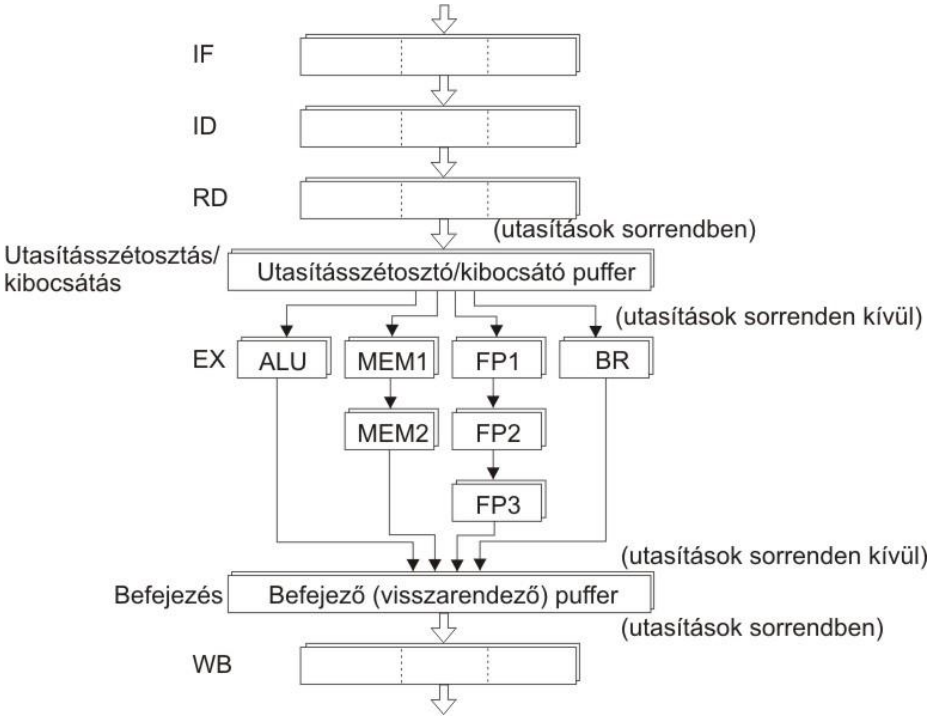
Copyright © 2021 / BME AUT

2.3.2.3 Dinamikus csővezeték

A dinamikus csővezeték-szervezés esetén a végrehajtó egységeket (EX fázis) két több-bemenetű puffer fogja körül. Az első, más néven utasításszétosztó/kibocsátó pufferbe programsorrendben töltődnek be az utasítások, de potenciálisan sorrenden kívül hagyhatják el azt. Egy utasítás ugyanis csak akkor kerül át az EX fázisba (fázisokba), ha a bemenő operandusai mind rendelkezésre állnak, s van az adott utasítás végrehajtásához szükséges szabad műveletvégző egység. A korábban tárgyalt megoldásokban a csővezeték-fázisok közötti pufferek átmenetileg annyi utasítást tároltak, amennyi párhuzamos csővezeték volt. Ezzel szemben most az utasításszétosztó/kibocsátó puffer a csővezetékek számánál több utasítást tárolhat.

Az utasítások végrehajtásának sorrendje a potenciálisan sorrenden kívüli utasításkiosztás mellett az egyes funkcionális egységek különböző végrehajtási ideje miatt is eltérhet a programbeli sorrendtől. A kivételek és megszakítások helyes kezelése érdekében azonban az utasítások befejezése, azaz a gépi állapot frissítése az eredeti utasítássorrendben kell, hogy történjen. Ezért egy másik pufferre, az ún. befejező (visszarendező) pufferre is szükség van, amely visszaállítja az eredeti utasítássorrendet. Ez visszarendező puffer tárolja az EX fázis által sorrendtől eltérően befejezett utasításokat, és az eredeti programsorrendnek megfelelően adja át azokat a WB fázisnak, amely beállítja a tényleges architekturális állapotot.

A mai modern szuperskalár processzorok többsége dinamikus csővezetéket tartalmaz. A későbbi fejezetekben több példát is bemutatunk mind a CISC (pl. Intel P6, Nehalem mikroarchitektúrák), mind pedig a RISC processzorok (pl. ARM Cortex-A család) köréből.



2.37. ábra – Dinamikus csővezeték

Copyright © 2021 / BME AUT

2.3.3 A szuperskalár csővezetékek dinamikus viselkedése

A szuperskalár csővezetékekben a skalár csővezetékekhez hasonlóan függőség léphet fel az utasítások között. Ezeket fel kell ismerni, és hatásukat a hatékonyság lehető legkisebb mértékű csökkentésével ki kell küszöbölni. Dinamikus viselkedés alatt itt a programutasítások függőségeinek és a csővezeték viselkedésének az egymásra hatását értjük. A következőkben bemutatott problémák és megoldások a szuperskalár csővezetékek mindhárom típusára jellemzők. A vizsgált függőségek a korábban látottakhoz hasonlóan itt is adatfüggőségek vagy vezérlési függőségek lehetnek.

2.3.3.1 Adatfüggőségek

Az adatfüggőségeket regiszterben tárolt vagy memóriabeli operandusok okozhatják. Ezek közül itt a regiszter-adatfüggőségekre fókuszálunk, a memória-adatfüggőségek kezelésére később, a konkrét mikroarchitektúrák bemutatásakor látunk majd példát.

Regiszter adatfüggőség aritmetikai/logikai, vagy más regiszter-regiszter típusú utasításokkal kapcsolatban léphet fel. Ahogy korábban is láttuk, a RAW típusú (valódi) adatfüggőségek mindenképpen sorrendi végrehajtást követelnek meg, mert a függő utasításnak meg kell várnia, míg az előző utasítások kimeneteként előállnak a bemenő operandusai. A felfüggesztés és az ebből adódó csővezeték „buborékok” elkerülésére itt is a fázisok közötti előrecsatoló hálózat megvalósítása jelenti a megoldást, ahogyan a skalár csővezeték esetében.

A WAW típusú (kimeneti) és WAR típusú (anti)függőségeket *hamis adatfüggőségnek* vagy *névfüggőségnek* is nevezik, ugyanis ezek nem igazi függőségek, hanem a regiszterek újrahasználatának következményei. Ha egy regisztert felülírunk, az azt jelenti, hogy a régi adattartalomra már nincs szükségünk. A felülírást a regiszterek véges száma teszi szükségessé. Ha lehetséges lenne, hogy egynél többször soha ne írjunk egy regisztert, ezek a függőségek nem fordulnának elő.

Statikus újrahasználatról beszélünk akkor, amikor a fordítóprogram például a memória-hozzáférések minimalizálása céljából regisztereket használ az adatok átmeneti tárolására a műveletek között. Ha a programsorrendben egy regiszter értékére már nincs szükség, akkor azt más célra felhasználja, új adatot ír bele. Több párhuzamos csővezeték és sorrenden kívüli végrehajtás esetén az újrahasznált regiszterekhez való hozzáférések sorrendje felcserélődhet, ami a program hibás működését okozhatja. Ezért ezeket a hazárdokat detektálni kell, és ki kell küszöbölni őket.

Dinamikus újrahasználat rövid ciklusmagok (FOR, WHILE stb.) esetén fordulhat elő. Azt a jelenséget nevezzük így, amikor egyidejűleg több iteráció utasításai is a csővezetékben vannak, ezért ugyanazt a regisztert egyszerre több utasítás (ugyanazon utasítás több példánya) írná vagy olvasná. Ezt az esetet szintén detektálni és kezelni kell.

Copyright © 2021 / BME AUT

2.3.3.2 Hamis adatfüggőségek kezelése

A hamis függőségek fellépése esetén megoldást jelenthet a függő utasítás felfüggesztése és várakoztatása addig, amíg a függőség fennáll. Ez a megoldás nem szerencsés, hiszen csökkenti a hatékonyságot.

Ehelyett elterjedt módszer a *regiszter-átnevezés* alkalmazása, melynek lényege, hogy az ISA által definiált, ún. architekturális regiszterkészlet mellett (vagy helyett) rendelkezésre áll egy nagyobb méretű hardver regisztertár, amely rejtve marad a programozó elől. Ha végrehajtás közben névfüggőség lép fel, akkor a végrehajtó hardver egy új regisztert allokál a foglalt helyett, és egy bejegyzést készít az ún. regiszter álnév táblába (register alias table, RAT). Ez teremti meg az összerendelést az architekturális regiszter neve és a hardveres „álnév” között, amely alapján a következő, ugyanazt a regisztert olvasó utasítások megtalálhatják a szükséges adatot, illetve a befejező (visszarendező) egység el tudja végezni a visszaírást.

A következő egyszerű programrészlet illusztrálja a regiszter-átnevezés működését:

```
ADD R3, R4, R5
LD  R7, (R3)
SUB R3, R12, R11
ST  (R15), R3
```

Látható, hogy az első (ADD) és a harmadik (SUB) utasítás között WAW függőség, illetve a második (LD) és harmadik (SUB) között WAR függőség áll fenn. Ahhoz, hogy a szuperskalár csővezetékben a SUB és az azt követő ST utasítás végrehajtása ne függjön az előző kettőtől, az R3 regisztert egy új hardver regiszterhez kell rendelni. Átnevezés után a hardveres regiszter allokáció például a következőképpen néz ki:

```
ADD hw3, hw4, hw5
LD  hw7, (hw3)
SUB hw20, hw12, hw11
ST  (hw15), hw20
```

2.3.3.3 Vezérlési függőség

A vezérlési függőség a csővezeték betöltési és dekódolási fázisához kapcsolódik. Ennek a két fázisnak a sávszélessége a teljes csővezeték sávszélességének felső korlátját jelenti. Emiatt fontos az optimális működésük, ugyanis, ahogy láttuk, az adatfüggőségekből adódó, hatékonyságot csökkentő hatások részben feloldódnak. A maximális sávszélesség akkor érhető el, ha az utasítások folyamatos betöltése nem szakad meg.

Elágazások, ugró utasítások és szubrutinhívások esetén ez nem teljesül. Feltételes elágazások esetén a következő utasítás az elágazási feltétel kiértékeléséig nem tölthető be. Feltétel nélküli ugrások esetén is várni kell, ha az ugrási cím valamilyen számítás eredményeként adódik.

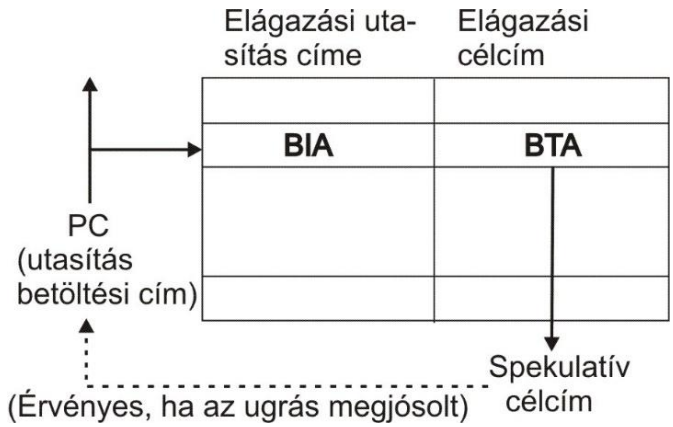
Copyright © 2021 / BME AUT

A vezérlési függőség a csővezeték-fázisok megfelelő felfüggesztésével feloldható, de ez a szuperskalár processzorokban súlyosabb „bedugulást” okoz a skalár processzorokhoz képest, mivel az egyik csővezetékbe bekerülő elágazás a többi csővezetéket is megakasztja. Annak érdekében, hogy a csővezetékek működését ne kelljen több ciklusra felfüggeszteni, különböző elágazásjövendölési (branch prediction) módszereket fejlesztettek ki.

2.3.3.4 Elágazásjövendölés

Az elágazásjövendölés vagy elágazásbecslés olyan spekulatív módszereket takar, amelyek kísérletet tesznek az elágazási feltétel és a célcím meghatározására közvetlenül az IF fázisban, még az utasítás dekódolása és kiértékelése előtt. Így a következő gépi ciklusban a megjósolt címtől kezdve folytatódhat az utasítások felhozatala. Amikor néhány ciklussal később megtörténik a tényleges kiértékelés és célcímszámítás, akkor ellenőrzést végzünk, és egyezés (helyes becslés) esetén minden gond nélkül mehet tovább a végrehajtás a csővezetékben. Ha kiderül, hogy a becslés hibás volt, akkor az ugró utasítás után betöltött összes utasítást el kell dobni az összes párhuzamos csővezetékből, és a helyes címről kell folytatni az utasítások betöltését. Ez utóbbi eset érthető módon jelentős hatékonyságcsökkenéssel jár. Modern processzorokban nagyon fontos szerepe van a hatékony elágazásjövendölésnek, mert a tipikus utasításablak-méretek esetén egyszerre akár 5-6 ugró utasítás is lehet a csővezetékben.

Az elágazásjövendölésnél központi szerepet tölt be az ún. *elágazási célpuffer* (Branch Target Buffer, BTB) alkalmazása. Ez a hardveregység egy hivatkozásokat tartalmazó táblázatot valósít meg, amely az elágazási célcím spekulatív megelőlegezését segíti (ld. 2.38. ábra). Minden egyes sor egy már kiértékelt ugró utasítás címét (Branch Instruction Address, BIA) és a legutóbb megvalósult ugrási célcímet (Branch Target Address, BTA) tartalmazza. A táblázatot a BIA alapján lehet címezni (CAM – Content Addressable Memory), így az IF fázisban már hozzáférhetünk a programszámláló tartalmával. Ha találatot jelez, akkor már az aktuális utasítás felhozatalával egyidőben (tehát még a dekódolás előtt) tudhatjuk, hogy ugró utasítás lesz, sőt a spekulatív célcímet is megkapjuk. A BTA érték azonnal a PC-be tölthető, ha a jövendölési stratégia szerint az ugrás végrehajtandó, így időkiesés nélkül folytatódhat az utasítások felhozatala a feltételezett ágról. A következőkben az ugrási feltétel becslésére alkalmazott tipikus stratégiákat tekintjük át.



2.38. ábra – Elágazási célpuffer

Copyright © 2021 / BME AUT

Statikus elágazásjövendölés

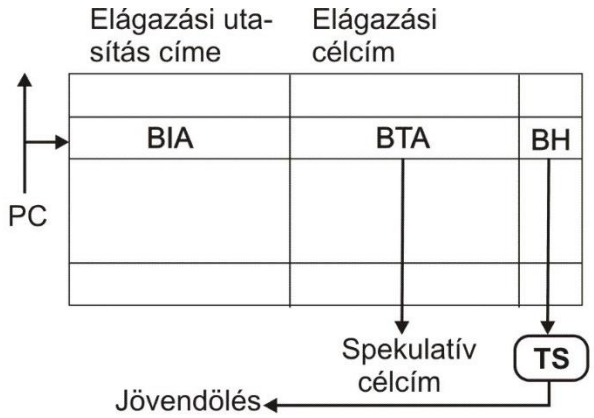
Az elágazásjövendölési módszerek első csoportjába azok tartoznak, amelyek valamilyen statikus szabály alapján becsülik meg az elágazási feltétel kimenetét. A legegyszerűbb megoldásban azt feltételezzük, hogy az elágazási feltétel nem teljesül, és a spekulatív utasításfelhozatal a soron következő címekről folytatódik. A módszer hatékonysága változó: if-then jellegű elágazások esetén hatékony lehet, de például N-szer lefutó ciklusok esetén a kilépési feltételt a ciklustörzs végén vizsgáló utasításnál $(N-1)$-szer tévesen jósol. Ezt a kedvezőtlen hatást ki lehet küszöbölni, ha a ciklustörzs végére plusz egy feltétel nélküli, egyszerű visszaugrást szúrunk be, és a ciklustörzs elején vizsgáljuk a kilépési feltételt. Egy megfelelően felkészített fordító ezt a módszert fogja alkalmazni ilyen statikus jövendölést használó processzornál.

Statikus elágazásjövendölés úgy is alkalmazható, hogy a feltétel teljesülését az elágazási célcím relatív elhelyezkedése alapján becsüljük. Ha a célcím nagyobb értékű, mint az ugró utasítás saját címe (előreugrás), akkor azt feltételezzük, hogy nem teljesül a feltétel, ha viszont kisebb (hátraugrás), akkor a feltétel teljesülését becsüljük. Ez hátul tesztelő ciklusok esetén is, és „hagyományos” if-then jellegű elágazások esetén is jó találati arányt eredményez.

Olyan is előfordul, hogy az elágazásjövendölés támogatása megjelenik az utasításkészlet-architektúra szintjén. Például a korábban említett Motorola 88110 processzor utasításkészletében a feltételes elágazó utasítások egy extra bitet tartalmaznak. Ezt a fordítóprogram arra használhatja, hogy a program analizálásával a legvalószínűbbnek ítélt kimenetelt előrejelezze.

Egyszintű dinamikus elágazásjövendölés

A dinamikus elágazásjövendölési módszerek felhasználják az ugró utasítások előtörténetét is a becsléshez. Ez esetben az elágazási célpuffer minden sorában a BIA és a BTA mellett egy harmadik bejegyzés, az elágazás előtörténet (Branch History, BH) is helyet kap, amelyet jellemzően 2 biten tárolnak (2.39. ábra).

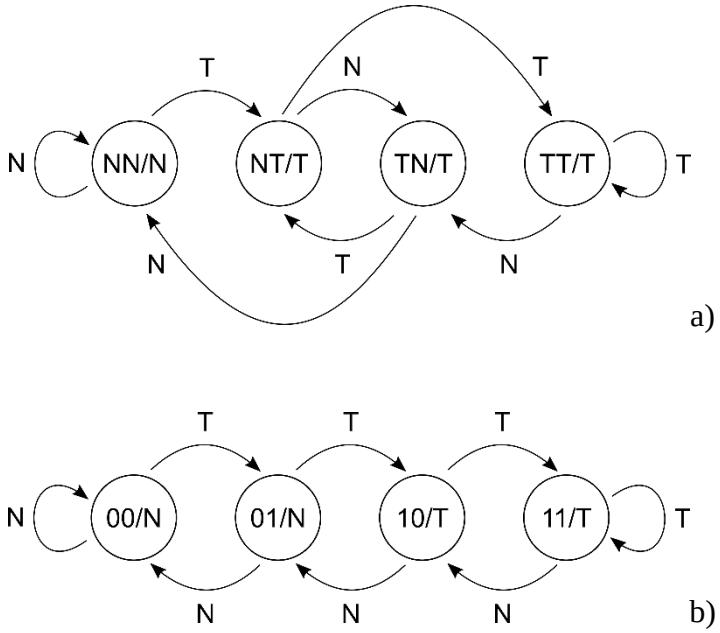


2.39. ábra – Előtörténet jelzése az elágazási célpufferben

A BH frissítésére egy állapotgépet használnak, amelyben az egyes állapotokat a BH lehetséges értékei jelentik. Minden egyes állapothoz rendelünk egy jósolt kimenetelt, amelynek értéke lehet „teljesül” (taken, T) vagy „nem teljesül” (not taken, N). Az állapotátmeneteket a feltétel valódi teljesüléseihez rendeljük.

A 2.40. ábrán két példát is láthatunk ilyen állapotgépre. A 2.40a ábrán az egyes állapotokat az adott ugró utasítás feltételének ténylegesen megvalósult utolsó két kimenetele adja. Megfigyelhető, hogy ez a megoldás az elágazás végrehajtását részesíti előnyben, mivel a négyből három esetben elágazást jósol (az állapotnevekben a „/” jel után feltüntetett érték jelöli a jövendölést). Azt is megfigyelhetjük, hogy hosszabb N vagy T sorozatok esetén néhány ismétlődés után az NN vagy a TT állapotba kerül, ezért például ciklusok esetén hatékonyan működik.

Egy másik, gyakran használt állapotgép telítődő számlálót valósít meg (2.40b ábra). A feltétel teljesülése esetén növeljük, nem teljesülése esetén csökkentjük az adott utasításhoz tartozó 2 bites számlálót. Az alsó két értéknél „nem teljesül”, a felső két értéknél „teljesül” kimenetelt becsül a spekulatív következtetés. Itt is igaz, hogy a hosszabb N vagy T sorozatoknál helyes becsléseket kapunk.



2.40. ábra – Állapotgépek ez elágazási előtörténet nyomon követésére.
a) Az elágazás utolsó két megvalósulásának nyilvántartása. b) Telítődő számláló.

Felmerül a kérdés, hogy milyen állapotgép nyújtja a legnagyobb találati arányt? Hiszen nem csak a fenti kettő, hanem számos egyéb állapotgép is alkotható. Ennek a kérdésnek a megválaszolására 1992-ben kiterjedt szimulációt végeztek az IBM RS/6000 számítógép felhasználásával. Kétbites előtörténeti állapotot feltételezve a lehetséges állapotgépek száma 2^{16} lenne, de a triviálisan érdekteleneket elhagyva 5248 marad. Ezeket vizsgálva a legtöbb

Copyright © 2021 / BME AUT

benchmark program a telítődő számlálós megoldást hozta ki a legjobbnak, 89–94% találati aránnyal. Számos processzor, például az egységes szerkezetű szuperskalár csővezetékkel rendelkező Intel Pentium is telítődő számlálós állapotgépet használ.

A bemutatott állapotgépek előnyös tulajdonságai mellett meg kell említenünk, hogy bizonyos esetekben nagyon le tud romlani a hatékonyságuk. Különösen akkor, ha egy ugró utasítás feltételének egymás utáni megvalósulásai valamilyen rövid, ismétlődő mintázatot követnek. Az alábbi táblázatok ilyen eseteket mutatnak be. Minden esetben a kezdőállapot az NN, illetve 00. Láthatjuk az adott állapotban adott becsléseket és a feltétel tényleges megvalósulását. Mindkét állapotgéphez könnyen találhatunk olyan mintázatokat, amelyeknél a találati arány csak 50% körüli, vagy teljességgel helytelen becsléseket szolgáltatnak. Az ilyen jellegű anomáliák kiküszöbölése érdekében vezették be az adaptív, kétszintű elágazásbecslési módszereket.

Sorszám		1	2	3	4	5	6	7	8	9	**Találati arány**
Utolsó két megvalósulás alapján működő állapotgép	Állapot	NN	NT	TN	NT	TN	NT	TN	NT	TN	~50%
	Jövendölés	**N**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	
Telítődő számlálós állapotgép	Állapot	00	01	00	01	00	01	00	01	00	~50%
	Jövendölés	**N**	**N**	**N**	**N**	**N**	**N**	**N**	**N**	**N**	
Feltétel tényleges megvalósulása		**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	

2.4. táblázat – Egyszintű dinamikus elágazásjövendölés hatékonysága #1

Sorszám		1	2	3	4	5	6	7	8	9	**Találati arány**
Utolsó két megvalósulás alapján működő állapotgép	Állapot	NN	NT	TT	TN	NT	TN	NT	TN	NT	~50%
	Jövendölés	**N**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	
Telítődő számlálós állapotgép	Állapot	00	01	10	01	10	01	10	01	10	0%
	Jövendölés	**N**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	
Feltétel tényleges megvalósulása		**T**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	

2.5. táblázat – Egyszintű dinamikus elágazásjövendölés hatékonysága #2

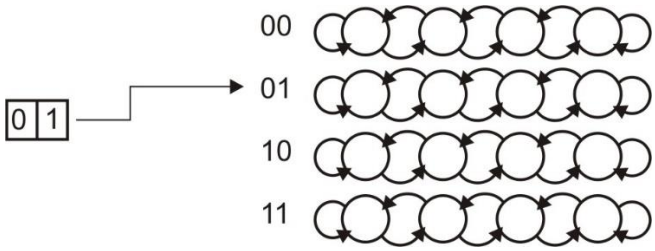
Sorszám		1	2	3	4	5	6	7	8	9	**Találati arány**
Utolsó két megvalósulás alapján működő állapotgép	Állapot	NN	NT	TN	NN	NT	TN	NN	NT	TN	0%
	Jövendölés	**N**	**T**	**T**	**N**	**T**	**T**	**N**	**T**	**T**	
Telítődő számlálós állapotgép	Állapot	00	01	00	00	01	00	00	01	00	~67%
	Jövendölés	**N**	**N**	**N**	**N**	**N**	**N**	**N**	**N**	**N**	
Feltétel tényleges megvalósulása		**T**	**N**	**N**	**T**	**N**	**N**	**T**	**N**	**N**	

2.6. táblázat – Egyszintű dinamikus elágazásjövendölés hatékonysága #3

Copyright © 2021 / BME AUT

Adaptív, kétszintű dinamikus elágazásjövendölés

A kétszintű elágazásprediktor ismétlődő mintázatok esetén is hatékony becslést biztosít. Ezt úgy éri el, hogy emlékszik az adott elágazás utolsó n db megvalósulására (ez az n bites BH érték), és az összes lehetséges (2^n-féle) előtörténethez külön telítődő számlálót alkalmaz. Az elágazási célpuffer minden sorának BH bejegyzéséhez egy Pattern History Table-nek nevezett táblázatot rendel, amelynek sorai egy-egy előtörténetet és a hozzá tartozó állapotgépet reprezentálják. Úgy is tekinthetjük, hogy a 2.40a ábrán látható előtörténet-állapotgép minden egyes állapotához rendelünk egy-egy 2.40b ábrának megfelelő számlálót, és ez utóbbiak állapotai alapján történik a feltétel megvalósulásának becslése. Például 2 bites BH esetén 4 bejegyzés van minden branch utasításhoz, külön-külön számlálóval. A módszer hatékonyságát az adja, hogy nem csupán magához az ugró utasításhoz szolgáltat becslést, hanem az ugrás korábbi megvalósulás-mintázataihoz, amelyeket az egyes előtörténetek reprezentálnak. Mivel a különböző mintázatokhoz külön-külön állapotgépek tartoznak, azok szabályos ismétlődés esetén hamar megtanulják az elágazási feltétel megvalósulásait, és hatékonyan tudják becsülni a későbbiekben, ha a mintázat folytatódik.



2.41. ábra – Pattern History Table

Az adaptív, kétszintű elágazásjövendölés működését a következő táblázatok segítségével követhetjük nyomon. Ezekben az egyszintű dinamikus elágazásjövendölésnél már látott példamintázatok szerepelnek újra. A szekvenciákat hosszabban ábrázoltuk, hogy jobban látható legyen, miként alkalmazkodik a kétszintű állapotgép az egyes mintázatokhoz. A különböző előtörténeteket a könnyebb követhetőség érdekében eltérő háttérszínnel is jelöltük. Megfigyelhetjük, hogy mindegyik esetben néhány kezdeti téves becslés után a jövendölés találati aránya gyakorlatilag 100%-os. Hatékonysága miatt a legtöbb modern processzor ehhez hasonló módszert alkalmaz az elágazások spekulatív kiértékelésére.

Sorszám	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**Találati arány**
Előtörténet	NN	NT	TN	NT	TN	NT	TN	NT	TN	NT	TN	NT	TN	NT	TN	100%-hoz tart
Számláló állapota	00	00	00	00	01	00	10	00	11	00	11	00	11	00	11	
Jövendölés	N	N	N	N	N	N	T	N	T	N	T	N	T	N	T	
Feltétel tényleges megvalósulása	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	

2.7. táblázat – Adaptív, kétszintű dinamikus elágazásjövendölés hatékonysága #1

Copyright © 2021 / BME AUT

Sorszám	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**Találati arány**
Előtörténet	NN	NT	TT	TN	NT	TN	NT	TN	NT	TN	NT	TN	NT	TN	NT	100%-hoz tart
Számláló állapota	00	00	00	00	01	01	00	10	00	11	00	11	00	11	00	
Jövendölés	**N**	**N**	**N**	**N**	**N**	**N**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	
Feltétel tényleges megvalósulása	**T**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	**T**	**N**	

2.8. táblázat – Adaptív, kétszintű dinamikus elágazásjövendölés hatékonysága #2

Sorszám	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**Találati arány**
Előtörténet	NN	NT	TN	NN	NT	TN	NN	NT	TN	NN	NT	TN	NN	NT	TN	100%-hoz tart
Számláló állapota	00	00	00	01	00	00	10	00	00	11	00	00	11	00	00	
Jövendölés	**N**	**N**	**N**	**N**	**N**	**N**	**T**	**N**	**N**	**T**	**N**	**N**	**T**	**N**	**N**	
Feltétel tényleges megvalósulása	**T**	**N**	**N**	**T**	**N**	**N**	**T**	**N**	**N**	**T**	**N**	**N**	**T**	**N**	**N**	

2.9. táblázat – Adaptív, kétszintű dinamikus elágazásjövendölés hatékonysága #3

2.3.4 Függőségek gyakorlati hatása a szuperskalár csővezetékben

A szuperskalár processzorok legfontosabb mikroarchitekturális megoldásait áttekintettük. Ebben az alfejezetben gyakorlati példákon keresztül vizsgáljuk meg, hogyan érhető tetten a szuperskalár működés egyik vagy másik komponense valós programok futtatása esetén. Első példánkban a vezérlési függőség, illetve az ennek feloldását célzó elágazásjövendölés teljesítőképességre gyakorolt hatását vizsgáljuk meg. Egy másik példán keresztül pedig azt figyelhetjük meg, hogyan tudja egy optimalizáló fordítóprogram befolyásolni az utasítások közötti adatfüggőségeket az utasításszintű párhuzamosság kihasználása, és így gyorsabb kód előállítása érdekében.

Copyright © 2021 / BME AUT

2.3.4.1 Elágazásjövendölés hatása a gyakorlatban

Tekintsük a következő egyszerű C nyelvű függvényt:[13]

```
long long sumloop(int* data, unsigned size)
{
    long long sum = 0;

    for (unsigned c = 0; c < size; c++)
    {
        if (data[c] >= 128)
            sum += data[c];
    }
    return sum;
}
```

A függvény bemenetként egy adattömböt kap, amelyet a kezdőcímével és méretével adunk meg. Az elvégzendő feladat annyi, hogy a tömb elemein végigiterálva mindazokat az elemeket összegezze, amelyek egy adott értéknél nagyobbak. A függvény futásidejét kétféle esetben vizsgáltuk egy valós szuperskalár processzoron (Intel Core i5-8250U @ 3,4GHz, Kaby Lake mikroarchitektúra). Első esetben a tömböt random értékekkel töltöttük fel, másodszor pedig a függvény hívása előtt a tömb elemeit sorba rendeztük. A tömb elemeinek száma 32768 volt, és 20000-szer ismételtük meg a hívást, hogy a futásidő pontosabban mérhető és emberi léptékkel is érzékelhető legyen. A mért futásidők a következőképpen alakultak:

- Tömb elemei random értékekkel feltöltve: 4,569 s
- Tömb elemei sorba rendezve: 1,285 s

A jelentős, több, mint háromszoros különbség oka az *if* típusú elágazásnál keresendő a *for* ciklus törzsében. Az elágazási feltétel teljesülése ugyanis a tömb aktuálisan vizsgált elemének értékétől függ. Amikor véletlen számok következtek egymás után, akkor az is véletlenszerű volt, hogy éppen nagyobb-e vagy kisebb az adott elem a küszöbértéknél. Így az elágazásprediktor nem tud a véletlen sorsolásnál hasznosabb tippet adni, hiszen nincs olyan ismétlődő mintázat az elágazási feltétel teljesülésében, amire rá tudna állni. Sorba rendezett tömbelemek esetén viszont egymás után két külön blokkban következnek a teljesülő és a nem teljesülő elágazási feltételek. Így, mivel a *for* ciklus összes iterációja alatt csak egyszer változik az *if* feltétel teljesülése, a dinamikus elágazásprediktor könnyedén előre tudja jelezni a helyes ágat.

Érdekességképpen nézzük meg, hogy a fenti függvényből milyen assembly kódot generál a fordítóprogram, és kövessük végig a működését. A fordításhoz a Microsoft Visual C/C++

[13] A példa ötletét a következő Stack Overflow fórumbejegyzés adta: https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

Compiler 19.27.29111.0 verzióját használtuk kikapcsolt optimalizálással (/Od). A fordítás az x86 ISA 32 bites verziójára (IA-32) történt.

```
; Function compile flags: /Odtp
; File c:\...\BranchPredictorTest\branch\branch.cpp
;   COMDAT ?sumloop@@YA_JPAHI@Z
_TEXT   SEGMENT                                     ; Stack relatív címek:
_sum$ = -12                     ; size = 8          ; lokális változó 1
_c$1 = -4                       ; size = 4          ; lokális változó 2
_data$ = 8                      ; size = 4          ; 1. bemenő paraméter
_size$ = 12                     ; size = 4          ; 2. bemenő paraméter
?sumloop@@YA_JPAHI@Z PROC       ; sumloop, COMDAT

; 7    : {

    push    ebp                                     ; Helyfoglalás a
    mov ebp, esp                                    ; lokális változóknak
    sub esp, 12                 ; 0000000cH         ; a stack-en

; 8    :     long long sum = 0;

    xorps   xmm0, xmm0                              ; 64 bites „0”
    movlpd  QWORD PTR _sum$[ebp], xmm0              ; betöltése gyorsan

; 9    :
; 10   :     for (unsigned c = 0; c < size; c++)

    mov DWORD PTR _c$1[ebp], 0                  ; c = 0
    jmp SHORT $LN4@sumloop                      ; Ugrás a „c < size” vizsgálatra
$LN2@sumloop:
    mov eax, DWORD PTR _c$1[ebp]                ; Ciklusváltozó inkrementálása
    add eax, 1
    mov DWORD PTR _c$1[ebp], eax
$LN4@sumloop:                                   ; „c < size” vizsgálat
    mov ecx, DWORD PTR _c$1[ebp]                ;
    cmp ecx, DWORD PTR _size$[ebp]              ;
    jae SHORT $LN3@sumloop                      ; Jump if above or equal
                                                ; (feltételes ugrás a végére)

; 11   :     {
; 12   :         if (data[c] >= 128)

    mov edx, DWORD PTR _c$1[ebp]                    ; Index változó betöltése
    mov eax, DWORD PTR _data$[ebp]                  ; Tömb eleje pointer betölt.
    cmp DWORD PTR [eax+edx*4], 128   ; 00000080H   ; Tömbelem értékvizsgálat
    jl  SHORT $LN5@sumloop                          ; Feltételes elágazás

; 13   :             sum += data[c];

    mov ecx, DWORD PTR _c$1[ebp]                    ; Index változó betöltése
    mov edx, DWORD PTR _data$[ebp]                  ; Tömb eleje pointer betölt.
    mov eax, DWORD PTR [edx+ecx*4]                  ; Tömb aktuális elem betölt.
```

Copyright © 2021 / BME AUT

```
    cdq                                             ; Előjelkiterjesztés
                                                    ; 64 bitre: EAX -> [EDX:EAX]
    add eax, DWORD PTR _sum$[ebp]                   ; Alsó 32 bit összeadása
    adc edx, DWORD PTR _sum$[ebp+4]                 ; Felső 32 bit összeadása
    mov DWORD PTR _sum$[ebp], eax                   ; Alsó 32 bit tárolása
    mov DWORD PTR _sum$[ebp+4], edx                 ; Felső 32 bit tárolása
$LN5@sumloop:

; 14   :     }

    jmp SHORT $LN2@sumloop                          ; Ugrás a ciklus elejére

$LN3@sumloop:

; 15   :
; 16   :     return sum;

    mov eax, DWORD PTR _sum$[ebp]                   ; Visszatérési érték
    mov edx, DWORD PTR _sum$[ebp+4]                 ; [EDX:EAX] regiszterben

; 17   : }

    mov esp, ebp                                    ; Stack frame
    pop ebp                                         ; visszaállítása és
    ret 0                                           ; visszatérés
?sumloop@@YA_JPAHI@Z ENDP                ; sumloop
_TEXT   ENDS
```

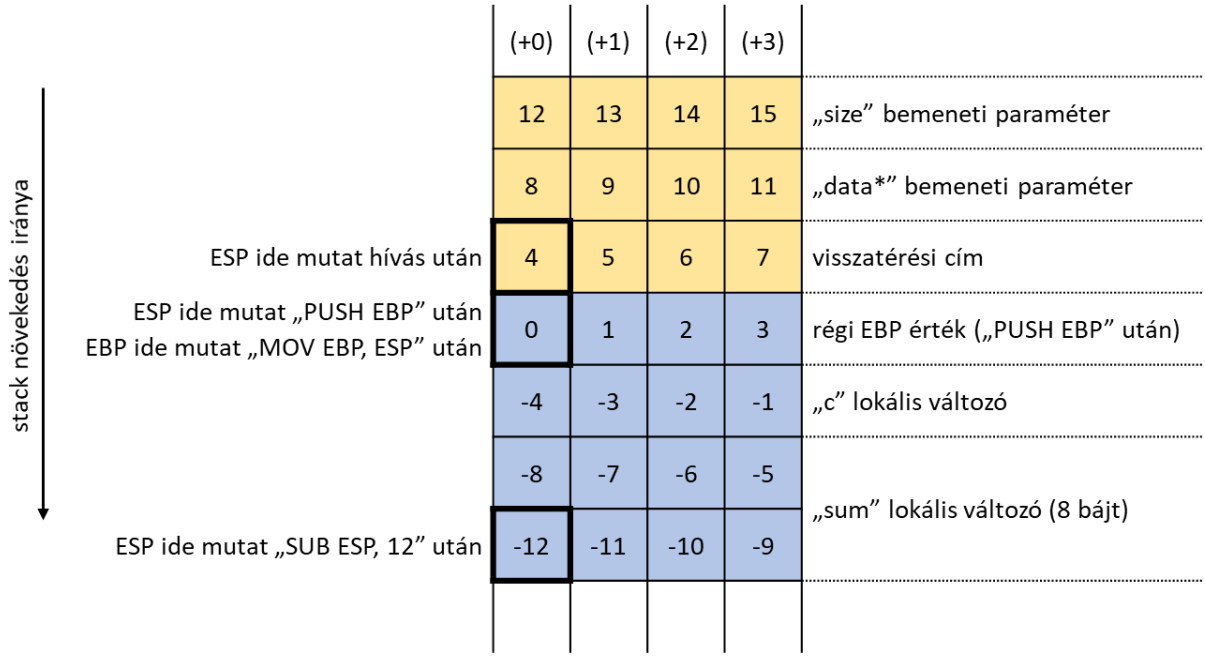
A generált assembly kódban láthatók a fordító által létrehozott megjegyzések, amelyek többek között az egy-egy kódrészletnek megfelelő C programsorokat tartalmazzák a könnyebb beazonosítás érdekében. Emellett a sorok végén több helyen utólag kézzel beírt magyarázó kommenteket is elhelyeztünk.

A kódrészlet elején a C függvény lokális változóinak és bemenő paramétereinek megfelelő szimbólumokat találunk. Ezek relatív mutatóként szolgálnak majd a memóriában tárolt több-bájtos változók eléréséhez. Például, ha egy utasítás egyik operandusa *_sum$[ebp]*, akkor a hivatkozott effektív memóriacím az EBP regiszterben tárolt érték, a *_sum$* szimbólum értékével módosítva (esetünkben 12-vel csökkentve).

Az EBP jelentése az x86 architektúrában Extended Base Pointer, amellyel a stack-en történő helyfoglalást valósítjuk meg a lokális változók számára. Az erre szolgáló, ún. *stack frame* létrehozásában, és egyáltalán a stack kezelésében az EBP mellett az ESP (Extended Stack Pointer) regiszternek van központi szerepe. A példánkban megvalósuló stack kezelés megértéséhez a 2.42. ábra nyújt segítséget, ahol táblázatos formában láthatjuk a függvényünk által kezelt stack területet. Az x86 architektúra memóriakezelése bájtos szervezésű, és a stack a csökkenő címek irányába növekszik. A könnyebb áttekinthetőség érdekében a táblázat minden sorában 4 bájtot ábrázoltunk, mert az eltárolt értékek mérete tipikusan 4 bájt egész számú többszöröse: a bemeneti paraméterek és lokális változók *unsigned int* (4 bájt) és *long long* (8 bájt) típusúak, a memóriacímek és mutatók 32 bitesek. A hívó függvény által írt

Copyright © 2021 / BME AUT

mezőket sárga, a vizsgált függvényünk által kezelt területet kék háttérszínnel jelöltük. Az egyes mezőkben a stack frame elejéhez viszonyított relatív memóriacímeket tüntettük fel.



2.42. ábra – A stack felépítése

A stack pointer (ESP) mindig a legutóbb írt memóriacímre mutat. Ha egy adatot teszünk a stack-re (PUSH), akkor mindig először az ESP lép (dekrementálódik), és utána történik az adat beírása az új címre. Ha egy adatot elveszünk a stack-ről (POP), akkor először kiolvassuk az adatot, majd inkrementáljuk a stack pointert.

Példánkban a hívó fél a bemenő paramétereket a stack-en keresztül adja át a függvényünknek. Hívás előtt elhelyezi a *size* és a *data** paramétert (ez utóbbi egy mutató, azaz 32 bites memóriacím) a stack-re, majd a szubrutinhívás a visszatérési címet is elmenti. Ekkor a stack pointer a visszatérési cím alsó bájtjára mutat, és az utasítások betöltése a függvényünk elején folytatódik (az assembly kódban ezt jelöli a *?sumloop@@YA_JPAHI@Z* címke). A hívott függvény első dolga a stack frame beállítása a lokális változók számára. Ehhez elmenti az EBP regiszter korábbi értékét (*push ebp*), majd eltárolja benne a stack pointer aktuális értékét (*mov ebp, esp*) a későbbi visszaállíthatóság érdekében. A lokális változók elhelyezéséhez 12 bájtnyi hely szükséges (1 db *unsigned int* és 1 db *long long*), ezért a stack pointer értékét további 12-vel csökkenti (*sub esp, 12*). Ekkor az EBP a stack pointer korábbi értékére mutat, miközben maga a stack pointer már 12-vel odébb jár. Következésképp a kettő közötti 12 bájtos tartomány védett a későbbi stack műveletektől, és könnyedén elérhető EBP-relatív címzéssel.

```
      push    ebp                                 ; Helyfoglalás a
      mov ebp, esp                                ; lokális változóknak
      sub esp, 12                 ; 0000000cH     ; a stack-en
```

Copyright © 2021 / BME AUT

A kódban további érdekes részletekre is bukkanhatunk. Ilyen például a 64 bites *sum* lokális változó inicializálása 0 értékre. Mivel a processzor rendelkezik 128 bites SIMD regiszterekkel (*XMMn*) is, amelyeken a SIMD utasítások nagyon gyorsan végrehajthatók (SSE kiegészítés az x86 ISA-hoz), ezért nagyobb bitszámú 0 érték egy ilyen regiszter önmagával vett XOR műveletének eredményeképp hatékonyan állítható elő. Az így keletkezett 128 bites 0 érték alsó 64 bitje pedig egy lépésben eltárolható a memóriában a MOVLPD utasítással (Move Low Packed Double-Precision Floating-Point Value), amely egyébként dupla pontosságú lebegőpontos számok gyors betöltésére és tárolására lett kitalálva, de mint látjuk, más célra is remekül használható.

```
; 8    :     long long sum = 0;

    xorps   xmm0, xmm0                          ; 64 bites „0”
    movlpd  QWORD PTR _sum$[ebp], xmm0          ; betöltése gyorsan
```

A következőkben láthatjuk a *for* ciklus fejlécének megvalósítását, azaz a *c* ciklusszámláló inicializálását (egy „közönséges” MOV utasítással), a kilépési feltétel vizsgálatát (*$LN4@sumloop* címke) és a ciklusszámláló inkrementálását (*$LN2@sumloop* címke). Külön érdekesség, hogy logikailag a ciklusváltozót a ciklusok végén kell inkrementálni, de itt ez a művelet is a ciklustörzs elején kapott helyet. A fordító úgy oldotta meg a ciklusszervezést, hogy minden iteráció végén feltétel nélkül visszaugrik a ciklus elejére, inkrementálja a ciklusváltozót, aztán megvizsgálja a kilépési feltételt, és a kilépést megvalósító feltételes elágazás is itt, a ciklus elején kapott helyet (*jae SHORT $LN3@sumloop*).

```
; 10   :     for (unsigned c = 0; c < size; c++)

    mov DWORD PTR _c$1[ebp], 0                  ; c = 0
    jmp SHORT $LN4@sumloop                      ; Ugrás a „c < size” vizsgálatra
$LN2@sumloop:
    mov eax, DWORD PTR _c$1[ebp]                ; Ciklusváltozó inkrementálása
    add eax, 1
    mov DWORD PTR _c$1[ebp], eax
$LN4@sumloop:                                   ; „c < size” vizsgálat
    mov ecx, DWORD PTR _c$1[ebp]                ;
    cmp ecx, DWORD PTR _size$[ebp]              ;
    jae SHORT $LN3@sumloop                      ; Jump if above or equal
                                                ; (feltételes ugrás a végére)
; 11   :     {

;            ...

; 14   :     }

    jmp SHORT $LN2@sumloop                          ; Ugrás a ciklus elejére
```

Copyright © 2021 / BME AUT

Ezután következik a ciklustörzs megvalósítása. Az *if* elágazás feltételvizsgálatához többszörös indirekcióval kell előállítani az aktuális tömbelem értékét. A tömb kezdőcímét EBP-relatív indirekt címzéssel töltjük be az EAX regiszterbe (*mov eax, DWORD PTR _data$[ebp]*), az index változó értéke pedig szintén EBP-relatív indirekcióval az EDX regiszterbe kerül (*mov edx, DWORD PTR _c$1[ebp]*). Ezután az értékvizsgálathoz skálázott bázis relatív címzéssel kapjuk meg a keresett tömbelem értékét (*cmp DWORD PTR [eax+edx*4], 128*).

```
; 11   :     {
; 12   :         if (data[c] >= 128)

    mov edx, DWORD PTR _c$1[ebp]                ; Index változó betöltése
    mov eax, DWORD PTR _data$[ebp]              ; Tömb eleje pointer betölt.
    cmp DWORD PTR [eax+edx*4], 128   ; 00000080H  ; Tömbelem értékvizsgálat
    jl  SHORT $LN5@sumloop                      ; Feltételes elágazás

; 13   :             sum += data[c];

    mov ecx, DWORD PTR _c$1[ebp]                ; Index változó betöltése
    mov edx, DWORD PTR _data$[ebp]              ; Tömb eleje pointer betölt.
    mov eax, DWORD PTR [edx+ecx*4]              ; Tömb aktuális elem betölt.
    cdq                                         ; Előjelkiterjesztés
                                                ; 64 bitre: EAX -> [EDX:EAX]
    add eax, DWORD PTR _sum$[ebp]               ; Alsó 32 bit összeadása
    adc edx, DWORD PTR _sum$[ebp+4]             ; Felső 32 bit összeadása
    mov DWORD PTR _sum$[ebp], eax               ; Alsó 32 bit tárolása
    mov DWORD PTR _sum$[ebp+4], edx             ; Felső 32 bit tárolása
$LN5@sumloop:

; 14   :     }
```

A tömb elemének hozzáadása a *sum* változóhoz ugyanilyen címzési módokkal történik. A 64 bites összeadást 2 db 32 bites ADD, illetve ADC (Add with Carry) művelettel valósította meg a fordító (emlékezzünk: a 32 bites ISA verzióra fordítottunk). Érdekes megfigyelnünk, hogy a lokális változókon végzett műveletek előtt és után mindig betöltés és tárolás található, regiszterben csak nagyon rövid ideig tartózkodnak a változók értékei.

A függvény törzsének végén a visszatérési érték előkészítését találjuk. Vegyük észre, hogy az eredményt regiszterekben adjuk vissza, nem pedig a stack-en, ahogy a bemenő paraméterek átadása történt. Ugyanis a függvény csak a saját stack frame területére írhat, az pedig hamarosan felszabadításra kerül.

```
; 16   :     return sum;

    mov eax, DWORD PTR _sum$[ebp]               ; Visszatérési érték
    mov edx, DWORD PTR _sum$[ebp+4]             ; [EDX:EAX] regiszterben
```

Copyright © 2021 / BME AUT

Végül a stack frame visszaállítását láthatjuk. Az ESP értéke az EBP-ből állítható vissza (*mov esp, ebp*), az EBP korábbi értéke pedig magáról a stack-ről (*pop ebp*). A legvégén a RET utasítással visszatérünk a hívó függvényhez.

```
mov esp, ebp                                    ; Stack frame
pop ebp                                         ; visszaállítása és
ret 0                                           ; visszatérés
```

2.3.4.2 Optimalizáló fordító hatása

Maradjunk az előző példánkban megvizsgált *sumloop()* függvénynél, de nézzük meg, mi történik, ha a fordítást sebességre optimalizálva végezzük (/O2 mód). A függvényt ugyanazokkal a paraméterekkel, ugyanannyiszor hívjuk meg, mint az előző példánkban. A mért futási időket alább láthatjuk:

- Kikapcsolt optimalizálás mellett (/Od)
  - Tömb elemei random értékekkel feltöltve: 4,569 s
  - Tömb elemei sorba rendezve: 1,285 s
- Sebességre optimalizálva (/O2)
  - Tömb elemei random értékekkel feltöltve: 2,066 s
  - Tömb elemei sorba rendezve: 0,333 s

Láthatjuk, hogy továbbra is nagy hatása van a futásidőre annak, ha a bemeneti adattömb elemeit sorba rendezzük a függvényhívás előtt, hiszen ezzel az elágazásbecslés dolgát könnyítjük meg. Viszont azt is megfigyelhetjük, hogy az optimalizálás nélküli esethez képest több, mint kétszeres gyorsulást érhetünk el a rendezés nélkül is. Ha a sorba rendezést elvégezzük a függvényhívás előtt, akkor az optimalizált kód közel 4-szer gyorsabban fut, mint az optimalizálás nélküli.

Mi lehet az oka ennek a jelentős sebességnövekedésnek? Nem célunk az optimalizáló fordítási technikák részletes ismertetése, de két szembetűnő változást észrevehetünk, ha végignézzük az így létrejött assembly kódot. Hamarosan meg is tesszük ezt, de előtte röviden foglaljuk össze a két eljárást, amit tetten érhetünk a példakódunkban is.

Az első lényeges különbség, hogy ha a bemenő paramétereket és a lokális változókat nem memóriában (stack-en), hanem regiszterben adjuk át, és a függvény élettartama alatt ott is tartjuk, akkor nagyon sok töltési és tárolási műveletet megspórolhatunk (az összetett indirekt címképzésről nem is beszélve). A másik technika, amelyet a fordítók az utasításszintű párhuzamosság jobb kiaknázása érdekében gyakran alkalmaznak, a ciklusok kiterítése (loop unrolling). Nézzük meg, mit jelent ez a fogalom, mielőtt a konkrét példánkban megfigyelnénk a hatását.

Copyright © 2021 / BME AUT

Ciklusok kiterítése (loop unrolling)

Ciklusszervezésnél gyakran előfordul, főleg rövid ciklustörzsek esetén, hogy az egymás után következő iterációk függőségei miatt alacsony az utasításszintű párhuzamosság kihasználhatósága. Sorrenden kívüli végrehajtásra képes processzoroknál az utasításfolyamba való viszonylag mély előretekintés következtében egyszerre több iteráció utasításai is végrehajtás alatt lehetnek (illetve az utasításpufferben várakozhatnak). Ez egyrészt a regiszterek dinamikus újrahasználatából következő hamis adatfüggőségekhez vezet, ami regiszter-átnevezéssel bizonyos szintig kezelhető. Másrészt viszont a rövid ciklusmagok nagy számú valódi adatfüggőséget is okozhatnak, például egy *for* ciklus esetén, ha az iterációnként inkrementált ciklusváltozót a ciklustörzsben is felhasználjuk. Ilyenkor a ciklusváltozó inkrementálása, mint írás és a következő iterációban történő felhasználása RAW típusú adatfüggőséget eredményez. Emlékezzünk vissza, hogy a RAW függőségek szigorúan sorrendi végrehajtást követelnek meg, ami nagy mennyiségben előfordulva jelentősen le tudja rontani egy szuperskalár processzor hatékonyságát.

A loop unrolling módszer ezt a hatást hivatott csökkenteni. Lényege, hogy egy N iterációból álló ciklust egy N/M iterációs ciklussá alakítja úgy, hogy az új ciklusban minden egyes iteráció az eredeti ciklus M iterációjának feladatát végzi el. Ezzel megnő a ciklusszervező elágazások közötti utasítások száma, és lecsökken a ciklusszervezésből adódó adatfüggőségek mennyisége. Így a processzor végrehajtó egységének több lehetősége adódik az utasítások párhuzamosítására. Ráadásul a kevesebb ugrás miatt kisebb lesz az összesen végrehajtandó utasítások száma is. Lássunk egy példát 4-szeres cikluskibontásra:

```
for (unsigned i = 0; i < 100; i++)
{
    a[i] = b[i] + c[i];
}
```

Ez a ciklus két tömböt ad össze elemenként, és az eredményt egy harmadik tömbbe írja. Kiterítés után az új ciklustörzs négy eredeti iteráció feladatát látja el:

```
for (unsigned i = 0; i < 100; i+=4)
{
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
}
```

Vegyük észre, hogy az *i* ciklusváltozó végértéke továbbra is 100 maradt, de iterációnként nem eggyel, hanem néggyel növekszik az értéke. A nagyobb fokú párhuzamosíthatóság abból adódik, hogy míg az első esetben a ciklustörzs egyetlen *a[i] = b[i] + c[i]* művelete mindig függ az *i++* művelettől, így minden összeadás sorban kell álljon, addig a kiterített ciklusban a négy

Copyright © 2021 / BME AUT

összeadás párhuzamosítható, ha megvan az új *i* érték. A kétféle ciklus végrehajtási szekvenciáját és párhuzamosítási lehetőségeit az alábbi pszeudokód szemlélteti:

```
Eredeti ciklus végrehajtása:

...

i++;                    ┐ RAW
a[i] = b[i] + c[i];     ▼

i++;                    ┐ RAW
a[i] = b[i] + c[i];     ▼

i++;                    ┐ RAW
a[i] = b[i] + c[i];     ▼

i++;                    ┐ RAW
a[i] = b[i] + c[i];     ▼

i++;                    ┐ RAW
a[i] = b[i] + c[i];     ▼

i++;                    ┐ RAW
a[i] = b[i] + c[i];     ▼

i++;                    ┐ RAW
a[i] = b[i] + c[i];     ▼

i++;                    ┐ RAW
a[i] = b[i] + c[i];     ▼

...
```

```
Kiterített ciklus végrehajtása:

...

i+=4;                   ┐ RAW
a[i] = b[i] + c[i];     ▼
a[i+1] = b[i+1] + c[i+1];
a[i+2] = b[i+2] + c[i+2];
a[i+3] = b[i+3] + c[i+3];

i+=4;                   ┐ RAW
a[i] = b[i] + c[i];     ▼
a[i+1] = b[i+1] + c[i+1];
a[i+2] = b[i+2] + c[i+2];
a[i+3] = b[i+3] + c[i+3];

...
```

Lássuk, hogyan történik mindez a korábbi példafüggvényünk esetében. Az egyszerűség kedvéért újra szerepeltetjük a függvény eredeti alakját is:

```
// size = 32768 (minden hívásnál)

long long sumloop(int* data, unsigned size)
{
    long long sum = 0;

 for (unsigned c = 0; c < size; c++)
    {
        if (data[c] >= 128)
            sum += data[c];
    }
 return sum;
}
```

Copyright © 2021 / BME AUT

Az optimalizáló fordító a következőképpen alakítja át a kódunkat:

```
long long sumloop(int* data/*, unsigned size*/)
{
    long long sum = 0;

 unsigned c = 2;
    for (unsigned p = 8196; p > 0; p--)
    {
        if (data[c-2] >= 128)
            sum += data[c-2];

        if (data[c-1] >= 128)
            sum += data[c-1];

        if (data[c] >= 128)
            sum += data[c];

        if (data[c+1] >= 128)
            sum += data[c+1];

        c += 4;
    }
 return sum;
}
```

Vegyük észre, hogy a fordító kihasználja azt a tényt, hogy a hívó példaprogramban a *sumloop* függvénynek bemenetként átadott *size* érték már fordítási időben ismert (32768), így a paraméterátadást is megspórolja. Természetesen a fenti módosított C függvény a valóságban nem jön létre, ezt csak szemléltetésképpen szerepeltettük. A fordító most is assembly kódot generál, de a loop unrolling eredményét könnyebben át lehet látni a fenti tömörebb formában.

Vizsgáljuk meg ezek után a fenti funkcionalitást megvalósító, optimalizált assembly kódot:

```
; Function compile flags: /Ogtp
; File c:\...\BranchPredictorTest2\branch2\branch2.cpp
;   COMDAT ?sumloop@@YA_JPAHI@Z
_TEXT   SEGMENT                                 ; Stack relatív címek
_sum$ = -8                  ; size = 8          ; lokális változó 1
?sumloop@@YA_JPAHI@Z PROC   ; sumloop, COMDAT   ; (a másik regiszterben lesz)
; _data$ = ecx                                  ; (1. bem. param. regiszterben)
; _size$dead$ = edx                             ; (2. bem. param. konstans!)

; 7    : {

    push    ebp                                 ; Helyfoglalás a
    mov ebp, esp                                ; lokális változóknak a
    sub esp, 12             ; 0000000cH         ; stack-en
                                                ; Később használt regiszterek
    push    ebx                                 ; mentése (EBX: tömbméret)
```

Copyright © 2021 / BME AUT

```
    push    esi                                 ; (ESI: „sum” másolat MSDW)
    xorps   xmm0, xmm0                          ; 64 bites „0” gyártása

; 8    :     long long sum = 0;

    add ecx, 8                                  ; ECX a data[2]-re mutasson
    movlpd  QWORD PTR _sum$[ebp], xmm0          ; sum = 0 (64 bites betöltés)
    mov ebx, 8192             ; 00002000H       ; Tömbméret: 32768/4 (konstans)
    mov esi, DWORD PTR _sum$[ebp+4]             ; „sum” másolat MSDW betöltése
    push    edi                                 ; (EDI: „sum” másolat LSDW)
    mov edi, DWORD PTR _sum$[ebp]               ; „sum” másolat LSDW betöltése
    npad    1                                   ; Alignment (NOP)

$LL9@sumloop:
; 11   :     {
; 12   :         if (data[c] >= 128)
    mov eax, DWORD PTR [ecx-8]                  ; Tömbelem betöltése
    cmp eax, 128                ; 00000080H     ; Tömbelem értékvizsgálat
    jl  SHORT $LN10@sumloop                     ; Feltételes elágazás
; 13   :             sum += data[c];
    cdq                                         ; Előjelkiterjesztés
                                                ; 64 bitre: EAX -> [EDX:EAX]
    add edi, eax                                ; Alsó 32 bit összeadása
    adc esi, edx                                ; Felső 32 bit összeadása

$LN10@sumloop:
; 11   :     {
; 12   :         if (data[c] >= 128)
    mov eax, DWORD PTR [ecx-4]                  ; Tömbelem betöltése
    cmp eax, 128                ; 00000080H     ; Tömbelem értékvizsgálat
    jl  SHORT $LN17@sumloop                     ; Feltételes elágazás
; 13   :             sum += data[c];
    cdq                                         ; Előjelkiterjesztés
    add edi, eax                                ; Alsó 32 bit összeadása
    adc esi, edx                                ; Felső 32 bit összeadása

$LN17@sumloop:
; 11   :     {
; 12   :         if (data[c] >= 128)
    mov eax, DWORD PTR [ecx]                    ; Tömbelem betöltése
    cmp eax, 128                ; 00000080H     ; Tömbelem értékvizsgálat
    jl  SHORT $LN15@sumloop                     ; Feltételes elágazás
; 13   :             sum += data[c];
    cdq                                         ; Előjelkiterjesztés
    add edi, eax                                ; Alsó 32 bit összeadása
    adc esi, edx                                ; Felső 32 bit összeadása

$LN15@sumloop:
; 11   :     {
; 12   :         if (data[c] >= 128)
    mov eax, DWORD PTR [ecx+4]                  ; Tömbelem betöltése
    cmp eax, 128                ; 00000080H     ; Tömbelem értékvizsgálat
    jl  SHORT $LN16@sumloop                     ; Feltételes elágazás
; 13   :             sum += data[c];
```

Copyright © 2021 / BME AUT

```
    cdq                                     ; Előjelkiterjesztés
    add edi, eax                            ; Alsó 32 bit összeadása
    adc esi, edx                            ; Felső 32 bit összeadása

$LN16@sumloop:
; 9    :
; 10   :     for (unsigned c = 0; c < size; c++)
    add ecx, 16                 ; 00000010H     ; Adat pointer léptetése
    sub ebx, 1                              ; Ciklusszámláló dekrementálása
    jne SHORT $LL9@sumloop                  ; Kilépési feltételvizsgálat
                                            ; és ugrás a ciklus elejére

; 14   :     }
; 15   :
; 16   :     return sum;

    mov eax, edi                            ; Visszatérési érték
    mov edx, esi                            ; [EDX:EAX] regiszterben

; 17   : }

    pop edi                                 ; Mentett regiszterek
    pop esi                                 ; visszaállítása
    pop ebx                                 ;
    mov esp, ebp                            ; Stack frame
    pop ebp                                 ; visszaállítása és
    ret 0                                   ; visszatérés
?sumloop@@YA_JPAHI@Z ENDP       ; sumloop
_TEXT   ENDS
```

A kódrészlet elején ezúttal is a lokális változók és a bemeneti paraméterek felsorolását találjuk. Ugyanakkor feltűnhet, hogy a bementő paraméterek szimbólumdefiníciója csak kommentben szerepel, jelezve az olvasó felé, hogy regiszterekben történik a paraméterátadás. Ráadásul a *size* paraméter esetében látható *$dead$* jelölés azt is mutatja, hogy ezt a paramétert a fordító kioptimalizálta. Továbbá az is észrevehetjük, hogy a szimbólumdefiníciók között nem találjuk a *c* indexváltozó nevét. Ugyanis a tömbelemek elérését nem az előző példában látott skálázott bázis relatív címzéssel (pl. *[eax+edx*4]*) oldja meg a program, hanem az egyszerűbb, ofszettel ellátott bázis regiszter címzéssel (pl. *[ecx+4]*) úgy, hogy a kezdetben a *data* tömb első elemére mutató ECX regiszter értékét iterációnként folyamatosan növeli. Ciklusszámlálónak szintén egy regisztert használ (EBX), melynek értékét kezdetben 8196-ra állítja be, majd iterációnként mindig eggyel csökkenti.

A stack alakulását a 2.43. ábra szemlélteti. Látható, hogy most a hívó fél nem helyezett el semmit a stack-en a visszatérési címen kívül. A stack frame beállítása ugyanúgy történik, mint az előző példában, de most csak a *sum* változó található itt.[14] Mivel most sokkal több változót tárolunk regiszterekben, mint az optimalizálatlan változatban, ezért több regisztermentést találunk a függvény elején (az EBX, ESI és EDI regiszterek is mentésre kerülnek). Ezek közül

[14] Bár a *sum* változó elfér 8 bájton, mégis 12-vel lépteti előre a stack pointert ez a programváltozat is. Így 4 bájt kihasználatlan marad a stack frame-ben.

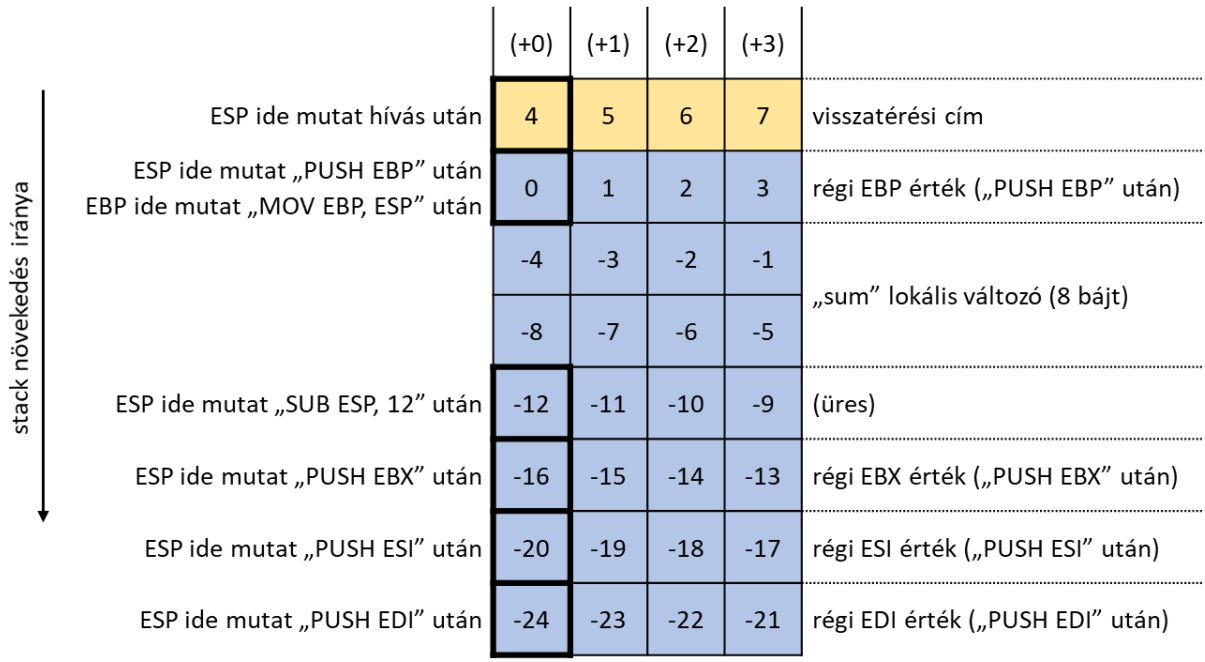
Copyright © 2021 / BME AUT

az ESI és EDI szerepéről nem esett még szó. Ez a két 32 bites regiszter fog szolgálni a *sum* értékének kumulálására és tárolására az iterációk folyamán. Érdekesség, hogy a stack-en a *sum* változónak lefoglalt 8 bájtos területet csak a változó inicializálásához használja a program, az oda elhelyezett 64 bitnyi nullát mindjárt az elején áttölti az ESI:EDI regiszterpárba, és ezekben történik minden további, a *sum* értékét érintő műveletvégzés. A függvény inicializáló része tehát a következő:

```
        push    ebp                             ; Helyfoglalás a
        mov ebp, esp                            ; lokális változóknak a
        sub esp, 12                 ; 0000000cH ; stack-en
                                                ; Később használt regiszterek
        push    ebx                             ; mentése (EBX: tömbméret)
        push    esi                             ; (ESI: „sum” másolat MSDW)
        xorps   xmm0, xmm0                      ; 64 bites „0” gyártása

; 8     :     long long sum = 0;

        add ecx, 8                              ; ECX a data[2]-re mutasson
        movlpd  QWORD PTR _sum$[ebp], xmm0      ; sum = 0 (64 bites betöltés)
        mov ebx, 8192               ; 00002000H ; Tömbméret: 32768/4 (konstans)
        mov esi, DWORD PTR _sum$[ebp+4]         ; „sum” másolat MSDW betöltése
        push    edi                             ; (EDI: „sum” másolat LSDW)
        mov edi, DWORD PTR _sum$[ebp]           ; „sum” másolat LSDW betöltése
        npad    1                               ; Alignment (NOP)
```



2.43. ábra – A stack felépítése kódoptimalizálás esetén

Copyright © 2021 / BME AUT

Ezek után következik a kiterített ciklus. A ciklustörzsben a következő kódrészlet ismétlődik négyszer egymás után, nagyon kis eltéréssel:

```
$LL9@sumloop:
; 11   :     {
; 12   :         if (data[c] >= 128)
    mov eax, DWORD PTR [ecx-8]                  ; Tömbelem betöltése
    cmp eax, 128                ; 00000080H     ; Tömbelem értékvizsgálat
    jl  SHORT $LN10@sumloop                     ; Feltételes elágazás
; 13   :             sum += data[c];
    cdq                                         ; Előjelkiterjesztés
                                                ; 64 bitre: EAX -> [EDX:EAX]
    add edi, eax                                ; Alsó 32 bit összeadása
    adc esi, edx                                ; Felső 32 bit összeadása
```

Az aktuális tömbelemet az EAX regiszterbe töltjük ofszettel ellátott bázis regiszter címzéssel (*mov eax, DWORD PTR [ecx-8]*), ahol a tömbindex négy változata rendre ECX–8, ECX–4, ECX és ECX+4 egy iteráción belül. Ezután következik a feltételes elágazás, majd a *sum* érték kumulálása az ESI:EDI regiszterpárban. A ciklustörzs végén helyezkedik el az adatpointer (ECX) léptetése, a ciklusszámláló (EBX) dekrementálása és a kilépési feltétel vizsgálata:

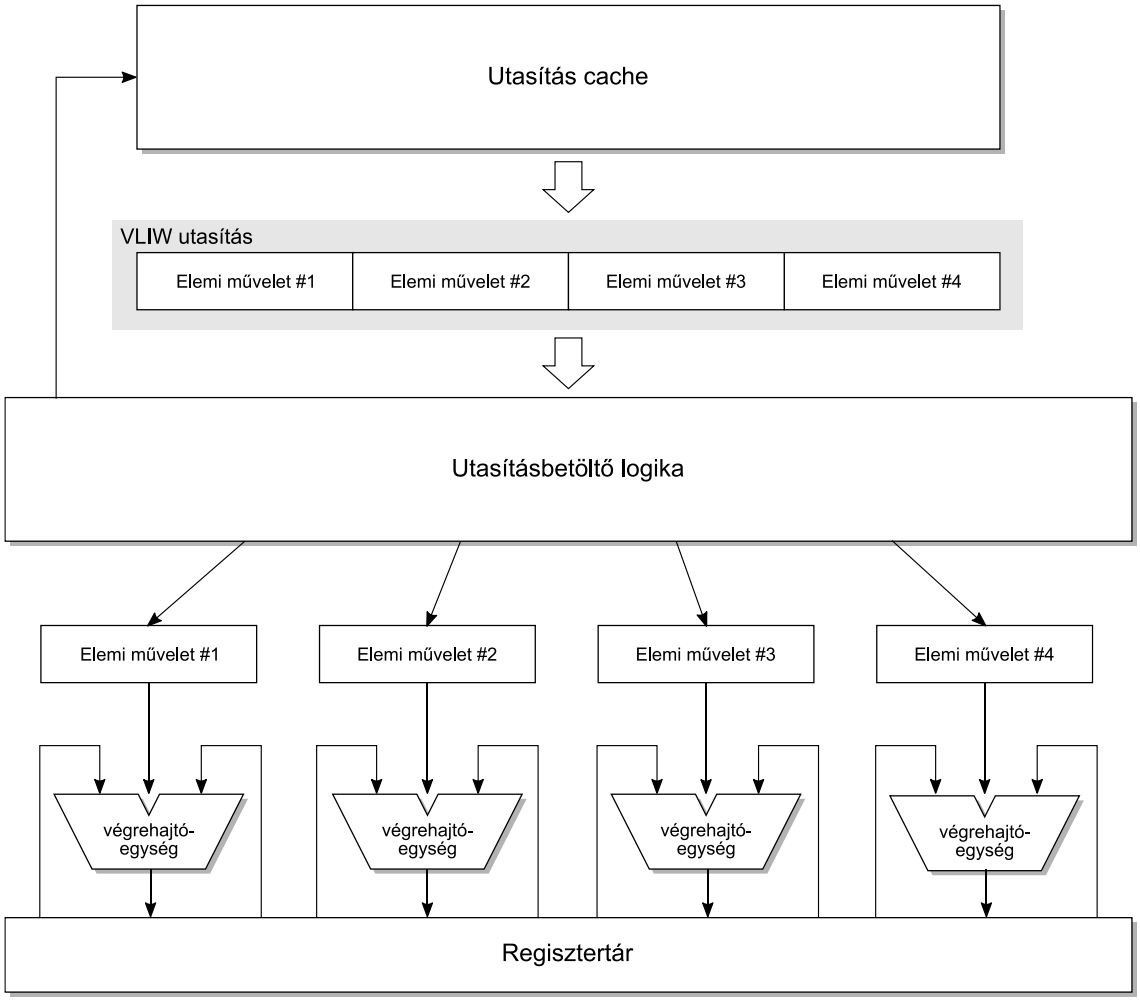
```
$LN16@sumloop:
; 9    :
; 10   :     for (unsigned c = 0; c < size; c++)
    add ecx, 16                 ; 00000010H     ; Adat pointer léptetése
    sub ebx, 1                                  ; Ciklusszámláló dekrementálása
    jne SHORT $LL9@sumloop                      ; Kilépési feltételvizsgálat
                                                ; és ugrás a ciklus elejére
```

Figyeljük meg, hogy az ECX írása iterációnként egyszer történik, olvasására viszont 4-szer kerül sor a kiterített ciklustörzsben. Az előző iteráció végi ECX írás és az aktuális iteráció 4 db ECX használata között ugyan RAW típusú adatfüggőség áll fenn, de a 4 db olvasás akár párhuzamosan is elindítható, amint a legutóbbi ECX frissítés megtörtént. A *sum* érték kumulálása miatt további RAW adatfüggőségek is fennállnak az ADD és ADDC utasítások négy példánya között az EDI és ESI regiszterek kapcsán a ciklustörzsön belül, és ezért a négy összeadás csak sorrendben hajtható végre. Ugyanakkor a memóriabetöltések párhuzamosíthatósága hozzájárul a sebességnövekedéshez.

Copyright © 2021 / BME AUT

2.4 VLIW processzorok

Az utasításszintű párhuzamosság kihasználására a szuperskalár architektúrákon kívül egy másik megközelítés is elterjedt, a nagyon hosszú utasításszavakkal (very long instruction word – VLIW) működő processzorarchitektúrák. Míg a szuperskalár processzorok egy tisztán szekvenciális, skalár processzorok által is végrehajtható gépi kódból hardver támogatással azonosítják a párhuzamosítható részeket, vagyis a független(íthető) utasításokat, addig a VLIW megközelítés ezt a felelősséget az assembly programozóra, illetve a fordítóra hárítja. A hosszú utasításszó, másnéven MultiOp tulajdonképpen olyan elemi utasításokból összeállított kombinált utasítás, amelynek részei egy skalár (RISC) processzor különálló utasításainak felelnek meg. Egy VLIW utasítás annyi elemi utasítást tartalmaz, ahány független párhuzamos végrehajtóegység található a processzorban. Az egyes végrehajtóegységek feladattal való ellátása és minél hatékonyabb kihasználása a fordító feladata, amely a hosszú utasításszavak összeállítását végzi. Ezáltal a hardver jelentősen egyszerűsödik, hiszen nincs szükség bonyolult utasításkiosztó és visszarendező egységekre. A VLIW processzorok programsorrendben hajtják végre az utasításokat, hardver szinten nem szólnak bele az elemi utasítások ütemezésébe.



2.44. ábra – VLIW processzor működési elve

Copyright © 2021 / BME AUT

2.4.1 Történet

A VLIW processzorok története az 1970-es, ’80-as évek fordulója körül kezdődött, nagyjából a RISC architektúrák kialakulásával egyidőben. Az akkori CISC gépek vezérlőművei megvalósításában már elterjedten alkalmazták a mikroprogramozás módszerét, amelynek lényege, hogy az összetett gépi utasításokat a processzorban egyszerű, elemi mikroutasítások szekvenciáival helyettesítették, így a tényleges végrehajtóegységeket kevesebbféle művelet elvégzésére kellett csak felkészíteni. Az egyes (makro)utasításokat megvalósító mikrokódokat ROM-ban megvalósított mikroprogramtárban tárolták a processzoron belül.

A mikrokódoknak két fajtája terjedt el. A *horizontális mikrokód* hosszú (akár több, mint 100 bites) mikroutasításszavakat használt, amelyben az egyes bitmezők közvetlen vezérlőjeleket biztosítottak a végrehajtómű funkcionális egységeinek. Az egyes funkcionális egységek párhuzamosan is működhettek, hiszen egy mikroutasításon belül is független bitcsoportok vezérelték őket. A *vertikális mikrokódban* ezzel szemben a mikroutasítások kódolva vannak, azaz a bitmezők egy további dekódolási fázison esnek át, mire vezérlőjelek lesznek belőlük. Vertikális mikrokód esetén egy mikroutasítás jellemzően csak egy funkcionális egységet vezérel, ezért több mikroutasításból álló, hosszabb szekvenciákra van szükség ugyanannak a (makro)utasításnak a leírásához. Ez a megközelítés kisebb méretű mikroprogramtárat igényelt, annak az árán, hogy több órajel kellett egy (makro)utasítás végrehajtásához.

Bizonyos szempontból tekinthetjük úgy is, hogy a RISC processzorok utasításkészlete a vertikális mikrokóddal, míg a VLIW processzorok hosszú utasításszavai a horizontális mikrokóddal mutatnak hasonlóságokat. Vagyis a fordítóprogramok fejlődésével az 1980-as évekre már hatékonyabbnak tűnt a hardver helyett a fordítóra bízni a komplex műveletek elemi műveletekre bontását, illetve a VLIW esetén ezek párhuzamosítását is. Így egyszerűsödött a hardver, ami nagyobb órajelfrekvenciát, és összességében nagyobb teljesítőképességet jelentett.

Az első VLIW architektúrát Joseph Fisher, a Yale Egyetem kutatója alkotta meg kollégáival. Az 1970-es években egy emulátor létrehozásán dolgozott a ’60-as évek sikeres CISC gépéhez, a CDC-6600-hoz. Mivel nagyon nehézkes volt a 64 bites szavakból álló horizontális mikrokód kézzel történő létrehozása és karbantartása, ezért kidolgozott egy olyan módszert, amely egy rövid mikroutasításokból álló, szekvenciális mikroprogramból képes volt hosszú, horizontális mikroutasításokat generálni. Hamar felismerte, hogy a „trace scheduling”-nak elnevezett módszer nem csak mikroprogramok esetén használható, hanem erre alapozva létrehozható egy olyan fordítóprogram, amely szekvenciális kódból párhuzamosan végrehajtható, kombinált utasításokat állít össze egy erre alkalmas, párhuzamos architektúra számára. A kutatások eredményeképpen jött létre az első kísérleti VLIW processzor ELI-512 néven (Enormously Longword Instructions, 512 bites utasításokkal, 16 párhuzamos végrehajtóegységgel), valamint a Bulldog névre keresztelt első ilyen fordítóprogram.

Fisher 1984-ben megalapította a VLIW-alapú számítógépek fejlesztésére és gyártására szakosodott Multiflow nevű cégét. Ezzel egyidőben egy másik cég, a Cydrome is megalakult. A következő években mindkét vállalat piacra dobott néhány modellt, de üzletileg egyik sem lett sikeres, 1990-re mindkettő bezárta kapuit. Mindkét cég alapítója később a Hewlett-Packard munkatársa, és az ott zajló architekturális fejlesztések vezetője lett. A HP 1989-ben kezdte el

Copyright © 2021 / BME AUT

az Explicitly Parallel Instruction Computing (EPIC) VLIW architektúra fejlesztését, amelyhez 1994-ben az Intel is csatlakozott. Az Intel ekkor már tudta, hogy hamarosan szükség lesz a 32 bites x86 (IA-32) utasításkészlet-architektúráról 64 bitesre átállni. Az újítást az x86 bővítése helyett egy teljesen újragondolt, nagyteljesítményű architektúrával akarta meglépni. Így hozták létre a HP-vel karöltve az EPIC-re alapozva az IA-64, másnéven Itanium architektúrát, amelyet elsősorban szerveralkalmazásokra szántak, és amelynek első processzora (Merced) 2001-ben jelent meg. A kezdeti ambiciózus tervekhez képest a fejlesztés lassabban haladt (a Merced megjelenését eleinte 1998-ra tervezték), és az első modell nem is váltotta be a számítási teljesítményéhez fűzött reményeket. Az egy évvel később kiadott Itanium 2 processzor sok hiányosságot orvosolt, de a teljesen új architektúra miatt a szoftvertámogatottság lassan kezdett csak felépülni. Mindeközben az AMD, amely az Intel licensze alapján már régóta x86-alapú processzorokat gyártott, kifejlesztette az x86 utasításkészlet-architektúra 64 bites, de visszafelé teljesen kompatibilis változatát (AMD64, x86-64), és 2003-ban piacra dobta az erre épülő Opteron processzort. Az Itanium kezdeti gyermekbetegségei és szoftvertámogatottságának hiánya a vásárlókat az x86-64 architektúra felé terelte. Az Intel maga is átvette az x86-64 architektúrát (Intel 64 néven), és 2004-től kezdve megjelentek az erre épülő processzorai. Az Itanium a rengeteg belefektetett fejlesztés ellenére nem lett kiugró siker: összesen 5 generációja jelent meg (az utolsó 2017-ben), és 2019 elején az Intel bejelentette, hogy nem folytatják a sorozatot.

Az Itanium története is jól illusztrálja, hogy a VLIW architektúra nem talált utat az általános célú processzorok világába, hiszen ott a kompatibilitás igénye sok más tényezőt felülír. Ennek ellenére a nagyteljesítményű beágyazott rendszerek speciális processzorai között vannak sikeres implementációk, úgymint a Texas Instruments C6000 jelprocesszor (DSP) családja, az Analog Devices SHARC (Super Harvard Achitecture Single-Chip Computer) DSP családja, valamint az ST Microelectonics ST200 sorozatú médiaprocesszorai. A VLIW architektúra egy időben kedvelt volt a grafikus processzorokban (AMD TeraScale mikroarchitektúra, 2007-2012 között), és megjelenik mai, modern system-on-chip rendszerekben is (Xilinx Versal platform).

2.4.2 Előnyök és hátrányok

A VLIW architektúrák előnye az egyszerűbb hardveres megvalósítás, így a kisebb tranzisztorszám miatt nagyobb órajelfrekvenciát lehet elérni, illetve a lapkán megspórolt helyet például további párhuzamos végrehajtóegységek elhelyezésére lehet használni. Mindkét esetben teljesítménynövekedés érhető el a szuperskalár kialakításhoz képest.

A szuperskalár processzoroknál szükségképpen viszonylag kicsi az az ablak, amiben az utasítások függetlenségének analízise zajlik futásidőben. Azzal, hogy az utasítások vizsgálatának és ütemezésének feladatát fordítási időben végezzük, nincs ilyen, lokalitásból adódó hátrány, tehát elméletileg globálisan jobb teljesítmény érhető el.

A VLIW legkomolyabb nehézsége, hogy nehezen tartható meg a kompatibilitás az egymást követő családtagok között. Ha például a végrehajtóegységek száma vagy típusa megváltozik a mikroarchitektúrában, akkor az előző családtag tárgykódja nem futtatható már, újra kell fordítani. Ha a végrehajtóegységek száma és típusa nem is változik, de a felépítésük, sebességük igen, akkor is szükség van az újrafordításra, mert az egy MultiOp-ba került elemi

Copyright © 2021 / BME AUT

utasítások függetlensége sérülhet. Például, ha két művelet között WAR függőség áll fenn, akkor a végrehajtási idők ismeretében a fordító ütemezheti őket egyszerre vagy akár az eredeti sorrendjüktől eltérően is, mindaddig, amíg a logikailag előbb lévő művelet olvasási fázisa a másik írási fázisa előtt lezajlik. Azonban, ha a végrehajtási idők egy architektúra frissítés következtében megváltoznak, akkor előfordulhat, hogy az eredeti ütemezésben a WAR függőség már hazárdot okoz, és a régi tárgykód bár futásképes, de hibás működésű lehet az új architektúrán. Az új mikroarchitektúra ismeretében a fordító feladata (újra) vizsgálni a függőségeket, és szükség esetén átrendezni az elemi utasításokat.

Mivel egy tipikus programban nincs mindig elegendő független művelet ahhoz, hogy minden ciklusban minden végrehajtóegységet el lehessen látni feladattal, szükség van elemi NOP műveletek beszúrására az utasításkódba a tétlen végrehajtóegységek számára. Az ilyen NOP-ok száma feladattól függően elég nagy is lehet, ami fölöslegesen meg tudja növelni a tárgykód méretét, és rontja a memória (még inkább az utasítás cache) kihasználtságát. Ennek enyhítésére léteznek tömörített utasításkészletű VLIW architektúrák is, amelyeknél a tétlen végrehajtóegységeknek szánt NOP-okat nem kell explicit módon jelezni az utasításkódban.

További nehézség, hogy nem minden időzítés határozható meg kellő pontossággal fordítási időben. Load/store jellegű műveleteknél nem lehet előre megmondani, hogy mennyi ciklus szükséges a végrehajtásukhoz, hiszen ez attól függ, hogy a keresett adat a benne van-e a gyorsítótárban, vagy sem. Gyorsítótárhiány (cache miss) esetén a sorrendi végrehajtás miatt nem lehet mást tenni, mint felfüggeszteni a végrehajtást addig, amíg a memóriareferens utasítás be nem fejeződik. Ez pedig, mint korábban is láttuk, az egyik leginkább kerülendő esemény a párhuzamos csővezetékekkel rendelkező processzorok esetében.

Copyright © 2021 / BME AUT

2.5 Többszálúság egy processzoron belül

Az utasításszintű párhuzamosság (ILP) különböző formáinak áttekintése után ebben a fejezetben egy magasabb szintű párhuzamosítási megközelítést, a többszálú program-végrehajtást járjuk körül. Nem kíséreljük meg a problémakör teljes feldolgozását, sokkal inkább a teljesítőképesség növelésében betöltött szerepe szempontjából igyekszünk megmutatni a többszálúság különböző fajtáit és legfontosabb sajátosságait.

Az elvégzendő feladatok független folyamatok és különálló programszálak segítségével történő parallelizálása nagyon fontos szerepet tölt be a többtaszkos és többfelhasználós operációs rendszerek működésében. Akár egyetlen program esetén is elengedhetetlen tud lenni a többszálú megvalósítás, gondoljunk csak az ablakban futó alkalmazásokra, ahol külön programszál kezeli a felhasználói felületet és a háttérben történő adatfeldolgozást. A szálak ütemezése, a közöttük történő kontextusváltás az operációs rendszer feladata, de a processzor hardveres támogatást nyújthat ehhez. A legfontosabb cél itt is az, hogy a lehető legmagasabb áteresztőképességet biztosítsuk, ehhez pedig a szálakat úgy kell ütemezni, hogy a processzor végrehajtóegységeinek kihasználtsága minél nagyobb legyen.

A többszálú programvégrehajtás témaköréhez kapcsolódik a többprocesszoros rendszerek és többmagos processzorok alkalmazása is. Korábban ez főleg a nagyteljesítményű szerverekre és workstation-ökre volt jellemző, de mára a többmagos processzorok jelenléte mindennapossá vált a személyi számítógépekben, hordozható eszközökben, mobiltelefonokban és egyéb beágyazott rendszerekben is. Ebben az alfejezetben azonban főként az egy processzormagon belül elérhető szálszintű párhuzamossági (thread-level parallelism, TLP) megközelítésekre fókuszálunk. A többmagos processzorokkal és többprocesszoros rendszerekkel a 4. fejezetben foglalkozunk majd részletesebben.

2.5.1 Processzoron belüli többszálúság típusai

Az egy processzoron belüli többszálúság motivációja ugyanaz, mint ami az utasításszintű párhuzamosság egyre magasabb fokú kihasználásához vezetett. A cél továbbra is a processzor teljesítőképességének, áteresztőképességének növelése. Az utasításszintű párhuzamosításnak azonban vannak korlátai. Minél több a párhuzamos végrehajtóegység, annál komolyabb feladat mindig ellátni azokat feladattal, hiszen a kihasználatlan műveletvégző egységek nem járulnak hozzá a sebességnövekedéshez. Ugyanakkor egyetlen programszálon belül az utasítások korlátozottan függetlenek egymástól, vagyis a kihasználható utasításszintű párhuzamosság véges. Emellett, ha például egy memóriaműveletnél gyorsítótárhiány lép fel, akkor mai processzorokban akár több mint száz órajelciklust is igényelhet a főmemóriához való hozzáférés. Egy sorrenden kívüli végrehajtásra képes szuperskalár processzornak ilyenkor megvan a lehetősége, hogy addig is további utasításokat ütemezzen be. Azonban igen ritka, hogy egyetlen programfolyamban akkora mennyiségű független utasítás forduljon elő, ami ki tudja tölteni a memóriából érkező adatra való várakozás idejét. Sokkal inkább az a jellemző, hogy a független utasítások viszonylag hamar elfogynak, és a processzor várakozni kényszerül a memóriaművelet befejezéséig.

Copyright © 2021 / BME AUT

A szálszintű párhuzamosság alapja, hogy az egyes programszálak függetlenek egymástól. Ezt úgy kell érteni, hogy különböző szálakhoz tartozó utasítások között nem állnak fenn vezérlési és adatfüggőségek, illetve, ha egy programszál futása közben cache hiány lép fel, az nem érinti a többi szál futásképességét.[15]

Az eddig tárgyalt „hagyományos" vagy „egyszálú" (single-threaded) processzoroknál a többszálú programvégrehajtás illúziója a *multitasking* segítségével érhető el. Ez teljes mértékben szoftveres eljárás, aminek keretében az operációs rendszer különböző programokat, programszálakat ütemez rövid időnként egymás után. Az egyes szálak végrehajtása így időben átlapolódik, és a felhasználó számára úgy tűnik, mintha egyszerre futnának. Ezt a megoldást illusztrálja a 2.45a ábra, amely egy processzor végrehajtóegységeinek kihasználtságát szemlélteti az egymást követő gépi ciklusokban. Az egy oszlopban, egymás felett elhelyezkedő négyzetek a processzoron belüli párhuzamos végrehajtóegységeknek felelnek meg. Az azonos színnel kitöltött négyzetek az egy programszálhoz tartozó, végrehajtás alatt lévő utasításokat reprezentálják. Az üres négyzetek kihasználatlan végrehajtóegységeket jelölnek. Az egyes szálak rövid ideig futnak, és ha lejár az aktuális időszelet, vagy más megszakítás, kivétel (pl. page fault), vagy rendszerhívás történik, akkor az operációs rendszer kontextusváltást hajt végre, és egy másik szálat ütemez be. Az egyszerű multitasking szerepe a programszálak látszólagos párhuzamosítása, de a teljesítőképesség növeléséhez csak annyival járul hozzá, hogy amíg egy szál valamire várakozni kényszerül, addig nem kerül újra és újra ütemezésre. A gyakori kontextusváltás miatt az operációs rendszer ütemező része számos gépi ciklust „elpazarol", amelyeket egyébként a „hasznos" felhasználói programokra lehetne fordítani.

Léteznek hardverrel is támogatott, valódi szálszintű párhuzamosítási eljárások. Ezeket a módszereket az időosztásos többszálúság (temporal multithreading), vagy a szimultán többszálúság (simultaneous multithreading) csoportjába sorolhatjuk.

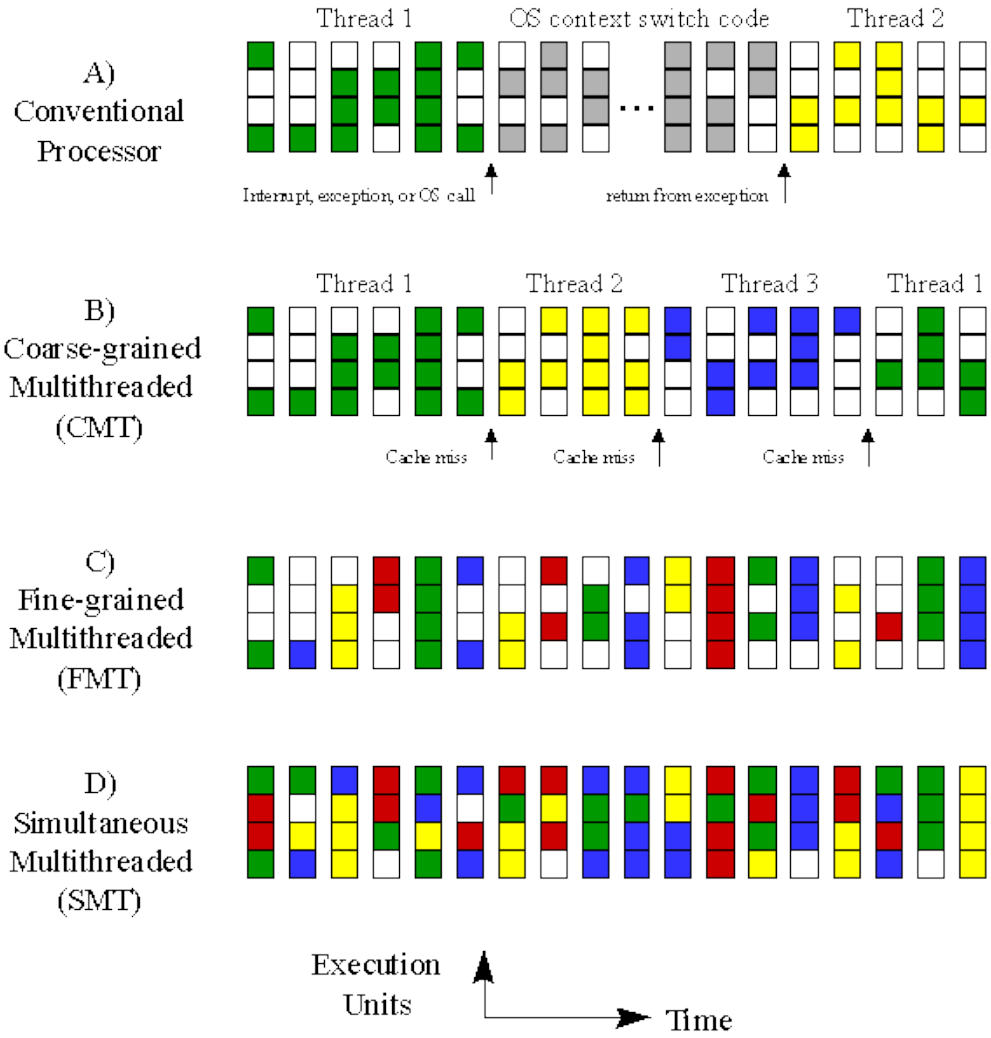
Coarse-grained Multithreading (CMT)

A hardver által támogatott TLP módszerek közül a legegyszerűbb az ún. coarse-grained multithreading, vagy „durva szemcsés" időosztásos többszálúság, amit a 2.45b ábra szemléltet. Ez esetben a processzor egyszerre kettő vagy több szál kontextusát is nyilvántartja, amit az architekturális állapotot tartalmazó regiszterek (programszámláló, státuszregiszter, stack pointer, általános regiszterek stb.) többszörözésével lehet megvalósítani. Ez nem jelent jelentős plusz hardverköltséget, mert ezek az egységek a processzor tranzisztorszámának csak nagyon kis százalékát teszik ki. Egy időben egyetlen szál aktív, mindaddig, amíg egy kivétel, vagy más, időkiesést okozó esemény (pl. cache hiány) meg nem akasztja a futását. Ha ez bekövetkezik, akkor a hardver automatikusan átvált egy másik szál kontextusára, és annak az utasításaival folytatja a végrehajtást. Ezzel a tisztán szoftveres multitasking-hoz képest a kontextusváltások időigényén sokat lehet spórolni, és növekszik az áteresztőképesség. A megoldás hasonlít a mikrokontrollerekben gyakran alkalmazott regiszterbankolási technikákra (pl. az Intel 8051 architektúrában az általános célú regiszterek 4 példányban érhetők el), amelyek gyors

[15] Természetesen nem értjük ide azokat az eseteket, amikor a szálak közötti szinkronizáció miatt bizonyos szálaknak kifejezetten meg kell várniuk egymást (pl. mutexek, szemaforok alkalmazása). Ez utóbbi azonban az utasításvégrehajtás időléptéke szempontjából jóval ritkább esemény, és egy magasabb logikai szinten előforduló függőséget jelent, ráadásul csak kitüntetett szálak között.

Copyright © 2021 / BME AUT

üzemmódváltásokat és megszakításkezelést tesznek lehetővé. Ezekben az esetekben a felhasználói és megszakítási „szálak” közötti gyors kontextusváltást segíti az architektúra.



2.45. ábra – Processzoron belüli többszálúság típusai[16]

Fine-grained Multithreading (FMT)

Egy másik időosztásos TLP módszer a fine-grained multithreading, vagy „finom szemcsés” többszálúság, amelynek működését a 2.45c ábra érzékelteti. Itt szintén több szál kontextusa van jelen egyszerre a processzorban, de a szálak ütemezése nem bizonyos eseményekhez kötött, hanem minden egyes órajelciklusban egy következő szál utasításait tölti be a processzor. Így, ha N db aktív szálunk van (az ábrán $N = 4$, a zöld, kék, sárga és piros színeknek megfelelően), akkor minden N-edik ciklusban lép be egy-egy újabb utasítás a csővezetékbe ugyanabból a szálból. Így bár egyetlen szál nézőpontjából látszólag N-ed részére csökken az

[16] Forrás: https://www.realworldtech.com/alpha-ev8-smt/ (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

áteresztőképesség, a helyzet valójában sokkal jobb, mert lecsökken az utasítások egymásra hatása a csővezetékben. Ugyanis egy szál utasításai között mindig van $N-1$ „szabad” ciklus, amiket a többi szál utasításaival töltünk fel, így a szálak összességére nézve javul a helyzet.

A megvalósítás hardverköltsége nem túl nagy: itt is meg kell többszörözni az architekturális állapotot tartalmazó regisztereket, és ki kell építeni a vezérlést a ciklusonkénti körbeforgáshoz, ugyanakkor az egymásra hatások lecsökkenése miatt az ILP kihasználhatósága megnő. Továbbá egyszerűbbé válik a csővezeték felépítése, mert nincs szükség például operandus-előrecsatoló hálózat kiépítésére a RAW típusú regiszter-adatfüggőségek automatikus feloldódása miatt. Az ilyen megoldást alkalmazó processzorokat barrel (hordó) processzornak is nevezik, és előszeretettel alkalmazzák hard realtime beágyazott rendszerekben a szálak determinisztikus ütemezése miatt.

Simultaneous Multithreading (SMT)

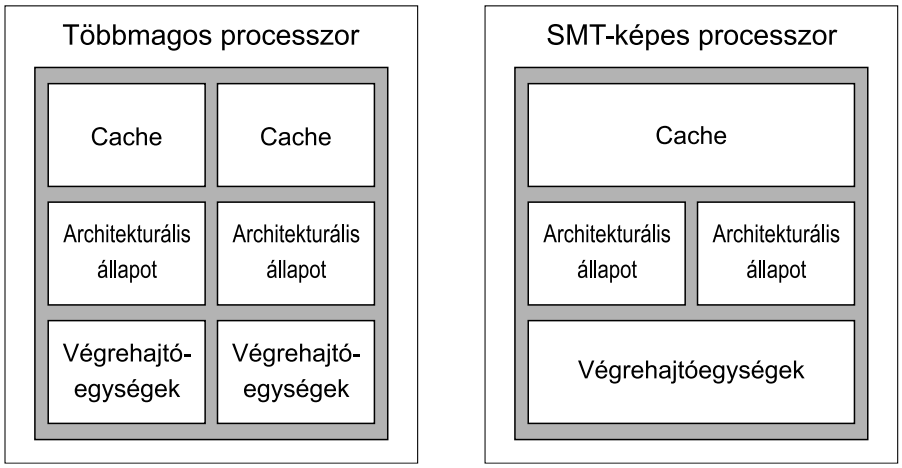
A legkifinomultabb TLP módszer a szimultán többszálúság, aminek lényege, hogy a hardver nem időben váltogat az egyes szálak végrehajtása között, hanem a processzor erőforrásait egyszerre több programszál is használhatja (ld. 2.45d ábra). Az SMT alkalmazásának alapja az, hogy egyetlen szál általában nem tudja kihasználni egy időben a processzor ILP lehetőségeit. Ilyenkor célszerű a fennmaradó, kihasználatlan erőforrásokat egy vagy több másik szál utasításainak végrehajtására felhasználni. Tehát az SMT-képes processzorokban minden egyes ciklusban több szál utasításai kerülnek kiosztásra a végrehajtóegységekre. Az SMT nagy előnye abban rejlik, hogy különböző jellegű terhelésekhez is rugalmasan tud alkalmazkodni. Ha alacsony az ILP kihasználhatósága, akkor több szál utasításaival tölti fel a végrehajtóegységeket. Ha viszont csak egyetlen szálat kell futtatnia, amelyben kismértékű az utasítások egymásra hatása, akkor a szuperskalár processzor teljes szélességben annak a rendelkezésére áll. Minden esetben a lehető legjobb a processzor kihasználtsága. Itt is igaz az, amit a CMT és FMT processzoroknál láttunk, hogy az architekturális állapot többszörözésével érhető el az egyes szálak rugalmas hozzárendelése a végrehajtóegységekhez.

Chip Multiprocessing (CMP)

Fontos, hogy a szimultán többszálúság fogalmát ne keverjük össze a többmagos processzorokkal. Az egy lapkán megvalósított több, teljes értékű processzormag alkalmazását chip multiprocessing-nek is nevezik. Ez lényegében nagyon hasonló a többprocesszoros rendszerekhez, hasonló problémákat vet fel, és ahogy már említettük, a 4. fejezetben fogunk részletesebben kitérni rájuk. A két fogalom leginkább azért keverhető össze, mert az operációs rendszer nézőpontjából, szoftveres szempontból nagyon hasonlítanak egymásra. Egy többmagos processzornál az operációs rendszer erőforrásként tekint az egyes processzormagokra a szálak ütemezése szempontjából. Egy SMT-képes processzorban a többszörözött architekturális állapot minden egyes példánya egy-egy logikai processzorként jelenik meg az operációs rendszer számára. Egy erre nem felkészített operációs rendszer nem tud különbséget tenni a fizikai és logikai magok között. Ugyanakkor, míg egy többmagos processzorban az egyes magok teljes értékűek, különálló végrehajtóegységekkel és dedikált cache memóriákkal, addig a logikai processzorok egy magon belül osztoznak mind a végrehajtóegységeken, mind a cache-en (ld. 2.46. ábra). Érthető módon a többmagos kialakítás

Copyright © 2021 / BME AUT

hardverköltsége sokkal nagyobb, cserébe teljes értékű párhuzamosítás érhető el vele, míg az egy magon belüli logikai processzorok egyetlen fizikai mag erőforrásain osztoznak. Ugyanakkor a két megoldás nem állítható szembe egymással, hiszen egy többmagos rendszerben az egyes processzormagok lehetnek SMT-képesek. Így például egy mai processzorban teljesen megszokott, hogy 2/4/6 fizikai magot tartalmaz, ami bekapcsolt SMT üzemmód esetén 4/8/12 logikai processzorként jelenik meg az operációs rendszer számára.



2.46. ábra – Többmagos és szimultán többszálúságot megvalósító egymagos processzorok különbsége

2.5.2 Az Intel Hyper-Threading Technology architektúra

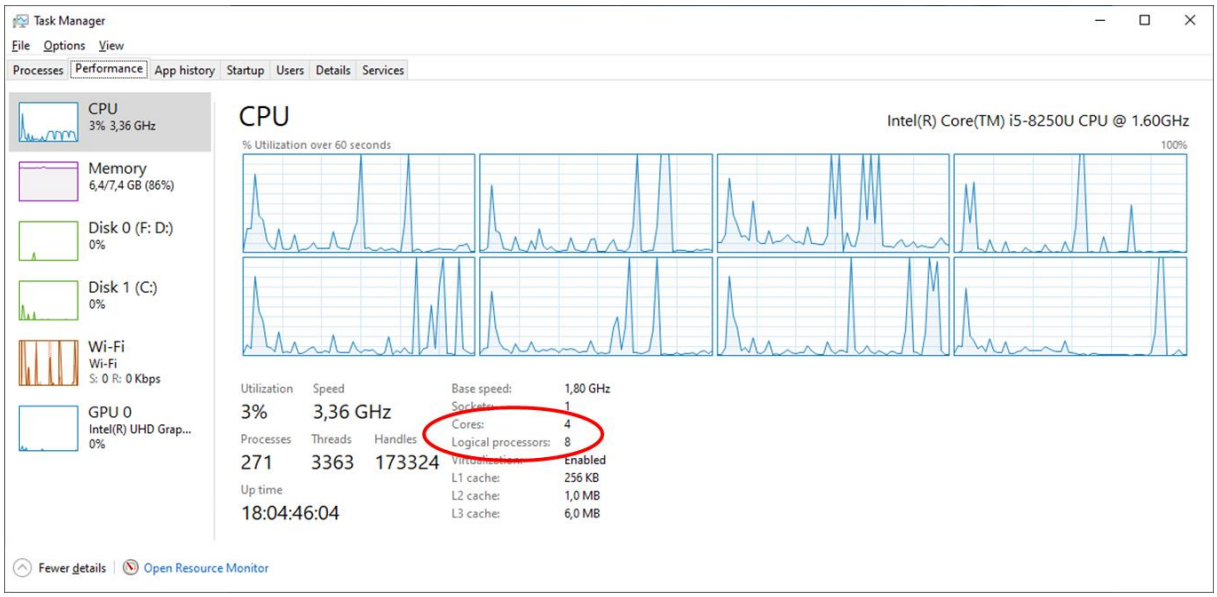
Az első, kereskedelmi processzorokban is napvilágot látott SMT implementáció az Intel Hyper-Threading Technology (HTT) névre keresztelt megoldása volt. Ezt megelőzte volna a DEC Alpha EV8-as processzora, amely 4-utas SMT megoldást tartalmazott, de a fejlesztését a végső szakaszban leállították. A nagyon ígéretes architektúra a DEC körül akkoriban (1990-es évek vége) zajló események áldozata lett. A DEC-t először megvette a Compaq, azt pedig nem sokkal később a Hewlett-Packard, amely ekkortájt nagy erőkkel dolgozott az Itanium nevű VLIW architektúrán az Intellel együtt. Az erőforrások összpontosítása miatt döntöttek úgy, hogy az EV8-as processzor fejlesztését leállítják.

A Hyper-Threading Technology két logikai processzort valósít meg egy magon belül (2-utas SMT). A technológia kifejlesztésének célja az volt, hogy minimális, kb. 5%-nyi plusz hardverrel jelentős, akár 20–30%-os teljesítőképesség-növekedést érjenek el a tipikus, általános célú alkalmazásokban. Bekapcsolt Hyper-Threading esetén az operációs rendszer két processzormagot lát egy helyett, de ahogy korábban is említettük, ez nem jelent kétszeres számítási teljesítményt.

Copyright © 2021 / BME AUT

2.5.2.1 Logikai processzorok ütemezése

Az operációs rendszerek számára az Intel HTT által definiált logikai processzorok a fizikai processzormagokhoz hasonlóan jelennek meg. Ha például megnyitjuk a Windows feladatkezelőjét, akár külön-külön láthatjuk az egyes logikai processzorok terhelését (2.47. ábra).



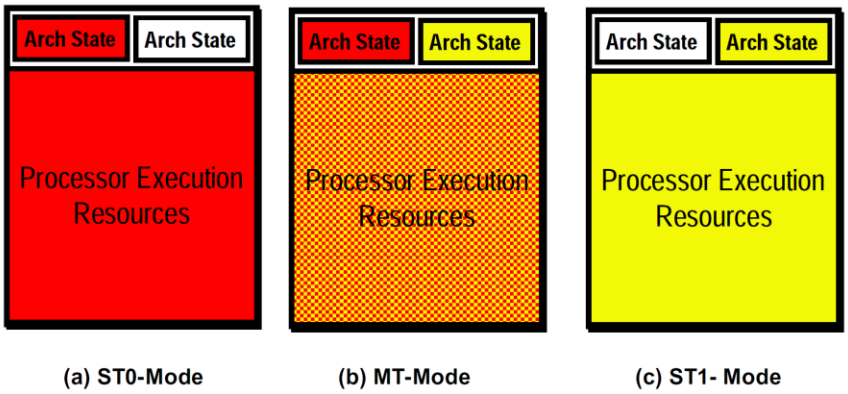
2.47. ábra – Logikai processzormagok a Windows feladatkezelőjében

Azonban fontos, hogy ne tévesszenek meg bennünket az itt látottak. Ha például kikapcsolt HTT mellett azt látjuk, hogy 4 db fizikai processzormagunk van, és mindegyik 100%-os terhelésen fut, később pedig bekapcsolt HTT mellett a 8 db logikai processzor van 100%-ig kiterhelve, akkor utóbbi esetben nem kétszer akkora a számítási teljesítmény. Ugyanis egy fizikai magon belül a két programszál csak annyival több erőforrást tud kihasználni, amennyi egyébként üresen járt volna a nem teljesen kiaknázható ILP miatt. Továbbgondolva a példánkat, ha bekapcsolt HTT mellett azt látjuk, hogy a 8 db logikai processzor közül 4 db jár 100%-os terhelésen, az két dolgot is jelenthet. Ha a 4 db logikai processzor mind-mind külön fizikai maghoz tartozik, akkor közel ugyanaz a helyzet, mint amikor a 4 db fizikai magot terheltük ki teljesen. Ha viszont a 4 db 100%-on járó logikai processzor közös fizikai magokon osztozik, akkor a tényleges számítási teljesítmény jóval kisebb, mint az első esetben.

Emiatt nagyon fontos, hogy az operációs rendszer tudatában legyen annak, hogy mely logikai processzorok tartoznak ugyanahhoz a fizikai maghoz. Sőt, ha nem jut elég futtatható programszál a logikai magoknak, akkor érdemes a félig kihasznált fizikai magokon leállítania az egyik logikai processzort. Ezt hívja az Intel HTT terminológia single-task üzemmódnak (ld. 2.48. ábra). Ilyenkor a statikusan 50–50%-ban partícionált erőforrások teljesen az aktív logikai maghoz lesznek rendelve, amivel megnő az érintett logikai processzor sávszélessége. Ha az operációs rendszer nincs tudatában annak, hogy HTT-képes processzorral van dolga (nem tudja megkülönböztetni a fizikai és logikai magokat), akkor ilyen félig kihasznált esetben a

Copyright © 2021 / BME AUT

rendszer teljesítőképessége rosszabb lesz, mint kikapcsolt HTT mellett lenne. A single-task üzemmód aktiválását leleményesen oldották meg. Ahhoz, hogy ne kelljen az utasításkészletet emiatt kibővíteni, az x86 architektúra HALT utasítását kell kiadni a leállítandó logikai processzornak. Ez az utasítás eredetileg alacsony fogyasztású állapotba viszi a processzort. Bekapcsolt HTT esetén azonban az történik, hogy csak az adott logikai processzor áll le, azaz a másik megkapja a fizikai mag összes erőforrását. Ha az egyetlen aktívan maradt logikai processzor is HALT utasítást kap, akkor az eredeti működés szerint valóban energiatakarékos üzemmódba kapcsol a fizikai mag. Ahogy alvó állapotból interrupt tudja felébreszteni a processzort, úgy single-task módból is interrupt segítségével lehet visszatérni multi-task módba, ha újra szükség van mindkét logikai magra. A logikai processzorok terhelés függvényében történő leállítása és aktiválása az operációs rendszer felelőssége a teljesítőképesség optimalizálása érdekében.



2.48. ábra – Multi-task és single-task módok az Intel HTT-képes processzorokban. A piros és sárga színek az egyes aktív logikai processzorokat és az általuk használt erőforrásokat illusztrálják.[17]

[17] Forrás: Deborah T. Marr et. al. Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal Q1, 2002. http://www.cs.virginia.edu/~mc2zk/cs451/vol6iss1_art01.pdf

Copyright © 2021 / BME AUT

2.6 Esettanulmány: Intel x86 mikroarchitektúrák

A következőkben két valós dinamikus szuperskalár mikroarchitektúrát mutatunk be. Világszerte nagy népszerűségnek örvendenek az x86 utasításkészlet-architektúrára épülő számítógépek, amelyek története 1976-ban, az Intel 8086-os processzorával kezdődött. Ez a processzor és utódai hamar elterjedtek. A kompatibilitás fenntartása miatt az x86 ISA az évtizedek során lényegében nem változott, csak folyamatosan bővült. A kezdeti 16 bites architektúrát 32, majd 64 bitesre bővítették, új utasításokat, regisztereket adtak hozzá, de a korábbi funkciók mindig elérhetők maradtak. A kezdetek óta eltelt több, mint 40 évben a processzorok hatalmas fejlődésen mentek keresztül. Az Intel az ISA bővítések mellett újabb és újabb mikroarchitektúrákat fejlesztett a teljesítőképesség folyamatos növelése érdekében. A 2.10. táblázat segítségével a jelentősebb x86 alapú Intel mikroarchitektúrákat, és az azokra épülő processzorokat tekinthetjük át.[18]

Év	Mikroarchitektúra	Processzorok
1976	8086	8086, 8088
1982	186	80186, 80188
1982	286	80286
1985	386	80386
1989	486	80486
1993	P5	Pentium, Pentium MMX
1995	P6	Pentium Pro, Pentium II, Pentium III
2000	NetBurst	Pentium 4
2003	Pentium M	Pentium M, Core Solo/Duo
2006	Core	Core 2 Solo/Duo/Quad, Pentium Dual Core
2008	Nehalem	Core i5, i7
2011	Sandy Bridge	Core i3, i5, i7 (2. generáció)
2012	Ivy Bridge	Core i3, i5, i7 (3. generáció)
2013	Haswell	Core i3, i5, i7 (4. generáció)
2014	Broadwell	Core i3, i5, i7 (5. generáció)
2015	Skylake	Core i3, i5, i7 (6. generáció)
2016	Kaby Lake	Core i3, i5, i7 (7., 8. generáció)
2017	Coffee Lake	Core i3, i5, i7 (8., 9. generáció)
2018	Cannon Lake	Core i3, i5, i7, i9 (9. generáció)
2019	Ice Lake	Core i3, i5, i7 (10. generáció)

2.10. táblázat – Intel x86 alapú mikroarchitektúrák

Az x86 ISA 32 bites változatát a 80386-os processzornál vezették be, ez már 3-fázisú csővezetéket tartalmazott, és támogatta a virtuális memóriakezelést. A 80486-os processzorban már FPU (Floating-Point Unit) is helyet kapott (a 386-os még társprocesszort alkalmazott a

[18] Népszerűsége miatt számos más gyártó is fejleszt x86 ISA alapú mikroarchitektúrákat, de ezek a táblázatban terjedelmi okok miatt nem szerepelnek.

Copyright © 2021 / BME AUT

lebegőpontos műveletvégzésre), és ötfázisú csővezetékkel rendelkezett. A P5 mikroarchitektúrában vezették be a szuperskalár kialakítást, ez gyakorlatilag 2 db párhuzamos 486-os csővezetéket jelentett.

Ezután egy nagy váltás következett. Az 1995-ben megjelent P6 volt az Intel első dinamikus csővezetékes szuperskalár mikroarchitektúrája, amely belső kialakítását tekintve teljesen különbözött az elődeitől. Lehetővé tette a sorrenden kívüli végrehajtást, regiszter-átnevezést alkalmazott, a csővezeték pedig a korábbi 5 helyett 14 fázisú lett. Ezt követte a NetBurst mikroarchitektúra, amely még mélyebb, 20-31 fokozatú csővezetéket (különböző alverziókban), és magas, akár közel 4 GHz-es órajelfrekvenciákat alkalmazott. A NetBurst architektúrát később nem fejlesztették tovább, ugyanis a magas órajelfrekvenciák miatt komoly disszipációs problémái voltak az erre épülő processzoroknak. Fontos megemlíteni, hogy az Intel Hyper-Threading Technology (HTT) megoldása a Netburst mikroarchitektúrában jelent meg először 2002-ben, az akkori Xeon és Pentium 4 (Northwood) processzorokban. Ugyanakkor a mikroarchitektúra egyéb sajátosságai miatt a Hyper-Threading Technology hatékonyságnövelő hatása nem volt egyértelműen sikeres.

A Netburst után az Intel visszatért a P6-hoz, és a következő Pentium M és Core mikroarchitektúrák a P6 közvetlen leszármazottainak tekinthetők. Ekkortájt jelent meg a többmagos kialakítás a teljesítőképesség további emelése érdekében, az órajelfrekvencia növelése helyett. Az Intel Core mikroarchitektúra támogatta először a 64 bites x86-64 ISA változatot. Mivel a Pentium M és Core mikroarchitektúrák újra a korábbi P6-ra épültek, nem tartalmaztak HTT támogatást.

A Core-t követő, 2008-ban megjelent Nehalem mikroarchitektúra szintén eléggé hasonlít a P6-ra, de számos jelentős újítást tartalmaz, ezek egyike a valóban hatékony Hyper-Threading Technology megvalósítás is. Elmondható róla, hogy megjelenése óta a mai napig meghatározza a széles körben alkalmazott x86-64 alapú Intel mikroarchitektúrákat. Ezzel jelent meg először az Intel x86-64 processzorok nagyfokú skálázhatósága a mobil alkalmazásoktól kezdve az asztali gépeken át a nagyteljesítményű szerverekig (Core i5, i7, Xeon processzorok; a Core i3 a következő, Sandy Bridge mikroarchitektúrától kezdve jelent meg).

Az Intel mikroarchitektúráknak számos későbbi generációja született azóta, de azok főleg kisebb ISA kiegészítéseket és gyártástechnológiai rajzolatfinomítást (45nm, 32nm, 22nm, 14nm, 10nm) jelentettek, illetve ez utóbbiból fakadóan nagyobb fokú integráltságot (több processzormag, integrált GPU).

Lényegében kijelenthető, hogy a P6-on alapuló Nehalem mikroarchitektúra mind a mai napig meghatározza az újabb Intel mikroarchitektúrákat.

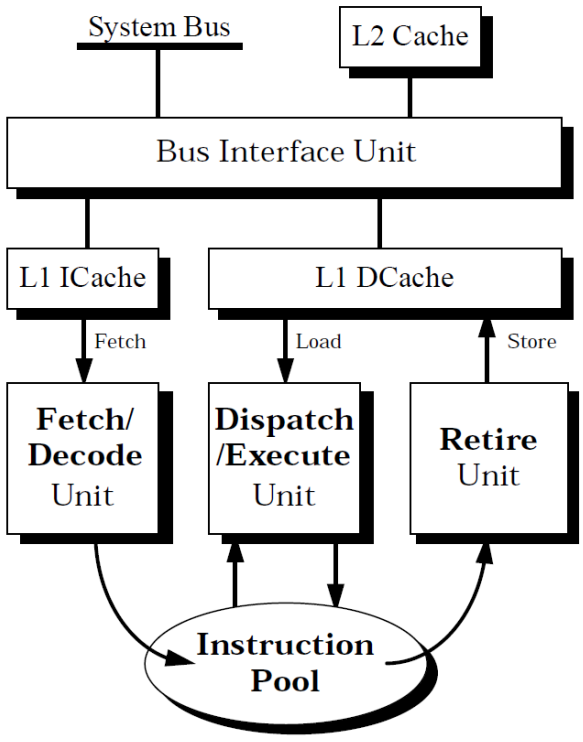
2.6.1 Az Intel P6 mikroarchitektúra

Jelentőségére való tekintettel először a P6 mikroarchitektúra felépítését és működését vesszük szemügyre kicsit közelebbről.

Copyright © 2021 / BME AUT

2.6.1.1 A csővezeték szétcsatolása

A P6 mikroarchitektúra az x86 ISA 32 bites változatát (IA-32, Intel Architecture 32) valósítja meg, a szuperskalár párhuzamosítás foka háromszoros (az előző, P5 architektúra kétszeres párhuzamosítást biztosított). Legfontosabb újítása a bemeneti (in-order front-end), a végrehajtó (out-of-order core), és a befejező (retirement) egységek szétcsatolása volt a sorrenden kívüli utasításvégrehajtás megvalósítása érdekében. A pipeline bemeneti szakaszát *betöltő/dekódoló egységnek* (Fetch/Decode Unit) is nevezik, amely programsorrendben hozza fel az utasításokat, és előkészíti azokat a végrehajtásra. A következő szakaszt *szétosztó/végrehajtó egységnek* (Dispatch/Execute Unit) hívják, amely az utasítások sorrenden kívüli kiosztását és végrehajtását végzi a diverzifikált műveletvégző egységek segítségével. Végül a *befejező egység* (Retire Unit) feladata az architekturális állapot frissítése az eredeti programsorrendnek megfelelően. A három szétcsatolt egység az *utasításpufferen* (Instruction Pool) keresztül kapcsolódik egymáshoz. A mikroarchitektúra áttekintő funkcionális vázlatát a 2.49. ábrán láthatjuk. Az ábra a szétcsatolt csővezeték fő elemein kívül a *buszinterfész egységet* (Bus Interface Unit) is szemlélteti, amelynek a processzormag külvilághoz (pl. memóriához) való kapcsolódásában van központi szerepe. Látható, hogy az architektúra kétszintű gyorsítótárral rendelkezik, az első szinten elkülönített utasítás és adat cache-sel.



2.49. ábra – A P6 mikroarchitektúra fő egységei[19]

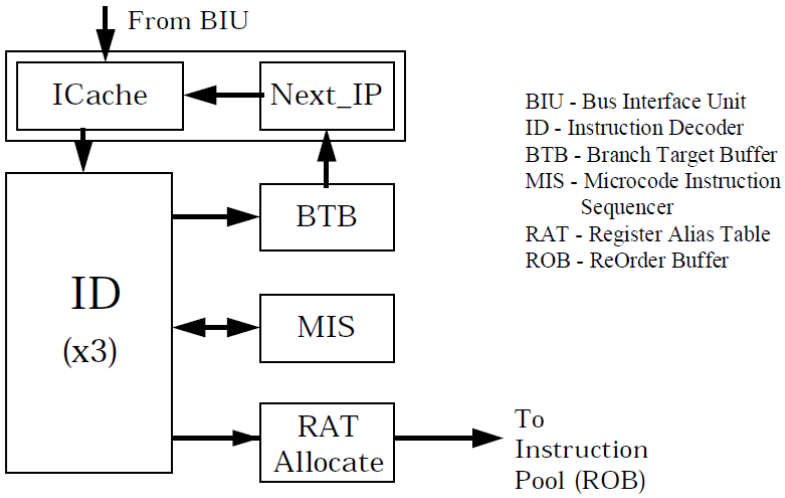
[19] Forrás: Intel Presents P6 Microarchitecture Details.
https://www.ece.uvic.ca/~amiralib/courses/p6 (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

Az architektúra tervezésekor a fejlesztők a korábbi alfejezetekben már bemutatott problémák megoldását célozták meg. Azt tapasztalták, hogy a statikus, sorba állított csővezeték-fokozatokra épülő szuperskalár szervezés nem elég hatékony, a műveletvégző egységek kihasználtsága alacsony. Például egy adatgyorsítótár-hiány (cache miss) sok órajelciklusnyi kiesést okoz, mert a lassabb főmemóriához való hozzáférést a soron következő utasításoknak is meg kell várniuk, akkor is, ha egyébként független műveleteket végeznének (a P5 mikroarchitektúra még csak egyszintű cache memóriát tartalmazott). Az utasításszintű párhuzamosság kellő hatékonyságú kihasználása érdekében a betöltő egység mélyen, akár 20-30 utasítással is előretekint az utasításfolyamban. Ebben az ablakban a lehívott utasítások között átlagosan 5 elágazás fordul elő, ezért hatékony elágazásjövendölésre van szükség, hogy a téves ágak gyakori kitakarítása ne akassza meg a folyamatos betöltést. A véges méretű architekturális regiszterkészlet miatt a viszonylag mély előretekintés következtében sok hamis adatfüggőség fordulhat elő a betöltött utasítások között, ezért regiszter-átnevezésre van szükség. Az architektúra által megcélzott dinamikus végrehajtás lényege, hogy előre becsülje a programfolyamot, és a program által definiált feladatok adatfolyamát felderítve kiválassza azt a végrehajtási sorrendet, amely minimalizálja az eredő végrehajtási időt.

2.6.1.2 Betöltő/dekódoló egység

A betöltő/dekódoló egység részletesebb vázlata a 2.50. ábrán látható. A gyorsítótárból programsorrendben hívja le az utasításokat, és azokat mikroutasításokba dekódolja (emlékezzünk, az x86 egy CISC architektúra). A lehívás spekulatív, az elágazásprediktor kétszintű adaptív elágazásjövendölést alkalmaz. Az elágazási célpuffer (BTB) 512 bejegyzést tartalmaz, az elágazások előtörténetét 4 biten tárolja.



2.50. ábra – Betöltő/dekódoló egység (Fetch/Decode Unit)[19]

A betöltő egység egyszerre két cache sort olvas be, és 16 illesztett bájtot ad át a dekódoló egységnek. Az illesztettség azt jelenti, hogy az átadott 16 bájtos csomag garantáltan egy utasítás első bájtjával kezdődik. Egy cache sor szintén 16 bájt hosszú. Azért van szükség mindig két sor

Copyright © 2021 / BME AUT

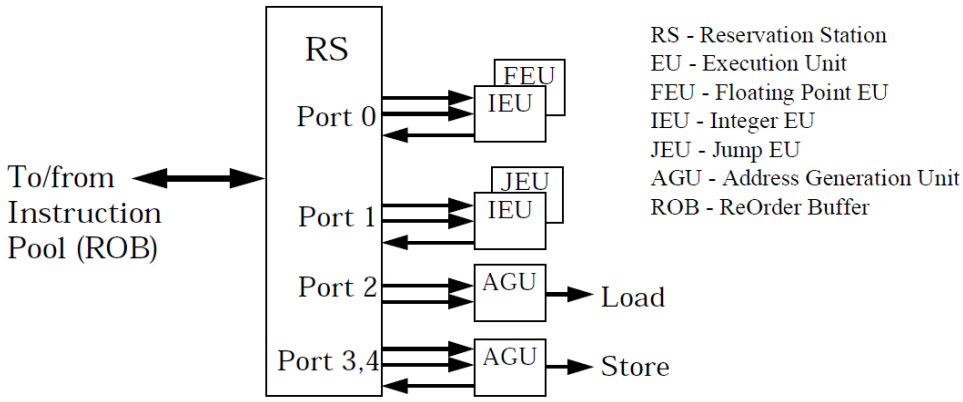
beolvasására, mert az x86 (IA) utasításfolyam bájtszervezésű, és a változó hosszúságú utasítások gyakran a cache sor közepén kezdődnek. A betöltött 16 bájt 1–16 db utasítást tartalmazhat. Mivel az utasítások eltérő hosszúságúak, a dekódolás első feladata meghatározni az utasítások határait. Ezt az elődekódolási feladatot a betöltő egység Instruction Length Decoder (ILD) alegysége végzi, megjelölve az utasítások első és utolsó bájtjait a 16 bájtos csomagban (az ILD nincs külön jelölve az ábrán). A dekódoló egység három párhuzamos utasításdekódert tartalmaz. Ezek kapják meg a megjelölt (makro)utasításokat, és mikroutasításokba (micro-operation, µop) dekódolják azokat. A legtöbb IA utasítás 1 µop-ból, mások 2–4 µop-ból állnak, és vannak több mikroutasítással megvalósítható komplex utasítások is. Ez utóbbiak mikroutasításai a dekóderek mellett található, Microcode Instruction Sequencernek (MIS) nevezett mikrokód ROM-ból töltődnek be. A mikroutasítások háromcímű (2 forrás és 1 céloperandus), egyszerű műveletek, amelyek lényegében RISC utasításoknak feleltethetők meg. A változatos utasítástípusok miatt a dekódolás változó számú órajelciklust igényel, és változó mennyiségű mikroutasítást eredményez makroutasításonként. Ezt az egyenetlenséget a dekódoló egységben található µop várakozási sor (instruction decoder queue) egyenlíti ki.

Dekódolás után a mikroutasítások regiszter-átnevezését a Register Alias Table (RAT) egység végzi el, és az Allocator egység státuszinformációval ellátva elhelyezi őket az utasításpufferbe. Az utasításpuffert (intruction pool) fizikailag a ReOrder Buffer (ROB) egység valósítja meg, amely 40 µop tárolására képes. Az átnevezéshez használt hardver regisztertár a ROB része.

2.6.1.3 Szétosztó/végrehajtó egység

A szétosztó/végrehajtó egység (2.51. ábra) feladata a mikroutasítások megfelelő ütemezése és kiosztása a műveletvégző egységeknek. A szétosztó egység másik neve Reservation Station (RS), amely a végrehajtásra váró µop-okat tárolja (maximum 20 db-ot), amíg sorra nem kerülnek. Az előző pontban említett Allocator ugyanis a µop-ok felcímkézésekor a ROB és az RS egységben is bejegyzi a végrehajtandó µop-okat. A Reservation Station a mikroutasításokat a rendelkezésre álló operandusok és a szabad erőforrások alapján ütemezi. Öt kapuval rendelkezik a műveletvégző egységek felé, így egy ciklusban akár 5 µop kiosztására is képes, de átlagosan 3 µop kiosztása a jellemző ciklusonként. A mikroarchitektúrában 6 db végrehajtó egység (Execution Unit) kapott helyet, ezek között van két egész (IEU) és egy lebegőpontos műveletvégző egység (FEU), egy elágazásokat kezelő egység (JEU) és két címképző egység (AGU) a memóriabetöltések és tárolások végrehajtásához.

Copyright © 2021 / BME AUT



2.51. ábra – Szétosztó/végrehajtó egység (Dispatch/Execute Unit)[19]

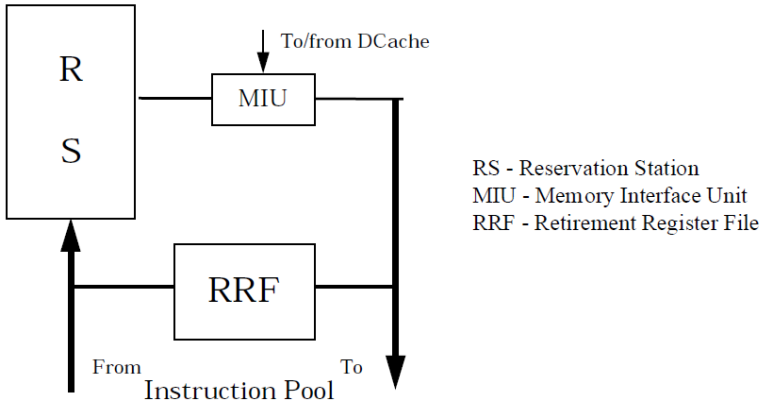
A végrehajtási sorrendet teljes mértékben az adatfolyam határozza meg. Amint rendelkezésre áll egy mikroutasítás összes forrásoperandusa, és szabad a hozzá tartozó végrehajtó egység, azonnal kiosztásra kerül. Abban az esetben, ha egyszerre több mikroutasítás is végrehajtható állapotba kerül, akkor az eredeti programsorrendben hamarabb következő élvez elsőbbséget. A végrehajtott mikroutasítás bejegyzése törlődik a Resetvation Station-ből, és a ReOrder Buffer-ben „végrehajtott” jelölést kap.

Az elágazásokat kezelő végrehajtó egységnek (Jump Execution Unit, JEU) fontos szerepe van a téves elágazásjövendölésekből való felépülésben és a megszakítások kezelésében. Amint azt a betöltő/dekódoló egységnél láttuk, egy ugró utasítás felhozatala után a további utasítások betöltése és végrehajtása spekulatívan folytatódik az elágazásjövendölés alapján. A végrehajtott µop-ok a ReOrder Buffer-ben várakoznak a sorrendi visszaírásra. Amikor az elágazás ténylegesen kiértékelődik, akkor kiderül, hogy helyes volt-e a jövendölés. Ha igen, akkor a spekulatívan felhozott µop-ok érvényesíthetők, azaz, ha már végre lettek hajtva, átadhatók a befejező egységnek. Ha téves volt a jövendölés, akkor a JEU az elágazás után felhozott µop-okat törli a ReOrder Buffer-ből, és az ugrás kiértékelésének eredményével frissíti a Branch Target Buffer-t, hogy a helyes ágról folytatódhasson tovább az utasítások felhozatala. Hasonló mechanizmus játszódik le abban az esetben is, ha egy megszakítás miatt kell máshonnan folytatni a programvégrehajtást. Ilyenkor a még nem visszaírt µop-ok szintén törlődnek az utasításpufferből, és a megszakítási rutin címétől folyik tovább az utasításbetöltés.

2.6.1.4 Befejező egység

A befejező egység (Retire Unit) fizikailag nem egy különálló egység, hanem a ReOrder Buffer része, feladata a végrehajtott mikroutasítások eredeti programsorrendben történő befejezésének vezérlése. A befejezés a permanens gépi állapot frissítését jelenti, azaz az utasítások eredményének eltárolását az architekturális regisztertárban (Retirement Register File, RRF), amely szoros kapcsolatban áll a ReOrder Buffer-rel. A Retire Unit 3 µop befejezésére képes órajelciklusonként.

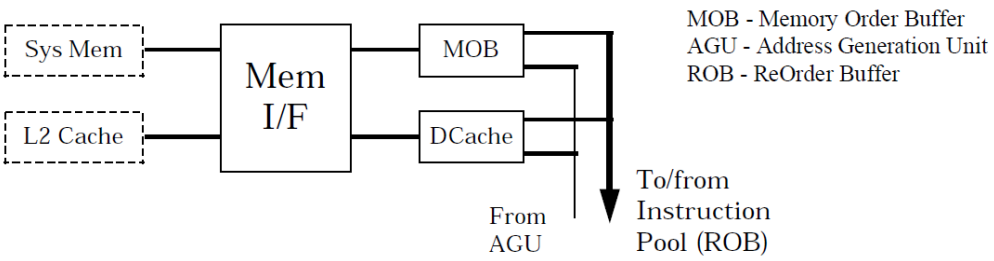
Copyright © 2021 / BME AUT



2.52. ábra – Retire Unit[19]

2.6.1.5 Buszinterfész egység

A buszinterfész egység (Bus Interface Unit, ld. 2.53. ábra) a memóriából való betöltéseket és tárolásokat vezérli. Ezek a műveletek különböznek a többitől, mert a végrehajtásuk nem csak a regisztertárat érinti, hanem memóriahozzáférést is igényel. A memóriát érintő adatfüggőségek feloldása nehézkesebb, mert sok esetben nem detektálhatók statikusan (mindjárt a dekódolás után), hanem bizonyos implicit függőségek csak végrehajtás közben derülnek ki. Például, ha egy regiszter indirekt címzésű tárolást (pl. *store r1, [r3]*) egy betöltés (pl. *load r2, b*) követ, akkor az indirekt címképzés elvégzése előtt nem tudhatjuk, hogy a betöltés által hivatkozott memóriacím (*b*) egyezik-e a tárolási címmel (*[r3]*), azaz nem tudni, hogy van-e adatfüggőség a két művelet között. Ezért a memóriaműveletek számára a ReOrder Buffer-hez hasonló működésű Memory Order Buffer (MOB) áll rendelkezésre, amely külön Store Buffer-t és Load Buffer-t tartalmaz a betöltő, illetve tároló utasítások számára.

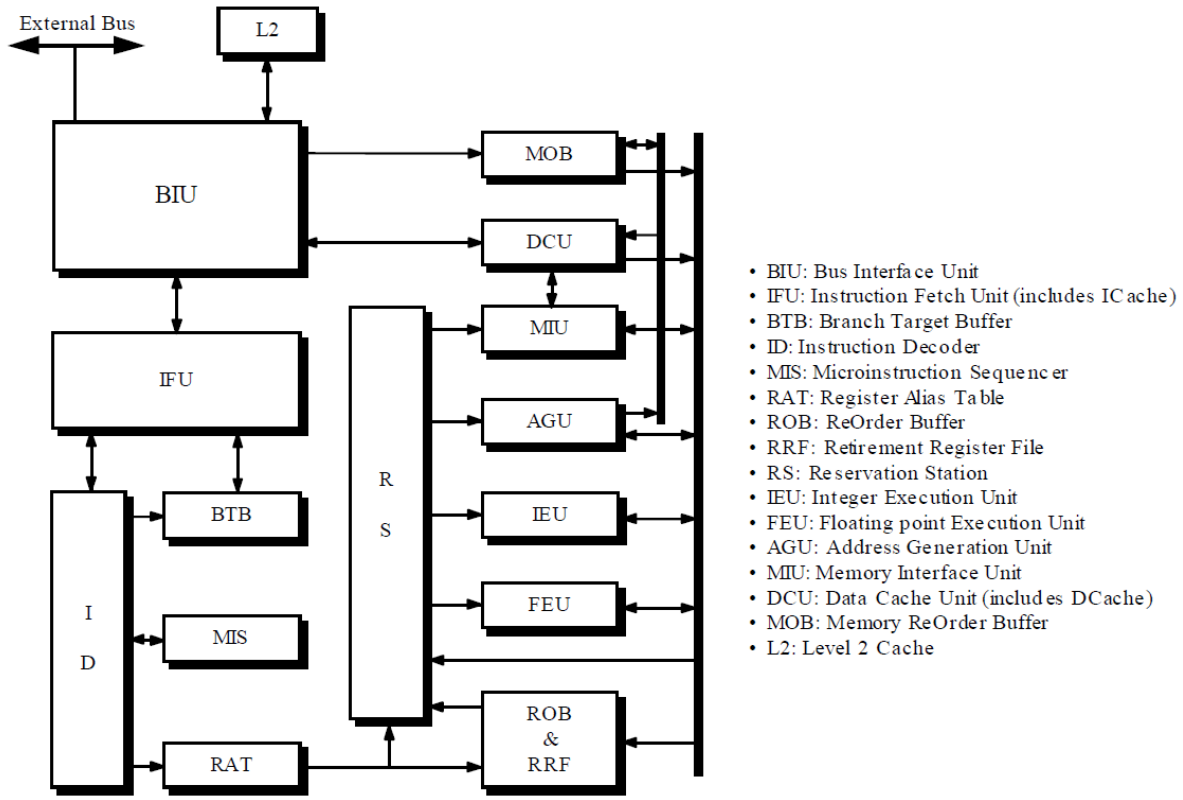


2.53. ábra – Bus Interface Unit[19]

A betöltő és tároló utasítások végrehajtása részben sorrendben, részben sorrenden kívül történik, az ütemezés a MOB feladata. A tárolások kizárólag sorrendben hajthatók végre, mert nem áll rendelkezésre a regiszter-átnevezéshez hasonló módszer a WAW adatfüggőségek feloldására. A Store Buffer csak akkor engedélyezi egy tároló művelet végrehajtását, ha a cím és az adat is rendelkezésre áll, és nincsen korábbi, végrehajtásra váró tárolás. A betöltések sorrendje felcserélhető. Bizonyos esetekben betöltés megelőzhet tárolást is:

Copyright © 2021 / BME AUT

- Egy betöltésnek mindaddig várakoznia kell a Load Buffer-ben, amíg minden őt megelőző store művelet célcíme nem került feloldásra.
- Ha ez megtörtént, akkor a load művelet minden olyan tárolást megelőzhet, amely eltérő memóriacímre hivatkozik.
- Ha van olyan korábbi store művelet a Store Buffer-ben, amelyik ugyanarra címre hivatkozik, de még nem áll rendelkezésre a kiírandó adat, akkor a load műveletnek várakoznia kell.
- Ha van olyan korábbi store művelet a Store Buffer-ben, amelyik ugyanarra címre hivatkozik, és az adat is rendelkezésre áll, akkor a load közvetlenül megkapja az adatot, még a store végrehajtása előtt (store-to-load forwarding).



2.54. ábra – A P6 mikroarchitektúra funkcionális vázlata[19]

A P6 mikroarchitektúra tárgyalásának végén egy összefoglaló blokkvázlatot láthatunk a 2.54. ábrán. Ezen egyben láthatók a dinamikus csővezetéket alkotó részegységek és a köztük lévő kapcsolatok.

Copyright © 2021 / BME AUT

2.6.2 Az Intel Nehalem mikroarchitektúra

A következőkben a Nehalem mikroarchitektúra utasítás-csővezetékének felépítését fogjuk megvizsgálni, mindig szem előtt tartva a HTT miatti sajátosságokat is. Az SMT kialakításnak megfelelően a processzor erőforrásait a két logikai szál megosztva használja. A megosztottság különböző formákban jelenik meg, erőforrásonként a következők valamelyike lehet:

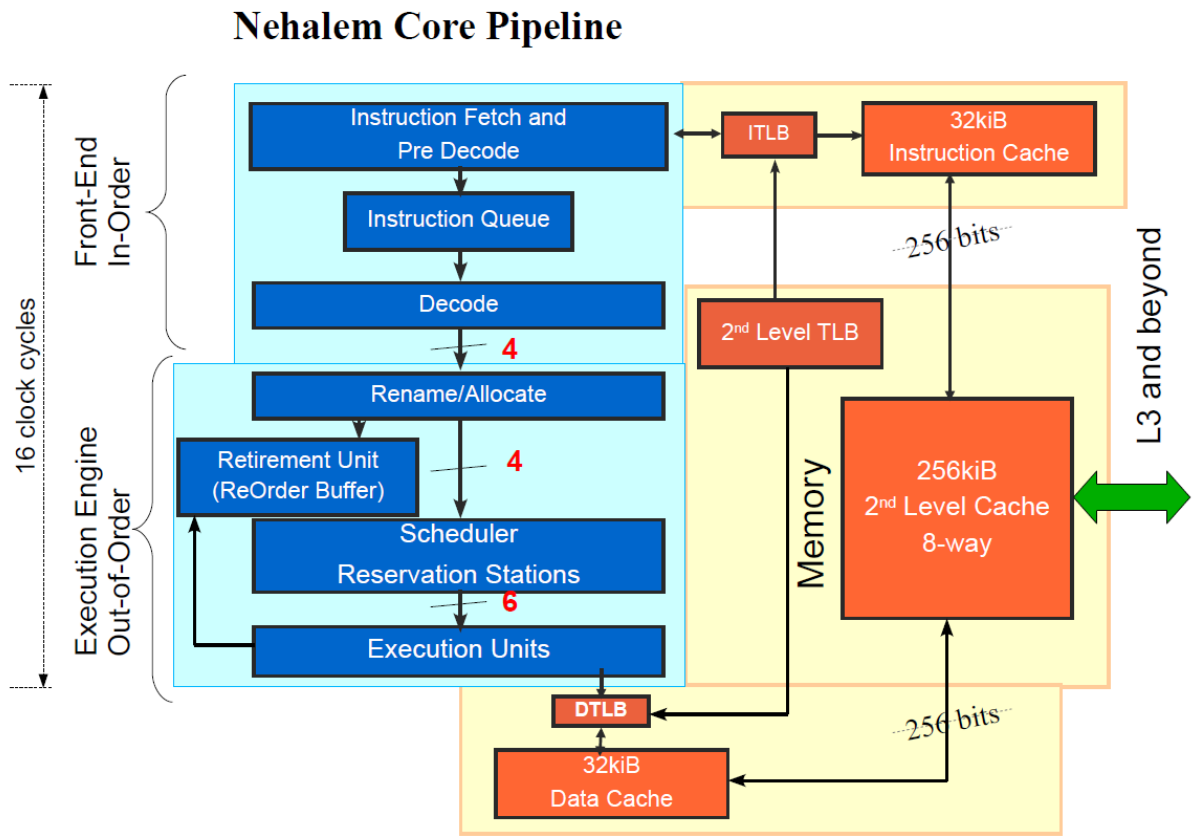
- Duplikált (pl. architekturális állapot regiszterei)
- Partícionált (statikusan 50–50%-ban megosztott)
- Dinamikusan megosztott (adott esetben az egyik szál akár 100%-ot is kaphat)
- HTT független (az adott funkcionális egység nem tud az SMT-ről)

A Nehalem csővezeték

A Nehalem mikorarchitektúra utasítás-csővezetéke (ld. 2.55. ábra) összesen 16 fázisból áll, de az egyes órajelciklusonkénti fázisokra való részletes lebontás nem lényeges a mostani vizsgálódásunk szempontjából. (A 16 ciklus hosszú végrehajtás a téves elágazásjövendölés miatti időkiesés alapján mérhető meg.) A P6 mikroarchitektúrához hasonlóan itt is sorrendi utasításbetöltést és sorrenden kívüli szuperskalár végrehajtást láthatunk. Szintén megjelenik az utasításpuffer (ReOrder Buffer) és a befejező egység is az eredeti programsorrend visszaállítása céljából. A csővezeték szélesebb, mint a P6 esetén: az ottani 3 helyett 4 szélességűnek tekinthető, ugyanis 4 párhuzamos utasításdekódert tartalmaz, illetve egyszerre 4 utasítás befejezését tudja gépi ciklusonként biztosítani.

A mikroarchitektúra további érdekessége, hogy kifejezetten támogatja a többmagos kialakítást, és magonként dedikált L1 és L2 cache memóriákat tartalmaz, amelyek közül az L1 szinten különválasztott utasítás és adat cache-t találunk. Az L3 gyorsítótár is a lapkán belül helyezkedik el, de ezt már a magok közösen használják. Szintén a Nehalem újítása volt az integrált DDR3 memóriavezérlő, és az ún. Quick Path Interconnect (QPI) interfész, amely a korábbi front side bus-t (FSB) váltotta le. Ezzel gyakorlatilag a korábban külön chip-ként megvalósított ún. északi híd (northbridge) áramkör a processzoron belülre került.

Copyright © 2021 / BME AUT



2.55. ábra – A Nehalem mikroarchitektúra csővezetéke[20]

Front-End Pipeline

A csővezeték első, sorrendi működésű szakasza az ún. front-end pipeline. Ez felelős a makroutasítások betöltéséért és mikroutasításokba való dekódolásáért, valamint a mikroutasítások „betárazásáért” a sorrenden kívüli végrehajtás számára. Egy ciklusban legfeljebb 4 makroutasítás dekódolására képes. Részletesebb vázlata a 2.56. ábrán látható.

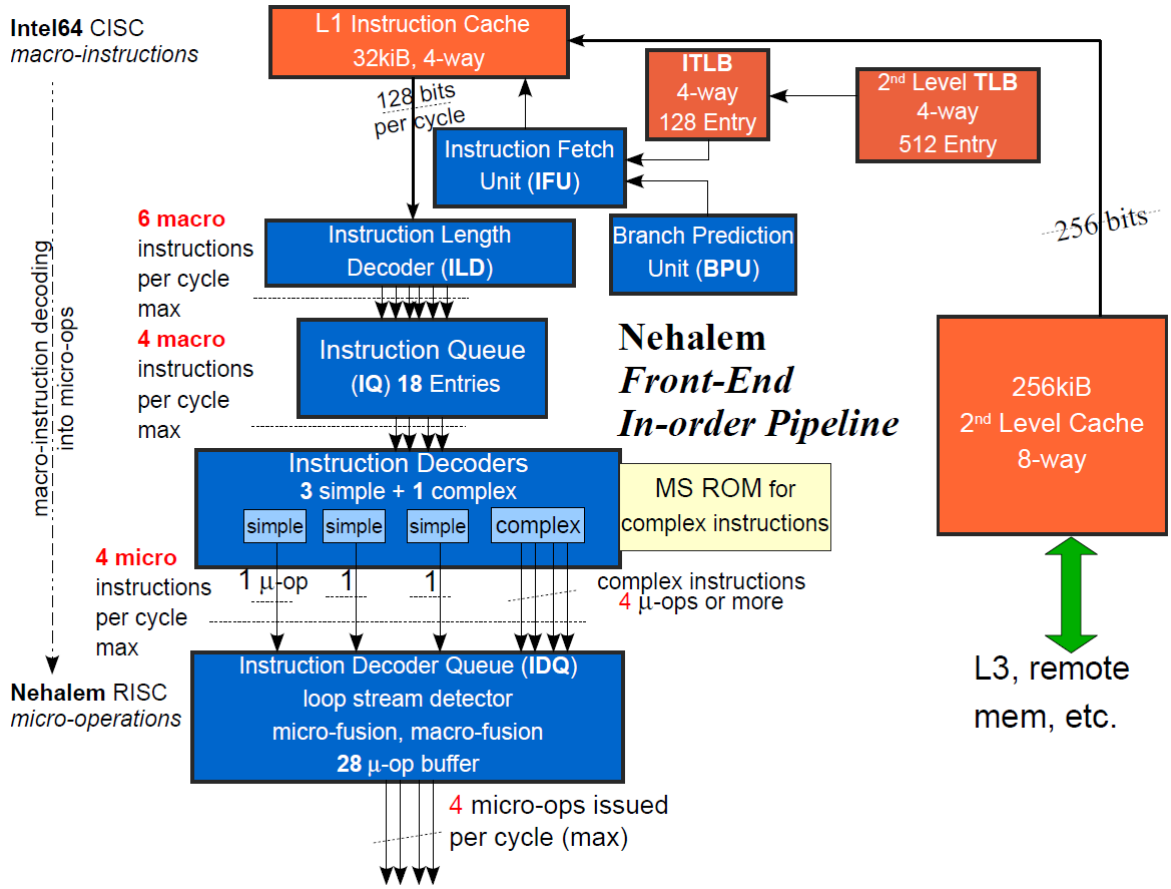
A P6-nál látottakhoz hasonlóan az *Instruction Fetch Unit* (IFU) ciklusonként 16 illesztett bájt betöltésére képes az utasítás cache-ből. A virtuális tárkezelést az ITLB (Instruction Translation Lookaside Buffer) segíti, ami egy speciális cache memória az utasításcímek transzformációjának gyorsítására (a virtuális tárkezelés részleteibe itt terjedelmi okokból nem megyünk bele részletesebben).

Az utasítások betöltése alternálva történik a két logikai processzor számára. Az ITLB két részből áll. Az ún. small-page ITLB (128 bejegyzés, 4 KB méretű lapokhoz) statikusan, azaz

[20] Forrás: Michael E. Thomadakis, The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. Texas A&M University, 201. https://www.semanticscholar.org/paper/The-Architecture-of-the-Nehalem-Processor-and-SMP-Thomadakis/0ebe76763a0c8f40851b9b75e1d51791c8565664

Copyright © 2021 / BME AUT

50–50%-ban megosztott a logikai magok között, az ún. large-page ITLB (7 bejegyzés, 2 MB méretű lapokhoz) pedig duplikálva van.



2.56. ábra – Nehalem Front-End Pipeline[20]

A spekulatív utasításfelhozatalban központi szerepet játszik a *Branch Prediction Unit* (BPU), amely tartalmazza a Branch Target Buffer-t (BTB) és az ún. Return Stack Buffer-t (RSB), ez utóbbi a RET utasítások célcím-előrejelzésére szolgál. A BTB dinamikusan megosztott a két logikai mag között, minden sorában szerepel egy logikai processzor ID, amely az adott logikai szálat azonosítja. Az RSB viszont két példányban létezik.

Az IFU által betöltött 16 illesztett bájt az *Instruction Length Decoder*-nek (ILD) nevezett előkódoló egységbe kerül. Ennek feladata, hogy detektálja az egyes utasítások elejét és végét, illetve jelölje az utasítások fontosabb tulajdonságait (pl. hogy egy utasítás elágazás-e) az utasításdekóderek számára. Ciklusonként maximum 6 utasítást tud betenni az utána következő 18 elemű *Instruction Queue*-ba (IQ), amelyből a dekódoló egység max. 4 utasítást tud kivenni minden ciklusban.

Az *Instruction Decode Unit* (IDU) 4 db párhuzamos utasításdekódert és egy mikrokód ROM-ot (micro-sequencer ROM, MS ROM) tartalmaz. Az utasítások nagy része 1 µop-ba dekódolható, ezeket 3 db ún. egyszerű dekóder dolgozza fel. A negyedik, ún. komplex dekóder az 1–4 µop-os

Copyright © 2021 / BME AUT

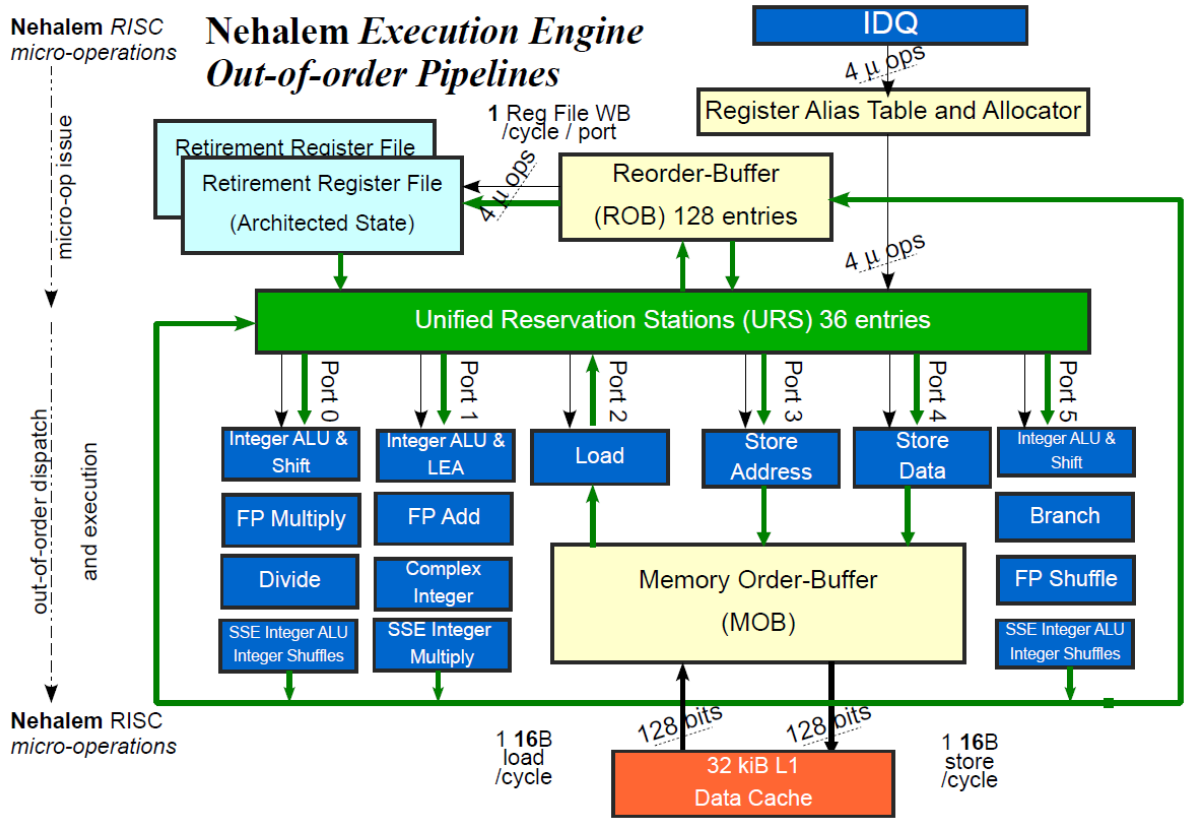
utasítások feldolgozására alkalmas. Az ennél összetetteb utasításokhoz az MS ROM szolgáltatja a mikroutasításokat. Az IDU további, ún. micro-op fusion és macro-op fusion szolgáltatásokat is nyújt. A *micro-op fusion* azt jelenti, hogy a dekóder bizonyos utasítások esetén 2 db egyszerű µop-ot egyetlen, több elemi feladatot magában foglaló µop-pal helyettesít. Ezek a kombinált µop-ok a csővezeték további részeiben (IDQ, RAT, ROB) mindenhol 1 µop-nyi helyet foglalnak el, de több elemi műveletet képviselnek (pl. egyszerre két műveletvégző egység dolgozza fel őket). A *macro-op fusion* pedig két makroutasítást von össze egyetlen µop-ba. Ez utóbbi akkor alkalmazható, ha egy logikai összehasonlítást feltételes elágazás követ. Ilyenkor 2 db x86-os architekturális gépi utasítás funkcióját egyetlen mikroutasítás valósítja meg.

A dekódolt mikroutasítások (ciklusonként max. 4 db) az *Instruction Decoder Queue*-ba (IDQ) kerülnek, amely 28 µop tárolására alkalmas. Ennek érdekes szolgáltatása a *Loop Stream Detector* (LSD), amely rövid programciklusmagok esetén kvázi utasítás cache-ként szolgál. A 28 µop általagosan 21–23 makroutasításnak felel meg. Ha a ciklusmag ekkora vagy kisebb, és így az összes hozzá tartozó utasítás elfér az IDQ-ban, akkor a Loop Stream Detector felismeri az ismétlődést, és felfüggeszti a betöltő (IFU), elágazásbecslő (BPU) és dekódoló (IDU) egységek működését. Ilyenkor az IDQ-ban lévő mikroutasítások képezik a további feldolgozás forrását az első téves elágazásbecslésig, azaz a ciklus végéig. Ez programciklusok feldolgozásakor nagyobb áteresztőképességet jelenthet alacsonyabb fogyasztás mellett, hiszen a front-end nagy része ki van kapcsolva, és ha például komplex (makro)utasításokat tartalmaz a ciklustörzs, akkor az egyetlen komplex dekódert tartalmazó IDU sem jelenthet szűk keresztmetszetet, hiszen az egyszer már dekódolt mikroutasításokat maximális sávszélességgel adhatjuk tovább a végrehajtó fokozatoknak.

Out-of-order Execution Engine

A csővezeték másik fele a mikroutasítások sorrenden kívüli végrehajtásáért és befejezéséért felelős. Felépítését részletesebben a 2.57. ábra mutatja. Az IDQ után következő *Register Rename and Allocator Unit* (RRAU) gépi ciklusonként 4 µop-hoz allokál erőforrásokat. Ennek során minden µop-ot bejegyez a ReOrder Buffer-be (ROB) és a Reservation Station-be (RS), valamint hozzárendeli egy megfelelő RS dispatch port-hoz. Memóriareferens µop esetén a Memory Order Buffer-ben (MOB) is bejegyzi a load vagy store buffer-be. Itt található a Register Alias Table (RAT), aminek segítségével megtörténik a µop-ok regiszter-operandusainak átnevezése a hamis adatfüggőségek feloldása érdekében.

Copyright © 2021 / BME AUT



2.57. ábra – Nehalem Out-of-order Execution Engine[20]

A *Unified Reservation Station* (URS, vagy egyszerűen csak RS) tárolja a végrehajtásra előkészített mikroutasításokat, amíg a bemeneteik vagy a hozzájuk rendelt végrehajtóegység nem áll rendelkezésre. 36 µop fér el benne, és a végrehajtóegységek 6 porton keresztül kapcsolódnak hozzá, így ciklusonként max. 6 µop kiosztására képes (4 µop az átlagosan jellemző, hiszen a csővezeték több pontján is ennyi a maximális keresztmetszet). Az RS ciklusonként 4 db eredmény-visszaírást tud elvégezni a hardver regisztertárba, és fejlett bypass-hálózattal rendelkezik az operandus-előrecsatolások támogatására, akár a különböző végrehajtóegységek között is. A Hyper Threading Technology szempontjából az RS bejegyzései dinamikusan megosztottak a logikai magok között. Ez azért előnyös, mert ha valamelyik szál várakozni kényszerül, akkor a kiosztás és így a feldolgozás teljes sávszélessége a másiké lehet.

A műveletvégző egységek 6 csoportba sorolhatók az RS 6 portjának megfelelően. Az egyszerre kioszthatő 6 µop közül 3 db lehet memóriaművelet és 3 db aritmetikai/logikai művelet. Az RS 2-es, 3-as és 4-es portján keresztül oszthatók ki a memóriaműveletek, ezek között lehet 1 db load, 1 db store címszámítás és 1db store adatelővétel. Az RS 0-ás, 1-es és 5-ös portjaihoz pedig számos aritmetikai/logikai egység csatlakozik, amelyek között találunk egész ALU és shift műveletvégzőket, dedikált SIMD végrehajtóegységeket (SSE utasítások közvetlen végrehajtásához), egyszeres és dupla pontosságú lebegőpontos műveletvégző egységeket (FP ADD, MUL stb.), osztóművet és elágazáskezelő egységet.

Copyright © 2021 / BME AUT

A *ReOrder Buffer* (ROB) minden végrehajtás alatt lévő (allokált és még nem befejezett) mikroutasítás állapotát nyilvántartja. Ebben foglal helyet a hardver regisztertár, illetve a befejező egység is, amely menedzseli a (makro)utasítások sorrendi befejezését és az architekturális állapot (*Retirement Register File*, RRF) frissítését. A ROB 128 µop nyilvántartására alkalmas, és engedélyezett Hyper-Threading esetén statikusan van felosztva fele-fele arányban a logikai szálak között. Ez igazságos megosztást eredményez, ami azt jelenti, hogy mindkét szál azonos mélységű utasításablakban tekinthet előre. Az architekturális regisztertár (RRF) duplikálva van, ez teszi lehetővé, hogy az operációs rendszer két szimultán működésű logikai processzort láthasson.

A P6 mikroarchitektúránál látottakhoz hasonlóan a Memory Order Buffer vezérli az adatmemória-referens utasítások (load/store) végrehajtását. A benne lévő Load Buffer 48 load µop, a Store Buffer 32 store µop tárolására képes, és mindkettő 128 bites dedikált olvasási, ill. írási interfésszel rendelkezik az L1 adat cache felé. A store műveletek itt is mindig csak programsorrendben hajthatók végre, de load műveleteknél megengedett a sorrenden kívüli végrehajtás, és meg van valósítva a store-to-load forwarding mechanizmusa is. A Load és a Store Buffer is statikusan van felosztva a HTT logikai processzorok között, 50–50% arányban.

A Nehalem mikroarchitektúra áttekintésének végén foglaljuk össze, hogyan férnek hozzá a Hyper-Threading technológia által definiált logikai magok az egyes erőforrásokhoz:

- Duplikált erőforrások:
  - Architekturális állapot (RRF)
  - Return Stack Buffer (RSB)
  - ITLB (large page)
- Partícionált (statikusan 50–50%-ban felosztott) erőforrások:
  - ITLB (small page)
  - Reorder buffer (ROB)
  - Load és store bufferek
- Dinamikusan megosztott erőforrások:
  - Branch Target Buffer (BTB)
  - Reservation station (RS)
  - Cache memóriák
  - Adat TLB-k
- HTT transzparens erőforrások:
  - Utasításdekóderek
  - Utasítás várakozási sorok (IQ, IDQ)
  - Műveletvégző egységek

Copyright © 2021 / BME AUT

Felhasznált és ajánlott irodalom

[13] Nicholas Carter. Schaum's Outline of Computer Architecture. McGraw-Hill, 2002. ISBN 0-07-136207-X

[14] Kai Hwang. Advanced Computer Architecture. Parallelism, Scalability, Programmability. McGraw-Hill, 1993. ISBN 0-07-031622-8

[15] V. G. Oklobdzija. Digital Systems and Applications, CRC Press, 2008. ISBN 978-0-8493-8619-0

[16] J. P. Shen, M. H. Lipasti. Modern Processor Design: Fundamentals of Superscalar Processors. Waveland Press, 2013. ISBN 9781478610762

[17] Wikipedia. Standard Performance Evaluation Corporation. https://en.wikipedia.org/wiki/Standard_Performance_Evaluation_Corporation (Hozzáférés: 2021.03.12)

[18] Wikipedia. LINPACK benchmarks. https://en.wikipedia.org/wiki/LINPACK_benchmarks (Hozzáférés: 2021.03.12)

[19] Wikipedia. Dhrystone. https://en.wikipedia.org/wiki/Dhrystone (Hozzáférés: 2021.03.12)

[20] Embedded Microporcessor Benchmark Consortium. CoreMark. https://www.eembc.org/coremark/ (Hozzáférés: 2021.03.12)

[21] Wikipedia. Amdahl's law. https://en.wikipedia.org/wiki/Amdahl's_law (Hozzáférés: 2021.03.12)

[22] Wikipedia. Instruction-level parallelism. https://en.wikipedia.org/wiki/Instruction-level_parallelism

[23] Wikipedia. Flip-flop (electronics) https://en.wikipedia.org/wiki/Flip-flop_(electronics) (Hozzáférés: 2021.03.12)

[24] Arithmetic Pipeline. https://www.javatpoint.com/arithmetic-pipeline (Hozzáférés: 2021.03.12)

[25] Wikipedia. Instruction pipelining. https://en.wikipedia.org/wiki/Instruction_pipelining (Hozzáférés: 2021.03.12)

[26] Wikipedia. Classic RISC pipeline. https://en.wikipedia.org/wiki/Classic_RISC_pipeline (Hozzáférés: 2021.03.12)

[27] Wikipedia. Hazard (computer architecture). https://en.wikipedia.org/wiki/Hazard_(computer_architecture) (Hozzáférés: 2021.03.12)

[28] Wikipédia. MIPS architektúra. https://hu.wikipedia.org/wiki/MIPS_architekt%C3%BAra (Hozzáférés: 2021.03.12)

[29] Wikipedia. Intel 80486. https://en.wikipedia.org/wiki/Intel_80486 (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

[30] John Crawford, „The execution pipeline of the Intel i486 CPU,” Digest of Papers Compcon Spring '90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage, San Francisco, CA, USA, 1990, pp. 254-258, doi: 10.1109/CMPCON.1990.63682.

[31] Wikipedia. Superscalar processor. https://en.wikipedia.org/wiki/Superscalar_processor (Hozzáférés: 2021.03.12)

[32] Wikipedia. Out-of-order execution. https://en.wikipedia.org/wiki/Out-of-order_execution (Hozzáférés: 2021.03.12)

[33] Wikipedia. Branch predictor. https://en.wikipedia.org/wiki/Branch_predictor (Hozzáférés: 2021.03.12)

[34] Wikipedia. Micro-operation. https://en.wikipedia.org/wiki/Micro-operation (Hozzáférés: 2021.03.12)

[35] Wikipedia. List of Intel CPU microarchitectures. https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures (Hozzáférés: 2021.03.12)

[36] Wikipedia. P6 (microarchitecture). https://en.wikipedia.org/wiki/P6_(microarchitecture) (Hozzáférés: 2021.03.12)

[37] Intel Presents P6 Microarchitecture Details. https://www.ece.uvic.ca/~amiralib/courses/p6.pdf (Hozzáférés: 2021.03.12)

[38] Ronny Ronen. The Pentium II/III Processor „Compiler on a Chip” http://www.cs.tau.ac.il/~afek/p6tx050111.pdf (Hozzáférés: 2021.03.12)

[39] Stack Overflow. Why is processing a sorted array faster than processing an unsorted array? https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array (Hozzáférés: 2021.03.12)

[40] Wikipedia. Very long instruction word. https://en.wikipedia.org/wiki/Very_long_instruction_word (Hozzáférés: 2021.03.12)

[41] Wikipedia. Multithreading (computer architecture) https://en.wikipedia.org/wiki/Multithreading_(computer_architecture) (Hozzáférés: 2021.03.12)

[42] Wikipedia. Temporal multithreading. https://en.wikipedia.org/wiki/Temporal_multithreading (Hozzáférés: 2021.03.12)

[43] Wikipedia. Barrel processor. https://en.wikipedia.org/wiki/Barrel_processor (Hozzáférés: 2021.03.12)

[44] Wikipedia. Simultaneous multithreading. https://en.wikipedia.org/wiki/Simultaneous_multithreading (Hozzáférés: 2021.03.12)

[45] Paul DeMone. Alpha EV8 (Part 2): Simultaneous Multi-Threat. https://www.realworldtech.com/alpha-ev8-smt/ (Hozzáférés: 2021.03.12)

[46] Wikipedia. Hyper-threading. https://en.wikipedia.org/wiki/Hyper-threading (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

[47] Deborah T. Marr et. al. Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal Q1, 2002. https://www.semanticscholar.org/paper/Hyper-Threading-Technology-Architecture-and-1-and-Marr-Binns/04b58af4fc0e5c3e8e614e2ddb0c41749cc9166c (Hozzáférés: 2021.03.12)

[48] Wikipedia. Nehalem (microarchitecture). https://en.wikipedia.org/wiki/Nehalem_(microarchitecture) (Hozzáférés: 2021.03.12)

[49] Michael E. Thomadakis, The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. Texas A&M University, 2011 https://www.researchgate.net/publication/235960679_The_Architecture_of_the_Nehalem_Processor_and_Nehalem-EP_SMP_Platforms (Hozzáférés: 2021.03.12)

[50] David Kanter. Inside Nehalem: Intel's Future Processor and System. https://www.realworldtech.com/nehalem/ (Hozzáférés: 2021.03.12)

[51] Antonio Valles. Performance Insights to Intel Hyper-Threading Technology. https://web.archive.org/web/20150217050949/https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/ (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

3 ARM processzorok

A beágyazott rendszerek körében ma már megkerülhetetlen az ARM architektúrára épülő processzorok alkalmazása. Az ARM Holdings cég által fejlesztett és licenszelt RISC architektúra vezető szerepet tölt be a mobil eszközök piacán (a cég saját adatai szerint a mobiltelefonok több mint 90%-a és általában a mobil eszközök közel 60%-a ARM architektúrára épül). Az ARM processzorok népszerűségének egyik oka a nagy számítási kapacitás mellett is kiemelkedő energiahatékonyságuk. Sokféle felhasználói és ipari elektronikai eszközben megtalálhatók, kezdve a mobiltelefonokban, tabletekben alkalmazott nagyteljesítményű System-on-Chip (SoC) rendszerektől egészen a célfeladatokra szánt, közepes és kisteljesítményű mikrokontrollerekig (szórakoztató-elektronikai eszközök, háztartási gépek, számítógép-perifériák, hálózati eszközök). Az energiahatékonyságra kevésbé érzékeny asztali számítógépekben, nagyteljesítményű szerverekben más architektúrák népszerűbbek, de az utóbbi időben kifejezetten ezt a szegmenst is megcélozta az ARM új, adatközpontokba és cloud infrastruktúrába szánt processzoraival.

Fontos megjegyezni, hogy az ARM Holdings maga nem gyárt processzorokat. Utasításkészlet-architektúrátkat és rájuk épülő processzor mikroarchitektúrákat, illetve más, ún. *intellectual property*-ket (IP) fejleszt, amelyeknek a használati jogát megvásárolhatják más rendszerfejlesztők és chip-gyártók. Az általános célú processzor IP-k mellett elérhetők grafikus processzorok, SoC rendszerek belső buszai, memóriavezérlői és egyéb komponensei is.

A licenszek különböző formában elérhetők. Meg lehet vásárolni egy-egy ARM processzormag kapuszintű tervét szimulációs modellel együtt, saját fejlesztésű SoC-ben való felhasználásra. Akár a processzorok egy-egy konkrét gyártástechnológiához illesztett fizikai implementációját is megkaphatjuk. Van olyan formája a processzormag licenszeknek, amelyek lehetővé teszik a mag módosítását, optimalizálását, kiegészítését. Arra is van lehetőség, hogy nem egy konkrét processzormag, hanem az ARM utasításkészlet-architektúra valamelyik verziójának felhasználási jogát vásárolja meg egy cég, és saját mikorarchitektúrát fejleszt hozzá. Számos ismertebb beágyazott processzorgyártó él ez utóbbi lehetőséggel (ilyenek például a Qualcomm Snapdragon, a Samsung Exynos, az Nvidia Tegra és az Apple „A” sorozatú mobilprocesszorok, SoC-k is).

3.1 Az ARM architektúra története és fejlődése

Az ARM architektúra fejlesztése 1983-ban kezdődött a Nagy-Britanniai Acorn Computers Ltd. laboratóriumában. Ez a cég akkoriban az Acorn Proton (BBC Micro márkanéven értékesített) személyi számítógépek fejlesztőjeként és gyártójaként volt ismert, amelyek egy kormányzat által támogatott projekt keretében készültek azzal a céllal, hogy minden iskolába eljuttassák a számítástechnikát. Ezek a gépek akkoriban a MOS Technology 6502-es, 8 bites processzora köré épültek, és az Acorn Computers célja az volt, hogy egy erősebb processzor alkalmazásával az üzleti felhasználókat is elérhessék a gépeikkel. Az akkoriban elérhető CISC

Copyright © 2021 / BME AUT

mikroprocesszorok (Motorola 68000, National Semiconductor 32016) tesztelése után arra jutottak, hogy inkább egy alapoktól újragondolt architektúrára van szükségük. Az ekkoriban fejlesztés alatt álló Berkeley RISC projekt publikációi nagy hatással voltak az Acorn szakembereire, és úgy döntöttek, hogy egy saját 32 bites processzor fejlesztésébe fognak. Az Acorn RISC Machine projekt 1983-ban indult el Sophie Wilson és Steve Furber vezetésével. Teljesen új utasításkészletet és mikroarchitektúrát alkottak, amelyek kialakításában a fő vezérlő elv az egyszerűség és az alacsony késleltetésű I/O és memóriakezelés volt (pl. a költséges DMA hardverkomponens kihagyásával). Szilíciumgyártó partnerként a VLSI Technology csatlakozott hozzájuk. Az első legyártott ARM processzor minták 1985-ben készültek el, és már az első teszteken megfelelően működtek. A később ARM1-nek nevezett processzor nem került kereskedelmi forgalomba. A BBC Micro másodprocesszoros egységében[21] alkalmazták, és az ARM2 processzor fejlesztéséhez használt CAD rendszer gyorsítását végezte. Az ARM2 processzor 32 bites adatbusszal, 26 bites címtartománnyal és 27 db 32 bites regiszterrel rendelkezett, és az első RISC alapú, otthoni felhasználásra szánt kereskedelmi számítógép, az 1987-ben elkészült Acorn Archimedes központi egységeként szolgált. Kevesebb, mint 30 ezer tranzisztort tartalmazott, és a mérések alapján jobb számítási teljesítményt nyújtott, mint az Intel 80286 CISC processzor a maga több, mint 130 ezer tranzisztorával. A már 4 KB cache-sel felszerelt következő modell (ARM3) 1989-ben jelent meg.



a) b)

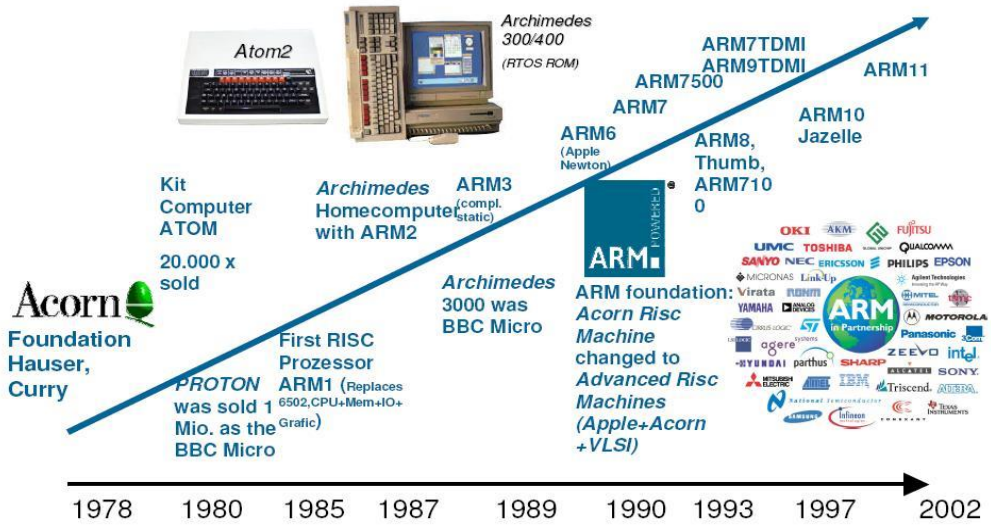
3.1. ábra – Acorn Computers fejlesztésű otthoni számítógépek. a) BBC Micro[22], b) Archimedes[23]

1990-ben az Acorn Computers a VLSI Technology-val és az Apple Computerrel összefogva külön cégbe szervezte ki az ARM processzorok fejlesztését, így jött létre az Advanced RISC Machines Ltd. (amelyet ma ARM Holdings néven ismerünk). Az együttműködés hátterében az állt, hogy az Apple használni szerette volna az ARM technológiát, de nem akarta termékeit a személyi számítógépek piacán ekkor konkurensnek számító Acorn licenszére építeni. A cég tőkéjét az Apple finanszírozta, az Acorn adta a szakembereket (ez egy 12 fős mérnökcsapatot jelentett), a VLSI pedig az eszközöket biztosította. A közös munka eredményeként 1991-ben

[21] A másodprocesszor a BBC Micro esetében egy külső egységet jelentett, aminek csatlakoztatása után a külső processzor vette át az elsődleges processzor szerepét, és az eredeti főprocesszor csak az I/O kezelést végezte.
[22] Forrás: https://en.wikipedia.org/wiki/BBC_Micro (Hozzáférés: 2021.03.12)
[23] Forrás: https://en.wikipedia.org/wiki/Acorn_Archimedes (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

született meg az ARM6 mikroarchitektúra, amely számos újítást tartalmazott az utasításkészletben is (pl. a címbusz 32 bites lett), de a mérete csupán kis mértékben, 35 ezer tranzisztorra növekedett. Az Apple az ARM6 alapú ARM610-es chip köré építette az 1993-ban bemutatott Apple Newton nevű PDA-ját, amely kezdeti hibái miatt viszont nem lett sikeres termék.



3.2. ábra – Az ARM történetének első két évtizede

Ekkoriban döntött az ARM vezetése a saját chip-ek gyártása helyett az IP licenszek eladására épülő üzleti modell mellett, hogy ne egy-egy konkrét termék sikeressége határozza meg a jövőjüket. Az ARM processzorok licenszét számos chipgyártó cég kezdte vásárolni, amelyek egy egyszeri licenszdíjért és a legyártott lapkák után fizetett jogdíjért cserébe használhatták a kész processzorterveket. A partnerek számára ez a modell jelentősen lecsökkentette termékeik piacra jutási idejét, az ARM-nak pedig megalapozta az üzleti jövőjét. Erre az időszakra esett az egyre kisebb méretben megvalósítható mobiltelefonok elterjedése. Az ARM egyik jelentős licenszpartnere a Texas Instruments volt, amely a Nokia beszállítójaként javasolta, hogy a következő Nokia telefont a TI ARM alapú rendszertervére építsék. A 32 bites architektúra memóriaillesztésének sajátosságaiból fakadó magasabb várható előállítási költségek miatt a Nokia az ARM processzor alkalmazása ellen érvelt. Ekkor az ARM egy új, 16 bites utasításkészlet-verzióval állt elő (Thumb, ld. később), ami csökkentette a memória alrendszerrel szemben támasztott követelményeket. Az új, kibővített architektúrára épülő processzor az 1994-ben megjelent ARM7TDMI volt, amely a TI által licenszelve az ikonikus Nokia 6110 telefon központi egysége lett, és ezt használták a következő,



3.3. ábra – Nokia 6110, az első ARM alapú mobiltelefon[24]

[24] Forrás: https://en.wikipedia.org/wiki/Nokia_6110 (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

kiemelkedően népszerű Nokia 3210 és 3310 készülékekben is. Az ARM7TDMI annyira sikeres lett, hogy azóta is ez az ARM egyik legtöbb példányban eladott processzora.

3.1.1 ISA generációk áttekintése

Az ARM utasításkészlet-architektúrákat generációkba sorolják, és ARMvX néven hivatkoznak rájuk, ahol X a verzió sorszáma. A historikusnak számító ARMv1 (ARM1 processzor), ARMv2 (ARM2, ARM3 processzorok) és ARMv3 (ARM6, ARM7 processzorok) architektúrák már nem támogatottak. Jelenleg az ARMv7 és ARMv8 architektúrákra épülnek a modern ARM processzormagok. A 3.1. táblázat áttekintést nyújt az egyes ISA változatokról, és a rájuk épülő, ARM Holdings és más cégek által fejlesztett processzormagokról (a teljesség igénye nélkül).

3.1.1.1 ISA profilok és a Cortex-A/R/M processzorcsaládok

Az ARMv7-től kezdve az ARM bevezette az A, R, és M architektúraprofilokat, amelyek a különböző felhasználási célokra szánt ISA változatokat és processzormagokat különböztetik meg.

Az **„A" profil (Application)** a „klasszikus" ARM utasításkészlet-architektúrát követi, és a nagyobb memóriaigényű és számítási teljesítményű alkalmazásokat célozza meg. Támogatja a virtuális memóriakezelést a korszerű operációs rendszerek futtatásának lehetővé tétele érdekében. Sok elterjedten használt operációs rendszer platform készült ezen processzorok számára (többféle Linux, Android, iOS, QNX, Windows Embedded), fő vadászterületük a mobiltelefonok, tabletek, smart TV-k, hálózati eszközök piaca.

Az **„R" profil (Real-time)** valós idejű alkalmazások számára nyújt megoldást nagymegbízhatóságú memóriavédelmi egységgel kiegészítve. Szintén a „klasszikus" ARM utasításkészlet-architektúrát valósítja meg. Legfőbb piaca: ipari irányítórendszerek, nagy tömegű adattárlók vezérlői, gépjárművek vezérlőegységei, orvosi berendezések, motorvezérlő elektronikák, videokamerák, robotirányító rendszerek.

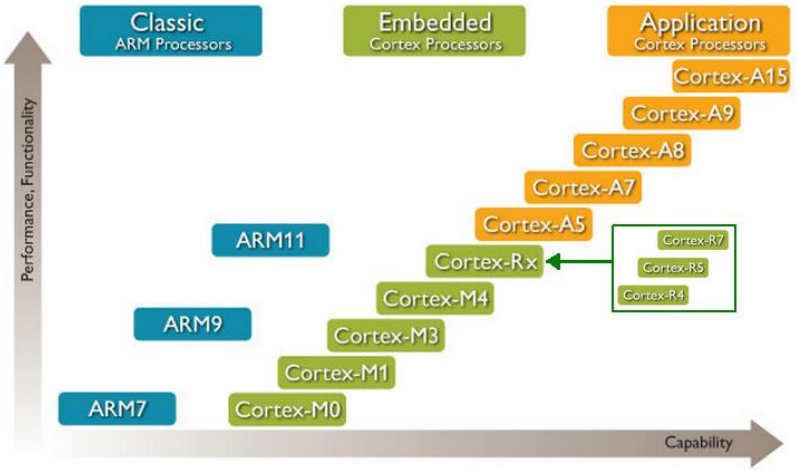
Az **„M" profil (Microcontroller)** a kifejezetten költségérzékeny beágyazott rendszerek piacára szánt egyszerűbb, kimondottan alacsony fogyasztású eszközök megvalósítását támogatja. A „klasszikus" ARM utasításkészlet-architektúrának egy módosított, kevesebb szolgáltatást nyújtó, a gyors megszakításkezelést megkönnyítő változata. Az erre épülő processzormagokat mikrokontrollerekben, FPGA-kban (soft processzorként) való alkalmazásra szánják, de megtalálhatók SoC rendszerekben is olyan mellékfunkciók megvalósítására, mint a tápmenedzsment, I/O kezelés (pl. érintőkijelző), vagy akkutöltés-vezérlés.

Copyright © 2021 / BME AUT

Év	ISA	Processzormagok (ARM Holdings)	Processzormagok (3rd party)
1985	ARMv1	ARM1	
1987	ARMv2	ARM2, ARM3	Amber
1991	ARMv3	ARM6, ARM7	
1994	ARMv4T	ARM7TDMI, ARM9TDMI	
1996	ARMv4	ARM8	DEC/Intel StrongARM, Faraday FA526
1999	ARMv5	ARM7EJ, ARM9E, ARM10E	Intel/Marvell XScale, Faraday FA626TE, Marvell Sheeva Feroceon, PJ1/Mohawk
2002	ARMv6	ARM11	
2004	ARMv7-M	ARM Cortex-M3	
2005	ARMv7-A	ARM Cortex-A5/A7, ARM Cortex-A8/A9, ARM Cortex-A12/A15/A17	Apple A6, Qualcomm Snapdragon Krait, Scorpion, Marvell Sheeva PJ4,
2007	ARMv6-M	ARM Cortex-M0/M0+/M1	
2010	ARMv7E-M	ARM Cortex-M4, Cortex M7	
2011	ARMv7-R	ARM Cortex-R4/R5/R7	
2012-2014	ARMv8-A ARMv8.1-A	ARM Cortex-A35/A53/A57/A72/A73	Apple A7/A8/A9/A10, Qualcomm Snapdragon Kryo, Nvidia Denver Samsung Exynos M1/M2/M3
2016	ARMv8.2-A ARMv8.3-A	ARM Cortex-A55/A65 ARM Cortex-A75/A76/A77/A78 ARM Cortex-X1 ARM Neoverse N1/E1	Apple A11/A12 Samsung Exynos M4/M5
2016	ARMv8-M	ARM Cortex-M23/M33	
2016	ARMv8-R	ARM Cortex-R52/R82	
2017-2019	ARMv8.4-A ARMv8.5-A ARMv8.6-A		Apple A13/A14 Apple M1

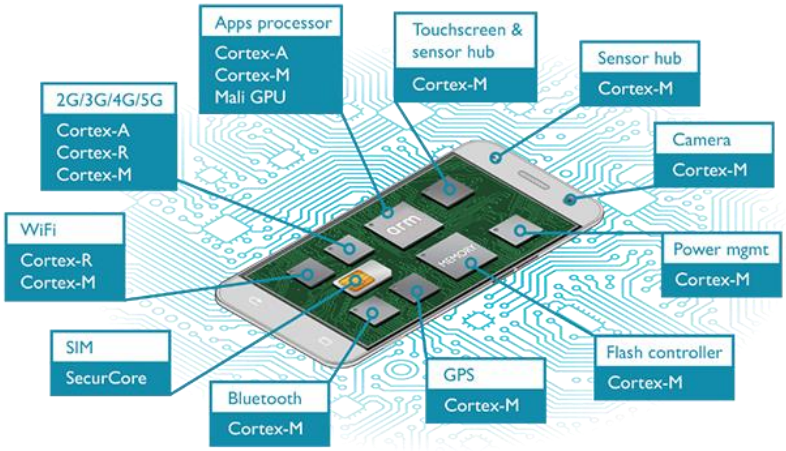
3.1. táblázat – ARM ISA generációk és processzormagok

Copyright © 2021 / BME AUT



3.4. ábra – Klasszikus és Cortex processzormagok[25]

Az ARMv7-től kezdve a különböző architektúraprofilokat megvalósító, ARM által fejlesztett processzormagokat a Cortex-A, Cortex-R és Cortex-M családokba sorolják, szakítva a korábbi elnevezési konvencióval (ld. 3.4. ábra). Bár a profilok különválasztása az ARMv7-tel indult, az ARM utólag bevezette az ARMv6-M architektúrát is (Cortex-M0/M0+/M1 processzormagok), amely valójában az ARMv7-M architektúra kevesebb utasítást tartalmazó változata. A Cortex-M magokra épülő mikrokontrollerek alacsony áruk, de 32 bites műveletvégzési képességük miatt a hagyományos 8 bites mikrokontrollereknek komoly konkurenseivé váltak. Nagybonyolultságú beágyazott rendszerekben nem ritkán mindhárom profilba tartozó processzormagokra épülő SoC-k és mikrokontrollerek is helyet kapnak, ahogy azt a 3.5. ábra is szemlélteti.



3.5. ábra – Különböző ARM Cortex processzorok egy alkalmazáson belül[26]

[25] Forrás: http://www.emcu.it/CortexFamily/CortexFamily.html (Hozzáférés: 2021.03.12)
[26] Forrás: https://developer.arm.com/architectures/learn-the-architecture/introducing-the-arm-architecture/single-page (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

A továbbiakban a nagyteljesítményű alkalmazásokra szánt „A” profil tulajdonságaira fókuszálunk. Először bemutatjuk a „klasszikus” 32 bites ARM utasításkészlet-architektúra legfőbb jellemzőit és a különböző generációktól elérhető kiegészítéseit az ARMv7-A változatig. Ezt követően pedig áttekintjük a legújabb, már 64 bites ARMv8 ISA változat újításait. Végül néhány konkrét példát láthatunk ARM Cortex-A mikroarchitektúrákra. Mindezek előtt azonban még hadd soroljunk fel néhány ismertebb eszközt, amelyek ARM Cortex-A családba tartozó processzormagot tartalmaznak:

- **Cortex-A8**: Nokia N9, Nokia N900, Samsung Galaxy S, Apple iPhone 3GS, iPhone 4
- **Cortex-A9**: Samsung Galaxy SII, SIII, Note, Note II; Apple iPhone 4S, iPad 2; Microsoft Surface; Google Nexus 7; HTC One X/X+; Xilinx Zynq-7000
- **Cortex-A15**: Samsung Galaxy S4, S5, Note 3, Note 10.1, Note Pro 12.2; Samsung Chromebook; Google Nexus 10
- **Cortex-A53**: Raspberry Pi 3; LG G4; Huawei P8, P9, P10
- **Cortex-A72**: Raspberry Pi 4; LG G5 SE; Samsung Galaxy A9 (2016); Sony Xperia X
- **Cortex-A76**: Huawei Mate 20 és P30 sorozat, Samsung Galaxy S20 és Note 20 sorozat
- **Cortex-A78** és **Cortex-X1**: Samsung Galaxy S21 sorozat

3.2 Az ARM utasításkészlet-architektúra

Az ARM architektúra a RISC elveket megfogalmazó kutatások és az első RISC processzorok (Berkeley RISC, Stanford MIPS) korában, az 1980-as évek első felében született meg, ennek köszönhetően egy nagyon „tiszta” RISC architektúrának tekinthető. Kezdetektől fogva 32 bites szóhosszúságot használt, amit az ARMv8 változatban bővítettek 64 bitre. Folyamatosan bővítették új szolgáltatásokkal, és opcionális kiegészítések is megjelentek hozzá az évek során.

3.2.1 Az ARM ISA általános jellemzői

Az eredeti „klasszikus” ARM utasításkészlet-architektúra (az ARMv7-A változatig) a következő RISC tulajdonságokkal rendelkezik:

- Load/store architektúra
- Csak 32 bites szóhatárra illeszkedő (aligned) memóriahozzáférés. (Az ARMv6-tól kezdve támogatott a non-aligned hozzáférés is.)
- Ortogonális utasításkészlet (bármely utasításban bármely címzési mód használható)
- Háromcímű utasítások
- Nagy (37 x 32 bit) regisztertár, speciális regiszterbankolással
- Egységesen 32 bites utasításszavak

Copyright © 2021 / BME AUT

Az egységes hosszúságú, szabályos felépítésű utasításkódok a RISC processzoroknál látottaknak megfelelően egyszerűbb dekódolást és csővezeték-kialakítást tesznek lehetővé a mikroarchitektúrában. Így már a korai processzoroknál is realitás volt az órajelciklusonkénti utasításkibocsátás, a modern változatokban pedig a szuperskalár kialakításnak köszönhetően jellemzők az 1-nél nagyobb IPC értékek is.

A „megszokott” RISC elveken túl az ARM ISA néhány figyelemre méltó egyéb jellemzővel is rendelkezik:

- 7 db különböző felhasználói és privilegizált üzemmód áll rendelkezésünkre, automatikus regiszterbank váltással.
- A legtöbb utasításkód (nem csak az ugró utasítások) végrehajtása feltéthez köthető. Ezzel nagymértékben csökkenthető az elágazások száma a programokban, ami hatékonyabbá teszi a pipeline műveletvégzést. Emlékezzünk, az ARM architektúra kialakításakor az elsődleges szempont az egyszerűség volt, és ebbe az elképzelésbe nagyon sokáig nem fért bele az elágazásprediktorok alkalmazása.
- Az adatfeldolgozó (pl. aritmetikai, logikai) utasításokban egy külön bit (az „S” bit) áll rendelkezésünkre annak vezérlésére, hogy az utasítás módosíthatja-e a feltételkód-regisztert (a flageket), és így hatással van-e az utána következő feltételes végrehajtású utasításokra.
- Egy lépésben történő 1–32 bites operandus eltolás lehetősége sok utasításban (ún. barrel shifter), amellyel rendkívül hatékonyan lehet aritmetikai műveleteket és cím-számításokat végezni.

3.2.1.1 Üzemmódok és regiszterkészlet

A „klasszikus” ARM utasításkészlet-architektúra 7 üzemmódot határoz meg, ezek között van 1 db nem privilegizált mód a felhasználói programok számára, és 6 db privilegizált üzemmód az operációs rendszer szolgáltatások és különböző megszakítás- és kivételkezelési taszkok számára:

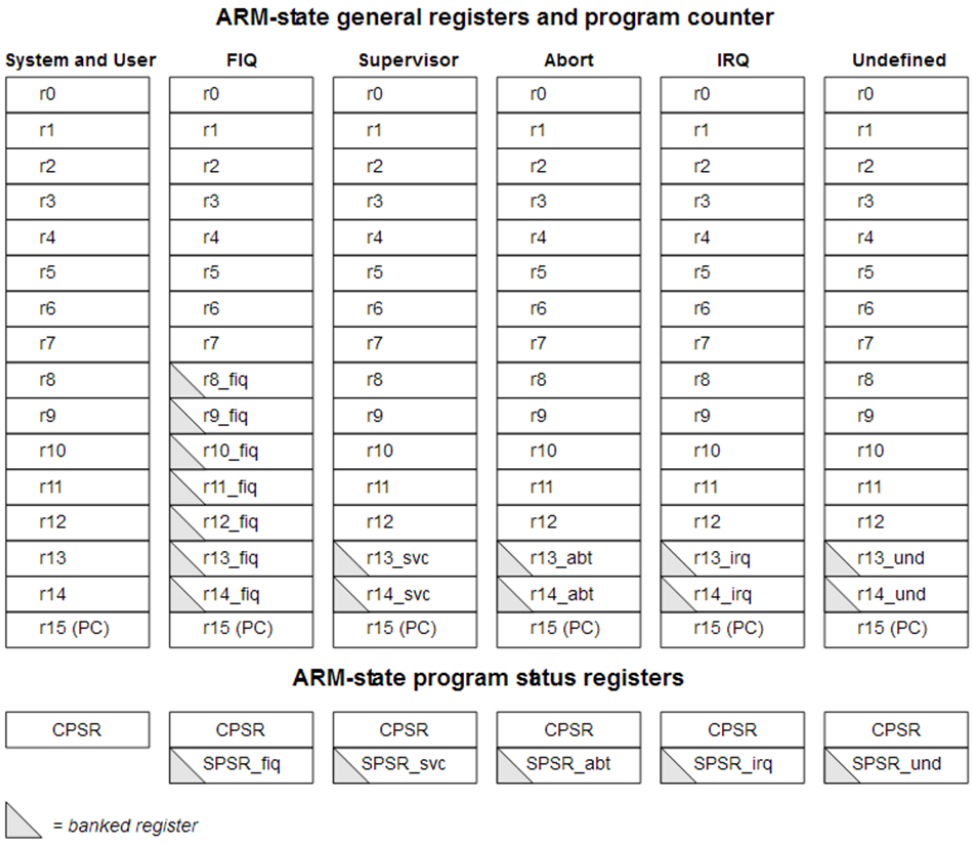
- **User** Nem privilegizált mód (a legtöbb taszk ebben fut)
- **System** Privilegizált, a „user” móddal közös regiszterkészlettel (az ARMv4-től kezdve)
- **FIQ** Magas prioritású (gyors) interrupt üzemmód
- **IRQ** Alacsony prioritású (normál) interrupt üzemmód
- **Supervisor** Reset után, ill. Software Interrupt (SWI, később SVC) utasítás hatására
- **Abort** Memóriahozzáférési kivételkezelés céljára
- **Undef** Nem definiált utasítás kivételkezelésére

Az architekturális regiszterkészlet összesen 37 regiszterből áll, üzemmódonként bankokba szervezve (ld. 3.6. ábra). Ezek között találunk

Copyright © 2021 / BME AUT

- 30 db általános célú regisztert (amelyek között vannak azért kitüntetett szerepűek, mint hamarosan látni fogjuk),
- 1 db dedikált programszámlálót (PC), amely regiszter operandusként is használható az utasításokban,
- 1 db dedikált státuszregisztert (Current Program Status Register, CPSR),
- 5 db dedikált mentett státuszregisztert (Saved Program Status Register, SPSR) a kivételkezelési üzemmódok számára.

A regiszterkészlet bankokba van rendezve, a bankváltás automatikusan történik az üzemmódváltásokkal együtt. Minden üzemmódban egyszerre 16 + 1 + (1) regiszter látszik, ezek közül *r0–r12* használható tetszőleges adattárolási célra. Az első 8 db regiszter (*r0–r7*, Low Registers) minden üzemmódban közös, és az *r8–r12* regiszterekből (High Registers) is csak az FIQ (fast interrupt) üzemmódnak van saját példánya, ami a gyors megszakításkezelést támogatja, hiszen kevesebb regisztermentést tesz szükségessé.



3.6. ábra – ARM regiszterkészlet[27]

A *r13* szolgál veremmutatóként (Stack Pointer, SP), amiből minden kivételkezelési üzemmódnak saját példánya van. A szintén teljesen bankolt *r14* neve Link Register (LR), amely

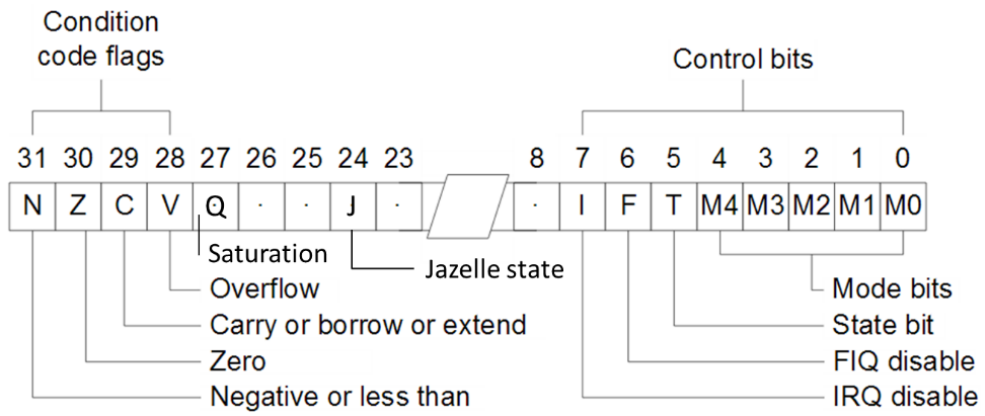
[27] Forrás: ARM7TDMI Technical Reference Manual

Copyright © 2021 / BME AUT

a gyors szubrutinhívások megvalósítását támogatja. A „szokásos” CALL utasítás céljára az ARM architektúrában a BL (Branch with Link) utasítás szolgál, amely az elágazással egyidőben a visszatérési címet az LR-be menti. Így megspóroljuk a stack használatát, ami gyorsabb működést eredményez. Egymásba ágyazott függvényhívások esetén viszont már a LR-t is mentenünk kell a stack-re.

Az *r15* a programszámlálót (PC) takarja, és értelemszerűen minden üzemmódból egyformán látszik. A PC „normál” regisztercímzéssel való elérésének előnye, hogy rugalmas elágazási megoldások valósíthatók meg vele. Például az LDR (load register) utasítással tetszőleges értéket tölthetünk bele, de lehet akár aritmetikai műveletek (pl. ADD, SUB) operandusaként is használni.

A regiszterkészlet elengedhetetlen eleme a státuszregiszter (Current Program Status Register, CPSR), amelynek felépítését a 3.7. ábrán láthatjuk. Emellett a kivételkezelési üzemmódoknak saját másolatai (Saved Program Status Register, SPSR) vannak erről, amelyek a CPSR-nek a kivétel, ill. megszakítás bekövetkezte előtti állapotát mutatják. Ez alapján mondható meg például, hogy milyen üzemmódban volt a processzor a megszakítás előtt. A státuszregiszterben megtaláljuk a szokásos flag-eket (negative, zero, carry, overflow), az üzemmódot jelző biteket (érdekes módon 5 bit van fenntartva a 7 db üzemmód számára, nyilván a bővítési lehetőségek miatt) és a globális interrupt engedélyező biteket. Bizonyos architektúraverziókban megjelentek egyéb állapotjelzők is, úgymint a Thumb vagy a Jazelle végrehajtási állapotok jelzőbitjei (ld. később) vagy a telítéses aritmetika által használt Q (saturation) bit. User módban a CPSR-nek csak egy része elérhető APSR (Application Program Status Register) néven, ami gyakorlatilag csak a feltételkód biteket tartalmazza.

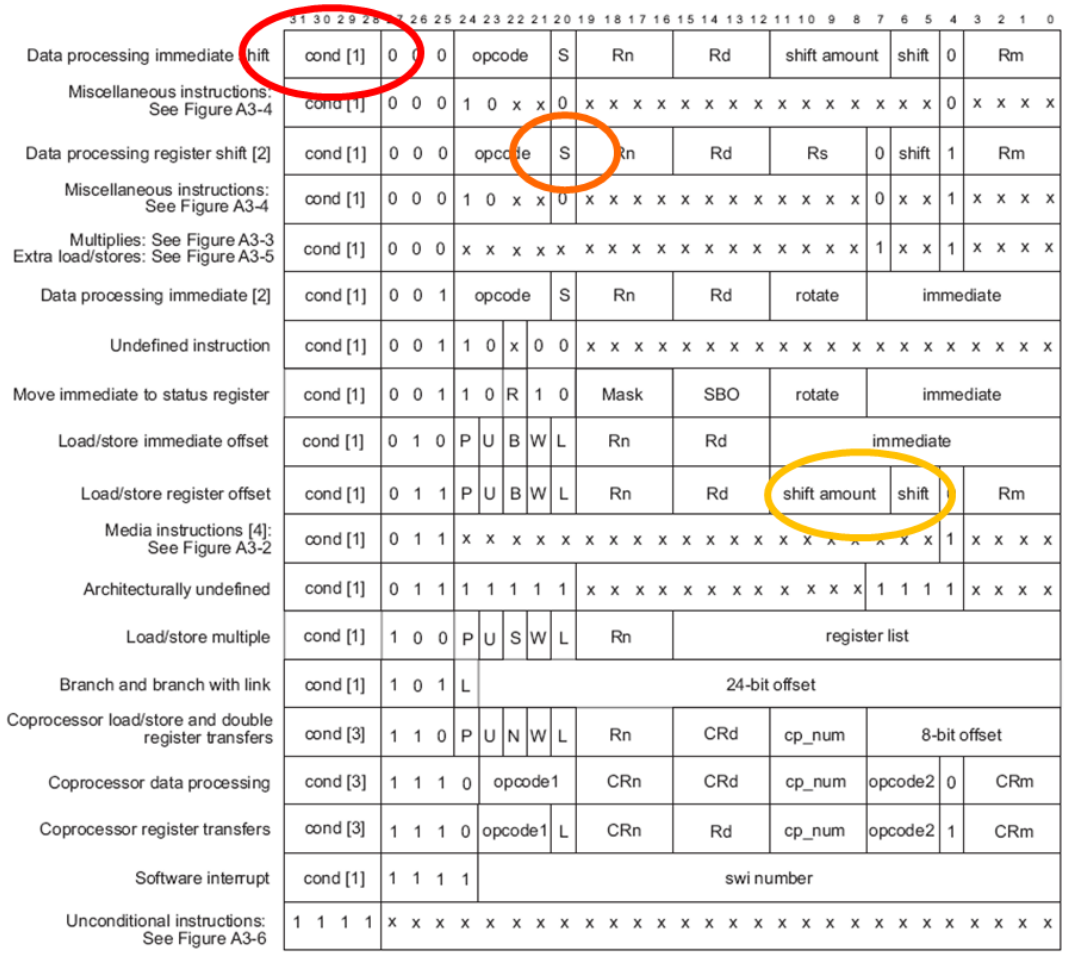


3.7. ábra – A Program Status Register felépítése

Copyright © 2021 / BME AUT

3.2.1.2 Az utasítások felépítése

A 3.8. ábra az utasításkódok felépítését mutatja, kiemelve a korábban említett jellegzetességeket. Megfigyelhető az utasítások kódolásának szabályossága. Szembetűnő, hogy minden utasítás felső 4 bitje a végrehajtás feltételkódját tartalmazza, aminek értelmezését a 3.2. táblázatban láthatjuk. De szintén észrevehetjük az adatfeldolgozó utasítások „S” bitjét, és a több helyen megtalálható operandus eltolás (barrel shifter) paramétereit.



3.8. ábra – Az ARM utasításkódok felépítése

Copyright © 2021 / BME AUT

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

3.2. táblázat – Feltételkódok

3.2.1.3 Feltételes utasítások használata

Az ugró utasításokon kívüli egyéb utasítások végrehajtásának feltételhez kötése (predication, predikatív végrehajtás) sok esetben alkalmazható előre irányuló (if-then-else típusú) elágazások helyettesítésére és összetett összehasonlítások megvalósítására. Több architektúrában is megtaláljuk ezt a megoldást, de általában csak feltételes adatmozgatások (conditional move) formájában. Az ilyen mértékű, legtöbb utasításra kiterjedő feltételes végrehajtás szinte egyedülálló az ARM architektúra esetében. A feltételes ugró utasítások kiváltásának egyik előnye a megnövelt kódsűrűség, hiszen kevesebb utasítással írható le ugyanaz a feladat. Másrészt növekszik a csővezetékes végrehajtás hatékonysága is, mivel így megspórolhatók a feltételes elágazások következtében kialakuló buborékok a csővezetékben. A feltételkódokat az assembly programban az utasítás mnemonikok utótagja jelöli, például:

```
ADDEQ r0, r1, r2          ; if zero flag set then r0 = r1 + r2
```

A feltételes utasítások használatának előnyeit jól illusztrálja az Euklideszi algoritmus megvalósítása ARM assembly nyelven. Ez az algoritmus két természetes szám legnagyobb közös osztójának (greatest common divisor, GCD) megkeresésére szolgál. Az algoritmus azon a tényen alapszik, hogy a legnagyobb közös osztó nem változik, ha a nagyobb számot a két szám különbségével helyettesítjük. Az algoritmus lényege, hogy a nagyobb számot mindaddig

Copyright © 2021 / BME AUT

helyettesítjük a két szám különbségével, amíg a két szám egyenlővé nem válik. A kapott érték az eddigi számpárok, így az eredeti számpár legnagyobb közös osztója lesz.

Illusztráljuk egy számpéldával az algoritmus működését a könnyebb megértés érdekében:

Keressük meg 609 és 462 legnagyobb közös osztóját!

1. $x_0 = 609;$ $y_0 = 462;$ $x_1 = x_0 - y_0 = 609 - 462 = 147;$ $y_1 = y_0$
2. $x_1 = 147;$ $y_1 = 462;$ $y_2 = y_1 - x_1 = 462 - 147 = 315;$ $x_2 = x_1$
3. $x_2 = 147;$ $y_2 = 315;$ $y_3 = y_2 - x_2 = 315 - 147 = 168;$ $x_3 = x_2$
4. $x_3 = 147;$ $y_3 = 168;$ $y_4 = y_3 - x_3 = 168 - 147 = 21;$ $x_4 = x_3$
5. $x_4 = 147;$ $y_4 = 21;$ $x_5 = x_4 - y_4 = 147 - 21 = 126;$ $y_5 = y_4$
6. $x_5 = 126;$ $y_5 = 21;$ $x_6 = x_5 - y_5 = 126 - 21 = 105;$ $y_6 = y_5$
7. $x_6 = 105;$ $y_6 = 21;$ $x_7 = x_6 - y_6 = 105 - 21 = 84;$ $y_7 = y_6$
8. $x_7 = 84;$ $y_7 = 21;$ $x_8 = x_7 - y_7 = 84 - 21 = 63;$ $y_8 = y_7$
9. $x_8 = 63;$ $y_8 = 21;$ $x_9 = x_8 - y_8 = 63 - 21 = 42;$ $y_9 = y_8$
10. $x_9 = 42;$ $y_9 = 21;$ $x_{10} = x_9 - y_9 = 42 - 21 = 21;$ $y_{10} = y_9$
11. $x_{10} = 21;$ $y_{10} = 21;$ $\rightarrow GCD = 21$

Tehát a két szám legnagyobb közös osztója a 21. ■

A feladat megvalósítása C nyelven a következő:

```
int gcd(int x, int y) {
   while (x != y)          /* while x and y are not equal */
      if (x > y) x -= y;   /* substract the smaller from
      else y -= x;            the greater                 */
   return x;               /* if x and y are equal the
}                             solution has been found     */
```

Vegyük észre, hogy a fenti algoritmus minden egyes kivonást követően egy feltételes elágazást hajt végre. ARM assembly-ben a fenti algoritmus viszont így írható le (x = Rx, y = Ry):

```
gcd:    CMP    Rx, Ry       ; set condition "NE" if (x != y),
                            ;               "GT" if (x > y),
                            ;               "LE" if (x =< y)
        SUBGT  Rx, Rx, Ry   ; if "GT" (Greater Than), x = x-y;
        SUBLE  Ry, Ry, Rx   ; if "LE" (Less Than Or Equal), y = y-x;
        BNE    gcd          ; if "NE" (Not Equal), then loop
```

Copyright © 2021 / BME AUT

Mivel a feltételkódok széleskörűen befolyásolják az utasítások működését, nem lenne célszerű, ha a végrehajtás alapértelmezetten mindig állítaná a státusz flag-eket. Például, ha a fenti programban az első kivonás szokásos módon megtenné ezt, az hatással lenne a második (feltételes) kivonás működésére, ami nyilvánvalóan a program hibás működéséhez vezetne. Emiatt az adatfeldolgozó (aritmetikai, logikai, adatmozgató) utasítások alapértelmezésben nem állítják a státusz flag-eket. Minden ilyen utasításnak van viszont egy flag-módosító változata is, amelyet az utasítás neve utáni „S” kiegészítés jelöl, például:

```
ADDS r0, r1, r2          ; r0 = r1 + r2 and set flags
```

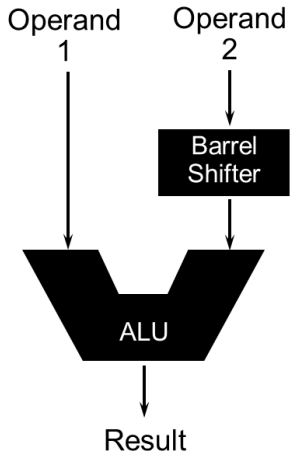
3.2.1.4 Barrel eltoló

Az utasításkészlet érdekessége az adatfeldolgozó utasításokban közvetlenül elérhető barrel shifter funkcionalitás. Ez azt jelenti, hogy a nem kifejezetten eltolásra vagy forgatásra használatos műveletek esetén is a második forrásoperandushoz megadható egy eltolási érték, és az adatfeldolgozási művelet már a módosított operanduson hajtódik végre (ld. 3.9. ábra). Például a C nyelvű

```
a += (j << 2);
```

kifejezés egyetlen ARM utasítással megvalósítható:

```
ADD Ra, Ra, Rj, LSL #2
```



3.9. ábra – Barrel eltoló

Itt az eltolást jelző LSL #2 (logical shift left) kifejezés a második forrásoperandus részének tekintendő. Ez az operandus a következők egyike lehet:

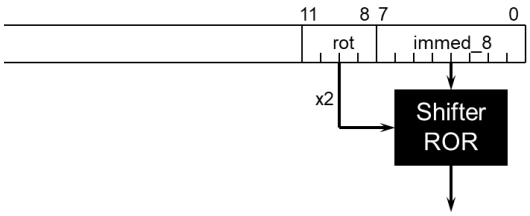
- Regiszter, 5 bites eltolási értékkel (mint a példában)
- Regiszter, egy másik regiszter alsó 8 bitjével eltolva
- Közvetlen (immediate) érték, amit egy páros számú bittel jobbra eltolt 8 bites számmal (0…255) adhatunk meg

3.2.1.5 Közvetlen (immediate) konstansok

Mivel az ARM utasításkódok egységesen 32 bitesek, limitált az utasításokba ágyazható közvetlen konstansok mérete. A 3.8. ábrát tanulmányozva észrevehetjük, hogy az opkódok 12 bitet biztosítanak az immediate értékek számára, ebből a legtöbb esetben 8 bit maga a szám, kiegészítve egy 4 bites eltolási információval (ami történetesen jobbra forgatást jelent,

Copyright © 2021 / BME AUT

ld. 3.10. ábra). Ez alól vannak kivételek, például, ha MOV utasítással akarunk konstans értéket írni egy regiszterbe, ott 16 bites immediate értéket használhatunk. A 12 bites konstansoknál az eltolás a felső 4 biten megadott (0–15 közé eső) érték kétszeresével (0–30 közötti páros számmal) történik, ahogyan az előző pontban is utaltunk rá. Például a 0x23000000 érték az assembly kódban „0x23 ROR 8” alakban adható meg (figyeljük meg, hogy a hasznos bitek jobbra kilépnek, de a forgatás hatására a magas helyértékeken újra megjelennek). Ez esetben az utasításba ágyazott 12 bit értéke 0x423 lesz (bináris alakban 0100 0010 0011).



3.10. ábra – Utasításba ágyazott közvetlen konstans megadása

Ha a 12 bitet közvetlenül használnánk fel, akkor csak a –2048 és +2047 közötti tartomány állna rendelkezésre a közvetlen operandusok számára. Ez nem lenne célszerű, hiszen nagy számok is gyakran szükségesek a műveleteinkhez (pl. 32 bites báziscím egy összetett címszámításhoz, vagy egyszerűbb felépítésű, de teljes méretű bitmaszkok mint pl. 0x00FF0000). A vizsgálatok azt mutatták, hogy az alkalmazások nagy részében a konstansok 50%-a –16 és +15 közé esik, 90%-a pedig –512 és +511 közé. A nagy értékű konstansok esetén viszont az alacsony helyértékű bitek nagyon gyakran 0 értékűek.

Ha a 8+4 bites eltolt formátummal nem írható le a használni kívánt konstans érték, akkor külön egy (32 bites) regiszterbe kell betöltenünk, hogy művelet tudjunk végezni vele. Mint láttuk, a MOV utasítás is csak legfeljebb 16 bites immediate konstanst tesz lehetővé, amit a későbbi műveletben egy 5 bites eltolási értékkel módosíthatunk a beépített barrel shifter segítségével. Ha a kívánt 32 bites érték nem írható ezzel az eltolt formátummal, akkor a memóriából kell betöltenünk az erre szolgáló load register (LDR) utasítással. Annak érdekében viszont, hogy a 32 bites konstansokat programozói szempontból egységesen kezelhessük értéktől függetlenül, az assembler LDR pszeudoutasítása siet segítségünkre:

```
LDR rd, =const
```

A konstans értéke határozza meg, hogy ebből milyen tényleges utasítás generálódik a tárgykódban. Ha belefér a 16 bites formátumba, akkor egyszerű MOV utasítás lesz belőle, például:

```
LDR r0,=0xFF                →           MOV r0,#0xFF
```

Copyright © 2021 / BME AUT

Általános 32 bites érték esetén viszont tényleges LDR utasítás valósul meg PC relatív címzéssel az automatikusan generált, ún. literal pool-ból:

```
LDR r0,=0x55555555          →          LDR r0,[PC,#Imm12]
                                       …
                                       …
                                       DCD 0x55555555
```

A DCD assembler direktíva 32 bites adat elhelyezésére szolgál a programkódban (Define Constant Double, ahol a double a „hagyományos” értelemben vett duplaszóra, tehát 32 bitre utal). Az LDR utasításnál az #Imm12 a letárolt érték címének ofszet értéke az aktuális utasítás címéhez (vagyis a PC pillanatnyi értékéhez) képest a már bemutatott 12 bites formátumban.

3.2.2 Az ARM ISA kiegészítései

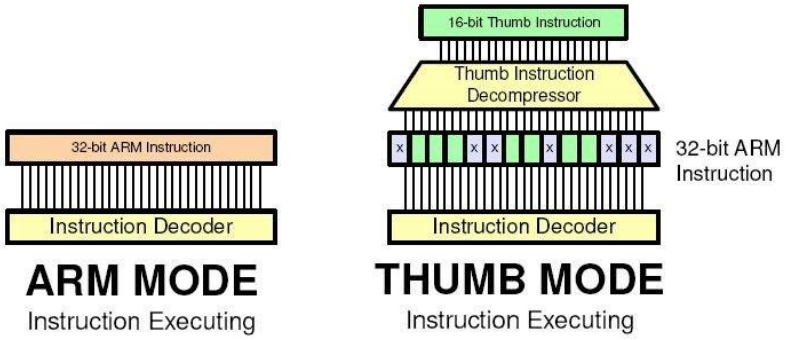
Az ARM architektúra fejlődése során újabb és újabb kiegészítések jelentek meg az „alap” utasításkészlethez, amelyek mindig egy-egy célalkalmazás igényeit célozták meg. Ezek között voltak olyanok, amelyek a későbbi változatok szerves részeivé váltak, és olyanok is, amelyek opcionálisak maradtak. Az ISA kiegészítéseket a következő három csoportba sorolhatjuk:

- Alternatív végrehajtási módok kódméret csökkentés vagy speciális nyelvi támogatás céljából (Thumb, Thumb-2, Jazelle, ThumbEE)
- Számítási teljesítmény növelése (DSP utasítások, VFP, NEON)
- Biztonság és virtualizáció (TrustZone, Virtualization)

3.2.2.1 Thumb és Thumb-2 végrehajtási módok

Az ARMv4T ISA verzióban vezették be a Thumb végrehajtási állapotot, és az ARM7TDMI volt az első processzormag, amely megvalósította ezt (mindkét elnevezésben a T betű a Thumb módra utal). A Thumb egy 16 bites utasításszavakból álló tömörített utasításkészlet hozzáadását jelenti, miközben az architektúra adategysége továbbra is 32 bit. A 16 bites utasításkódok a „hagyományos” 32 bites ARM utasítások tömörített változatai, bizonyos bitmezők (és ezzel bizonyos szolgáltatások) elhagyásával. A rövidebb utasításkódokat egy, az utasításbetöltési fázis után elhelyezkedő dekompresszor egység „bővíti fel” az eredeti ARM utasításokra a hiányzó részek alapértelmezett bitekkel való kitöltésével (ld. 3.11. ábra). A csővezeték további része (a dekódolástól kezdve) változatlan maradt. Így kevesebb funkció érhető el egy-egy utasítás által, például csak az elágazások köthetők feltételhez, az általános regiszterek közül csak az alsó 8 db használható, és sok utasításkód nem lehet háromcímű. A rövidebb utasításkódok használatával növekszik a kódsűrűség, annak ellenére, hogy néhány művelet megvalósításához esetleg több utasítás szükséges. Az ötletet leginkább úgy lehetne jellemezni, hogy a bő (és sokszor kihasználatlan) lehetőségek helyét megspóroljuk, viszont, ha mégis szükség lenne a hiányzó lehetőségre, akkor azt csak két utasítás használatával tudjuk pótolni.

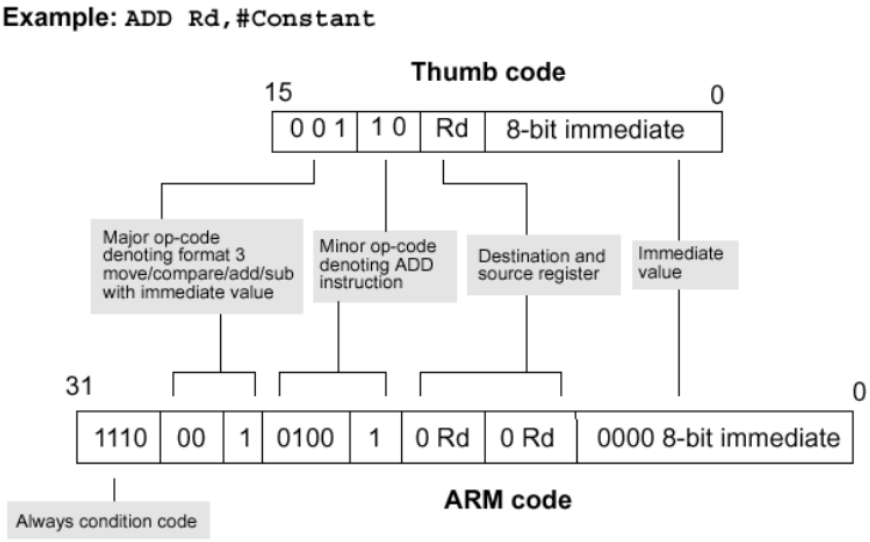
Copyright © 2021 / BME AUT



3.11. ábra – Utasítások dekódolása ARM és Thumb módban

Fentiek könnyebb megértésére egy konkrét példát mutat a 3.12. ábra. Egy konkrét 16 bites Thumb utasítás (ADD Rd, #konstans) „átalakítása” az ARM utasításkészlet 32 bites megfelelőjére a következőképpen történik:

A Thumb utasítás első 3 bitje a művelet utasításcsoportjára utal (a „001” jelentése Data Processing and FSR Transfer), ezt követi 2 biten a csoporton belüli azonosító (ADD esetén „10”). Ezután következik 3 biten az Rd regiszter azonosítója, majd az utasításba beépülő 8 bites immediate konstans.



3.12. ábra – Thumb utasítás kitömörítése

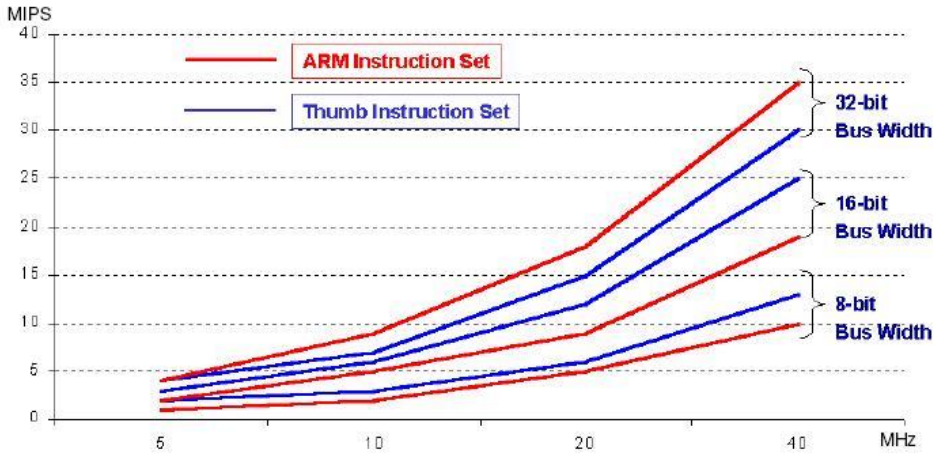
Az utasítás 32 bites megfelelője az ARM utasításkészletben a következő:

- A 31–28. biteken található a minden utasításra jellemző, a végrehajtás feltételét előíró feltételkód, ami ez esetben mindig „always” (AL), vagyis a feltétel nélküli változat.
- A 27–25. biteken a művelet utasításcsoport-besorolása található a Thumb kódhoz hasonlóan.

Copyright © 2021 / BME AUT

- A 24–21. bitek azonosítják a csoporton belül a műveletet (ADD=0100).
- A 20. bit az S vezérlőbit (a művelet állítsa-e a flageket), fixen „1” értékkel.
- A 19–16. és a 15–12. bitek a cél- és forrásregisztereket azonosítják. Mivel Thumb módban nincs háromcíműség, mindkét mezőbe az Rd regiszter azonosítója kerül.
- A 11–0. bitek tartalmazzák az utasításba beépülő 12 bites konstanst. Ebből az alsó 8 bit a számérték, a 4 bites eltolási érték pedig fixen 0 lesz.

Különösen azokban a helyzetekben, amelyekben az utasításmemória buszszélessége kevesebb, mint 32 bit-re van korlátozva, a rövidebb Thumb utasításkódok lényegesen jobb teljesítményt nyújtanak, mint az eredeti 32 bites utasítások, a limitált memória-sávszélesség jobb kihasználása miatt. A beágyazott hardvereknek sok esetben nincs 32 bites adatbusza, inkább 16 bites, vagy még kevesebb (emlékezzünk például a Nokia kezdeti érvelésére az 1990-es évek elején a 32 bites ARM architektúra mobiltelefonokban való alkalmazása ellen). A rövidebb utasításokkal nagyobb kódtömörség érhető el, hiszen, ha minden ARM utasítás helyett 2 db Thumb utasítást használnánk, akkor kapnánk meg az eredeti kódméretet. Ilyen esetekben komoly előnyt jelent a Thumb utasításkészletre való fordítás, esetleg a CPU-t inkább igénybe vevő részeket át kell írni a 32 bites ARM utasításkészletre, és utóbbiakat a 32 bites buszszélességű memóriába kell elhelyezni. A 3.13. ábra diagramján is látszik, hogy az alkalmazott buszméret függvényében az ötlet beváltja a hozzá fűzött reményeket: 8 vagy 16 bites busz esetében a Thumb, 32 bites busznál az eredeti ARM utasításkészlet alkalmazható hatékonyabban. Ezt a megállapítást még tovább árnyalhatja a kód tömörsége (mérete), amit a diagram nem tartalmaz.



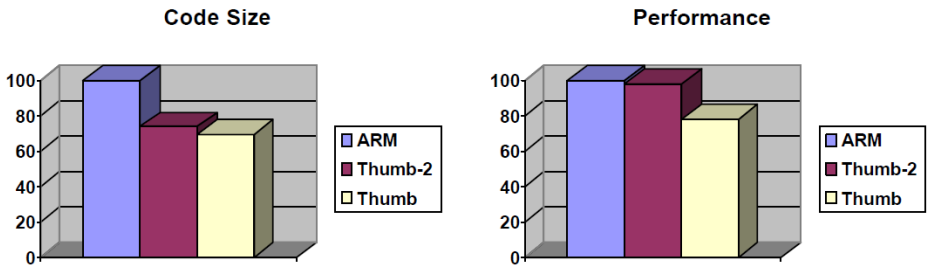
3.13. ábra – Utasítások végrehajtási sebessége az adatbusz méretének függvényében

A Thumb és az ARM végrehajtási állapotok között kitüntetett ugró utasításokkal (BX – Branch and eXchange, BLX – Branch with Link and eXchange) lehet váltani, ahol a célcím legalsó bitje dönti el, hogy az ugrás után ARM (1) vagy Thumb (0) állapotban folytatódik a végrehajtás. Mivel ARM módban 32 bites, Thumb módban pedig 16 bites utasításszavakkal dolgozunk, garantált, hogy az utasítások mindig páros címre igazítottak, azaz a legalsó címbit fixen 0-nak tekinthető, így nyugodtan felhasználható a következő végrehajtási állapot jelzésére a PC

Copyright © 2021 / BME AUT

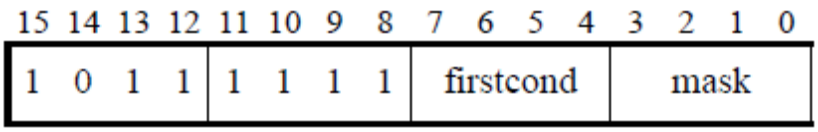
módosításakor. A kitüntetett ugró utasításokon kívül a PC-t közvetlenül felülíró MOV (move) vagy LDR (load) utasítások is használhatók üzemmódváltással egybekötött elágazásra. A következő üzemmódot itt is a PC-be írt érték legalsó bitje jelzi.

A Thumb utasításkészlet továbbfejlesztéseként vezették be a Thumb-2 technológiát az ARMv6T2 ISA verzióban, az első ezt alkalmazó processzormag az ARM1156T2-S volt 2003-ban. Ez a Thumb 16 bites utasításkódjait egészíti ki 32 bites utasításokkal úgy, hogy az utasításkészlet átfogóbb legyen. A 16 és 32 bites utasításszavak keverve szerepelnek a programfolyamban. Így a Thumb-2 megközelítőleg a Thumb kódsűrűségével (az ARM programkód méret 74%-a) és az ARM utasításkészlet sebességével (98%) bír. A Thumb-2 mind az ARM, mind a Thumb utasításkészlethez képest tartalmaz új utasításokat, így például bitmező módosításokat, ugrótáblákat (elágazási tábla) és a feltételes végrehajtás új formáját. A 3.14. ábra az ötlet tényleges programokon mért hatékonyságát mutatja mind a kódsűrűség, mind a teljesítőképesség függvényében.



3.14. ábra – ARM, Thumb és Thumb-2 végrehajtási módok összehasonlítása

Külön említést érdemel a Thumb-2 utasításkészletben a feltételes utasítás végrehajtás visszahozása. A Thumb utasításkód tömörsége „magával vitte” az egyes utasítások feltételhez köthetőségét, ami pedig nagyon hatékonyan működik az ARM utasításkészletben. Ezért vezették be az IT, vagyis az if-then-else szerkezetet definiáló utasítást, amelynek szerepe annyi, hogy kijelölje a feltételt az őt követő néhány utasítás számára. Felépítését a 3.15. ábra mutatja, ahol a *firstcond* mező az ARM utasításoknál megismert 4 bites feltételkódot tartalmazza (ez a teljes szerkezetre vonatkozik), míg a *mask* mező az utasítást követő 1–4 utasítás feltételes végrehajtását definiálja. Ez utóbbi egy speciális kódolással azt tartalmazza, hogy követi-e az első feltételes utasítást második-harmadik-negyedik, és hogy azokat a megadott feltételkód (then), vagy annak az ellenkezője (else) szerint kell-e végrehajtani.



3.15. ábra – Az IT utasítás felépítése

Copyright © 2021 / BME AUT

Egy példa segítségével hasonlítsuk össze az ARM, a Thumb és a Thumb-2 által megvalósítható feltételes szerkezetek hatékonyságát kódtömörség és futásidő szempontjából. Az egyszerű példa pszeudokódja a zero flag értékétől függően különböző összeadási műveleteket ír elő:

```
if (Z = 1)
then R0 = R3 + [R1]
else R0 = R4 + [R2]
```

Az ARM, Thumb és Thumb-2 implementációkat a 3.16. ábra szemlélteti. Láthatjuk, hogy a leggyorsabb kódot az ARM utasítások eredményezik, hiszen az elágazás nélküli 4 db utasítás teljes sebességgel át tud haladni a csővezetéken. A Thumb utasításkészletben csak körülményesebben, ugró utasításokkal tudjuk leírni a feladatot, ez több utasítást, de a feleakkora szóhosszúságnak köszönhetően kisebb kódméretet eredményez. Az ugró utasítások miatt a végrehajtás ideje bizonytalanabbá válik, mert téves elágazásjövendölés esetén megjelennek a „buborékok” a csővezetékben. A Thumb-2-nél az IT utasítás alkalmazása növeli csak eggyel az utasítások számát az ARM-hoz képest, és a 16 bites szóhossz következtében ez eredményezi a legtömörebb kódváltozatot. A végrehajtás hatékonyságát nem korlátozzák ugró utasítások, a plusz 1 db utasítás csak egy órajelciklussal nyújtja meg a futásidőt.

ARM	Thumb	Thumb-2
LDREQ R0, [R1] LDRNE R0, [R2] ADDEQ R0, R3, R0 ADDNE R0, R4, R0	BNE L1 LDR R0, [R1] ADD R0, R3, R0 B L2 L1 LDR R0, [R2] ADD R0, R4, R0 L2	ITETE EQ LDR R0, [R1] LDR R0, [R2] ADD R0, R3, R0 ADD R0, R4, R0

Kódméret	4 x 32 bit = 16 bájt	6 x 16 bit = 12 bájt	5 x 16 bit = 10 bájt
Ciklusok száma	4 T	4–20 T (elágazásbecsléstől függ)	5 T

3.16. ábra – Az ARM, Thumb és Thumb-2 utasításkészletek feltételes elágazási eszköztárának összehasonlítása

Az ARM assembler az ún. Unified Assembly Language (UAL) segítségével lehetővé teszi, hogy ugyanabból a forráskódból ARM és Thumb-2 formátumú tárgykódot is generálhassunk. Ehhez például azt kell megtennie a programozónak (ill. a fordítónak), hogy mindig használja az IT utasítást az if-then-else jellegű szerkezeteknél, illetve az IT-t követő utasításoknál is kiírja a megfelelő feltételkód-utótagokat (annak ellenére, hogy az ARM utasításkészletben nincs IT, a Thumb-2-ben pedig az utasítások nem rendelkeznek feltételkód bitekkel). Lássunk erre két példát:

Copyright © 2021 / BME AUT

Egyszerű feltételes adatmozgatás:

if R0 == R1, then R0 = R2 else R0 = R3

```
CMP    R0, R1   ; Set flags
ITE    EQ       ; ARM: no code ...; Thumb2: IT instruction
MOVEQ  R0, R2   ; ARM: conditional; Thumb2: cond. via ITE 'T' (then)
MOVNE  R0, R3   ; ARM: conditional; Thumb2: cond. via ITE 'E' (else)
```

Abszolútérték képzése:

R0 = abs(R1)

```
MOVS   R0, R1        ; R0 = R1 and set flags according to R1
IT     MI            ; if MI (negative), then execute the next
                     ; instruction, else skip it
RSBMI  R0, R0, #0    ; R0 = 0 – R0
```

(Az RSB utasítás jelentése Reverse Substract, azaz fordított operandus-sorrendű kivonás.) ■

3.2.2.2 Jazelle

A mobiltelefonos alkalmazásokban széles körben elterjedt Java nyelv támogatását célozta meg a **Jazelle DBX** (Direct Bytecode eXecution) kiegészítés, amely az ARMv5TEJ ISA verzióban jelent meg először. Az első processzormag, amelyben implementálták, az ARM926EJ-S volt 2001-ben. (Mindkét elnevezésben a J betű utal a Jazelle jelenlétére.) A Jazelle DBX a Java bájtkód közvetlen futtatását tette lehetővé hardverben, ezzel nagyban felgyorsítva az egyébként erőforrásszegény mobil eszközökön a Java virtuális gépek (JVM-ek) alkalmazását. Egy harmadik végrehajtási állapotot jelentett az ARM és Thumb módok mellett. A kiegészítés a szabványos Java bájtkódnak egy (publikusan nem dokumentált) részhalmazát valósította meg egy plusz csővezeték-fokozat beiktatásával az utasításbetöltés és dekódolás között, amely a felismert bájtkódokat egy vagy több ARM utasítássá alakította. A Jazelle módba való belépésre a BXJ (Branch and eXchange Jazelle) ugró utasítás szolgált.

Az ARMv7-A architektúrában bevezették a **Jazelle RCT** (Runtime Compilation Target), vagy másnéven **ThumbEE** (Thumb Execution Environment) kiegészítést, amelynek szándékolt célja a Jazelle DBX háttérbe szorítása, illetve kiváltása volt (azzal együtt, hogy minkettő megfért egymás mellett az architektúrán belül). Az ARMv7-A alapú processzormagokban kötelező az implementációja, először a Cortex-A8 magban jelent meg 2005-ben. A ThumbEE egy negyedik végrehajtási állapotot jelent, amely gyakorlatilag a Thumb-2 módosított változata, és futásidőben generált kód (JIT compilation) futtatásának támogatására szolgál felügyelt futtatási környezetben. Olyan szolgáltatásokat tartalmaz (pl. nullpointer ellenőrzés, új

Copyright © 2021 / BME AUT

hibakezelési mechanizmusok), amelyek lehetőséget nyújtanak a JIT fordítóknak arra, hogy tömörebb kódot generáljanak a teljesítmény csökkenése nélkül. Ez az újabb Jazelle variáns modernebb megközelítést képvisel az elődjénél, mivel ma már a Java virtuális gépek is leginkább a JIT fordításra épülnek a hagyományos bájtkód interpretálás helyett. Ugyanakkor a Jazelle RCT nem kizárólag Java specifikus, hanem célnyelvként szolgál C#, Perl, Python és más futtatókörnyezetek számára is.

Fontos megemlítenünk, hogy az ARMv8-A ISA verzióban megszüntették mindkét Jazelle variáns támogatottságát.

3.2.2.3 DSP utasítások

Az ARMv5TE változatban vezettek be először olyan új utasításokat, amelyek jelfeldolgozó processzorokban (digital signal processor, DSP) széles körben használatosak. Ide tartoznak az összevont szorzó és összeadó (multiply-accumulate, MAC), valamit a telítéses aritmetikai utasítások (ld. 3.3. táblázat).

Instruction	Operation	Purpose
SMLAxy{cond}	16 x 16 + 32 → 32	Signed MAC
SMLAWy{cond}	32 x 16 + 32 → 32	Signed MAC wide
SMLALxy{cond}	16 x 16 + 64 → 64	Signed MAC long
SMULxy{cond}	16 x 16 → 32	Signed multiply
SMULWy{cond}	16 x 32 → 32	Signed multiply long
QADD Rd, Rm, Rs	SAT(Rm + Rd)	Saturating add
QDADD Rd, Rm, Rs	SAT(Rm + SAT(Rs x 2))	Saturating add double
QSUB Rd, Rm, Rs	SAT(Rm – Rd)	Saturating subtract
QDSUB Rd, Rm, Rs	SAT(Rm – SAT(Rs x 2))	Saturating subtract double
CLZ{cond} Rd, Rm	COUNTZ(Rm)	Count leading zeros

3.3. táblázat – DSP utasítások az ARM architektúrában

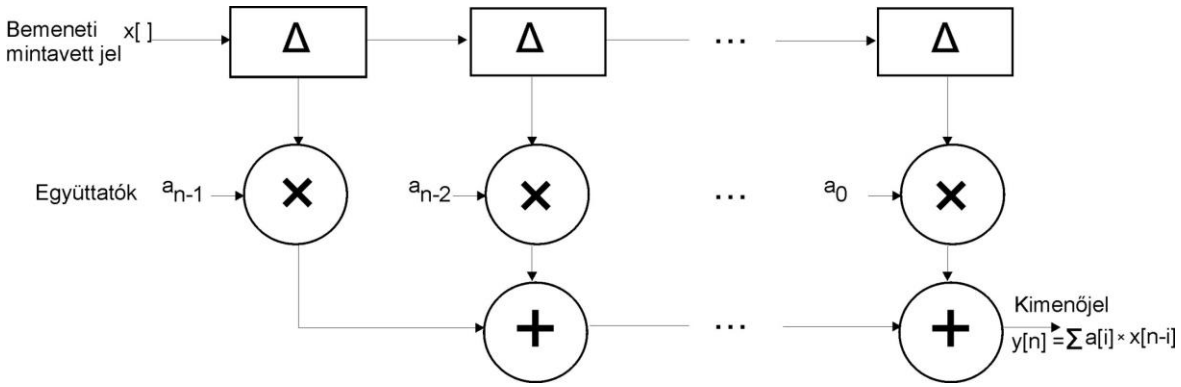
A MAC utasítások jelfeldolgozó algoritmusokban nagyon hasznosak, a különböző szűrők (FIR, IIR), a konvolúció és a Fourier transzformáció intenzíven használja a szorzás-összeadás műveleteket. Példaként tekintsük a FIR (finite impulse response) szűrőt (3.17. ábra), amelynek lényege, hogy minden mintavételi pontban a bemeneti jelsorozat előző valahány mintáját a szűrőegyütthatókkal súlyozva összegezze, és így hozza létre a kimeneti jelet:

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + a_3 x[n-3] + \cdots \qquad (3.1)$$

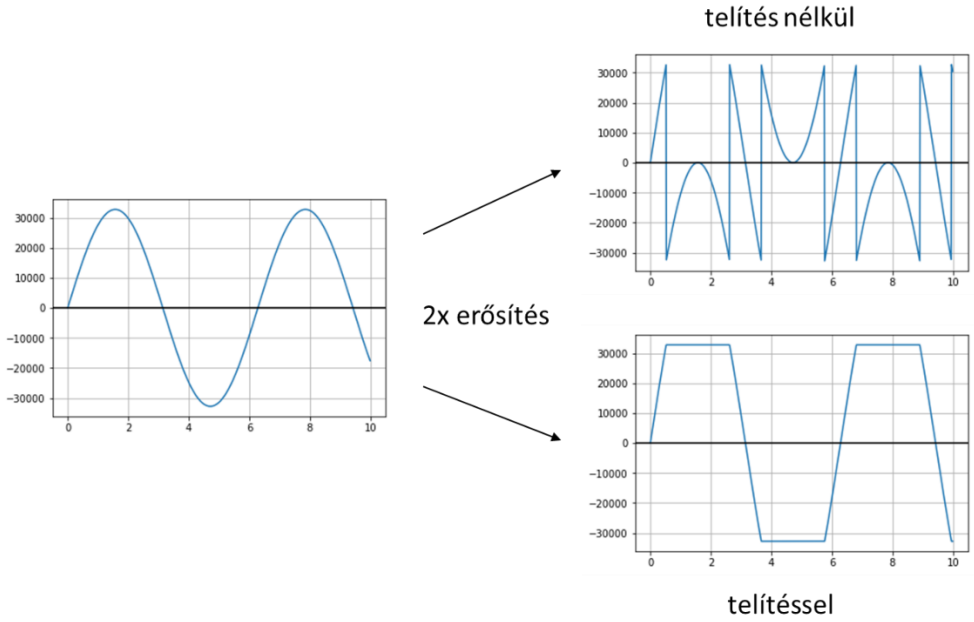
A telítéses aritmetikai műveleteket (összeadás, kivonás) elterjedten használják audio és video kodekekben, de más jelfeldolgozási alkalmazásokban is nagyon hasznosak. Lényegük, hogy megakadályozzák egy aritmetikai művelet során az eredmény túl- vagy alulcsordulását, ehelyett az ábrázolható számtartomány legnagyobb, ill. legkisebb értékét szolgáltatják. A

Copyright © 2021 / BME AUT

3.18. ábra egy túlvezérelt jelerősítés példáján mutatja be a telítés nélküli és a telítéses aritmetika közötti lényeges különbséget. Ezek az utasítások a státuszregiszter Q bitjében jelzik a telítés bekövetkeztét. Hasznos eszköz még a CLZ (Count Leading Zeros) utasítás, amely egy regiszterben a legmagasabb helyértékű 1-es bit felett található 0 értékű bitek számát adja vissza. Ez információt ad arról, hogy milyen messze vagyunk a telítődés határától, ami például normalizálás esetén hasznos.



3.17. ábra – FIR szűrő vázlata



3.18. ábra – Telítéses aritmetika használata a jelfeldolgozásban

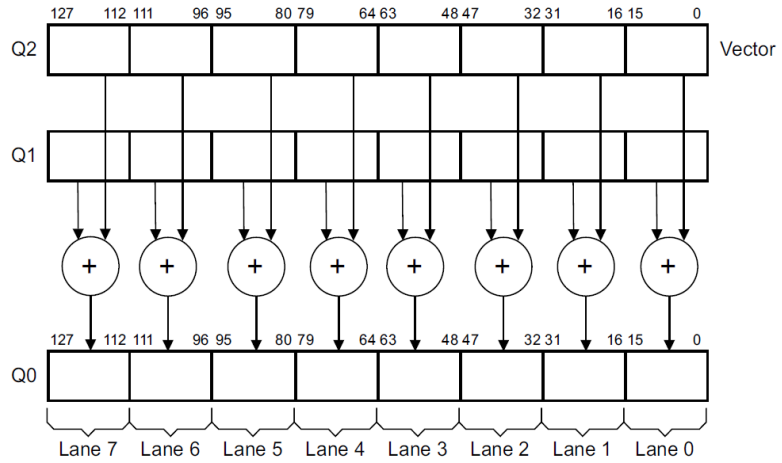
Copyright © 2021 / BME AUT

3.2.2.4 Vector Floating Point (VFP)

A Vector Floating Point kiegészítés lebegőpontos koprocesszor funkcionalitást ad az alap utasításkészlethez, amely először az ARMv5 ISA verzióban jelent meg. Az IEEE-754 lebegőpontos szabványnak megfelelő működést biztosít, egyszeres (32 bit) és dupla (64 bit) pontosságú számokkal való műveletvégzést támogat. Összeadást, kivonást, szorzást, osztást és négyzetgyökvonást is megvalósít, ezekhez verziótól függően 16 vagy 32 db dedikált 64 bites regisztert használ. Vektor módban SIMD (single instruction multiple data) jellegű műveletvégzést is támogatott, azonban szekvenciálisan dolgozta fel a vektor elemeit. A vektor módot későbbi verziókban megszüntették, mert más SIMD ISA kiegészítések is készültek, amelyek valódi párhuzamos feldolgozást tettek lehetővé. A VFP nevet a kiegészítés későbbi verziói is megtartották, de csak skalár lebegőpontos műveleteket támogattak.

3.2.2.5 Advanced SIMD Extension (NEON)

Az ARMv7-A ISA verzióban jelent meg NEON néven egy univerzális, valódi párhuzamos SIMD funkcionalitást biztosító koprocesszor kiterjesztés, amely egész és lebegőpontos vektorműveleteket, valamint a DSP műveletek vektoros megfelelőit támogatja. A SIMD utasítások 64 vagy 128 bites vektorokon hajthatók végre, amelyek akár 16 db 8 bites, 8 db 16 bites vagy 4 db 32 bites értéket tartalmazhatnak (64 bites vektorok esetén ennek a felét). A 32 bites maximális vektorelem-méretből következik, hogy a lebegőpontos műveleteknél csak az egyszeres pontosságot támogatja. Tudni kell róla azt is, hogy csak részben valósítja meg az IEEE-754 szabványban foglaltakat, például nem kezeli a denormalizált formátumot (a denormalizált számokat 0-ra kerekíti). Emiatt a NEON nem váltja ki teljesen a VFP funkcionalitását, általában a kettő együtt kerül alkalmazásra.



3.19. ábra – Advanced SIMD műveletvégzés elve az ARM processzorokban[28]

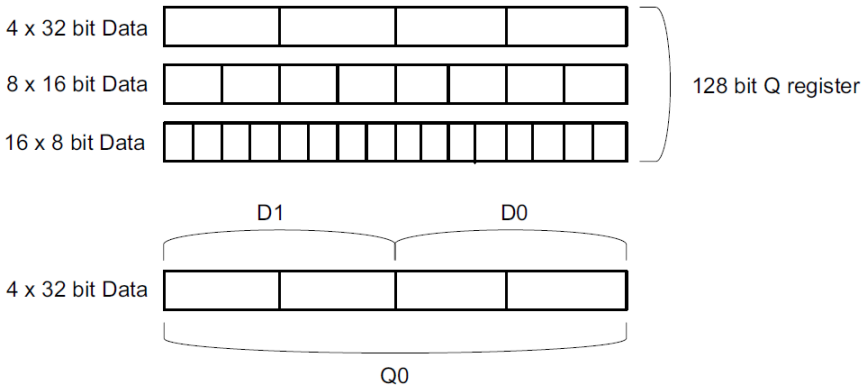
[28] Forrás: NEON Programmer's Guide Version: 1.0

Copyright © 2021 / BME AUT

A NEON 32 db dedikált 64 bites vektor regisztert ad az architektúrához („D” regiszterek), amelyek páronként összevontan 128 bites regiszterekként is használhatók („Q” regiszterek), ahogyan a 3.20. ábrán is látható.

A NEON funkcionalitását a mikroarchitektúrában egy elkülönített végrehajtóegységként valósítják meg (media processing engine). Nagyon jól alkalmazható média- és jelfeldolgozási feladatokban, például audio és video kodekek esetén. A NEON segítségével az MP3 audio dekóder már 10 MHz-es órajel mellett futtatható, valamint 13 MHz-es órajellel a beszédhang kódolására szolgáló GSM AMR (Adaptive Multi-Rate) kodek is megvalósítható.

Hozzá kell tennünk, hogy a NEON előfutáraként az ARMv6 architektúrához is elérhető volt már egy SIMD kiegészítés, de szűkebb funkcionalitással. Ez a SIMD változat az alap 32 bites regiszterkészletet használta 4x8 bites formátumban, és csak egész műveleteket támogatott.



3.20. ábra – ARMv7 NEON vektor regiszterek[29]

3.2.2.6 Security Extensions (TrustZone)

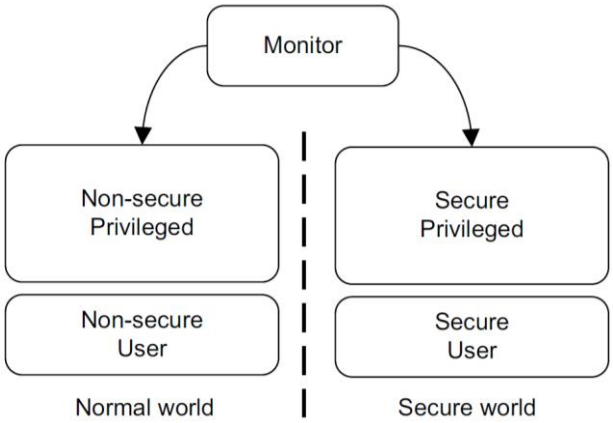
A TrustZone biztonsági kiegészítés, amelyet először az ARMv6KZ ISA változatban vezettek be, hardvertámogatott eszközöket nyújt biztonságos rendszerek kiépítésének elősegítésére. A biztonságos rendszer fogalma itt azt jelenti, hogy védelmet biztosít érzékeny adatok (pl. jelszavak, PIN kódok, bankkártyaadatok) tárolása során, megakadályozva az illetéktelen hozzáférést. A biztonságos működést azzal éri el, hogy minden hardver és szoftver erőforrást két „világ”, a biztonságos világ (Secure world) és a normál világ (Normal world) egyikéhez rendel hozzá, és hardveres korlátozásokkal támogatja az adatszivárgás megakadályozását a két világ között. A két világ közötti kapcsolatot az ún. Secure Monitor teremti meg (ld. 3.21. ábra).

Az architektúrakiegészítés lehetővé teszi, hogy egyetlen fizikai magban megvalósítható legyen a biztonságos és a normál kód futtatása is, időosztásos jelleggel. A világváltás ortogonális a processzor egyéb képességeire, így a két világ független virtuális működést biztosít két, eltérő

[29] Forrás: ARM Cortex-A Series Programmer’s Guide Version: 4.0

Copyright © 2021 / BME AUT

biztonsági kritériumú szoftverkörnyezet számára. A két világ között a memóriarendszer fel van osztva, és minden memóriaműveletnél (ideértve a cache tag-eket, laptáblákat, rendszermemória-hozzáféréseket és periféria-hozzáféréseket is) megjelenik egy járulékos NS (non-secure) bit. Ez felfogható egy plusz címbitnek is, tehát a teljes 32 bites címtartomány elérhető marad mindkét világ számára, de az egyes címek mögött fizikailag más-más tartalom lehet. A normál világ nem fér hozzá az NS bithez, minden normál világból induló memóriahozzáférésnél NS = 1, így a biztonságos világ memóriaterületei védettek. A biztonságos világ viszont hozzá tud férni a normál világ memóriatartalmához is.



3.21. ábra – A TrustZone működési elve[30]

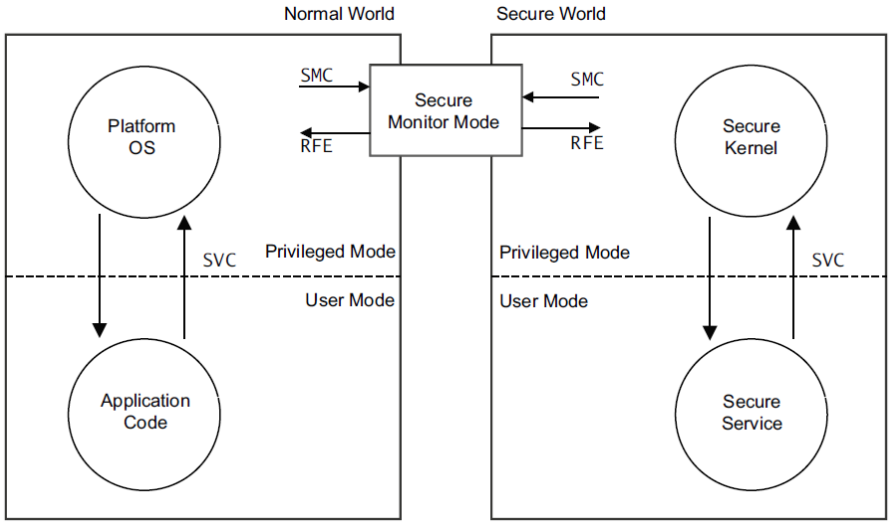
Az időosztásos működés velejárója, hogy a processzormagnak időről-időre váltania kell a normál és a biztonságos világban futó kódok között. Ennek érdekében a TrustZone egy új privilegizált üzemmódot vezet be az eddigiek (System, FIQ, IRQ stb.) mellé, ez a monitor mód. Ebben fut a Secure Monitor, amely a két világ közötti váltásért felelős. A monitor módra való áttérést az erre elkülönített, csak privilegizált üzemmódból hívható SMC (Secure Monitor Call) utasítással lehet kezdeményezni (ld. 3.22. ábra). Arra is van lehetőség, hogy bizonyos megszakítási vagy kivételi események közvetlenül a monitor módba irányítsák a végrehajtást.

A TrustZone-ra épülő biztonságos rendszerek szoftver megvalósítása általában úgy néz ki, hogy az operációs rendszer nagy része a normál világban fut, és rendelkezik egy kisméretű biztonságos kernellel, az ún. *Trusted Execution Environment*-tel (TEE), amely a biztonságos világban futó szolgáltatásokat (például a titkosítást, ujjlenyomat-olvasást, online fizetést stb.) kezeli. A normál világban futó operációs rendszer az SMC hívással tudja elérni ezeket a szolgáltatásokat az érzékeny adatokhoz való hozzáférés nélkül. Ha egy felhasználói szintű alkalmazás biztonsági szolgáltatást szeretne igénybe venni (például fizetési tranzakciót kezdeményezni), akkor általában egy magasszintű szolgáltatásspecifikus API-t vesz igénybe (pl. egy *reqPayment()* függvény meghívásával), amelyet egy normál világban létező függvénykönyvtár biztosít. A könyvtár ugyanattól a szerzőtől származik, amelytől maga a biztonsági szolgáltatás is (pl. egy bankkártyaszolgáltatótól). A folyamatot a 3.23. ábra szemlélteti. A könyvtár egy rendszerhíváson keresztül értesíti az operációs rendszert, amely a TrustZone

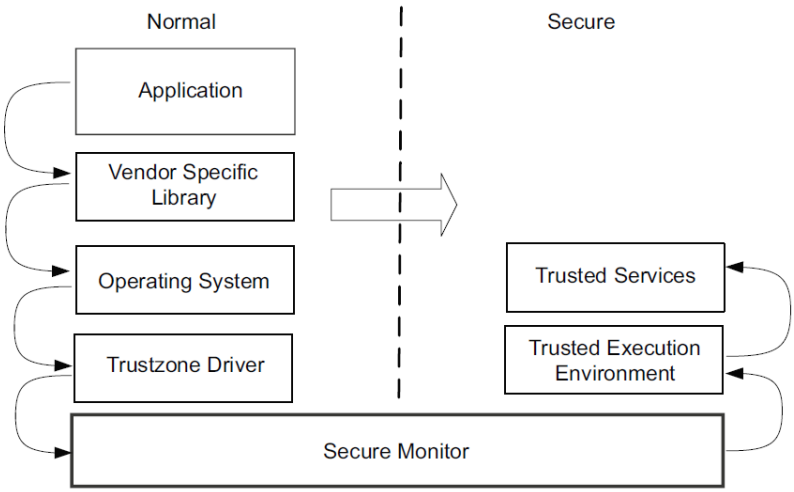
[30] Forrás: ARM Cortex-A Series Programmer's Guide Version: 4.0

Copyright © 2021 / BME AUT

meghajtóprogramnak továbbítja a kérést, ez pedig a Secure Monitor meghívásával átadja a végrehajtást a biztonságos futtatókörnyezetnek.



3.22. ábra – Üzemmódok és világok közötti váltás lehetőségei: SVC (Supervisor Call) utasítás, SMC (Secure Monitor Call) utasítás, RFE (Return From Exception) utasítás[31]



3.23. ábra – Biztonságos szolgáltatás hívásának menete[31]

3.2.2.7 Virtualization Extensions

Az ARMv7-A ISA változatban jelent meg először kiegészítő funkcióként a rendszerszintű virtualizáció hardveres támogatása. Rendszerszintű virtualizációról akkor beszélünk, ha ugyanazon a hardveren több, egymástól független operációs rendszer létezhet egymás mellett,

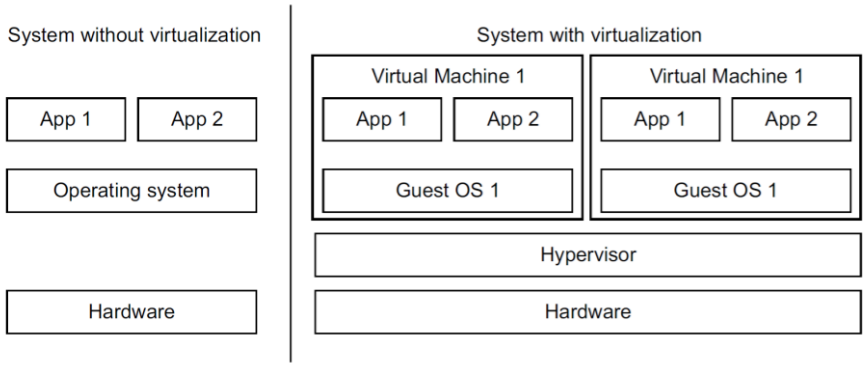
[31] Forrás: ARM Cortex-A Series Programmer's Guide Version: 4.0

Copyright © 2021 / BME AUT

amelyek nem is értesülnek egymásról. Az operációs rendszerek ilyenkor nem a fizikai, hanem egy-egy virtuális hardvert (VM-et) látnak.

A virtualizációt felügyelő speciális szoftvert *hypervisor*-nak vagy *Virtual Machine Manager*-nek (VMM) nevezzük, és kétféle megvalósítása ismeretes:

- 1-es típusú, vagy „bare metal” virtualizáció esetén a hypervisor a „vendég” (guest) operációs rendszereknél magasabb szinten, közvetlenül a hardveren fut (ld. 3.24. ábra).
- 2-es típusú, vagy „hosted” virtualizációnál a hypervisor maga is egy „gazda” (host) operációs rendszer alatt fut, és úgy kezel további virtuális gépeket.



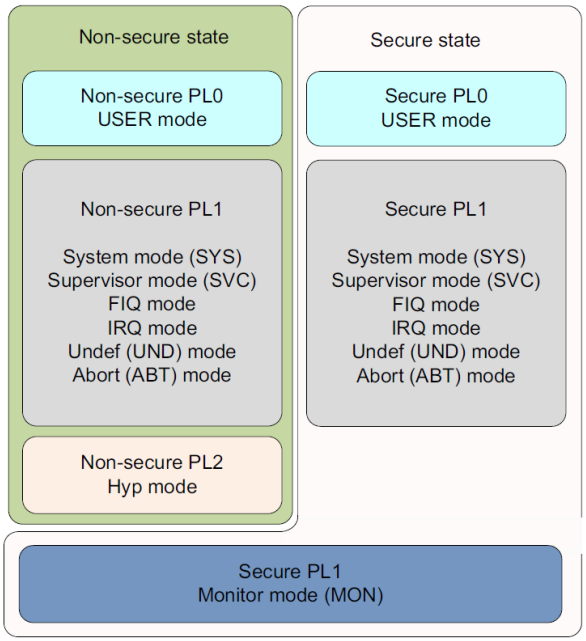
3.24. ábra – Az 1-es típusú rendszerszintű virtualizáció elve[32]

Az ARM virtualizációs kiegészítés az 1-es típusú hypervisor megvalósítását támogatja hardveres eszközökkel. Ehhez bevezet egy új, privilegizált HYP üzemmódot a korábbiak mellé (ld. 3.25. ábra). Ráadásul ez az üzemmód egy új, magasabb privilégiumszintet (PL2) valósít meg. Az eddig ismert üzemmódok közül a User mód az egyetlen privilegizálatlan, vagyis PL0 (Privilege Level 0) szintű üzemmód, a többi korábban tárgyalt mód (System, FIQ, IRQ, Supervisor, Undef, Abort, Monitor) a PL1 szinten helyezkedik el, és az új HYP mód mindezek fölött áll a PL2 szinten. Ez érthető, hiszen az operációs rendszerek a PL1 privilégiumszint üzemmódjait használják, ezért a fölöttük álló hypervisornak szüksége van olyan erőforrásokra és lehetőségekre, amelyek a guest operációs rendszerek elől rejtve maradnak. A HYP mód például saját segédregiszterekkel rendelkezik, valamint saját megszakítási vektortáblával is. Így például a megszakítások hardveresen a hypervisor-ba irányíthatók, az pedig eldöntheti, hogy melyik vendég operációs rendszernek továbbítja az egyes kéréseket.

A virtualizáció a TrustZone kiegészítéssel együtt is alkalmazható. Ez esetben a virtualizáció csak a normál világban érhető el, a hypervisor nem fér hozzá a biztonságos világ védett erőforrásaihoz annak ellenére sem, hogy a Secure Monitor az eggyel alacsonyabb PL1 privilégiumszinten fut.

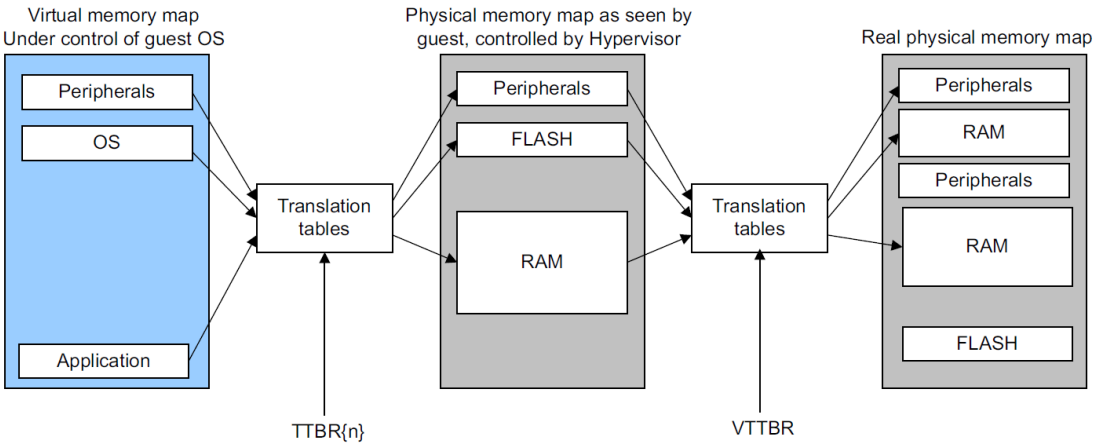
[32] Forrás: ARM Cortex-A Series Programmer’s Guide Version: 4.0

Copyright © 2021 / BME AUT



3.25. ábra – ARMv7 privilégiumszintek és üzemmódok virtualizáció és TrustZone használata esetén[33]

A rendszerszintű virtualizáció a virtuális memóriakezelés kibővítését is igényli, hiszen több operációs rendszer működik egymás mellett, amelyek közül mindegyik azt gondolja, hogy egyedül használja a hardvert, és ennek megfelelően végzi a virtuális–fizikai címtranszformációt. Azért, hogy ne zavarhassák egymást, a fölöttük álló hypervisor egy második szintű transzformációt is végez ezeken a köztes „fizikai” címeken, ehhez pedig saját második szintű laptáblákkal rendelkezik. A kétszintű címtranszformációt a 3.26. ábra szemlélteti.



3.26. ábra – Kétszintű címtranszformáció hypervisor használata esetén[32]

[33] Forrás: ARM Cortex-A Series Programmer’s Guide for ARMv8-A

Copyright © 2021 / BME AUT

3.2.3 Az ARMv8-A ISA sajátosságai

Az ARM architektúrák történetében a 2012-ben bevezetett ARMv8-A verzió az, ami a legszéleskörűbb változásokat hozta a kezdetek óta. Az azóta közzétett alverzióival együtt ez a jelenleg elérhető legfrissebb ARM ISA változat. Az ARMv8-A architektúrában került sor a 64 bites átállásra, és ezzel együtt gyakorlatilag a teljes addigi architektúra újragondolására. A bevezetett újítások nem csak plusz funkciókkal való bővítéseket jelentettek, hanem talán még fontosabb, hogy mertek szakítani olyan korábbi tulajdonságokkal is, amelyek az idők során nem bizonyultak kellően hatékonynak, vagy a mai mikroarchitekturális és technológiai lehetőségek mellett nem indokolt (esetleg egyenesen hátrányos) a használatuk.

Mindezzel együtt megmaradt a teljes visszafelé irányuló kompatibilitás az ARMv7-A architektúrával. Ezt két megkülönböztetett végrehajtási állapot (execution state) bevezetésével érték el. Az ún. *AArch64* állapotban érhető el a 64 bites utasításkészlet az összes újítással együtt, az ún. *AArch32* állapotban pedig a korábbi 32 bites architektúrával kompatibilis funkcionalitást találjuk.

Az ARMv8-A architektúrán belül három utasításkészletet különböztetünk meg. Az *A64* utasításkészlet jelenti az új 64 bites változatot, amely 64 bites adatok közvetlen kezelését teszi lehetővé, 64 bites regisztereket használ, és 64 bites virtuális címtartományt kezel. Az *A32* felel meg a korábbi „ARM”, a *T32* pedig a korábbi „Thumb” utasításkészleteknek. Ezeknek a funkcionalitása visszafelé kompatibilis az ARMv7-A-val, de tartalmaz újításokat is. Ez a gyakorlatban azt jelenti, hogy egy ARMv7-A-ra fejlesztett kód futtatható az ARMv8-A AArch32 állapotában, de egy ARMv8-A A32+T32 kód nem feltétlenül lesz futtatható ARMv7-A alapú processzoron.

Az A64 utasításkészlet az AArch64 végrehajtási állapotban használható, és nincs mellette tömörített „T64” változat. Az AArch32 állapotban pedig az A32 és T32 utasításkészletek használhatók. Az AArch64 teljesen független az AArch32-ben elérhető A32 és T32 utasításkészletek tulajdonságaitól, és egy külön állapotváltási mechanizmussal lehet áttérni az AArch64 és az AArch32 között.

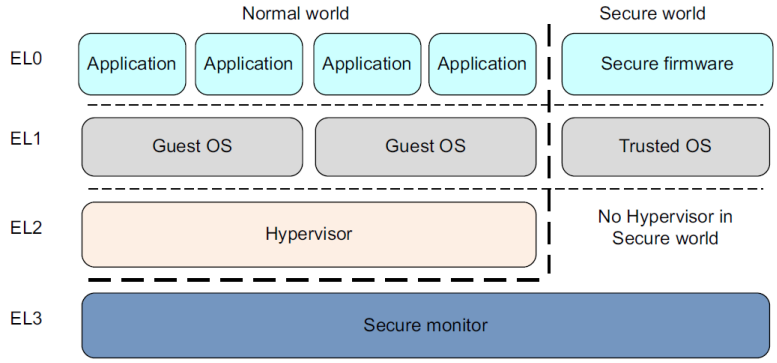
Jelentős újdonság az AArch64-ben, hogy újraszervezték a végrehajtási üzemmódokat és privilégiumszinteket. A korábbi 9 üzemmód és 3 privilégiumszint helyett bevezettek 4 ún. kivételi szintet (Exception Level), ezeken belül viszont már nem különböztetnek meg további üzemmódokat.

Az előző alfejezetben tárgyalt ISA kiegészítések körében is több jelentős változás történt. Például a biztonsági kiegészítés (TrustZone) és a virtualizáció az ARMv7-A-ban opcionális lehetőség volt, ehhez képest az ARMv8-A-ban már mindkettő alapszolgáltatásnak számít, vagyis az erre épülő processzorokban már kötelező az implementációjuk. Ugyanígy a korábbi VFP és Advanced SIMD (NEON) szolgáltatások egymást kiegészítő funkcionalitását összevonták, és egységes kezeléstechnikával az ARMv8-A szerves részévé tették. A Jazelle és ThumbEE szolgáltatások már nem támogatottak az ARMv8-ban (az AArch32 végrehajtási állapotban sem).

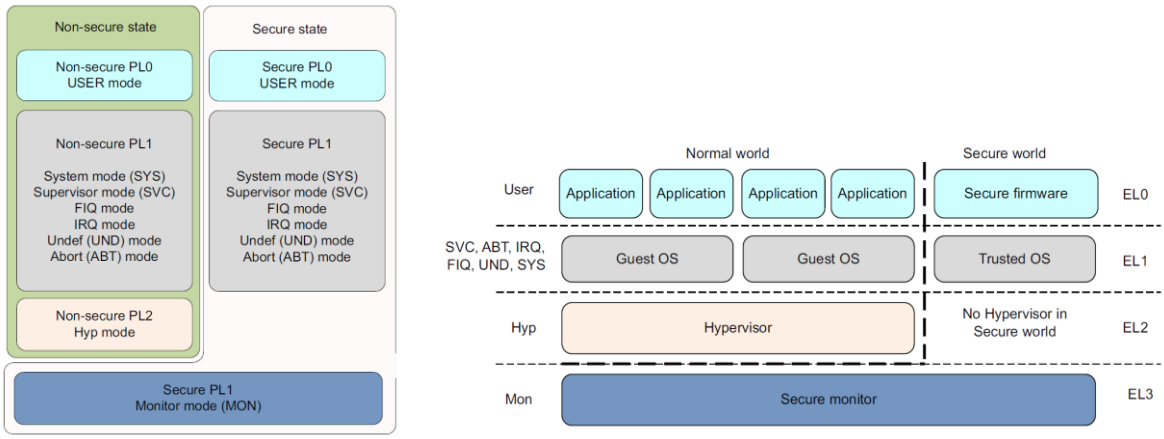
Copyright © 2021 / BME AUT

3.2.3.1 Privilégiumszintek és üzemmódok újragondolása

Ahogy korábban láttuk, az ARMv7-A 1 db privilegizálatlan és 6+2 db privilegizált üzemmódot definiált, amelyek három különböző privilégiumszinten helyezkedtek el. Egyedül a hypervisor (HYP mód) futott a legmagasabb PL2-es szinten, és a biztonságos és normál világok közötti váltásokat felügyelő Secure Monitor (TrustZone) is az eggyel alacsonyabb PL1-es szintet használta, ahogy a többi privilegizált üzemmód is. Ehhez képest az ARMv8-A 4 db Exception Level-t (EL0–EL3) használ, külön üzemmód megkülönböztetés nélkül (ld. 3.27. ábra). Az EL0 képviseli a privilegizálatlan felhasználói szintet, az EL1 szinten futnak az operációs rendszer(ek) szolgáltatásai, az EL2 szinten pedig a hypervisor. A Secure Monitor a legprivilegizáltabb, EL3-as szintre került. A korábbiakhoz hasonlóan a virtualizáció csak a normál világban elérhető, tehát a biztonságos világban nem találjuk meg az EL2 kivételi szint megfelelőjét.



3.27. ábra – Kivételi szintek az ARMv8 architektúrában[34]



ARMv7A ARMv8-A AArch32

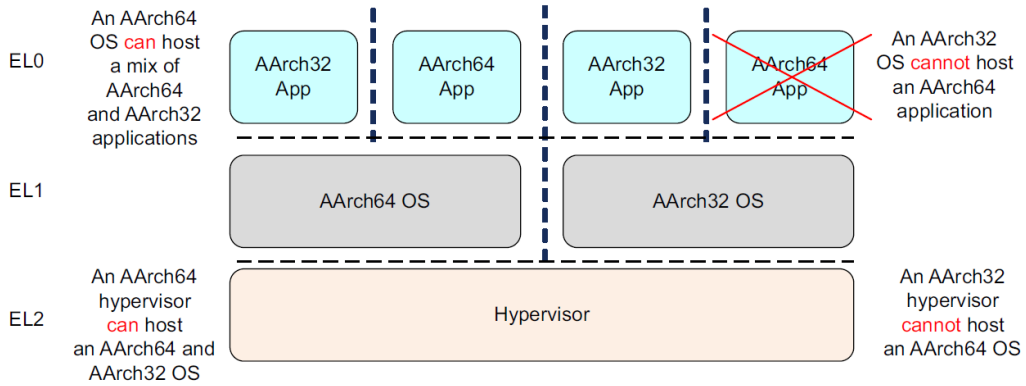
3.28. ábra – Az ARMv7-A és az ARMv8-A AArch32 üzemmódok és privilégiumszintek összehasonlítása[34]

[34] Forrás: ARM Cortex-A Series Programmer's Guide for ARMv8-A

Copyright © 2021 / BME AUT

Ugyanez a felosztás érvényes az AArch64 és az AArch32 végrehajtási állapotban is. Az AArch32 a kompatibilitás miatt megtartotta a korábbi üzemmódokat, a 3.28. ábrán látható ezek hozzárendelése a kivételi szintekhez.

Vannak olyan esetek, amikor váltani kell a rendszer végrehajtási állapotai között. Például, ha egy 64 bites operációs rendszer egy 32 bites alkalmazást szeretne futtatni, akkor az AArch64 állapotból át kell térni az AArch32-be. Amikor a 32 bites alkalmazás véget ért, vagy más miatt tér vissza az operációs rendszerhez, akkor újra az AArch64 állapotban folytatódik a végrehajtás. Fordítva ez nem lehetséges: egy AArch32-es operációs rendszer nem futtathat 64 bites alkalmazást. Ugyanígy egy AArch64-es hypervisor futtathat AArch32-es operációs rendszert, de fordítva ez már nem tehető meg. A 3.29. ábra mutatja mindezt, és az is látható rajta, hogy a végrehajtási állapotok közötti váltás mindenképpen kivételi szintek közötti áttérést is jelent. Az előzőekből következik, hogy az AArch32-ből AArch64-be való áttérés csak magasabb kivételi szintre lépés mellett lehetséges, illetve AArch64-ből AArch32-be csak a kivételi szint csökkenése mellett lehet átlépni. Azonos kivételi szinten belül nem lehetséges váltani a két végrehajtási mód között. Ha például egy 32 bites és egy 64 bites alkalmazás között szeretnénk váltani (mindkettő az EL0 szinten fut), akkor pl. egy rendszerhívással előbb vissza kell térni az operációs rendszerhez (EL1 szint), majd onnan meghívni a másik alkalmazást. Ez a gyakorlatban azt jelenti, hogy nem lehet vegyes 32/64 bites alkalmazásokat létrehozni, mert nincs közvetlen hívási mechanizmus a két végrehajtási állapot között.



3.29. ábra – Végrehajtási állapotok közötti kompatibilitás[35]

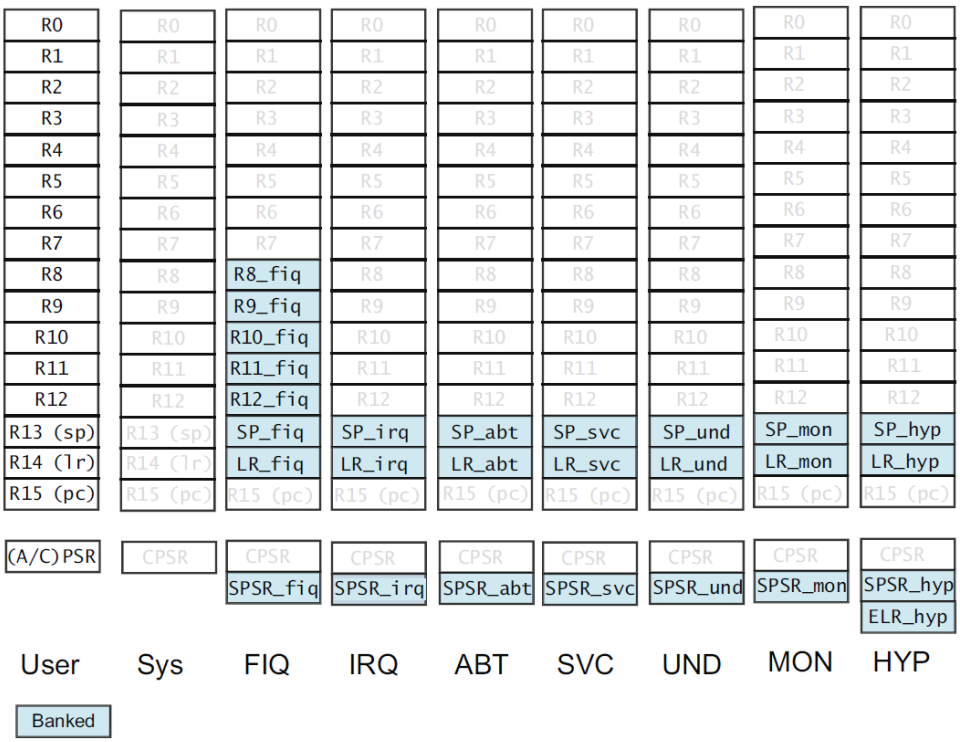
3.2.3.2 Regiszterkészlet

Ahogy korábban láttuk, az ARMv7-A ISA verzióban a regiszterkészlet üzemmódok szerint bankokba van szervezve (jelentős átfedésekkel), és a TrustZone és virtualizációs kiegészítésekkel együtt összesen 35 db általános célú, 32 bites regisztert tartalmaz, ebbe beleértve a regisztercímzéssel elérhető PC-t, valamint a stack pointer és link regiszter bankolt változatait (ld. 3.30. ábra). Ezeken kívül rendelkezésre áll 9 db speciális státuszregiszter is. Ez az elrendezés egyfelől rugalmas működést és gyors kivételkezelést tesz lehetővé, ugyanakkor

[35] Forrás: ARM Cortex-A Series Programmer's Guide for ARMv8-A

Copyright © 2021 / BME AUT

vegyük észre, hogy legfeljebb 17–19 regiszter használható egyszerre, ami az erőforrásoknak nem a legjobb kihasználását jelenti.



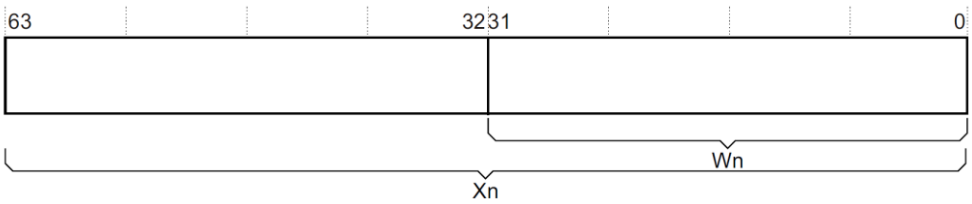
3.30. ábra – ARMv7-A regiszterkészlet[36]

Az ARMv8-A jelentős újításokat eszközöl a regiszterek vonatkozásában is. Az AArch64 végrehajtási állapot 31 db 64 bites általános célú regisztert tartalmaz (X0–X30 néven), amelyek minden kivételi szinten egységesen elérhetők (nincs bankolás). Minden 64 bites regiszter 32 bites változatban is hozzáférhető W0–W30 néven, amelyek az X0–X30 regiszterek alsó 32 bitjének felelnek meg (ld. 3.31. ábra). Az általános célú regiszterek között megtaláljuk a Procedure Link Register-t (X30/W30), amely az EL0 szinten futó alkalmazások link regisztere. A stack pointerek és a programszámláló az új architektúrában nem érhetők el általános regiszterként.

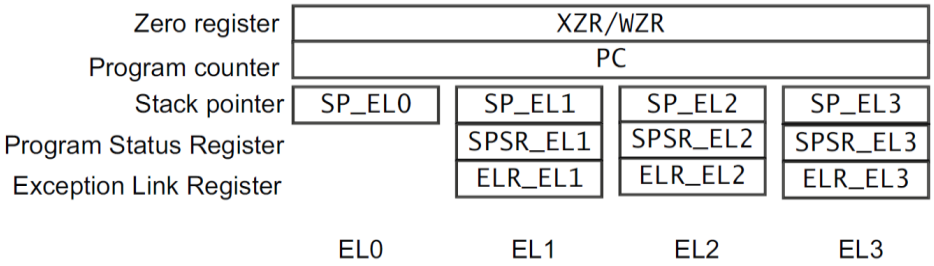
Első ránézésre furcsának tűnhet, hogy az általános regiszterkészlet csupán 31 db regiszterből áll (tapasztalataink alapján 2 hatványát, 32 db-ot várnánk), az X31/W31 regiszter hiányzik. Valójában ezt a nevet is használhatjuk regiszterreferens utasításokban, az X31/W31 a legtöbb utasításnál az ún. zéró regisztert jelenti (hogy ez mit takar, arról hamarosan szót ejtünk), bizonyos esetekben pedig a stack pointert.

[36] Forrás: ARM Cortex-A Series Programmer's Guide for ARMv8-A

Copyright © 2021 / BME AUT



3.31. ábra – Az ARMv8-A általános regiszterek 64 bites (Xn) és 32 bites (Wn) verziói[37]



3.32. ábra – Az AArch64 speciális regiszterei[37]

Az általános regiszterkészlet mellett speciális regisztereket is találunk az új 64 bites architektúrában, ezek némelyike újdonság, másoknak volt hasonló vagy azonos funkcionalitású megfelelője a korábbi architektúraverziókban is. A speciális regisztereket a 3.32. ábrán láthatjuk, az alábbiakban pedig részletezzük az architektúrában betöltött szerepüket.

Zéró regiszter (XZR/WZR)

A zéró regiszter olvasás esetén mindig nulla értéket ad vissza, írásnál pedig egyszerűen eldobja a beleírt adatot. Ez az első ránézésre furcsának tűnő viselkedés sok esetben igen jól használható. Például, ha 64 bites nulla értékre van szükségünk, akkor úgy juthatunk a legegyszerűbben hozzá, ha forrásoperandusnak az XZR/WZR néven elérhető zéró regisztert adjuk meg (sok utasításnál az X31/W31 is ugyanezt jelenti). Egyébként a nulla értéket általában csak plusz egy regisztertöltéssel vagy XOR művelettel tudnánk előállítani. Abban az esetben, ha nincs szükség egy művelet eredményére, csak például a státusz flag-ek állítására, akkor céloperandusként a zéró regisztert megadva nem kell más (esetleg értékes adatot tartalmazó) regisztert „elrontanunk”.

A zéró regiszternek más, első látásra meglepő, „trükkös” alkalmazásai is vannak. Segítségével az architektúra tervezői többek között csökkenteni tudták az utasításkódok számát. Például a *MUL Wx, Wy, Wz* alakú szorzó utasításnak nincsen külön opkódja, hanem megegyezik a szorzó-összeadó *MADD Wx, Wy, Wz, WZR* utasítás kódolásával (persze mindkettő használható a programozás során, az assembler elintézi a cserét). Még meglepőbb lehet, hogy például a MOV utasítás is csak a logikai VAGY (ORR) utasítás álneveként szerepel az utasításkészletben. Ha a programban *MOV Wx, Wy* szerepel, azt valójában az *ORR Wx, WZR, Wy* utasításkódjával helyettesíti az assembler.

[37] Forrás: ARM Cortex-A Series Programmer’s Guide for ARMv8-A

Copyright © 2021 / BME AUT

Programszámláló (PC)

Az ARMv7 és korábbi architektúrák egyik sajátossága az volt, hogy a PC általános célú regiszterként címezhető volt *R15* néven. Ez elmés programozási megoldásokra adott lehetőséget, de megnehezítette a fordítóprogramok dolgát és a komplex csővezetékek kialakítását. Az ARMv8-ban a PC közvetlen elérésének megszüntetése egyszerűsíti a return predikciókat, és hatékonyabb pipeline megvalósítást tesz lehetővé. Az AArch64 architektúrában a PC egyetlen utasítás operandusaként sem jelenik meg explicit módon. Vannak olyan utasítások, amelyek implicit módon felhasználják az értékét (például PC-relatív betöltés vagy címgenerálás), de például nem adható meg adatfeldolgozó vagy load utasítások céljaként.

Stack pointer (SP_ELn)

Az ARMv7-A architektúrában minden üzemmódnak saját példánya volt a stack pointerből, amely az R13 (bankolt) regiszterként volt elérhető. Az ARMv8-A AArch64-ben az SP elkülönül az általános regiszterektől, de az továbbra is igaz, hogy minden kivételi szintnek saját példánya van belőle (SP_ELn). Ugyanakkor a stack pointer megválasztásában a privilegizált kivételi szinteknek bizonyos fokig szabadsága van. Az EL0-nál magasabb szinteken a processzor az adott kivételi szint saját SP példányát és az EL0-hoz tartozó stack pointert is használhatja. Ezenkívül az SP ugyan már nem általános regiszter, de néhány aritmetikai utasításban (pl. ADD) szolgálhat forrás- és céloperandusként is, ami például egy függvény meghívása után a stack frame létrehozásához elengedhetetlen.

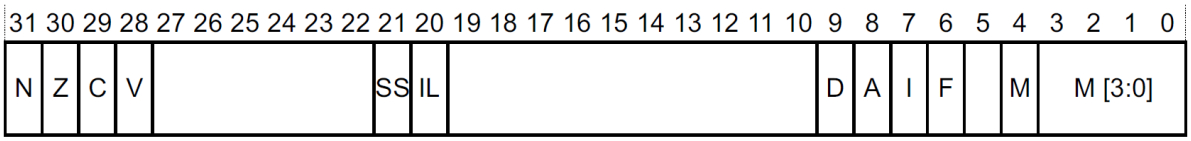
Exception Link Register (ELR_ELn) és Saved Program Status Register (SPSR_ELn)

A kivételkezelési eljárások számára mindhárom privilegizált kivételi szint saját Exception Link Register-rel és mentett státuszregiszterrel rendelkezik, hasonlóan a korábbi architektúrákban látottakhoz. Ezek tartalmazzák a kivétel vagy megszakítás előtti utolsó utasítás címét és az akkori processzorállapotot, amelyek szükségesek a kivétel okának meghatározásához és a későbbi visszatéréshez. Az ELR esetén az újdonság az, hogy nem érhető el általános regiszterként (az ARMv7-ig az R14 volt a link regiszter a privilegizált üzemmódokban is). Érdekesség, hogy a privilegizálatlan (EL0 szintű) alkalmazások számára az ún. Procedure Link Register az AArch64-ben továbbra is a normál regiszterkészlet része (X30/W30).

A mentett státuszregiszterek (Saved Program Status Register, SPSR) felépítése szinte teljesen megegyezik a korábbi architektúráknál látottal (ld. 3.33. ábra). Megtalálhatók benne például az N (negative), Z (zero), C (carry) és V (overflow) flag-ek, az IRQ és FIQ engedélyező bitek (I, F) és az üzemmód jelző M[4:0] bitek. Ez utóbbiaknál érdekesség, hogy az M[4] bitnek kitüntetett szerepe van: ez jelzi a 64 vagy 32 bites végrehajtási állapotot (0 – AArch64; 1 – AArch32). Ha visszaemlékszünk, a „klasszikus” ARM architektúránál megemlítettük, hogy a 7 (később 9) üzemmódot már ott is 5 biten kódolták a státuszregiszterben. A „tartalékolás” haszna itt látszik: korábban minden üzemmód esetén az M[4] bit értéke fixen 1 volt, hiszen az üzemmódok számának megfelelően elég volt az alsó 4 bitet felhasználni az aktuális üzemmód azonosítására. Az ARMv8-A architektúrában az M[4] = 1 eset az M[3:0] bitekkel együtt ugyanazokat a 32 bites üzemmódokat jelenti, mint a korábbi változatokban. M[4] = 0

Copyright © 2021 / BME AUT

választással viszont az alsó M[3:0] bitek teljesen szabaddá váltak az AArch64 kivételi szintjeinek azonosítására, hiszen a 0-val kezdődő kódokat nem használták az ARMv8-A előtt.



3.33. ábra – AArch64 Saved Program Status Register[38]

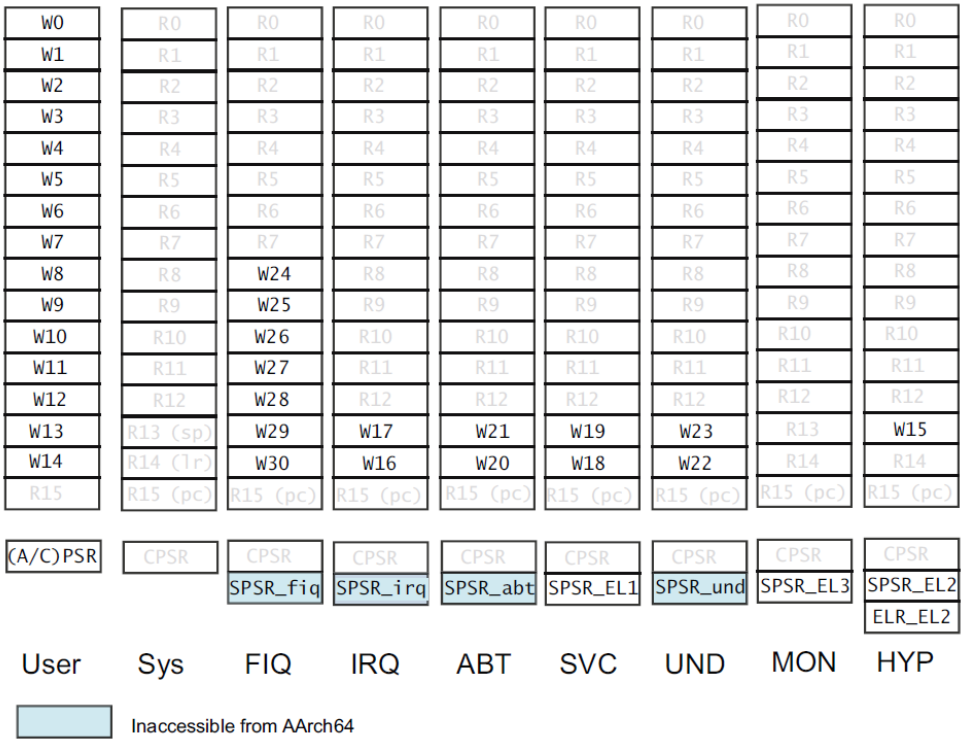
Az SPSR regiszterek hasonlósága ellenére az ARMv8-A AArch64 architektúrában az aktuális processzorállapot tárolása és elérése jelentősen különbözik a korábban megszokottól. Nincs ugyanis egyben elérhető Current Program Status Register (CPSR). Ehelyett létezik egy PSTATE (Processor State), amely viszont regiszterként egyben nem, csak külön-külön mezőnként érhető el az egyes utasítások számára. Hasonlóan az ARMv7-A és korábbi architektúráknál látott, User módban elérhető csökkentett tartalmú APSR-hez, az EL0 szinten csak a PSTATE.{N, Z, C, V} mezők (feltétel flag-ek) érhetők el.

Az AArch64 és AArch32 regiszterkészlet megfeleltetése

Az AArch64 és AArch32 végrehajtási állapotok közötti váltásról már szót ejtettünk a 3.2.3.1. fejezetben a kivételi szintek szempontjából. Az állapotváltásnak van néhány sajátossága az eltérő regiszterhasználat miatt is. Mivel az AArch32 állapot kompatibilis az ARMv7-A architektúrával, ezért a regiszterkiosztása is teljesen megfeleltethető a 3.30. ábrán látottaknak. Ugyanakkor nincs külön AArch32 és AArch64 regiszterkészlet, hanem a processzor mindkét állapotban ugyanazokat a fizikai regisztereket használja. Természetesen a 32 bites üzemmódban a 64 bites regisztereknek csak az alsó 32 bitje használható. Az ARMv8-A architektúra 32 bites W0–W30 regiszterei és néhány speciális regisztere a 3.34. ábrán látható módon képezhetők le meg az AArch32 üzemmód bankolt regisztereire. Az ábrán az is látszik, hogy nem teljes a megfeleltetés, mert vannak olyan AArch32 regiszterek is (néhány SPSR példány), amelyeknek nincs AArch64-es megfelelője, ezért nem is érhetők el 64 bites üzemmódban.

[38] Forrás: ARM Cortex-A Series Programmer’s Guide for ARMv8-A

Copyright © 2021 / BME AUT



3.34. ábra – AArch64 és AArch32 regiszterek megfeleltetése[39]

3.2.3.3 Az A64 utasításkészlet

Az A64 utasításkészlet számos újítást tartalmaz a 64 bites bővítésen (adathossz, címtartomány) túl is. Az utasítások hossza továbbra is egységesen 32 bites, de felépítésük és kódolásuk eltér az A32 utasításkészletétől.

Az új utasításkészletnél kiemelten figyeltek az ortogonalitásra. Ez igaz volt a „klasszikus” ARM utasításkészletre is, azonban a verziók fejlődése során a bővítésekkel és kiegészítésekkel az ortogonalitás itt-ott „megkopott”. Például a load/store utasítások (LDR, STR) később bevezetett 16 bites adatokat is támogató (half-word, félszavas) változatának kódolása eltért a korábbi változatoktól, ami kissé eltérő címzési módokat eredményezett. Vagy például a lebegőpontos és NEON regiszterek is más címzési módokkal voltak elérhetők, mint az általános regiszterek. Az A64 utasításkészletben az ilyen jellegű eltéréseket megszüntették, ami a különböző szóhosszúságú egész és lebegőpontos típusok kezelését egységesebbé, átláthatóbbá teszi.

Az A64 utasításkészletben talán az új funkcióknál is fontosabb az, hogy a hatékonyság érdekében szakít olyan korábbi, nagyra értékelt megoldásokkal, mint az aritmetikai utasításokba épített barrel shifter, vagy a mindenre kiterjedő feltételes végrehajtás lehetősége.

[39] Forrás: ARM Cortex-A Series Programmer’s Guide for ARMv8-A

Copyright © 2021 / BME AUT

Barrel shifter

Az A64 utasításkészletben nincs meg az a flexibilis eltolási lehetőség, ami az ARMv7-ig közvetlenül elérhető volt az adatfeldolgozó (pl. aritmetikai) utasításokban a második forrásoperandusnál. Mivel a magasszintű programnyelvekben ritkán fordulnak elő ehhez hasonló műveletek, ezért a fordítók csak kis mértékben tudták ezt a funkciót kihasználni. Ugyanakkor az opkódban számos bitet elfoglalt az eltolás fajtájának és mértékének megadása. A 32 bites utasításszavakban minden egyes bitnyi hely nagyon értékes, gondoljunk csak a közvetlen konstansok trükkös megadására, amit az erre szánható 12 bit leghatékonyabb kihasználása érdekében vezettek be. Ráadásul a barrel shifter megfelelő időzítésű megvalósítása egyre nehezebb az egyre növekvő órajelfrekvenciák mellett a komplikált huzalozás miatt. A minden utasításban megtalálható implicit barrel shifter funkció megnyújtja a csővezetékfázisok minimális idejét, és korlátozza a maximális órajelfrekvenciát. Mindezekre való tekintettel az A64 utasításkészletből kihagyták ezt a ritkán használt lehetőséget, és a különféle eltolások és forgatások csak külön utasítások formájában érhetők el.

Feltételes végrehajtás

Egy másik figyelemre méltó változás az A64 utasításkészletben a feltételes végrehajtás körének jelentős leszűkítése. Ugyanis ez a lehetőség sem bizonyult annyira hasznosnak, mint amennyi bitet az összes utasítás opkódjában igényelt. Ráadásul a modern szuperskalár processzorokban (pl. Cortex-A9) a feltételhez kötött utasítások már a feltételes elágazásoknál nagyobb mértékben ronthatják a csővezeték hatékonyságát az utasítások közötti egymásra hatások miatt. Ezekben a processzorokban már természetes az elágazásprediktorok jelenléte, amelyek nagymértékben lecsökkentik az ugró utasítások költségét.

Az A64-ben háromféle utasítástípus van, amely feltételhez köthető: elágazások, bizonyos összehasonlítások (conditional compare) és az ún. feltételes kiválasztás (conditional select). Ez utóbbi más architektúrákban is megtalálható a predikatív végrehajtás megvalósítására (pl. ilyen az x86 architektúrában a CMOVE utasítás).

A feltételes végrehajtás elhagyására, illetve alternatív megvalósítására már az ARM architektúrákon belül is láttunk példát. A Thumb utasításkészlet például egyáltalán nem támogatta ezt a szolgáltatást, hiszen a 16 bites utasításhosszba egyszerűen nem fértek bele a feltételbitek. A Thumb-2 az IT utasítás bevezetésével visszahozta ezt a funkciót, de már ott is látszott, hogy a korábbi, teljes megvalósítás hatékonysága gyakorlatilag elérhető úgy is, ha erre a célra elkülönített utasításokat használunk.

A **Conditional Select** (CSEL) utasítás két regiszter tartalmának egyikét másolja át a kimeneti regiszterbe. Ez a C nyelvből ismert hármas feltételes operátor (?:) funkcionalitásának felel meg:

```
C = feltétel ? A : B
```

Az if-then-else típusú rövid feltételes szekvenciák úgy valósíthatók meg vele elágazás nélkül, hogy mindkét ágat végrehajtjuk, és a végén választjuk ki a feltétel alapján, hogy melyik ág

Copyright © 2021 / BME AUT

eredménye kerüljön elmentésre. Vannak olyan változatai is, amelyek a második forrásregiszter módosított értékével operálnak:

```
CSINC:    C = feltétel ? A : (B+1)
CSINV:    C = feltétel ? A : ~B
CSNEG:    C = feltétel ? A : –B
```

A CSINC a második forrásregiszter eggyel növelt értékét, a CSINV a bitenkénti inverzét, a CSNEG pedig az ellentettjét adja vissza, ha a feltétel nem teljesül.

Lássunk két egyszerű példát ezek használatára!

---

1. Adott a következő egyszerű C programrészlet:

```
if (i == 0) r = r + 2;
else r = r – 1;
```

Megvalósítás A64 assembly-ben a CSEL alkalmazásával:

```
CMP  Wi, #0              ; if (i == 0)
SUB  Ws, Wr, #1          ; s = r - 1
ADD  Wr, Wr, #2          ; r = r + 2
CSEL Wr, Wr, Ws, EQ      ; r = EQ ? r : s
```

2. Abszolútérték-képzés CSNEG használatával:

X0 = abs(X1)

```
CMP   X1, #0            ; Compare X1 to zero (set condition flags)
CSNEG X0, X1, X1, GE    ; X0 = (X1 >= 0) ? X1 : -X1
```

■

A korábbi, minden utasításra kiterjedő feltételes végrehajtáshoz hasonlóan ez az új megközelítés is hatékony és tömör kódot eredményez. Ezt az állítást a korábban már bemutatott Euklideszi algoritmus példájával illusztráljuk.

Copyright © 2021 / BME AUT

Az Euklideszi algoritmus C nyelvű implementációja:

```
int gcd(int x, int y) {
   while (x != y)          /* while x and y are not equal */
      if (x > y) x -= y;   /* substract the smaller from
      else y -= x;            the greater                 */
   return x;               /* if x and y are equal the
}                             solution has been found     */
```

A32 assembly:

```
gcd: CMP    Rx, Ry          ; set condition "NE" if (x != y),
                            ;               "GT" if (x > y),
                            ;               "LE" if (x =< y)
     SUBGT  Rx, Rx, Ry      ; if "GT" (Greater Than), x = x-y
     SUBLE  Ry, Ry, Rx      ; if "LE" (Less Than Or Equal), y = y-x
     BNE    gcd             ; if "NE" (Not Equal), then loop
```

A64 assembly:

```
gcd: SUBS  Wt, Wx, Wy       ; t = x-y, and set condition
                            ;          ("GT" if x-y > 0)
     CSEL  Wx, Wt, Wx, GT   ; if "GT" then x = t, else x = x
     CSNEG Wy, Wy, Wt, GT   ; if "GT" then y = y, else y = -t
     B.NE  gcd              ; if "NE" then loop
```

■

A **Conditional Set** (CSET, CSETM) utasítások a feltételes kiválasztás egy speciális formájának tekinthetők. A feltétel teljesülésétől függően a célregiszter értékét 1-re vagy 0-ra (CSET), illetve –1-re vagy 0-ra (CSETM) állítják:

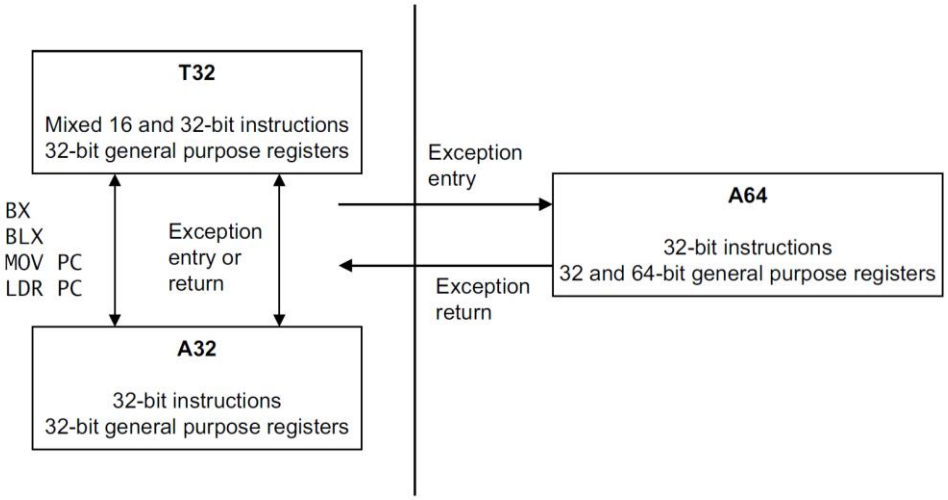
```
CSET  W0, EQ    ; If equal (Z=1) then W0 = 1, else W0 = 0
CSETM X0, NE    ; If not equal then X0 = -1, else X0 = 0
```

A **Conditional Compare** (CCMP, CCMN) utasítások feltételhez kötött összehasonlítást végeznek. Az egyszerű összehasonlításhoz hasonlóan céloperandusuk nincsen, csak a feltétel flag-eket állítják. Jól használhatók egymásba ágyazott vagy összetett összehasonlítások megvalósításához.

Copyright © 2021 / BME AUT

Váltás az utasításkészletek között

Ahogy korábban is említettük, nincs arra lehetőség, hogy egy alkalmazáson belül váltsunk az AArch64 és AArch32 végrehajtási állapotok között. Ebből következik, hogy nincs közvetlen áttérési lehetőség az A64 és A32 vagy T32 utasításkészletek használata között sem, ahogyan ez megvan az A32 és T32 között (BX, BLX utasítások, vagy a PC-t közvetlenül felülíró MOV vagy LDR műveletek). A 32 bites utasításkészletek és az A64 közötti váltás csak kivételi szint váltással együtt képzelhető el, ahogyan az AArch32 és AArch64 állapotok közötti áttérésnél láttuk.



3.35. ábra – Váltás az egyes utasításkészlet-változatok között[40]

Lebegőpontos műveletek és NEON

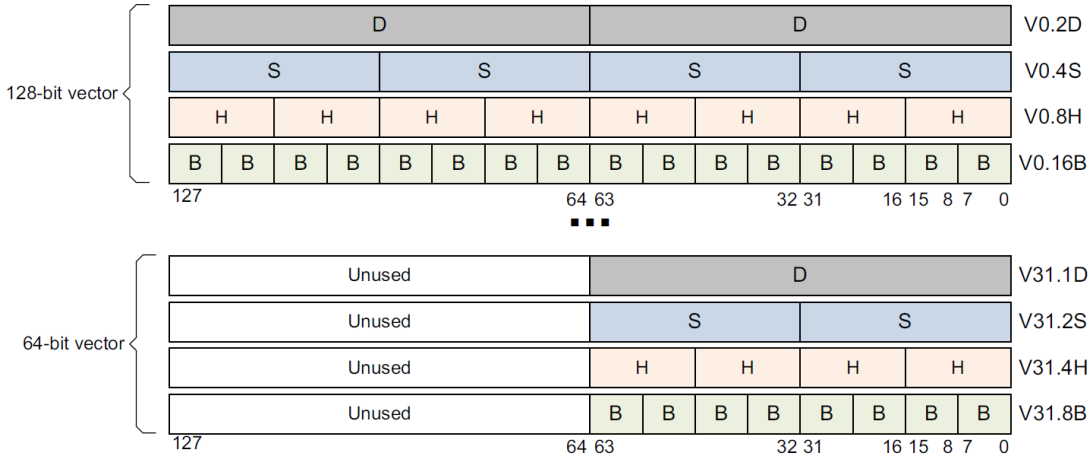
Az A64 utasításkészlet szerves részét képezik a lebegőpontos és SIMD (NEON) műveletek. Az architektúra egységesen kezeli ezeket, például közös (de az általános 64 bites regiszterekkel nem azonos) regiszterkészletet használnak a skalár lebegőpontos és a vektorműveletek. Az ARMv7 esetén láttuk, hogy a VFP műveletek teljesen kompatibilisek voltak az IEEE-754 szabvány előírásaival, de a NEON csak egyszeres pontosságú lebegőpontos számokat támogatott, és például a denormalizált számokat nullára kerekítette. Ezzel szemben az ARMv8 NEON vektorműveletei már teljes lebegőpontos funkcionalitással bírnak, és AArch64 állapotban támogatják a dupla pontosságot is. (Természetesen a SIMD műveletek nem csak lebegőpontos, hanem egész számokat is kezelnek.)

Az általános célú 64 bites regiszterek mellett az ARMv8-A architektúra 32 db 128 bites NEON regisztert tartalmaz (V0–V31). Ezek támogatják a double (64 bit), single (32 bit) és half (16 bit) lebegőpontos formátumokat, valamint a 64, 32, 16 és 8 bites egész formátumokat is. Használhatók skalár regiszterekként is mind egész, mind lebegőpontos műveletvégzéshez. Ilyenkor olvasásnál mindig csak a szükséges számú alsó bitet vesszük figyelembe, íráskor pedig a felső, nem használt bitek nullázódnak. Vektor módban itt is 64 bites vagy 128 bites

[40] Forrás: ARM Cortex-A Series Programmer's Guide for ARMv8-A

Copyright © 2021 / BME AUT

vektorokkal dolgozhatunk (ahogy az ARMv7-ben láttuk), amelyek adatmérettől függő számú vektorelemet tartalmaznak. Mivel az ARMv7-hez képest dupla mennyiségű regiszter áll rendelkezésre teljes, 128 bites kiépítettségben, nincs szükség a 64 bites vektor regiszterek ott látott párba állítására. Minden 64 bitesként hivatkozott vektor regiszter a 128 bites regiszterek alsó felének felel meg. A különböző vektorméretek, elemméretek és elemszámok egy egységes elnevezési konvencióval adhatók meg a SIMD műveletek operandusaiként (ld. 3.36. ábra). A vektor regiszter neve (*Vn*) után ponttal elválasztva kell megadni, hogy hány elemű és milyen elemméretű vektorként értelmezzük a regisztertartalmat. Például V3.4S azt jelenti, hogy a V3 regisztert 4 db 32 bites (single-precision) szóból álló vektorként (tehát 128 bitesként) tekintjük. Ha pedig például V3.4H-ként hivatkozunk rá, akkor az egy 4 db 16 bites (half-precision) elemből álló vektort (vagyis a 64 bites változatot) jelenti. AArch32 állapotban a korábbi *Dn* (64 bit) és *Qn* (128 bit) regiszternevek és az ott megszokott konvenciók használatosak továbbra is a kompatibilitás miatt.



3.36. ábra – ARMv8 NEON vektor regiszterek és adatméretek[41]

[41] Forrás: ARM Cortex-A Series Programmer's Guide for ARMv8-A

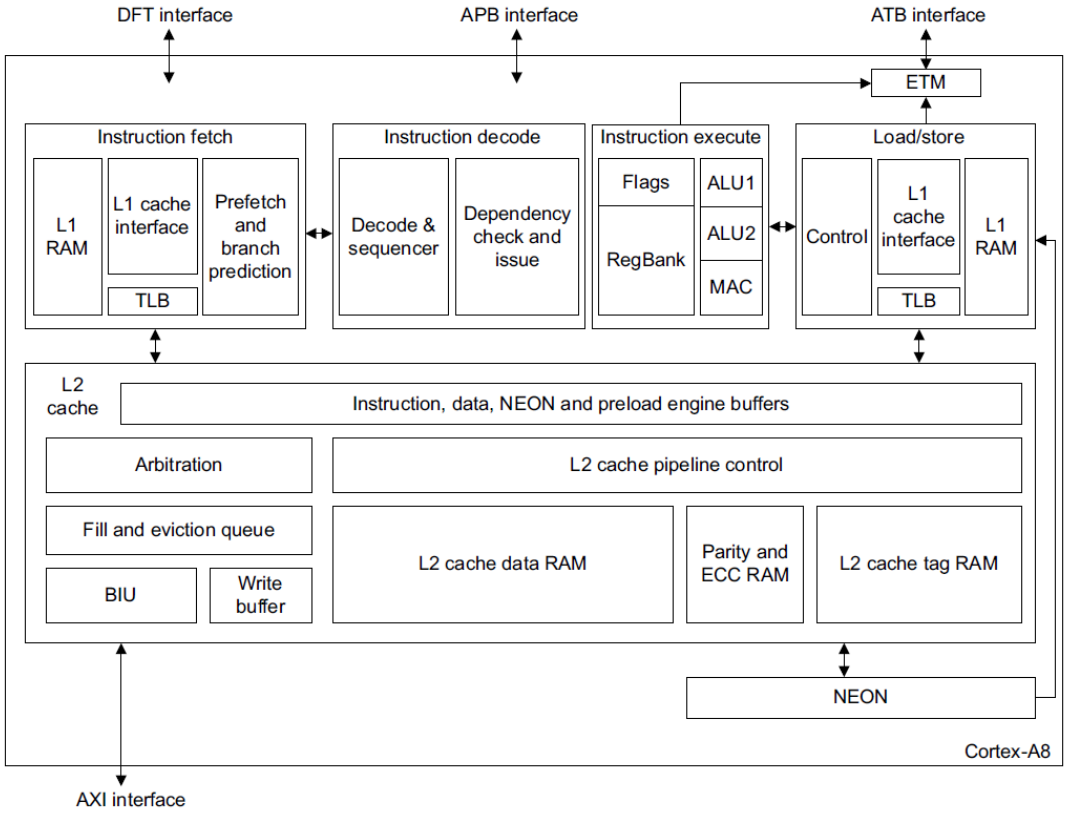
Copyright © 2021 / BME AUT

3.3 ARM Cortex-A mikroarchitektúrák

Az ARM processzorokról szóló fejezetünk végén az utasításkészlet-architektúra tulajdonságainak áttekintése után bemutatunk néhány ARM Cortex-A családba tartozó processzormagot. A mikroarchitektúrák részletes elemzése nem célunk, inkább csak betekintést szeretnék adni abba, hogyan valósulnak meg a nagyteljesítményű ARM processzorok esetén a Mikroarchitektúrák c. fejezetben látottak.

3.3.1 ARM Cortex-A8

A 2005-ben megjelent ARM Cortex-A8 volt a Cortex-A család első tagja, vagyis az ARMv7-A architektúra első implementációja. Olyan ismertebb SoC-kben került felhasználásra, mint az Apple A4 (az iPhone 3G okostelefon központi egysége) vagy a Samsung Exynos 3110 (a Galaxy S okostelefon, a Galaxy S család első tagjának központi egysége). Belső felépítésének blokkvázlata a 3.37. ábrán látható.



3.37. ábra – Az ARM Cortex-A8 processzor mikroarchitektúrája[42]

[42] Forrás: ARM Cortex-A8 Technical Reference Manual Revision: r3p2

Copyright © 2021 / BME AUT

Kétsávos (dual issue) szuperskalár csővezetékkel látták el, amely 13 fázisból áll, és sorrendi végrehajtást valósít meg. A sorrendi végrehajtás választását az alacsony fogyasztás (kisebb komplexitás) indokolja. Kétszintű integrált cache struktúrával rendelkezik, az első szinten különválasztott utasítás és adat cache-sel. A 13 fázisból 3 fázis az utasításbetöltéshez, 5 fázis a dekódoláshoz és további 5 fázis a végrehajtáshoz tartozik.

Az utasításbetöltési szakasznál megfigyelhetjük az elágazásprediktor jelenlétét, amely kétszintű adaptív elágazásjövendölést alkalmaz. A csővezeték végrehajtási szakaszában 2 db szimmetrikus ALU-t és egy különálló szorzó pipeline-t találunk. A regiszter-visszaírási fázis is ebben a szakaszban található. A működésnél korlátozást jelent, hogy mivel egyszerre két utasítás kibocsátására képes (dual issue) csővezetékről beszélünk, aktív szorzóegység vagy load/store egység mellett csak egy ALU lehet aktív a kettőből. Elérhető a processzormagban a NEON egység is, amely különálló médiamotorként az előzőtől teljesen elkülönülő 10 fázisú csővezetéket tartalmaz, amely a normál utasítás-csővezeték végéhez csatlakozik.

3.3.2 ARM Cortex A9

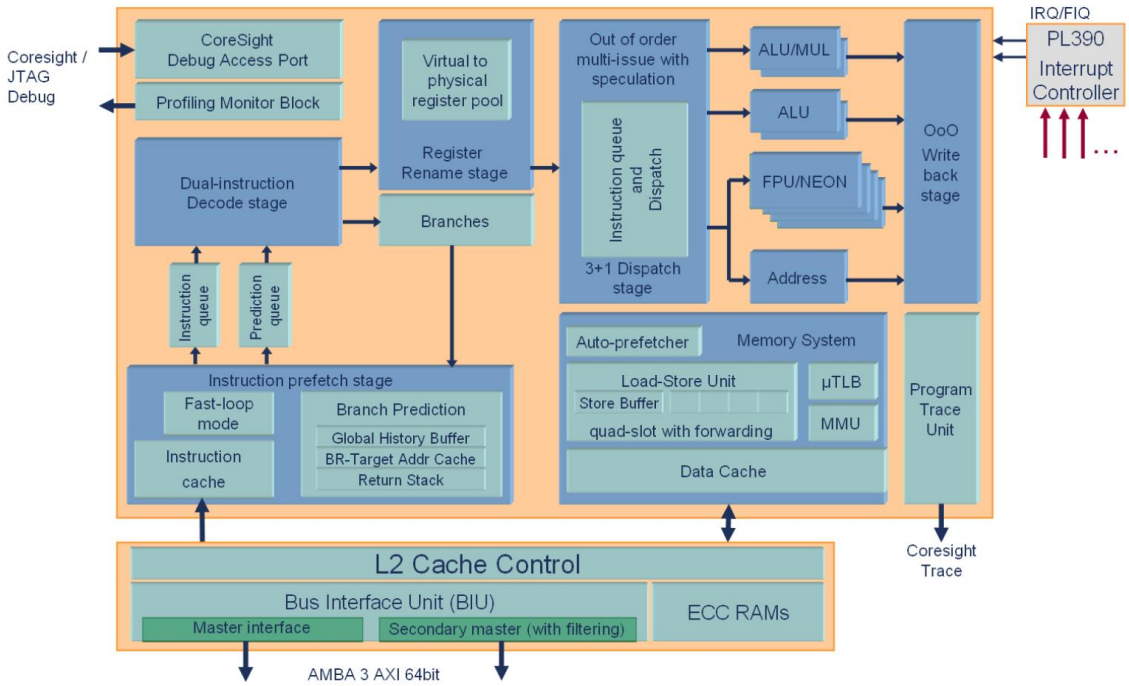
Az ARM Cortex-A9 2007-ben jelent meg, és egy széles körben használt, népszerű processzormag lett. Az elődjéhez hasonlóan ez is egy ARMv7-A architektúrára épülő, dual issue szuperskalár mikroarchitektúrát valósít meg, viszont már a sorrenden kívüli végrehajtás támogatásával. Ebből látszik, hogy ennél a processzornál a teljesítmény volt a fontosabb szempont az alacsony fogyasztással szemben.

A későbbi ARM Cortex-A magoknál is megfigyelhető az az irányelv, hogy ugyanarra az architektúrára építve közzétesznek kimondottan alacsony fogyasztású és kimondottan nagyteljesítményű magokat is. A többprocesszoros rendszerekről szóló fejezetben azt is látni fogjuk, hogy ezeket a különböző típusú magokat előszeretettel alkalmazzák egy rendszerben együtt is a szélsőségesen változó terhelések és az egymásnak ellentmondó felhasználói igények rugalmas kiszolgálására (pl. mobil eszközökben, ahol néha nagy számítási teljesítményre van szükség, de ugyanakkor az akkutöltések közötti minél hosszabb üzemidő is fontos).

Az ARM Cortex-A9 magot sokféle gyártó számos SoC-ben felhasználta, lássunk ezek közül ízelítőül néhányat:

- Apple A5, A5X (iPhone 4S, iPad 2/3)
- Samsung Exynos 4 Dual/Quad (Galaxy S2/S3)
- Nvidia Tegra 3 (Microsoft Surface, Nexus 7, HTC One X+)
- MediaTek MT6575, MT 6577 (LG Optimus L4 II/L5 II, LG G Pro Lite, Alcatel OneTouch Idol Ultra)
- Xilinx Zynq-7000 (Erről a hibrid SoC-ről részletesen is szó lesz az egylapkás rendszerekről szóló fejezetben.)

Copyright © 2021 / BME AUT



3.38. ábra – Az ARM Cortex-A9 processzor mikroarchitektúrája[43]

A csővezeték utasításbetöltési szakaszában itt is megtaláljuk a kétszintű, adaptív jövendölést alkalmazó elágazásbecslőt Return Stack Bufferrel együtt. Érdekességként megjelenik a *Fast-loop mode* az utasításbetöltésnél, amely kisméretű programciklusok esetén alacsonyabb fogyasztású működést biztosít (emlékezzünk az Intel Nehalem mikroarchitektúra Loop Stream Detector egységére).

A dekódolási szakasz után láthatjuk a sorrenden kívüli végrehajtás miatt elengedhetetlen regiszter-átnevezésért felelős blokkot, amely 32 db architekturális regisztert 56 fizikai regiszterre tud leképezni. A CPSR flag-ek átnevezéséhez pedig 8 db 9 bites fizikai regiszter áll rendelkezésre. Ezután következik az utasításszétosztó egység, a párhuzamos végrehajtó csővezeték-szakaszok, és a befejező egység. Vegyük észre, hogy a NEON csővezeték már nem különül el, hanem beépült a többi végrehajtóegység közé, ami a sorrenden kívüli végrehajtás teljesen logikus következménye

3.3.3 ARM Cortex A72

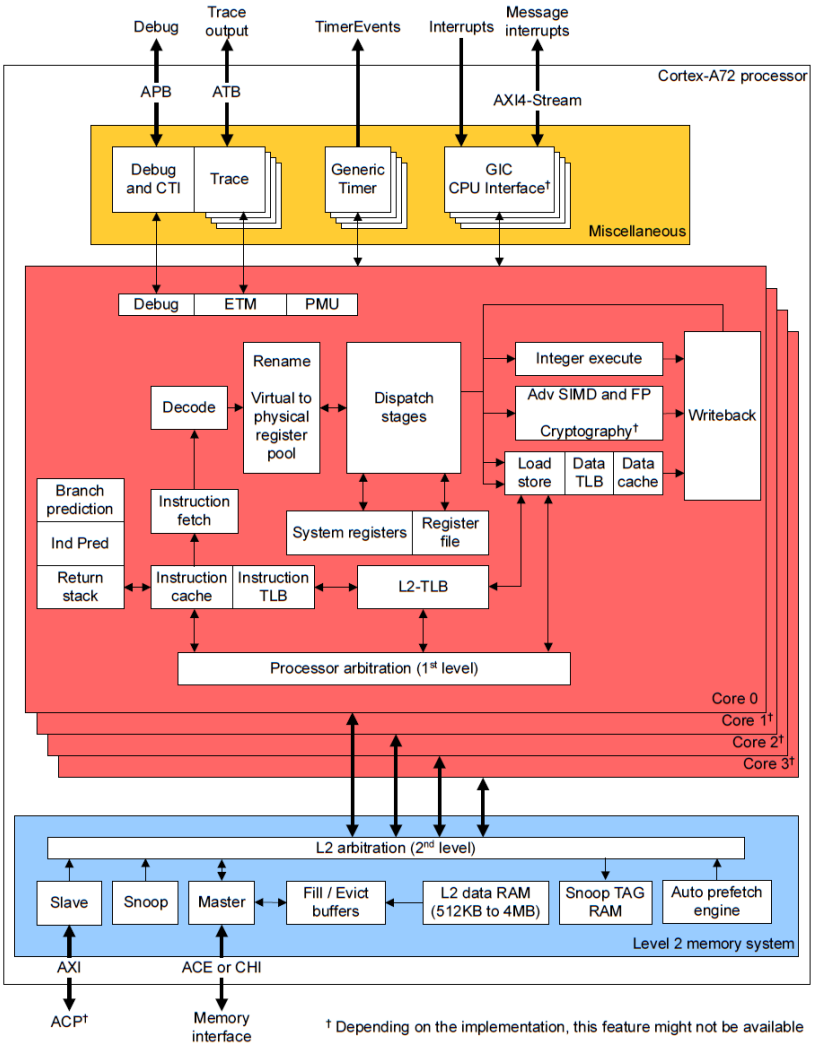
A 2016-ban megjelent ARM Cortex-A72 processzor már az ARMv8-A architektúrát valósítja meg, támogatja mind az AArch32, mind az AArch64 végrehajtási állapotokat. Ez is egy nagyteljesítményű alkalmazásokra szánt 3-utas szuperskalár processzor, sorrenden kívüli végrehajtással. Kimondottan többmagos kialakításra tervezték, a mikroarchitektúrája 1–4 magból álló ún. cluster kialakítására alkalmas. Ismertebb erre épülő SoC-k és eszközök:

[43] Forrás: ARM White Paper – The ARM Cortex-A9 Processors

Copyright © 2021 / BME AUT

- Broadcom BCM2711 (Raspberry Pi 4)
- Qualcomm Snapdragon 650, 652, 653 (Sony Xperia X, Xiaomi Redmi Note 3, Samsung Galaxy A9 (2016), LG G5 SE, Asus ZenFone 3 Ultra)

A csővezetékben itt is megtaláljuk a kétszintű dinamikus elágazásjövendölést és a Return Stack Buffert. A dekóder mindhárom utasításkészlet (A64, A32, T32) dekódolására képes. Itt is láthatjuk a regiszter-átnevezést, a szétosztó, végrehajtó és befejező egységeket. A végrehajtóegységek között találunk 2 db ALU egységet, egy integer MAC és ALU egységet, egy iteratív integer osztóegységet, egy elágazás- és feltételkiértékelő egységet, valamint egy SIMD és lebegőpontos (NEON) egységet.



3.39. ábra – Az ARM Cortex-A72 processzor mikroarchitektúrája[44]

[44] Forrás: ARM Cortex-A72 MPCore Processor Technical Reference Manual Revision: r0p3

Copyright © 2021 / BME AUT

Felhasznált és ajánlott irodalom

[52] ARM Developer. Introducing the Arm architecture. https://developer.arm.com/architectures/learn-the-architecture/introducing-the-arm-architecture/single-page (Hozzáférés: 2021.03.12)

[53] Wikipedia. ARM architecture. https://en.wikipedia.org/wiki/ARM_architecture (Hozzáférés: 2021.03.12)

[54] Wikipedia. List of ARM microarchitectures. https://en.wikipedia.org/wiki/List_of_ARM_microarchitectures (Hozzáférés: 2021.03.12)

[55] Ben Walshe. A Brief History of Arm: Part 1. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-brief-history-of-arm-part-1 (Hozzáférés: 2021.03.12)

[56] Ben Walshe. A Brief History of Arm: Part 2. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-brief-history-of-arm-part-2 (Hozzáférés: 2021.03.12)

[57] ARM7TDMI Technical Reference Manual. https://developer.arm.com/documentation/ddi0210/c/ (Hozzáférés: 2021.03.12)

[58] ARM Developer. Instruction Set Architecture. https://developer.arm.com/architectures/instruction-sets (Hozzáférés: 2021.03.12)

[59] ARM Cortex-A Series Programmer's Guide (Version: 4.0) https://developer.arm.com/documentation/den0013/d/ (Hozzáférés: 2021.03.12)

[60] Wikipedia. Euclidean Algorithm. https://en.wikipedia.org/wiki/Euclidean_algorithm (Hozzáférés: 2021.03.12)

[61] Wikipedia. Jazelle. https://en.wikipedia.org/wiki/Jazelle (Hozzáférés: 2021.03.12)

[62] NEON Programmer's Guide (Version: 1.0) https://developer.arm.com/documentation/den0018/a/ (Hozzáférés: 2021.03.12)

[63] ARM Cortex-A Series Programmer's Guide for ARMv8-A https://developer.arm.com/docs/den0024/a (Hozzáférés: 2021.03.12)

[64] Real World Technologies. ARM Goes 64-bit. https://www.realworldtech.com/arm64/ (Hozzáférés: 2021.03.12)

[65] ARM Cortex-A8 Technical Reference Manual (Revision: r3p2) https://developer.arm.com/documentation/ddi0344/k/ (Hozzáférés: 2021.03.12)

[66] ARM Cortex-A9 Technical Reference Manual (Revision: r4p1) https://developer.arm.com/documentation/100511/0401/ (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

[67] ARM White Paper – The ARM Cortex-A9 Processors
https://web.archive.org/web/20141117060156/http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf (Hozzáférés: 2021.03.12)

[68] ARM Cortex-A72 MPCore Processor Technical Reference Manual Revision: r0p3
https://developer.arm.com/documentation/100095/0003/
(Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

4 Többprocesszoros rendszerek

A többprocesszoros rendszerek alatt általános célú processzorok párhuzamos felhasználását értjük. Hasonlót már említettük a szálszintű párhuzamosság vizsgálatakor, amikor egyetlen processzormagon belül időosztással vagy szimultán módon egyszerre több programszál az utasításoknál magasabb szintű párhuzamosságot valósított meg. A többprocesszoros rendszereknél viszont arról van szó, hogy több fizikai processzor, vagy egy lapkán belül több processzormag, vagy akár több többmagos processzor egy nagyobb rendszerré összekapcsolva hajt végre különböző szálakat, folyamatokat.

4.1 Sebességnövekedés

Miért alkalmazunk több processzort egy rendszerben? Ahogy a korábban tárgyalt párhuzamosítási megoldásoknál, itt is a sebességnövekedés elérése a célunk. A számítógépek teljesítőképességének vizsgálatakor említettük az Amdahl-törvényt, ami egy bizonyos architekturális fejlesztésnek a sebességnövekedésre gyakorolt hatását adja meg. Ott párhuzamos processzorok esetére is megadtuk az összefüggést: ha N db processzorunk van, és a párhuzamosítható programrészek aránya p, akkor a sebességnövekedés a következőképpen adható meg:

$$S = \frac{1}{(1-p) + \frac{p}{N}} \tag{4.1}$$

Egy többprocesszoros rendszerben párhuzamosításra ad lehetőséget, ha egy programot több szál valósít meg, vagy több processz fut egymás mellett. Az is lehet, hogy egymástól független felhasználók folyamatai futnak egy időben, például egy szerveren. A párhuzamosítást akkor lehet megvalósítani, ha legalább annyi futásra kész programszálunk van egy időben, amennyi párhuzamos processzormag. Ha ideális esetet feltételezünk, és azt mondjuk, hogy a teljes párhuzamosíthatóság elérhető, azaz minden pillanatban minden processzor számára van feldolgozandó feladat, akkor a képletből $p = 1$ helyettesítéssel $S = N$ következik. Tehát ideális esetben a sebességnövekedés a processzorok számával egyenesen arányos.

A valóságban azonban ez mindig kevesebb ennél, még akkor is, ha feltesszük, hogy minden processzornak mindig van feladata. Ennek az az oka, hogy több processzor esetén elkerülhetetlen a közöttük zajló kommunikáció. Ugyanis gyakran előfordul, hogy különböző processzorokon futó folyamatok vagy szálak adatot osztanak meg egymással. Sok esetben szükség van a processzorok időbeli szinkronizációjára is. Például egy fizikai folyamat szimulációját végző szoftver esetén, amikor párhuzamosan, elosztott módon végezzük a részszámításokat, az időbeli mintavételezésnek megfelelő szimulációs ciklusok végén a processzoroknak be kell várniuk egymást, hiszen a következő időszelet szimulációjához

Copyright © 2021 / BME AUT

minden előző ciklusbeli részeredményre szükség lehet. Ha egy processzor hamarabb végez egy adott részfeladattal, akkor a következő kijelölt szinkronizációs ponton be kell várnia a többieket, hiszen nem áll még rendelkezésére minden bemenő adat a következő fázis számításaihoz.

Az is gyakran előfordul, hogy nem tökéletes a feladatelosztás a processzorok között. Ilyenkor időnként egyes processzorok feladat híján várakozni kényszerülnek, vagyis a teljes párhuzamosítás nem valósul meg minden pillanatban, azaz $p < 1$ lesz.

4.2 Osztályozás

A következőkben áttekintjük azokat a szempontokat, amelyek szerint a többprocesszoros rendszereket osztályozni lehet. Szerepeljen itt egy rövid felsorolás, mielőtt az egyes szempontokat és az általuk meghatározott kategóriákat részletesebben is kifejtjük:

- Feladat-hozzárendelési mód szerint
- Processzorok közötti kapcsolat szerint
- Memóriamegosztottság szerint
- Processzorok típusa és kezelése szerint

4.2.1 Feladat-hozzárendelési mód szerinti osztályozás

Az első szempont a feladat-hozzárendelés módja. Ez nem architekturális kérdés, hanem sokkal inkább a feladatok elosztásának kérdése, de természetesen az architektúra meghatározza, hogy egyes rendszerekben melyiket lehet megvalósítani.

Dinamikus feladat-hozzárendelés

A dinamikus feladat-hozzárendelés esetén a processzorok egyenértékűek, és mindenféle feladatot elláthatnak. A feladatok elosztása futásidőben alakul ki. Ennek a megközelítésnek előnye a processzorok rugalmas kihasználhatósága és az, hogy a terhelés menet közben kiegyenlíthető. Ezenkívül, ha egy processzor meghibásodik a rendszerben, annak kisebb hatása van a rendszer egészére nézve, hiszen a többi át tudja venni a szerepét. Hátránya, hogy bonyolultabb feladatkiosztó, ütemező szoftvert igényel, mert futásidőben, az aktuális helyzet elemzésével kell eldönteni, hogy melyik processzor éppen mit csináljon. Az általános célú többprocesszoros rendszerek tipikusan dinamikus feladat-hozzárendelést alkalmaznak.

Statikus feladat-hozzárendelés

A statikus feladat-hozzárendelés esetén különböző kialakítású processzorokból áll a rendszer, emiatt előre meghatározott, hogy melyik processzor milyen feladatok ellátására képes. Itt már a rendszer összeállításakor, illetve programozásakor eldől, hogy melyik processzornak mi lesz a feladata. A megközelítés előnye, hogy egyszerűbb ütemező szoftvert igényel. Hátránya, hogy

Copyright © 2021 / BME AUT

nem lehet annyira jól kiegyenlíteni a processzorok terhelését, tehát adott esetben, ha egy feladatot csak az arra alkalmas célprocesszor tud elvégezni, akkor egy másfajta, esetleg tétlen processzornak nem tudjuk azt átadni. Emiatt a rendszer érzékenyebb a meghibásodásokra is, hiszen, ha egy célprocesszor kiesik, és nincs vele egyenértékű másik processzor, akkor az egész rendszer működése meghiúsulhat.

Akár a kétféle feladat-hozzárendelési mód keveréke is megvalósítható, amikor többfajta processzort alkalmazunk egy rendszeren belül úgy, hogy mindegyikből több példány van jelen. Ilyenkor az azonos processzorok csoportjain belül dinamikus lehet a hozzárendelés, a csoportok között pedig statikus.

4.2.2 Processzorok közötti kapcsolat szerinti osztályozás

Az egyes processzorok közötti kapcsolatok alapján is megkülönböztethetjük a multiprocesszoros rendszereinket. Ez alapján lazán csatolt és szorosan csatolt rendszerekről beszélhetünk.

Lazán csatolt rendszerek

A lazán csatolt rendszereket üzenetorientált (message-passing) rendszereknek is nevezzük, mert esetükben a processzorok közötti kommunikáció üzenetváltásokon alapszik. Explicit, előre meghatározott üzenetek formájában cserélnek adatot az egyes processzorok, és az üzenetek egyben szinkronizációs eseményekként is szolgálnak. Az üzenetek adásának és vételének idejét és körülményeit a programozó határozza meg még futási idő előtt, tehát előre adott, hogy milyen események hatására történik üzenetváltás. Ennek előnye, hogy az üzenetvételi pontoknál automatikusan megvalósul a processzorok közötti szinkronizáció.

A lazán csatolt rendszerekben gyakori, hogy az egyes processzorok különböző operációs rendszerek felügyelete alatt működnek. Lazán csatolt rendszert alkothat egy számítógépen belül több processzor, vagy több különálló, hálózatba kapcsolt, közös feladaton dolgozó számítógép. Lazán csatolt rendszereket leginkább tudományos alkalmazásokban használnak, ahol előre jól meghatározható, hogy milyen adatcserék szükségesek.

Szorosan csatolt rendszerek

Általános célú alkalmazások esetén inkább a szorosan csatolt rendszerek terjedtek el. Ezekre az a jellemző, hogy a processzorok közötti kommunikáció osztott memórián alapul (shared-memory system). Ilyen kialakításban a processzorok mindig egyetlen operációs rendszer felügyelete alatt állnak. Más szóval az operációs rendszer számítási erőforrásokként tekint az egyes processzorokra, általában dinamikus módon osztva ki az egyes feladatokat.

A kommunikáció hardver által menedzselve, implicit módon valósul meg. Az adatcsere megosztott változókon, megosztott memóriaterületeken alapszik. Mivel a hardver megoldja az egyes processzorok közötti adatkonzisztenciát, ezért szoftverben nem tudhatjuk, hogy melyik processzor éppen mikor fér hozzá a megosztott adatterületekhez. Emiatt, ha szinkronizációra

Copyright © 2021 / BME AUT

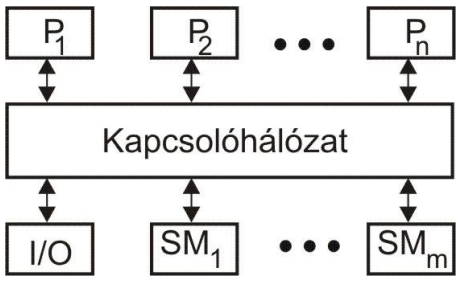
van szükség, azt speciális műveletekkel lehet kikényszeríteni (pl. barrier és lock típusú utasítások). A barrier azt jelenti, hogy ha egy adott pontra egy ilyen műveletet elhelyezünk, akkor minden egyes párhuzamosan futó szálnak oda kell érnie, mielőtt bármelyik továbbhaladhatna, tehát összevárják egymást a párhuzamosan futó feladatok. A lock pedig azt jelenti, hogy a lock típusú utasítások közötti blokkban csak egyetlen szál tartózkodhat, amivel például egy erőforrás védelme oldható meg.

4.2.3 Memóriamegosztottság szerinti osztályozás

A következő osztályozási szempont a memóriamegosztottság szerinti csoportosítás. Amint a megnevezéséből is következik, ez csoportosítás csak a szorosan csatolt (osztott memóriás) rendszerekre vonatkozik.

Uniform Memory Access (UMA)

Az első memóriamegosztottsági típus a Uniform Memory Access (UMA) modell, vázlata az 4.1. ábrán látható. Ennek lényege, hogy úgy csatlakoznak a memóriaegységek a processzorokhoz, hogy a hozzáférési idők azonosak, függetlenül attól, hogy melyik processzor melyik fizikai memóriához akar hozzáférni. A kapcsolóhálózat, amely megteremti az összeköttetést a processzorok és a memóriaegységek között, lehet busz elrendezésű, crossbar típusú, vagy többfokozatú hálózat (az utóbbi kettőre hamarosan kissé bővebben is kitérünk).



4.1. ábra – UMA rendszer vázlata

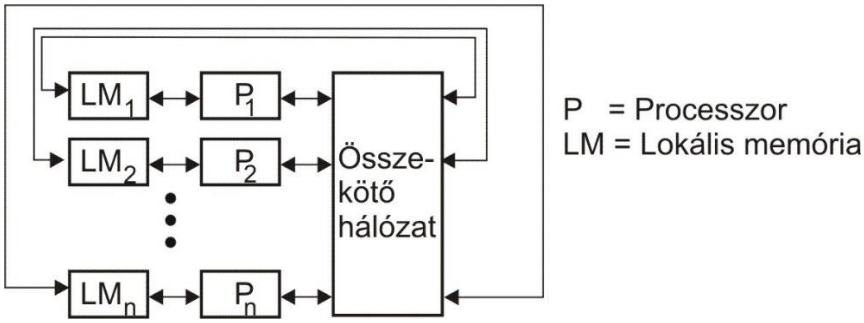
Non-Uniform Memory Access (NUMA)

A másik memóriamegosztottsági típus a Non-Uniform Memory Access (NUMA) modell, ahol a hozzáférési idő függ attól, hogy a hivatkozott adat melyik fizikai memóriában található. Ez esetben az összekötőhálózat aszimmetrikus vagy hierarchikus kapcsolatot valósít meg a processzorok és a memóriamodulok között.

Az egyik tipikus megvalósítási formája a lokális memóriájú NUMA rendszer. Az 4.2. ábrán az látható, hogy minden processzor rendelkezik egy lokális memóriával, amelyet közvetlenül elér egy gyors hozzáférési csatornán keresztül. Az összekötő hálózaton keresztül pedig a procsesszorok egymás lokális memóriáihoz is hozzáférhetnek, de itt az indirekt kapcsolat miatt

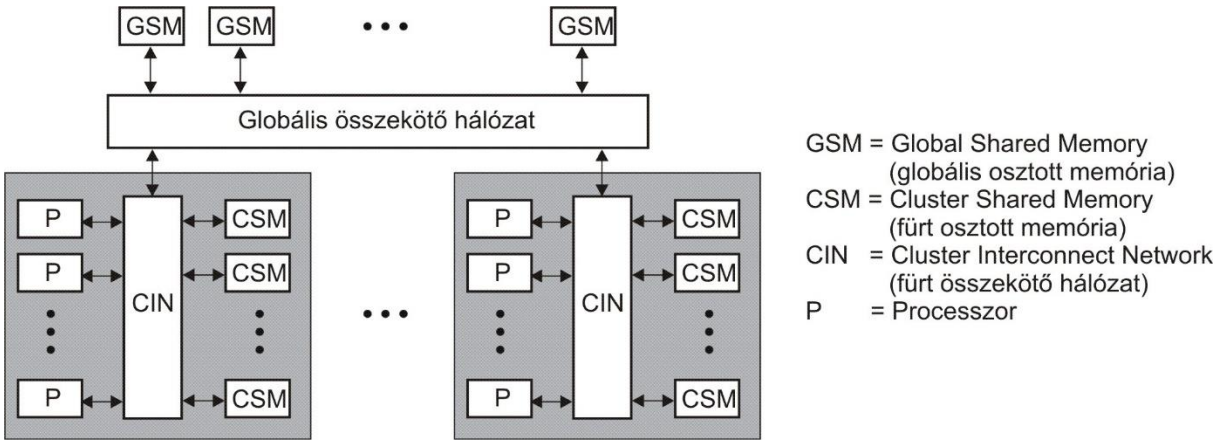
Copyright © 2021 / BME AUT

nagyobb a késleltetés. Így a keresett adat fizikai elhelyezkedésétől függően eltérő hozzáférési időkkel kell számolnunk.



4.2. ábra – Lokális memóriájú NUMA rendszer vázlata

Egy másik megközelítés a hierarchikus struktúrájú NUMA rendszer. Az 4.3. ábrán látjuk ezt az elrendezést, ahol a processzorok fürtökbe (cluster) vannak rendezve, és minden cluster rendelkezik saját megosztott memóriákkal. Ezenkívül egy globális összekötő hálózaton keresztül rendelkezésre állnak közös, globálisan megosztott memóriamodulok is. A cluster-en belüli memóriák hozzáférési ideje érthető módon rövidebb, mint a messzebb lévő, globális memóriáké.



4.3. ábra – Hierarchikus struktúrájú NUMA rendszer vázlata

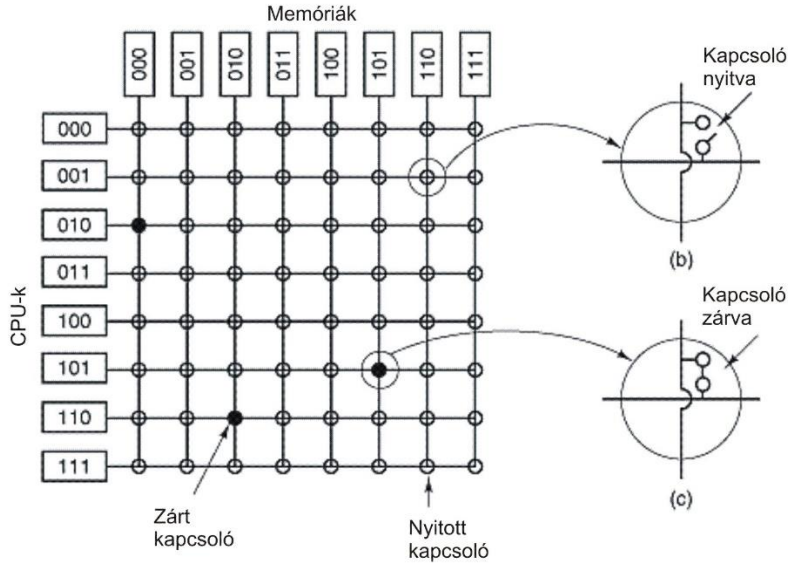
Összekötő hálózatok

Ezek után nézzük meg a kapcsolóhálózatok tipikus megvalósítási formáit. A busz elrendezésre nem térünk ki közismert volta miatt, de ejtsünk néhány szót a crossbar típusú és a többfokozatú hálózatokról.

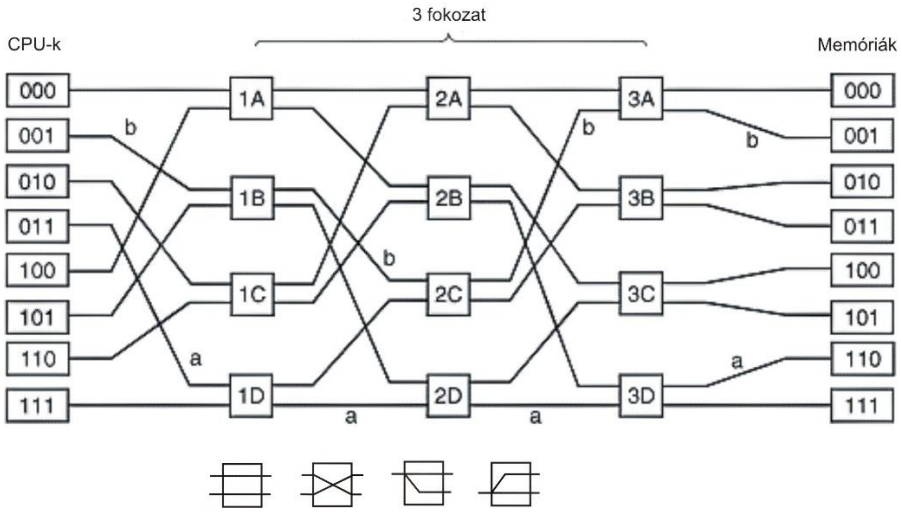
A crossbar egy nem blokkoló hálózat, ami azt jelenti, hogy egyidejűleg több processzor is hozzáférhet különböző memóriamodulokhoz (ld. 4.4. ábra). Ehhez N db processzor és N db

Copyright © 2021 / BME AUT

memória esetén N^2 kapcsolóelem szükséges bitenként. M bites kapcsolat esetén MN^2 kapcsolóelemet kell megvalósítani, tehát elég költséges a kialakítása, viszont rendkívül rugalmas hozzáférést biztosít.



4.4. ábra – Crossbar hálózat



4.5. ábra – Omega hálózat

A többfokozatú hálózatok (multi-stage interconnection network) egyik képviselője az Omega hálózat, amelynek kialakítása az 4.5. ábrán látható. Ennek lényege, hogy az egyes csatornák között köztes kapcsolófokozatok találhatók. N bemenet esetén $\log_2 N$ köztes fokozat szükséges, fokozatonként $N/2$ db kétbemenetű és kétkimenetű kapcsolóegységgel, amelyek az ábrán látható többféle módon tudják a bemeneteik és kimeneteik közötti összeköttetést megteremteni. Ennek kialakítási költsége jóval alacsonyabb a crossbar-hoz képest, viszont

Copyright © 2021 / BME AUT

blokkoló kapcsolatot biztosít, tehát nem tud bármelyik processzor bármelyik memóriához azonos időben csatlakozni, adott esetben várakozni kell arra, hogy a kívánt út felszabaduljon.

4.2.4 Processzorok típusa és kezelése szerinti osztályozás

Az utolsó osztályozási szempont szerint szintén a szorosan csatolt rendszerek altípusait különböztetjük meg. A processzorok típusa és kezelési módja szerint beszélhetünk szimmetrikus, aszimmetrikus és heterogén multiprocesszoros rendszerekről.

Szimmetrikus multiprocesszálás (SMP)

Szimmetrikus multiprocesszálás (SMP) esetén azonos típusú processzor(mag)okból épül fel a rendszer, amelyeket az operációs rendszer egyformán kezel. Ez megfeleltethető a dinamikus feladat-hozzárendelés fogalmának, hiszen, ha azonos processzoraink vannak, akkor rugalmasan lehet elosztani közöttük a feladatokat.

Aszimmetrikus multiprocesszálás (AMP)

Az aszimmetrikus multiprocesszálás (AMP) esetén az egyes processzor(mag)ok nagyon gyakran azonos architektúrájúak, de más kiépítettséggel vagy más szerepkörrel rendelkeznek. Ez esetben a feladat-hozzárendelés statikus, hiszen a kiépítettség vagy az előre rögzített szerepkör meghatározza, hogy melyik processzor mit csinálhat. Például elő lehet írni, hogy az operációs rendszer futtatásához csak az egyik processzor(mag)nak van joga, vagy az is lehetséges, hogy a különböző processzor(mag)oknál különböző periféria-hozzáféréseket építünk ki, és ez alapján oszlik meg a feladat közöttük.

Heterogén multiprocesszálás (HMP)

A harmadik ebben a felosztásban a heterogén multiprocesszálás (HMP) esete, amely funkcionalitásában hasonlíthat mind a szimmetrikus, mind az aszimmetrikus multiprocesszáláshoz, de fontos ismérve, hogy az egyes processzor(mag)ok különböző architektúra szerint épülnek fel.

A fejezet végén láthatunk majd erre példát a többmagos ARM processzorok köréből, ahol az egyes magok azonos utasításkészlet-architektúrát valósítanak meg, de mikroarchitektúrájukban eltérnek. Ilyenkor a teljes mértékben kompatibilis ISA miatt a feladatok hozzárendelése dinamikusan történik.

Ebbe a kategóriába tartoznak az olyan rendszerek is, amikor teljesen különböző utasításkészlet-architektúrájú processzorok, például egy jelprocesszor és egy általános célú processzor található egymás mellett. Az ilyen rendszereknél a feladat-hozzárendelés az eltérő képességek miatt értelemszerűen statikus.

Copyright © 2021 / BME AUT

4.3 Gyorsítótárak multiprocesszoros környezetben

Az osztott memóriás többprocesszoros rendszerekben a nagyteljesítményű működés elérése érdekében központi szerepük van a gyorsítótáraknak. Ebben az alfejezetben először átismételjük a gyorsítótárak működését és alapfogalmait, utána áttekintjük a többszintű gyorsítótárak legfontosabb jellemzőit. A mai processzorokban tipikusan két- vagy háromszintű gyorsítótárakkal találkozunk, és többprocesszoros esetben bizonyos cache szintek meg is vannak osztva a processzorok, processzorcsoportok között. Végül pedig kitérünk a gyorsítótár-koherencia kérdésére. Ez a problémakör központi jelentőségű az osztott memóriás többprocesszoros rendszereknél, hiszen ugyanannak a memóriatartalomnak több, lokális példánya lehet a különböző gyorsítótárakban. Ha egy adatot több processzor aszinkron módon használ (olvasásra és írásra is), akkor a lokális, gyorsítótárazott példányok között adatinkonzisztencia léphet fel. Ennek elkerülésére szolgálnak a gyorsítótárkoherencia-protokollok.[45]

4.3.1 Gyorsítótárak

A memóriák két elterjedt fizikai megvalósítási formája a dinamikus RAM (DRAM) és a statikus RAM (SRAM). A DRAM-ok nagyobb adatsűrűségűek és fajlagosan olcsóbbak, mert az SRAM-okhoz képest kevesebb tranzisztor szükséges egy adatbit megvalósításához. Cserébe a hozzáférési sebességük sokkal lassabb, mint az SRAM memóriáké. A nagyméretű (több GB-os) rendszermemóriákat DRAM-okkal valósítják meg. Az SRAM-ok drágábbak, de hozzáférési sebességük gyors, összemérhető a processzorok sebességével. Ezért a processzoron belüli regisztereket, kisebb memóriaegységeket tipikusan SRAM-ból alakítják ki.

A fentiekből következik, hogy a gyorsítótárakat (cache memóriákat) is SRAM-ból valósítják meg. Ezek olyan kisméretű memóriamodulok, amelyek a processzor és a főmemória között helyezkednek el, és a gyakran használt memóriablokkok másolatát tartalmazzák, éppen a gyors hozzáférés biztosítása érdekében.

A cache-ek működése az *időbeli és térbeli lokalitás elvén* alapul. Az időbeli lokalitás azt jelenti, hogy ha most használunk egy adatot, akkor valószínűleg a közeljövőben ismét hozzá fogunk férni. Például ciklusok végrehajtása esetén a ciklustörzs utasításait újra és újra fel kell hozni, valamit az általuk használt változókhoz is rövid időn belül többször hozzá kell férni. A térbeli lokalitás pedig azt takarja, hogy egy most használt adat közelében (szomszédos memóriacímeken) lévő többi adathoz is nagy valószínűséggel hozzá fogunk férni a közeljövőben. Tipikus példa erre a szekvenciális programok utasításainak felhozatala (egy

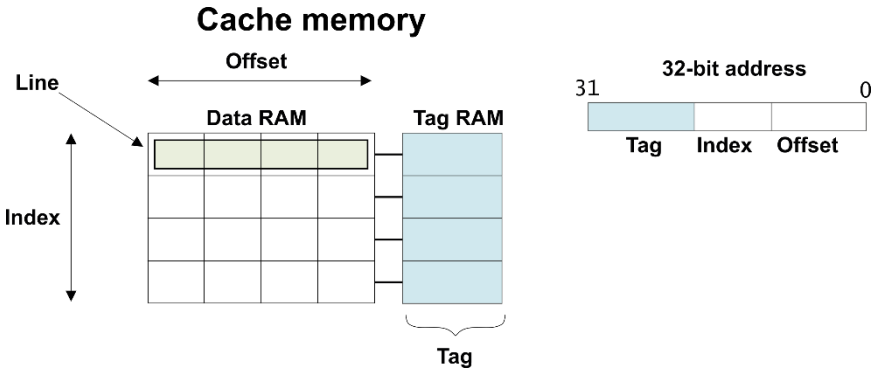
[45] Ne zavarjon meg minket a *konzisztencia* és a *koherencia* szavak együttes használata. A konzisztencia az adatokra vonatkozik, és azt jelenti, hogy a különböző memóriaterületeken található megosztott adattartalmak minden időpillanatban megfelelnek annak, amit a program alapján elvárhatunk (ezt az elvárást formálisan az ún. memóriakonzisztencia-modell írja le). A gyorsítótáraknál alkalmazott koherenciaprotokollok pedig azokat a mechanizmusokat takarják, amelyek biztosítják a konzisztenciamodell által támasztott elvárások teljesülését egy adott gyorsítótárazott memóriaterületre nézve.

Copyright © 2021 / BME AUT

utasítás után nagy valószínűséggel a következő címen lévőt kell betölteni), vagy adatmemória esetén a stack használata, illetve adattömbök kezelése.

4.3.1.1 Alapfogalmak, terminológia

Nézzük meg először a cache memóriák általános felépítését (4.6. ábra). A cache az adatok mellett címinformációt is tárol. Mivel kevesebb adat fér el benne, mint a főmemóriában, ezért az adatokat valami plusz információ alapján tudjuk csak egyértelműen azonosítani. Ez a plusz adat a **lapsorszám (tag)**, amely a memóriacím legfelső T db bitje. Ha a processzor kéréssel fordul a cache memóriához, akkor az a cím felső bitjeit összehasonlítja a letárolt tag-ekkel, és ez alapján tudja megmondani, hogy a keresett adat benne van-e a cache-ben. A tag-ek által elfoglalt hely nem számít bele a cache névleges kapacitásába.



4.6. ábra – Gyorsítótár alapfogalmak

Az adatokat a gyorsítótárban sorokba rendezve tároljuk. Egy cache **sor (line)** egyben a hozzáférés legkisebb adategysége, amely 2^M db szomszédos bájtot foglal magába. Minden sorhoz a tag-en kívül státuszinformáció is tartozik, amelyet legalább két biten tárolnak. Ezek közül az egyik a *valid bit*, amely a sor érvényességét jelzi (van-e benne használható adat). A másik pedig a *dirty bit*, amely azt mutatja meg, hogy a sor módosítva lett-e a betöltés óta, azaz a cache-ben lévő adat frissebb-e, mint a memóriabeli eredeti példány. Az **ofszet (offset)** a memóriacím alsó M db bitje, ez azonosít egy adott bájtot a soron belül.

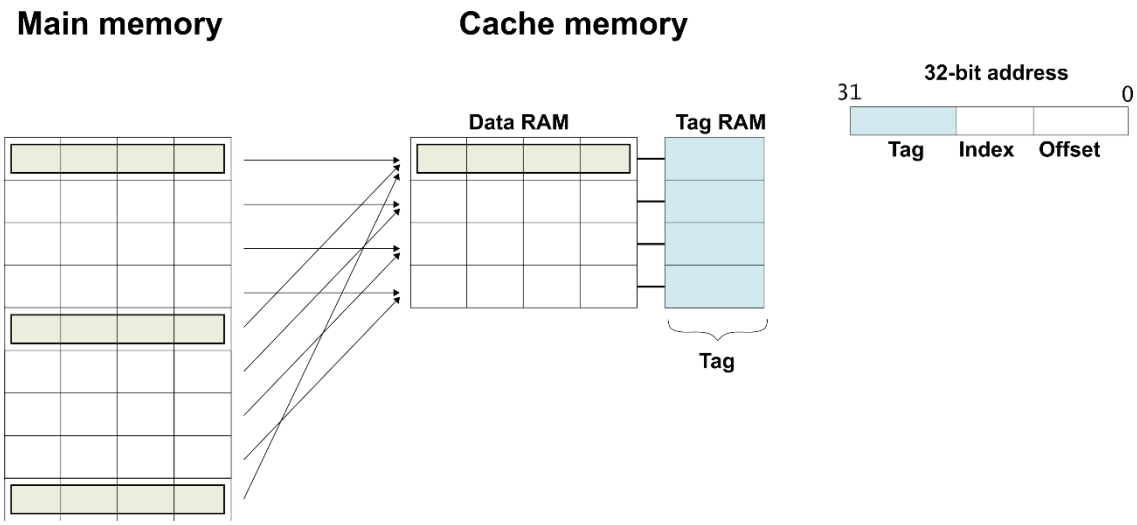
A memóriacím középső része az **index**. Ez a maradék $I = N - M - T$ db bitet jelenti, ahol N a teljes memóriacím bitszáma. Az index a sorok címzésére szolgál a cache-en belül, ezért nem kell eltárolni. A cache-ben a megcímezhető sorok száma 2^I.

4.3.1.2 Gyorsítótár-típusok

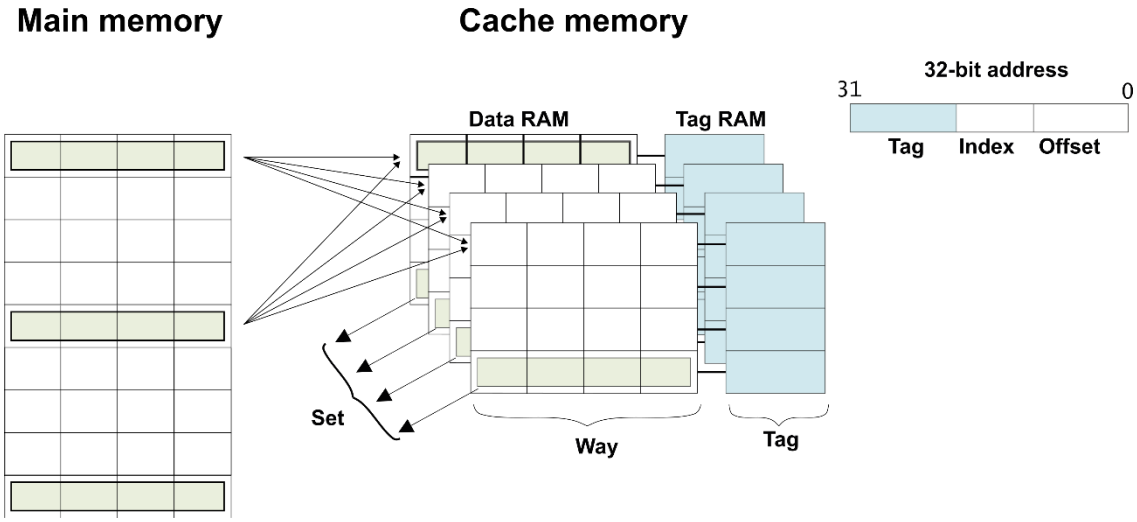
A főmemóriában található adatok hozzárendelése a gyorsítótárhoz modulo címzéssel történik az index bitek alapján. Mivel a cache-ben kevesebb (2^I db) sor található, mint a főmemóriában (2^{T+I} db), ezért ugyanarra a cache sorra több (összesen 2^T-féle) különböző memóriabeli adat is leképződhet.

Copyright © 2021 / BME AUT

Ezt az egyszerű elvet valósítják meg a **direkt leképezésű (direct-mapped) cache**-ek (ld. 4.7. ábra). Mivel minden memóriabeli sor csak egyetlen cache sorra képezhető le, ezért nagy az adatütközés valószínűsége, azaz gyakran lehet szükség egy-egy cache sor kiürítésére annak érdekében, hogy a legfrissebben hivatkozott adatot be lehessen tölteni.



4.7. ábra – Direkt leképezésű gyorsítótár



4.8. ábra – Csoportos asszociatív gyorsítótár

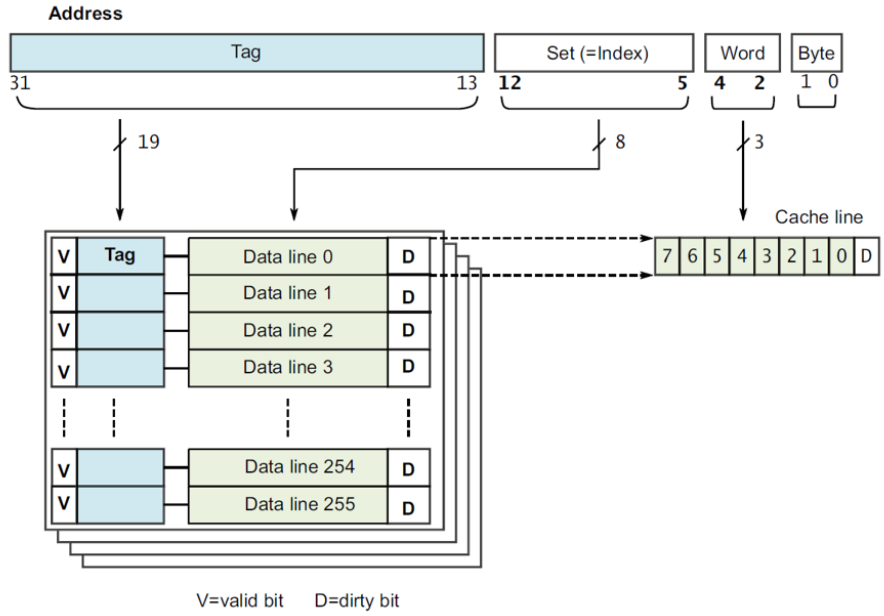
Az ütközések gyakoriságának csökkentésére szolgál a **csoportos asszociatív (set associative) cache** kialakítás (ld. 4.8. ábra). Ennek lényege, hogy a cache egyenértékű, párhuzamos **utakból (way)** áll, és az egyes utakon belül az index-szel történő címzés teljesen egyformán zajlik. Az azonos indexű, de különböző párhuzamos utakhoz tartozó cache sorok alkotnak egy **csoportot (set)**. Egy n-utas csoportos asszociatív cache esetén (az 4.8. ábrán $n = 4$) az azonos index-szel rendelkező 2^T db memóriablokk egy n-elemű csoport bármelyik sorába leképződhet. Így kisebb az ütközések valószínűsége, de azon az áron, hogy keresés esetén az azonos indexhez

Copyright © 2021 / BME AUT

tartozó összes párhuzamos cache sor tag mezőjét le kell ellenőrizni, ami a cache kialakításának komplexitását és költségét növeli.

A másik véglet a **teljesen asszociatív (fully associative) cache**, ahol minden egyes út egyetlen cache sort tartalmaz, így bármelyik memóriablokk bármelyik cache sorra leképeződhet (az index bitek száma 0). Ez a megoldás rendkívül rugalmas hozzárendelést eredményez, de csak kisméretű cache-eknél valósítható meg gazdaságosan. A gyakorlatban leggyakrabban 4-, 8-, vagy 16-utas csoportos asszociatív cache-eket szoktak használni, ugyanis ennél nagyobb asszociativitás már csak elenyésző javulást jelent a találati arányban.

Egy valós példa segítségével illusztráljuk a cache szervezés alapfogalmait. Az 4.9. ábrán az ARM Cortex-A9 processzor elsőszintű adatgyorsítótárának felépítését láthatjuk. Ez egy 32 kB méretű, 4-utas csoportos asszociatív cache. Az ARM processzorok 32 bites szavakban gondolkodnak, de a memóriájuk bájtos címzésű, a memóriacímek 32 bitesek (ARMv7-A architektúráról lévén szó). Egy cache sor 8 szóból, azaz 32 bájtból áll, így a memóriacím alsó 5 bitje lesz az ofszet. Illesztett (aligned) memóriahozzáférések esetén a legalsó két bit értéke mindig 0, ezért az ofszetbitekből csak 3 db szükséges a soron belüli szavak azonosításához. Határozzuk meg az index és a tag bitek számát ezek alapján!



4.9. ábra – Az ARM Cortex-A9 processzor L1 adat cache felépítése[46]

A 32 kB-os cache 4 útra oszlik, ez 8–8 kB-ot jelent utanként. Mivel 32 bájtos soraink vannak, minden útban $8192/32 = 256$ sor áll rendelkezésre. Ebből következően az index bitek száma 8 lesz. A 32 címbitből az 5 db ofszet biten és a 8 db index biten kívül a felső 19 bit marad, ezek alkotják a tag mezőt, amelyet minden egyes cache sorhoz le kell tárolnunk. Érdekességként azt is kiszámolhatjuk, hogy a tag-ek tárolására $4\,[út] \cdot 256\,[sor] \cdot 19\,[bit] = 19{,}456\,kbit$

[46] Forrás: ARM Cortex-A Series Programmer's Guide (Version 4.0)

Copyright © 2021 / BME AUT

járulékos tárhelyre van szükség a cache-en belül (tag RAM), ami a cache névleges kapacitásához képest plusz 7,4%-ot jelent.

Az összehasonlíthatóság kedvéért nézzük meg azt is, hogyan alakulna az index és tag bitek száma, illetve a tag RAM mérete akkor, ha ugyanez a cache direkt leképezésű vagy teljesen asszociatív lenne. Ha direkt leképezésű gyorsítótárat feltételezünk, akkor a 32 kB-nyi kapacitás nem oszlik párhuzamos utakra, hanem az összes sornak külön-külön indexe van, tehát a külön címezhető sorok száma $32\ [kB]\ /\ 32\ [bájt/sor] = 1024$. Ezeknek a címzéséhez 10 db index bit szükséges. A 32 címbitből így az 5 db ofszet biten és a 10 db index biten kívül 17 bit marad az adattal együtt letárolandó tag mező számára. A tag RAM mérete ebben az esetben $1024\ [sor] \cdot 17\ [bit] = 17{,}408\ kbit$ lenne.

Ha azt feltételezzük, hogy a példánkban szereplő cache teljesen asszociatív, akkor a sorok száma ugyanúgy 1024, de minden egyes cache sor külön-külön útnak felel meg. Ilyenkor az index bitek száma 0, tehát a 32 bites memóriacímből az 5 db ofszet biten kívül mind a 27 db maradék bit a tag-et alkotja. A tag RAM ez esetben $1024\ [sor] \cdot 27\ [bit] = 27{,}648\ kbit$ lenne, ez körülbelül másfélszer nagyobb, mint amit az előző két esetben láttunk.

4.3.1.3 Cache policy típusok

A cache-ekkel összefüggésben további fontos fogalom a **gyorsítótár-találat (cache hit)** és a **gyorsítótárhiány (cache miss)**. A találat azt jelenti, hogy a gyorsítótárnak leadott kérés esetén a hivatkozott sor benne van a gyorsítótárban, és érvényes. Hiány esetén a keresett sor nincs benne a gyorsítótárban, vagy érvénytelen. Mindkét esetben megkülönböztetjük az írási és olvasási találat (read hit, write hit), valamint az írási és olvasási hiány (read miss, write miss) eseteit.

Gyorsítótárhiány fellépésekor **sorbetöltés (linefill)** történik a memóriából. Abban az esetben, ha a betöltendő adat címében az index olyan cache sorra mutat, amely foglalt (illetve egyik úton sem szabad az adott sor a csoporton belül), akkor **sorhelyettesítést (line replacement)** kell végrehajtani. Ehhez egyrészt ki kell választanunk a törlendő sort (victim, azaz áldozat), másrészt az új adat betöltése előtt el kell végeznünk a „kilakoltatást” (eviction). Ha a victim sor módosított, akkor előbb vissza kell írni a memóriába, majd érvényteleníteni kell.

A gyorsítótárak működését különböző események bekövetkeztekor ún. cache policy-k, működési szabályok határozzák meg.

Betöltési szabály (allocation policy)

A betöltési szabály azt határozza meg, hogy milyen események hatására hajt végre sorbetöltést a cache. **Write allocation policy** esetén mind az olvasási, mind az írási hiány sorbetöltést vált ki (ezt read-write allocation-nek is nevezhetnénk). Ennek az a következménye, hogy adott esetben egy memóriaírási művelet implicit memóriaolvasást válthat ki. **Read allocation policy** alkalmazásakor csak read miss esetén történik sorbetöltés, write miss esetén a cache továbbadja az írási kérést a következő memóriaszintnek (másik gyorsítótárnak vagy a főmemóriának).

Copyright © 2021 / BME AUT

Sorcsere szabály (replacement policy)

A sorcsere szabály asszociatív gyorsítótáraknál a törlendő sor kiválasztására szolgál az azonos indexűek közül. Az elv lehet **round-robin**, amikor egy ciklikus számláló, az ún. victim counter egy csoporton belül sorban egymás után jelöli ki a sorokat a következő potenciális kilakoltatásra. **Pszeudorandom** algoritmus esetén a victim counter növelése (ál)véletlen értékekkel történik. A leggyakrabban alkalmazott megoldás a **Least Recently Used (LRU)** módszer, amelyik mindig a legrégebben használt sort jelöli ki a következő cserére.

Frissítési szabály (update policy)

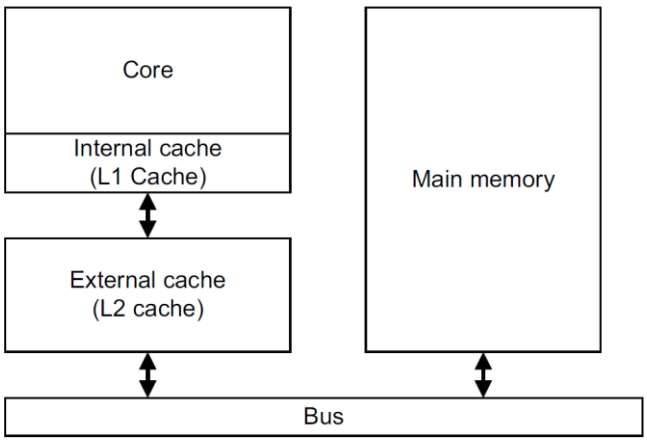
A frissítési szabálynak megfelelően megkülönböztetünk **write-through** és **write-back** cache-eket. Write-through esetben, ha írás történik, akkor az adat nem csak a cache-ben, hanem egyidejűleg a memóriában is frissül. Így a cache tartalom mindig megegyezik a memóriáéval, viszont lassabb működést eredményez a sok memóriahozzáférés miatt.

Ezzel ellentétben a write-back policy szerint működő cache írási művelet esetén nem frissíti a memóriatartalmat, csak megjelöli módosítottként (beállítja a dirty bitet). Visszaírás csak az érintett sor cseréje esetén történik. Ez a megoldás hatékonyabb működést tesz lehetővé, mert kevesebb memóriaforgalommal jár.

4.3.2 Többszintű gyorsítótárak

A modern processzorokban többszintű, hierarchikus cache struktúrák kerülnek alkalmazásra. Az 4.10. ábra egy tipikus elrendezést mutat be. Itt egy processzor látható, amelyben a lapkán belül kapott helyet az integrált elsőszintű (L1) cache. A processzor mellett található a külső modulként megvalósított második szintű (L2) gyorsítótár, amely már a főmemóriával közös buszon helyezkedik el. Az L1 cache szinte minden esetben Harvard architektúrát valósít meg, azaz különválasztott utasítás- és adatgyorsítótárként jelenik meg a processzor számára. A második szintű cache jellemzően már egyesített, tehát utasításokat és adatokat is tárol. A mai processzorokban már elterjedtebbek a háromszintű cache struktúrák, és nem ritka, hogy akár mindhárom szint a chip-en belül helyezkedik el. Többmagos processzorok esetén gyakori, hogy a második vagy a harmadik szintű cache meg van osztva a magok (esetleg a processzormagok csoportjai) között. Többszintű cache struktúrák esetén a processzortól egyre távolabb lévő gyorsítótárak egyre nagyobb kapacitásúak, viszont egyre hosszabb elérési idővel rendelkeznek.

Copyright © 2021 / BME AUT



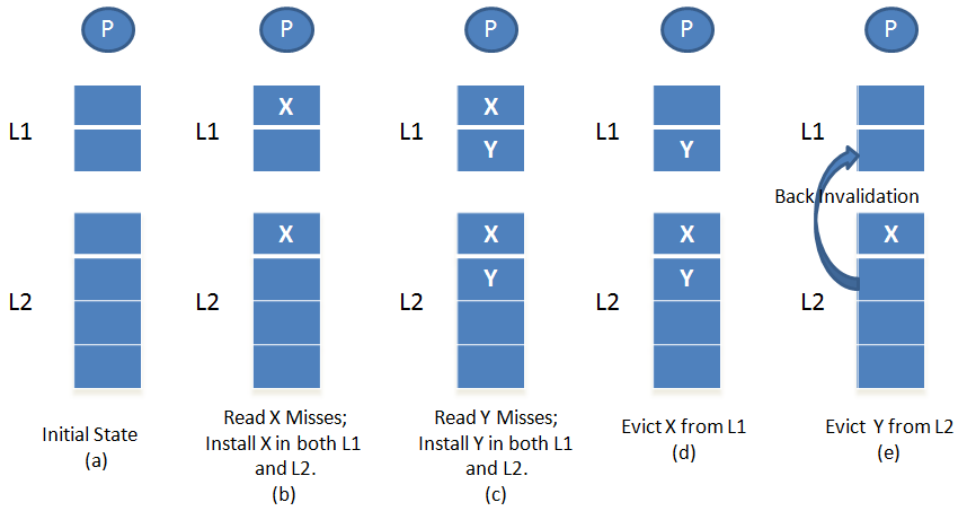
4.10. ábra – Tipikus többszintű gyorsítótár-elrendezés[47]

Többszintű gyorsítótárak esetén fontos megemlítenünk a különböző tartalmazási megoldásokat (inclusion policy). A magasabbb cache szintek más és más elv szerint tartalmazhatják az alacsonyabb szintű cache-ekben is előforduló adatokat. A tartalmazás jellege mindig két szomszédos szint között értelmezhető. A következőkben az egyszerűség miatt L1 és L2 cache-nek fogjuk nevezni ezeket, de természetesen ugyanúgy érvényes lehet mindez az L2 és L3 cache-ek között is.

Inclusive cache-ről akkor beszélünk, ha L2-nek tartalmaznia kell mindazt az adatot, amely L1-ben is megtalálható. A tartalmazás nem teljes leképezést jelent, mert csak az adat jelenlétére értendő, a hiányára nem. Vagyis, ha L1-ben érvénytelenítődik egy sor, attól L2 továbbra is tartalmazhatja. Ha viszont L2-ben kerül érvénytelenítésre egy adat, akkor azt L1-ben is érvényteleníteni kell. Tehát, ha egy adat nincs meg L2-ben, akkor biztosak lehetünk benne, hogy L1-ben sincs benne. Az inclusive cache működését az 4.11. ábrán követhetjük nyomon különböző sorbetöltési és törlési események esetén. A későbbiekben látni fogjuk, hogy ez a policy a cache-koherenciaprotokollok esetén előnyös, ott fordul elő ugyanis, hogy egy cache tartalmát nem csak a processzor felől, hanem kívülről is figyeljük. Hátránya viszont, hogy redundancia áll fenn a cache szintek között, azaz értékes helyet „pazarlunk el” azzal, hogy ugyanazt az adatot több cache-ben is benne tartjuk. Inclusive policy megvalósításához L2-nek azonos számú vagy több úttal kell rendelkeznie, mint L1-nek.

[47] Forrás: ARM Cortex-A Series Programmer’s Guide (Version 4.0)

Copyright © 2021 / BME AUT



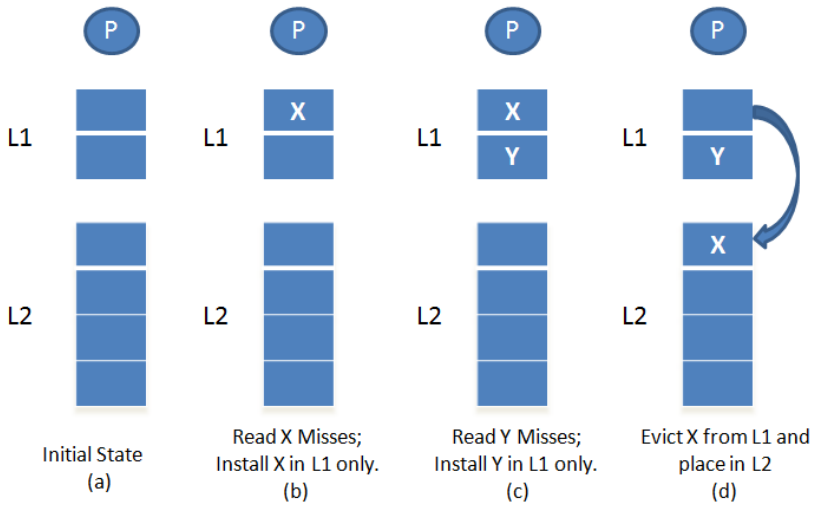
4.11. ábra – Inclusive policy[48]

Exclusive cache esetén L1 és L2 tartalma mindig diszjunkt, tehát ami L1-ben megtalálható, az nem lehet benne L2-ben. Ez tárhelykihasználás szempontjából előnyös. Az 4.12. ábrán követhetjük nyomon egy példán keresztül az exclusive cache működését. Sorbetöltéskor csak L1-ben tárolódik el az új adat, ami az exclusive L2-be csak akkor kerülhet be, amikor kiürítésre kerül L1-ből. Ilyen kialakításban az L2-t victim cache-nek is szokták nevezni, hiszen az L1-ből „kilakoltatott” áldozatok kapnak benne helyet.

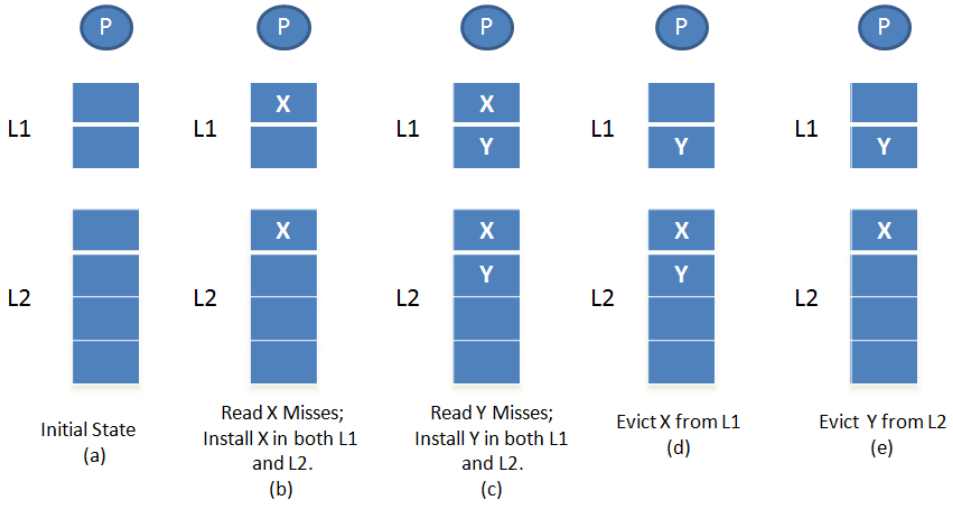
A harmadik tartalmazási megoldás a **non-inclusive non-exclusive (NINE)** policy, amelyik az előző kettő közül egyik szigorú szabályt sem alkalmazza. Sorbetöltésekor mindkét cache-ben megjelenik az adott sor (ez a non-exclusive tulajdonság), de a tartalmazást nem kötelező fenntartani a magasabb szinten. Tehát, ha L2-ben érvénytelenítésre kerül egy adat, attól még L1 továbbra is tartalmazhatja (ez a non-inclusive tulajdonság). Az 4.13. ábrán ennek a működésére is láthatunk egy példát.

[48] Forrás: https://en.wikipedia.org/wiki/Cache_inclusion_policy (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT



4.12. ábra – Exclusive policy[48]

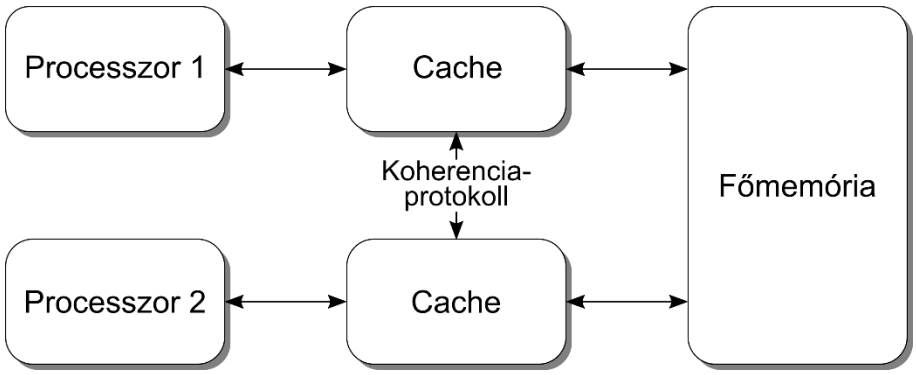


4.13. ábra – NINE policy[48]

4.3.3 Gyorsítótár-koherencia

Osztott memóriás többprocesszoros (ill. többmagos) rendszerekben processzoronként saját, lokális gyorsítótárak vannak jelen, amelyek a főmemória és az egyes processzorok között helyezkednek el. A megosztott adatok több példányban is léteznek, hiszen a főmemórián kívül a lokális cache-ekben is jelen vannak. Ilyenkor meg kell akadályozni, hogy az egyes cache-ek és a memória között adatinkonzisztencia lépjen fel, azaz nem fordulhat elő, hogy ugyanazon a címen eltérő adatot lássanak a processzormagok. Az inkonzisztencia megakadályozása a cache-koherenciaprotokollok feladata.

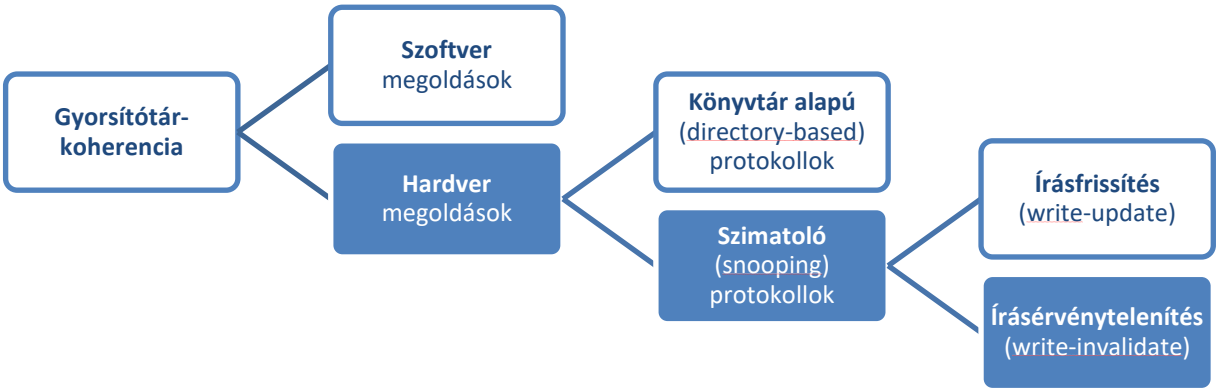
Copyright © 2021 / BME AUT



4.14. ábra – A gyorsítótárkoherencia-protokollok szerepe

Inkonzisztenciát a megosztott adatok írása okozhat. Write-back cache esetén az írott adat csak a lokális cache-ben frissül, a memóriában és a többi cache-ben nem. A write-back elv csak azért vállal felelősséget, hogy sorhelyettesítés esetén visszaírja a módosítást a memóriába. Ha még ezelőtt egy másik processzor is felülírja az adat nála lévő lokális példányát, akkor a visszaírások után valamelyik cache inkonzisztens másolattal fog rendelkezni, hiszen a memória a legutóbbi visszaírás utáni állapotot tükrözi. Ennek elkerülése érdekében a koherenciaprotokoll feladata, hogy adat írásakor a többi lokális gyorsítótárat is értesítse erről, még a memóriába való visszaírás előtt.

Write-through cache esetében ugyanígy előfordulhatna a cache tartalmak közötti inkonzisztencia, hiszen annak ellenére, hogy ott minden íráskor azonnal frissül a memóriatartalom is, a többi cache-ben jelenlévő példányok nem frissülnek. Ilyenkor is szükség van a koherenciaprotokollra a többi cache értesítése érdekében.



4.15. ábra – Gyorsítótárkoherencia-protokollok csoportosítása

A gyorsítótár-koherencia megvalósítására számos megoldás született. Az 4.15. ábrán egy hierarchikus struktúrában láthatjuk ezek csoportosítását. Léteznek szoftver és hardver megoldások a probléma kezelésére. A hardveres megközelítések között könyvtár-alapú (directory-based) és szimatoló (snooping) protokollokat különböztetünk meg. Ez utóbbiak

Copyright © 2021 / BME AUT

között pedig találhatunk írásfrissítéses (write-update) és írásérvénytelenítéses (write-invalidate) módszereket. Az ábrán sötét háttérrel jelöltük az elterjedtebben alkalmazott megközelítéseket.

A következőkben áttekintjük az egyes kategóriák legfontosabb jellemzőit, majd egy hardver alapú, írásérvénytelenítéses szimatoló protokollt, a széles körben használt MESI protokollt fogjuk részletesen bemutatni.

4.3.3.1 Szoftver megoldások

A szoftver alapú gyorsítótár-koherencia megoldások járulékos hardver nélkül, a fordítóprogram és az operációs rendszer segítségével kezelik a problémát. Általában konzervatív döntéseket kell hozniuk, amelyek alacsonyabb hatékonysághoz vezetnek, viszont így a hardver komplexitása megspórolható. A szoftver megoldások fordítási időben határozzák meg, hogy mely adatokat nem biztonságos gyorsítótárba helyezni a megosztottság miatt, és futásidőben az operációs rendszernek a feladata megakadályozni a betöltést. Ennek hátránya, hogy éppen a gyakran használt megosztott adatok esnek el a gyorsítótárazás lehetőségétől. Egy hatékonyabb szoftveres megközelítés az, amikor a kód analízise során azt állapítják meg, hogy melyek az egyes adatok biztonságos és kritikus periódusai. A kritikus időszakokban pedig explicit szinkronizációs utasításokkal biztosítják a koherenciát.

4.3.3.2 Hardver megoldások

A hardver megoldások futási időben észlelik az inkoherencia esélyét, és egy előre rögzített protokoll szerint avatkoznak be ennek megakadályozására. Ezek a szoftver megoldásoknál nagyobb hatékonyságot eredményeznek, hiszen a problémával csak akkor foglalkoznak, amikor az fellép. Másik nagy előnyük, hogy a programozó és a fordítóprogram számára transzparensen működnek, vagyis egy megfelelően működő hardveres koherencia-protokoll esetén az inkoherencia problémája a szoftver elől teljesen rejtve marad.

A hardver megoldásoknak két fő típusa van: a könyvtár alapú protokollok nagyobb rendszerekben, a szimatoló protokollok pedig kisebb, néhány processzorból álló rendszerek esetében működnek hatékonyabban.

Könyvtár alapú (directory-based) protokollok

A könyvtár alapú koherencia-protokollok egy központi vezérlőt alkalmaznak a cache-ek állapotának szinkronizálására. Ez egy központi könyvtár (directory) segítségével tartja nyilván, hogy egyes memóriablokkoknak mely gyorsítótárakban van másolata, és az milyen állapotú. A központi vezérlőt minden egyes cache műveletről értesíteni kell. Olvasási hiány (read miss) esetén a központi vezérlő intézi a betöltést a memóriából vagy egy másik cache-ből, amely már tartalmazza a kért sort. Ha egy megosztott sor írására kerül sor, akkor a központi vezérlő vezényletével egy többlépcsős kommunikáció zajlik le:

1. Az írni szándékozó cache kizárólagos jogot kér az adott sorra.
2. A vezérlő értesíti a többi gyorsítótárt az érvénytelenítésről.

Copyright © 2021 / BME AUT

3. Az érintett sort tartalmazó többi gyorsítótár elvégzi az érvénytelenítést (ha valamelyiknél módosított példány van, az ennek keretében visszaírja az adott sort a memóriába), majd nyugtázza az érvénytelenítés megtörténtét.
4. A vezérlő megadja a kizárólagos írási jogot a kérelmezőnek.

Ez sok járulékos kommunikációt eredményez, ami késleltetést okoz. Előnye viszont, hogy jól skálázódik nagyobb rendszerekre is, mivel az egyes tranzakciókról csak az érintett cache-eknek kell tudomást szerezniük. Emiatt olyan, komplex kommunikációs hálózattal rendelkező, nagy rendszerekben hatékony, amelyekben az elosztott, központi vezérlő nélküli megoldások nem megvalósíthatóak (tipikusan a legalább több tíz processzorból álló rendszerekre kell itt gondolni).

Szimatoló (snooping) protokollok

Szimatoló protokollok esetén a felelősség el van osztva az egyes cache vezérlők között. A vezérlőknek a többieket is érintő eseményekről (mint pl. egy sor módosítása) broadcast jelleggel értesítést kell küldenie. Minden vezérlőnek figyelnie, „szimatolnia” kell az adatforgalmat, és annak megfelelően frissítenie kell a saját cache-ében a sorok állapotát és tartalmát. Emiatt ez a megközelítés leginkább busz típusú kommunikációs rendszerekben alkalmazható, hiszen itt valósítható meg legkönnyebben az üzenetszórásos kommunikáció. Mivel minden vezérlő minden eseményről automatikusan értesül, ezért nincs szükség a könyvtár alapú protokolloknál látott handshake-jellegű kommunikációra, ez kisebb késleltetést, gyorsabb működést eredményez. Ugyanakkor ez a módszer nem skálázódik jól nagy rendszerekre, mert egyre több processzor esetén a cache vezérlők számának növekedésével a broadcast üzenetek egyre jobban elárasztják a közös buszt, és a busz sávszélessége jelenti a szűk keresztmetszetet. Másrészt a nagyobb rendszerekben elterjedtebben használt komplex kommunikációs hálózatokban (crossbar, többfokozatú hálózat) a szimatolás nehezen kivitelezhető.

A szimatoló protokollokon belül az **írásérvénytelenítés (write-invalidate)** alapú megközelítés azt jelenti, hogy ha egy cache írni akar egy megosztott sort, akkor a többi, ugyanezen sort tartalmazó cache-nek érvénytelenítenie kell a saját példányát. Így minden adatmódosítás után csak egyetlen érvényes példány lesz megtalálható a cache-ekben, és ha egy másik processzor később hozzá akar férni a kérdéses adathoz, akkor azt újra be kell tölteni a hozzá tartozó cache-be.

Az 4.1. táblázatban egy egyszerű példán keresztül követhetjük nyomon az írásérvénytelenítés működését. Itt három processzor egy adott cache sorra vonatkozó műveleteit láthatjuk, és megfigyelhetjük a három cache-ben ennek a sornak az állapotváltozásait. Félkövérrel jelöltük azt, ha változás lép fel a sor állaptában, az alsó index pedig a sor adattartalmára utal. Figyeljük meg a táblázat utolsó előtti sorában, hogy a megosztott adat írásakor az érintett adat érvénytelenítésre kerül az 1-es és 2-es cache-ekben, és amikor a 2-es processzor újra olvasni akarja az adatot, akkor ismét sorbetöltést kell végeznie a 2-es cache-nek. Fontos hozzátennünk, hogy ilyen esetekben nem szükséges mindig a főmemóriából betölteni a módosított adatot, hiszen az a legutóbbi módosítást végző cache-ben megtalálható. Így tipikusan a gyorsabb elérést biztosító szomszéd cache-ből történik az adat betöltése.

Copyright © 2021 / BME AUT

Események	Cache 1	Cache 2	Cache 3
Kezdeti érték	Nincs másolat (érvénytelen)	Nincs másolat (érvénytelen)	Nincs másolat (érvénytelen)
Processzor 1 olvassa a sort	**Csak olvasható másolat$_1$**	Nincs másolat (érvénytelen)	Nincs másolat (érvénytelen)
Processzor 2 olvassa a sort	Csak olvasható másolat$_1$	**Csak olvasható másolat$_1$**	Nincs másolat (érvénytelen)
Processzor 3 írja a sort (a többi másolat érvénytelenítődik)	**Nincs másolat (érvénytelen)**	**Nincs másolat (érvénytelen)**	**Írható másolat$_2$**
Processzor 2 olvassa a sort	Nincs másolat (érvénytelen)	**Csak olvasható másolat$_2$**	**Csak olvasható másolat$_2$**

4.1. táblázat – Írásérvénytelenítés példa

Az **írásfrissítéses (write-update)** módszer lényege, hogy megosztott sor írása esetén az érintett cache nem csak egy értesítést küld ennek tényéről, hanem magát az új adatot is kiteszi a buszra. Így a többi cache is frissíteni tudja a sor nála lévő példányát. Ez azért előnyös, mert a többi cache-nek emiatt nincs szüksége újabb betöltési műveletre az adat későbbi használata előtt. Viszont a generált buszforgalom általában nagyobb, mint az írásérvénytelenítéses módszer esetén.

Az előző példánkban látott események hatását írásfrissítés esetén az 4.2. táblázat tartalmazza. Jól észrevehető különbség, hogy olvasási kérésnél is írható másolat keletkezik a kért adatról az egyes cache-ekben, hiszen minden írás hatása azonnal meg fog jelenni a többi megosztott példányban is. Egy ilyen eseményt a táblázat utolsó előtti sorában láthatunk. Amikor az adat módosítása után egy másik processzor olvasni akarja azt, a saját cache-ében már a friss példányt találja, tehát nincs szükség járulékos sorbetöltésre.

Események	Cache 1	Cache 2	Cache 3
Kezdeti érték	Nincs másolat (érvénytelen)	Nincs másolat (érvénytelen)	Nincs másolat (érvénytelen)
Processzor 1 olvassa a sort	**Írható másolat$_1$**	Nincs másolat (érvénytelen)	Nincs másolat (érvénytelen)
Processzor 2 olvassa a sort	Írható másolat$_1$	**Írható másolat$_1$**	Nincs másolat (érvénytelen)
Processzor 3 írja a sort (update küldése a többi cache felé)	**Írható másolat$_2$**	**Írható másolat$_2$**	**Írható másolat$_2$**
Processzor 2 olvassa a sort	Írható másolat$_2$	Írható másolat$_2$	Írható másolat$_2$

4.2. táblázat – Írásfrissítés példa

Copyright © 2021 / BME AUT

Az írásérvénytelenítés akkor hatékonyabb, ha nagymértékű adatlokalitás érvényesül, azaz egy megosztott adatot egy processzor gyakran módosít, de a többiek csak ritkán használják. Ilyenkor az írásfrissítés által generált plusz forgalom fölöslegesen terhelné a buszt.

Az írásfrissítés viszont akkor jelent hatékonyabb megoldást, ha egy megosztott adatot egy processzor gyakran ír, a többi processzor pedig gyakran olvassa, vagy a megosztott adatot gyakran írja több processzor is. Ilyen esetben az írásérvénytelenítés rengeteg sorbetöltést okozna, ami nagyobb buszterhelést jelentene annál, mint amikor minden módosításkor csak egyszer kerül ki a buszra az új adat.

4.3.4 A MESI protokoll

Az egyik széles körben használt szimatoló, írásérvénytelenítéses módszer az ún. MESI protokoll, amely write-allocate, write-back típusú gyorsítótárak számára nyújt megoldást a cache-koherencia biztosítására. Vagyis minden írási kérés esetén sorbetöltés történik (write-allocate), és a módosítások nem azonnal, csak érvénytelenítés esetén íródnak vissza a memóriába (write-back).

A MESI protokoll neve a cache sorok lehetséges állapotainak kezdőbetűiből áll össze. A sorok Modified, Exclusive, Shared vagy Invalid állapotban lehetnek.

Modified (módosított) állapotban az a sor van, amelyik kizárólagosan csak az aktuális cache-ben fordul elő, és tartalma módosult a memóriához képest. Ebből következik, hogy ha később egy másik cache is igényli ezt a sort, akkor előbb visszaírásra lesz szükség.

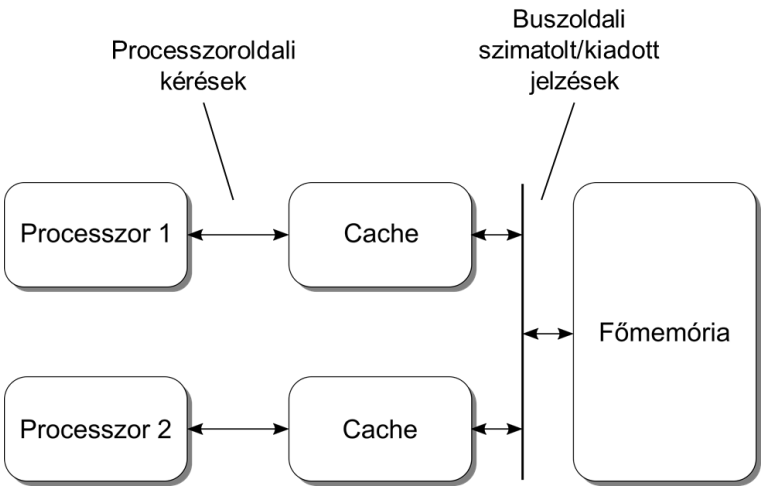
Exclusive (kizárólagos) állapot esetén a sor szintén csak az aktuális cache-ben van jelen, de tartalma a memóriához képest változatlan. Ha más cache is olvassa ugyanezt a sort, akkor Shared állapotba kerül. Ha a lokális példányt módosítja a processzor, akkor a sor állapota Modified lesz.

A **Shared (megosztott)** állapot jelentése, hogy a sor több cache-be is benne lehet, és tartalma megegyezik a memóriabeli példánnyal. Itt fontos, hogy a megosztott állapot csak valószínűsíti, de nem garantálja, hogy tényleg több példányban is megvan az érintett adat. Ha például két cache tartalmazza ugyanazt a sort, de az egyikből valamiért törlésre kerül, attól a másikban lévő példány állapota továbbra is Shared marad, nem kell Exclusive-ra állítani.

Invalid (érvénytelen) állapotban lévő cache sor pedig nem tartalmaz használható adatot.

Az egyes cache sorok állapotváltozásait a MESI protokoll különböző eseményekhez rendeli. Az 4.16. ábrán egy osztott memóriás többprocesszoros rendszer cache elrendezésének egyszerűsített vázlatát látjuk. Az egyes gyorsítótárakhoz a saját processzoruk olvasási és írási igényekkel fordulhat, ezeket az eseményeket *processzoroldali kéréseknek* nevezzük. A gyorsítótárakat és a memóriát összekötő buszon pedig a cache-ek által vezérelt kommunikáció folyik a memóriával és egymással. A memóriaműveleteket kezdeményező gyorsítótárak különböző *buszoldali jelzéseket* adnak le, amelyeket a többi cache „kiszimatol”, és azoknak megfelelően cselekszik.

Copyright © 2021 / BME AUT



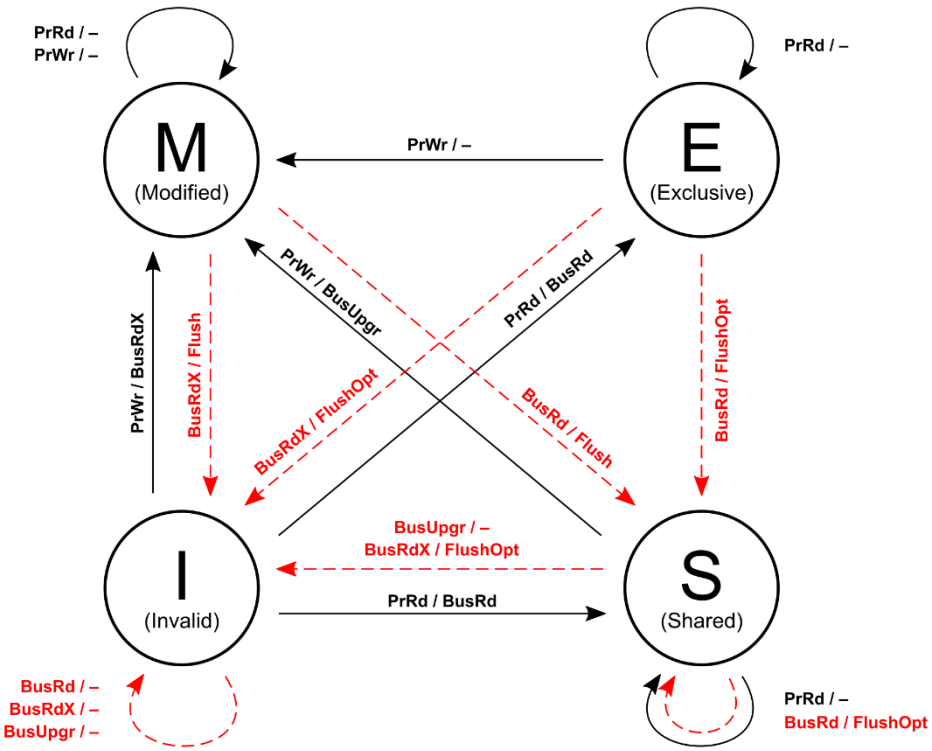
4.16. ábra – Processzoroldali és buszoldali kérések szimatoló protokoll esetén

A MESI protokollban a következő események váltanak ki állapotváltozásokat, illetve lehetnek állapotváltozások következményei:

- Processzoroldali kérések
  - **PrRd** – Processor Read
  - **PrWr** – Processor Write
- Buszoldali szimatolt/kiadott jelzések
  - **BusRd** – Read miss esetén kérelem adat olvasásra
  - **BusRdX** – Write miss esetén kérelem adat betöltésére, kizárólagos írási joggal
  - **BusUpgr** – Write hit esetén jelzés, hogy módosult az adat
  - **Flush** – Módosított cache sor tartalma a buszra kerül, memória-visszaírás és cache-to-cache transzfer céljából
  - **FlushOpt** – Cache sor a buszra kerül cache-to-cache transzfer céljából

A MESI protokoll állapotátmeneteit az 4.17. ábra szemlélteti. A processzoroldali kérések által kiváltott állapotváltozásokat fekete folytonos nyilak jelzik, a buszoldali szimatolt jelzések hatására bekövetkező állapotváltásokat pedig piros szaggatott nyilak. Az egyes nyilakra ráírtuk, hogy milyen kérés vagy jelzés a kiváltó ok, és a „/” jel után az is szerepel, hogy milyen további eseményt vagy jelzést vált ki az állapotváltozáson kívül. Az egyes processzoroldali kérések különböző állapotokban kiváltott hatását az 4.3. táblázat részletezi, az 4.4. táblázatban pedig a buszoldali, szimatolt jelzések állapotfüggő hatásait kísérhetjük figyelemmel.

Copyright © 2021 / BME AUT



4.17. ábra – A MESI protokoll állapotgépe

Állapot	Kérés	Válasz
Invalid	PrRd	**Read miss:** (1) **BusRd** kiküldése. (2) Ha valamelyik cache találatot jelez és elküldi az adatot, akkor → **Shared**, (3) egyébként → **Exclusive**, és adat betöltése a memóriából.
	PrWr	**Write miss:** (1) **BusRdX** kiküldése. (2) Ha valamelyik cache-ben megvan az adat, akkor elküldi, egyébként betöltés a memóriából. (3) A többi cache, (amelyik tartalmazza) érvényteleníti a sort. (4) → **Modified**
Exclusive	PrRd	**Read hit:** Nincs buszra küldött jelzés. Nincs állapotváltás.
	PrWr	**Write hit:** Nincs buszra küldött jelzés. → **Modified**
Shared	PrRd	**Read hit:** Nincs buszra küldött jelzés. Nincs állapotváltás.
	PrWr	**Write hit:** (1) **BusUpgr** kiküldése, hiszen másnál is megvolt az adat. (2) A többi cache érvényteleníti a sort. (3) → **Modified**
Modified	PrRd	**Read hit:** Nincs buszra küldött jelzés. Nincs állapotváltás.
	PrWr	**Write hit:** Nincs buszra küldött jelzés. Nincs állapotváltás.

4.3. táblázat – A MESI protokoll állapotátmenetei processzoroldali kérések hatására

Copyright © 2021 / BME AUT

Állapot	Kérés	Válasz
Invalid	BusRd, BusRdX, BusUpgr	Nincs állapotváltás. Jelzés figyelmen kívül hagyása.
Exclusive	BusRd	(1) → **Shared** (2) **FlushOpt** jelzés + tartalom megosztása a buszon **a kérelmező cache számára**
	BusRdX	(1) → **Invalid** (2) **FlushOpt** jelzés + tartalom megosztása a buszon **a kérelmező cache számára**
Shared	BusRd	(1) Nincs állapotváltás (másik cache olvassa az adatot, ami továbbra is megosztott lesz) (2) **FlushOpt** jelzés + tartalom megosztása a buszon **a kérelmező cache számára**
	BusRdX	(1) → **Invalid** (a BusRdX küldője módosítani fogja az adatot) (2) **FlushOpt** jelzés + tartalom megosztása a buszon **a kérelmező cache számára**
	BusUpgr	→ **Invalid** (a BusUpgr küldője módosította az adatot)
Modified	BusRd	(1) → **Shared** (2) **Flush** jelzés + tartalom megosztása a buszon **a kérelmező cache és a memória számára**
	BusRdX	(1) → **Invalid** (2) **Flush** jelzés + tartalom megosztása a buszon **a kérelmező cache és a memória számára**

4.4. táblázat – A MESI protokoll állapotátmenetei buszoldali jelzések hatására

Nézzünk egy példát a MESI protokoll működésének illusztrációjára. Az 4.5. táblázatban egy négyprocesszoros rendszerben egy kitüntetett cache sor állapotváltozásait követhetjük nyomon a négy cache-ben a különböző processzoroldali műveletek hatására. Félkövér kiemeléssel jelöltük az állapotváltozásokat, az alsó index pedig a kérdéses sor adattartalmát jelzi.

Események	Cache 1	Cache 2	Cache 3	Cache 4
Kezdeti érték	I	I	I	I
Processzor 1 olvassa a sort	**E_1**	I	I	I
Processzor 2 olvassa a sort	**S_1**	**S_1**	I	I
Processzor 3 olvassa a sort	S_1	S_1	**S_1**	I
Processzor 4 írja a sort	I	I	I	**M_2**
Processzor 1 olvassa a sort	**S_2**	I	I	**S_2**

4.5. táblázat – MESI protokoll példa

Copyright © 2021 / BME AUT

Az első beolvasáskor a sor Exclusive állapotba kerül, hiszen csak egy gyorsítótárban található meg. Amikor egy másik (aztán a harmadik) processzor is ugyanazt az adatot olvassa, akkor az értintett cache sor Shared állapotba kerül. A táblázat utolsó előtti sorában az egyik processzor írási műveletet hajt végre, ilyenkor a többi megosztott példány érvénytelenítődik, és a felülírt cache sor kizárólagos lesz, állapota Modified-ra változik. Amikor egy másik processzor újra olvasási kérést ad le ugyanerre a sorra, megkapja a kért adattartalmat attól a cache-től, amelyik tartalmazza, és a sor mindkét cache-ben megosztott állapotba kerül.

4.4 Az Intel Nehalem mikroarchitektúra multiprocesszoros tulajdonságai

Az Intel Nehalem mikroarchitektúrával már találkoztunk a szimultán többszálúság (SMT) vizsgálatakor. Akkor a dinamikus utasítás-csővezeték felépítésére és a Hyper-Threading Technology mikroarchitekturális sajátosságaira fókuszáltunk. Most a memória-architektúráját és a gyorsítótár-szervezését fogjuk megvizsgálni.

A Nehalem mikroarchitektúrát kifejezetten többmagos kialakításra tervezték, és támogatja a kettő chip-ből álló szimmetrikus multiprocesszoros kialakítást is. 2008-ban jelent meg, és erre épültek többek között az első generációs Intel Core-i7, Core-i5 és hasonló processzorok. Azóta sok utódja is megjelent már, de az alapvető tulajdonságok azokban is megmaradtak, inkább a gyártástechnológiából következő mennyiségi fejlődésen mentek keresztül, illetve egyre több, korábban csak külső modulként megvalósítható funkcionális egységet (pl. integrált GPU-t) tartalmaznak. Tehát kijelenthetjük, hogy a Nehalem mikroarchitektúra mind a mai napig meghatározza az elterjedten használt Intel processzorok tulajdonságait.

A korábban említett tulajdonságai mellett memóriaszervezés tekintetében is úttörő volt ez a mikroarchitektúra. Az Intel x86-os családjában elsőként tartalmazott integrált DRAM memóriavezérlőt, valamint egy ettől különválasztott nagysebességű külső interfészt (Quick Path Interconnect). Így már nem igényelte a processzort a memóriával és más nagysebességű perifériákkal összekötő Northbridge chip jelenlétét, amely a korábbi generációkban az ún. Front Side Bus-on keresztül kapcsolódott a processzorhoz.

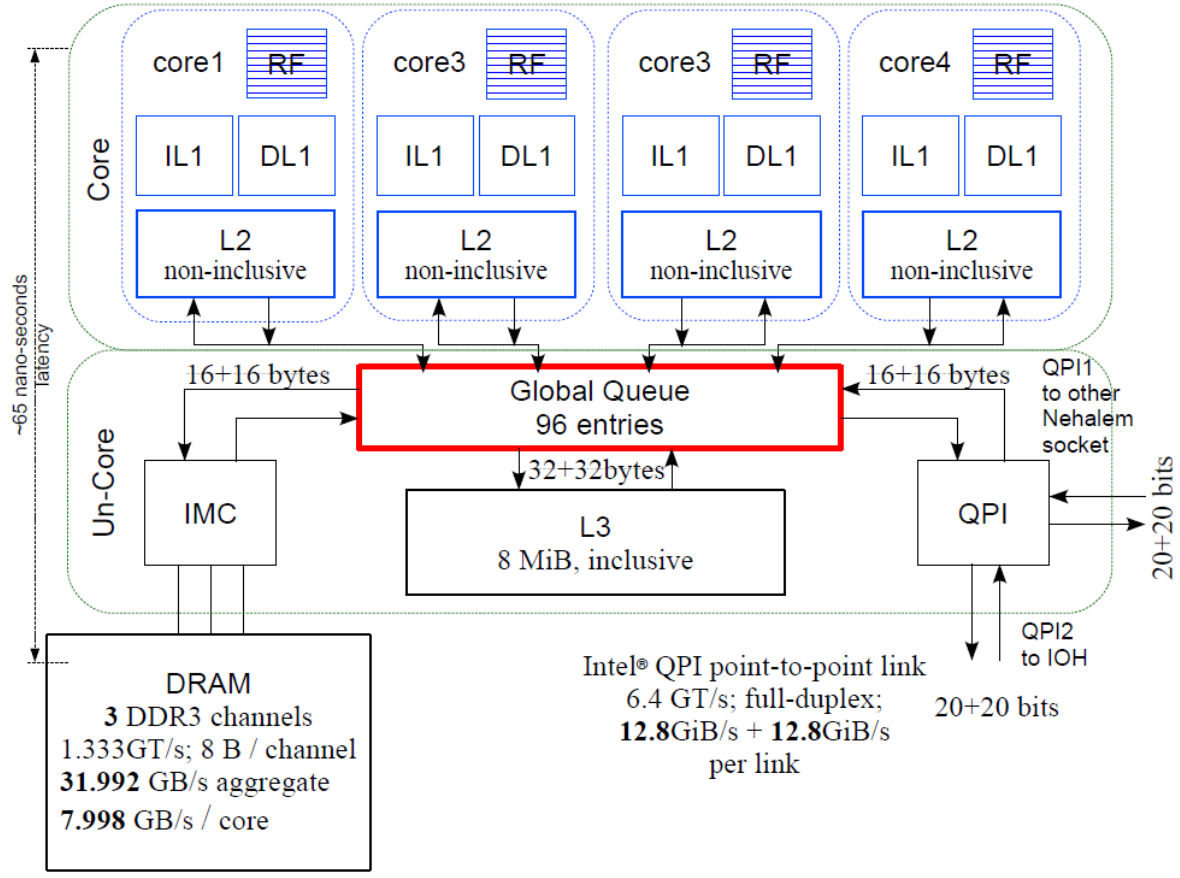
4.4.1 Memóriaszervezés

A mikroarchitektúra legfeljebb négymagos kialakítást tesz lehetővé. A processzormagok, a különböző cache modulok, valamint a memóriainterfész az 4.18. ábrán látható. A processzor háromszintű cache-struktúrával rendelkezik. A cache sorok 64 bájtosak (ez legkisebb mozgatható adategység a cache-ek és a memória között). Minden mag dedikált L1 és L2 cache-ekkel rendelkezik, ezek közül az L1 cache osztott, különálló utasítás (IL1) és adat cache-ekre (DL1) oszlik. Az L1 utasítás cache 32 kB méretű és 4-utas csoportos asszociatív felépítésű. Az L1 adat cache 8-utas csoportos asszociatív, és szintén 32 kB-os. Az L1 cache hozzáférési ideje 4 órajelciklus.

Copyright © 2021 / BME AUT

Az L2 cache már egyesített, tehát utasításokat és adatokat is tárol. 256 kB méretű, 8-utas csoportos asszociatív kialakítású. Non-inclusive tartalmazási szabály szerint működik, tehát nem kell mindent tartalmaznia, ami az L1 cache-ekben is benne van. Az L2 cache hozzáférési ideje 10 órajelciklus.

A még szintén integrált L3 gyorsítótárat a processzormagok közösen használják, ezért az ún. *un-core* tartományban helyezkedik el. 8 MB méretű, 16-utas csoportos asszociatív kialakítású, és inclusive policy szerint működik. Ez a kialakítás a gyorsítótár-koherencia protokoll szempontjából előnyös, mert elég az L3 cache tartalmát szimatolni (akár a chip-en kívülről, a szomszédos többmagos processzorból) ahhoz, hogy kiderüljön, megvan-e egy adat bármelyik gyorsítótárban a chip-en belül. Az L3 cache hozzáférési ideje 35–40+ órajelciklus. Ennek pontos értéke a processzormagok és az eltérő órajellel működő un-core alrendszer órajelfrekvenciáinak arányától függ.



4.18. ábra – A Nehalem mikroarchitektúra memóriaszervezése[49]

Az un-core tartomány a lapkán belül gyakorlatilag a korábbi Northbridge chip funkcionalitását vette át. Amint láttuk, itt található az L3 cache, valamint egy 3-csatornás integrált DDR3

[49] Forrás: Michael E. Thomadakis, The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. Texas A&M University, 2011

Copyright © 2021 / BME AUT

memóriavezérlő (Integrated Memory Controller, IMC) és a QuickPath Interconnect (QPI) interfész[50] vezérlője. Ez utóbbi a különböző perifériákkal való összeköttetést teremti meg az ún. I/O Hub-on (IOH) keresztül, illetve a QPI szolgál a szomszédos processzorral történő kommunikációra is kétprocesszoros kialakítás esetén.

A legfeljebb 4 processzormag és az un-core tartomány közötti adatcserét a Global Queue bonyolítja. Itt dől el, hogy egy adatírási vagy olvasási kérés esetén az L3 cache-hez, vagy az IMC-n keresztül a DRAM memóriához forduljunk-e, esetleg a QPI-n keresztül a szomszédos processzor cache-ében vagy a hozzá illesztett memóriában keressük a kérdéses memóriablokkot. Külön várakozási sorokat tartalmaz a lokális magok írási és olvasási kérései, valamint a QPI-n keresztül érkező külső kérések számára. A Global Queue által megvalósított kapcsolóhálózat crossbar felépítésű.

4.4.2 Cache-koherencia

A Nehalem mikroarchitektúrában a cache-koherencia biztosítására a szimatoló *MESI+F protokollt* alkalmazzák, egy kis könyvtár-alapú kiegészítéssel. Ez a protokoll a chip-en belüli magokhoz tartozó gyorsítótárak közötti és a szomszédos processzor chip-ek közötti koherenciát is biztosítja. Ahogy már említettük, a koherenciaprotokoll chip-ek közötti megvalósítását segíti az L3 cache inkluzív jellege. A könyvtár-alapú kiegészítés azt jelenti, hogy az L3 cache minden sorához tartozik egy 4-bites „valid vector”, amelynek bitjei azt jelzik, hogy az adott sor melyik mag L1/L2 gyorsítótárában található meg.

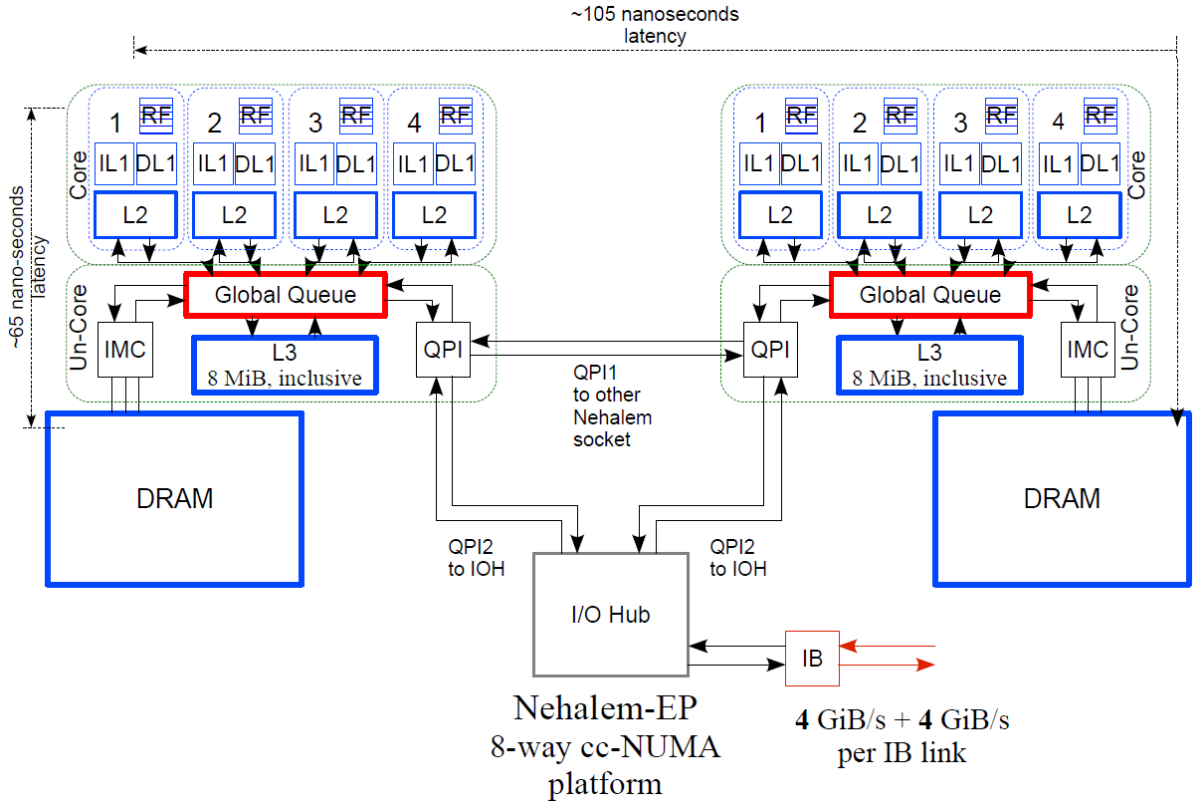
A MESI+F protokoll annyival egészíti ki a már részletesen bemutatott MESI protokollt, hogy bevezet egy új „Forward” állapotot. Ez a „Shared” állapot speciális esete. Annak érdekében, hogy olvasási kérés (BusRd vagy BusRdX jelzés) esetén ne mindegyik cache vezérlő tegye ki a buszra az adatot, amelyik rendelkezik vele, az egyik cache-ben „Forward” állapotba tesszük a blokkot. Így kérés esetén ez a példány van kijelölve arra, hogy a buszra tegye a tartalmát (FlushOpt jelzés kíséretében). A módosított protokoll lényegében ugyanazt a működést biztosítja, de csökkentett buszforgalom mellett.

Cache-Coherent Non-Uniform Memory Access (cc-NUMA)

Az integrált memóriavezérlő alkalmazásának következménye, hogy a DRAM memóriamodulok közvetlenül csatlakoznak a processzor chip-ekhez. Kétprocesszoros elrendezésben ezért az adatok hozzáférési ideje eltér attól függően, hogy a keresett adat a lokális memóriában vagy a másik processzorhoz csatlakoztatott memóriában található meg. Ezért a Nehalem mikroarchitektúrára épülő szimmetrikus multiprocesszoros rendszer memóriamegosztottság szerint NUMA jellegű. Mivel a QPI interfészen keresztül a processzorok között is megvalósul a cache-koherencia, ennek hangsúlyozása érdekében az elrendezést *cc-NUMA* struktúrának is nevezik.

[50] A QPI hivatalos megjelenése előtt az Intel Common System Interface (CSI) néven hivatkozott rá, ezért a szakirodalomban részben ezen a néven szerepel.

Copyright © 2021 / BME AUT



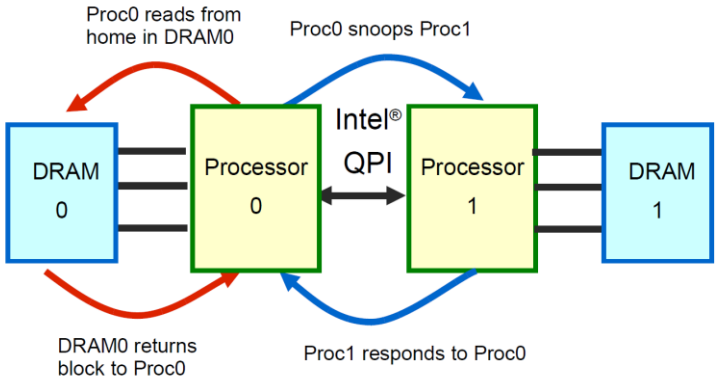
4.19. ábra – A Nehalem mikroarchitektúra cc-NUMA memóriastruktúrája[51]

A kétprocesszoros kialakítás cc-NUMA memóriastruktúráját az 4.19. ábra foglalja össze. Ezen láthatjuk, hogy a lokális memóriák elérési ideje kb. 65 ns, míg a távoli (másik processzorhoz csatlakozó) memória esetén ez az idő már kb. 105 ns, tehát valóban jelentősen eltér a kettő. Az adatok fizikai elhelyezkedése cím alapján ismert, ugyanis a fizikai címtartomány fel van osztva az egyes memóriák között. Minden memóriablokknak a címe alapján megmondható, hogy hol van a „home” lokációja.

A következő két ábra segítségével tekintsük át, hogyan történik egy lokális és egy távoli DRAM-ba tartozó adat elérése a kétprocesszoros cc-NUMA rendszerben. Az 4.20. ábrán látható esetben az egyik processzor egy, a saját lokális memóriájába tartozó adathoz szeretne hozzáférni. Ha mindhárom lokális cache szint hiányt jelez, akkor az integrált memóriavezérlőjén keresztül a lokális DRAM-hoz fordul, hiszen ez az adat „home” lokációja. Ezzel egyidőben a QPI interfészen keresztül a másik processzornak is elküldi a kérést, abban bízva, hogy esetleg az ottani gyorsítótárakban megvan a keresett adat másolata. Mivel az L3 cache inkluzív, találat esetén egy ilyen kívülről jövő kérést nagyon gyorsan ki tud szolgálni, hiszen nem kell az alacsonyabb cache szintekre lemenni az adatért. Abban az esetben, ha megvan az adat a szomszéd processzor L3 cache-ében, a köztes QPI kommunikációval együtt is hamarabb megérkezik onnan, mint a lokális DRAM-ból.

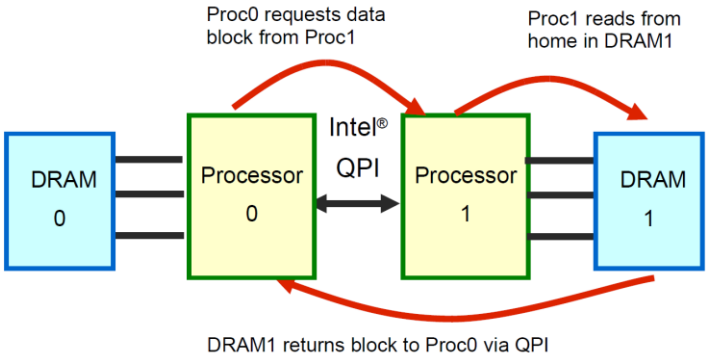
[51] Forrás: Michael E. Thomadakis, The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. Texas A&M University, 2011

Copyright © 2021 / BME AUT



4.20. ábra – Lokális DRAM elérés, távoli szimatolással[52]

Az 4.21. ábra egy olyan esetet mutat be, amikor a keresett adat „home” lokációja a másik processzorhoz tartozó memóriában van. Tegyük fel, hogy az adat nincs jelen a kérelmező processzor belső gyorsítótáraiban. Ilyenkor már a cím alapján kiderül, hogy az adatot nem a lokális DRAM-ban, hanem a szomszédos processzor memóriájában kell keresni, ezért a QPI-n keresztül a kérés továbbításra kerül a másik processzornak. A túloldali Global Queue utánanéz az adatnak az ottani L3 cache-ben, és ha megvan benne, akkor onnan, ha pedig nincs, akkor a DRAM-ból hozza fel a kért adatot, és szolgálja ki a kérést.



4.21. ábra – Távoli DRAM elérés[52]

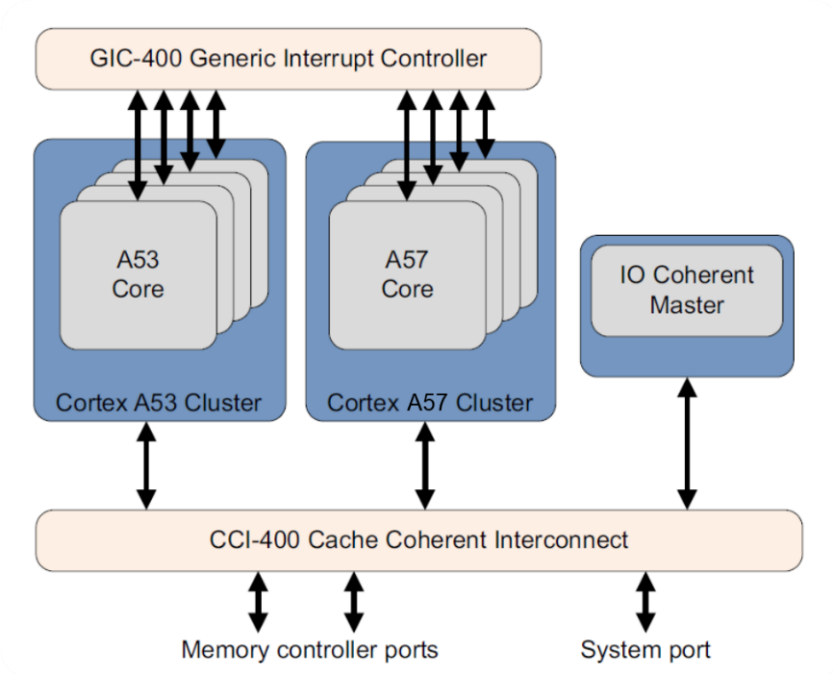
[52] Forrás: Michael E. Thomadakis, The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. Texas A&M University, 2011

Copyright © 2021 / BME AUT

4.5 Többmagos ARM processzorok

Az ARM architektúrák „A” profilú változatai (ARMv7-A, ARMv8-A) jelentős támogatást nyújtanak többmagos kialakítású processzorok létrehozásához. Az ARM Cortex-A processzorcsalád legtöbb tagja kifejezetten többmagos (multi-core) megvalósításra lett tervezve. Egy-egy ilyen processzor 1–4 magot tartalmaz, amelyek egy ún. cluster-t (fürtöt) alkotnak. A clusteren belül a magok teljesen azonos kialakításúak, és az operációs rendszer szabadon ütemezheti őket a különböző feladatok végrehajtására. Kevesebb feladat esetén egyes magokat az energiatakarékosság érdekében le is állíthat.

Gyakoriak az ARM Cortex-A alapú multi-cluster System-on-Chip (SoC) rendszerek is, amelyekben több (tipikusan kettő) többmagos cluster működik együtt, egy operációs rendszer felügyelete alatt. Jellemző az a megvalósítás, hogy eltérő magokat tartalmazó cluster-ek találhatók az SoC-n belül, egy speciális heterogén multiprocesszoros (HMP) rendszert képezve (ld. 4.22. ábra). Az ARM HMP rendszerek sajátossága, hogy az egyes clustereket alkotó processzormagok mikroarchitektúrája lényegesen különbözhet (vannak alacsony fogyasztásra, és vannak nagy teljesítményre specializált magok), de az általuk implementált utasításkészlet-architektúra teljes mértékben azonos. Így az egyes feladatok teljesen szabadon, dinamikusan oszthatók ki a magokra az aktuális teljesítményigénynek megfelelően. Ezt a heterogén kialakítást az ARM big.LITTLE technológiának nevezi.



4.22. ábra – Egy tipikus ARM Cortex-A többmagos processzor felépítése[53]

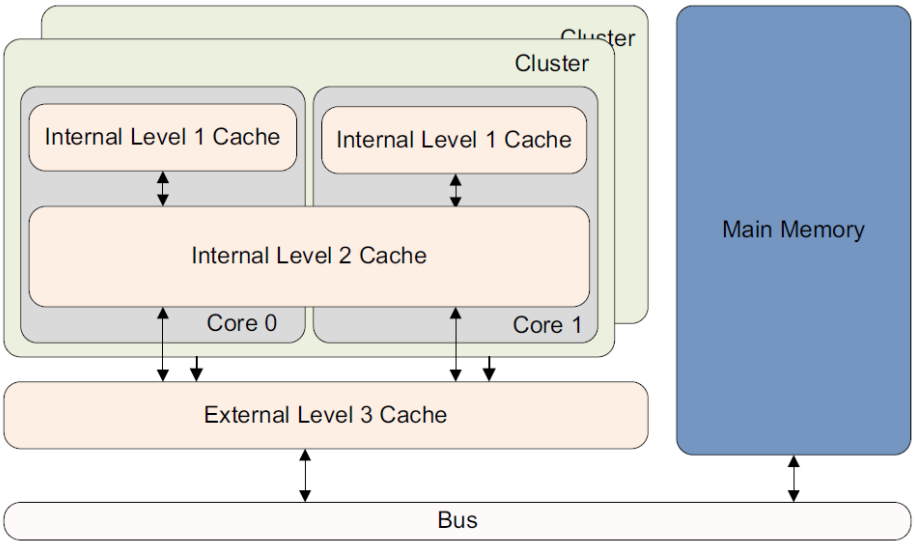
[53] Forrás: ARM Cortex-A Series Programmer’s Guide for ARMv8-A (Version 1.0)

Copyright © 2021 / BME AUT

4.5.1 Cache-koherencia

A többmagos ARM processzorokban a magok között, illetve a több cluster-ből álló HMP rendszerekben a clusterek között is biztosítható a cache-koherencia, hardver támogatással.

Az 4.23. ábrán láthatjuk az ARM Cortex-A processzorok jellemző cache-struktúráját. Minden mag dedikált L1 utasítás- és adatgyorsítótárral rendelkezik (az ábrán ezek összevontan „Internal Level 1 Cache” néven láthatók, de valójában az utasítások és adatok számára külön modulok állnak rendelkezésre). Egy cluster-en belül a magok közös, egyesített L2 cache-t használnak, és a cluster-eken kívül foglal helyet a szintén megosztott L3 cache.



4.23. ábra – Többmagos ARM Cortex-A processzorok cache struktúrája[54]

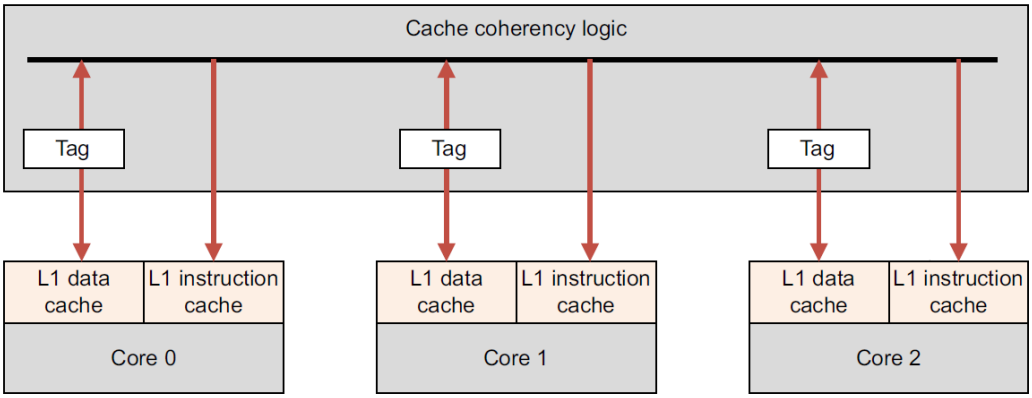
Az egyes clustereken belül az ún. Snoop Control Unit (SCU) felel a gyorsítótár-koherenciáért (ld. 4.24. ábra). Ez az egység az L1 adatgyorsítótárak között biztosítja a koherenciát, ezen kívül pedig az L2 cache hozzáférések arbitrációjáért is felelős. Ez utóbbi miatt az L1 utasítás cache-ekhez is csatlakozik, bár azokra nem terjed ki a koherencia-protokoll hatásköre. Az egyes L1 adat cache-ekben lévő sorokat úgy tartja nyilván, hogy másolatban tartalmazza azok tag-jeit, ami a könyvtár-alapú megoldásokat idézi, bár, mint látni fogjuk, maga a koherencia-protokoll egy, a MESI-hez hasonló szimatoló megoldás.

A clusterek közötti cache-koherencia biztosítása érdekében az SCU-k egy erre kialakított buszon keresztül képesek kommunikálni egymással. Ennek a busznak a tulajdonságait az AMBA 4 ACE specifikáció[55] határozza meg, és egy konkrét megvalósítása az 4.22. ábrán is látható ARM CoreLink CCI-400 Cache Coherent Interconnect. Ezek alapján a cache-

[54] Forrás: ARM Cortex-A Series Programmer’s Guide for ARMv8-A (Version 1.0)
[55] AMBA – Advanced Microcontroller Bus Architecture, ACE – AXI Coherency Extensions, AXI – Advanced eXtensible Interface. Az AMBA specifikációk és az erre épülő különböző buszok (pl. AXI) az SoC-ken belüli komponensek kommunikációját biztosítják. Részletesen tárgyaljuk majd őket a későbbi, egylapkás rendszerekről szóló fejezetekben.

Copyright © 2021 / BME AUT

koherencia megvalósítását úgy jellemezhetjük, hogy clusteren belül könyvtár-jellegű, clusterek között pedig az SCU-k buszkommunikációjára alapozott szimatoló megoldás.



4.24. ábra – Snoop Control Unit[56]

A cache-koherencia logikáját az ún. *MOESI protokoll* határozza meg. Ez a MESI protokoll kibővítését takarja egy további „Owned” állapottal. Ez az állapot olyan módosított cache sort jelöl, amely egyben megosztott is, azaz több példányban is létezik. Az eredeti MESI protokollban minden módosított sort kötelezően vissza kell írni a memóriába, mielőtt megosztásra kerülhetne. Az „Owned” állapot bevezetése lehetővé teszi a „dirty sharing” megvalósítását, aminek segítségével az memóriába való visszaírás elhalasztható. A MOESI protokollban, ha egy cache sor „Modified” állapotban van, és egy másik cache olvasási kérést ad le rá, akkor az írási joggal rendelkező példány „Owned” állapotba kerül, és kötelező közvetlenül kiszolgálnia a másik cache kérését. A másolati példány a másik cache-ben „Shared” állapotú lesz (tehát csak olvasható), és az írási jog továbbra is kizárólagos marad az „Owned” állapotú sort tartalmazó cache számára. Ez a megoldás a visszaírások elhalasztásával sok memóriahozzáférést megspórol.

4.5.2 Az ARM big.LITTLE technológia

Az ARM processzorok népszerűsége, és mobil eszközökben történő széleskörű alkalmazása (hagyományos mobiltelefonok, okostelefonok, tabletek stb.) nagy részben az energiahatékonyságuknak köszönhető. Szinte a többmagos ARM processzorokkal egy időben jelent meg a big.LITTLE technológia, amelynek kialakulását kifejezetten a mobil eszközök speciális igényeinek kiszolgálása motiválta. Az okoseszközök fejlődésével a felhasználói igények között megjelent a nagyteljesítményű alkalmazások futtatásának igénye (pl. videólejátszás, 3D megjelenítés, grafika-intenzív játékok), de természetesen a minél hosszabb akkumulátor-üzemidő iránti igény is megmaradt. Ez a kettő egymásnak erősen ellentmondó feltétel, aminek egyetlen architektúrával megfelelni szinte lehetetlen.

[56] Forrás: ARM Cortex-A Series Programmer’s Guide for ARMv8-A (Version 1.0)

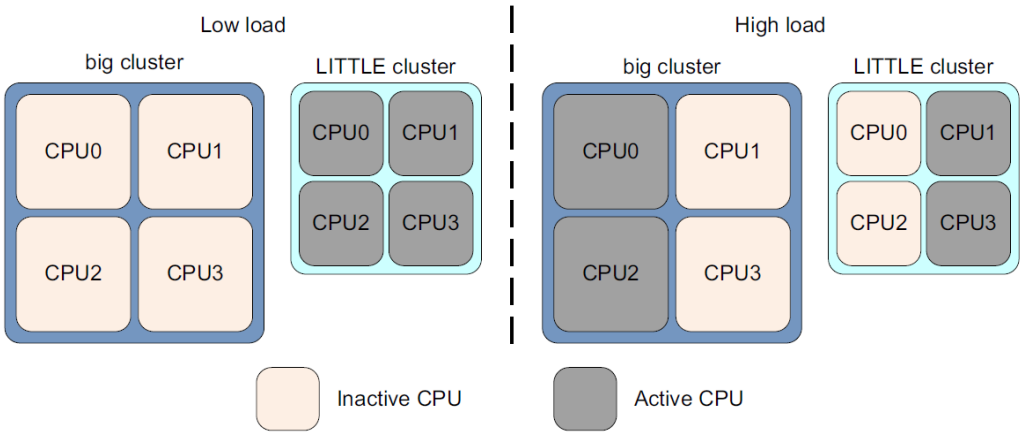
Copyright © 2021 / BME AUT

A big.LITTLE technológia lényege, hogy egyetlen chip-en belül energiahatékony (LITTLE) és nagyteljesítményű (big) magok is megtalálhatók. A magok azonos utasításkészlet-architektúrát valósítanak meg, de eltérő működési paramétereiket különböző mikroarchitektúrájuknak köszönhetik. Például megfigyelhető, hogy a LITTLE magok között szinte kizárólag sorrendi végrehajtásúakat találunk (pl. Cortex-A7, A32, A35, A53), a big magok pedig általában támogatják a sorrenden kívüli végrehajtást (pl. Cortex-A15, A17, A57, A72, A77). A magok menet közben a terhelés függvényében ki- és bekapcsolhatók, és az azonos ISA miatt a programok akár futás közben is migrálhatók a különböző magok között.

A big feladatok tipikus példái a grafika-intenzív alkalmazások, a videólejátszás, vagy a weblapok renderelése. LITTLE feladatokra példa a zenelejátszás, a weblapok böngészése (le-fel görgetés a betöltés után), illetve az operációs rendszer háttérfolyamatai. Egy tipikus alkalmazásnál a rendszer az idő nagy részében a LITTLE magokon fut, és a big magok csak rövidebb időszakokra, a nagy terhelést jelentő feladatoknál aktiválódnak.

4.5.2.1 ARM big.LITTLE végrehajtási modellek

A big.LITTLE technológia többféle végrehajtási modellt definiál. Ezek közül a legegyszerűbb a **cluster migráció**, ami azt írja elő, hogy egyszerre csak az egyik cluster lehet aktív. Alacsonyabb terhelés esetén minden feladat a LITTLE magokon fut, nagyobb terhelésnél pedig az összes futó programszál átköltözik a big magokra.



4.25. ábra – CPU migráció a big.LITTLE architektúrában[57]

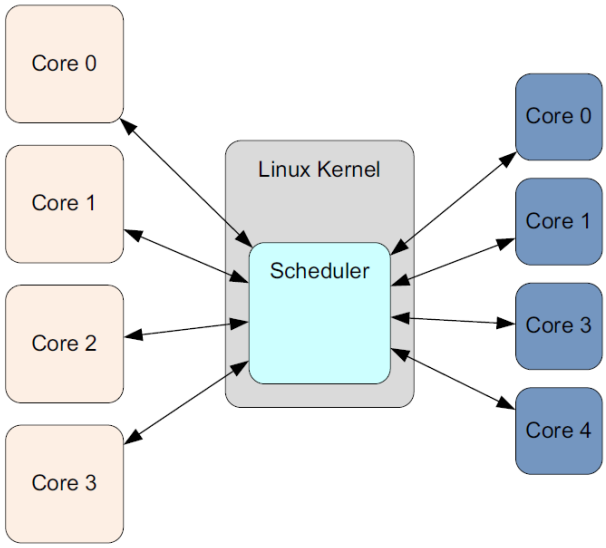
Ennél rugalmasabb megoldást jelent a **CPU migráció** (ld. 4.25. ábra). Ez a modell párokat képez a processzormagokból úgy, hogy minden pár egy big és egy LITTLE magból áll. A működési modell azt írja elő, hogy páronként mindig csak az egyik mag lehet aktív. Ez azért rugalmasabb a cluster migrációnál, mert ha csak egy-két teljesítményigényes programszál van, akkor elég csak egy-két processzormag-párnál végrehajtani a migrációt a big magra, a többinél

[57] Forrás: ARM Cortex-A Series Programmer's Guide for ARMv8-A (Version 1.0)

Copyright © 2021 / BME AUT

továbbra is a LITTLE magok futhatnak tovább. A cluster és a CPU migrációs modellnek is sajátossága, hogy a 8-magos processzorból egyidejűleg mindig csak 4 magot tudunk használni.

A legrugalmasabb működési modell a **Global Task Scheduling** (ld. 4.26. ábra), amely nem csoportosítja vagy párosítja a magokat, hanem az operációs rendszer számára mindegyik függetlenül hozzáférhető. Ilyenkor az operációs rendszer határozza meg a dinamikus feladat-hozzárendelést, és így akár mind a 8 magot egyszerre használhatja, ha a megnövekedett terhelés miatt erre van szükség.



4.26. ábra – Global Task Scheduling[58]

4.5.2.2 ARM DynamIQ

Az ARM 2017-ben bejelentette a big.LITTLE utódjaként a DynamIQ technológiát, ami kibővíti és még rugalmasabbá teszi a többmagos lehetőségeket, és még hatékonyabb energiafelhasználást biztosít. Az új technológiában egy cluster-en belül lehetővé válik akár 8 heterogén mag jelenléte, tehát tetszőleges konfigurációban big és LITTLE magok is szerepelhetnek egy cluster-ben (pl. 1 db big és 7 db LITTLE vagy 2 db big és 4 db LITTLE stb.). Minden mag saját dedikált L2 cache-sel rendelkezik, és a cluster-en belül helyezkedik el a megosztott L3 cache. Ez a korábbinál szorosabb kapcsolatot jelent a különböző kialakítású magok között, ami a feladatok futás közbeni migrációját egyszerűbbé és kevesebb buszforgalommal járóvá teszi. Az energiahatékonyság további növelése érdekében az egyes processzormagoknak külön-külön lehet állítani az órajelfrekvenciáját a terhelés függvényében. A vegyes kialakítású cluster-ek a termékfejlesztést is segítik, hiszen a korábbinál differenciáltabb képességű SoC-k összeállítására adnak lehetőséget.

[58] Forrás: ARM Cortex-A Series Programmer's Guide for ARMv8-A (Version 1.0)

Copyright © 2021 / BME AUT

Felhasznált és ajánlott irodalom

[69] Nicholas Carter. Schaum's Outline of Computer Architecture. McGraw-Hill, 2002. ISBN 0-07-136207-X

[70] Wikipedia. Multiprocessing. https://en.wikipedia.org/wiki/Multiprocessing (Hozzáférés: 2021.03.12)

[71] Wikipedia. Multi-core processor. https://en.wikipedia.org/wiki/Multi-core_processor (Hozzáférés: 2021.03.12)

[72] Wikipedia. Uniform memory access. https://en.wikipedia.org/wiki/Uniform_memory_access (Hozzáférés: 2021.03.12)

[73] Wikipedia. Non-uniform memory access. https://en.wikipedia.org/wiki/Non-uniform_memory_access (Hozzáférés: 2021.03.12)

[74] Wikipedia. Crossbar switch. https://en.wikipedia.org/wiki/Crossbar_switch (Hozzáférés: 2021.03.12)

[75] Wikipedia. Multistage interconnection networks. https://en.wikipedia.org/wiki/Multistage_interconnection_networks (Hozzáférés: 2021.03.12)

[76] Wikipedia. Symmetric multiprocessing. https://en.wikipedia.org/wiki/Symmetric_multiprocessing (Hozzáférés: 2021.03.12)

[77] Wikipedia. Asymmetric multiprocessing. https://en.wikipedia.org/wiki/Asymmetric_multiprocessing (Hozzáférés: 2021.03.12)

[78] Wikipedia. Heterogeneous computing. https://en.wikipedia.org/wiki/Heterogeneous_computing (Hozzáférés: 2021.03.12)

[79] Wikipedia. Cache (computing). https://en.wikipedia.org/wiki/Cache_(computing) (Hozzáférés: 2021.03.12)

[80] Wikipedia. CPU cache. https://en.wikipedia.org/wiki/CPU_cache (Hozzáférés: 2021.03.12)

[81] Wikipedia. Cache inclusion policy. https://en.wikipedia.org/wiki/Cache_inclusion_policy (Hozzáférés: 2021.03.12)

[82] Wikipedia. Cache coherence. https://en.wikipedia.org/wiki/Cache_coherence (Hozzáférés: 2021.03.12)

[83] Wikipedia. Directory-based cache coherence. https://en.wikipedia.org/wiki/Directory-based_cache_coherence (Hozzáférés: 2021.03.12)

[84] Wikipedia. Bus snooping. https://en.wikipedia.org/wiki/Bus_snooping (Hozzáférés: 2021.03.12)

[85] Wikipedia. MESI protocol. https://en.wikipedia.org/wiki/MESI_protocol (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

[86] Wikipedia. MESIF protocol. https://en.wikipedia.org/wiki/MESIF_protocol (Hozzáférés: 2021.03.12)

[87] Wikipedia. MOESI protocol. https://en.wikipedia.org/wiki/MOESI_protocol (Hozzáférés: 2021.03.12)

[88] ARM Cortex-A Series Programmer's Guide (Version 4.0) https://developer.arm.com/documentation/den0013/d/ (Hozzáférés: 2021.03.12)

[89] ARM Cortex-A Series Programmer's Guide for ARMv8-A (Version 1.0) https://developer.arm.com/docs/den0024/a (Hozzáférés: 2021.03.12)

[90] Michael E. Thomadakis, The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. Texas A&M University, 2011 https://www.researchgate.net/publication/235960679_The_Architecture_of_the_Nehalem_Processor_and_Nehalem-EP_SMP_Platforms (Hozzáférés: 2021.03.12)

[91] David Kanter. Inside Nehalem: Intel's Future Processor and System. https://www.realworldtech.com/nehalem/ (Hozzáférés: 2021.03.12)

[92] David Kanter. The Common System Interface: Intel's Future Interconnect. https://www.realworldtech.com/common-system-interface/ (Hozzáférés: 2021.03.12)

[93] Wikipedia. ARM big.LITTLE. https://en.wikipedia.org/wiki/ARM_big.LITTLE (Hozzáférés: 2021.03.12)

[94] Wikipedia. Comparison of ARMv7-A cores. https://en.wikipedia.org/wiki/Comparison_of_ARMv7-A_cores (Hozzáférés: 2021.03.12)

[95] Wikipedia. Comparison of ARMv8-A cores. https://en.wikipedia.org/wiki/Comparison_of_ARMv8-A_cores (Hozzáférés: 2021.03.12)

[96] ARM Technologies. DynamIQ. https://www.arm.com/why-arm/technologies/dynamiq (Hozzáférés: 2021.03.12)

[97] Govind Wathan. Where does big.LITTLE fit in the world of DynamIQ? https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/where-does-big-little-fit-in-the-world-of-dynamiq (Hozzáférés: 2021.03.12)

Copyright © 2021 / BME AUT

Nagyteljesítményű mikrokontrollerek és interfészek

Elektronikus jegyzet

I. rész

Nagyteljesítményű mikrokontrollerek

5. fejezet: Grafikus processzorok

Tárgyfelelős: **Dr. Tevesz Gábor**
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117 Budapest, Magyar tudósok körútja 2.
Q ép. B szárny II. em. B216.
Tel: 463-2881
E-mail: tevesz@aut.bme.hu

Szerző(k): **Kovács Viktor**
BME Automatizálási és Alkalmazott Informatikai Tanszék

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki mesterszak (MSc)
Számítógép alapú rendszerek specializáció hallgatói

Utolsó módosítás: 2021. 03. 23.

Verzió: 1.5

Copyright © 2021 / BME AUT

COPYRIGHT

Jelen dokumentum a BME Villamosmérnöki és Informatikai Kar hallgatói számára készített elektronikus jegyzet. A dokumentumot a Nagyteljesítményű mikrokontrollerek és interfészek c. tantárgyat (BMEVIAUMA07) felvevő hallgatók jogosultak használni, és saját céljukra 1 példányban kinyomtatni. A dokumentum módosítása, bármilyen eljárással részben vagy egészben történő másolása tilos, illetve csak a szerző(k) előzetes engedélyével történhet.

Copyright © 2021 / BME AUT

Tartalom

5 Grafikus processzorok 4

5.1 Történet, kialakulás 4
5.2 A CPU és GPU kapcsolata 7
5.3 3D grafika 11
5.3.1 Sugárkövetés 11
5.3.2 Raszterizálás 12
5.3.3 Shader-ek 16
5.3.4 GPGPU 17
5.4 GPU memória 17
5.5 GPU architektúrák 20
5.5.1 nVidia G80, GeForce 8 (Tesla) 23
5.5.2 nVidia GF100 Fermi 26
5.5.3 AMD Terascale VLIW architektúra 26
5.5.4 AMD GCN 1.0 (2012) 28
5.5.5 AMD GCN variánsok 41
5.5.6 AMD RDNA (Navi) 42
5.5.7 GeForce RTX 2080 Turing 43
5.5.8 AMD RDNA 2 45
5.5.9 Cell processzor 45
5.5.10 Intel Xeon Phi 46
5.6 Szoftver 47
5.6.1 Vertex és pixel shaderek 48
5.6.2 OpenCL 52
5.6.3 CUDA 56
5.6.4 Esettanulmány, példák 57
5.7 Neurális hálók 61
Felhasznált és ajánlott irodalom 66

Copyright © 2021 / BME AUT

5 Grafikus processzorok

A fejezetben ismertetjük a grafikus processzorok kialakulásának rövid történetét, bemutatunk néhány architektúrát, különös tekintettel arra, hogyan fejlődtek fix funkciós eszközökből általános célú számításra is alkalmas megoldásokká. Az AMD GCN architektúráját részletesebben is megvizsgáljuk. Kitérünk a GPU-k memóriájára, valamint a csatolófelületekre. A hardver mellett áttekintjük a szoftveres illesztőket, API-kat, továbbá röviden bemutatunk két további alkalmazási területet, melyek a korszerű GPU-k előtt gyakorlati szempontból szinte megvalósíthatatlannak tűntek: valósidejű sugárkövetés és nagyméretű neurális hálók kezelése.

5.1 Történet, kialakulás

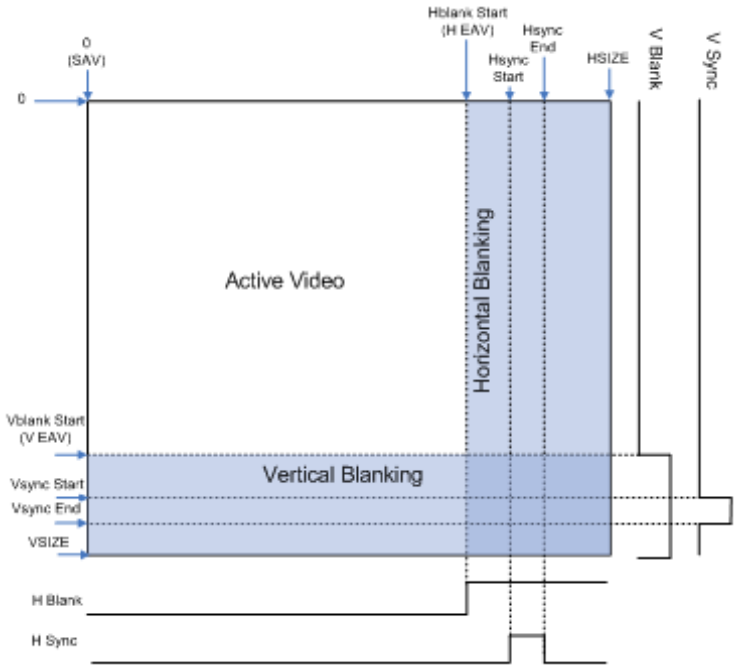
A grafikus processzorokat (GPU, Graphics Processing Unit) elsősorban grafikai feladatokhoz kapcsolódó számítási feladatok kiszolgálására fejlesztették ki, melyek eltérően az általános célú processzoroktól, kevesebb–féle, de nagyobb számú és párhuzamosan végrehajtható művelet elvégzésére képesek.

A megjelenítők illesztését és kiszolgálását végző egységek korán leváltak az általános célú processzorokról, hiszen időkritikus feladat a kijelzők folyamatos ellátása adatokkal, továbbá a megjelenítésért felelős egységek relatív nagy, és a központi tártól függetleníthető dedikált memóriaigénnyel rendelkeznek, gondoljunk itt az egyre növekvő felbontásra és színmélységre. A korai és legegyszerűbb illesztők elsődleges feladata az volt, hogy interfészt biztosítsanak az általános célú processzor felől jövő adatoknak, szabványos videojelet hozzanak létre DA átalakítóval (ld. kompozit videó, VGA videojel), valamint továbbítsák azt a megjelenítőeszköz (monitor) felé.

A 5.1 ábrán látható a képátvitel során használt jelek és a funckióik. A kétdimenziós kép sorokra történő felbontásával alakítható egydimenziós, időben változó jellé. Az átvitel során nem csak a látható képet továbbítják a rendszerek, a nem látható terület használatának történelmi és gyakorlati okai is vannak. A katódsugárcsöves megjelenítők esetén egy elektronsugár bombázza a foszfortartalmú képernyőt, mely a becsapódó elektronok hatására látható fényt bocsát ki. Az elektronsugár gyors mozgatásával és intenzitásának változtatásával jeleníthető meg a képernyőn képi információ. A sorok végéről a sorok elejére történő ugrás csak véges idő alatt hajtható végre, ezért a sorok között „sor/képkioltási” időre van szükség (a mágneses tér megváltoztatásához), ekkor az elektronsugarat is szüneteltetni kell. Az aktív és nem aktív tartományt megadó jel lehet ponált (line valid) vagy negált is (blanking). A szinkronizációs jel teszi lehetővé, hogy az oszlopok ne „csússzanak” el, minden sort helyes pozícióban lehessen kirajzolni. Hasonlóan működnek a vertikális vezérlőjelek is. Az elektronsugárt a kép kirajzolásának befejeztével vissza kell mozgatni a jobb alsó sarokból a bal felső sarokba, addig az rajzolást szüneteltetni kell. A nem aktív tartomány használata a katódsugárcsöves megjelenítők után is megmaradt, a sorok és képek közötti idő lehetővé teszi egyszerűbb

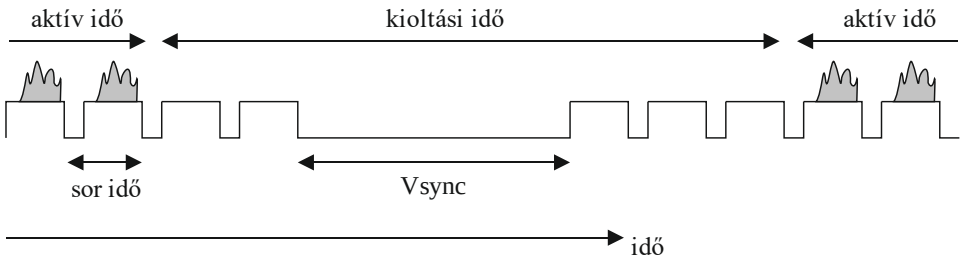
Copyright © 2021 / BME AUT

feldolgozási lépések végrehajtását, szinkronizálási vagy egyéb jel kezelési feladatok megvalósítását: ilyen lehet pl. a memória vezérlésével járó extra idő.



5.1. ábra – Tipikus koncepció videojelek esetén: aktív képtartomány közben intenzitásjelek, nem aktívban szinkronizáció és idő a kijelző működéséhez[1]

A 5.2 ábrán látható a kép egydimenziós jellé alakítva. A szinkronizáló jelek lehetnek magába a jelbe is integrálva, a kompozit videoszabvány egyetlen vezetéken tette lehetővé színes videojelfolyam átvitelét, analóg módon. A rövid alacsony szintű jelek a sorok szinkronizálására szolgáltak és a sorkioltási időt határozták meg, a hosszú alacsony szintű szünet az első sor szinkronizálására és a képkioltási idő meghatározására szolgált.

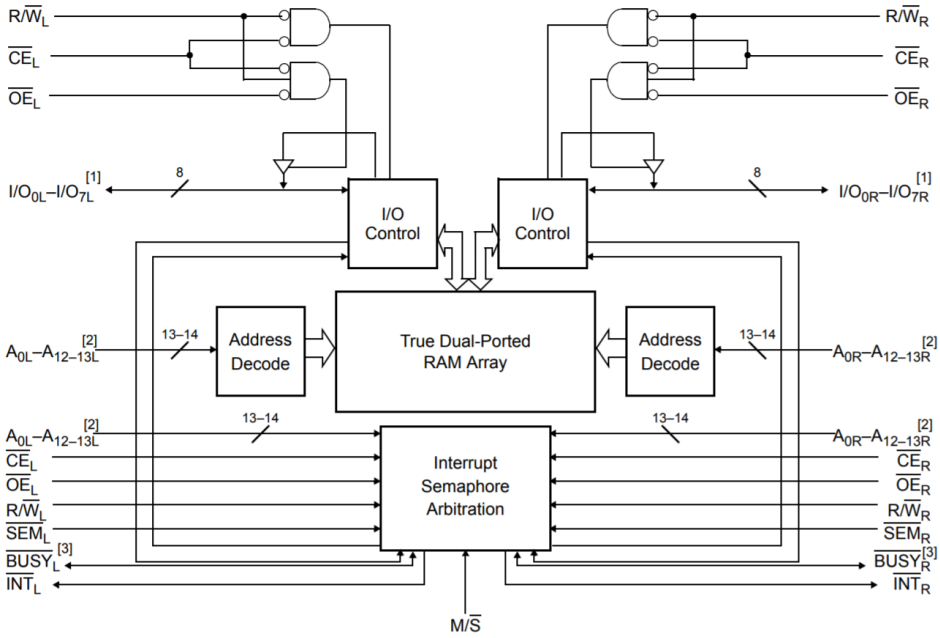


5.2. ábra – Soronként átvitt videojel, jelbe ágyazott horizontális és vertikális szinkronizáció, a szinkronizáció dedikált vezetékeken is átvihető

[1] Forrás: Video Timing Controller v6.1 LogiCORE IP Product Guide, Xilinx, https://www.xilinx.com/support/documentation/ip_documentation/v_tc/v6_1/pg016_v_tc.pdf (Hozzáférés 2020.06.29.)

Copyright © 2021 / BME AUT

Legegyszerűbb esetben ez az interfész egy „framebuffer” képmemória, melyet a processzor ír és a video illesztő olvas. Az adapter ilyenkor a host által generált bitképet jeleníti meg. Ilyen volt a 80-as évek elején a CGA (Color Graphics Adapter) kártya.

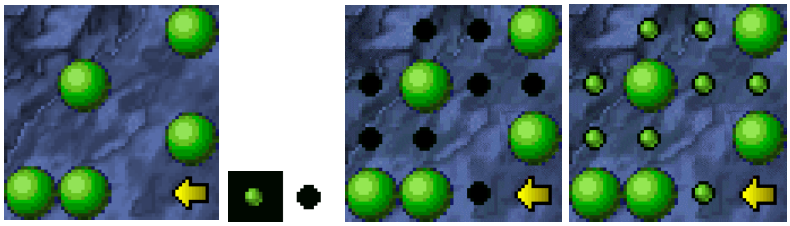


5.3. ábra – Kétportos memória: processzor írja, grafikus illesztő olvassa[2]

A fent vázolt esetben a memóriába történő rajzolás továbbra is a processzor feladata, a megjelenített ábrák leggyakrabban grafikus elemekből (vonal, téglalap stb.), betűkből, vagy akár kisebb képekből állnak. A képek memóriába történő rajzolását gyorsították a következő generációs illesztők, melyekkel a memóriamásolás mellett már egyszerű bitműveletek is elvégezhetők voltak. Ilyen művelet a Bitblt (Bit-Block Transfer), mely gyorsan képes memóriát másolni, figyelembe véve a cél és a rajzolandó kép méretét, miközben akár logikai műveletek is elvégezhetők a képpontokon. Nem téglalap alakú rajzolás is megvalósítható, ha a képen egy kitüntetett színt átlátszónak tekint a rendszer, vagy meghatározható egy külön maszk (átlátszóság) kép is logikai műveletekkel támogatva.

[2] Forrás: Cypress CY7C144AV datasheet (Hozzáférés 2020.06.29.)

Copyright © 2021 / BME AUT



5.4. ábra – Bitblt: Forrás kép, kirajzolandó kis kép maszkkal, AND művelettel a cél helyek törlése, OR művelettel a cél helyekre beírás[3]

A 2D grafikát támogató adapterek később további funkciókkal egészültek ki: 2D primitívek (vonalak, ellipszisek, poligonok) rajzolása, területek kitöltése, Bitblt hardveres gyorsítása, így további teljesítményt szabadítottak fel a központi processzorban. Az adapterek első ilyen képviselője az IBM 8514, 1987-ből. Az 1990-es évek elején a Windowsban megjelenik a GDI interfész, mely szoftveres oldalról támogatta és biztosította a rajzolási műveletek gyorsítását. 1991-ben jelent meg az S3 86C911-es IC, mely egychipes kialakításban biztosított a 2D grafikai gyorsítást. Az évtized közepére ezek a fix funkciós eszközök elterjedtek, kiszorították az általános célú társprocesszorokat.

A 3D grafikát támogató videoadapterek iránt a 90-es évek közepén jelent meg az igény. Kezdetben különálló, csak 3D támogatású kártyák is megjelentek, kiegészítve a már elérhető 2D-s videoadaptereket. Később ezeket a funkciókat egy IC-be integrálták. A 3D képességek kihasználásához szoftveres interfész is szükséges, ebből több versengő megoldás is megjelent (Glide, OpenGL, DirectX), végül a hardver/gyártó függő megoldások eltűntek, csak az univerzális API-k maradtak meg. A hardverek és az API-k folyamatosan fejlődtek az évek során, a képességeik egyre bővültek (transzformációk, fények, tesszelláció, programozható shaderek, fizikai szimuláció, sugárkövetés, gépi tanulás stb). A grafikai és általános célú párhuzamosított számítások gyorsítása ma már olyan alapkövetelmény, hogy a gyorsítók asztali, mobil és alkalmazásprocesszorok szerves részévé váltak. A 3D grafikán túl a 2D és egyéb multimédia képességek is fejlődtek, immáron integráltan tartalmaznak video kódolást és dekódolást gyorsító egységeket, illetve DSP-t hangfeldolgozás céljából.

5.2 A CPU és GPU kapcsolata

A videoadapterek sokáig külön bővítőkártyaként voltak elérhetők, kezdetben ISA, PCI bővítőhelyeken, majd később kifejlesztettek egy videokártya illesztéséhez dedikált csatolót: az AGP-t (Accelerated Graphics Port). Az AGP csatoló a PCI továbbfejlesztéseként jött létre, azonban nagyobb sávszélességet, valamint a dedikált hozzáférés miatt kisebb késleltetést tett lehetővé. A csatolónak több generációja is készült, a sebesség folyamatos növelése érdekében fokozatosan csökkentették a jelzési feszültségszinteket, növelték a tranzakciók hosszát, órajelenként egyre több adatot vittek át (2x, 4x, 8x mód), illetve dedikált címvezetékeket is bevezettek.

[3] Forrás: https://en.wikipedia.org/wiki/Bit_blit (Hozzáférés 2020.06.29.)

Copyright © 2021 / BME AUT

A fotorealisztikus 3D grafika iránti igény, valamint az egyre nagyobb számítási igény kiszolgálása növelte a magas sávszélesség iránti igényt. Egyre több adatot kellett másolni a központi memória és a GPU által elérhető dedikált memória között: 3D grafika esetén az egyre részletesebb modellek, valamint nagyobb felbontású textúrák igényelték a növekvő sávszélességet és memóriaméretet.

A párhuzamos AGP csatolót végül leváltotta a soros átvitelű, általános célú PCI-Express, mely kezdetben sávonként 250MB/s (B: bájt, b: bit) átviteli sebességet támogatott, és a sávok számát rugalmasan kezelhetővé tette: 1x, 2x, 4x, 8x és 16x kiépítések a leggyakoribbak, a szélesebb alaplapi fizikai csatoló kompatibilis a keskenyebb bővítőkártyákkal. Az átvitel során 8b/10b kódolást alkalmaznak, azaz 10 átvitt bit határoz meg egy átvitt bájtot, a jelzési sebesség megadásánál nem a Gbps hanem a GT/s (gigatranszfer) terjedt el, mint jelzési sebesség. Már az első generáció 16 kiépített sáv esetén közel 4GB/s-ot eredményezett, ami meghaladta az addigi AGP képességeit. Az egyes PCI-Express generációk szinte rendre duplázták az elérhető átviteli sebességet (ld. 5.1. táblázat), azonban fontos kiemelni, hogy ez a sebesség még mindig eltörpül a külső memóriák által biztosított sávszélességhez képest. A nagy átviteli sebesség mellett előnye az egységes csatolófelület egyéb bővítőkártyákkal, valamint lehetővé vált akár több grafikus adapter használata egy rendszeren belül, amellyel tovább lehetett fokozni a felhasználható számítási kapacitást. A PCI-Express host a processzorba van integrálva, de a processzor melletti chipkészlet is biztosít akár újabb sávokat is bizonyos perifériákhoz, mint a SATA, NVMe, hang, hálózati vagy USB csatolók. Az Intel processzorai egy PCI-Express-hez hasonló, de zárt szabványú interfésszel kommunikál a chipkészlettel (DMI 3.0), míg az AMD szabványos PCI-Express vonalakat használ. A processzor PCI-Express vonalaira csatlakoznak a legnagyobb sávszélességű eszközök, mint az integrált vagy dedikált GPU, bizonyos esetekben hálózati csatoló vagy gyors tárolóeszközök. A PCI-Expresst később, az interfészek fejezetben részletesebben tárgyaljuk.

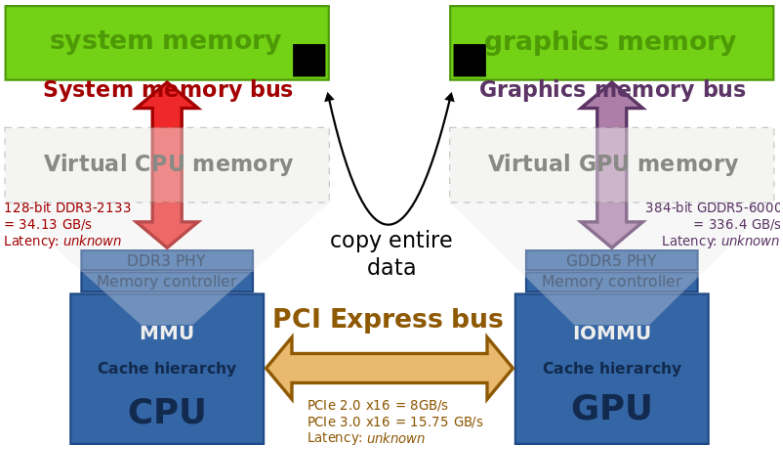
Csatoló	Megjelenés	Jelzési sebesség	Átviteli sebesség
PCI	1992	33MHz 32 bit	133MB/s
PCI 2.1	1995	66MHz 32 bit	266MB/s
AGP 1.0 (1x/2x)	1997	66MHz 32 bit	266MB/s vagy 533MB/s
AGP 2.0 (4x)	1998	66MHz 32 bit	1066MB/s, 4x transfer/clock
AGP 3.0 (8x)	2000	66MHz 32 bit	2133MB/s, 8x transfer/clock
PCI-Express 1.0	2004	2.5GT/s (8b/10b)	1x: 250MB/s 16x: 4GB/s
PCI-Express 2.0	2007	5GT/s (8b/10b)	1x: 500MB/s 16x: 8GB/s
PCI-Express 3.0	2010	8GT/s (128b/130b)	1x: 984.6MB/s 16x: 15.75GB/s
PCI-Express 4.0	2017-2019	16GT/s (128b/130b)	1x: 1968MB/s 16x: 31.51GB/s

Copyright © 2021 / BME AUT

PCI-Express 5.0	2019	32GT/s (128b/130b)	1x: 3938MB/s 16x: 63.02GB/s
(PCI-Express 6.0)	(2021)	64GT/s (128b/130b)	1x: 7877MB/s 16x: 126.03GB/s

5.1. táblázat – Bővítőkártya csatolók összehasonlítása

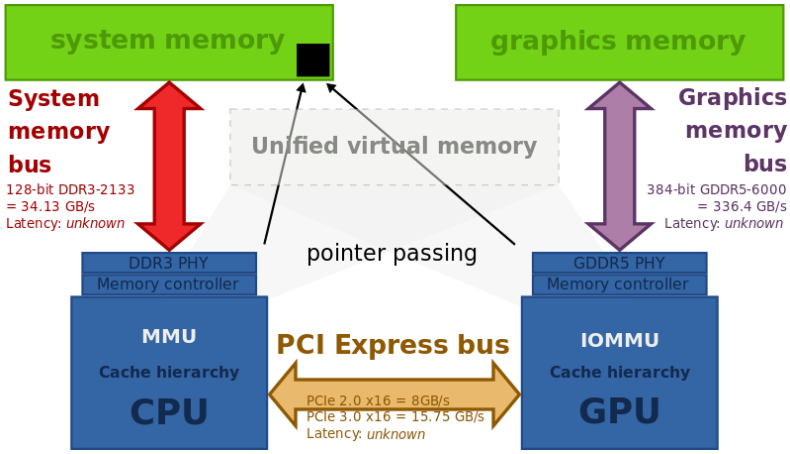
A kis méretre és a nagyfokú integráltságra törekvés miatt kisteljesítményű GPU-kat integráltak szorosan a processzorok mellé. A mobil és beágyazott rendszerek mellett ez a tendencia a PC-ket sem kerülte el. A dedikálthoz képest az integrált megoldásoknak fogyasztási keretük és fizikai méretük miatt jelentősen kisebb a számítási teljesítményük. Általában saját dedikált RAM-mal sem rendelkeznek, a CPU-val közös központi memóriát használják, melynek sebessége alacsonyabb és gyakran a beállítható mérete is kisebb a dedikált video RAM-hoz (VRAM) képest. Több gyártó együttesen megalkotta a HSA platform (Heterogeneous System Architecture) specifikációt, melynek célja szorosan összekapcsolni a CPU-t és GPU-t (és egyéb koprocesszort) alacsony késleltetésű kommunikációs megoldással, elkerülve a felesleges memória-másolásokat, valamint egységes felületet, egyesített virtuális memóriatartományt biztosítani az alkalmazások számára.



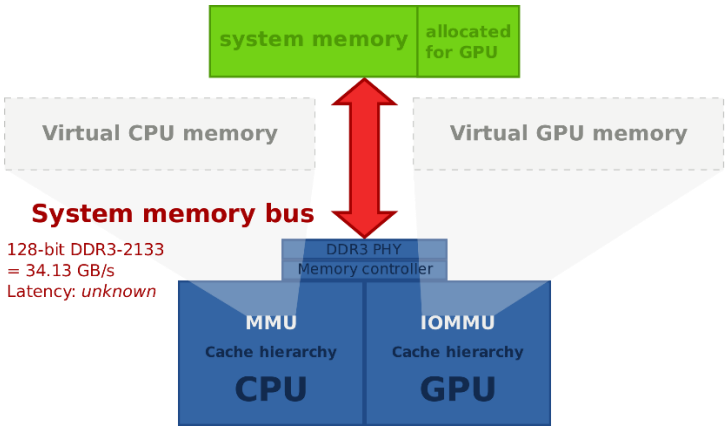
5.5. ábra – Hagyományos (nem HSA) architektúra különálló CPU és GPU esetén. A két egység dedikált memóriával rendelkezik, másolás szükséges [4]

[4] Forrás: https://en.wikipedia.org/wiki/Heterogeneous_System_Architecture (Hozzáférés 2020.06.29.)

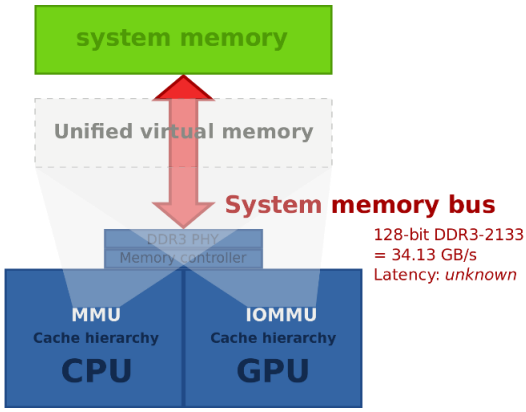
Copyright © 2021 / BME AUT



5.6. ábra – HSA-kompatibilis architektúra, a mutatók egységes címtartományban működnek, nem szükséges teljes pufferek átmásolása



5.7. ábra – Hagyományos integrált GPU-s architektúra. A GPU számára elkülönített memóriaterület áll rendelkezésre a központi tárból, a kettő között memóriamásolás szükséges



5.8. ábra – HSA kompatibilis megoldás, a GPU és a CPU közös memóriát használ, elkerülhető a felesleges másolás

A felhasználói alkalmazások jellemzően nem érik el közvetlenül a GPU-t (kivéve feladatspecifikus esetben, pl. játékkonzolok esetén), hanem szabványos API-kon keresztül

Copyright © 2021 / BME AUT

kommunikálnak az eszközmeghajtóval, mely a parancsokat az adott GPU típusnak megfelelő módon továbbítja. A kommunikáció jellemzően parancs–puffereken keresztül történik, melyek segítenek kiegyenlíteni a folyamatosan változó terhelést mind a CPU, mind a GPU oldalán. Hagyományosan egyetlen grafikai parancspuffer állt rendelkezésre, a GPU sorban hajtotta végre az itt érkező parancsokat (pl. objektum rajzolás). Később megjelentek a másolási (copy) és számítási feladatokat leíró compute parancssorok. A hatékonyság és kihasználtság rovására ment a soros végrehajtás, mert a feladatoktól függően eltérő mértékben sikerült kihasználni a rendelkezésre álló erőforrásokat (számítási egységeket, memória hozzáférést stb). A parancssorokat a CPU szoftvere töltötte, azonban nem volt járható út minden GPU architektúrához és kiépítéshez optimalizálni a szoftvereket, hogy milyen ütemezéssel adjon ki GPU parancsokat. Az ütemező később a hardverbe került, valamint preemptív képességekkel is kiegészítették (pl. async compute), így sokkal hatékonyabbá vált a rendelkezésre álló erőforrások (compute shaderek, grafikához raszterizáló és textúrázó egységek, másoláshoz DMA) kihasználása.

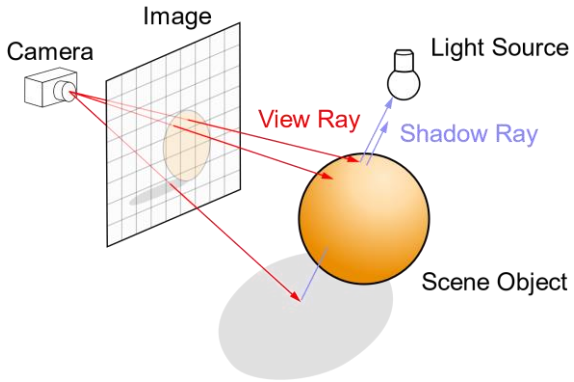
5.3 3D grafika

3D alapú fotorealisztikus grafikák előállítása során a felhasználói program összeállít egy virtuális színteret (scene), melyben elhelyezhet 3D modelleket, fényeket és kamerákat. A 3D modellek jellemzően egy felületet írnak le. A felület definiálható analitikusan (pl. egy gömb, véges vagy végtelen sík, téglatest stb.) vagy megadható háromszöghálók segítségével, valamint ezekhez a felületekhez textúrakép is rendelhető. A modellek alakja vagy a modellek egymáshoz viszonyított elhelyezkedése időben animálható, így élethű mozgás látszata is elérhető.

5.3.1 Sugárkövetés

3D alapú fotorealisztikus grafikák előállításra különböző megközelítések léteznek. A logikailag egyszerűbb, ám számítási igény szempontjából komplexebb megközelítés a sugárkövetés (raytracing, path tracing). Ebben az esetben egy virtuális kamerából az elkészítendő kép minden pontján keresztül egy-egy sugarat indítunk és vizsgáljuk, mely objektum felületét metszi el először. Ebben a pontban mintavételezhetjük a textúrát, így az adott képponthoz (amelyből a sugár indult) szín is hozzárendelhető. A kapott kép úgy tehető még élethűbbé, ha a virtuális térben elhelyezett fények hatását is felhasználjuk: indíthatunk újabb sugarakat minden egyes fényforrás felé, így meghatározhatjuk, hogy vet-e egy másik objektum árnyékot a vizsgált felületpontra, valamint árnyalhatjuk a képpont színét a fényforrás iránya és a felületi normális által bezárt szöggel, így kidolgozva realisztikusabb fény-árnyék hatásokat.

Copyright © 2021 / BME AUT



5.9. ábra – Sugárkövetés: a kamerából indulnak a sugarak az objektumok felé és tovább a fényforrásokhoz [5]

A fent vázolt módszert alkalmazva viszonylag kevés munkával szép eredmény érhető el, illetve további finomításokkal (tükröződések, fénytörés, objektumok felületéről visszaverődő fény kezelése) még tovább javítható az eredmény. Azonban vegyük észre, hogy nagyobb felbontás esetén, egyre több objektumot használva a megvalósításhoz szükséges számítási igény hatványozottan nő: minél több új sugarat indítunk (tükröződések, több objektum, több fény felé, sőt, fényekből is), annál több időt vesz igénybe a számítás. Léteznek trükkök, amelyekkel bizonyos problémák részben egyszerűsíthetők (pl. az objektumok pozíciója és az általuk elfoglalt tér alapján jól megválasztott adatstruktúrák esetén gyorsítható a sugarakkal történő metszések keresése), azonban tisztán sugárkövetésre alapozni valósidejű 3D fotorealisztikus grafikát egyelőre nem jellemző, fényhatások (árnyék, tükröződés, fizika) valóságosabbá tételére inkább más módszerrel kombinálják (ld. raszterizáció). A kép előállítása során használt sugarak számának limitálásával tartható a valósidejűség, azonban ez sokszor zajos képet eredményez, amit utólag szűrni, javítani kell. A sugárkövetés elsősorban nem valósidejű alkalmazások esetén bevett módszer (3D animációk, filmek készítése), főleg CPU-kon valósítják meg, de jól párhuzamosítható: a képkockák vagy akár a képek részei külön-külön renderelhetők.

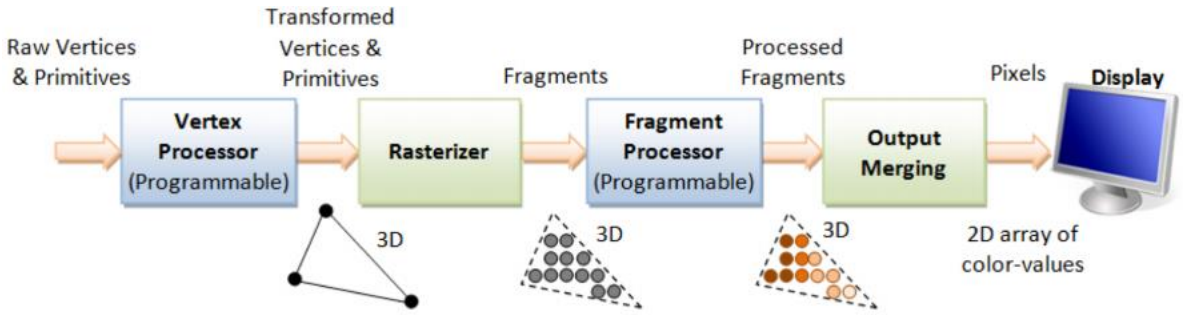
A leginkább valósághű képek előállítása a photon mapping módszerrel történik, itt minden hatást a fizikai törvények szerint szimulálnak: a fényforrásokból kibocsátott fotonok útját egyenként határozzák meg, azonban ez a leglassúbb módszer is egyben, hiszen a fényforrásokból induló fotonok nagyon kis része fog a kamerába megérkezni, így sokáig kell futtatni.

5.3.2 Raszterizálás

Valósidejű 3D grafika esetén bevált módszer a raszterizálás, melynek lépéseit a grafikus csővezeték (graphics pipeline) szemlélteti. A látványt a virtuális 3D színtérben elhelyezkedő objektumok és egy virtuális kamera képe adja. Az objektumok primitívekből állnak: pontok, élek, poligonok, felületek leírására leggyakrabban háromszöghálókat alkalmaznak.

[5] Forrás: https://en.wikipedia.org/wiki/Ray_tracing_(graphics) (Hozzáférés 2020.06.29.)

Copyright © 2021 / BME AUT



5.10. ábra – Raszterizálás során a háromszöghálót rajzolják a képernyőre. A valósághű fényezés és színezéshez számos trükkre, „csalásra" van szükség [6]

A processzoron a felhasználói program előállítja a 3D világ reprezentációját (scene), melyben az objektumok felületét térbeli háromszögháló írja le. Amennyiben a modellezés során analitikus felületeket alkottak meg, a rajzolás előtt ezeket (a részletességnek megfelelő mennyiségű háromszögből álló) háromszöghálóvá alakítják. A processzor elküldi a GPU-nak a geometriai információkat, azaz a háromszöghálók összes 3D pontját (vertexét), valamint az összes háromszög csúcspontját alkotó vertex sorszámát (indexét) és normálvektorait, valamint a megjelenítéshez szükséges „anyag" (material) információkat (textúrák, és a vertexek 2D textúra koordinátái).

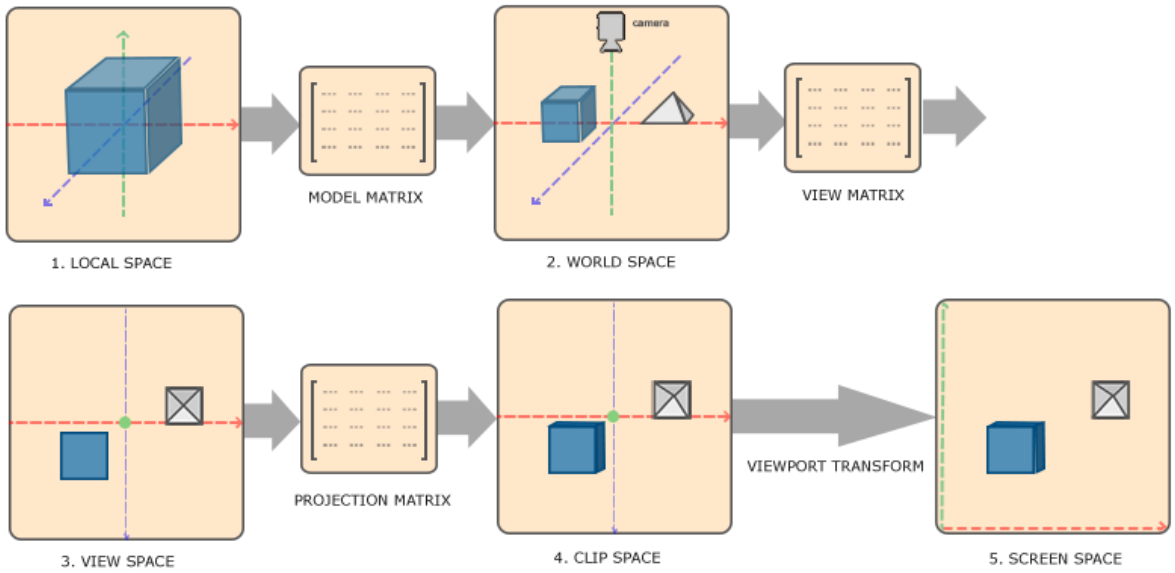
A grafikus processzor első lépésként a vertexeket dolgozza fel, minden vertexet először a modell koordinátákból világkoordinátába, majd egy kiválasztott kamera koordinátarendszerébe transzformálja. A vertexek eredetileg az objektum saját koordinátarendszerében vannak megadva (így akár több objektum egymásba is ágyazható), ezért szükséges az objektum pozíciója alapján minden vertexet előbb a világ, majd a kamera koordináta-rendszerébe transzformálni. A 3D transzformációs lépések egyszerű mátrixszorzással megadhatók, minden 3D vertexre forgatás és eltolás műveleteket kell végrehajtani. A háromdimenziós vertexek forgatása egy 3x3 méretű mátrixszal, eltolásuk pedig egy 3 elemű vektorral lehetséges. A háromdimenziós reprezentáció bővíthető négydimenziós homogén koordinátarendszerbe, ebben az esetben egy 4x4-es mátrixszorzással kezelhető mind a forgatás, mind az eltolás.

$$V' = M_{transform} V$$

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

[6] Forrás: https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT



5.11. ábra – Koordináta transzformációk az objektum koordináta rendszerétől a képernyőig [7]

Többszöri koordinátatranszformáció esetén a transzformációk összeszorozhatók, így nem kell a pontokon újra és újra elvégezni. Ez a módszer alkalmazható egymásba ágyazott objektumok esetén is.

$$V_{clip} = M_{projection} M_{view} M_{model} V_{local}$$

Természetesen a vertexek mellett a fények és a háromszögek normáljainak is a kamera koordináta rendszerébe kell kerülniük. A lighting (fényezés) lépés során minden vertexhez meghatározható egy fényesség érték, ami az egyes fényforrások, a normálok, valamint az anyagtulajdonságok függvénye.

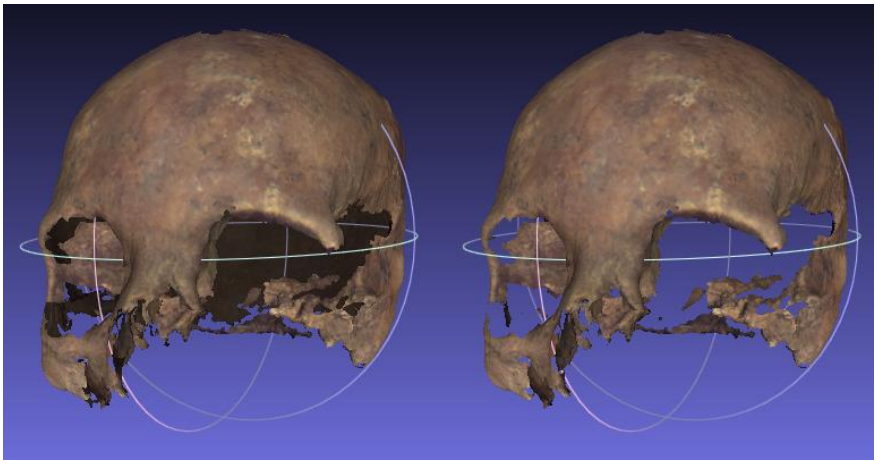
Következő lépésként megtörténik a kamera belső paraméterei által meghatározott projekciós mátrix alapján a projekció (vetítés), ez biztosítja a perspektivikus hatást, valamint ez definiálja a látószöget. Természetesen a belső paraméterek egyedi megválasztásával akár ortogonális vetítés is megadható.

A következő, clipping (vágás) lépés során a látható tartományból kieső háromszögeket távolítja el a rendszer. A láthatónak tekintett tartomány a (–1;–1) és (1;1) között megadott négyzet által kijelölt rész. A látómezőből részben kilógó háromszögek esetén a kilógó vertexeket a látható tartomány szélén újat meghatározva pótolja. A látható tartomány terét az úgynevezett (csonka gúla alakú) frustrum határozza meg, a látószög által definiált teret közeli és távoli vágási síkkal kiegészítve. Végül a Window-Viewport transzformáció során egy eltolás-skálázás transzformáció történik annak érdekében, hogy az egyes vertexek vetületei a cél kép koordináta rendszerében értelmezhetőek legyenek. Jellemzően a kép közepét eddig kijelölő (0,0) koordináta a bal felső sarokba kerül, míg a normalizált (2x2) szélességű és magasságú tartomány a kép szélességének és magasságának megfelelő tartományba képződik le (pl.

[7] Forrás: https://www.programmersought.com/article/39244034502/ (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

1920x1080). A back-face culling módszer további gyorsítást tesz lehetővé: eldobja azokat a háromszögeket, melyeknek a normálja nem a néző felé, hanem tőle távolra mutat. Ebben az esetben figyelni kell arra a modellezés során, hogy ahonnan a modell látható, ott mindenképpen legyenek megfelelő normálú háromszögek.



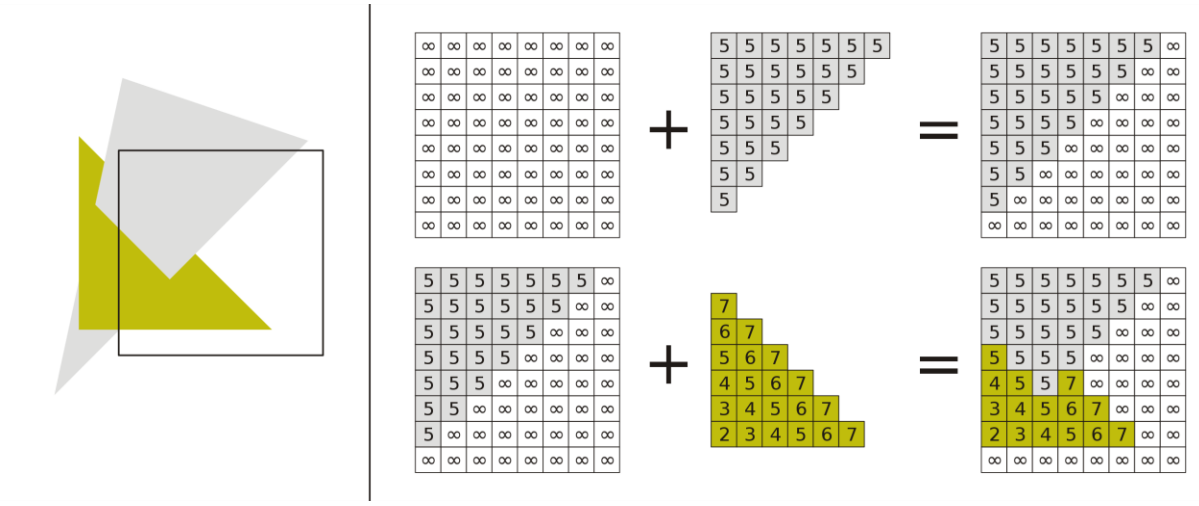
5.12. ábra – Backface culling: zárt felületek esetén jó, nyitottnál eltűnik a „hátsó oldal” [8]

A koordináta-transzformációk után következik a raszterizáció. A korábban meghatározott háromszögeket egy rácsra rajzolja a rendszer, így határozza meg a „fragment”-eket, rajzolás közben a rendszer perspektivikus interpolációt alkalmaz. Minden fragment egy képpontot (pixelt) fog reprezentálni a végleges képen, azonban a fragment-ek még nem a végleges képpontok, így előfordulhat, hogy több háromszög is átfedésbe kerül egy képponton, ezért további műveletekre lehet szükség.

Ahhoz, hogy a rendszer meghatározza, mely fragment-ekből lesznek látható képpontok, szükség van az adott képpont esetén a kamerához legközelebb eső fragment meghatározására. Ehhez az úgynevezett Z-puffert alkalmazza (mérete megegyezik a cél kép felbontásával), mely nyilvántartja a kirajzolt fragmentek z (mélység) koordináta értékeit minden képpont pozícióban, ezáltal nem teszi lehetővé, hogy a rendszer felülírjon távolabbi fragment-ekkel már kirajzolt közelebbieket (átlátszóságot nem kezelve).

[8] Forrás: https://en.wikipedia.org/wiki/Back-face_culling (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT



5.13. ábra – Z-puffer algoritmus [9]

A fragment színének meghatározásához szükség van az objektum anyaga által meghatározott textúra mintavételezésére. A rendszer interpolációval meghatároz egy textúra koordinátát a háromszög csúcsai és a csúcsokhoz rendelt textúra koordináta alapján, valamint a vizsgált fragment koordinátája függvényében, a mintavételező egység pedig visszaad egy színt az adott valós textúra koordináta alapján, újabb interpolációt is alkalmazva. A fényezéssel kapcsolatos információt is interpolálja a rendszer a fragment számára a háromszög csúcsai alapján. Az eredményül kapott fragment szín pedig beíródik a kép-pufferbe.

5.3.3 Shader-ek

Kezdetben az egyes feldolgozó blokkok fix funkcióval rendelkeztek, majd fokozatosan váltak általánosabban programozhatóvá. A fix transzformációs blokkot felváltotta egy úgynevezett vertex shadert futtató programozható blokk, így egy tetszőleges felhasználói program volt felelős a vertex transzformációért. Alapesetben a vertex shader a korábbi funkciót valósította meg, de lehetővé vált tetszőlegesen pozícionálható, vagy akár az idő függvényében pl. hullámzó felületeket vagy mozgó objektumokat létrehozni anélkül, hogy a CPU-nak további terhelést jelentett volna. A programozható pixel (később fragment) shaderek lehetővé tették pixelek színének rugalmasabb előállítását: a GPU-n futó kis program állíthatja elő a képpont végleges színét számos rendelkezésre álló információ függvényében: a háromszög vertexei, normálisa, textúra koordináta, fragment koordináta, z-érték, árnyékok, átlátszóság, egyéb pufferek adatai stb, lehetővé téve sokkal látványosabb vagy realisztikusabb rajzolást. Az egyes GPU generációk során egyre szélesebb körűen programozhatóvá váltak az egységek, végül az egyesített (unified) shader-ek bevezetésével már nem is különböztették meg a hardverben az egyes funkciókra dedikált blokkokat, egységes végrehajtó motorok készültek minden shader típushoz. Ezzel a megoldással jól lehetett kezelni az eltérő bonyolultságú shaderek futtatását, mindig a lehető legtöbb végrehajtót lehetett ütemezni, nem alakultak ki üresjáratok, szűk keresztmetszetek a ki nem használt erőforrások miatt.

[9] Forrás: https://de.wikipedia.org/wiki/Z-Buffer (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

5.3.4 GPGPU

A programozható shader egységek lehetővé tették, hogy bizonyos korlátozások mellett egyszerű, ám tetszőleges programot hajtsunk végre a GPU-n (GPGPU: General Purpose GPU programming). Mivel a shader programok kezdetben elsősorban koordináta-transzformációkra, színezésre készültek, általános célra meglehetősen erős korlátozással voltak csak használhatók. Az elágazások, ciklusok, többféle típus támogatása sokáig hiányzott a lehetőségek közül, idővel mégis lehetővé vált bizonyos számolási részfeladatokat visszavezetni ilyen műveletekre, valamint a sok párhuzamosan és hatékonyan működő végrehajtó új lehetőségeket nyitott meg. Kezdetben a vertex shadereknél szigorúbb követelmények voltak a pontosságra vonatkozóan, mint a pixel shaderek esetén. Utóbbiak hiába kaptak 32 bites lebegőpontos adatokat, takarékossági és teljesítménynövelési okokból kisebb pontossággal (pl. 24 biten) végezték el a műveleteket, később vált csak követelménnyé a teljes 32 bites pontosság. Az egész típusokkal történő és a bitenkénti logikai műveletek is később kerültek bevezetésre. A shader végrehajtók fejlődésével és a unified shaderek bevezetésével egyre szélesebb körben lehetett számítási feladatokat a GPU-ra bízni. A rendszer hatékonysága a nagy számban, független módon végrehajtható számítási részfeladatok megoldásában mutatkozott meg. A GPU-k számítási kapacitásának általános célú felhasználásához szükséges a megfelelő szoftveres támogatás (API-k, eszközmeghajtók) is, ezekről is lesz szó a továbbiakban.

5.4 GPU memória

A nagyszámú, párhuzamosan működő számítási egységeket folyamatosan adattal kell ellátni a hatékony, üresjárat nélküli működés érdekében. A dedikált GPU-k számára dedikált külső memória érhető el, kiépítéstől függően 2–16GB kapacitással. A kapacitás mellett két további lényeges paraméter a sávszélesség és a késleltetés.

Felépítésüket tekintve különböző típusú memóriák érhetők el, melyek mind különböző jellemzőkkel (fogyasztás, kapacitássűrűség) és interfészekkel rendelkeznek. Az egyik legkorábbi memóriatípus, az SRAM (Statikus RAM) flip-flop-okat használ egy-egy bit tárolására, jellemzően hat MOSFET tranzisztor szükséges egy bit tárolásához és eléréséhez. Előnye, hogy gyors, megbízható, felépítése viszonylag egyszerű, azonban működés közben magas a fogyasztása, drága és kicsi a kapacitássűrűsége. Elsősorban a processzoron belül regiszterek, cache-ek megvalósítására használják. Vezérlésüket, interfészüket tekintve is egyszerűek, a cím- és adatvezetékeken túl néhány vezérlőjel szükséges, mint pl. CE (chip enable), WE (write enable), OE (output enable).

Nagykapacitású központi memóriaként a dinamikus memóriák terjedtek el (DRAM), ezekből is a szinkron (órajelvezérelt) SDRAM változat korán leváltotta az aszinkron megoldásokat. Előnyük, hogy olcsóbbak, nagyobb kapacitássűrűség érhető el velük, ugyanis egy bit tárolásához elegendő egy tranzisztor, mint kapacitás, azonban bonyolultabb vezérlést igényel, és lassabb elérést tesz lehetővé. Az adattároló kapacitások adott idő után kisülnek, ezért időnként szükséges a frissítésük. A vezérlés bonyolultabb állapotgépet igényel: a vezérlőjelek (nCS, nCAS, nRAS, nWE) alkalmazásával különböző parancsok adhatók ki, pl. aktiválható egy adott bank egy adott sora, melyben a vezérlő tetszőleges sorrendben végezhet írási vagy

Copyright © 2021 / BME AUT

olvasási műveleteket, végül a sort le kell zárni (precharge fázis). Az egyes műveletek között adott számú órajel ütemet kell várni, azonban a bankok között szabadon válthatunk.

Írás után adott időn belül egy frissítés parancsot kell kiadni, hogy a sor ne veszítse el a tárolt adatot. Jellemző érték DRAM-ok esetén, hogy 64ms-on belül meg kell történnie a frissítésnek, amely elvégezhető egy lépésben (ekkor más művelet nem végezhető párhuzamosan a DRAM-mal), vagy időben elosztva, csak soronként frissítve (pl. 4096 sor esetén 15,6 µs-onként ki kell adni egy sorfrissítés parancsot, a kiválasztott sort pedig egy számláló segítségével a memória modul belső vezérlője tartja nyilván). A DRAM burst módban, ha sok adatot ír vagy olvas egyszerre, nagy sávszélességre képes, azonban véletlenszerű elérések esetén kifejezetten rossz, nagy késleltetéssel működik. Ezért terjedtek el a cache-ek, gyorsítótárak, melyek átmeneti pufferként működnek a processzor és a külső memóriák között.

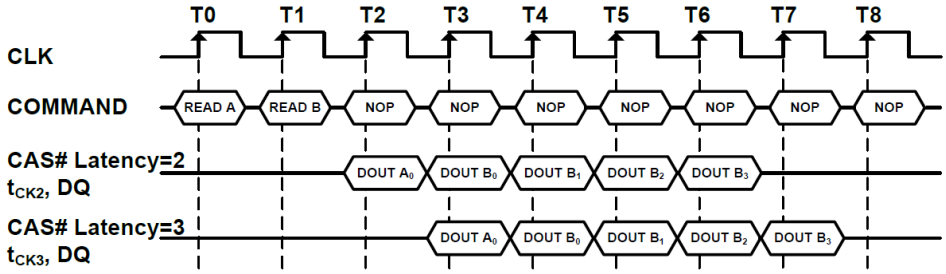


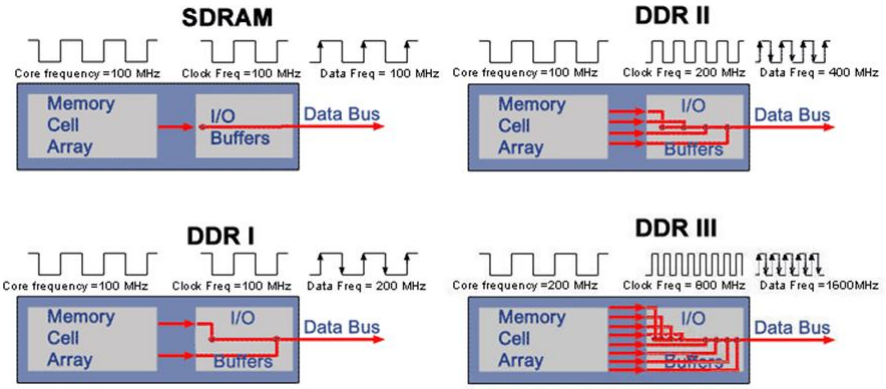
Figure 5. Read Interrupted by a Read (Burst Length = 4, CAS# Latency = 2, 3)

5.14. ábra – Egy egyszerű SDR SDRAM esetén a kiolvasott adat két vagy három órajel késéssel jelenik meg [10]

A sávszélesség növelésében az első ugrás a DDR (double data rate) típusú SDRAM bevezetése volt, melynél mind a fel, mind a lefutó éleken is megtörtént a jelek mintavételezése. A belső memória órajelenként és adatvonalanként 2-2 bithez fért hozzá, amelyeket egy-egy órajelre adott ki vagy olvasott be. A későbbiekben az átviteli sebességet tovább növelték, változatlan belső működési órajel mellett 4-4 bithez fértek hozzá (DDR2) vagy 8-8 bithez (DDR3) és duplázott, illetve négyszerezett külső órajel mellett továbbították vagy fogadták az adatokat. Az egyes generációk átviteli sebességének növelésével párhuzamosan az adatelérés órajelekben számolt késleltetése is növekedett, azonban a magasabb interfész órajel ezt kompenzálni tudta. Az átviteli sebesség tovább növelhető több memóriamodul párhuzamos használatával, ezt a GPU-k esetén ki is használták.

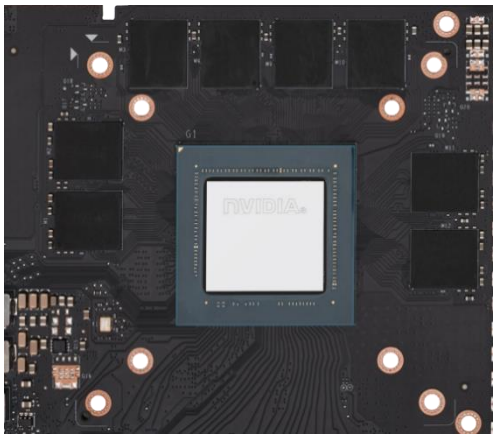
[10] Forrás: ISSI IS42S32200L Datasheet (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT



5.15. ábra – Prefetch az egyes memóriatípusok esetén [11]

Dedikált RAM-mal rendelkező grafikus adapterek esetén alkalmazzák a GDDR DRAM (SGRAM) típusú memóriákat, melyek magasabb frekvenciákon működnek, valamint speciális, grafikus alkalmazásoknál gyakori funkciókkal is rendelkeznek, mint pl. bit maszkolás textúra színcsatornák külön-külön eléréséhez, illetve akár két lap is megnyitható egyszerre.



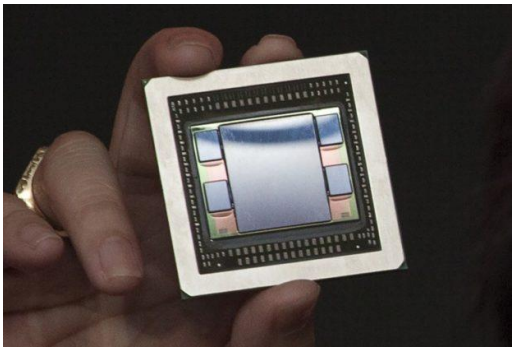
5.16. ábra – GPU és körülötte a diszkrét DRAM lapkák

2015-ben az AMD Fiji kódnevű GPU-ja speciális memóriaillesztéssel jelent meg. A HBM (High Bandwidth Memory) megoldás lényege, hogy a memóriát fizikailag is nagyon közel illesztik a GPU mellé, egy közös szilícium interposerre helyezik mindkettőt, DRAM lapkákat egymásra rétegeznek, a GPU és a memóriák között pedig nagyon széles, összességében akár 4096 bites interfészt biztosítva, bár alacsonyabb frekvencián működtetve. A 8 rétegű, darabonként 128 bites memórialapkákból összesen négy modul került bele: 8 x 128 bit x 4 = 4096 bit. 500MHz-es órajel esetén DDR átvitelt alkalmazva a maximális sávszélesség 1GT/s jelvezetékenként. A megoldás előnye nem csak a nagy sávszélesség, hanem a kis fizikai méret,

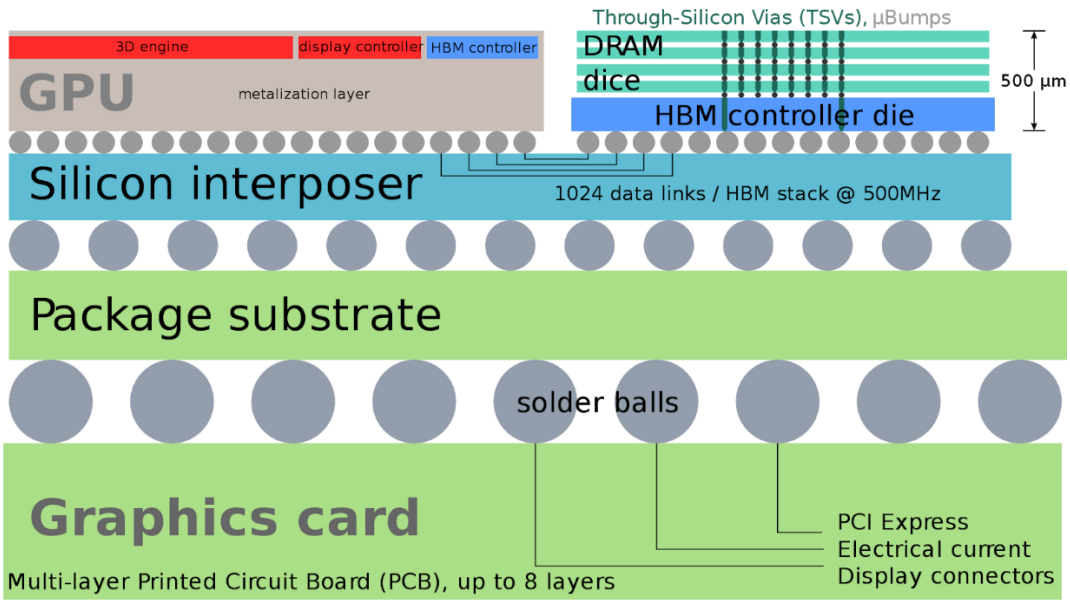
[11] Forrás: http://www.x86-secret.com/articles/ram/ddr2_667/ddr2667-6en.htm (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

ugyanis nincs szükség a nyomtatott áramköri lapon diszkrét DRAM IC-k jeleit huzalozni. A továbbfejlesztett HBM2, HBM2E változatoknál a vezetékeken elérhető maximális adatátviteli sebesség tovább növekedett 2GT/s, illetve 3.6GT/s-ra, valamint akár még több memórialapka is egymásra rétegezhető.



5.17. ábra – HBM memória és GPU közös hordozón (AMD)



5.18. ábra – Többrétegű multi chip module kialakítás interposerrel, nagysűrűségű vezetékezéshez[12]

5.5 GPU architektúrák

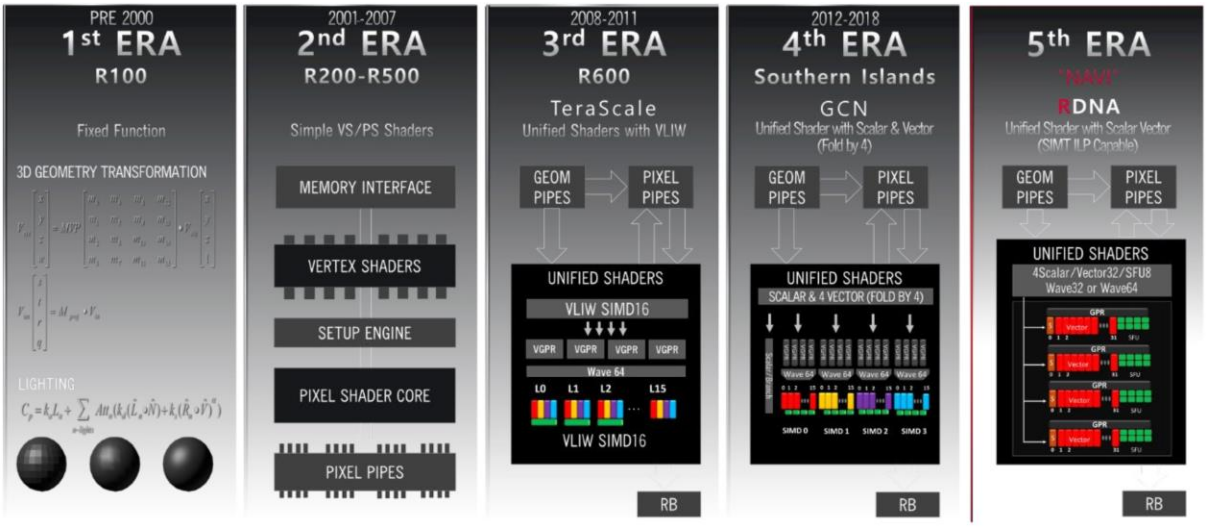
Az első (kezdetleges) programozható shader egységet az nVidia GeForce 3 GPU-ja tartalmazta (2001). Kompatibilitás miatt egy fix funkciós Transform&Lighting egységet is beépítettek, mert a programozható egység nem volt elég gyors a korábbi hardveres megoldáshoz képest. Az ATI Radeon 9700-as terméke (2002) már lehetővé tette akár ciklusok vagy bonyolultabb számítások megvalósítását, így már megkötésekkel, de léteztek erre támaszkodó, gyorsítást

[12] Forrás: https://en.wikipedia.org/wiki/Through-silicon_via (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

használó nem grafikai alkalmazások. Az nVidia GeForce 8-as sorozata (2006) új, általános célú stream processzorokra épült és emellett elérhetővé tette a CUDA nevű szoftverkönyvtárat, ami kifejezetten általános célú számítások GPU-n történő futtatását könnyítette meg. Az ATI (AMD felvásárolta) HD2000-es sorozata (2007) vezette be a unified shader modellt egy VLIW (Very Long Instruction Word) alapú GPU architektúrával, majd a HD7000-es sorozattal (2012) váltott új SIMD alapú (GCN) architektúrára.

Az nVidia a GeForce 200-as sorozatban (2008) vette át a unified shader megoldást, majd mindkét gyártó folyamatosan bővítette a termékeik képességét. Az nVidia a 20xx-as sorozattal (2018) mutatta be az első, sugárkövetés gyorsítására is képes GPU-t. A hardvergyártók és a szoftverfejlesztők együttműködve, folyamatosan derítették fel a szűk keresztmetszeteket akár az API-kban, akár a hardver architektúrákban és minden generációban tovább optimalizálták a megoldásokat. A hardveres fejlesztések jellemzően megelőzik az API-kat, az alkalmazásfejlesztők pedig még később tudják kihasználni a megjelent technológiákat: nem ritka, hogy akár 5 év is eltelhet, mire az alkalmazások megfelelően ki tudnak használni egy-egy hardveres újítást. A GPU-k fejlődése során erős korlátozó tényező volt a gazdaságos gyárthatóság: az ár, a teljesítmény, a tranzisztorok száma, a lapka mérete és a fogyasztás összehangolása. Külön nem térünk rá ki, de a GPU-k további fix funkciós multimédiás egységekkel is rendelkeznek, mint pl. különböző típusú video kódolás és dekódolás gyorsítása, mely sokkal hatékonyabb dedikált hardverrel, mint szoftverrel. Az 5.19 ábrán látható a GPU-k felépítésének fejlődése a fix funkciós megoldásoktól az általánosan programozható megoldásokig.



5.19. ábra – A fix funkciós egységektől az általánosan programozhatók felépítésig[13]

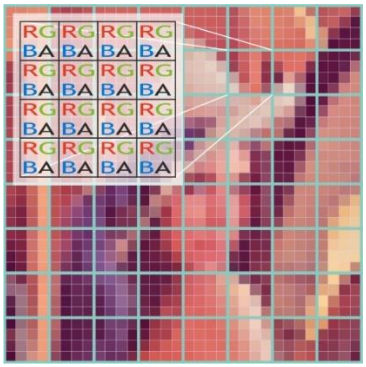
Ahogy már említettük, dedikált GPU-k mellett CPU mellé integrált változatokban is léteznek grafikus processzorok. Az Intel saját HD Graphics megoldását integrálja CPU-iba, az AMD is saját dedikált GPU-i kevesebb számolóegységgel rendelkező változatait építi APU („accelerated processing unit” kombinált CPU és GPU) egységeibe. Beágyazott

[13] Forrás: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

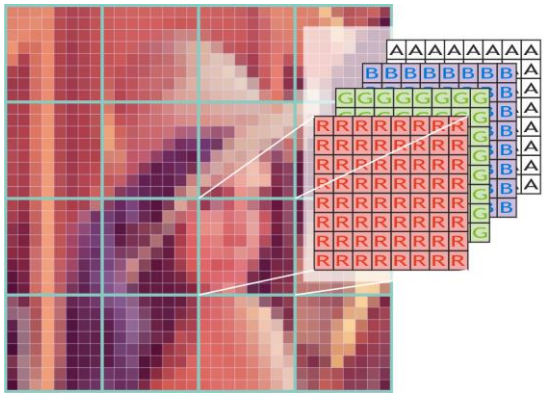
alkalmazásprocesszorok is rendelkeznek grafikus gyorsítókkal. Az nVidia ARM processzorok mellé integrálja saját GPU számolóegységeit, a Qualcomm Adreno megoldásai AMD licencek alapján készültek. Az ARM a Mali grafikus gyorsító IP-t biztosítja a CPU magok mellé. Az Imagination Technologies PowerVR grafikus magjai MIPS vagy ARM CPU-k mellé is építhetők.

A grafikai feladatok esetén hatékony az SIMD koncepció (single instruction multiple data), ez a megközelítés az általános célra használható GPU-knál is megmaradt. Ebben az esetben már nem több képponton vagy vertexen végzik ugyanazokat az utasításokat, hanem valamilyen pufferek elemein. GPU-k esetén nem az ütemezési egységeket hívják szálaknak (thread), hanem az adatokat, amin ugyanazt a kódot (kernelt) futtatja a rendszer. A szálakat hívják work-item-nek is, melyeket warp-nak vagy wavefront-nak nevezett egységekbe csoportosítanak, ezeken az elemeken hajtják végre a multiprocesszorok az utasításokat. Jellemzően 32 vagy 64 elemből áll egy wavefront. Grafikai alkalmazás esetén ezek képpontok vagy vertexek lehetnek, ahol az egyes komponenseket egymás után, külön utasításokkal dolgozza fel a rendszer. SIMD végrehajtás esetén külön utasításokat hajtanak végre az egyes színkomponensekre vagy koordinátákra. Természetesen a GPU-k méretükből adódóan akár több warp vagy wavefront párhuzamos feldolgozására is képesek. A korábbi architektúrák esetén egy lépésben kevesebb képpontot (vagy vertexet) dolgoztak fel, de azok minden komponensét egy egyszerre, vektorutasításokkal. Ebben az esetben gondot okozott, ha az eltérő komponensekre eltérő műveletet kellett végrehajtani, ilyenkor lassabbá vált a feldolgozás. Ha például textúrák esetén az átlátszóság kezelésére nem volt szükség, akkor is ugyanannyi ciklus kellett a feldolgozáshoz.



5.20. ábra – Grafikai célokra korábban jellemző volt vektorosan kezelni a képpontokat a színkomponenseikkel, 4x4-es (vagy 4x2-es) pixel blokkon hajtották végre a 4 elemű vektoros műveleteket a 4 színcsatornán (3 szín + átlátszóság)

Copyright © 2021 / BME AUT

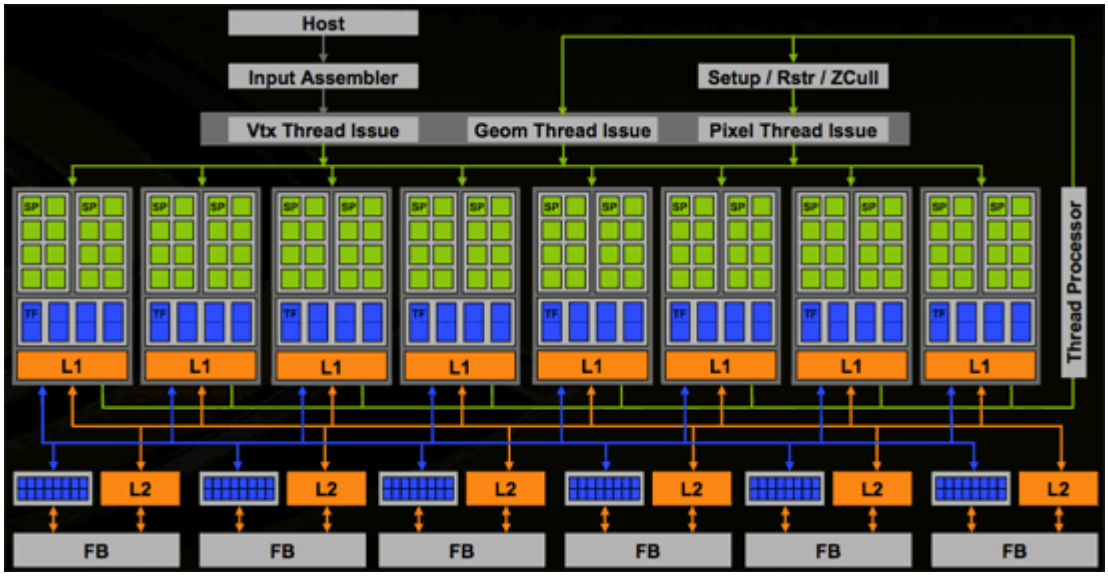


5.21. ábra – Általános célú feldolgozók esetén megmaradt a 64 (vagy 32) adat párhuzamos feldolgozása, de egy ciklusban azonos típusú adatok feldolgozása történik, pl. képek esetén az egyes színkomponensek külön ciklusban, utasításban kerülnek feldolgozásra, így a feldolgozás hatékonysága növekedhetett, a végrehajtóegységek egyszerűsödtek a homogénebb adatok miatt

5.5.1 nVidia G80, GeForce 8 (Tesla)

A 2006-ban bemutatott GPU összesen 128 stream processzort (SP) tartalmaz. Egy stream processzor skalár műveletek végrehajtására képes. 16 SP kerül egy blokkba, ezek osztoznak 4 textúra címképző, 8 textúra szűrő egységen, valamint egy L1 (level 1) cache-sen. A processzor elődjében, a G70-ben egy négylépcsős, egyenként 4 elemű vektorprocesszor képzett egy blokkot. A G80-ban ezt osztották fel 16 önálló skalár processzorra, vagyis architektúra szinten is egy lépéssel távolabb került a vertex és pixel feldolgozása esetén eddig használt vektoroktól és közelebb került az általános célú processzorkohoz. Minden SP egy MAD (Multiply and Addition) és egy MUL (Multiply) művelet kibocsátásra képes órajelciklusonként (pipeline-os végrehajtással). Az SP-k működése erősen csővezetékezett, ezért a processzor magas órajelen képes működni. A hosszú csővezeték általában nem okoz problémát, ugyanis grafikai számításoknál ritkább az utasítások egymásra hatása. A szál menedzsment blokk (vertex, geometry, pixel thread issue) látja el az SP blokkokat utasításokkal. Fontos, hogy a stream processzorok nem rendelkeznek külön-külön utasításdekódolóval, nem képesek egyszerre eltérő utasításokat végrehajtani, a szál menedzsment blokk lát el utasításokkal egy-egy blokkot és ezt az utasítást hajtja végre minden SP az adott blokkban a saját adatain. Az architektúra lehetővé teszi, hogy ha a feldolgozó program véget ér, kiírja a kimeneteket a kimeneti pufferbe és felszabaduljanak az erőforrások. Korábbi processzorok esetén az adatok végigmentek a teljes grafikus csővezetéken, és a képpufferbe íródtak.

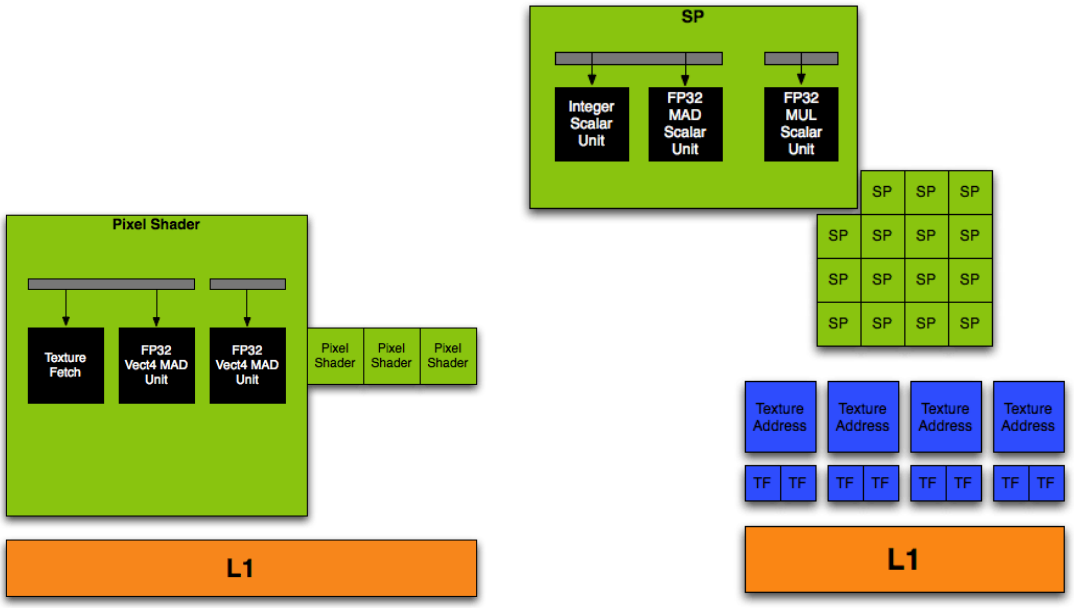
Copyright © 2021 / BME AUT



5.22. ábra – G80 blokkvázlat (nVidia)

Az elődnek tekinthető G70 GPU-ban még elkülönültek a pixel és vertex feldolgozók. A pixel feldolgozók négyes blokkokban helyezkedtek el, és minden blokk 2 db négy elemű vektor végrehajtóból (MAD: Multiply Add), valamint egy textúrabetöltő egységből állt. Ilyen négyesek osztoztak az L1 szintű cache memórián, ugyanis jellemzően szomszédos képpontoknak a lokalitás elv alapján hasonló adatokra van szüksége. Látható, hogy ez a 4x4-es számolóegység szolgált a 16 SP-ből álló blokk alapjául, immáron egyesített (unified) megvalósításban, a textúra egységet is leválasztva a kis processzorokról. Pixelek esetén a 3+1 színcsatorna (RGBA), vertexek esetén a homogén 3D koordináták igénylik a vect4 (4x32 bites) blokkokat. Míg a G70 egy shader blokkja egyszerre négy pixel vagy vertex 4 komponensét dolgozta fel, addig a G80 16 pixel vagy vertex egy-egy komponensét dolgozta fel párhuzamosan. Látszólag nincs nagy különbség, mégis, ha nincs szükség minden komponens feldolgozására (pl. átlátszóság), a G80 gyorsabb működésre képes a szükségtelen lépések kihagyásával.

Copyright © 2021 / BME AUT



5.23. ábra – Balra G70 pixel shader, jobbra: G80 stream processzor blokkok [14]

Minden SP saját „szálat” futtat, a szál itt pl. egy-egy vertex koordináta, primitív vagy pixel egy-egy színkomponensének feldolgozását jelenti, nem programszálakat. A G80 egyszerre egy programkódot (kernel) volt képes kezelni. Azoknak az SP-knek, amelyek közös erőforrást használnak, azonos típusú szálat kell futtatniuk, tehát minden 16 SP-ből álló blokk ugyanazt az utasítást hajtja végre, de különböző adatokon. Amennyiben a kód tartalmaz elágazásokat, de minden pixel feldolgozásánál ugyanaz a feltétel eredménye, nincs probléma. A branch granularity értéke 32, azaz, ha 32 pixel egyidejű feldolgozása során akár egy esetben is eltérő egy elágazási feltétel értéke, a 32 pixel esetén mindkét ág végrehajtódik, feljegyezve, hogy mely ágakon keletkezett eredményeket mely esetben kell érvényesíteni. Vertex feldolgozása esetén ez az érték 16. A processzor memóriavezérlője 5-6 csatornásra bővült a korábbi 2-4 csatornásakkal ellentétben, ami összesen 320-384 bites hozzáférést jelent, illetve órajelenként ennek dupláját, lévén DDR-ről van szó.

A gyártó által biztosított CUDA (Compute Unified Device Architecture) lehetővé teszi általános, C nyelven megfogalmazott kód fordítását, majd futtatását a GPU architektúráján. A rendszer akkor igazán hatékony, ha ugyanazt, vagy kevés különböző programkódot szeretnénk futtatni nagyszámú forrásadaton. A rendszer lehetővé teszi CUDA és grafikus feldolgozás futtatását is. Minden 16 SP-ből álló blokkban képesek az SP-k adatot megosztani egymással a blokkon belül anélkül, hogy a központi memóriába kellene kiírni és visszaolvasni (pl. helytakarékossági okokból hasznos, ha nem kell többször tárolni az adatot, vagy köztes eredmények a többi processzor számára is rendelkezésre állnak).

[14] Forrás: https://www.anandtech.com/show/2116/6 (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

5.5.2 nVidia GF100 Fermi

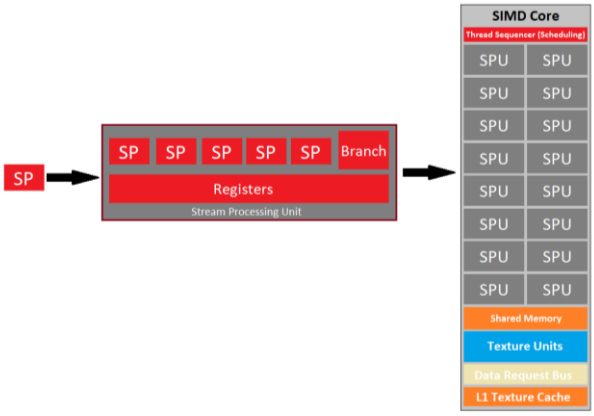
A G80-at követő, 2008-ban bemutatott GT200-as GPU (GeForce 280) már támogatta a dupla pontosságú lebegőpontos számításokat.

A későbbi, 2010-es nVidia Fermi (GF100) architektúra már javította azt a hiányosságot, hogy egyszerre csak egyféle kernelt kezelt a GPU. A 64 bites lebegőpontos típus támogatása tovább javult, a szimpla pontossághoz képest fele sebességgel volt képes végrehajtani, míg korábban csak nyolcad sebességgel. A műveletvégzés legkisebb egységét már CUDA magnak hívták, összesen 512 található belőlük a GPU-ban. Az egész típusú műveletvégzés már teljes 32 bites szavakban történt, korábban ezt a lebegőpontos egység emulálta 24 biten, ezért nem lehetett 24 bitesnél szélesebb egész szorzásokat végezni. Egy SM (streaming multiprocessor) egység 32 CUDA magot tartalmazott, saját utasítás ütemezővel. A legnagyobb kiépítésű GPU 16 SM egységből állt. Az egyes SM egységek már különböző utasításokat is végrehajthattak egy időben, így a feldolgozás különböző warpok között átlapolódhatott, így növelve a hatékonyságot, ez persze megkövetelte, hogy különböző kernelek is futhassanak a GPU-n egy időben.

5.5.3 AMD Terascale VLIW architektúra

Az AMD GPU-i több generáción keresztül (Radeon 2000-es – 6000-es sorozat, 2007–2012) a VLIW (Very Long Instruction Word) megközelítést alkalmazták. A feldolgozóegységeket Streaming Processor Unit-nak (SPU-nak) nevezték. Egy SPU 4-5 számolóegységet („Radeon core” vagy „Stream Processor”), regisztereket, elágazáskezelő egységet és egyéb modulokat tartalmaz. 16 SPU alkot egy SIMD magot (SIMD core), melyből kialakítástól függően, pl. 10 db van egy GPU-ban. Az SIMD egységen belül minden SPU ugyanazt a VLIW utasítást hajtja végre. Minden SPU-ban az egyik mag speciális, eltérő utasításkészlettel rendelkezik, pl. transzcendens (szögfüggvények, exponenciális, logaritmus) utasításokat csak a speciális SP tud végrehajtani.

Copyright © 2021 / BME AUT



5.24. ábra – AMD Terascale VLIW SP, SPU, SIMD core [15]

Az SPU-k VLIW utasításokat hajtanak végre, melyek ideális esetben több, párhuzamosan végrehajtható, független kis utasításból állnak, melyek az SPU-kat alkotó műveletvégzőkben hajtódnak végre. Az architektúra hatékonysága akkor mutatkozik meg, ha sok, egymástól független utasításdarabkából álló hosszabb utasítások érkeznek, ezeket párhuzamosan hajthatják végre a kisebb egységek. Ez a gondolatmenet jól illeszkedik a grafikus processzorokhoz, amikor nagyon sok hasonló műveletet kell elvégezni a bejövő adatokon, ráadásul ezek függetlenek egymástól (koordináták, színek). A VLIW azt is lehetővé teszi, hogy különböző adatokon egyszerre különböző műveletek is végrehajthatók legyenek. Ehhez fontos, hogy a VLIW utasítások hatékonyan legyenek összeállítva. Természetesen az elágazások, ugró utasítások és nem független műveletek esetén a hatékonyság csökken, legrosszabb esetben az SPU egy ciklusban csak egy utasítást hajt végre. A VLIW utasítások lehetővé teszik hosszabb kifejezések kiszámolását, egymástól függő utasítások végrehajtását is, szemben az SIMD-vel, ahol csak több lépésben (bár párhuzamosan több adaton) lehetséges, pl:

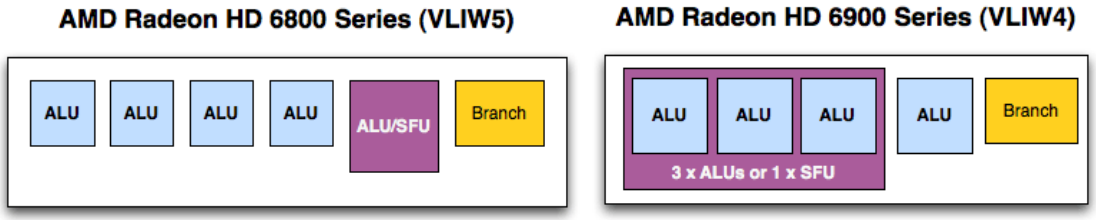
- Dot4 (A=A * B + C * D + E * F + G * H),
- Dual Dot2 (A = A * B + C * D; F = G * h + I * J)
- Dual Dependent Multiplies (A = A * B * C; F = G * H * I)
- Dual Dependent Adds (A = B + C + D; E = F + G + H;)
- Dependent Muladd (A= A * B + C + D * E; F = G * H + I + J * K)

A rendszer úgynevezett wavefront-okat (a másik gyártó warp-nak nevezi) használ a bejövő adatok csoportosítására: egy wavefront 64 feldolgozandó adatot, képpontot vagy vertexet jelent, amelyeken ugyanazokat a műveleteket akarjuk végrehajtani. Egy SIMD mag a 16 SPU-val, négy órajel alatt képes kezelni egy 64 elemű wavefrontot, teljes VLIW utasításokkal, tehát az utasításszintű párhuzamosítással akár teljes képpontok vagy vertexek is képezhetik a wavefrontot, nem csak a komponenseik. A VLIW esetén a (komplex) fordító végezi a

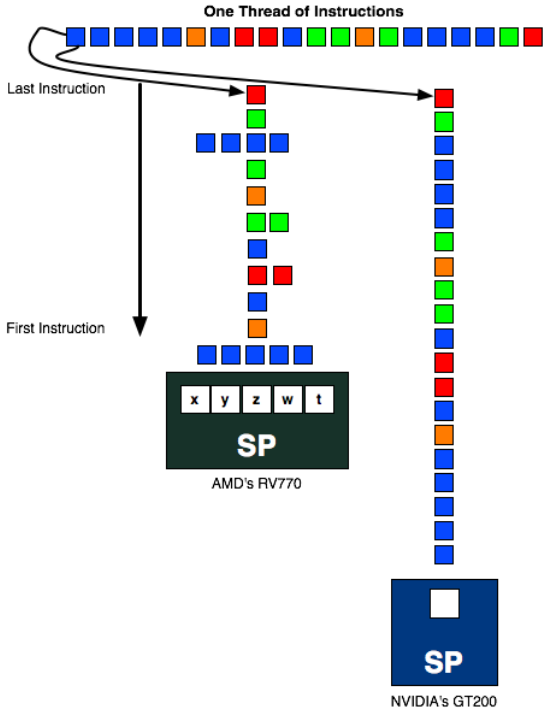
[15] Forrás: https://medium.com/high-tech-accessible/an-architectural-deep-dive-into-amds-terascale-gcn-rdna-gpu-architectures-c4a212d0eb9 (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

párhuzamosítást, melynek előnye, hogy a teljes kódot ismeri, hátránya azonban, hogy nem rendelkezik semmilyen információval az aktuális futásidejű statisztikákról és az adatokról. A gyakorlat azt mutatta, hogy egy átlagos shader program átlagosan 3,4 SP-t használt a rendelkezésre álló négyből vagy ötből.



5.25. ábra – A kihasználtság növelése érdekében a 6900-as sorozatban eltávolították az ötödik SP-t, az ábrán ALU-t [16]



5.26. ábra –A VLIW egység párhuzamosan több utasítást hajt végre, míg a skalár processzor egyszerre egyet. A VLIW utasítások összeállítása a fordító feladata.

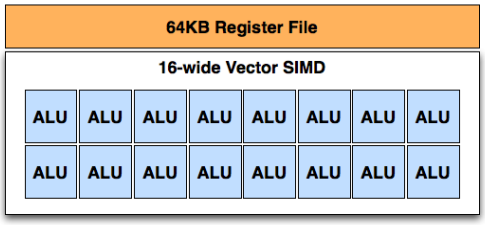
5.5.4 AMD GCN 1.0 (2012)

Bár a VLIW nem volt rossz megoldás grafikai célokra, ám általános számítási célra nem volt ideális, alkalmazástól függően drasztikusan lecsökkenhetett a hatékonyság. A következő alaposabb architekturális frissítés szakított a VLIW alapú utasításszintű párhuzamosítással, és

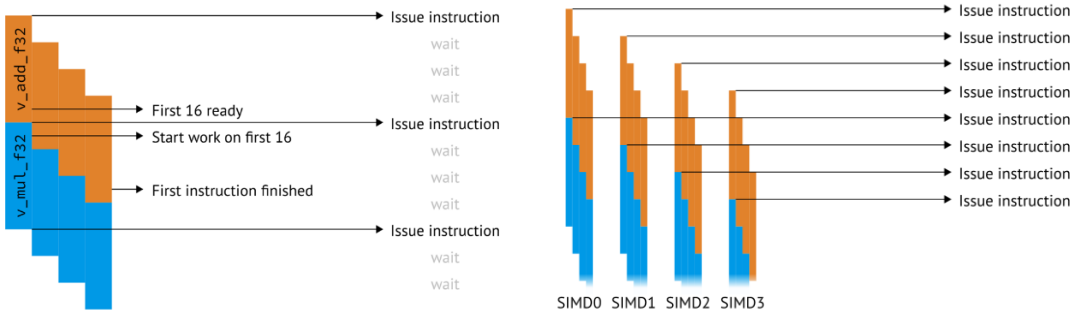
[16] Forrás: https://www.anandtech.com/show/4061/amds-radeon-hd-6970-radeon-hd-6950/4 (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

SIMD alapú végrehajtószál párhuzamosításra váltott. A vektor SIMD blokk egy 64 KB-os regisztertömbből, valamint 16 ALU blokkból épül fel, ezek egy órajel alatt hajtanak végre utasításokat. Egy teljes wavefront feldolgozása így 4 órajelciklusig tart (ugyanazt az utasítást négyszer végzi el 16-16 különböző adaton). Az ALU egységek több különböző egész, valamint lebegőpontos számokkal való műveletet is támogatnak. A grafikai alkalmazásokban nem volt lényeges, de általános célú számítások esetén fontos a 64 bites lebegőpontos típus támogatása. Az első GCN architektúra a 32 biteshez képest akár felezett sebesség mellett képes ilyen típussal műveleteket végrehajtani, egyszerűbb, kisebb GPU variánsok 1/4 vagy 1/16-od sebesség mellett kezeltek duplapontosságú lebegőpontos számokat.



5.27. ábra – Az SIMD blokk 16 adaton hajtja végre ugyanazt az utasítást, egy wavefrontot 4 órajel alatt dolgoz fel



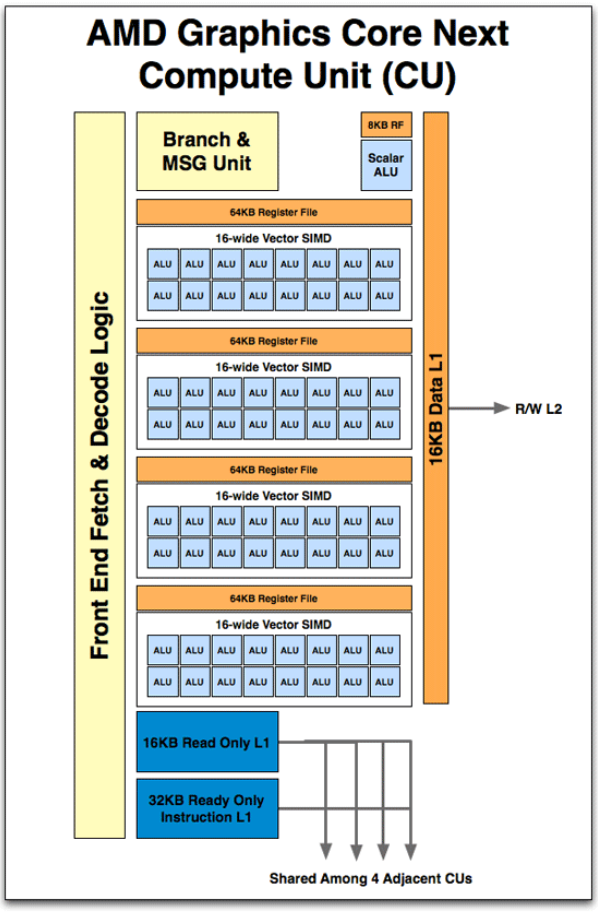
5.28. ábra – Az SIMD blokk 4 órajelenként kaphat új utasítást, 4 párhuzamosan működő SIMD-vel minden órajelben osztható ki új vektor utasítás [17]

A végrehajtás alapegysége a CU (Compute Unit). Egy CU tartalmaz utasításbeolvasó, dekódoló és ütemező logikát, elágazáskezelőt, egy skalár ALU-t saját 8KB-os regisztertömbbel, négy vektor SIMD blokkot – amelynek mindegyike rendelkezik saját 64KB-os regisztertömbbel –, egy gyors megosztott 64KB-s memóriát (Local Data Share – LDS), 16KB-os L1 adat cache-t, valamint a szomszédos CU-kkal közös utasítás cache-t. Grafikus alkalmazásoknál a textúra adatokat is az L1 cache-ben tárolják, azonban hardveresen tömörítve, így még jobb a memória kihasználtsága. A VLIW-vel ellentétben az ütemezés dinamikus, nem a fordító, hanem a CU végzi futás közben, ez természetesen bonyolultabb felépítést és több tranzisztort igényel. A CU képes arra, hogy tetszőlegesen válasszon az ütemezett wavefront-ok közül, melyik feldolgozását melyik SIMD motornak adja. Minden SIMD motornak 10 wavefront-os puffer áll rendelkezésére. Sőt, akár különböző taszkok között is váltogathat, így folyamatosan el tudja látni az SIMD motorokat feladattal, növelve a hatékonyságot és csökkentve az üresjáratokat,

[17] Forrás: https://anteru.net/blog/2018/more-compute-shaders/ (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

amelyek utasítások függőségéből eredhetnek. A különálló skalár ALU célja, hogy a párhuzamos végrehajtás hatékonyságát csökkentő skalár utasításokat (pl. egész műveleteket) hajtson végre, így tehermentesítse a vektor motorokat. A skalár ALU egy utasítást egy órajel alatt hajt végre, ezzel szemben a vektor motorok egy utasítást egy wavefront-on 4 ciklus alatt, így a CU minden órajelperiódusban egy SIMD motort képes ellátni új utasítással. A CU-ban található négy SIMD motor képes négy különböző utasításon dolgozni egyidőben (akárcsak a VLIW), azonban itt minden utasítás egy teljes wavefronton hajtódik végre 4-4 ciklus alatt. A maximális kihasználtság érdekében 1-1 órajel eltolással működnek az SIMD egységek, így minden órajelben befejeződik egy vektor utasítás, miközben ezzel párhuzamosan órajelenként skalár utasítások is végrehajtódnak (ld. 5.28 ábra). Természetesen a végrehajtás hatékonysága nagyban függ az utasítások sorrendjétől és egymásra hatásától. A memóriaműveletek lassúak, ha függés lép fel utasítások között, akkor a rendszernek várakoznia kell (ha tud, akkor másik feladatra vált).

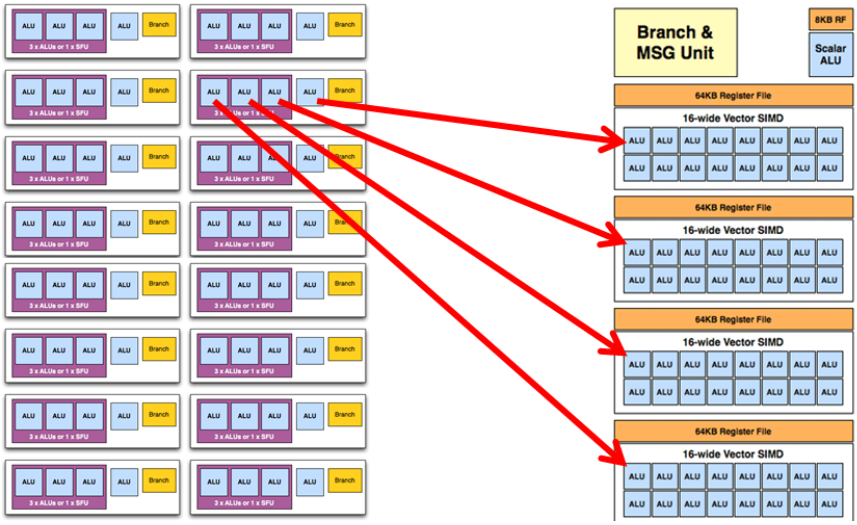


5.29. ábra – A CU tartalmaz négy vektor SIMD-t, egy skalár ALU-t, vektor és skalár regisztertömböket, utasítás betöltőt és dekódert, adat és utasítás cache-eket [18]

[18] Forrás: https://www.anandtech.com/show/4455/amds-graphics-core-next-preview-amd-architects-for-compute/4 (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

Vizsgáljuk meg, hogyan optimalizálta a gyártó a GCN CU-t a VLIW SIMD core-hoz képest: a VLIW SIMD core 16 példányban hajtott végre egy VLIW utasítást, míg a GCN CU-ban ezt szétszedték, átstrukturálták: 4 független 16 elemű vektor motor jelent meg. Összességében továbbra is ugyanannyi utasítást lehet maximálisan végrehajtani – 4 különbözőt 16 adatelemen –, azonban megszűnt a VLIW okozta kötöttség, a 4 SIMD motor sokkal függetlenebbé vált. Kvázi a VLIW utasítások összeállítása a CU végrehajtójába került és nem a fordító, hanem a CU állítja össze futás közben. Ez egy kis egyszerűsítés, hiszen a 4 SIMD motor egy-egy órajel eltolással kap utasítást. Továbbá az új megoldásnak köszönhetően a rövid skalár utasítások nem rontják már a VLIW kihasználtságát, ugyanis külön skalár ALU-t építettek be erre a célra.



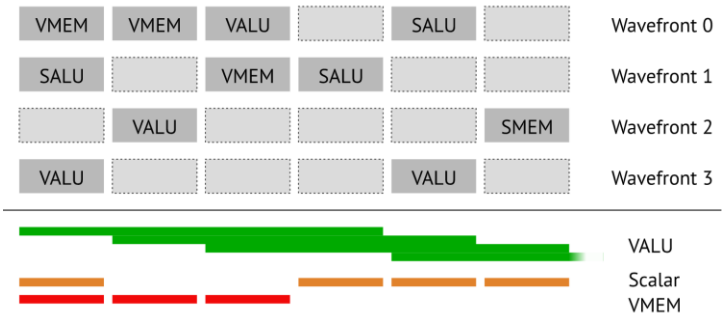
5.30. ábra – VLIW és GCN feldolgozók megfeleltetése, egymás „kifordítottja”

5.5.4.1 GCN Utasításkészlet, végrehajtás

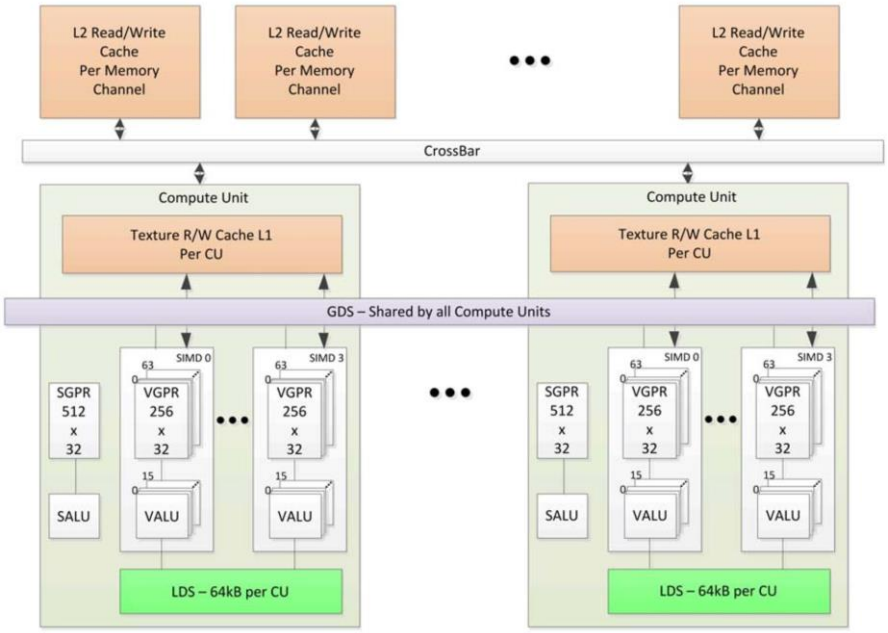
A GPU-ban található egy Graphics Command Processor, illetve több Async Compute Engine (ACE), ezek a drivertől érkező parancsokat fogadják és ütemezik. Előbbi a grafikus parancsokért felel, míg az ACE modulok általános számításhoz kapcsolódó parancsprocesszorok. A DMA modulok a CPU és GPU memória közötti másolásokért felelnek. Az architektúrát párhuzamos munkavégzésre optimalizálták, az ACE modulok felelősek az erőforrás allokációért, kontextusváltásért, valamint a CU-k vezérléséért a feladatok prioritásának figyelembevételével. Az architektúra utasításszinten in-order, tehát a wavefrontokon végrehajtott alacsonyszintű gépi utasítások nem cserélhetőek fel. Az ACE egységek megválaszthatják, hogy a feladatokat (nem az utasításokat) milyen sorrendben küldik el (preemptív, prioritást figyelembe vevő).

A CU-kban hat különböző egység van, melyeket utasításokkal lehet ellátni: elágazáskezelés és üzenet a hosztnak, skalár ALU vagy skalár memória elérés, vektor ALU, vektor memória elérés, helyi adatcsere, illetve globális adatcsere és export. A CU egyszerre akár 5 különböző utasítást is ki tud adni, de minden végrehajtónak órajelenként maximum egyet. Így további utasításszintű párhuzamosítás is lehetővé válik.

Copyright © 2021 / BME AUT



5.31. ábra – 4 wavefront feldolgozás ütemezve, egyszerre akár három utasítás végrehajtása párhuzamosan. A VALU utasítások négy órajelig tartanak SIMD végrehajtónként[19]



5.32. ábra – Regiszterek és megosztott memóriák a GPU-ban [20]

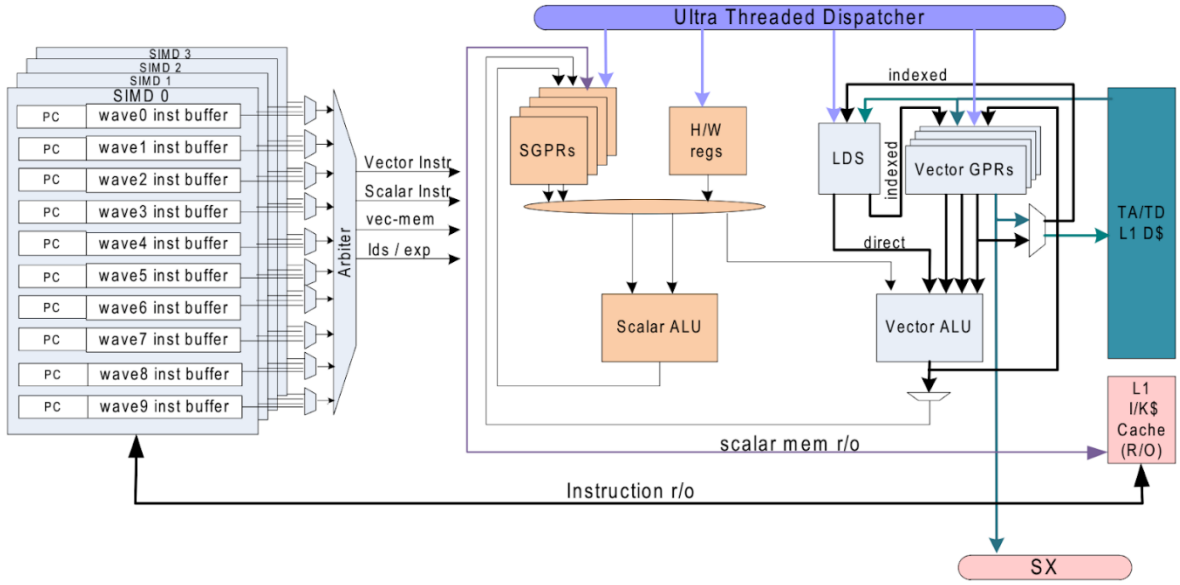
Az utasítások típusukat tekintve a következők lehetnek:

- Skalár ALU: 32 vagy 64 bites bitmezőn vagy egészként értelmezett adatokon végez műveleteket
- Vektor ALU: 32 vagy 64 bites lebegőpontos, egész vagy bitmezőként értelmezett adatokon végez műveleteket egy SIMD blokk egy wavefront mind a 64 elemén. Összehasonlításra is képes vektor regiszterekkel, ekkor az eredmény bitmezőként érhető el.

[19] Forrás: https://anteru.net/blog/2018/even-more-compute-shaders/ (Hozzáférés: 2020.06.29.)
[20] Forrás: Southern Islands Series Instruction Set Architecture (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

- Skalár memória hozzáférés: konstansok olvasására szolgál a konstans cache memóriából az SGPR (skalár regiszterek) területre (pl. ALU, memória puffer, képi konstansok vagy mintavételezőegységek adatainak betöltése).
- Vektor memória hozzáférés: VGPR regiszterek és memória közötti adatmozgatás, általános célú pufferek és képek esetén is használható. Támogat atomi memória műveleteket is, pl. közvetlen memóriatartalom változtatást.
- Data Share: pl. VGPR és LDS memória közötti vektoros adatmozgatás, támogat atomi műveleteket is.
- Programvezérlés és kivételkezelés: skalár műveletek, ugrások, elágazások, kivételek kezelése, végrehajtás szüneteltetése és befejezése.



5.33. ábra – Egy CU-n belül az egyes SIMD végrehajtóknál 10-10 wavefront állhat sorban, a különböző típusú utasítások eltérő részegységeket vezérelnek

Az aktuálisan végrehajtott wavefront állapotát számos regiszter adja meg. A 40 bites **PC** (Program counter) a kernel első utasítására mutat, amikor létrejön a wavefront. Az ugró utasítások (Branch) esetén a cím mindig a következő utasítás kezdőcíméhez relatívan értelmezett.

A vektorutasítások végrehajtásánál az SIMD egységek figyelembe veszik az „**EXEC**” 64 bites maszk regiszter értékét, mely bitenként megadja, hogy az utasítás végre legyen-e hajtva az adott wavefront elemen. Használatával sok esetben elkerülhetőek az ugróutasítások if-then-else blokkokban. A **VCC** regiszter bitjei vektoros összehasonlító utasítás után tartalmazzák az egyes work-item-ekre vonatkozó összehasonlítások eredményét.

Az aktuális wavefronthoz tartozik egy olvasható **Status Register** amelyben több, végrehajtással kapcsolatos információ érhető el:

Copyright © 2021 / BME AUT

- SCC (Scalar Condition Code) a skalár összehasonlítás eredményét tárolja,
- SPI_PRIO: a wavefronthoz rendelt prioritás,
- EXECZ: EXEC regiszter értéke 0,
- VCCZ: VCC regiszter értéke 0,
- IN_TG: az aktuális wavefront mellett vannak-e más wavefrontok is a work-groupban,
- IN_BARRIER: vár-e a wavefront egy barrier-ben,
- illetve több, egyéb hibakezelést segítő mező.

A work-group a work-item-ek olyan csoportja, amin belül elérnek közös lokális memóriát. A barrier egy szinkronizációs lehetőség, a wavefront feldolgozása nem halad tovább, míg a wavefrontban található minden work-item el nem érte a barrier utasítást.

A **Mode Register** a végrehajtással kapcsolatos paramétereket tartalmazza, skalár utasításokkal írható, olvasható:

- FP_ROUND: kerekítés mód float és double típusok esetén (legközelebbi páros felé, + végtelen, - végtelen vagy 0 felé),
- FP_DENORM: denormalizált számok kezelése,
- DX10_CLAMP: NaN-ok nullába állítása vagy sem,
- EXCP_EN: kivételkezelés beállítható kivétel típusonként (nullával osztás, túl-, alulcsordulás, egész 0-val osztás stb.),
- VSKIP: vektor utasítások kihagyása (órajelenként akár 10 wavefront esetén is képes 1-1 vektor utasítást kihagyni),
- illetve egyéb működést befolyásoló paraméterek.

Az **M# Memory Descriptor** regiszter a címzésben segít: LDS (Local Data Store) és GDS (Global Data Store) Load/Store utasításai esetén, illetve indirekt regiszter címképzéshez használható.

A program alapvető vezérlésére szolgáló utasítások a skalár utasítások csoportjába tartoznak, pl.:

- S_ENDPGM: wavefront befejezése, „return” (akár többször is előfordulhat a kódban),
- S_NOP: No OPeration,
- S_SETPRIO: prioritás beállítás,

Copyright © 2021 / BME AUT

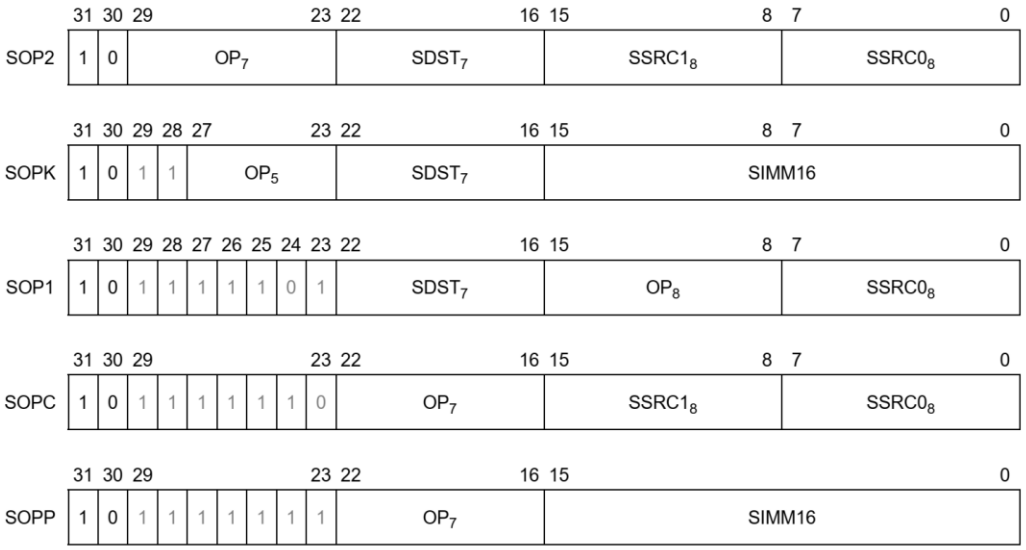
- S_SLEEP: 64-448 órajelig nem futtatja a wavefrontot, a várakozás közben más wavefrontokat kezd végrehajtani,
- S_SENDMSG: üzenet (megszakítás) küldése a hoszt CPU-nak,
- S_TRAP és S_RFE: megszakításkezelőbe ugrás és visszatérés.

Ugyanebbe a kategóriába tartoznak az ugróutasítások, pl.:

- S_BRANCH: feltétel nélküli ugrás,
- S_CBRANCH <test>: ugrás igaz feltétel esetén (VCCZ, VCCNZ, EXECZ, EXECNZ, SCCZ, SCCNZ),
- S_SETPC: egy SGPR (scalar general purpose register) regiszterpárból tölti be PC értékét,
- S_SWAPPC: SGPR regiszterpár és PC értékét megcseréli,
- S_GETPC: regiszterpárba másolja a PC értékét,
- S_SETVSKIP: VSKIP bit beállítása adott értékre, hatékonyan használható vektor utasítások kihagyására ugrás helyett.
- Az S_CBRANCH_FORK és S_CBRANCH_JOIN segítségével összetettebb elágazások, szubrutinok hozhatók létre akár 6 mélységű stack-ben, ekkor minden hívási szint több SGPR regisztert használ fel.
- Az S_BARRIER utasítás szinkronizálási célt szolgál, hatására a wavefront várni fog, míg minden egyéb wavefront a work-group-ban el nem jut ugyanehhez az utasításhoz, majd innen folytatódik az összes végrehajtása. Lassabban végrehajtható utasítások esetén (pl. memória hozzáférés) dedikált számlálók felfelé számolnak az utasítás indítása esetén és lefelé, amikor befejeződnek.
- Az S_WAITCNT utasítással megvárható, míg az összes kiadott memória írás vagy olvasás befejeződött. Az utasítások átlapoltan, típusonként eltérően, akár sorrenden kívül hajtódnak végre.

A skalár ALU utasítások egy skalár értéken végeznek műveletet egy wavefronton belül, elsősorban 32 bites egész vagy 32 és 64 bites bitenkénti műveletek elvégzésére szolgálnak. Az utasításokat 32 bites szavak kódolják, hozzáférnek a működést leíró regiszterekhez (Mode, Status, SGPR, M0, PC stb.), maszkokhoz, utasításba kódolt konstansokhoz, azonban az LDS területhez és a vektor (VGPR) regiszterekhez nem. 32 bites konstansok az utasítás adatfolyamba lehetnek beágyazva az utasítások után.

Copyright © 2021 / BME AUT



5.34. ábra – Skalár ALU utasítások különböző kódolásai, látható milyen egységes a felépítésük (OP: opcode, SDST: cél skalár regiszter (SGPR), SSRC0,1: első és második forrásoperandus, SIMM16: előjeles 16 bites utasításba kódolt konstans egész).
Az ISA dokumentációja tartalmazza részletesen az egyes utasítások leírását.

Az egyes műveletek forrásoperandusát 8 bites kód azonosítja, mely egy előre definiált táblázat alapján különböző értékeket jelent: az alsó 104 érték SGPR cím, majd a speciális regiszterek (VCC, M0, EXEC_LO, EXEC_HI stb.), valamint gyakran használt konstansok (pl. egész 0, 1..64, –1..–16), majd gyakran használt lebegőpontos értékek (0.5, –0.5, 1.0, –1.0, 2.0, –2.0, 4.0, –4.0), 0xFF esetén pedig a következő 32 bites érték az utasítás memóriában. A célcímek kódolása is megegyezik, azonban csak az alsó 128 érték használható, a konstansok a felső 128 bájt címen találhatóak. Az aritmetikai utasítások 32 bites előjeles és előjel nélküli egészeken végezhetők. Tartalmaz összeadást, kivonást, szorzást, maximum, minimum kiválasztást, abszolút értéket, előjel kiterjesztést 8 és 16 bitről 32 bitre. A szorzásnak és összeadásnak van egy-egy speciális változata, ahol a második operandus az utasításba ágyazott 16 bites egész. A carry az SCC (scalar condition code) bitbe kerül, illetve az összehasonlító utasítások is az SCC bitet állítják. Összehasonlítás elvégezhető két regiszter között, vagy egy regiszter és egy 16 bites utasításba ágyazott konstans között. Egy 32 vagy 64 bites mező adott bitjének értéke tesztelhető nullára vagy egyre.

A bitenkéti műveletek csoportjában található a másolás, valamint az alapvető logikai műveleteken túl negált operandussal vagy negált eredménnyel operáló utasítások is. Egy utasításban elvégezhető tetszőleges számú bit eltolás, vagy bitmező kiolvasása is (adat, melyik bittől hány bit). Továbbá van bitenkénti forgatás, kiolvasható az egyes és nullás bitek száma, az első egyes vagy nulla pozíciója, lehetőség van közvetlenül egy bitet beállítani stb. A HW_ID regiszter tartalmaz még információt az aktuális futásról: melyik sorban álló wavefront fut (0..9), melyik SIMD motor, CE hajtja végre, stb.

Copyright © 2021 / BME AUT

		Code	Meaning	
Scalar Source (8 bits)	**Scalar Dest (7 bits)**	0 - 103	SGPR 0 to 103	Scalar GPRs.
		104	reserved	
		105	reserved	
		106	VCC_LO	vcc[31:0].
		107	VCC_HI	vcc[63:32].
		108	TBA_LO	Trap handler base address, [31:0].
		109	TBA_HI	Trap handler base address, [63:32].
		110	TMA_LO	Pointer to data in memory used by trap handler.
		111	TMA_HI	Pointer to data in memory used by trap handler.
		112-123	ttmp0 to ttmp11	Trap handler temps (privileged). {ttmp1,ttmp0} = PC_save{hi,lo}.
		124	M0	Temporary memory register.
		125	reserved	
		126	EXEC_LO	exec[31:0].
		127	EXEC_HI	exec[63:32].
		128	0	Immediate (constant value 0).
		129-192	int 1 to 64	Positive integer values.
		193-208	int -1 to -16	Negative integer values.
		209-239	*reserved*	*unused*
		240	0.5	single or double floats
		241	-0.5	
		242	1.0	
		243	-1.0	
		244	2.0	
		245	-2.0	
		246	4.0	
		247	-4.0	
		248-250	*reserved*	*unused*
		251	VCCZ	{ *zeros*, VCCZ }
		252	EXECZ	{ *zeros*, EXECZ }
		253	SCC	{ *zeros*, SCC }
		254	*reserved*	
		255	Literal constant	32-bit constant from instruction stream.

5.35. ábra – A forrás- és céloperandusok 8, illetve 7 bites kódolásának értelmezése

Vektor utasítások 32 és 64 bites kódolásban is előfordulnak, némelyik mindkettőben. Az utasítások elérik a VGPR és SGPR regisztereket, tartalmazhatnak utasításba ágyazott konstansokat, utasításmemóriában elhelyezett teljes 32 bites konstansokat, valamint az LDS területről is olvashatnak. Az utasítások operandusainak címe a skalárhoz hasonlóan kódolt, de 9 bites, a további 256..511-es címeken a VGPR tartomány is elérhető. Minden vektor utasítás végrehajtódik egy wavefront mind a 64 elemén.

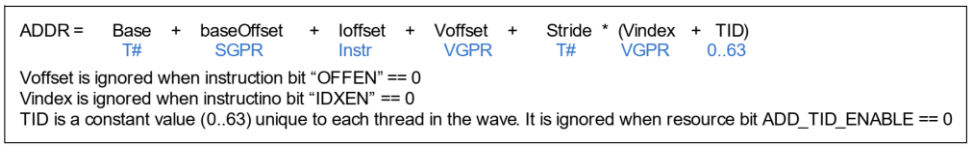
A skalár memóriahozzáférés utasításokkal lehet az SGPR regiszterekbe tölteni adatot a memóriából, puffer erőforrásból (egyszerre akár 1, 2, 4, 8 vagy 16 db 32 bites érték is tölthető), érvényteleníthető a cache vagy egy időzítő-számláló is olvasható. Ezek az adatok a wavefront minden eleme számára elérhetőek.

Copyright © 2021 / BME AUT



5.36. ábra – Skalár memóriaolvasás utasítás formátuma (OP: utasítás kód, SDST: cél SGPR regiszterek kezdőszáma, több adat olvasása esetén páros vagy néggyel osztható címnek kell lenni, SBASE: azon SGPR-ek kezdőszáma, ahol a forrás 64 bites memóriacím, IMM: ofszet értelmezése, OFFSET: IMM-től függően egy 8 bites cím eltolás az utasításba kódolva vagy egy SGPR cím, ami cím ofszetet tartalmaz)

Vektor memóriaműveletekkel lehet minden work-item számára egy adatot írni vagy olvasni a VGPR vagy LDS tartományba vagy tartományból. Az utasításokat különböző felépítésű 64 bites utasításkódok írják le. Az utasítások része, hogy mely VGPR regiszterek tartalmazzák a szükséges címeket, mely VGPR-ek tartalmazzák a forrást vagy a célt, valamint mely SGPR regiszterek tartalmaznak memória puffer leírókat. Az 5.37 ábrán látható az összetett címképző logika felépítése. Kép típusú műveletek esetén textúra szűrés is történik, ekkor egy összetett puffer leíróra is szükség van a regiszterekben, mely megadja, hogy pontosan milyen formátumban tartalmazza a memória a képet, hogy ennek megfelelően történhessen meg a mintavételezés és szűrés. A báziscím utáni különböző ofszetek lehetővé tesznek például képekben vízszintes tengely mentén (soron belül) eltolásokat, ezek az eltolások jöhetnek skalár regiszterből, utasításba ágyazva vagy vektor regiszterből is. A Stride érték a megcímzendő memóriaterület sorhosszát jelöli ki, a stride többszörösével történő cím eltolás sor eltolást tesz lehetővé. Itt opcionálisan még a work-item (thread) ID is használható. A bázis cím és a stride pufferleíróban található SGPR területen.



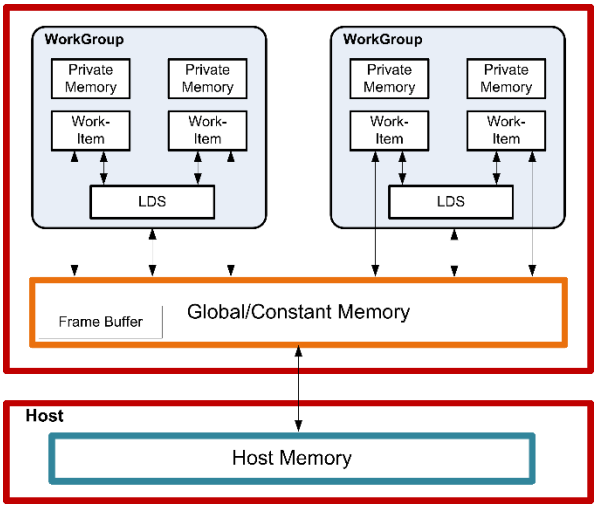
5.37. ábra – A rendszer bonyolult címképzővel rendelkezik, de erre szükség is van az összetett adattípusok, pl. 2D képek eléréséhez

5.5.4.2 GCN memóriahierarchia

CU-n belüli nagysebességű, kis késleltetésű memória az LDS (Local Data Share), mely külön utasításokkal érhető el. Írható és olvasható, a globális memóriánál legalább egy nagyságrenddel gyorsabb. Egy work-group-on belül minden work-item közösen használhatja. Kapacitása 64KB minden CU-ban, az LDS 32 bankba van csoportosítva és minden bank 512x32 bites két portos memória. Órajelenként így 32-32 db párhuzamos 32 bites írás és olvasás, vagy 32 párhuzamos 64 bites írás vagy olvasás is elvégezhető. Az LDS tartalmaz 32 egész atomi egységet is a gyors atomi művelet-végrehajtáshoz.

A GPU rendelkezik 64KB globális közös megosztott memóriával (GDS Global Data Share). Ez szintén 32 bankból áll, bankonként 512x32 bites memória, valamint szintén tartalmaz 32 atomi egységet. A GDS-t bármelyik CU eléri, de sok konkurens hozzáférés esetén lassú, várakozások alakulhatnak ki.

Copyright © 2021 / BME AUT



5.38. ábra – Privát, LDS, globális és hoszt memória hierarchia[21]

Az általános célra használható GPU-k máig tartalmaznak fix grafikai funkciós egységeket, melyeket nem lenne hatékony általános funkciókkal implementálni. A ROP (raster operations pipeline, render output unit) blokkok felelnek a raszterizálásért: mintavételezik a képernyő koordinátákba vetített háromszögeket, meghatározzák a pixelek végső színét, az eredményt pedig a kiválasztott képpufferbe írják. A TMU (texture mapping unit) blokkok felelnek textúrák címzéséért és szűréséért, a shadereket szolgálják ki textúra adatokkal.

Egy-egy GPU család több tagból áll a piaci igények lefedése érdekében. A GPU-k jól skálázhatóak, a párhuzamos végrehajtók számának meghatározásával befolyásolható a szükséges lapka mérete, így az ára is. A GCN alapú GPU-kat elsősorban a CU-k száma, a memóriainterfész szélessége és egyéb kiegészítő blokkok száma határozza meg. Processzorba integrált GPU esetén 3-8-11 CU jellemző, dedikált GPU-k változó számú, 8-60 közötti CU-t tartalmazhatnak.

A maximális számítási kapacitás meghatározása a következő módon lehetséges: minden CU-ban 4 SIMD motor található, melyek órajelenként 16 MAD utasítás elvégzésére képesek. 32 CU egység esetén a GPU 2048 MAD utasítás kibocsátásra képes órajelenként. 1GHz-es órajel frekvenciánál ez 4,096 TFLOPS-nak felel meg (floating point operations per second), a MAD utasítást ugyanis két lebegőpontos műveletnek tekintik.

[21] Forrás: https://rocmdocs.amd.com/en/latest/GCN_ISA_Manuals/testdocbook.html (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

5.5.4.3 GCN Assembly példák

```
float fn0(float a, float b)
{
   if (a>b)
      return (a*a-b);
   else
      return (b*b-a);
}
```

```
// Registers r0 contains "a", r1 contains "b"
// Value is returned in r2

    v_cmp_gt_f32      r0, r1           // a>b
    s_mov_b64         s0, exec         // Save current exec mask
    s_and_b64         exec, vcc, exec  // Do "if"
    s_cbranch_vccz    label0           // Branch if all lanes fail
    v_mul_f32         r2, r0, r0       // result = a * a
    v_sub_f32         r2, r2, r1       // result = result - b
label0:
    s_not_b64         exec, exec       // Do "else"
    s_and_b64         exec, s0, exec   // Do "else"
    s_cbranch_execz   label1           // Branch if all lanes fail
    v_mul_f32         r2, r1, r1       // result = b * b
    v_sub_f32         r2, r2, r0       // result = result - a
label1:
    s_mov_b64         exec, s0         // Restore exec mask
```

5.39. ábra – Példa C kód és GCN assembly megfelelője vektorosan #1

Az 5.39. ábrán látható példakódban *r0* és *r1* regiszterek az adott szálhoz tartozó *a* és *b* értéket tartalmazza (ezek a C függvény bemenő paraméterei). A *v_cm_gt_f32* utasítás elvégzi *r0* és *r1* regiszterek összehasonlítását vektorosan, az eredmény VCC-be kerül (vector condition code). A skalár *s0* regiszternél kezdődő területre elmenti az aktuális 64 bites EXEC maszkot, majd ÉS kapcsolatba hozza az újonnan kapott VCC-vel. Ha VCC nulla volt, azaz egyik work-item-re (szálra) sem teljesül a feltétel, elugrik *label0* címkére. Ha VCC nem nulla volt, végrehajtja a szorzás és kivonás műveleteket azokon a vektorelemeken, ahol teljesült a feltétel. Label0 címke után az EXEC maszkot negálják és ÉS kapcsolatba hozzák a korábban elmentett eredeti EXEC maszkkal, így a vizsgált feltétel ellentettje fog teljesülni azoknál a vektorelemeknél, ahol engedélyezve volt a végrehajtás. Ha itt egyik elemre sem kell végrehajtani a kiértékelést, a program elugrik *label1* címkéhez és visszaállítja az eredetileg elmentett EXEC maszkot. Ha legalább egy vektorelemen végre kell hajtani a műveletet, elvégzi a szorzást és a kivonást, majd ugyanúgy visszaállítja az eredeti EXEC maszkot. Az eredmény *r2* regiszterbe kerül.

Copyright © 2021 / BME AUT

```
float fn0(float a,
          float b,
          float c){
   float d=0.0;
   for(int i=0;i<100;i++)
   {
      if(c>113.0)
         break;
      c=c*a+b;
      d=d+1.0;
   }
   return d;
}
```

```
//Registers r0 contains "a", r1 contains "b",
//          r2 contains "c"  r3 contains "d"
//   Value is returned in r3
v_mov_b32 r3,#0.0 //float d=0.0
s_mov_b64 s0,exec //Save execution mask
s_mov_b32 s2,#0 //i=0
label0:
s_cmp_lt_s32 s2,#100 //i<100 (SCC=i<100?1:0;
s_cbranch_sccz label1 //Exit loop if not true
v_cmp_le_f32 r2,#113.0 //c>113.0  VCC=c>113?0:1
s_and_b64 exec,vcc,exec //Update exec mask on fail,
amik nagyobbak, azokra nem hajtjuk végre
s_branch_execz label1 //Exit if all lanes pass
v_mad_f32 r2,r2,r0,r1 //c=c*a+b r2=r2*r0+r1
v_add_f32 r3,r3,#1.0 //d=d+1.0
s_add_s32 s2,s2,#1 //i++
s_branch label0 //Jump to start of loop
label1:
s_mov_b64 exec, s0 //Restore exec mask
```

5.40. ábra – Példa C kód és GCN assembly megfelelője vektorosan #2

Az 5.40. ábrán látható másik példakódban a C függvény bemenetei (*a*, *b*, *c*) az (*r0*, *r1*, *r2*) regiszterekben érhetők el. A *d* lokális változó az *r3* regiszterbe kerül. Első lépésként a program beállítja *r3* értékét lebegőpontos 0.0-ra. Az *s_mov_b64* skalár utasítással elmenti az *s0-s1* skalár regiszterpárba a 64 bites EXEC maszk értékét. A ciklusváltozót *s2*-ben tárolja, kezdeti értéke 0 lesz *s_move_b32* utasítással. Mivel a ciklusváltozó közös minden work-item-re, skalár regiszterben is tárolható. Skalár összehasonlító utasítással megvizsgálja, hogy a ciklusváltozó kisebb-e, mint 100. Ha kisebb, SCC értéke 1 lesz, ha nagyobb vagy egyenlő, akkor 0. Ha SCC értéke 0, akkor elugrik a kód végére, ahol visszaállítja az elmentett EXEC maszkot. Ha nem értünk a ciklus végére, vektoros utasítással megvizsgálja, hogy *c* (*r2*) kisebb vagy egyenlő-e 113-nál. Ha *c* nagyobb, akkor 0 kerül az aktuális work-item indexnek megfelelően a VCC 64 bites regiszterbe, ha kisebb vagy egyenlő, akkor 1. Skalár ÉS (*s_and_b64*) utasítással frissíti az EXEC maszkot, tehát innentől kezdve már csak azokra a work-item-ekre alkalmazza az utasításokat, amelyeknél *c* kisebb vagy egyenlő volt 113-nál. Ha a végrehajtási maszk 0, akkor nem maradt több feldolgozandó work-item, ugrik a kód végére. Amennyiben legalább egy work-item-re még érvényes a végrehajtás, elvégzi a vektoros műveleteket: szorzás és összeadás (*v_mad_f32*), *d* lebegőpontos növelése 1.0-val (*v_add_f32*). A skalár ciklusváltozó növelése után a végrehajtás visszaugrik a ciklus elejére a ciklusváltozó vizsgálatához.

5.5.5 AMD GCN variánsok

A GCN architektúra több frissítésen esett át. A GCN 2-vel javítottak az energiahatékonyságon, növelték a memória sávszélességét. Több végrehajtóegység és ACE került a GPU-kba.

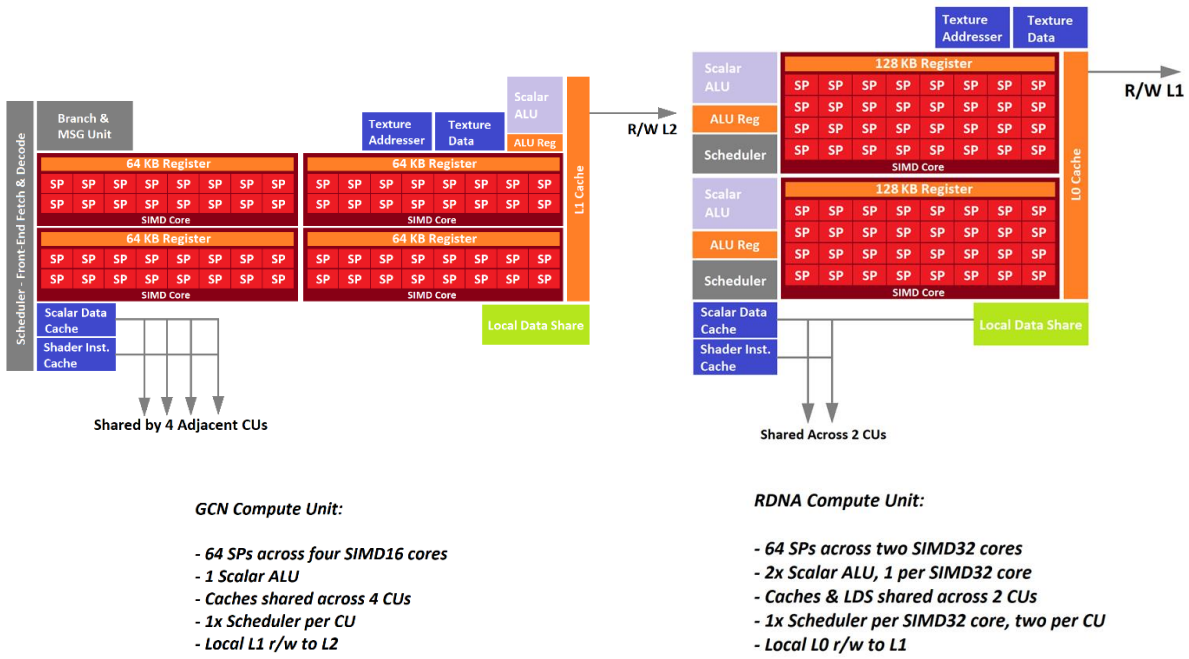
A harmadik generációs GCN-ben javult a taszkváltási sebesség, megjelent az FP16 felezett pontosságú lebegőpontos típus támogatása, azonban ennek végrehajtási sebessége megegyezett az FP32-vel, csak kevesebb memóriát igényelt. Megjelent a hardveres veszteségmentes textúratömörítés támogatása, amellyel tovább növelhető a memória sávszélesség hatékonysága. Ennek a GPU-nak már megjelent HBM (High Bandwith Memory) memóriával szerelt változata is.

Copyright © 2021 / BME AUT

A GCN4-ben (Polaris) tovább csökkentették a fogyasztást, javítottak az utasítás-betöltésen: a rendszer próbálja előre meghatározni a szükséges utasításokat és előtölti azokat a megnövelt utasítás-gyorsítótárba.

Az ötödik generációs GCN (Vega) tette lehetővé a dupla sebességű FP16-os számítást, az SIMD vektor motorok órajelenként 16 db helyett 32 db FP16 művelet végrehajtására lettek képesek. Megjelent a primitive shaderek támogatása (új shader lépcső, mellyel a hatékonyság jelentősen növelhető, a nem látható háromszögek korai törlését teszi lehetővé), javult a terhelés elosztása GPU-n belül. HBM2 memóriával szerelt változatok is jelentek meg.

5.5.6 AMD RDNA (Navi)



5.41. ábra – balra GCN, jobbra RDNA CU [22]

A gyártónál a GCN architektúrát az RDNA váltotta le 2019-ben, amiben több módosítást is eszközöltek a hatékonyság javítása érdekében. A legfontosabb változás, hogy a wavefront méretét csökkentették, 32 szálból áll 64 helyett, de a támogatás 64-re is megmaradt. Két-két SIMD vektoros végrehajtót összeolvasztottak, 4x16 helyett 2x32 eleműek lettek (a vektor regiszterek mérete is duplázódott), így nem szükséges négy órajel utasításonként, egy alatt is fel tudnak dolgozni egy teljes 32 elemű wavefrontot. Minden SIMD motornak saját ütemezője lett. Egy CU-n belül két skalár ALU jelent meg, minden SIMD blokkhoz kapcsolódott egy. A CU egységek párosával egy WGP (Workgroup Processor) egység alá kerültek, ezek osztoznak

[22] Forrás: https://medium.com/high-tech-accessible/an-architectural-deep-dive-into-amds-terascale-gcn-rdna-gpu-architectures-c4a212d0eb9 (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

skalár adat és utasítás gyorsítótáron, valamint LDS-en. Továbbá új gyorsítótárakat is bevezettek, jelentősen növekedett a hatékonyság.

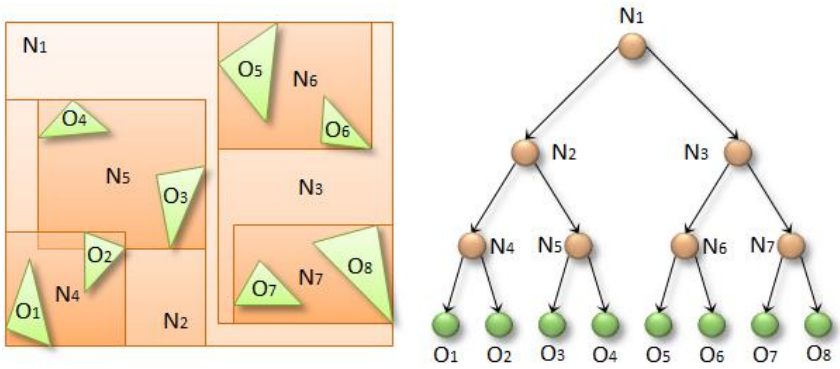
5.5.7 GeForce RTX 2080 Turing

A 2019-ben bemutatott architektúra számos újdonságot tartalmaz. Új fix funkciós blokkok jelentek meg: RT magok a sugárkövetés támogatásáért, Tensor mag neurális hálós alkalmazásokhoz.

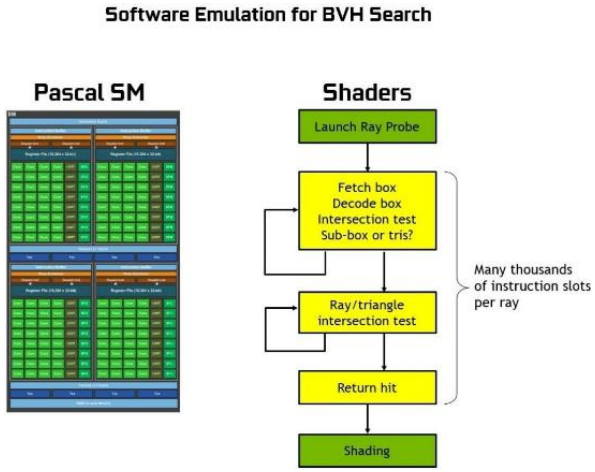
A raszterizálás alapú számítógépes grafika egyértelműen jó útnak tűnt limitált számítási kapacitás esetén, azonban az egyre valósághűbb megoldások számítási igénye is folyamatosan növekedett, lassan közelíti a sugárkövetéshez szükségeset. A sugárkövetés megvalósításához nem ideálisak a GPU-k jelenlegi architektúrái, így a gyártó a GPU-ba épített sugárkövetést támogató blokkokat, mellyel hibrid grafikai csővezeték valósítható meg: a raszterizálás kiegészítése sugárkövetéssel meghatározott adatok alapján. Így valósághűbb fény-árnyék hatások érhetők el, az objektumok egymásra szórt fénye, tükröződések, fénytörések pontosabbak lesznek.

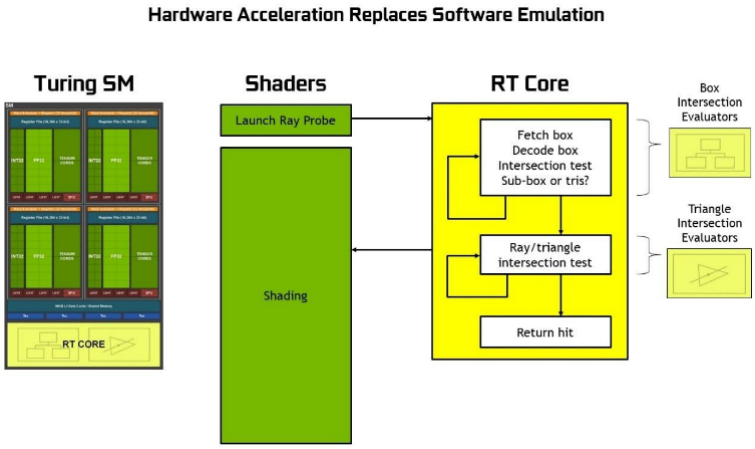
Az RT magok elsődleges feladata meghatározni egy sugár esetén, hogy van-e metszéspontja egy sugárnak és háromszögnek, és ha igen, hol. Az algoritmus nem működne hatékony adatstruktúra nélkül. A GPU drivere a háromszöghálóhoz egy optimalizált BVH (Bounding Volume Hierarchy) reprezentációt készít, amelyben a teret hierarchikusan egyre kisebb részekre osztja fel. A sugárkövetés során nem az összes háromszöget vizsgálja, hanem megkeresi az első szintű befoglaló térrészt, amit elmetsz a sugár. Majd ennek a befoglaló térrésznek a következő szintű részeiből azt, amellyel először találkozik a sugár, majd így tovább, gyorsítva a keresést ahhoz hasonlóan, ahogy a bináris keresés működik egy rendezett listán. Amikor megfelelően kevés jelöltre sikerült leszorítani a lehetséges metsző háromszögeket, megkísérli a pontos metszést a háromszögekkel. Az RT magok az SM (Streaming Multiprocessor) magok mellé kerültek egyfajta koprocesszorként. Az SM magok indítják a metszések keresését, azonban amíg az RT magok dolgoznak, az SM magok nem várnak, folytathatják a hagyományos shader kód futtatását.

Copyright © 2021 / BME AUT



5.42. ábra – Bounding Volume Hierarchy [23]





5.43. ábra – Sugárkövetés a korábbi Pascal architektúrán (felül) és az újabb Turing-on (alul), a dedikált RT mag gyorsítja a számolást [24]

[23] Forrás: https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/ (Hozzáférés: 2020.06.29.)
[24] Forrás: https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/ (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

A tenzor magok kifejezetten neurális hálós alkalmazásokhoz kerültek be a GPU-ba. Elsősorban tenzor vagy mátrix műveletek nagyon hatékony végrehajtására képesek. Megjelent az INT8 és INT4 típusok támogatása, gyakran a half (FP16) pontosságra sincs szükség már betanított hálók futtatásához. FP16 pontosság mellett órajelenként 64 multiply-add vagy 128 egyszerű művelet végrehajtására képesek. INT8 esetén 128, INT4 esetén 256 műveletet tud elvégezni egyszerre. Egy SM 8 tenzor magot tartalmaz, így a fenti számok egy SM-re vetítve is komoly számítási kapacitást jelentenek.

A TU102-es (GeForce 2080 Ti) GPU variáns összesen 72 SM-et tartalmaz, minden SM 64 CUDA maggal, 8 Tenzor maggal, egy RT maggal, egy 256KB-os regiszter tömbbel, 96KB-os megosztott memóriával rendelkezik, és természetesen a raszterizáláshoz kapcsolódó blokkok is megtalálhatóak benne. Az RT magok képesek 10 milliárd sugár metszését meghatározni másodpercenként. A Streaming Multiprocessor mag esetén az egész és lebegőpontos feldolgozó teljesen szét lett választva, így képes párhuzamosan mindkét típusú műveletet elvégezni. Az egész műveletek, például az összeadás címképzéshez hasznos és jól párhuzamosítható akár a grafikai lebegőpontos műveletekkel.

5.5.8 AMD RDNA 2

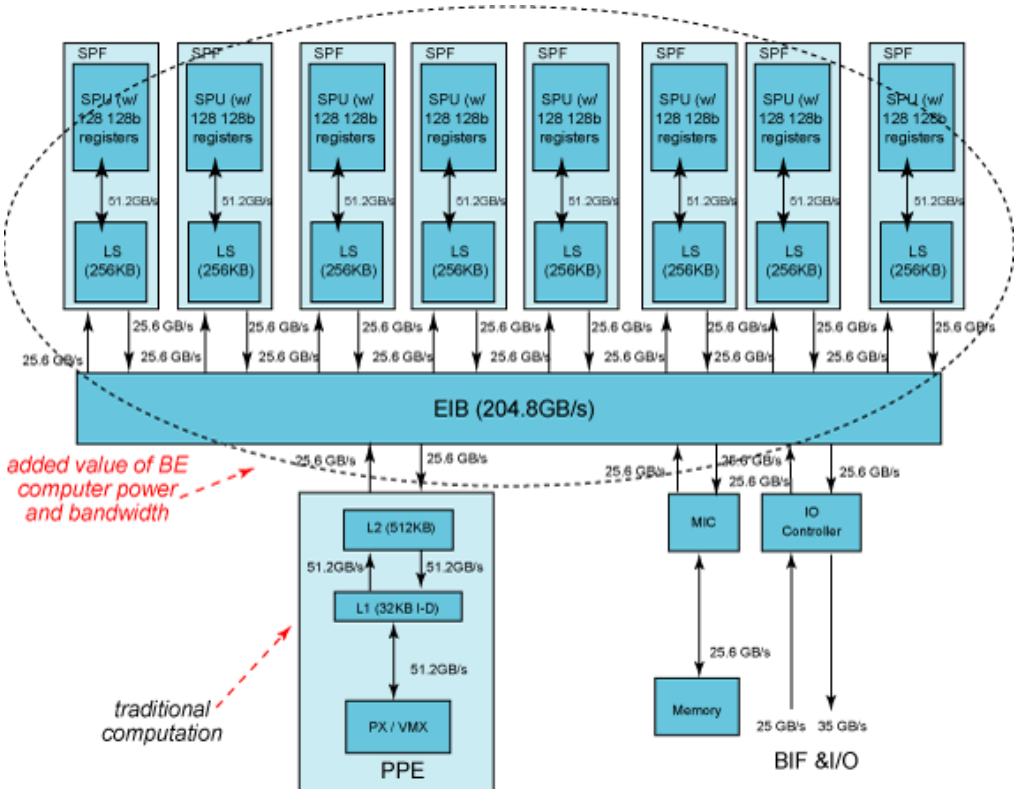
Az AMD Radeon 6000-es sorozatában (valamint a Playstation 5 és az Xbox Series X processzorában) is megjelent a sugárkövetés hardveres támogatása. A konkurens gyártó megoldásától eltérően nem dedikált RT magokat implementáltak, hanem a meglévő CU-kat egészítették ki sugár-box és sugár-háromszög metszés számolást gyorsító hardverrel (Ray Accelerator), valamint ennek megfelelően egy-két új utasítás is bekerült a végrehajtóegységekbe sugárkövetési céllal (IMAGE_BVH_INTERSECT_RAY, IMAGE_BVH64_INTERSECT_RAY). A sugárkövetést, BVH bejárást a CU hajtja végre programozott módon, a hardveres gyorsító csak a metszések meghatározását végzi. A dedikált hardveres megoldás hatékonyabb, azonban a programozott változat univerzálisabb, nagyobb szabadságot biztosít a fejlesztők számára. A metszést számoló utasítások paraméterei a VGPR regiszterekben adhatók meg, a BVH adatstruktúra pedig a központi memóriában helyezkedik el. A feladat nagy memória sávszélesség-igényű, ezen a problémán a GPU-ba épített nagy kapacitású (128MB) és nagysebességű újabb (L3) gyorsítótár hivatott enyhíteni.

5.5.9 Cell processzor

A felsorolt gyártókon és architektúráikon túl léteztek egyéb fejlesztések is, amelyek több-kevesebb sikert értek el a GPU piacon. A Sony, a Toshiba és az IBM közös fejlesztéseként készült a Cell processzor 2007-ben, mely a Playstation 3 központi egységeként ért el sikereket. A processzor tartalmazott egy általános célú PowerPC alapú magot (PPE: Power Processing Element), valamint nyolc hatékony vektoros koprocesszort (SPE: Synergistic Processing Element). A PPE 64 bites, két szál futtatására képes, 23 lépéses csővezetékkel rendelkezett, limitáltan volt képes out-of-order (OoO) végrehajtásra, tartalmazott szimpla és dupla pontosságú FPU-t, mely órajelenként két dupla vagy nyolc szimpla pontosságú lebegőpontos művelet elvégzésére volt képes 3,2GHz-es órajelen. Ez a processzor futtatta az operációs rendszert és vezérelte az SPE-ket. Az SPE-k szintén kétutas processzorok, elágazásbecslés és

Copyright © 2021 / BME AUT

OoO nélkül. Saját utasításkészlettel rendelkeztek és 128 bites SIMD szervezésben végeztek műveleteket négy szimpla vagy két dupla pontosságú, valamint 16x8 bit, 8x16 bit, 4x32 bites egész típusokon órajelenként. Az SPE-k 256KB dedikált SRAM memóriával rendelkeztek, de nem voltak képesek közvetlenül hozzáférni a központi memóriához. A Cell processzor erősen támaszkodott a fordító szoftverre optimális kódgenerálás érdekében, nem tartalmazott elágazás becslő egységet, de tartalmazott olyan utasítást, amellyel a fordító figyelmeztette, hogy elágazás várható. A processzor egységei gyűrűs kialakítású EIB-n (Element Interconnect Bus) keresztül kommunikáltak.



5.44. ábra –A hagyományos PPE processzort nyolc SPE koprocesszor segíti[25]

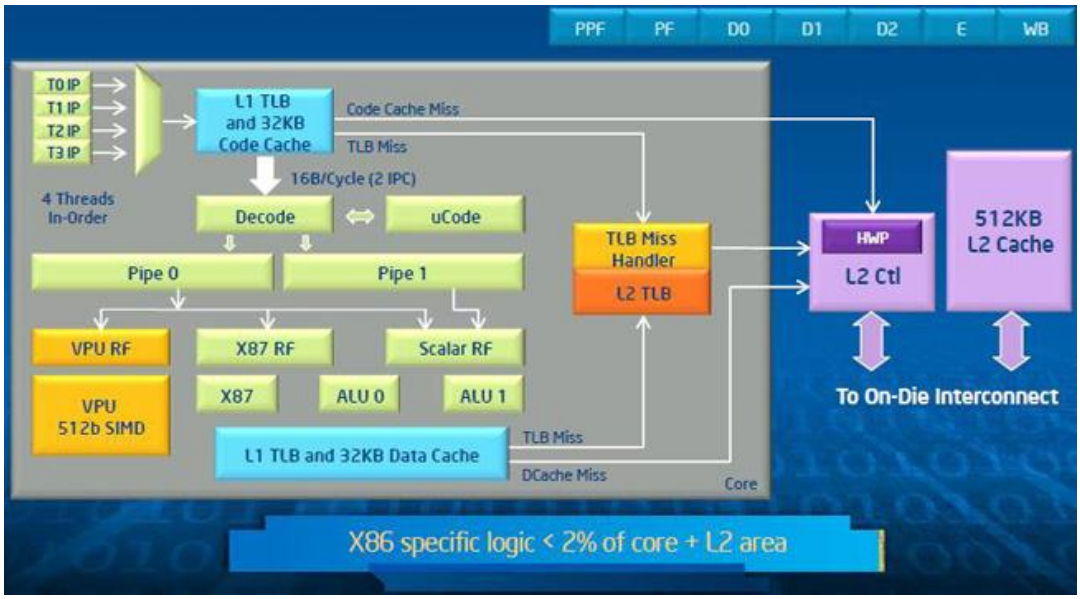
5.5.10 Intel Xeon Phi

2009 környékén az Intel próbálkozott egy ambiciózus tervvel, több (8-48) kisebb Pentium processzor integrálásával. A projekt kódneve Larrabee volt, azonban végül nem valósult meg GPU-ként, de Xeon Phi gyorsító koprocesszorként megjelent a piacon. Az eredeti tervek szerint egy hibrid CPU-GPU jött volna létre, olyan teljes értékű általános processzorokkal, amelyek önmagukban is képesek operációs rendszer futtatására, de tartalmaznak SIMD és GPU specifikus kiegészítéseket, hogy GPU-ként is használható legyen, akár valósidejű sugárkövetéshez is. A mag alapját egy P5 alapú (P54C) szuperskalár architektúra adta, azonban OoO logika nélkül, ami a kisebb méret eléréséhez volt szükséges. Minden mag négy szál

[25] Forrás: https://www.ibm.com/developerworks/library/pa-cellperf/index.html (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

végrehajtására volt képes időosztásban, minden regisztert négy példányban tartalmazott. Rendelkezett egy 512 bites vektor végrehajtóval, mely ekkor négyszer szélesebb volt az elterjedt x86 SSE egységeknél, így akár 16 egyszeres pontosságú lebegőpontos számon is tudott műveleteket végrehajtani. A magok tartalmaztak egy textúra mintavételező egységet grafikai alkalmazásokhoz, az egységek 1024 bit széles buszon kommunikáltak egymással és a memóriával. A processzor magok teljesen gyorsítótár koherensen működtek. A termék végül Xeon Phi bővítőkártyaként jelent meg, szuperszámítógépekben alkalmazták. A legutolsó változat 72 magot tartalmazott, 1500-1600MHz-es órajelre volt képes, 384GB DDR4 memóriával. Nem terjedtek el széles körben, így a szoftveres támogatásuk sem volt kiváló, habár támogatták az OpenMP és OpenCL keretrendszereket.



5.45. ábra –A Xeon Phi magjai x86 alapúak, egyszerre négy szálat kezelnek, a vektoros SIMD egység 512 bit széles (Intel)

5.6 Szoftver

A felhasználói alkalmazások szabványos API-kon keresztül kommunikálnak a gyártói eszközillesztővel, melyek a parancsokat a megfelelő módon továbbítják a grafikus kártyák felé. Grafikus célokra a legelterjedtebb a Microsoft zárt DirectX felülete, valamint a Khronos Group által gondozott, nyílt OpenGL. A legújabb API-k egyszerűbbé, „vékonyabbá” váltak, több feladat megvalósítása került át a felhasználói alkalmazásokba, amelyeket korábban a driverekben, API-kban implementáltak, ezáltal nagyobb rugalmasságot és több optimalizálási lehetőséget biztosítanak. Ilyen alacsony szintű API-k a DirectX 12 és a Vulkan, utóbbi az OpenGL-t váltja le.

Általános célú számítási keretrendszerek közül a legelterjedtebbek az nVidia CUDA platformja (mely csak a gyártó saját eszközeit támogatja), valamint a szintén Khronos Group által

Copyright © 2021 / BME AUT

fejlesztett OpenCL (mely széleskörű támogatást élvez, GPU-kon, CPU-kon is futtatható, valamint más platformokon, akár FPGA-n végrehajtható kód is generálható vele).

5.6.1 Vertex és pixel shaderek

A DirectX API csomag 8-as változata vezette be a programozható pixel és vertex shadereket (1.0-1.4), programozásuk HLSL nyelven történt (High Level Shader Language). A DirectX 9 bevezette a Shader Model 2.0 szintet, mely meghatározott követelményrendszert állított a gyártók elé, hogy szabványos kód készüljön, amely garantáltan futtatható a szabványt teljesítő GPU-kon. A szabványosítás hátránya, hogy az egyes architektúrák egyedi jellemzői, előnyei elveszhetnek és nem képesek a gyártó számára valós fölényt biztosítani, így hátráltatva egyedi, optimálisabb megoldásokat. A két folyamat váltakozása jól megfigyelhető a GPU API-k fejlődése során. A további újabb Shader Model generációk egyre több limitációt oldottak fel, egyre általánosabb (több típus, konstans, utasítás, regiszter támogatása) és bonyolultabb kódok (hosszabb kódok, ciklusok, elágazások) futtatását tették lehetővé, elsősorban grafikai alkalmazásokhoz.

A HLSL shader program szintaktikailag hasonló egy C nyelvű kódhoz, azonban számos kiegészítést (pl. textúra mintavételezők, matematikai függvények, vektor, mátrix műveletek...) és típust (float3, float4, half, float4x4…) tartalmaz az egyedi alkalmazás miatt. A HLSL kód vagy az alacsonyabb szintű shader assembly gyártófüggetlen assembly bájtkóddá fordul, majd az alkalmazás futtatásakor a gyártó eszközmeghajtója fordítja és optimalizálja az adott GPU architektúra gépi kódjára. Ez a módszer lehetővé teszi, hogy ugyanaz az alkalmazással szállított kód bármely, az adott modell szintet támogató GPU architektúrán futtatható legyen, valamint a forráskód ne legyen közvetlenül elérhető. A következőkben egy példakód látható egy vertex és egy pixel shader HLSL kódjára. A minták csak szemléltetésként jelennek meg, nem célunk alaposan elemezni a kód működését. A változó- és függvénydeklarációk után szerepelhetnek előre definiált kulcsszavak, ezek biztosítják, hogy a megfelelő értékekkel töltse fel a változók tartalmát a rendszer. A vertex shader számos különböző adatot állít elő a kimenetén, amelyet a pixel shader megkap és hasznosít. A mátrixszorzáshoz használt *mul()* függvény is közvetlenül használható. A globálisan definiált változók értékét a rendszer inicializálja. Példa vertex shader HLSL-re:

```
float4x4 view_proj_matrix;
float4 view_position;
float4 light0;
float4 light1;
float4 light2;
struct VS_OUTPUT
{
 float4 Pos : POSITION;
 float3 View : TEXCOORD0;
 float3 Normal: TEXCOORD1;
 float3 Light1: TEXCOORD2;
 float3 Light2: TEXCOORD3;
 float3 Light3: TEXCOORD4;
};
VS_OUTPUT main( float4 inPos : POSITION,
```

Copyright © 2021 / BME AUT

```
    float3 inNorm : NORMAL )
   {
    VS_OUTPUT Out = (VS_OUTPUT) 0;
    // Output transformed vertex position:
    Out.Pos = mul( view_proj_matrix, inPos );
    Out.Normal = inNorm;
    // Compute the view vector:
    Out.View = normalize( view_position - inPos );
    // Compute vectors to three lights from the current vertex position:
    Out.Light1 = normalize (light0 - inPos); // Light 1
    Out.Light2 = normalize (light1 - inPos); // Light 2
    Out.Light3 = normalize (light2 - inPos); // Light 3
    return Out;
   }
```

A shader assembly változatában láthatjuk, hogy a gépi utasítások mennyire igazodnak a célokhoz (*mul, mad, dp4* stb). A bejövő globális „paraméterek” (*view_proj_matrix c0, c1, c2, c3, view_position c4…*) címkékkel érhetők el, a vertexek pozíciója és normálja *v0, v1*–gyel érhető el. A 4x4 mátrix és 4x1-es vektor szorzása 4 utasítással valósítható meg. Példa vertex shader assembly-re:

```
   // Parameters:
   // float4 light0;
   // float4 light1;
   // float4 light2;
   // float4 view_position;
   // float4x4 view_proj_matrix;
   //
   // Registers:
   // Name Reg Size
   // ---------------- ----- ----
   // view_proj_matrix c0 4
   // view_position c4 1
   // light1 c5 1
   // light2 c6 1
   // light0 c7 1
    vs_1_1
    dcl_position v0
    dcl_normal v1
    mul r0, v0.x, c0
    mad r2, v0.y, c1, r0
    mad r4, v0.z, c2, r2
    mad oPos, v0.w, c3, r4
    add r1, -v0, c4
    dp4 r1.w, r1, r1
    rsq r1.w, r1.w
    mul oT0.xyz, r1, r1.w
    add r8, -v0, c7
    dp4 r8.w, r8, r8
    rsq r8.w, r8.w
    mul oT2.xyz, r8, r8.w
    add r3, -v0, c5
    add r10, -v0, c6
```

Copyright © 2021 / BME AUT

```
 dp4 r3.w, r3, r3
 rsq r3.w, r3.w
 mul oT3.xyz, r3, r3.w
 dp4 r10.w, r10, r10
 rsq r10.w, r10.w
 mul oT4.xyz, r10,
```

A speciális hardver egységek, mint pl. a textúra mintavételező szoftveres leképezése is megfigyelhető: *sampler* típus, *tex1D* vagy *tex2D* függvény. A képpont adatai a bemenő paramétereken keresztül érkeznek, a címkék alapján tudja a rendszer, melyik változónak milyen értéket kell átadni. Példa HLSL pixel shader-re:

```
float4 Material;
sampler Outline;
float4 main( float3 View: TEXCOORD0,
 float3 Normal: TEXCOORD1,
 float3 Light1: TEXCOORD2,
 float3 Light2: TEXCOORD3,
 float3 Light3: TEXCOORD4 ) : COLOR
{
 // Normalize input normal vector:
 float3 norm = normalize (Normal);

 float4 outline = tex1D(Outline, 1 - dot (norm, normalize(View)));
 float lighting = (dot (normalize (Light1), norm) * 0.5 + 0.5) +
 (dot (normalize (Light2), norm) * 0.5 + 0.5) +
 (dot (normalize (Light3), norm) * 0.5 + 0.5);
 return outline * Material * lighting;
}
```

A DirectX 9-cel megjelent Shader Model 2.0 sok opcionális lehetőséget biztosított, például a pixel shaderek esetén elegendő volt a 24 bites pontosság, szemben a vertex shaderek 32 bites pontosságával, valamint létezett partial precision opció, amely 16 bites pontosságot várt el, annak ellenére, hogy az ábrázolás 32 biten történt. A shader kód limitált, maximum 96 utasításból állhatott. A DirectX 10-zel jelent meg a Shader Model 4.0, amelyben bevezették a geometry shader-t, valamint a Unified Shader architektúrát. A hardverben már nem különültek el az egyes shader feldolgozók, univerzálisan végrehajthattak vertex, pixel vagy geometria feldolgozót, optimálisan alkalmazkodva az alkalmazás igényeihez. Az egész típusok és bitenkénti logikai műveletek teljesen támogatottá váltak, így megoldódtak a memória-, illetve tömbcímzési anomáliák, amelyeket a lebegőpontos számítások eredménye okozhatott. A HLSL kód objektum-orientálttá vált, bevezették a compute shader típust kifejezetten GPGPU alkalmazásokhoz. Az alapvető változásokat hozó 12-es verzió új Shader Model-eket is bevezetett, folyamatosan frissült és kapott új funkciókat, megjelent a DirectX Raytracing, amely lehetővé teszi szabványos módon a hardveres vagy szoftveres sugárkövetés implementációjának használatát.

Copyright © 2021 / BME AUT

	DX8 SM1.x	DX9 SM2	DX9 SM3	DX10
Vertex Instructions	128	256	512	64k
Pixel Instructions	4+8	32+64	512	
Vertex Constants	96	256	256	16x4096
Pixel Constants	8	32	224	
Vertex Temps	16	16	16	4096
Pixel Temps	2	12	32	
Vertex Inputs	16	16	16	16
Pixel Inputs	4+2	8+2	10	32
Render Targets	1	4	4	8
Vertex Textures	n/a	n/a	4	128
Pixel Textures	8	16	16	
2D Tex Size			2k x 2k	8k x 8k
Int Ops	-	-	-	✔
Load Op	-	-	-	✔
Derivatives	-	-	✔	✔
Vertex Flow Control	n/a	Static	Static/Dyn	Dynamic
Pixel Flow Control	n/a	n/a	Static/Dyn	

5.46. ábra – Egyre összetettebb kódok váltak megvalósíthatóvá az újabb verziókkal: egyre nagyobb regiszterterület állt rendelkezésre, egyre hosszabb shader kód használata vált megvalósíthatóvá, az elvégezhető műveletek típusa is bővült

A DirectX API-val párhuzamosan a nyílt OpenGL is fejlődött. A shaderek leírása a szintén C-szerű GLSL nyelven történt. A hordozhatóság érdekében a GLSL forrásokat az alkalmazással szállították (mint string) és a gyártó eszközillesztője felelt a gépi kódra történő fordításért. A speciális nyelvi elemek szintaktikája eltér a HLSL-től, pl. a be- és kimenő paraméterek definiálása esetén. Példa GLSL vertex shader-re:

```
#version 330 core
layout (location = 0) in vec3 aPos;  //the position variable has attrib position 0
layout (location = 1) in vec3 aColor;//the color variable has attribute position 1

out vec3 ourColor; // output a color to the fragment shader

void main()
{
gl_Position = vec4(aPos, 1.0);
ourColor = aColor; // set ourColor to the input color we got from the vertex data
}
```

Példa GLSL pixel shader-re:

```
   #version 330 core
   out vec4 FragColor;
   in vec3 ourColor;

   void main()
   {
```

Copyright © 2021 / BME AUT

```
        FragColor = vec4(ourColor, 1.0);
    }
```

5.6.2 OpenCL

Az OpenGL-ben is megjelentek a compute shaderek, és egyre inkább elérhetővé vált az általános számítású felhasználás, majd a Khronos Group létrehozta a specializált OpenCL API-t (CL: Compute Library), mely az nVidia CUDA vagy a Microsoft-féle DirectCompute számára biztosított nyílt konkurenciát. Az OpenCL nem csak GPU-kon, de akár CPU-kon is futtatható, kihasználva a speciális vektor műveletvégző utasítás bővítményeket (SSE, AVX), beágyazott SoC-k is támogatják, vagy akár FPGA hardverleíró is generálható belőle. A C-szerű forrás SPIR (Standard Portable Intermediate Representation) köztes kódra fordul, majd ezt fordítják tovább az adott architektúrára a gyártói illesztők. Az OpenGL-t váltó Vulkan API is már ezt a megközelítést alkalmazta, SPIR-V köztes nyelvre fordulnak a shader kódok.

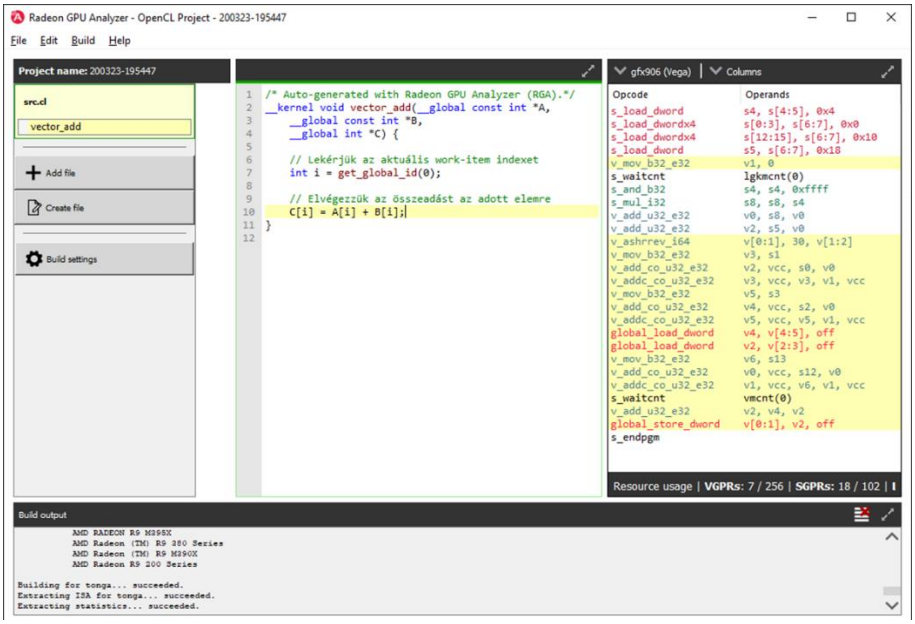
Az OpenCL kódokat kernelnek hívják, melyek párhuzamosan sok adaton hajtódnak végre. Természetesen a CPU-n futó hívó programnak számos egyéb feladata van, mielőtt a feldolgozást a GPU-nak adná át. Megkeresi és kiválasztja a használandó OpenCL eszközt (CPU, esetleg több GPU is elérhető lehet platformtól függően), létrehozza a forrás és cél adatpuffereket a központi memóriában, majd létrehozza az OpenCL által felügyelt reprezentációkat, platformtól függően elvégzi a másolást közöttük. Betölti a kernel kódot forráskód vagy bináris formában, ütemezi a kernelt, majd az eredmények visszaolvasását, végül felszabadítja az erőforrásokat. Példa OpenCL kernel egy egyszerű vektoros összeadásra:

```
__kernel void vector_add(__global const int *A, __global const int *B, __global
int *C) {

        // Lekérjük az aktuális work-item indexet
        int i = get_global_id(0);

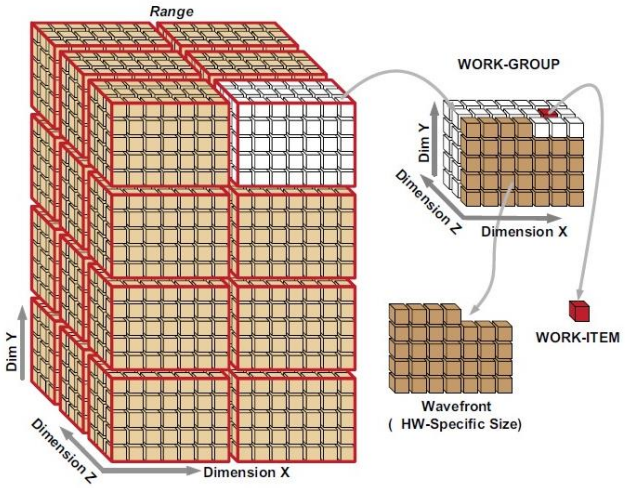
        // Elvégezzük az összeadást az adott elemre
        C[i] = A[i] + B[i];
    }
```

Copyright © 2021 / BME AUT



5.47. ábra – A Radeon GPU Analyzer programmal vizsgálhatjuk C-ben írt OpenCL kernelek vagy Vulkan shaderek lefordított kódját különböző architektúra variánsokra különböző fordítási paraméterek esetén

Az OpenCL egy számítási „szálat”, egységet work-item-nek hív, melyek local_work_size méretű work-group-okba vannak rendezve, így végzi a számítások ütemezését. Work-group-on belül az egyes work-item-eket feldolgozó kernelek közös helyi memóriával (local memory) rendelkeznek, adatot cserélhetnek, valamint helyi atomi műveletek is optimálisan használhatók. Lokális helyett globális memória használata konzisztencia problémákkal vagy alacsonyabb sebességgel járhat együtt.



5.48. ábra – Work-item csoportosítási szintek [26]

[26] Forrás: AMD APP SDK, OpenCL User Guide

Copyright © 2021 / BME AUT

A kernel kód a paraméterlistában kapja meg a létrehozott puffereket, mint mutatók. A bemeneti adatok jellemzően tömbökként helyezkednek el a memóriában. Struktúrák esetén a felhasználóra van bízva, hogy az egyes struktúrákat helyezi egymás után, vagy a struktúrák egyes mezői, mint tömbök helyezkednek el folytonosan a memóriában. A GPU-k nagy központi memóriával rendelkeznek, azonban ehhez képest kevés lap fér el a belső lokális gyorsítótárakban, így nagyon figyelni kell, melyik megoldás az előnyösebb memóriahasználat szempontjából. A végrehajtási sebesség drasztikusan lecsökkenhet, ha a GPU-nak sokat kell várnia az adatok betöltésére. Ilyenkor másik work-group végrehajtására vált, amennyiben elérhetőek a számoláshoz szükséges adatok. Az természetesen alkalmazásfüggő, hogy tényleg szüksége van-e ugyanannyi adatra a forrás pufferekben, ahány work-item-et ütemezünk.

A korábban bemutatott GCN 1.0 architektúra esetén egy CU egy kernel kódot tud futtatni, nagyobb mennyiségű adat esetén több CU is futtathatja ugyanazt a kódot, természetesen más adatokon. Egy work-group több wavefrontot is tartalmazhat (GCN 1.0 esetén maximum 16 wavefront kerülhet egy work-group-ba, azaz 1024 work-item), de biztosan egy CU-n fog futtatni a helyi adatmegosztás érdekében. A CU-kban lévő 4 db 16 széles SIMD vektormotor négy órajel alatt végez 64 adattal, ez a wavefront mérete. Minden SIMD motor 10 PC és wavefront regiszterrel és rendelkezik, így 10 wavefront áll sorban a kód végrehajtásának különböző pontjaiban, egy CU-ban így összesen 40 wavefront várakozhat végrehajtásra.

A kernel számára alapvető fontosságú, hogy pontosan melyik work-item-et hajtja végre éppen. A *size_t get_global_id (uint dimindx)* függvény visszatér a globális work-item sorszámmal, míg a *size_t get_local_id (uint dimindx)* a work-group-on belüli azonosítót adja. Ezek az értékek használhatók a bemeneti vagy kimeneti memóriák indexeléséhez, az egyes work-item-ekhez tartozó adatok címzéséhez. A paraméterként megadható dimenziószám többdimenziós bemeneti adatstruktúrák esetén használható (pl. 2D képek képpontjainak feldolgozása esetén). Az összes ütemezett work-item száma (dimenziónként) a *size_t get_global_size(uint dimindx)* függvénnyel kapható meg. Mivel ez híváskor ismert érték, akár paraméterként vagy konstansként is átadhatjuk a kernelnek. Amennyiben a kernel ütemezése nem történik meg egy puffer minden elemére és mégis szükségünk van a puffer méretére, figyeljünk, hogy ne a *get_global_size*-t használjuk!

A következő részben röviden áttekintünk egy egyszerű hoszt kódot, amely meghívja az OpenCL kernelt.

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define MAX_SOURCE_SIZE (0x100000)

int main(void) {
    // Létrehozunk két (A és B) puffert a hoszt memóriában, ez a bemenet
    int i;
```

Copyright © 2021 / BME AUT

```
        const int LIST_SIZE = 1024;
        int *A = (int*)malloc(sizeof(int)*LIST_SIZE);
        int *B = (int*)malloc(sizeof(int)*LIST_SIZE);
        // Feltöltjük adatokkal
        for(i = 0; i < LIST_SIZE; i++) {
            A[i] = i;
            B[i] = LIST_SIZE - i;
        }

        // Betöltjük a kernel forrást fájlból, de lehetne akár konstans sztringként
vagy köztes bináris kódban is tárolni
        FILE *fp;
        char *source_str;
        size_t source_size;

        fp = fopen("vector_add_kernel.cl", "r");
        if (!fp) {
            fprintf(stderr, "Failed to load kernel.\n");
            exit(1);
        }
        source_str = (char*)malloc(MAX_SOURCE_SIZE);
        source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
        fclose( fp );

        // Lekérdezzük a platform és az eszköz adatait
        cl_platform_id platform_id = NULL;
        cl_device_id device_id = NULL;
        cl_uint ret_num_devices;
        cl_uint ret_num_platforms;
        cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
        ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
                &device_id, &ret_num_devices);

        // Létrehozunk egy OpenCL kontextust, kiválasztjuk az eszközt
        cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL,
&ret);

        // Létrehozunk egy parancssort
        cl_command_queue command_queue = clCreateCommandQueue(context, device_id,
0, &ret);

        // Allokálunk eszközoldali puffereket (két bemenet és egy kimenet)
        cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                LIST_SIZE * sizeof(int), NULL, &ret);
        cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                LIST_SIZE * sizeof(int), NULL, &ret);
        cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                LIST_SIZE * sizeof(int), NULL, &ret);

        // Átmásoljuk a bemenő adatokat az OpenCL eszköz memóriájába
        ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,
                LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
        ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
                LIST_SIZE * sizeof(int), B, 0, NULL, NULL);
```

Copyright © 2021 / BME AUT

```
    // Betöltjük a kernelt
    cl_program program = clCreateProgramWithSource(context, 1,
            (const char **)&source_str, (const size_t *)&source_size, &ret);

    // Lefordítjuk a kernelt
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // Példányosítjuk a kernelt
    cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);

    // Beállítjuk a kernel argumentumait (eszköz oldali pufferek)
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

    // Elindítjuk a kernelt
    size_t global_item_size = LIST_SIZE; //Teljes listán végrehajtjuk
    size_t local_item_size = 64; //64-es csoportokban
    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
            &global_item_size, &local_item_size, 0, NULL, NULL);

    // Hoszt oldalon foglalunk memóriát az eredménynek (C) és visszaolvassuk
    int *C = (int*)malloc(sizeof(int)*LIST_SIZE);
    ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(int), C, 0, NULL, NULL);

    // Kiírjuk az eredményt
    for(i = 0; i < LIST_SIZE; i++)
        printf("%d + %d = %d\n", A[i], B[i], C[i]);

    // Felszabadítjuk az erőforrásokat
    ret = clFlush(command_queue);
    ret = clFinish(command_queue);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(a_mem_obj);
    ret = clReleaseMemObject(b_mem_obj);
    ret = clReleaseMemObject(c_mem_obj);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);
    free(A);
    free(B);
    free(C);
    return 0;
}
```

5.6.3 CUDA

A CUDA platformot (Compute Unified Device Architecture) az nVidia dolgozta ki saját termékei általános számítási képességeinek hasznosítására. Zárt megoldás, csak nVidia termékekkel használható. Előnye, hogy kifejezetten ezekhez a termékekhez igazították, azonnal

Copyright © 2021 / BME AUT

implementálásra kerülnek új hardveres funkciók, valamint az OpenCL előtt jelent meg, így nagyobb a felhasználói bázisa, több szoftver használja, valamint a gyártónak is több ideje volt finomítani a megoldást. Az OptiX keretrendszeren keresztül sugárkövetési feladatok valósíthatók meg, illetve a keretrendszer kihasználja az RT maggal rendelkező termékeinek a sugárkövetést gyorsító képességeit is. A cuDNN könyvtár neurális hálós feladatok megoldásához kínál keretrendszert, mely szintén kihasználja termékeinek hardveres gyorsító képességeit.

A CUDA és az OpenCL keretrendszer egyes esetekben eltérő nevezéktant használ, de jól megfeleltethetők egymásnak:

CUDA	OpenCL
Multiprocessor (SM)	Compute Unit (CU)
Scalar core / Stream processor	Processing Element
Global Memory	Global Memory
Shared memory	Local Memory
Local Memory	Private Memory
Kernel	Kernel
Block	Work-group
Thread	Work-item
Warp	Wavefront

5.2. táblázat – CUDA és OpenCL elnevezések

5.6.4 Esettanulmány, példák

A számolási feladatok párhuzamosítása egyszerű, ha a feldolgozandó adatok között nincs függés: minden adatelemre ugyanazokat az utasításokat kell végrehajtani, és a kimenetek száma megegyezik a bemenetek számával. Ilyen típusú feladatok egyszerű képfeldolgozási feladatok, pl. képpontonkénti fényesség vagy kontraszt állítás. Egy work-item lehet egy képpont, de akár egy egész sor is. Utóbbi esetben a kernel egy ciklust tartalmaz, mely a sor minden képpontján végrehajtja a módosítást. A work-item kifejezés inkább a feldolgozandó adat koncepcióját, míg a thread magát a feldolgozás folyamatát hangsúlyozza ki, de lényegét tekintve ugyanarról van szó.

Nem független adatok esetén a párhuzamosíthatóság összetettebb kérdés: ha több, változó számú bemenet és ezek értékei, részeredmények határozzák meg a kimeneteket, vagy ha a lefutás erősen függ az adattartalomtól. Egy olyan egyszerű feladat esetén, mint egy vektor elemeinek összegzése, az adatok között erős függés figyelhető meg, hiszen az összes adatelemre szükség van a kimenet meghatározásához. Ilyen feladat esetén használható a párhuzamos redukció algoritmus, melyet az említett példán keresztül szemléltetünk.

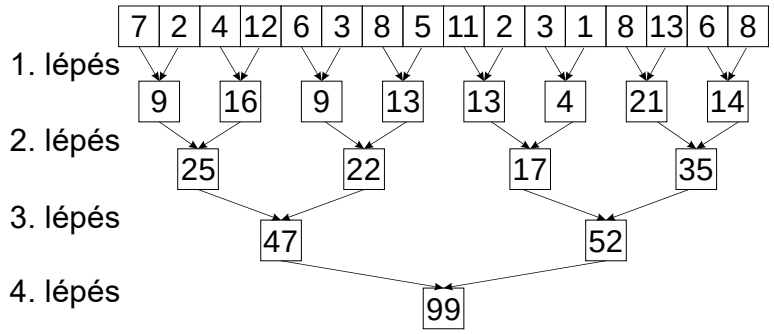
Amennyiben a kimenetek száma is függ a bemenetek értékeitől, további megfontolásokkal élhetünk – a feladattól függően. Amennyiben meghatározható egy felső (elfogadható) korlát a kimeneti elemek számára, lehetőség van ennek megfelelően a maximális memóriaméret lefoglalására. Amennyiben ez nem tehető meg, akár atomi inkrementációkkal meghatározható

Copyright © 2021 / BME AUT

elemenként az írandó kimeneti elem címe, vagy akár több lépésben is futtatható az algoritmus. Első lépésként a szükséges minimális számolással meghatározható a kimeneti elemek maximális száma, lefoglalható a memória, majd végül ténylegesen elvégezhető a kimeneti értékek kiszámolása.

Fejlesztés során minden esetben készítsünk naiv, soros implementációt hagyományos CPU-ra, amiről biztosan tudjuk, hogy helyes eredményt ad. A párhuzamosított megoldást fejlesztés közben minden esetben hasonlítsuk össze a referencia megoldással tesztadatok használatával. Érdemes CPU-ra olyan változatot is fejleszteni, ami hasonló módon értékelhető ki, mint a kernelek a work-item-ekre.

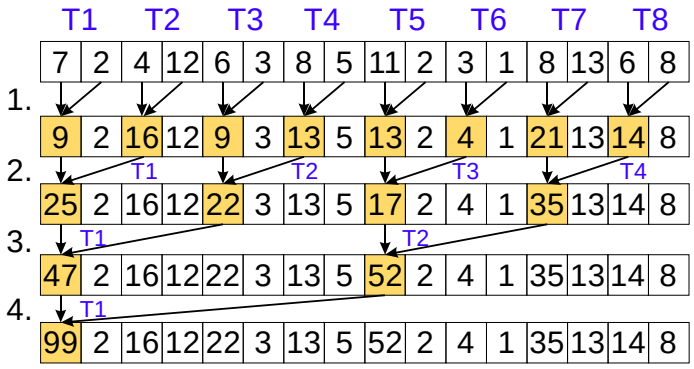
Tekintsük a következő, már korábban említett példát: összegezni akarjuk egy vektor elemeit, kihasználva a párhuzamos végrehajtás lehetőségét. Egyszerűsítésképpen legyen a vektor hossza kettő hatványa. A naiv referencia implementáció egyszerű lesz, az összeg változóhoz egyenként hozzáadjuk a vektor elemeit. A feladat gyorsítható fa gráf alapú párhuzamos redukcióval. Első lépésként meghatározhatóak részösszegek, melyek egyenként kizáró elempárok összegei. Következő lépésként a részösszegek páronkénti újabb összegei határozhatók meg, és így tovább, míg végül egy érték marad. A megoldás $O(\log_2 N)$ lépésben kiszámolható elegendően sok párhuzamos számolóegységgel.



5.49. ábra – Párhuzamos redukció

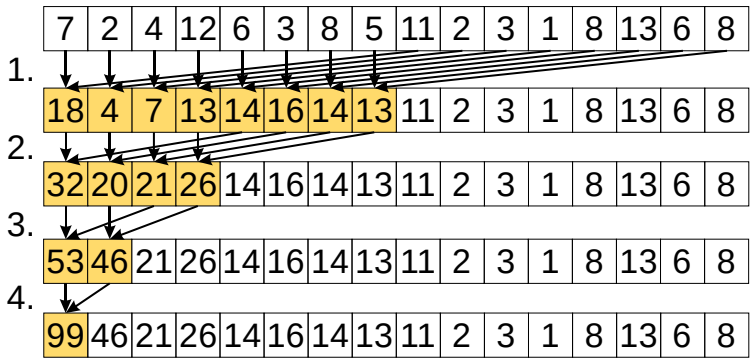
Összefésült vagy átlapolt (interleaved) adatstruktúra esetén szomszédos párokon végezzük el az összeadásokat, a részeredmények a párok első elemének helyére írhatók. Ez a megoldás nem hatékony memóriaelérés tekintetében, mert a gyakran használt címek egymástól távol helyezkednek el, a cache nem lesz jól kihasználva, a globális memória elérése lassú. Az 5.50 ábrán kiemelten láthatók a módosított cellák, a Tx jelölés pedig az egyes párhuzamosan működő összeadókat jelöli. Minden lépésben egyre kevesebb párhuzamos egységre van szükség.

Copyright © 2021 / BME AUT



5.50. ábra – Párhuzamos redukció interleaved adatstruktúra esetén

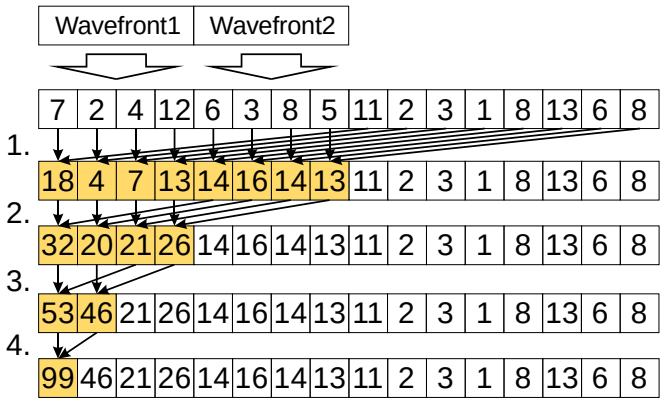
Folytonos (sequential) adatelérés esetén (5.51 ábra) a fél blokkmérettel eltolt elemeket adjuk össze. Memóriaelérés szempontjából ez kedvezőbb, a gyakran írt és olvasott értékek egymáshoz közel helyezkednek el.



5.51. ábra – Párhuzamos redukció folytonos adateléréssel

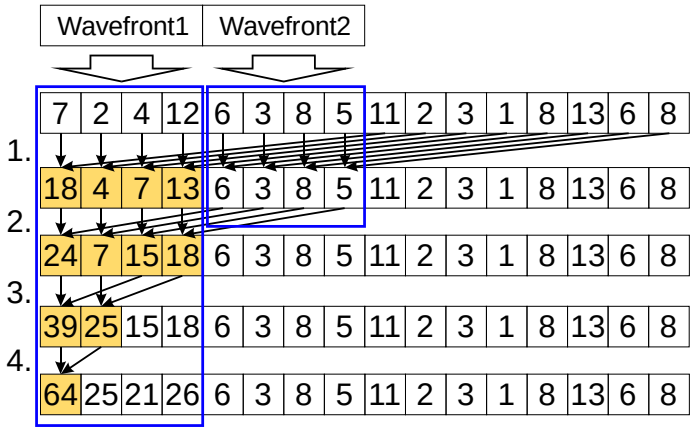
Felmerülhet a kérdés, hány elemet dolgozzunk fel párhuzamosan, hogyan szervezzük a végrehajtást? A végrehajtás alapegysége a work-item (thread). A work-item lehet a példában egy adatelem, egy adatpár vagy akár több adat is, melyeket lépésenként, párhuzamosan össze akarunk adni. A feldolgozás az eszközön wavefrontokban valósul meg (egyszerre pl. 32, 64 work-item-en, vektorosan). A wavefrontok feldolgozása is történhet konkurens módon, egy CU feldolgozhat egyszerre akár több wavefrontot is (ld. GCN esetén 4 SIMD blokk egy CU-ban, mindegyiknél akár 10 wavefront állhat sorban), ezzel javítva a rendszer kihasználtságát.

Copyright © 2021 / BME AUT



5.52. ábra – A work-item-ek wavefrontba csoportosítva

A wavefrontok végrehajtásának ütemezése jelentősen befolyásolhatja az eredményt. Amennyiben az 5.53 ábrán látható módon wavefront1-en a kód előbb lefut, mint wavefront2-n, wavefront1 a még nem módosított adatokkal dolgozik, így hibás eredményt ad. A probléma elkerüléséhez szinkronizációra van szükség.



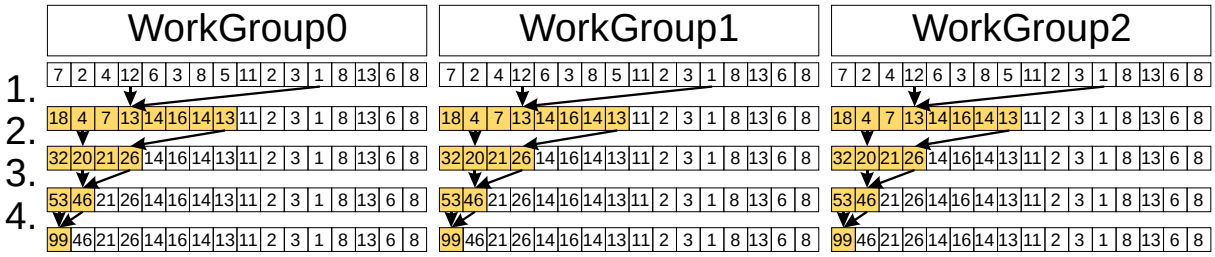
5.53. ábra – Wavefront1 és wavefront2 közötti szinkronizáció nélkül hibás eredmény kapható

A szinkronizáció a barrier() utasítással valósítható meg, ezt korábban a GCN architektúra utasításkészletében már említettük. A barrier hívására a work-item-ek végrehajtása megáll, míg a csoportban lévő összes work-item el nem éri a barrier utasítást. A példában az egyes lépések közé barrier() függvényhívásokat kell elhelyezni.

A GPU-kban több CU is elérhető, melyek párhuzamosan használhatóak, tovább rövidítve a végrehajtási időt. A work-item-ek ütemezése során megadható a csoportok (workgroup) mérete. A keretrendszer garantálja, hogy az egy csoportban lévő elemek feldolgozása egy CU-n valósul meg. A csoportok számának növelésével lehetővé tehető, hogy a feladaton több CU is osztozzon. Mivel több CU párhuzamosan is dolgozik, írja-olvassa a memóriát, érdemes a helyileg használt adatokat a központi memóriából egy lépésben a CU-n belüli lokális memóriába másolni, mert ezután a redukció során csak egyetlen részösszeg visszaírására van

Copyright © 2021 / BME AUT

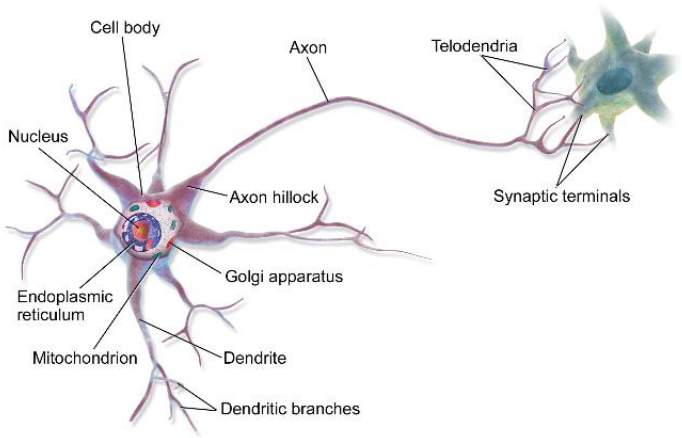
szükség. Fontos, hogy a redukciót csak csoportonként végezzük el, meghatározva egy részösszeget. Ezeket a részösszegeket utolsó lépésként akár a CPU-n is összegezhetjük.



5.54. ábra – A teljes range workgroup-okba csoportosítható, tovább növelve a párhuzamosítás mértékét

5.7 Neurális hálók

A neurális háló alapú mesterséges intelligencia architektúrák alapját képezi a biológiai analógiára visszavezethető idegsejtek és kapcsolataik (szinapszisok) modellezése. Akárcsak a biológiai neuronok, a mesterséges modellek is sok bemenettel (dendrit) rendelkeznek, amelyeknek valamilyen súlyozott kombinációjának a függvénye jelenik meg a kimeneten (axon). A biológiai neuronok az axonon tüzelési impulzusokat generálnak, ez modellezhető a kimeneten valamilyen folytonos értékkel. Az emberi agyban a neuronok száma egyes becslések szerint kb. 10^{11} darab, míg a szinapszisoké kb.10^{14}-10^{15}, tehát egy neuronnak akár több ezer kapcsolata is lehet más neuronokkal. A struktúra sem olyan kötött és egyszerű, mint a mesterséges modellek esetén.



5.55. ábra – Egy idegsejt felépítése [27]

Egy egyszerű neuron modellje látható a 5.56. ábrán. M+1 bemenettel rendelkezik, ebből egy jellemzően fix értéket vesz fel, így ofszet is kezelhetővé válik. A bemenetek és a hozzájuk rendelt súlyok alapján kialakul egy súlyozott összeg, melyből a megadott aktivációs függvény

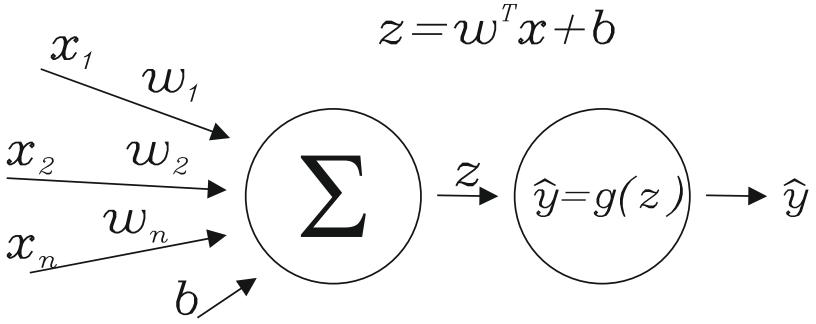
[27] Forrás: https://en.wikipedia.org/wiki/Neuron (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT

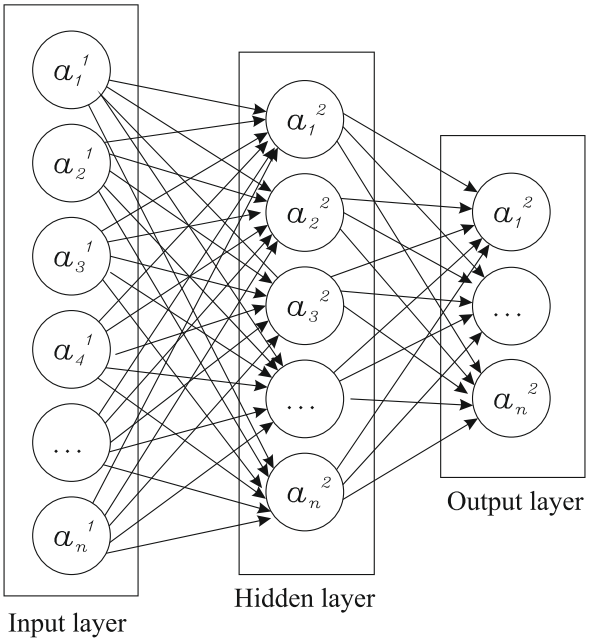
állítja elő a tényleges kimenetet. Aktivációs függvényekből számtalan variáció használható az egyszerűbb azonosságtól vagy küszöbtől a bonyolultabb Gauss, szigmoid, tangens függvényekig. A neuronok kimenetei újabb rétegek bemeneteihez kapcsolódhatnak.

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

$$z = w^T x + b$$



5.56. ábra – Tipikus neuronmodell



5.57. ábra – A neuronok többrétegű hálókba rendezhetők több be-és kimenettel

A GPU-k számára kifejezetten előnyös a feladat, hiszen sok szorzást és összeadást kell elvégezni párhuzamosan, valamint a számítások között kevés az egymásrahatás. Hagyományosan a számoláshoz 32 bites lebegőpontos ábrázolást használtak, hiszen az architektúrák grafikai számítások miatt erre lettek optimalizálva, azonban ez a pontosság nem biztosított minőségi előrelépést kisebb pontossággal történő számoláshoz képest, valamint magasabb fogyasztással és lassabb végrehajtással párosult. Újabb GPU architektúrák bevezették a half 16 bites lebegőpontos típusokat, melyeket dupla mennyiségben tudnak

Copyright © 2021 / BME AUT

feldolgozni a 32 bitesekhez képest. Kutatások azt mutatták[28], hogy a gyakorlati alkalmazások esetén nem romlanak szignifikánsan az eredmények akár erősebb kvantálás alkalmazásával, pl. INT8 típusokkal történő számítások esetén sem. A vektoros műveletvégzők a rövidebb szavakat még több példányban tudták feldolgozni, pl. 32 bites regiszterben 4 db INT8-cal történő párhuzamos utasítást. Sőt, akár még extrémebb kvantálással is alkalmazhatók a modellek: 2-3 állapotú ábrázolás esetén a szorzás műveletre nincs is szükség, összeadásokkal és kivonásokkal kezelhető a feladat. Az nVidia Turing architektúrájában kifejezetten neurális hálós alkalmazásokhoz jelent meg a vektoros half mellett az INT8 és INT4 típus támogatása.

A neurális hálókkal kapcsolatban több feladat fogalmazható meg: a probléma megoldására szolgáló struktúra kiválasztása, az együtthatók meghatározása tanítóadatok segítségével, valamint a betanított adatok alkalmazásával a modell „futtatása", azaz dedukció (inference), a modell kimeneteinek meghatározása új bemeneti adatok alapján. Számos algoritmus létezik a modellek tanításához, a súlyok becsléséhez (optimalizáláshoz), mely a tanító adathalmazok esetén az elvárthoz közeli kimenetet adja. A gradiens alapú módszerek esetén a kimenettől vezetik vissza a hibát a neuronok hozzájárulása függvényében, majd folyamatosan korrigálják a súlyokat. A feladat rendkívül számításigényes, nagyteljesítményű párhuzamos gyorsítók nélkül nem lehetne hatékonyan használni nagyobb hálók esetén. Egy betanított háló esetén az inference feladat lényegesen egyszerűbb, a bemenetek függvényében végig kell számolni rétegenként a hálók kimeneteit az egyes neuronok kimeneteinek meghatározásával, így közvetlenül megkaphatók a kimenetek.

Az nVidia saját GPU-it beágyazott környezetben is elérhetővé teszi ARM alapú CPU magok mellé csomagolva (ld. 5.3. táblázat). A Volta GPU-t tartalmazó változatok már a Turing architektúránál bemutatott tenzor magokat is tartalmazzák neurális hálós alkalmazásokhoz. A GPU-k rendelkeznek továbbá kamera és kijelző interfészekkel, videó kódoló- és dekódolóval, PCIe bővítési lehetőségekkel, az őket tartalmazó kártyák rendelkeznek integrált RAM-mal, flash memóriával, hálózati csatolóval, így már önmagukban is számos alkalmazási területen használhatók (gépi látás, képfeldolgozás, térképezés, objektumfelismerés stb).

Termék	CPU	GPU
Jetson Nano	4 x Cortex A57	128 magos Maxwell GPU
Jetson TX1	4 x Cortex A57	256 magos Maxwell GPU
Jetson TX2	2 x nVidia Denver 4x Cortex A57	256 magos Pascal GPU
Jetson Xavier NX	6 x nVidia Carmel	384 magos Volta GPU 48 Tensor mag
Jetson AGX Xavier	8 x nVidia Carmel	512 magos Volta GPU 64 Tensor mag

5.3. táblázat – nVidia Jetson termékcsalád

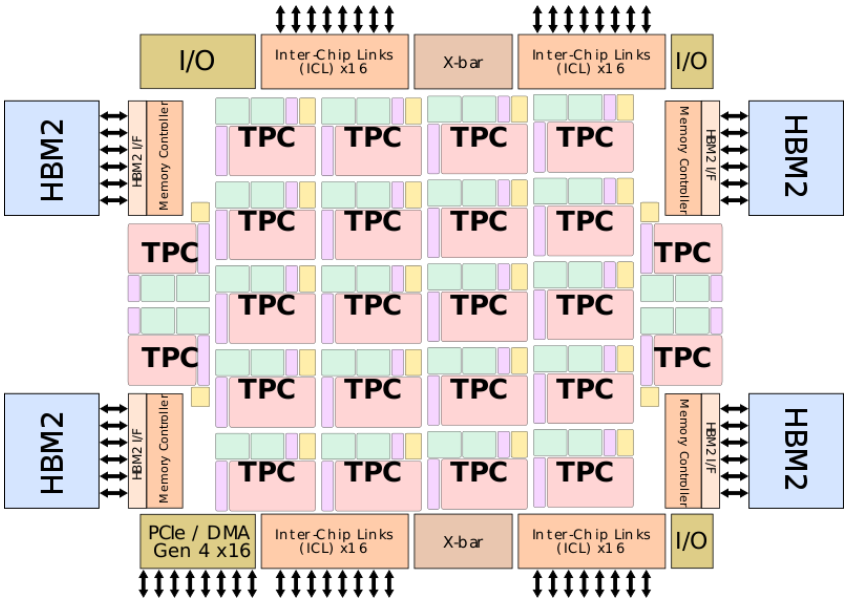
Neurális hálók gyorsítására megjelentek dedikált (nem GPU és nem általános célú) architektúrák és hardverek is, ilyenek pl. a Google Coral, az Intel Nervana és a Tesla

[28] Forrás, további információ: https://nervanasystems.github.io/distiller/quantization.html

Copyright © 2021 / BME AUT

Hardware 3. A dedikált megoldás miatt még hatékonyabban képesek a feladatukat ellátni, a tanításra és a tanított háló futtatására eltérő lapkák is készülnek.

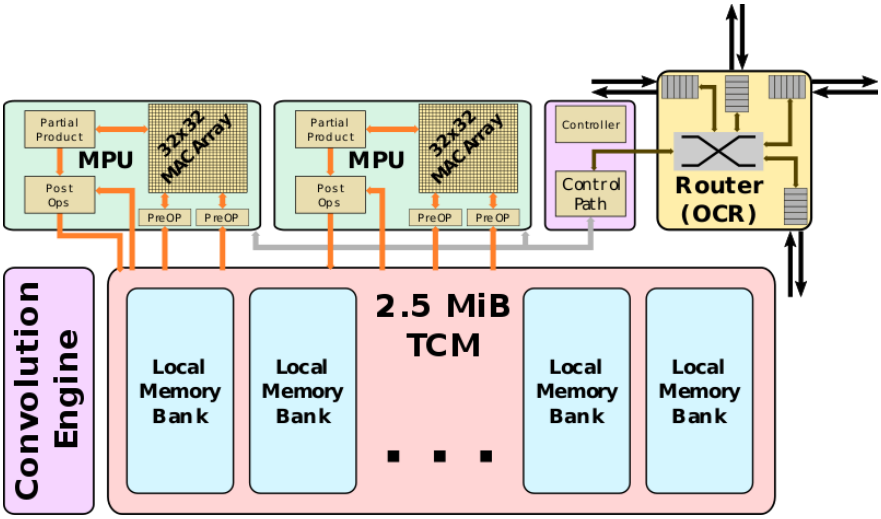
Az Intel Nervana processzorcsaládjában a Spring Crest típus 24 TPC (Tensor Processor Cluster) blokkot tartalmaz, a memóriainterfészei szélesek, HBM memóriához kapcsolódnak. Minden TPC blokk rendelkezik helyi memóriával és két 32x32 (darab) MAC szorzó-osztó egységgel. Minden TPC egység így 32x32x2=2048 szorzásra és összeadásra képes órajelenként. Lebegőpontos számoláshoz bfloat16 típust használják (ez megegyezik az IEEE-754 egyszeres pontosságú ábrázolással, kivéve, hogy a mantissza 23 helyett 7 bites), az eredményt pedig 32 bitesen tárolják. A működési frekvenciája 900MHz–1,1GHz közötti, 24TPU-ra vonatkoztatva messze túlmutat a GPU-kkal megvalósítható teljesítményen.



5.58. ábra – Az Intel Nervana Spring Crest blokkvázlata, amely 24 TPC blokkot tartalmaz [29]

[29] Forrás: https://en.wikichip.org/wiki/nervana/microarchitectures/spring_crest (Hozzáférés: 2020.06.29.)

Copyright © 2021 / BME AUT



5.59. ábra – Nervana TPC blokk részletesen

Copyright © 2021 / BME AUT

Felhasznált és ajánlott irodalom

Hozzáférés: 2020.06.29.

[1] AMD Reference Guide, Southern Islands Series Instruction Set Architecture: http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf

[2] AMD Reference Guide, Sea Islands Series Instruction Set Architecture: http://developer.amd.com/wordpress/media/2013/07/AMD_Sea_Islands_Instruction_Set_Architecture.pdf

[3] AMD Reference Guide, Graphics Core Next Architecture Generation 3: http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf

[4] AMD Reference Guide, ”Vega” Instruction Set Architecture: https://developer.amd.com/wordpress/media/2017/08/Vega_Shader_ISA_28July2017.pdf

[5] AMD Reference Guide, ”RDNA 1.0” Instruction Set Architecture: https://gpuopen.com/wp-content/uploads/2019/08/RDNA_Shader_ISA_7July2019.pdf

[6] AMD: RDNA Architecture: https://gpuopen.com/wp-content/uploads/2019/08/RDNA_Architecture_public.pdf

[7] AMD: Fusion Developer Summit Slides 2011: http://developer.amd.com/wordpress/media/2013/06/2620_final.pdf

[8] AMD: Advanced Shader Programming on GCN, GDC 2017: https://gpuopen.com/wp-content/uploads/2017/03/GDC2017-Advanced-Shader-Programming-On-GCN.pdf

[9] Learn OpenGL, Website: https://learnopengl.com/

[10] Khronos OpenCL Website: https://www.khronos.org/opencl/

[11] nVidia CUDA Website: https://developer.nvidia.com/cuda-zone

[12] GPUOpen Website: https://gpuopen.com/

[13] HSA: https://en.wikipedia.org/wiki/Heterogeneous_System_Architecture

[14] Sugárkövetés: https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

[15] Grafikus csővezeték: https://en.wikipedia.org/wiki/Graphics_pipeline

[16] HBM memória: https://en.wikipedia.org/wiki/High_Bandwidth_Memory

Copyright © 2021 / BME AUT

[17] TI OpenCL User's Guide, Understanding Kernels, Work-groups and Work-items: http://downloads.ti.com/mctools/esd/docs/opencl/execution/kernels-workgroups-workitems.html

[18] Thr Tech Report: Nvidia's GeForce 8800 graphics processor: https://techreport.com/review/11211/nvidias-geforce-8800-graphics-processor/

[19] Anandtech: The NVIDIA Titan V Deep Learning Deep Dive: It's All About The Tensor Cores: https://www.anandtech.com/show/12673/titan-v-deep-learning-deep-dive

[20] Asm4GCN - GCN Assembler for AMD GPUs: https://github.com/SunsetQuest/Asm4GCN

[21] DeCUDA project: https://github.com/laanwj/decuda/wiki

[22] NVIDIA Deep Learning Performance: https://docs.nvidia.com/deeplearning/performance/index.html

[23] nVidia GPU Gems 1-3: https://developer.nvidia.com/gpugems/gpugems3/contributors

[24] Anandtech, AMD Radeon HD 7970 Review: https://www.anandtech.com/show/5261/amd-radeon-hd-7970-review

[25] Techspot, Navi vs. Turing: An Architecture Comparison: https://www.techspot.com/article/1874-amd-navi-vs-nvidia-turing-architecture/

[26] OpenCL Programming by Example, An Example of OpenCL Program: https://subscription.packtpub.com/book/application_development/9781849692342/1/ch01lvl1sec12/an-example-of-opencl-program

[27] Erik Smistad: Getting started with OpenCL and GPU Computing: https://www.eriksmistad.no/getting-started-with-opencl-and-gpu-computing/

[28] Oak Ridge National Laboratory: OpenCL Vector Addition Example: https://www.olcf.ornl.gov/tutorials/opencl-vector-addition/

[29] AMD OpenCL SDK 3.0, A simple OpenCL example

[30] ATI OpenGL Programming and Optimization Guide: https://developer.amd.com/wordpress/media/2012/10/ATI_OpenGL_Programming_and_Optimization_Guide.pdf

[31] Jason L. Mitchell Jason L. Mitchell: Advanced Vertex and Pixel Advanced Vertex and Pixel Shader Techniques Shader Techniques:

Copyright © 2021 / BME AUT

http://developer.amd.com/wordpress/media/2012/10/ATI_EGDC_AdvancedShaders.pdf

Copyright © 2021 / BME AUT

Nagyteljesítményű mikrokontrollerek és interfészek

Elektronikus jegyzet

II. rész

Buszok és interfészek

Tárgyfelelős: **Dr. Tevesz Gábor**
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117 Budapest, Magyar tudósok körútja 2.
Q ép. B szárny II. em. B216.
Tel: 463-2881
E-mail: tevesz@aut.bme.hu

Szerző(k): **Dr. Gál Tibor, Dr. Tevesz Gábor**
BME Automatizálási és Alkalmazott Informatikai Tanszék

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki mesterszak (MSc)
Számítógép alapú rendszerek specializáció hallgatói

Utolsó módosítás: 2021. 04. 20.

Verzió: 2.0

Copyright © 2021 / BME AUT

COPYRIGHT

Jelen dokumentum a BME Villamosmérnöki és Informatikai Kar hallgatói számára készített elektronikus jegyzet. A dokumentumot a Nagyteljesítményű mikrokontrollerek és interfészek c. tantárgyat (BMEVIAUMA07) felvevő hallgatók jogosultak használni, és saját céljukra 1 példányban kinyomtatni. A dokumentum módosítása, bármilyen eljárással részben vagy egészben történő másolása tilos, illetve csak a szerző(k) előzetes engedélyével történhet.

Copyright © 2021 / BME AUT

Tartalom

6 Buszok és interfészek általános jellemzői 5

6.1 Interfészek tulajdonságok szerinti osztályozása 5
6.2 Múlt és jelen 10
6.3 Az aktuális igények 14
6.4 A 8b/10b kódolás 23
6.5 Videojel-tömörítés 28

7 A SATA interfész 32

7.1 A SATA elődje - az ATA (IDE) interfész 32
7.2 A háttértárolók fontosabb jellemzői 37
7.3 A soros ATA interfész 40
7.3.1 Általános áttekintés 41
7.3.2 A SATA rétegszerkezete 42
7.3.3 A fizikai réteg 43
7.3.4 Az adatkapcsolati réteg 49
7.3.5 Transzportréteg 51
7.3.6 Native Command Queuing (NCQ) 53

8 Universal Serial Bus 55

8.1 Az USB rendszer architektúrája 60
8.2 Elektromos jellemzők 65
8.2.1 Az USB 1.1-es hub és eszközök kapcsolata 66
8.2.2 Az USB 2.0-s hub és eszközök kapcsolata 68
8.2.3 Az USB 3.x-es hub és eszközök kapcsolata 69
8.2.4 Az USB4 szabvány elektromos jellemzői 72
8.2.5 Az USB szabványok elektromos jellemzőinek összefoglalása 72
8.3 Logikai jellemzők 73
8.3.1 A fizikai réteg tulajdonságai 73
8.3.2 Az adatkapcsolati és a transzportréteg 75
8.4 Szoftver (driver) modell 76
8.4.1 USB leírók 76
8.4.2 Az USB host-szoftver 77

9 A PCI buszcsalád 80

9.1 A PCI busz 82

Copyright © 2021 / BME AUT

9.1.1 Mechanikai jellemzők ... 83
9.1.2 Elektromos jellemzők ... 84
9.1.3 Logikai jellemzők ... 86
9.2 A PCI-X busz ... 98
9.3 A PCI Express busz ... 101
9.3.1 A PCI Express rendszer architektúrája ... 102
9.3.2 A PCI Express rétegszerkezete ... 104
10 Korszerű soros buszok kapcsolódása ... 119
10.1 A DisplayPort interfész ... 119
10.2 A HDMI interfész ... 121
10.3 A Thunderbolt interfész ... 122
10.4 Kapcsolódások ... 126

Copyright © 2021 / BME AUT

6 Buszok és interfészek általános jellemzői

Az előző fejezetekben megismerkedtünk a nagyteljesítményű mikrokontrollerek (mikroprocesszorok) legfontosabb architektúrális tulajdonságaival, néhány jellemző képviselő tényleges felépítésével. Ezen mikrokontrollerek közös jellemzője a lehető legnagyobb számítási teljesítmény biztosítása általános célú feladatok, vagy tipizálható célfeladatok ellátására. Mind az architektúrák, mind a konkrét képviselők paraméterei között visszatérő elemként jelent meg a lehető legnagyobb sebesség, az időegység alatt elvégezhető legtöbb művelet, a legnagyobb feldolgozható adatmennyiség fogalma.

Ezekből az elemekből (áramkörökből, chipekből) épülnek fel a digitális számítógépek. Már korábbi tanulmányaink során is megismerkedtünk azzal, hogy a digitális integrált áramkörök belsejében érvényes jelszintek, működési sebességek, terhelési paraméterek az áramkörök határain (az integrált áramkörök kivezetéseinél) megváltoznak – a környezetből érkező, zajokkal terhelt jelek fogadását védőkapcsolások és jelformáló kapcsolások végzik, a környezetbe induló kimenetek jelszintjei, jelváltási sebessége, terhelhetősége eltér az áramkörök belsejében alkalmazható értékektől. Gondoljunk arra, hogy ameddig egy chip belsejében a távolságok a chip méretéből adódóan korlátosak, az alkatrészek között, vagy a fizikailag is egymástól távolabb elhelyezkedő egységek között már ezekkel a távolságokkal, pontosabban az ezekből adódó jelterjedési időkkel is számolnunk kell. Az elektromos jelek a fémekben közel fénysebességgel terjednek, mégis vegyük figyelembe azt az egyszerű fizikai tényt, hogy a fénysebesség közismert $c_0 = 3 \cdot 10^8$ m/s értéke pusztán más egységekben kifejezve $c_0 = 30$ cm/ns. Figyelembe véve, hogy áramköreink már gigahertzes frekvenciával (amelynek a periódusideje 1 ns) működnek, érzékelhetjük, hogy az egységek közötti távolságok már akár egy digitális egységeket tartalmazó kártyán belül sem feltétlenül elhanyagolhatók.

Mindez azt eredményezi, hogy az előbb említett egységeket, áramköröket összekapcsoló interfészek és buszok tulajdonságai (mechanikai, elektromos, logikai tulajdonságok) között legtöbbször meghatározó paraméterként jelenik meg az interfész sebessége, összekapcsolva az időegység alatt átvihető információ mennyiségével. Digitális rendszerekben ezt nevezzük csatornakapacitásnak vagy sávszélességnek.

6.1 Interfészek tulajdonságok szerinti osztályozása

Valamennyi interfész és buszrendszer rendelkezik

- mechanikai,
- elektromos és
- logikai

Copyright © 2021 / BME AUT

specifikációval. Ezen tulajdonságok biztosítják, hogy a világ különböző pontjain különböző gyártók által előállított egységek egy rendszerben összekapcsolhatók és ezt követően az interfész specifikációja szerint működőképesek legyenek.

Mechanikai tulajdonságok:

1. Mechanikai méretek
2. interfészkártya mérete, tűrése;
3. direkt (a nyomtatott áramkörbe fixen beépítésre kerülő) csatlakozók típusa, mérete, elhelyezése, tűrése;
4. Indirekt csatlakozók (pl. szalagkábeles összekötések) típusa, osztálya;
5. Mechanikai szilárdság adatok (nyomtatott áramkör és csatlakozók vonatkozásában is);
6. A hibás csatlakoztatást meggátló mechanikai elemek (kulcsok) fajtája, helye, tűrése;
7. A megbízhatóságot nem befolyásoló csatlakoztatások (betételek-kivételek) száma;

stb.

Elektromos tulajdonságok:

1. Jelátvitel jellege (aszimmetrikus, szimmetrikus/differenciális);
2. Csatlakoztathatóság jellege (feszültségmentes állapotban, hot swap/hot plug);
3. Logikai értékek reprezentálása (feszültség/áram, jelszintek értéktartományai);
4. A jelek terhelhetősége (min/max áramterhelés, induktivitás és kapacitásértékek);
5. A rendszer statikus zavartűrése (zajtávolság);
6. Meghajtóképesség és fogyasztás;
7. Dinamikus tulajdonságok (szekvenciák, min/tip/max időzítések stb.);
8. Rendszer paraméterek (terjedési idők, lezárások, reflexiók stb.)

Logikai tulajdonságok:

1. Rendszeren belüli (intrasystem) vagy rendszerek közötti (intersystem) kommunikáció;
2. Kommunikáció iránya (egyirányú, kétirányú, váltakozó irányú);
3. Jelleg (soros, párhuzamos);
4. Topológia (gyűrű, csillag, lánc, sín/busz stb.);
5. Topográfia (az egységek fizikailag hol/hogyan helyezkedhetnek el);
6. Tranzakciók (egyenértékű egységek, master/slave);
7. Protokoll;
8. Arbitráció (busz vezérlés jogának kiosztása) – centralizált/elosztott;
9. Adatátvitel (nem multiplexelt/multiplexelt, blokkos);
10. Címzés (logikai, geografikus);
11. Időzítés (szinkron, aszinkron, szemiszinkron, osztott);

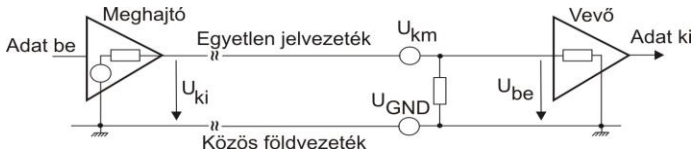
A mechanikai tulajdonságok rendkívül fontosak egy-egy egység (pl. fizikai kártya) mechanikai méretei, felépítése, az alkalmazott csatlakozók típusa, elhelyezkedése szempontjából. Ezeknek az előírásoknak a pontos betartása a különböző gyártóktól származó egységek fizikai csatlakoztathatóságát, annak mechanikai feltételeit, ismételhetőségét garantálja.

Az elektromos tulajdonságok rendkívül fontosak az ilyen rendszereket tervező és üzemeltető mérnökök számára. A „bármit összedugva annak működnie kell” sok esetben csalódások

Copyright © 2021 / BME AUT

forrása, az ezekből adódó helytelen működés pedig komoly károk forrása lehet. Nézzük ezzel kapcsolatban a legfontosabb általános ismereteket:

Aszimmetrikus jelátviteli rendszer: az információátvitel hordozója az U_{be} bemenő feszültség, melyet a közös földvezetékhez képest mérünk a fogadó (vevő) oldalon.



6.1 ábra – Aszimmetrikus jelátvitel

A 6.1 ábra alapján azonnal látható, hogy a megoldás legnagyobb problémái:

- az U_{GND} földvezeték potenciáljának eltolódása;
- U_{km} közös módusú zajfeszültség a jelvezetékben.

A megoldás előnyei:
- egyszerű, alacsony költségű;
- egyetlen vezeték szükséges a jelátvitelhez;

és hátrányai:
- zajos környezetben
  - árnyékolt kábel és jelvezetékenként külön földvezeték szükséges;
  - 1 m távolság felett már sodrott érpár javasolt (indukciós hurok!);
  - mindezek gyakorlatilag megszüntetik az alacsony költségeket;
- alacsony zajimmunitás
  - közös módusú zajok miatt;
  - áthallás nagyobb frekvenciákon;
  - jelentős EMI[1];
  - belső zajok forrása a lezáratlan jelvezetékek miatt.

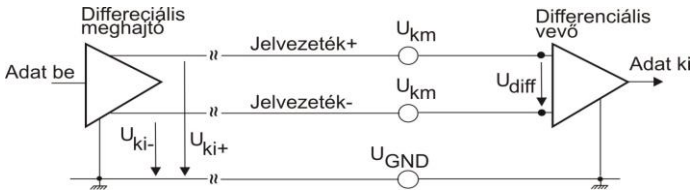
Szimmetrikus (vagy differenciális) jelátviteli rendszer: az információ hordozója a két bemeneti pont között mérhető feszültség ($U_{diff} = U_{ki+} – U_{ki-}$) polaritása.

Általánosságban elmondható, hogy számottevő előnyei miatt napjainkban az egyik legelterjedtebben használt jelátviteli módszer (Ethernet, CAN, USB, RS-422, RS-485 stb.). A két jelvezeték („Jel+” és „Jel-”) tipikusan egy sodrott érpárt alkot, ily módon igazak a következő kijelentések:

[1] EMI: elektromágneses interferencián azt a szabad térben kibocsátott vagy sugárzott, táp- vagy jeltovábbító vezetékeken továbbított jelet értjük, amely veszélyezteti a rádiónavigációs rendszerek vagy egyéb biztonsági szolgáltatások működését, illetve jelentős mértékben rontja, zavarja, vagy időről-időre megszakítja a rádió-távközlési szolgáltatást.

Copyright © 2021 / BME AUT

- a két vezeték nem távolodik el egymástól, ugyanazon külső elektromágneses behatások érik mindkettőt;
- a két vezeték között a lehető legkisebb a hurokfelület a külső indukciós hatásokkal szemben (továbbá a sodrás miatt ezek hatása is váltakozó előjelű).



6.2 ábra – Szimmetrikus (differenciális) jelátvitel

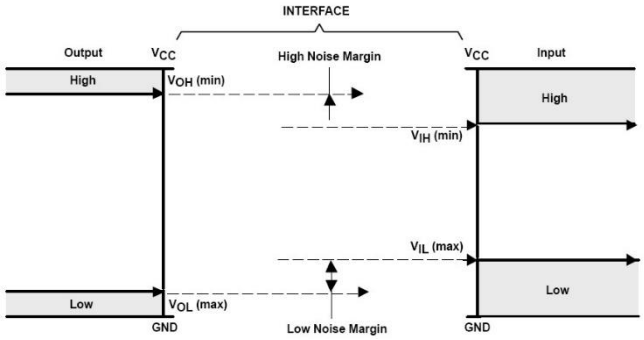
A megoldás előnyei:

- kevésbé érzékeny a közös módusú zajokra, ezek elvileg kioltják egymást a vevő bemeneténél;
- sodrott érpárral akár 10-50 GHz-ig is használható;
- kevesebb EMI-t termel;

és hátrányai:

- magasabb költségek a dupla vezetékszám miatt;
- differenciális meghajtót és vevőt igényel – ezek korábban drágábbak voltak a szimpla meghajtóknál (ma már nem);
- nagy frekvenciákon definit lezárásokat és stabil hullámimpedanciát igényel.

Aszimmetrikus jelek esetében emlékeztetünk az adó és a vevő áramkörök jelszintjeire és jeltartományaira (ld. 6.3 ábra):



6.3 ábra – Logikai jelszint-kompatibilitás bemenetek és kimenet között[2]

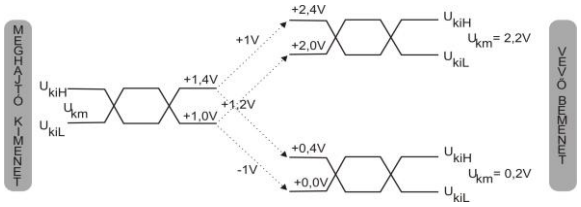
A még Digitális technikában tanult elméletre alapozva emlékezzünk arra, hogy bármely illesztés során a meghajtó áramkör V_{OH} és V_{OL} kimeneti jelszintje magasabb ill. alacsonyabb kell legyen, mint a fogadó áramkör V_{IH} és V_{IL} küszöbértékei. A megfelelő értékek közötti

[2] Texas Instruments: Understanding and Interpreting Standard-Logic Data Sheets, http://www.ti.com/lit/an/szza036c/szza036c.pdf (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT

távolságot nevezzük DC zajtávolságnak – a kimenő jelre kerülő ekkora zajfeszültség még nem okozhat hibás jelszint-felismerést a fogadó áramkörben. Mind a kimeneti, mind a bemeneti küszöbszintek egy-egy tartományt határolnak. Kimenet esetében a kimenő jel szintje V_{OHmin} és a tápfeszültség (V_{CC}) közé ill. V_{OLmax} és a földpotenciál (GND) közé fog esni. Mindkét küszöbérték a terhelőáram (I_{OH}, I_{OL}) függvénye, ezt a kapcsolás tervezésénél figyelembe kell vegyük. Hasonlóan alakulnak a sávok a digitális bemeneteknél – itt viszont a $V_{IHmin} – V_{CC}$ ill. a $V_{ILmax} –$ GND sávokban biztonságos a jelszintek helyes felismerése. A kimeneti sávoknak teljes terjedelmükben „bele kell találniuk" a bemeneti sávokba, a határok közötti tartalék jelenti ismét a zajtávolságot.

Szimmetrikus (a továbbiakban differenciális) jelátvitel esetén csak a jelvezetékek közötti feszültség előjele az információhordozó, a vonalak DC közös módusú komponense csak zavaró tényező (pl. a parazita kapacitások vonatkozásában), a tényleges jelfelismerést nem befolyásolják (akár pozitív, akár negatív irányba tolódik el mindkét jelvezeték potenciálja).

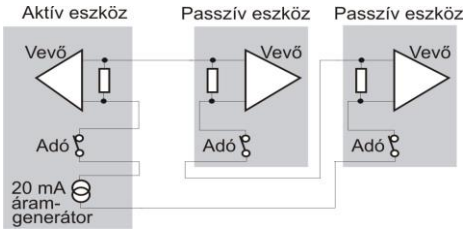


6.4 ábra – Szimmetrikus (differenciális) jelátvitel jelszintjei

A differenciális jelátvitelnek sok esetben alkalmazott módszerét képviselik az áramhurkok, ezeknek 3 fajtája ismeretes:

1. Digitális áramhurok
2. Analóg áramhurok
3. Bipoláris áramhurok

A digitális áramhurok esetében az áram hiánya vagy megléte felel meg a logikai értékeknek. A 60-as évektől alkalmazták távnyomtatók vezérlésére, az áram nagysága alkalmazásonként eltérő (20 mA, 60 mA) volt. Egészen 19 200 bps sebességig működőképes volt, az alacsony impedanciás („áram") jelleg miatt zajérzékenysége alacsony volt. Az RS-485 (-422) rendszerek mára gyakorlatilag kiszorították.



6.5 ábra – Digitális áramhurok

Copyright © 2021 / BME AUT

Az analóg áramhurok 4-20 mA-es rendszerként is ismert. A 4 mA-es alsó áramérték biztosította az érzékelő tápellátását, az e feletti tartomány volt a mért jel tényleges értékének hordozója.

A bipoláris áramhurok rendszerben a két logikai értékhez ellentétes irányú áramot rendelünk. Lényegében így működik az USB nagyobb sebességeken (480 Mb/s). Vagy a D+ vagy a D- vonalat hajtja meg 17,78mA-es árammal, hogy a 45 ohmmal a föld felé lezárt vonalai között +400mV vagy -400mV feszültség lépjen fel (ld. később).

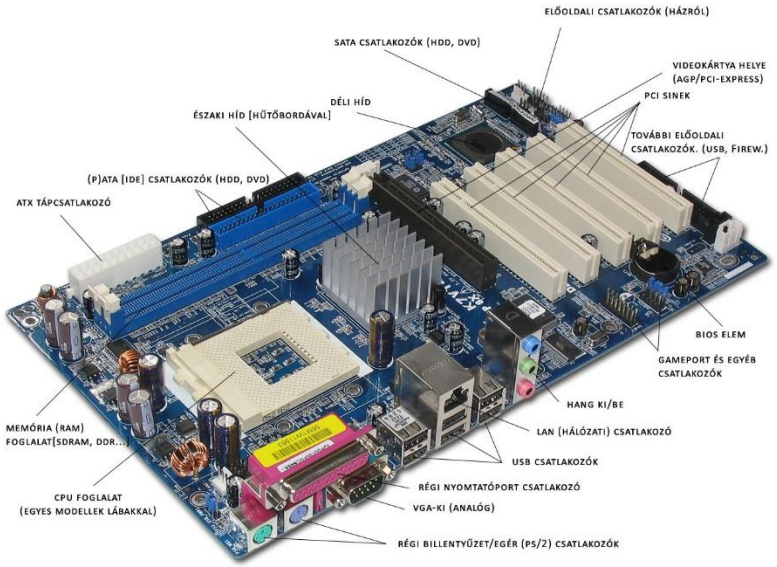
6.2 Múlt és jelen

Az interfészek vonatkozásában a számítógépekben három célcsoportot különíthetünk el:

A számítógépek alaplapjaihoz csatlakozó önálló, adott célfeladatra készült egységek csatlakozófelületeit (”alaplapi buszok”). A számítógépen belül elhelyezkedő bővítőkártyák még az alaplaphoz hasonló zajvédett környezetben látják el feladatukat, kiterjedésük és a jelátviteli távolságok korlátosak, így általában a vezetékeken még logikai jelszintek használatosak.

Külön csoportba sorolandók a háttértárak (merevlemezek, hajlékony lemezek, CD/DVD írók/olvasók, stb.) csatlakoztatására alkalmas interfészek – ezeknél a nagytömegű, blokkos adatátvitel és viszonylag kevesebb interakció jellemző.

A számítógép szűkebb/tágabb környezetében található egységekhez (kezelő- és megjelenítőeszközök, nyomtatók, szkennerek, stb.), tágabb értelemben esetleg a számítógépes hálózatokhoz kapcsolódó interfészfelületek (Ethernet, WLAN, stb.).



6.6 ábra – PC alaplap csatlakozó interfészekkel[3]

[3] Forrás: https://hu.wikipedia.org/wiki/Alaplap (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT

A 6.6 ábra egy korábbi PC alaplapot mutat, melyen jól látható az interfészcsatlakozók sokasága (az előző fejezetben tárgyaltuk, hogy az Intel Nehalem mikroarchitektúra az északi hidat már a processzorba integrálta, napjaink alaplapjain ilyen egység elkülönítve már nem található). A fejlődés mértékének megítélhetőségéhez egy rövid összefoglaló formájában tekintsük át ezen interfészcsoportok fejlődését a múltban – egészen napjainkig!

Belső interfészek (alaplapi buszok)

Év	Név	Bit/s	Bájt/s v. T/s	Felhasználás	Fejlesztő
1981	ISA (AT-bus) 16 bit	64 Mbit/s	**8 MB/s**	PC/AT alaplap	IBM
1987	ISA (Micro Channel) 32 bit	528 Mbit/s	**66 MB/s**	PS/2 alaplap	IBM
1988	EISA 32 bit	264 Mbit/s	**33 MB/s**	PC alaplap	Gang of Nine
1992	VESA 32 bit	1 064 Mbit/s	**133 MB/s**	PC alaplap	VESA
1993	PCI 2.0 32 bit/33 MHz	1 064 Mbit/s	**133 MB/s**	PC alaplap	Intel
1995	PCI 2.1 32 bit/66 MHz	2 128 Mbit/s	**266 MB/s**	PC alaplap	Intel
1995	PCI 2.1 64 bit/33 MHz	2 128 Mbit/s	**266 MB/s**	PC alaplap	Intel
1995	PCI 2.1 64 bit/66 MHz	4 264 Mbit/s	**533 MB/s**	PC alaplap	Intel
1997	AGP 1x	2 128 Mbit/s	**266 MB/s**	PC grafikus kártya	Intel
1997	AGP 2x	4 264 Mbit/s	**533 MB/s**	PC grafikus kártya	Intel
1998	PCI-X 1.0 64 bit/66 MHz	4 264 Mbit/s	**533 MB/s**	Szerver, workstation	IBM, HP, Compaq
1998	PCI-X 1.0 64 bit/133 MHz	8 528 Mbit/s	**1 066 MB/s**	Szerver, workstation	IBM, HP, Compaq
1999	AGP 4x	8 528 Mbit/s	**1 066 MB/s**	PC grafikus kártya	Intel
2002	AGP 8x	17 064 Mbit/s	**2 133 MB/s**	PC grafikus kártya	Intel
2003	PCI-X 2.0 64 bit/266 MHz	17 064 Mbit/s	**2 133 MB/s**	Szerver, workstation	IBM, HP, Compaq
2003	PCI-X 2.0 64 bit/533 MHz	34 128 Mbit/s	**4 266 MB/s**	Szerver, workstation	IBM, HP, Compaq
2003	PCI Express 1.0 (16 lanes)	32 Gbit/s	**2.5 GT/s**	többcélú	Intel, Dell, HP, IBM
2007	PCI Express 2.0 (16 lanes)	64 Gbit/s	**5 GT/s**	többcélú	Intel, Dell, HP, IBM
2010	PCI Express 3.0 (16 lanes)	126 Gbit/s	**8 GT/s**	többcélú	Intel, Dell, HP, IBM
2011	PCI Express 4.0 (16 lanes)	252 Gbit/s	**16 GT/s**	többcélú	Intel, Dell, HP, IBM
2017	PCI Express 5.0 (16 lanes)	504 Gbit/s	**32 GT/s**	többcélú	Intel, Dell, HP, IBM
2019	PCI Express 6.0 (16 lanes)	1008 Gbit/s	**64 GT/s**	többcélú	Intel, Dell, HP, IBM

6.1. táblázat – Az alaplapi buszok fejlődése

Néhány megjegyzés a táblázat jobb megértéséhez:

a) A táblázatban jól látható a személyi számítógépek alaplapi buszainak fejlődése. Az induló busz (ISA) továbbfejlesztései (EISA, VESA) sokáig tartották magukat, de végül szerepüket egy kiforrottabb, nagyobb teljesítményű buszrendszer, a PCI busz vette át. Ennek a nagyteljesítményű szerverek és workstation-ok által igényelt kiterjesztése a PCI-X busz. 2003-tól a PCI buszt újradefiniálták PCI Express néven. Alapvetően soros jellegű, de az igényektől függően minden linkje 1..32 párhuzamos sávot (lane) használhat. Innen kezdve a specifikáció már a transfer rate (T/s) fogalmat vezeti be a sebesség definiálására, hiszen a sávszélesség a rendelkezésre álló sávok számától is függ.

b) Ahogyan azt az előző fejezetekben már hallottuk, a gyorsan fejlődő grafikus interfészek az általánostól elkülönülő, speciális interfészfelületet igényeltek. Az AGP csatlakozófelületek a személyi számítógépek grafikus videókártyáinak interfészeként jelentek meg a PC alaplapokon.

Copyright © 2021 / BME AUT

c) Az eddig alapvetően párhuzamos illesztőfelületek az utóbbi években alakulnak át fokozatosan soros megoldásokká, a kevesebb vezetéket igénylő, igény szerint modulárisan párhuzamosítható (sávok) soros PCI Express busz sávszélessége utolérte, sőt meghaladta a párhuzamos buszokét. Ezekkel a megoldásokkal a következő fejezetekben fogunk részletesen megismerkedni.

Háttértároló interfészek

Év	Név	Bit/s	Bájt/s	Felhasználás	Fejlesztő
1986	SCSI 1	40 Mbit/s	**5 MB/s**	adatszerverek	Shugart Associates
1993	Fibre Channel 133 Mbps	**133 Mbit/s**	12,5 MB/s	adatszerverek	INCITS
1994	Fast SCSI 2	80 Mbit/s	**10 MB/s**	adatszerverek	Shugart Associates
1996	Fast Wide SCSI 2	160 Mbit/s	**20 MB/s**	adatszerverek	Shugart Associates
1996	Ultra Wide SCSI 40	320 Mbit/s	**40 MB/s**	adatszerverek	Shugart Associates
1997	Ultra DMA ATA 33	264 Mbit/s	**33 MB/s**	PC háttértárolók	WD, Compaq
1997	Ultra-2 SCSI	320 Mbit/s	**40 MB/s**	adatszerverek	Shugart Associates
1998	Ultra-2 Wide SCSI	640 Mbit/s	**80 MB/s**	adatszerverek	Shugart Associates
1999	Ultra-3 SCSI 160	1.2 Gbit/s	**150 MB/s**	adatszerverek	Shugart Associates
1999	Ultra DMA ATA 66	528 Mbit/s	**66 MB/s**	PC háttértárolók	WD, Compaq
2002	Ultra-320 SCSI	2.56 Gbit/s	**320 MB/s**	adatszerverek	Shugart Associates
2002	Ultra DMA ATA 100	800 Mbit/s	**100 MB/s**	PC háttértárolók	WD, Compaq
2003	Serial ATA 1 (SATA 1,5 Gbit/s)	**1.5 Gbit/s**	150 MB/s	PC háttértárolók	SATA-IO
2003	Ultra-640 SCSI	5.12 Gbit/s	**640 MB/s**	adatszerverek	Shugart Associates
2004	Serial ATA 2 (SATA 3 Gbit/s)	**3 Gbit/s**	300 MB/s	PC háttértárolók	SATA-IO
2004	SAS-1 (Serial Attached SCSI)	**3 Gbit/s**	300 MB/s	adatszerverek	INCITS
2005	Ultra DMA ATA 133	1064 Mbit/s	**133 MB/s**	PC háttértárolók	WD, Compaq
2005	Fibre Channel 8GFC	**8.5 Gbit/s**	800 MB/s	adatszerverek	INCITS
2009	Serial ATA 3 (SATA 6 Gbit/s)	**6 Gbit/s**	600 MB/s	PC háttértárolók	SATA-IO
2009	SAS-2 (Serial Attached SCSI)	**6 Gbit/s**	600 MB/s	adatszerverek	INCITS
2011	Fibre Channel 16GFC	**14.025 Gbit/s**	1.6 GB/s	adatszerverek	INCITS
2013	SAS-3 (Serial Attached SCSI)	**12 Gbit/s**	1.2 GB/s	adatszerverek	INCITS
2013	SATA Express (SATA-3.2)	**16 Gbit/s**	2 GB/s	PC háttértárolók	SATA-IO
2016	Fibre Channel 32GFC	**28.05 Gbit/s**	3.2 GB/s	adatszerverek	INCITS
2017	SAS-4 (Serial Attached SCSI)	**22.5 Gbit/s**	2.25 GB/s	adatszerverek	INCITS

6.2. táblázat – A háttértároló interfészek fejlődése

Itt is néhány megjegyzést fűzünk a táblázathoz a könnyebb megértés érdekében:

a) A személyi számítógépek világában sokáig meghatározó szerepet játszott az IDE (Integrated Drive Electronics) illesztőfelület, amely szinonim névként az ATA (Advanced Technology Attachment), UDMA (Ultra-DMA), később a soros interfészek megjelenése miatt utólag a PATA (Parallel ATA) nevet is viselte.

b) Az SCSI (Small Computer Systems Interface) a közép és felsőkategóriás számítógépek számára készült. Igényesebb, hatékonyabb kialakítása nagyobb teljesítőképességet biztosított, azonban a széles körű elterjedés számára megoldásai túl drágák voltak.

c) A táblázatból feltűnhet, hogy a nagytömegű adattovábbítás ellenére a párhuzamos illesztőfelület itt is a soros megoldás irányába változott (SATA, SATA-Express,

Copyright © 2021 / BME AUT

SAS), a PCI-Express busz előretörése azonban várhatóan ezeket is felváltja a közeljövőben.

d) A soros interfészek sávszélesség adatainál feltűnhet, hogy a bit/s és a bájt/s adatok között nem 8-szoros, hanem 10-szeres arányt fedezhetünk fel. Ennek oka a 8b/10b kódolás alkalmazása, amely 1 bájtnyi információt 10 biten továbbít (részletesebben ezt a 6.4 alfejezetben tárgyaljuk).

Külső interfészek

Év	Név	Bit/s	Bájt/s	Felhasználás	Fejlesztő
1960	Soros RS-232 általában	**9.6 kbit/s**	960 B/s	Szg. kapcsolat	CCITT
	Soros RS-232 max	**230.4 kbit/s**	23.0 kB/s	Szg. kapcsolat	CCITT
1970	Centronics Interface	1.2 Mbit/s	**150 kB/s**	PC környezet	Centronics, IBM
1975	Soros RS-422 max	**10.0 Mbit/s**	1.25 MB/s	12m környezet	ANSI/TIA/EIA
1990	PCMCIA vagy PC Card	1056 Mbit/s	**132 MB/s**	Félvezető tároló	PCMCIA
1995	FireWire (IEEE 1394) S100	**100 Mbit/s**	12.5 MB/s	PC környezet	Apple, Sony, Panasonic stb.
1995	FireWire (IEEE 1394) S200	**200 Mbit/s**	25 MB/s	PC környezet	Apple, Sony, Panasonic stb.
1995	FireWire (IEEE 1394) S400	**400 Mbit/s**	50 MB/s	PC környezet	Apple, Sony, Panasonic stb.
1996	Universal Serial Bus USB 1.0 (LS)	**1.5 Mbit/s**	187.5 kB/s	PC környezet	Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel
1996	Universal Serial Bus USB 1.0 (FS)	**12.0 Mbit/s**	1.5 MB/s	PC környezet	Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel
2001	Universal Serial Bus USB 2.0 (HS)	**480 Mbit/s**	60 MB/s	PC környezet	Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel
2002	HDMI 1.0, 1.1, 1.2	**4.95 Gbit/s**	627 MB/s	PC környezet	Hitachi, Panasonic, Philips, Sony, Toshiba stb.
2006	HDMI 1.3, 1.4	**10.2 Gbit/s**	1,275 GB/s	PC környezet	Hitachi, Panasonic, Philips, Sony, Toshiba stb.
2007	FireWire (IEEE 1394b) S800	**0.8 Gbit/s**	100 MB/s	PC környezet	Apple, Sony, Panasonic stb.
2007	DisplayPort 1.1	**10.8 Gbit/s**	1,35 GB/s	PC környezet	VESA
2010	DisplayPort 1.2	**21.6 Gbit/s**	2,7 GB/s	PC környezet	VESA
2011	Universal Serial Bus USB 3.0 (SS)	**5 Gbit/s**	640 MB/s	PC környezet	Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel
2011	Thunderbolt 1	**10 Gbit/s**	1,25 GB/s	PC környezet	Intel, Apple
2012	FireWire (IEEE 1394-2008) S1600	**1.6 Gbit/s**	200 MB/s	PC környezet	Apple, Sony, Panasonic stb.
2012	FireWire (IEEE 1394-2008) S3200	**3.2 Gbit/s**	400 MB/s	PC környezet	Apple, Sony, Panasonic stb.
2013	HDMI 2 .0	**18 Gbit/s**	2,25 GB/s	PC környezet	Hitachi, Panasonic, Philips, Sony, Toshiba stb.
2013	Thunderbolt 2	**20 Gbit/s**	2,5 GB/s	PC környezet	Intel, Apple
2014	Universal Serial Bus USB 3.1 (SS+)	**10 Gbit/s**	1,25 GB/s	PC környezet	Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel
2014	DisplayPort 1.3, 1.4	**32,4 Gbit/s**	4,05 GB/s	PC környezet	VESA
2016	Thunderbolt 3	**40 Gbit/s**	5 GB/s	PC környezet	Intel, Apple
2017	Universal Serial Bus USB 3.2 (SS+)	**20 Gbit/s**	2,5 GB/s	PC környezet	Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel
2017	HDMI 2 .1	**48 Gbit/s**	6 GB/s	PC környezet	Hitachi, Panasonic, Philips, Sony, Toshiba stb.
2019	Universal Serial Bus USB 4 (SS+ és TB3)	**40 Gbit/s**	5 GB/s	PC környezet	Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel
2019	DisplayPort 2.0	**80 Gbit/s**	10 GB/s	PC környezet	VESA

6.3. táblázat – A külső interfészek fejlődése

A szokásos megjegyzések a könnyebb áttekintéshez:

Copyright © 2021 / BME AUT

a) Vegyük észre, hogy a külső interfészek világában meghatározó módon a soros megoldások dominálnak, talán ez is az oka annak, hogy az aszinkron soros interfészek teljesítőképessége hatalmas fejlődésen ment keresztül az elmúlt 20-30 évben.

b) A régóta, és rendkívül sokáig létező hagyományos interfészek (RS-232, Centronics, PS-2) hatalmas vezetékhalmazt képeztek a személyi számítógépek körül, a billentyűzet, egér, monitor, nyomtató, scanner, modem, hálózati csatlakozás egységesítésért kiáltottak az ezzel foglalkozó szakemberek számára. Elindult egy először alacsonyabb elvárásokat megfogalmazó, majd elterjedése után nagyobb teljesítőképességű, egyre több eszközt integrálni képes buszrendszer, az USB busz fejlesztése.

c) Még az USB busz megjelenése előtt a nagyobb sebességű adatkapcsolatot, nagyobb mennyiségű külső adattovábbítást lehetővé tevő FireWire interfész terjedt el. Képességeit az USB, majd a Thunderbolt interfészek (mindkettővel foglalkozunk a későbbi fejezetekben) utolérték, a FireWire interfész fejlesztése leállt.

d) Az egyre nagyobb teljesítményű megjelenítők (monitorok, digitális televíziók, projektorok), az otthoni használatban is egyre jobban elterjedő audio-video adatfolyamok továbbítása iránti idény alapozta meg a HDMI (High Definition Multimedia Interface) szabvány létrejöttét és fejlődését. Konkurense a számítógépeket a nagyteljesítményű monitorokkal összekapcsoló DisplayPort interfész lett.

e) A nagyteljesítményű soros interfészek világában is igyekeztek ”rendet tenni” a fejlesztők, így az USB 4 szabvány már magában foglalja a Thunderbolt, a DisplayPort és a PCI-Express interfészek előnyeit és lehetőségeit is – a fejlesztők közelítették a nagyon sok tekintetben hasonló megoldásokat az egységesítés előnyeinek kihasználása érdekében.

6.3 Az aktuális igények

Érdemes kicsit körültekinteni: mire képes ma (ez a jegyzet 2020-ban íródik) egy személyi számítógép, milyen elvárásokat fogalmaznak meg napjaink külső-belső számítógép-egységei az őket a számítógéphez kapcsoló interfészek számára? Az előző fejezetben vázolt interfészek sebességei, sávszélességei létező, valós igényeket fednek le, vagy inkább az azokat fejlesztő cégek presztízsháborúja az illesztő felületek verziószámainak évenkénti változása?

A következőkben a három fő interfészcsoportot használó egységek főbb tulajdonságait tekintjük át illesztőfelületeik vonatkozásában. Mindhárom csoportban a kommersz alkalmazások felsőbb kategóriás képviselőit választottuk ki a szükséges adatátviteli sebességek elemzéséhez.

Belső interfészek (alaplapi buszok)

Amennyiben a személyi számítógépek belső buszai által igényelt sávszélességet keressük, a grafikus (video-) kártyák által igényelt adatátviteli sebességeket kell vizsgálnunk. Ezek a kártyák rendkívül aktív adatforgalmat bonyolítanak le az alaplapok processzoraival, emiatt az alaplapi buszrendszerek között sokáig a többi egységtől elkülönített, speciális busszal rendelkeztek (AGP).

Copyright © 2021 / BME AUT

Az AMD Radeon RX 5700 XT videokártya (megjelenés: 2019 Q3) grafikus processzora 1,6 GHz névleges frekvenciával, 9,75 TFLOPS egyszeres pontosságú műveletvégzési sebességgel működik (225W tip. teljesítményfelvétel mellett). A kártya alaplapi csatlakozó interfésze PCI Express 4.0 x16 lanes.



6.7 ábra – AMD Radeon RX 5700 XT videokártya[4]

Az előbbinél magasabb teljesítményt és árkategóriát képviselnek az NVIDIA RTX 2080 Ti videokárták. A grafikus processzor órajele 1,35 GHz, amely 1,635 GHz-ig növelhető, számítási teljesítménye 14,231 TFLOPS. A kártya névleges teljesítményfelvétele 260 W. A kártya alaplapi csatlakozó interfésze PCI Express 3.0 x16 lanes.



6.8 ábra – NVIDIA RTX 2080 Ti videokártya[5]

Háttértároló interfészek

A számítógépek háttértárolói komoly változásokon mentek át az elmúlt 5-10 évben. Gyakorlatilag eltűntek a hajlékony lemezek (floppy diszkek) és meghajtóik, illetve a hétköznapi használatban háttérbe szorultak az optikai meghajtók (CD, DVD, Blu-ray). Helyüket szinte teljes mértékben átvették a hordozható félvezető alapú tárolók (pen drive), melyek kapacitása

[4] https://www.amd.com/en/products/graphics/amd-radeon-rx-5700-xt (Letöltés: 2020.04.10.)
[5] https://www.nvidia.com/en-eu/geforce/graphics-cards/rtx-2080-ti/ (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT

ma már jóval meghaladja ezeknek a korábbi adathordozóknak a kapacitását (floppy: 1.44 MB, CD: 650 MB, DVD: 4,7/9.4 GB, Blu-ray 25/50 GB).

A hagyományos, az információt mágneses elven tároló merevlemezek is csak alacsonyabb áruk és nagyobb tárolókapacitásuk miatt tudják egyelőre fenntartani szerepüket, helyüket egyre inkább átveszik a méretre kisebb, alacsonyabb teljesítményigényű, lényegesen nagyobb sebességű – de jelenleg még azonos adatmennyiségre viszonyítva magasabb költségű félvezető háttértárolók (SSD – Solid State Drive).

Az információt mágneses elven tároló háttértárolók egy korszerű képviselője napjainkban a Seagate Exos 15E900 winchestere.



Specifications	
Capacity	900GB
Standard Model[1]	ST900MP0146
Seagate Secure™ Model (SED)[1,2]	ST900MP0156
Seagate Secure FIPS 140-2/Common Criteria Model[1,2,3]	ST900MP0166
Performance	
Average Latency (ms)	2
Sustained Transfer Rate (Outer to Inner Diameter, MB/s)	315 to 215
Mixed Workload Performance (at 5ms)	700
Max. Instantaneous Transfer Rate (SAS dual port) MB/s	2,400
Cache, Multi-segmented (MB)	256
Interface	12 Gb/s SAS
Intelligent NAND Endurance Management	Yes

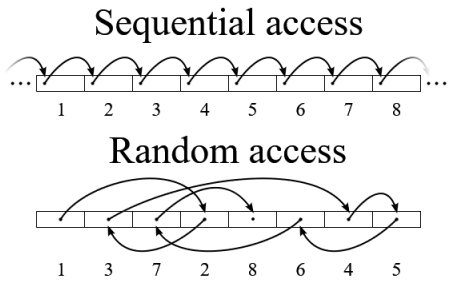
6.9 ábra – A Seagate Exos 15E900 winchester és főbb paraméterei[6]

[6] https://www.seagate.com/files/www-content/datasheets/pdfs/exos-15-e-900DS1958-2-1710GB-en_GB.pdf (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT

A háttértároló magasnak számító, max. 315 MB/s adatátviteli sebességet tud biztosítani (ez 2,52 Gb/s tényleges bitsebességnek felel meg). Illesztőfelülete az előző fejezetben bemutatott SAS-3 (Serial Attached SCSI) soros interfész, melynek 12 Gb/s maximális – még kódolást tartalmazó, így 9.6 Gb/s tényleges – sávszélessége bőven elegendő a feladathoz.

A merevlemez gyártó cégek alapvetően két paraméter javításán dolgoznak folyamatosan: az adathozzáférési sebesség emelésén és az adatrögzítés sűrűségének (az egységnyi felületen rögzíthető adatok mennyiségének) növelésén. Az átviteli sebesség sok paraméter függvénye, ebben meghatározó a fejek száma, a lemezek fordulatszáma, de még az adatrögzítés sűrűsége is befolyásolja ezt a paramétert. Rendkívül sok múlik az adatok elhelyezkedésén és az egyszerre sorban (szekvenciálisan) kiolvasható adatok mennyiségén. Az olvasófej alatt egymás után elhaladó adatok olvasása sokkal nagyobb sebességgel lehetséges, mint a kisméretű, véletlenszerűen elhelyezkedő fájlok adatai (ideértve a nagyobb fájlok töredezetten, különböző cilindereken és szektorokban elhelyezkedő részeit is). A 6.10 ábra véletlen sorrendű olvasást bemutató részén például a 2-es számú szektor után olvasásra kerülő 3-as számú szektor beolvasásáig a lemez egy teljes körülfordulását ki kell várnia az olvasófej vezérlőjének (emiatt a winchesterek legtöbbször „előre olvasnak” valószínűségi döntések alapján és a nem kért adatokat átmeneti cache tárolókban helyezik el).



6.10 ábra – Szekvenciális és véletlen sorrendű adathozzáférés[7]

A Seagate 2019 márciusában mutatta be a HAMR (Heat-Assisted Magnetic Recording) adatrögzítési eljáráson alapuló két karos (Mach 2 névre keresztelt, külön író és külön olvasó fejet tartalmazó) merevlemezét. Az olvasási sebességet közel 500 MB/s-re emelik, az ezek szerint működő 16 és 20 TB változatok 2020 őszén meg is jelentek.

Mint a bevezetőben már említettük, a háttértárolók piacán egyre nagyobb teret hódítanak a félvezető technológián alapuló tárak (SSD – Solid State Drive). A szilárdtest megnevezés arra utal, hogy az eszköz nem tartalmaz mozgó mechanikus alkatrészt, emiatt nem kopik, kevésbé sérülékeny (pl. ütésre), hangtalan, alacsonyabb a teljesítményfelvétele, így kevesebb veszteséghőt termel. Mivel az adathozzáférések nem igényelnek mechanikai mozgást (fejmozgatást, mágneslemez elfordulást), adathozzáférésük sokkal egyenletesebb és gyorsabb. A korábbi merevlemezekkel való egyszerű felcserélhetőség érdekében a számítógépekhez azok megszokott blokkos adatelérést biztosító interfészeinek egyikén csatlakoznak (SATA, SAS, PCI Express, USB stb.)

[7] Wikipédia: Sequential access, https://en.wikipedia.org/wiki/Sequential_access (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT

Technológiailag az SSD-k nem felejtő NAND szervezésű (szekvenciális elérésű) memóriák. Vezérlésük nem egyszerű, mivel – mint tudjuk – a flash memóriák programozása az elszigetelt vezérlőelektróda túlfeszültséggel történő feltöltésén és kisütésén alapul, amelyet a tárolócella csak adott számban képes elviselni, utána a cella tönkremegy. Ez a szám azonban olyan magas (10^6-10^7 művelet), hogy átlagos használat mellett – amihez még a flash belső vezérlője arra is ügyel, hogy egyes tárolóterületek lehetőleg ne kerüljenek túl gyakran újraprogramozásra – a gyártók az egységek adatlapjain legalább 10 évig garantálják az adatok biztonságos tárolását. A biztonság mind az elemek esetleges tönkremenetelére (és ilyenkor tartalék tárolóterülettel történő helyettesítésre), mind a szigetelő gate-en felhalmozott töltések (amik az információ tárolói) minimális szivárgására is vonatkozik.

Felépítésükből adódóan az SSD-k sebessége jóval meghaladja a mechanikus mozgást igénylő winchesterekét, ugyanakkor a tárolható adatmennyiség egy egységben kisebb. Már 1996-ban, mikor a flash memóriákat még „csak” nem felejtő programmemóriaként alkalmazták a számítógépekben, felfedezték annak lehetőségét, hogy egyetlen tárolócella helyén egymás fölé épített ún. többrétegű architektúrában több információ tárolható (3D). A bitsűrűség is folyamatosan fejlődik az 1 bitet (SLC: Single-Level Cell), 2 bitet (MLC:Multi-Level Cell), 3 bitet (TLC: Triple-Level Cell), 4 bitet (QLC: Quad-Level Cell), vagy 5 bitet (PLC: Penta-Level Cell) tárolni képes cellák formájában. Ugyanakkor tudnunk kell, hogy a több-bites architektúrában tárolt információk hozzáférési sebessége kisebb, azaz a nagyobb tárolt adatmennyiség az adatátviteli sebesség rovására megy.

Nézzünk ezek után konkrét mai SSD tárolókat!



MODEL	Intel® SSD 665p
Capacity and Form Factor	M.2 2280-S3-M 1024 GB (1 TB), 2048 GB (2 TB)
	Height and Weight: 80mm, <10 grams
Interface	PCIe 3.0x4, NVMe
Media	96-layer, QLC, 3D NAND
Performance	Sequential Read: Up to 2,000 MB/s; Sequential Write: Up to 2,000 MB/s
	Random 4KB Reads: Up to 250,000 IOPS; Random 4KB Writes: Up to 250,000 IOPS
Endurance	1 TB: 300 TBW 2 TB: 600 TBW TBW=Terabytes written
Power	Active: 100mW, Idle: 40mW
Operating Temperature	0° C to 70° C
Warranty	5-year limited warranty

6.11- Az Intel SSD 665p háttértárolója[8]

[8] https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/665p-ssd-series-brief.pdf (Letöltés: 2020.04.10.)

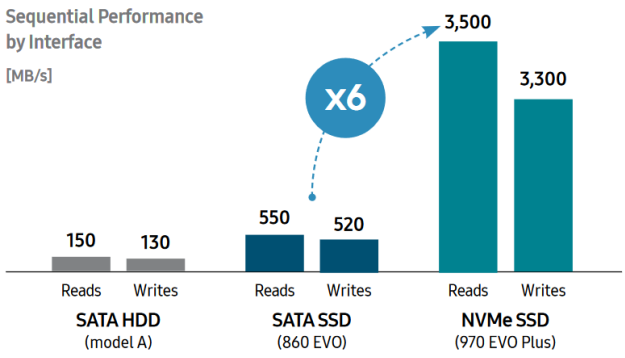
Copyright © 2021 / BME AUT

Az Intel SSD 665p tárolója QLC 3D NAND technológiára építve 1 és 2 TB-os háttértárolót kínál a korábbiakhoz képest igen figyelemreméltó 2 GB/s szekvenciális adathozzáférési sebességgel. Az ehhez szükséges tápenergia-teljesítményt (100mW) és az élettartamot (300/600 TBW – Terabytes Written) is érdemes figyelnünk – utóbbinál pont az optimalizált adatelhelyezés miatt az írások száma nem az egyes cellák írhatóságának a számát jelenti.

Meg kell még említsünk két fontos tulajdonságot a paraméterek jobb megértéséhez: bár az SSD tárolók adathozzáféréseit nem korlátozza a mechanikus elemek mozgása, az egyes cellaterületek elérhetősége, a cellák egyenletes használatát biztosító optimalizáló adatelhelyezés, a belső cache memória működtetése és az illesztő interfész tulajdonságai miatt az SSD-k esetében is megkülönböztetik a szekvenciális hozzáférést a véletlenszerű adateléréstől. Ennek jellemzésére a sokkal használhatóbb IOPS (I/O per Second) mennyiséget vezették be. Értéke önmagában keveset jelent, tudni kell hozzá, hogy a HDD/SSD milyen beállítások mellett érte el az adott értéket. A legfontosabb ilyen paraméter a blokkméret, az egyszerre írt/olvasott adatmennyiség mérete – kisméretű (legtöbbször 4 kB) blokkok esetén egészen más értékeket kapunk, mint pl. nagyméretű (1 MB) blokkméret esetén. Az Intel SSD 665P háttértároló esetében 4 kB-os blokkok mellett 250 000 IOPS értéket garantál a gyártó, azaz véletlenszerű adathozzáférés mellett a sávszélesség már csak 4 kB · 250 000 /s = 1 GB/s a korábban említett 2 GB/s érték helyett.

Az adatátvitel speciális szervezése miatt az SSD gyártók többsége az NVM Express (Non-Volatile Memory Host Controller Interface Specification Express – rövidítve csak NVMe) protokollt használja. A szabványt kifejezetten a félvezető memóriák tulajdonságait figyelembe véve alakították ki, segítségével a host hardver és szoftver nagymértékben ki tudja használni a modern SSD-k párhuzamosítási lehetőségeit. Alkalmazásával csökken az I/O hozzáférések overhead-je. A korábbi hasonló interfészeket a lassabb merevlemezes meghajtókhoz alakították ki, ahol az időkésés a CPU parancs és az adatátvitel között nagy volt, valamint az adatátvitel sebessége is sokkal lassabb volt, mint a félvezető memóriák esetében. Az Intel SSD 665P háttértároló esetében a hétköznapi gyakorlatban sokszor tévesen használt „M.2-es SSD” megnevezés a meghajtó fizikai formáját, a PCIe 3.0x4 a fizikai interfészt (PCI Express 3.0, 4 lanes – a 6.1. táblázat alapján 31.5 Gb/s ≈ 4 GB/s), míg az NVMe az alkalmazott kommunikációs protokollt jelenti.

A protokoll hatékonyságát mutatja a 6.12 ábra a következő bemutatásra kerülő eszköz esetében:



6.12 ábra – Az NVMe protokoll hatékonysága (PCIe 3.0 x4 interfész mellett)

Copyright © 2021 / BME AUT

Az SSD háttértárolók egyik legnagyobb gyártója a világon a Samsung cég. Csúcsmodellje és annak fontosabb paraméterei a 6.13 ábrán láthatók:



		970 PRO		970 EVO Plus			
Model Code		MZ-V7P1T0	MZ-V7P512	MZ-V7S2T0	MZ-V7S1T0	MZ-V7S500	MZ-V7S250
General Feature	Application	Client PCs					
	Capacity	1TB	512GB	2TB	1TB	500GB	250GB
	Form Factor	M.2 (2280)					
	Interface	PCIe Gen 3.0 x4, NVMe 1.3					
	Dimension (WxHxD)	80.15 x 22.15 x 2.38 (mm)					
	Weight	Max 8.0 g					
	Storage Memory	Samsung V-NAND 2-bit MLC		Samsung V-NAND 3-bit MLC			
	Controller	Samsung Phoenix Controller					
	Cache Memory	1GB LPDDR4	512MB LPDDR4	2GB LPDDR4	1GB LPDDR4	512MB LPDDR4	
Performance	Sequential Read	up to 3,500 MB/s		up to 3,500 MB/s			
	Sequential Write	up to 2,700 MB/s	up to 2,300 MB/s	up to 3,300 MB/s		up to 3,200 MB/s	up to 2,300 MB/s
	Random Read (4KB, QD32)	up to 500K IOPS	up to 370K IOPS	up to 620K IOPS	up to 600K IOPS	up to 480K IOPS	up to 250K IOPS
	Random Write (4KB, QD32)	up to 500K IOPS		up to 560K IOPS	up to 550K IOPS		
	Random Read (4KB, QD1)	up to 15K IOPS		up to 19K IOPS			up to 17K IOPS
	Random Write (4KB, QD1)	up to 55K IOPS		up to 62K IOPS	up to 60K IOPS		

6.13 ábra – A Samsung 970-es SSD családja[9]

A szintén M.2 formájú, PCIe 3.0 x4 interfésszel rendelkező háttértároló 0.25, 0.5, 1 vagy 2 TB tárolókapacitás mellett 3,5 GB/s szekvenciális adathozzáférést biztosít. Véletlenszerű adathozzáférés esetén 4 kB-os adatblokkok és 32 szintű I/O adatátviteli parancstároló (QD=Queue Depth) mellett az IOPS érték 500 000 (átszámítva 500 000 /s · 4 kB = 2 GB/s), míg 1 szint alkalmazásakor (I/O hozzáférések sorban állnak) ez 15-55k IOPS-re csökken. Ez is jól mutatja a belső SSD vezérlő és az NVMe protokoll hatékonyságát.

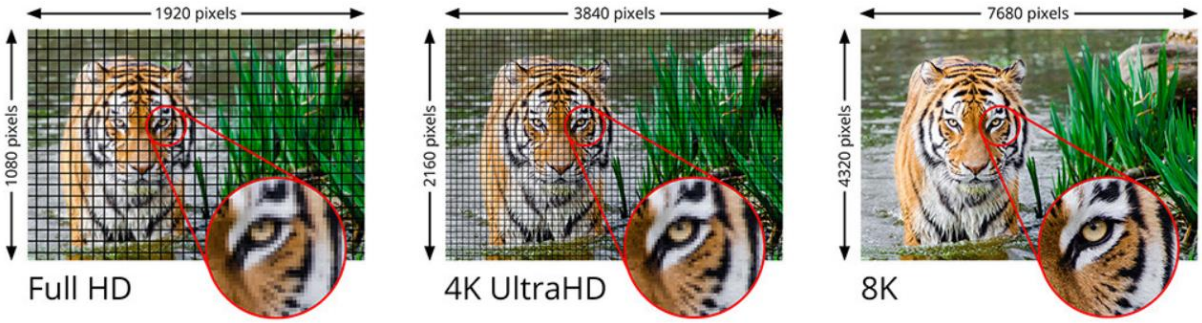
[9] https://s3.ap-northeast-2.amazonaws.com/global.semi.static/SSD_970_PRO_EVO_Plus_Brochure_190311.pdf (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT

Külső interfészek

A számítógéphez csatlakozó külső egységek közül a legnagyobb kihívást egyrészt a hálózati hozzáférést biztosító interfészek (ennek a tantárgynak a keretei között ezekkel nem foglalkozunk), másrészt a megjelenítő egységek – monitorok, digitális megjelenítő egységek, projektorok – képezik.

A digitális megjelenítő egységek – leginkább a digitális képrögzítés, a digitális játékok és a hallatlan tempóban fejlődő GPU-k követelményeinek kielégítésére – soha nem látott ütemben fejlődnek napjainkban. A „2K-4K-8K-10K” számháború jogosságát már sokan megkérdőjelezik azzal, hogy szemünk valóban képes-e a technikai szinten megkülönböztethetővé tett vizuális információk befogadására? A reklámok mindig megtalálják azt a lehetőséget, hogy mivel lehet elkápráztatni és meggyőzni a felhasználót a minél jobb (és természetesen egyre drágább) médiaeszközök gyakori cseréjére (ld. 6.14 ábra):



6.14 – A felbontás és a vizuális élmény szemléltetése[10]

De mik is ezek a rövidítések és mit jelentenek a műszaki szakemberek számára technikai paraméterekben?



6.15 ábra – A képernyőfelbontások szemléltetése[11]

A felbontások közötti gyors váltások folyamatát a full HD felbontás általánossá válása indította el. A HDTV (High Definition Television) sugárzási szabvány teremtette meg a digitális mozik

[10] https://vsbytes.com/hdmi-2-1/ (Letöltés: 2020.04.10.)
[11] https://www.dignited.com/26732/understanding-high-definition-video-resolutions/ (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT

élményéhez közelítő 1920 x 1080 pixeles képmegjelenítést (16:9 képarány). Természetesen ezt a számítógépes játékokat futtató számítógép monitorok és a laptopok kijelzői is követték, a dinamikusan fejlődő GPU-k számára sem okozott problémát a szükséges felbontás előállítása. A kijelzők technológiájának fejlődése (LED, OLED – Organic LED, QLED – Quantum LED) egyre nagyobb méretű és felbontású megjelenítők létrehozását tette lehetővé, az elterjedten használt „xK" rövidítés az egy sorban található képpontok száma ezerre kerekítve, tehát a full HD 1920 x 1080-as felbontása felel meg a 2K-nak, a 3840 x 2160 (területre négyszeres!) felbontás a 4K-nak, és így tovább.

A kijelzőre átviendő információk mennyiségét azonban nem csak a képernyő felbontása szabja meg. Figyelembe kell vennünk a színmélységet (color depth) és a képernyő frissítésének frekvenciáját (frame rate) is. Ezek ismeretében már elég nagy számok adódnak a használatos felbontások függvényében (6.4. táblázat).

Név	Vízszintes képpontok száma	Függőleges képpontok száma	Oldalarány (aspect ratio)	Összes képpont száma	24 bit/pixel 30 fps 60 fps [bit/s]	30 bit/pixel 30 fps 60 fps [bit/s]	48 bit/pixel 30 fps 60 fps [bit/s]
Full HD	1 920	1 080	16 : 9	2 073 600	1 492 992 000	1 866 240 000	2 985 984 000
					2 985 984 000	3 732 480 000	5 971 968 000
4K	3 840	2 160	16 : 9	8 294 400	5 971 968 000	7 464 960 000	398 131 200
					11 943 936 000	14 929 920 000	23 887 872 000
5K	5 120	2 160	21 : 9	11 059 200	7 962 624 000	9 953 280 000	15 925 248 000
					15 925 248 000	19 906 560 000	31 850 496 000
8K	7 680	4 320	16 : 9	33 177 600	23 887 872 000	29 859 840 000	47 775 744 000
					47 775 744 000	59 719 680 000	95 551 488 000
10K	10 240	4 320	21 : 9	44 236 800	31 850 496 000	39 813 120 000	63 700 992 000
					63 700 992 000	79 626 240 000	127 401 984 000

6.4. táblázat – Átviendő információ mennyisége a kijelzők esetében

Az elsősorban a videofilmek és a videojátékok kapcsán megnövekedett igények miatt napjainkban a 3x8=24 bites színmélységnél (true color) magasabbak az elvárások. A HDMI 1.3 szabvány már a színmélységet 30, 36 vagy 48 bitben (deep color) definiálja. A nagysebességű mozgóképek lejátszására a legújabb videojelszabványok a hagyományos 30 fps (frames per second) sebességnél magasabb, 60 sőt 120 fps sebességet követelnek (HDMI 2.1). A táblázat elméleti adatai jóval meghaladják a ténylegesen megkövetelt (és a szabványok által előírt) sávszélességeket, ennek oka az ezen interfészekben alkalmazott információtömörítés (DSC – Display Stream Compression). Ennek tulajdonságaival a következő fejezetekben foglalkozunk.

Összefoglalva: ha összevetjük a jelen fejezetben megfogalmazott igényeket az előző fejezetben bemutatott interfésztulajdonságokkal, azt mondhatjuk, hogy azok követelményeiben semmi túlzás vagy „presztízsharc" nem található. Napjaink perifériái megkövetelik azokat a nagyteljesítményű interfészeket, melyeket a gyártók folyamatosan fejlesztenek és évről évre bővítenek.

Copyright © 2021 / BME AUT

6.4 A 8b/10b kódolás

Két egység között átvinni kívánt adatok vonatkozásában nem tudunk megfogalmazni semmi olyan tulajdonságot, hogy az információfolyamban milyen adatok milyen sorrendben fordulhatnak elő. Ha ilyen tulajdonságot előírnánk, az adatátviteli interfész használhatósága kérdőjeleződne meg ezzel. Ugyanakkor a ma legelterjedtebben használt soros differenciális jelátvitel logikai és elektromos tulajdonságai a megfelelő működés érdekében megkövetelnek bizonyos speciális tulajdonságokat az átvinni kívánt információfolyamban. Mik ezek a specialitások?

- A nagysebességű adatátvitel megköveteli az átvitt fizikai jel egyenáramú kiegyenlítettségét, vagyis azt, hogy a bitfolyamban az egymást követő 1-esek és 0-k száma lehetőleg minél rövidebb bithosszon belül közel azonos legyen. Ekkor az előjelét váltogató differenciális feszültségjel középértéke közel nulla lesz, nem tolódik el jelentősen az átviteli csatorna egyenfeszültségű szintje egyik jelszint irányába sem, nem töltődnek fel a mindenképpen jelenlévő parazitakapacitások, melyek kisütése az átviteli sebesség ellen hat.
- Elegendő állapotváltozást kell tartalmazzon az átvitt jel az órajel vételi oldalon történő kinyeréséhez (hosszú 0 vagy 1 „folyamok” esetében jelváltozás hiányában nem végezhető el a vevő áramkör újraszinkronizálása).
- A kódszavak között meg kell tudni különböztetni néhány speciális, vezérlési és státuszcélokra felhasználható kódszót, melyek léte nem szabad korlátozza a felhasználót az adatszavak tartalmának tetszőleges összeállításában.

A fentieknek eleget tesz az 1983-ban publikált 8b/10b kódolási rendszer, amely azóta is széles körben használatos az aszinkron soros buszok jelátviteli interfészeiben. A rendszer a 8 bites kódszavakat 10 bites kódszavakba transzformálja. Ebben a kódolási rendszerben az 1-esek és a 0-k számának különbsége minden legalább 20 bites sztringben nem több, mint 2, ezen kívül legfeljebb 5 db azonos 1-es vagy 0 követi közvetlenül egymást a bitfolyamban. A kódolási rendszer sikerét jól mutatja, hogy az elterjedten használt interfészek (SATA, PCI Express, IEEE 1394b, SAS, Fibre Channel, SSA, Gigabit Ethernet, USB stb.) rendszere is alkalmazza. A választott 8/10 arány eléggé redundáns (az átvitt kód 20%-a redundancia), ezért az újabb rendszerekben már hasonló elvű, de jobb arányt biztosító (pl. 128b/130b) kódolást használnak.

A kódolási rendszer

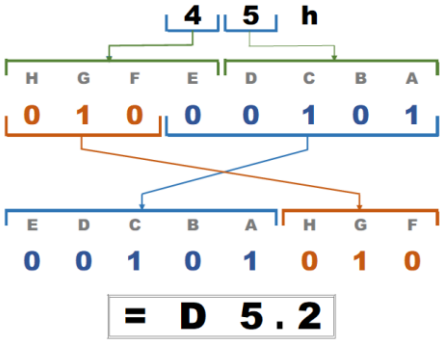
Amint a neve is mutatja, ez a kódrendszer a 8 bites adatoknak 10 bites entitásokat, más néven *szimbólumokat* vagy *karaktereket* feleltet meg. A 8 bites adatokat aszimmetrikusan két csoportra bontjuk: az alsó 5 bitet 6 bitbe, míg a felső 3 bitet 4 bitbe transzformálja. Vezessük be a következő egyszerű jelölésrendszert (6.16 ábra):

a) Jelöljük a bináris szám helyiértékeit a legkisebb helyiértéktől kezdve A..H betűkkel;

b) Cseréljük fel az 5 és a 3 bites csoportot (megtartva a bitek sorrendjét és az előbbi A..H betűk hozzárendelését;

Copyright © 2021 / BME AUT

c) A csoportok bináris értékeit decimális megfelelőjükkel helyettesítve kapjuk a Dx.y jelölést (x=0..31, y=0..7), ami egyértelműen azonosítja az eredeti számot.



6.16 ábra – Adatbájt jelölése a 8b/10b kódolási rendszerben

Az így kapott adatkód két részét egy-egy rögzített összerendelés (szabályrendszer) szerint alakítjuk át 6, illetve 4 bites bitcsoporttá. A 6 bites bitcsoport képezi a 10 bites kódszó magasabb helyiértékű bitjeit, míg a 4 bites az alacsonyabb helyiértékeket. A kódszó bitcsoportjait jelöljük (legmagasabb helyiértékű bitjeiktől kezdve) *a, b, c, d, e, i* és *f, g, h, j* betűkkel, ily módon a kapott kódszó egyes bitjeinek jelölése *a, b, c, d, e, i, f, g, h, j*. (Vigyázat, a betűk nem ABC sorrendben követik egymást!) A 6 bites bitcsoport kódolását a 6.5. táblázat mutatja:

Input		RD = −1	RD = +1
	EDCBA	abcdei	
D00	00000	100111	011000
D01	00001	011101	100010
D02	00010	101101	010010
D03	00011	110001	
D04	00100	110101	001010
D05	00101	101001	
D06	00110	011001	
D07	00111	111000	000111
D08	01000	111001	000110
D09	01001	100101	
D10	01010	010101	
D11	01011	110100	
D12	01100	001101	
D13	01101	101100	
D14	01110	011100	
D15	01111	010111	101000

Input		RD = −1	RD = +1
	EDCBA	abcdei	
D16	10000	011011	100100
D17	10001	100011	
D18	10010	010011	
D19	10011	110010	
D20	10100	001011	
D21	10101	101010	
D22	10110	011010	
D23 *	10111	111010	000101
D24	11000	110011	001100
D25	11001	100110	
D26	11010	010110	
D27 *	11011	110110	001001
D28	11100	001110	
D29 *	11101	101110	010001
D30 *	11110	011110	100001
D31	11111	101011	010100
K28	11100	001111	110000

* A jelölt 4 kódszó Kx.7 formában, valamint a K28.y kódszavak vezérlőkaraktereket alkotnak

6.5. táblázat – A 6 bites bitcsoport kódolása

Copyright © 2021 / BME AUT

A 4 bites kódcsoport kódolását pedig a 6.6. táblázat mutatja:

Input		RD = −1	RD = +1
	HGF	fghj	
Dx.0	000	1011	0100
Dx.1	001	1001	
Dx.2	010	0101	
Dx.3	011	1100	0011
Dx.4	100	1101	0010
Dx.5	101	1010	
Dx.6	110	0110	
Dx.P7 *	111	1110	0001
Dx.A7 *	111	0111	1000

Input		RD = −1	RD = +1
	HGF	fghj	
Kx.0	000	1011	0100
Kx.1 **	001	0110	1001
Kx.2 **	010	1010	0101
Kx.3	011	1100	0011
Kx.4	100	1101	0010
Kx.5 **	101	0101	1010
Kx.6 **	110	1001	0110
Kx.7 **	111	0111	1000

* A Dx.7 kódszó alsó bitcsoportja az előtte lévő felső bitcsoporttól függően a P (primary) vagy az A (alternate) változat is lehet annak érdekében, hogy az egymást követő 0-k vagy az 1-ek száma két egymást követő kódszó határán se haladja meg a 4-et.

** A jelölt vezérlőszavak alternáló kódolása speciális felismerést tesz lehetővé (nem részletezzük).

6.6. táblázat – A 4 bites bitcsoport kódolása

A fenti, első látásra talán furcsának tűnő szabályok könnyebb megértéséhez vegyük figyelembe a következőket:

a) A 8 bites adatot azért bontjuk meg aszimmetrikusan 3+5 bites csoportokra, hogy azokat +1 bittel kiegészítve páros bitszámú (4, ill. 6 bites) kódszórészeket kapjunk. Ezzel biztosítható ugyanis, hogy pl. a 6 bites kódszórészben 2, 3 vagy 4 db 0 vagy 1 bit felhasználásával a szükséges számú 32 kódkombináció előállítható legyen úgy, hogy 4 azonos bitnél több ne kerülhessen egymás mellé, és hogy ugyanez a kódszórészek határán is teljesíthető legyen, 3 db azonos értékű bitnél több ne kerülhessen a kódszórész szélére. Tekintsük ugyanis ezt az egyszerű kombinatorikai feladatot:
   i. 6 biten 3 db 1-est $N_1 = \binom{6}{3} = 20$ -féleképpen tudunk elhelyezni (az 1-esek sorrendje nem számít). Két esetben fog előfordulni 3 db azonos bit a kódszórész szélein: a 000111 ill. az 111000 kombinációk esetén. Ezt a két esetet vegyük ki (indoklást ld. később), így keletkezik 18 db 6 bites kódszórész.
   ii. 6 biten 2 db 1-est $N_2 = \binom{6}{2} = 15$, 4 db 1-est $N_3 = \binom{6}{4} = 15$ -féleképpen tudunk elhelyezni (az 1-esek sorrendje itt sem számít). A kapott $N_2 + N_3 = 30$ kódszórészből kizárjuk azokat az eseteket, ahol 4 azonos bit kerül a kódszórész valamelyik szélére (111100, 001111, 000011, 110000), marad tehát a feltételeinknek megfelelően 30-4 = 26 kombinációnk.

b) A 6 bites mezők tehát 3-3 vagy 4-2 arányban tartalmaznak 1-es és 0-s biteket. Ha a 0-s és 1-es bitek száma megegyezik, a kód minden további nélkül használható, nem változtat a korábbi bitfolyamban kialakult 0-1 egyensúlyon. Ha viszont eltérő (és ez az alkalmazott 4-2 arány miatt csak ±2 lehet), akkor a kódszórész és a negáltja is rendelkezésre áll kódolásra a 0-k és 1-esek számának lehető legrövidebb szakaszon

Copyright © 2021 / BME AUT

történő kiegyenlítéséhez. Ha az eddigi bitsorozatban az 1-esek aránya volt magasabb, akkor a jelen lépésben csökkenteni fogjuk azt, és megfordítva: alacsonyabb induló 1-es arány esetén növeljük azt. Az eltérés soha nem lehet nagyobb 2-nél.

c) A lehetséges kódszórészek számát figyelembe véve kaptunk tehát 18 db, az arányt nem változtató (egyértelmű) kódot, 26 darab olyan kombinációt, amelyek 13 párba rendezve (mindig egymás inverzei lesznek egy párban) alkalmasak a 0-1 arány eltérésének ellenkező irányba fordítására, végül az a) i. pontban kivett kombinációpárral együtt kapjuk a kiinduló 5 bitet lefedő 32 szükséges kódszórészt.

d) A kisebb (4 bites) kódszórészekre fentiek értelemszerűen alkalmazhatók: 1, 2 vagy 3 db 1-es elhelyezésével kapunk 4 db „egyensúlyban lévő” kódot (melyek határain nem fordul elő 2 azonos bit) és 5 db olyan kódszórész-párt, melyekben a 0-k és egyesek száma ±2-vel eltér, a párok itt is egymás inverzei. A párok közül 2 pár olyan, hogy a kódszórész határán 3 azonos bit fordul elő (Dx.P7, Dx.A7).

Fentiekkel már csak egy lépésre vagyunk a teljes megoldástól: az algoritmus teljes megértéséhez még be kell vezetnünk a görgetett paritáskülönbség (más fordításban futó diszparitás, eredeti nevén running disparity – RD) fogalmát. A kódoló ezen státuszbit segítségével ellenőrzi mind a kódszó két részének, mind az egymást követő kódszavaknak az összeállításánál, hogy az a 0-k és az 1-esek egyensúlya a lehető legrövidebb időn belül helyreállításra kerüljön. A folyamatot a 6.7. táblázat segítségével tudjuk megérteni:

a) Egy kódszó összeállításánál figyeljük, hogy az előző kód összeállítása után hogyan áll a 0-k és az 1-esek egyensúlya. Ha a megelőző bitfolyamban több 1-es fordult elő, akkor RD értéke pozitív (RD = +1), több 0 esetén negatív (RD = –1). Egy bitsorozat indításakor definíció szerint RD = –1. A 6.7. táblázatban a ”Bejövő RD” mutatja az adott karaktert megelőző bitfolyam paritását.

b) RD értékétől függően választjuk a táblázat szerinti kódszórészeket, ettől a diszparitás értéke 0 vagy ±2 értékkel változhat. A két mező kódértékének kiválasztását követően RD értéke újra +1 vagy –1 értéket vesz fel, ez képezi a következő karakter kódolásának a bemenetét.

A Dx.y és a Kx.y kódok bitcsoportjait 9:4 (*abcdei*) ill. 3:0 (*fghj*) elnevezéssel jelölve az RD változásának lehetséges értékeit mutatja be a 6.7. táblázat. A ±2-es (dupla) kódpárok közül mindig azt választjuk, amely RD aktuális értékét az ellenkező előjel irányába mozdítja.

Bejövő RD	9:4 kód paritás	9:4 kód választás	Paritás jelenleg	3:0 kód paritás	3:0 kód választás	Kimenő RD
-1	0	0	-1	0	0	-1
				±2	+2	+1
	±2	+2	+1	0	0	+1
				±2	-2	-1
+1	0	0	+1	0	0	+1
				±2	-2	-1
	±2	-2	-1	0	0	-1
				±2	+2	+1

6.7. táblázat – A 0-k és 1-esek egyensúlyának fenntartása

Copyright © 2021 / BME AUT

A táblázatok és a leírt algoritmus alapján még nyitva maradt kérdések magyarázatai:

a) A két kódrész összeillesztésénél *abcdeifghj* bitsorrend keletkezik. Alkalmazott kódolásunk révén a 6 bites mezők végén maximálisan 3 db azonos értékeket tartalmazó 1-es vagy 0-s bitsorozat szerepelhet. 3 db 1-es esetén RD = +1, 3 db 0 esetén RD = –1 lesz, ilyenkor a 4 bites mezők elején mindíg úgy alakulnak a 0-1 bitek, hogy az összeillesztésnél soha nem keletkezhet 4-nél több azonos bit. (Az alternáló Dx.P7 és Dx.A7 páros azért szerepel duplázva a táblázatban, hogy ez a feltétel minden esetben biztosítható legyen.)

b) A kódszavak soros kiküldése az *a*-bittel kezdődően indul, két egymást követő kódszónál mindig a *ghj-abc* bitek követik egymást fizikailag. Az előző pontban elmondottak itt is igazak: két kódszó átmeneténél sohasem keletkezhet 4-nél több azonos bit (a D07 kódnál szükség lehet ehhez az a) i. pontban elkülönített 000111 ill. 111000 párosra, a Dx.P7 és Dx.A7 duplázott páros pedig itt is segít ennek a feltételnek a biztosításában).

c) A K28.0 .. K28.7 kódszavak (összesen 8 db) képezik az adatkódoktól eltérő vezérlőszavakat, ezek az adatfolyam biztos felismerését biztosítják és további, az egyes szabványokban részletezett vezérlési funkciókat látnak el. Kiemelendők a K28.1, K28.5 és a K28.7 vezérlőszavak (ún. ”kommakarakterek”), amelyeknek felismerését az is segíti, hogy ezek bármilyen bitfolyamban 5 egymást követő 0-t vagy 1-et fognak tartalmazni, ami adatkódok esetében soha nem fordulhat elő (ld. 6.8. táblázat). A K23.7, K27.7, K29.7 és K30.7 kódszavak szintén használhatók vezérlő kód szekvenciákban, azonban ezeket mindig a K28.x vezérlőszavakkal együtt alkalmazzák, mivel kódjaik nem térnek el a Dx.7 megfelelőiktől.

Kommakarakter	RD = −1	RD = +1
	abcdei fghj	abcdei fghj
K28.1	001111 1001	110000 0110
K28.5	001111 1010	110000 0101
K28.7	001111 1000	110000 0111

6.8. táblázat – A kommakarakterek könnyen felismerhetők

A szabványban rögzített, minden részletében pontosan előírt kódolás elvégzésére nagysebességű céláramkörök kaphatók különböző gyártók kínálatában, a kommunikációs célprocesszorok általában integrált formában tartalmazzák ezeket áramkörükön belül.

Egy egyszerű példa az elmondottak alkalmazására: 3Fh = D31.1 kódolása:

Input	RD = −1	RD = +1
HGFE DCBA	abcdei fghj	abcdei fghj
0011 1111	101011 1001	010100 1001

6.9. táblázat – A 3Fh = D31.1 adatbájt 8b/10b kódolása

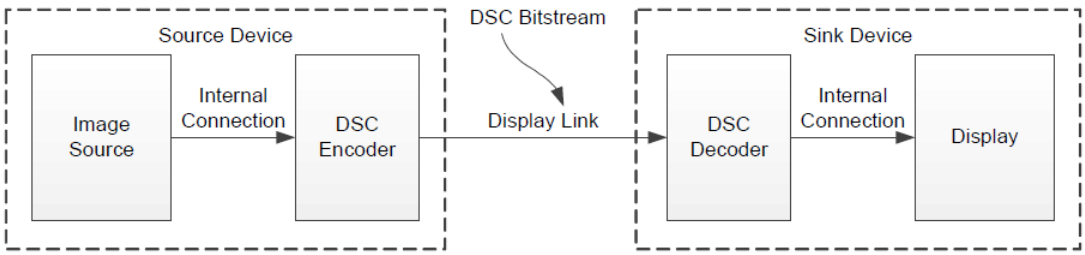
Copyright © 2021 / BME AUT

6.5 Videojel-tömörítés

Ebben a fejezetben nagyon rövid áttekintést adunk egy, a videojelek átvitelénél használatos tömörítési eljárásról. Jelen tantárgynak nem célja az információtömörítés elméletének és módszereinek elemzése, ugyanakkor a videojelet továbbító interfészek szükséges sávszélességének megértéséhez szükséges minimális ismeret erről a területről is.

Amint az előző fejezetben láthattuk, a kijelzők felbontása az utóbbi években szakadatlanul nő. A képek adattovábbítását végző interfészek egy darabig bírták a képernyőpontok (pixelek), a színmélység és a képfrissítési frekvenciából adódó natív sávszélességigényt, de az hamar látható volt, hogy információtömörítés nélkül a fejlődés elakad, a hardver adatátvitel sebessége nem képes ilyen mértékben növekedni.

2014-ben a VESA (Video Electronics Standards Association) kidolgozta a DSC (Display Stream Compression) szabványt, amelynek jelenleg az 1.2a verziója a legfrissebb. Eredetileg a DisplayPort csatolóhoz készült, de már a HDMI 2.1-es szabvány is alkalmazza. Az elve rendkívül egyszerű: a képi adatot tömörítve viszik át az interfészen, aminek ily módon kisebb sávszélességet kell biztosítania a továbbításhoz. Ehhez az információforrás kimeneténél egy DSC kódolóra, míg a fogadóegységnél egy DSC dekódolóra van szükség. A rendkívül egyszerűnek látszó feladatot nehezítik azok az elvárások, melyek szerint a kép minőségét meg kell őrizni és a késleltetési idő sem lehet számottevő.



6.17 ábra – A DSC kódolás/dekódolás elve[12]

A DSC szabványt „vizuálisan veszteségmentesnek” tekintik, ami azt jelenti, hogy bár az adattömörítés veszteséges (dekódolás után az információ nem állítható vissza teljes mértékben), azonban ez az emberi szem számára észrevehetetlen. Mindez úgy történik, hogy a tömörítés az információfolyamot a jel formátumától függően hozzávetőlegesen 50-33%-ára csökkenti, azaz az adatcsatorna terhelése 50-67%-kal csökken. A tömörítő algoritmusok világában az ilyen mértékű tömörítés nem számít jelentősnek, de ezen múlik annak „észrevehetetlensége”. Ily módon olyan információmennyiséget (olyan felbontású, színmélységű és képfrekvenciájú videojelet) is továbbítani tudunk, amire tömörítés nélkül a jelcsatorna nem lenne képes.

A tömörítési eljárás megértéséhez tudnunk kell, hogy az emberi szem a fényerő változásaira sokkal érzékenyebb, mint a színekére. Emiatt az alapelv az, hogy minden egyes képpont

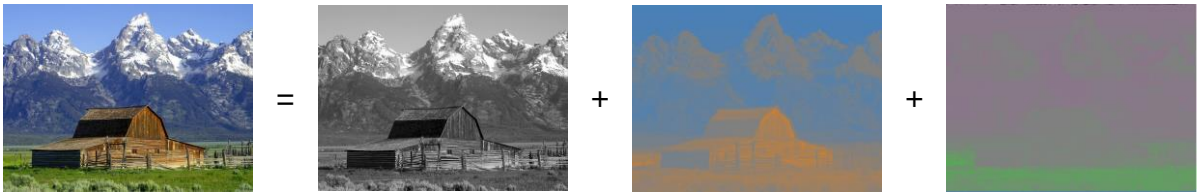
[12] VESA Display Stream Compression (DSC) Standard V 1.2a, https://vesa.org/vesa-display-compression-codecs/dsc/

Copyright © 2021 / BME AUT

fényerejét megőrizzük a tömörítéstől függetlenül, ugyanakkor a színinformáció szempontjából szomszédos képpontok összevonhatók anélkül, hogy a változást szemünk érzékelné. Ehhez először is színteret kell váltanunk: a megszokott RGB (Red-Green-Blue) színtérből áttérünk a számunkra legfontosabb fényerőt elkülönítetten tartalmazó YCoCg (Y=luminance, Co=chrominance orange, Cg=chrominance green). A választás indoklásával itt nem foglalkozunk, de megjegyezzük, hogy több hasonló színtér is használatos a gyakorlatban, elterjedt pl. az YCbCr (blue-red) színtér is. Az áttérés az RGB és az YCoCg színterek között rendkívül egyszerű:

$$\begin{bmatrix} Y \\ Co \\ Cg \end{bmatrix} = \begin{bmatrix} 0{,}25 & 0{,}5 & 0{,}25 \\ 0{,}5 & 0 & -0{,}5 \\ -0{,}25 & 0{,}5 & -0{,}25 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \qquad \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 0 & 1 \\ 1 & -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} Y \\ Co \\ Cg \end{bmatrix} \tag{6.1}$$

Mindezt vizuálisan is szemlélteti a 6.18 ábra:



6.18 ábra – Váltás RGB színtérből YCoCg színtérbe[13]

A transzformációt követően az egyes képpontokhoz tartozó világosságjel (Y) és színinformációkat (Co, Cg) eltérő módon kezeljük: a szemünknek fontosabb világosságinformációt minden képpontban megőrizzük, azonban a színinformációkban összevonást végzünk. Képezzünk 4 pixel széles és 2 pixel magas képernyőpont-csoportokat és vezessük be a következő jelölést:

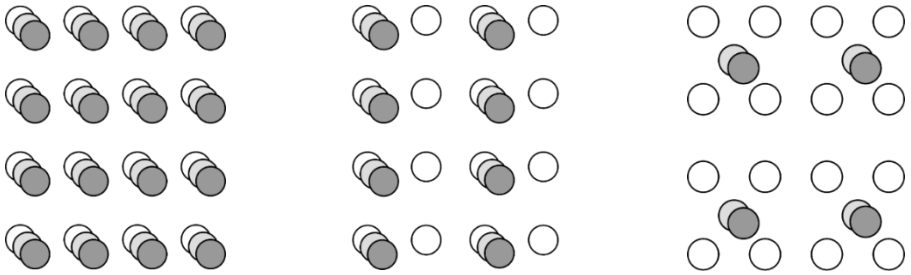
$$A : B : C \tag{6.2}$$

ahol A pixelcsoport szélessége (=4), B két színcsatorna mintavételezésének aránya A-hoz képest (pl. B=2 jelentése, hogy A-hoz képest csak minden második színinformáció kerül mintavételezésre), C jelentése ugyanaz, mint B-é, de az első helyett a második sorban az adott pixelcsoportban. A kapott számhármast szín-mintavételi aránynak (chroma subsampling) nevezzük. A furcsának tűnő definíciót az NTSC televíziószabvány vezette be, a pixelfrekvencia-arányait definiálták korábban hasonló módon.

A 6.19 ábrán a fehér körök a világosságjel-információt mutatják, a világos- és a sötétszürke körök pedig a két színjel-információt. A baloldali mintavételi arány (4:4:4) a tömörítetlen állapotot mutatja, valamennyi képpont mindhárom információját megtartjuk az eredeti képből.

[13] Wikipédia: YCoCg, https://en.wikipedia.org/wiki/YCoCg (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT



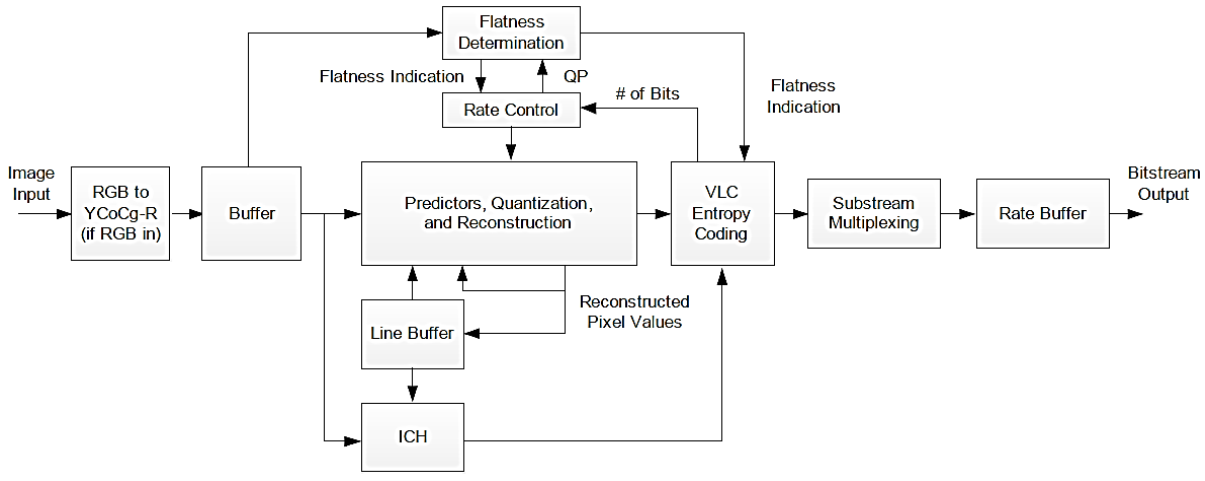
a) 4:4:4 b) 4:2:2 c) 4:2:0

6.19 ábra – Szín-mintavételi arányok[14]

A középső mintavételnél (ez a nagyon gyakran használt 4:2:2 mintavételi arány) minden világosságjel-információt megtart, azonban csak minden második színinformációt. Végül a jobboldali ábra (4:2:0 mintavételi arány) a soronként minden második színinformációt tart meg, és ezt is csak minden második sorban (a színinformációk akár átlagolhatók is a pixelnégyesek között, ezért ábrázoltuk a pixelek között középen a színinformációt). Pusztán ezen mintavételi arányokból az információtömörítés mértéke

a) (4x1+4x2)/4x3 = 12/12 = 100% (4:4:4)
b) (4x1+2x2)/4x3 = 8/12 = 66.7% (4:2:2)
c) (4x1+1x2)/4x3 = 6/12 = 50% (4:2:0)

A DSC tömörítés blokkvázlata a 6.20 ábrán látható. A működés részleteinek ismertetéséhez speciális kódolástechnikai ismeretek szükségesek, ezek részleteitől itt eltekintünk.

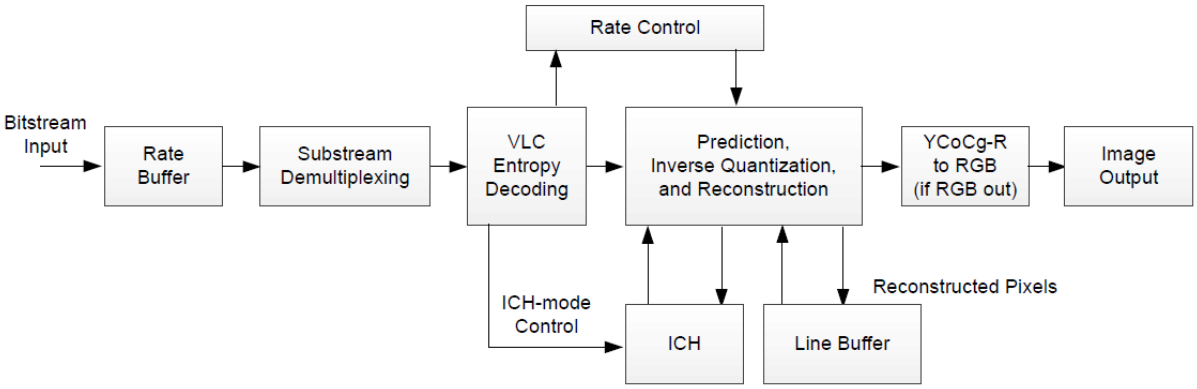


6.20 ábra – A DSC enkóder blokkvázlata[15]

[14] https://de.wikipedia.org/wiki/Farbunterabtastung (Letöltés: 2020.04.10.)
[15] VESA Display Stream Compression (DSC) Standard V 1.2a, https://vesa.org/vesa-display-compression-codecs/dsc/

Copyright © 2021 / BME AUT

Vételi oldalon természetesen a tömörített videojel dekódolására lesz szükség.



6.21 ábra – A DSC dekódoló blokkvázlata[16]

[16] VESA Display Stream Compression (DSC) Standard V 1.2a, https://vesa.org/vesa-display-compression-codecs/dsc/

Copyright © 2021 / BME AUT

7 A SATA interfész

A SATA (Serial ATA) interfész egy számítógépes pont-pont kapcsolat tárolóeszközök (tipikusan merevlemezes, félvezetős és optikai meghajtók) illesztésére. Elődje az ATA (Advanced Technology Atachment) szabvány által meghatározott párhuzamos interfész volt, melyet a SATA megjelenése után átneveztek PATA-nak (Parallel ATA). A SATA interfész 2003-as megjelenését követően vékonyabb és olcsóbb adatkábelével, a vele elérhető nagyobb adatsebességgel fokozatosan kiszorította párhuzamos elődjét. Az egyik legnagyobb winchester gyártó, a Western Digital 2013-ban hivatalosan is bejelentette az ATA interfésszel gyártott háttértárolók leállítását.

7.1 A SATA elődje - az ATA (IDE) interfész

A háttértárolók illesztését az első IBM PC-XT és IBM-PC AT számítógépekben még az ISA-8, illetve ISA-16 buszra csatlakozó bővítőkártyák segítségével valósították meg (7.1 ábra). Az illesztőegység igen bonyolult feladatot látott el a soros olvasó- és írójelekkel kommunikáló tárolók illesztésekor. Olvasáskor vezérelnie kellett a megfelelő sávra állást, fel kellett ismernie az olvasandó szektor azonosítóját, el kellett végeznie az adatszeparálást, az adatbiteket és az órajelet reprezentáló impulzusok – fluxusváltozások – szétválasztását (a mágneses adatrögzítés részleteivel terjedelmi okokból itt nem foglalkozunk). Ezen kívül hibafelismerő és hibajavító kódolást kellett végrehajtania, hibás olvasás esetén meghatározott számban meg kellett ismételnie az olvasást, a soros adatokat párhuzamosítania kellett és megfordítva, stb.



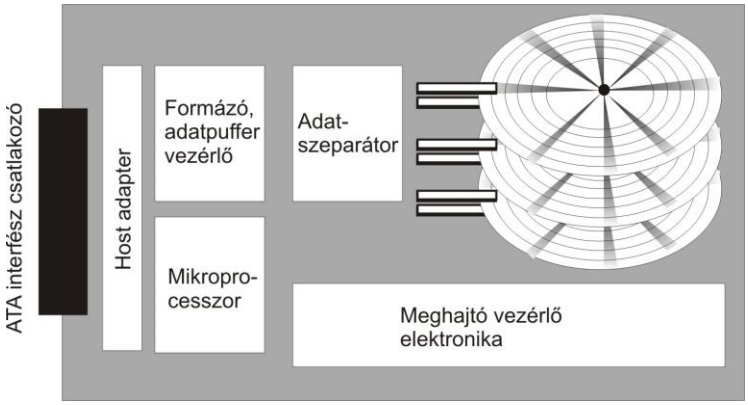
7.1 ábra – ISA buszos illesztőkártya háttértárolók illesztéséhez (WD42C22A-JU)[1]

A merevlemezek gyártói azonban hamar felismerték, hogy az ISA buszra illeszkedő logika beintegrálható magába a merevlemezes tárolóba és így egy lényegesen egyszerűbb interfészhez

[1] https://www.retropages.hu/theme_437.html (Letöltés: 2020.04.10.)

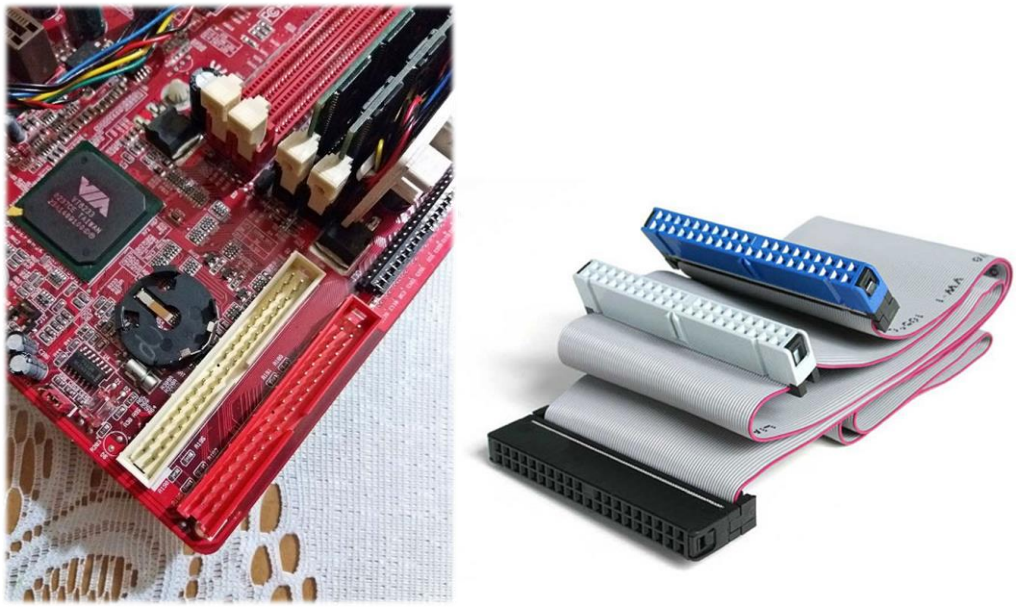
Copyright © 2021 / BME AUT

jutottak (ld. 7.2 ábra). Az első ilyen PC-t a Compaq hozta forgalomba 1985-ben. Maga az interfész egy 40 pólusú csatlakozón keresztül gyakorlatilag az ISA busz némileg egyszerűsített változata volt. Az interfészt logikusan IDE (Integrated Drive Electronics) interfésznek nevezték el, ám a szabványosításkor a hivatalos neve ATA lett.



7.2 ábra – ATA interfészű merevlemezes tároló

Amint a bevezetőben már említettük, az interfész egészen a 2010-es évekig a legelterjedtebben alkalmazott háttértárolók (winchester, floppy, optikai meghajtók – CD, DVD, Blu-ray) illesztéséhez szükséges illesztőfelület szerepét látta el. A széles, indirekt csatlakozókkal előre konfekcionált szalagkábelek jellemző „akadályai" voltak a számítógépek belsejéhez való hozzáférésnek.



7.3 ábra – PC alaplap ATA csatlakozókkal és szalagkábellel[2]

[2] https://en.wikipedia.org/wiki/Parallel_ATA (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT

A merevlemezes meghajtók sebességének és tárolókapacitásának állandó növekedése szükségszerűen megkövetelte az ATA szabvány újabb és újabb változatainak megjelentetését. Az ATA interfész kétféle üzemmódban tudott működni: PIO és DMA módban. PIO (Parallel I/O) módban a CPU, az alaplapi chipset és a meghajtó vezérlője együttesen bonyolították le az adatátvitelt, a CPU értelemszerűen ez alatt más feladatot nem tudott végrehajtani. DMA (Direct Memory Access) módban az átvitelhez nincs szükség a CPU segítségére, az adatok közvetlenül a RAM-ba kerülnek (viszont ezalatt a memória foglalt a CPU számára is). Az egyre fejlődő ATA szabványok ezt a kétféle üzemmódot fejlesztették lépésről lépésre. A fejlődés állomásait a 7.1. táblázat mutatja.

Év	Név	Új adatátviteli mód	Tulajdonság/újdonság	Drive kapacitás
1985	**IDE (Pre-ATA)**	PIO 0	22 bit LBA	2 GB (0,002 TB)
1994	**ATA-1**	PIO 0, 1, 2	28 bit LBA (16 cyl +4 head + 8 sec)	137 GB (0,137 TB)
		Single-word DMA 0, 1, 2		
		Multi-word DMA 0		
1996	**ATA-2**	PIO 3, 4	PCMCIA csatlakozó	
		Multi-word DMA 1, 2		
1997	**ATA-3**	Single-word DMA megszűnt	S.M.A.R.T.	
1998	**ATA/ATAPI-4**	Ultra DMA 0, 1, 2		
2000	**ATA/ATAPI-5**	Ultra DMA 3, 4	80 eres kábel	
2002	**ATA/ATAPI-6**	Ultra DMA 5	48 bit LBA	144 PB (144 115 TB)
2005	**ATA/ATAPI-7**	Ultra DMA 6	SATA 1.0	
2008	**ATA/ATAPI-8**	nincs	SATA fejl., PATA megtart.	

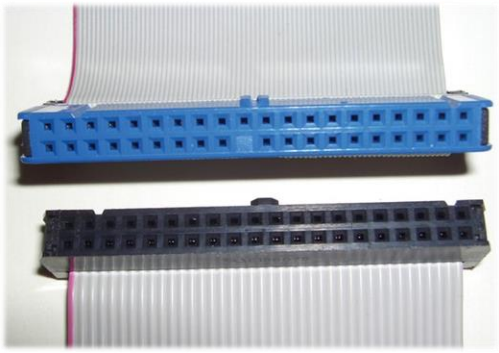
Mode	Adatsebesség [MB/s]	Mode	Adatsebesség [MB/s]
PIO 0	3,3	Ultra DMA 0	16,7
PIO 1	5,2	Ultra DMA 1	25
PIO 2	8,3	Ultra DMA 2	33,3
PIO 3	11,1	Ultra DMA 3	44,4
PIO 4	16,6	Ultra DMA 4	66,7
Single-word DMA 0	2,1	Ultra DMA 5	100
Single-word DMA 1	4,2	Ultra DMA 6	133
Single-word DMA 2	8,3		
Multi-word DMA 0	4,2		
Multi-word DMA 1	13,3		
Multi-word DMA 2	16,7		

7.1. táblázat – Az ATA szabványok fejlődése

A szabványok egyes változatainak jobb megértéshez a következőket kell tudnunk:

Copyright © 2021 / BME AUT

1) A PIO 0-4 üzemmódok az egy 16 bites szó átviteléhez szükséges idő felső határát specifikálták. Ezek rendre 600 ns, 383 ns, 240 ns, 180 ns, és 120 ns voltak.
2) DMA 0-2 üzemmódban a single-word átvitel egyetlen 16 bites szót, a multi-word átvitel több 16 bites szó blokkos átvitelét jelentette. A szabvány a teljes átviteli ciklus maximális idejét specifikálta rendre 960 ns, 480 ns és 240 ns-ban.
3) Az Ultra-DMA (UDMA) üzemmódban a STROBE jel fel- és lefutó élére is történt adatátvitel, ezzel azonos ciklusidő mellett megduplázták az adatátviteli sebességet.
4) Az ATAPI (ATA with Packet Interface Extension) elnevezést az ATA-4 szabvány vezette be. Ún. csomagparancsok definiálásával CD ROM, CD R/W eszközök, ZIP-meghajtók, mágnesszalagos egységek csatlakoztatása lett lehetséges az interfészen keresztül. (A mindig jelenlévő mágneslemezekkel szemben a cserélhető médiahordozókat használó meghajtók esetében a hordozó jelenlétének ellenőrzése, behúzása és kiadása is szükségessé vált.)
5) A logikai címzés (LBA) értelmezését és fejlődését a következő fejezetben tárgyaljuk.
6) A S.M.A.R.T. elnevezés a Self-Monitoring, Analysis and Reporting Technology szavak kezdőbetűiből áll össze. Segítségével az eszköz nyomon követi azokat a fellépő problémákat, melyek hibákhoz és így adatvesztéshez vezethetnek.
7) Az ATA-4 szabvány opcionálisan bevezette, az ATA-5 szabvány kötelezővé tette 66,7 MB/s adatátviteli sebességtől a 80 eres szalagkábelek használatát (és a 40/80 ér automatikus detektálását). Valamennyi plusz ér földpotenciálon van és befésülődik a korábbi 40 ér közé, csökkentve ezzel a jelvezetékek kapacitív csatolását és így a jelek közötti áthallást. A csatlakozó továbbra és 40 pólusú maradt, belsejében kellett gondoskodni a plusz erek földpotenciálra kötéséről (ld. 7.4 ábra).
8) Az ATA-8 szabvány célja a már folyamatban lévő soros ATA specifikációk továbbfejlesztése volt a párhuzamos ATA-7 specifikációk megtartása mellett.



7.4 ábra – 80 (fent) és 40 (lent) eres szalagkábel 40 pólusú ATA csatlakozókkal[3]

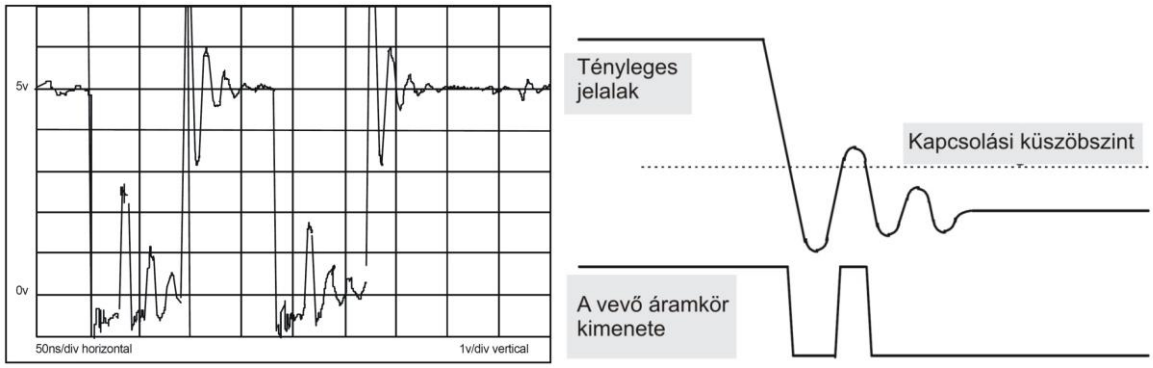
Az ATA szabványok szigorúan ügyeltek arra, hogy az újabb változatok megőrizzék a kompatibilitást a korábbiakkal. Ugyanakkor a fejlődés során az egyes specifikációkban az

[3] https://www.ifixit.com/Wiki/Hard_Drive_Interfaces (Letöltés: 2020.04.11.)

Copyright © 2021 / BME AUT

átviteli sebesség 3,3 MB/s-ról 133 MB/s-ra növekedett, ami elektromos problémák sorát vetette fel (reflexiók, áthallás, EMI stb.).

Az első ATA szabványok még nem írták elő a jelvezetékek lezárását, a meghajtó TTL áramkörök sebessége akkoriban még alacsonyabb volt, így nem voltak észlelhetők hullámjelenségek. A szokásos ATA kábelek hullámimpedanciája $Z_0 = 110\ \Omega$, maximális hosszúságuk L = 48 cm (18 inch) volt. A távvezetékmodellből pontosan is levezethető, de ökölszabályként azt az elvet használják, hogy ha a jelek változási ideje (fel- és lefutó él időtartama) kisebb, mint a kábel ún. futási idejének hatszorosa, akkor távvezetékként kell kezelni a kábelt, egyébként megfelelő a koncentrált paraméterű LC-modell. A szokásos ATA kábeleken a terjedési sebesség kb. $\tau = 0{,}6 \cdot c_0$ (c_0 a fény vákuumban mért terjedési sebessége), amiből a terjedési idő $t_{futás} = L/\tau = 2{,}67$ ns. A host és az eszközmeghajtó áramkörei gyorsabbak, mint $6 \cdot 2{,}67 = 16$ ns. Emiatt a vezetékeken rezgések léphetnek fel, ami megzavarja a jelátvitelt.



7.5 ábra – Tipikus rezgés az ATA interfész jelén, és annak hatása

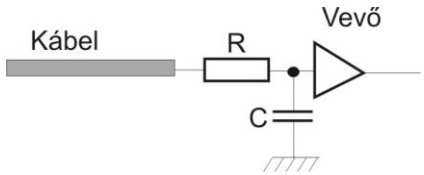
A 7.5 ábrán jól látható, hogy a jelvezetéken fellépő rezgés miatt a vevő áramkör kimenetén hamis jelváltás lép fel. A megoldás a fenti nehézségre a kábel mindkét végének helyes lezárása, de ez a visszafelé inkompatibilitás és a költséges nagyáramú meghajtók miatt a gyakorlatban nem jöhet szóba. A szabvány ezért az ATA-4-es változattól (az UDMA mód megjelenésétől) egy nem teljesen egzakt, de a korlátozásoknak eleget tevő megoldás-hármast javasol. Ezek: a meghajtóoldali részleges soros lezárás, a vevőoldali AC lezárás és az élmeredekség-szabályozás.

Az ATA specifikáció előírja, hogy 4 mA terhelés esetén a kimeneti meghajtó L szintje nem haladhatja meg a 0,5 V-ot. Egy, a kimenettel sorba kötött (lezáró) ellenállás viszont megemeli a tényleges kimenet feszültségét. Ha feltételezzük, hogy az eredeti kimenet L szintjén a maximális feszültség 0,4V volt, akkor egy soros 22 Ω-os (±5%) ellenállás beiktatása ezen az értéken $22\ \Omega \cdot 4$ mA = 88 mV-ot emel, ami a specifikációba még belefér. A soros 22 Ω csak részleges lezárást jelent ugyan, de a reflexiók és az általuk okozott rezgések nagysága már jelentősen csökken.

A vevőoldali AC lezárásnál megint a specifikációból kell kiinduljunk. H szinten a meghajtónak 400 µA-nél nagyobb áramot nem kell tudnia leadni. A fogadó eszköz (amiből kettő is lehet azonos kábelen) bemenő ellenállása minimálisan $Rbe = 2 \cdot 5V / 400\ \mu A = 25\ k\Omega$ kellene legyen,

Copyright © 2021 / BME AUT

ami 110 Ω-os szalagkábel esetén gyakorlatilag végtelen értéket, azaz teljes lezáratlanságot jelentene. A DC lezárás tehát nem járható út, ezért a szabvány az AC lezárást javasolja, amit a 7.6 ábra mutat:



7.6 ábra – Vevőoldali AC lezárás az ATA buszon (R = 82 Ω, C = 10 pF)

A lezárás nagy frekvenciákon úgy viselkedik, mintha ohmos lezárás lenne, a bemenő feszültséget L szinten a soros ellenállás max. $\Delta U = 2\ mA \cdot 82\ \Omega = 0{,}164$ V-tal emeli meg (ne felejtsük el, hogy a max. 4 mA meghajtó áramon két egységnek kell osztoznia), ami a meghajtó $V_{OLmax} = 0{,}5V$ és a fogadó $V_{ILmax} = 0{,}8$ V távolságába még belefér.

A szabvány a minimális jelváltozási meredekségre 1,25 V/µs (UDMA 5 és 6 esetén 1 V/µs) értéket ír elő. A modern meghajtó áramkörök ezt jelentősen túllépik, ezért a kimenetet a soros lezáró impedancia után egy kapacitással a föld felé kell söntölni. A söntölő kapacitások összege a be- és a kimeneti oldalon nem haladhatja meg a szabvány által előírt 50 pF értéket.

Amint azt már korábban említettük, az áthallás csökkentésére a nagysebességű UDMA üzemmódokban a szabvány előírja minden két jelvezeték között egy földpotenciálon lévő ér vezetését (80 eres szalagkábel).

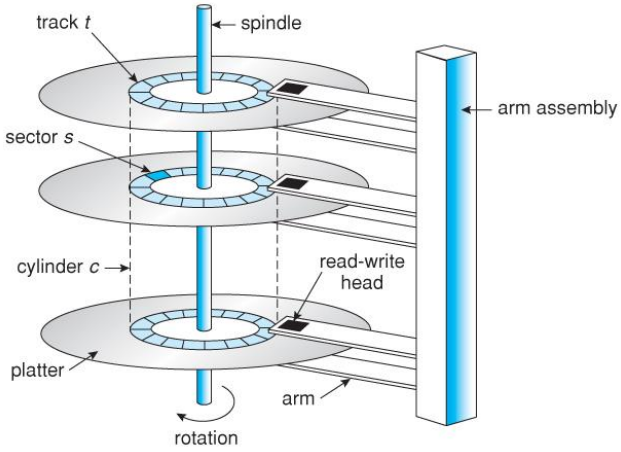
7.2 A háttértárolók fontosabb jellemzői

A merevlemezes háttértárolók (szokásos elnevezései: hard disc, HDD, winchester), ahogy a nevük is mutatja, adatok nagy tömegű, nem felejtő tárolására szolgálnak. A merevlemez az adatokat mágnesezhető réteggel bevont lemezeken tárolja, amelyeket a lemezoldalak felett mozgó író-/olvasófejek segítségével írjuk vagy olvassuk. A lemezek állandó sebességgel forognak (általában 5400 / 7200 ford/perc fordulatszámmal, de léteznek 10 – 15 000 ford/perc fordulatszámú meghajtók is). A fejek kb. 1 nm (ez lecsökkenhet akár 0,2 nm-re is) vastag légpárnán repülnek a lemezek felett, így a lemezekre kerülő legkisebb szennyeződés is végzetes lehet a működés szempontjából. A merevlemezes tárolók emiatt légmentesen zárt házban helyezkednek el.

Copyright © 2021 / BME AUT



7.7 ábra – Meghibásodott merevlemez: szennyeződés került a lemez és az író-/olvasófej közé[4]



7.8 ábra: Merevlemezes tároló felépítése[5]

Egy winchesterben általában több lemez (platter) is található, jellemzően mindkét oldalát használják adattárolásra. A lemezekre azonos középpontú, koncentrikus körök, ún. sávok (track) mentén írjuk fel az információt, az egymás alatt elhelyezkedő lemezeken (ideértve azok mindkét oldalát) azonos sugáron elhelyezkedő sávokat együttesen cilindernek (cylinder) nevezzük. Mivel a lemez mindkét oldalát használjuk, a fejek (head) száma a duplája a lemezekének. A lemezoldalakat képzeletben egyenletesen körcikkekre osztjuk, melyek a sávokon körív-szeleteket, ún. szektorokat (sector) határoznak meg. Időnként szükség van több szektor logikai összefogására, ezeket csoportoknak (cluster) nevezzük.

[4] Wikipédia: Merevlemez, https://hu.wikipedia.org/wiki/Merevlemez (Letöltés: 2020.04.11.)
[5] https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/10_MassStorage.html (Letöltés: 2020.04.11.)

Copyright © 2021 / BME AUT

Capacity (Blocks)

Sector Size	900GB Models	
	Decimal	Hex
512	1,758,174,768	68CB9E30h
520	1,720,961,544	6693CA08h
528	1,673,624,336	63C17B10h
4096	219,771,846	D1973C6h
4160	215,480,776	CD7F9C8h
4224	211,150,576	C95E6F0h

	900GB Models	
Drive capacity	900	GB (formatted, rounded off value)
Read/write data heads	6	
Bytes per track	1055/1081	KBytes (avg, rounded off values)(4KN or 512E / 512N)
Bytes per surface	150,000	MBytes (unformatted, rounded off values
Tracks per surface (total)	138.8	KTracks (user accessible) (4KN or 512E / 512N)
Tracks per in	335	KTPI (average) (4KN or 512E / 512N)
Peak bits per in	2091/2277	Kbits/in
Areal density	684/744	Gb/in2 (4KN or 512E / 512N)
Disk rotation speed	15K	RPM
Avg rotational latency	2.0	ms

7.9 ábra – A A Seagate Exos 15E900 winchester belső paraméterei[6]

A 7.9 ábrán az előző fejezetben már megismert Seagate Exos 15E900 winchester jellemző paramétereit látjuk. A szektorok méretét alapértelmezésben 512 bájtra állítják. A tárolóban 3 lemez (6 fej) található, lemezoldalanként 150 GB tárolását tudják elvégezni. Minden lemezt 138 800 sávra (track) osztanak, egy-egy sávon 1081 kbájt található – a paraméterek alapján gyors fejszámolással 138 800 sáv/oldal · 1081 kbájt/sáv = 150 042 800 kbájt/oldal. A lemezek viszonylag magas fordulatszámon (15 000 ford/perc) forognak, és mivel valószínűségi alapon egy fél fordulat szükséges a következő szektor eléréséhez, így a forgásból adódó átlagos késleltetési idő 2 ms.

A szektorok a legkisebb hozzáférési adategységek mind írás, mind olvasás esetén. Kezdetben az egymástól függetlenül mozgó egységek egyszerű szétválaszthatósága érdekében

- a cilinder sorszámhoz vagy indexhez (melyik sávon vagyunk = a fejeknek milyen mélyen kell „benyúlniuk” a lemezek közé),
- a fejek számához (melyik lemez melyik oldalán vagyunk, melyik fejet kell aktiválni) és
- a szektor sorszámához vagy indexéhez (melyik körcikkben vagyunk a körön belül, azaz mekkora az egy körön belüli szögelfordulás)

adott számú bitet rendeltek, ezek a mezők eleinte rendre 10, 8 és 6 bites hosszúságúak voltak. A bitek számából adódóan 1024 cilinder, 256 fej és 63 szektor volt megcímezhető (vigyázat, a lemezeken a 0. szektor foglalt, a szektorok számozása a cilindereken 1-től indul), a teljes tárolókapacitás 1024 · 256 · 63 · 512 ≈ 8,45 GB volt. Ezzel a számhármassal jellemezték az adatok elhelyezkedését a tárolólemezeken.

A host gépnek nem feltétlenül kell törődnie a winchester belső felépítésével (lemezoldalak, sávok, szektorok száma) a szektorokat gondolatban egyetlen hosszú sorba rendezzük, és egyetlen számmal címezzük meg, melyet LBA-nak (Logical Block Addressing) hívunk. Az LBA egyszerűen lineárisan számozza a szektorokat, az LBA 0 a (logikailag) legelső szektor, melynek fizikai azonosítói: cilinder 0, fej 0, szektor 1, ezt követi az LBA 1 (cilinder 0, fej 0, szektor 2), stb.

[6] https://www.seagate.com/files/www-content/datasheets/pdfs/exos-15-e-900DS1958-2-1710GB-en_GB.pdf (Letöltés: 2020.04.10.)

Copyright © 2021 / BME AUT

Az ATA-1 szabvány bevezette a 28 bites LBA-t, 16 bites cilinder, 4 bites fej és 8 bites szektorcímmel. Az elvi tárolókapacitás tehát 65 536 · 16 · 255 · 512 = 136,9 GB-ra nőtt. Ez a kapacitás nem feltétlenül volt kihasználható, mivel

a) a fizikai meghajtó paraméterei nem biztos, hogy ki tudták használni a rendelkezésre álló bitszámokat (pl. nem rendelkeztek 16 fejjel),

b) a kezdeti operációs rendszerek még mindig cilinder-fej-szektor számhármas formában állították elő a paramétereket, és adták át a BIOS INT13h szoftvermegszakításban. Ha ezek a bitmező méretek nem egyeztek az ATA szabvány előírásaival, a kettő közül csak a kisebbiket lehetett használni.

Emiatt bevezették az ún. CHS (cylinder-head-sector) paramétertranszformációt, amely megoldást jelentett a fenti problémákra. A transzformáció lényege a következő képletben foglalható össze:

$$LBA = (C \cdot HPC + H) \cdot SPT + (S - 1) \tag{7.1}$$

ahol

- C a cilinder sorszáma (vagy indexe);
- H a fej sorszáma;
- S a szektor sorszáma (indexe);
- HPC a fejek maximális száma (heads per cylinder) – diszk meghajtó paraméter
- SPT a szektorok maximális száma sávonként (sectors per track) – diszk meghajtó paraméter

A képlet megértése rendkívül egyszerű: minden sávnak önálló indexe van a *C·HPC+H* kifejezés révén; ezt *SPT*-vel szorozva kapjuk a lehetséges szektorok számát, amit *S-1* indexel. A diszk vezérlő számára a képlet inverze szükséges: adott LBA-ból visszanyerni C, H és S értékét:

$$\begin{aligned} C &= LBA \div (HPC \cdot SPT) \\ H &= (LBA \div SPT)\ mod\ HPC \\ S &= (LBA\ mod\ SPT) + 1 \end{aligned} \tag{7.2}$$

ahol ÷ az egész osztást, *mod* a moduló műveletet (az egész osztás maradékát) jelöli.

7.3 A soros ATA interfész

A merevlemezes tárolók egyre növekvő sebessége megkövetelte az ATA-interfész specifikációjának folyamatos módosítását. Az ATA-7 változatban specifikált Ultra DMA 6 üzemmód (133 MB/s) az elfogadható költségek mellett azonban a megvalósíthatóság felső határát jelentette. Az igények ezt is hamar túlszárnyalták, így más megoldásokat kellett keresni.

Copyright © 2021 / BME AUT

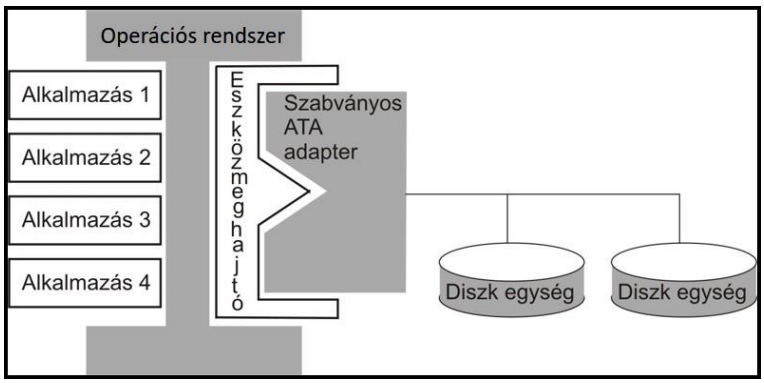
Az új megoldás a soros ATA, amely valójában a már kialakított párhuzamos ATA soros implementációja. Bevezetésként tekintsük a két interfész egy gyors összehasonlítását:

Az összehasonlítás tárgya	SATA	PATA
Másik használt név	Serial ATA	Parallel ATA
Jelenlegi helyzet	Használatban	Megszűnt
Sebesség	Gyors	Mérsékelt
Feszültség alatti csatlakoztathatóság (hot swap)	Támogatott	Nem támogatott
Külső interfész	Támogatott	Nem támogatott
Max. kábelhossz	1m (39.6 inch)	45,7 cm (18 inch)
Kábel mérete	Kevés ér	Sok ér
Átviteli sebesség	150 MB/s - 600 MB/s	16 MB/s - 133 MB/s

7.2. táblázat – A SATA és a PATA interfészek összehasonlítása

7.3.1 Általános áttekintés

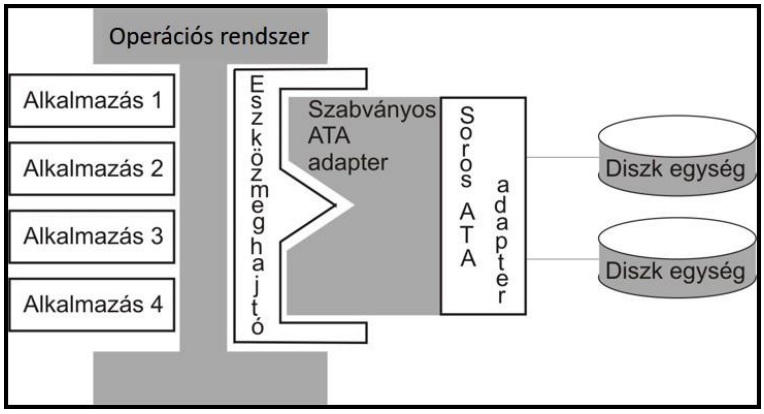
A soros ATA kifejlesztésekor alapvető szempont volt, hogy a párhuzamos ATA-interfészt kezelő szoftver rendszerek változtatás nélkül használni tudják. A korábbi rendszerek általános felépítését a 7.10 ábra mutatja. Két diszkegység volt csatlakoztatható egyetlen porton master/slave kommunikációval, egyetlen (szalag-) kábelt használva az illesztéshez.



7.10 ábra – A párhuzamos ATA interfész felépítése

Két diszkegység csatlakoztatását a soros ATA-interfészhez a 7.11 ábrán láthatjuk. A párhuzamos ATA-interfész valamennyi összetevője megmaradt, csak kiegészültek egy soros ATA adapterrel. A párhuzamos ATA-ra írt szoftver rendszerek a soros ATA-hoz ugyanúgy férnek hozzá, mintha semmi sem történt volna, az egyetlen különbség az, hogy a rendszer most mindkét diszk egységet mesternek látja, mintha azok két különböző porthoz csatlakoztak volna korábban. A soros ATA struktúra minden egyes diszk egységhez saját kábellel csatlakozik pont-pont topológiával, kiküszöbölve ezzel inicializáláskor az ütközések problémakörét.

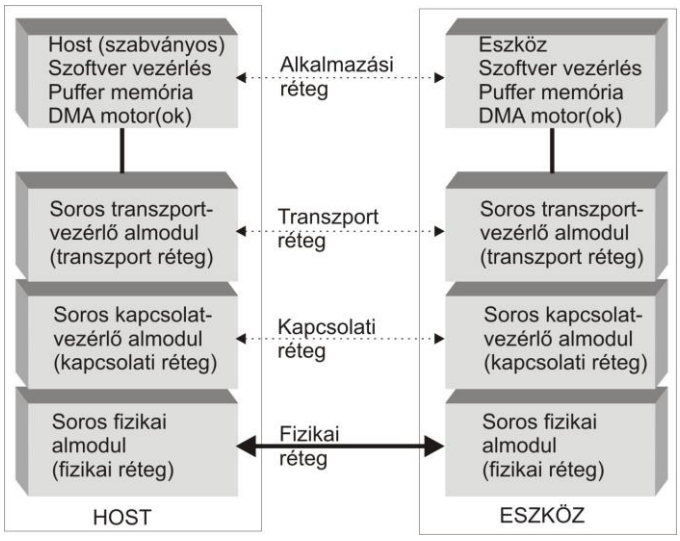
Copyright © 2021 / BME AUT



7.11 ábra – A soros ATA interfész felépítése

7.3.2 A SATA rétegszerkezete

Az OSI modell a rétegekbe rendezett rendszerek általános leírása, amelyben minden réteg csak az alatta található rétegek funkcióira támaszkodhat, az általa megvalósított funkciókat pedig a felette lévő rétegek számára nyújtja. Szerepe rendkívül fontos a különböző gyártók által kínált meghajtóprogramok szabad cserélhetőségéhez és összeilleszthetőségéhez. A referencia modell hét réteg hierarchikus szerkezetét írja le, a mindennapi gyakorlatban a szabványok és a gyártók általában ennek egy részhalmazát (leginkább az alsó rétegeket) definiálják és használják. A SATA-interfész specifikációjában is az alsó 3 réteg definíciója található meg részletesen, ezt mutatja be a 7.12 ábra: a fizikai réteg, az adatkapcsolati réteg és a transzportréteg.

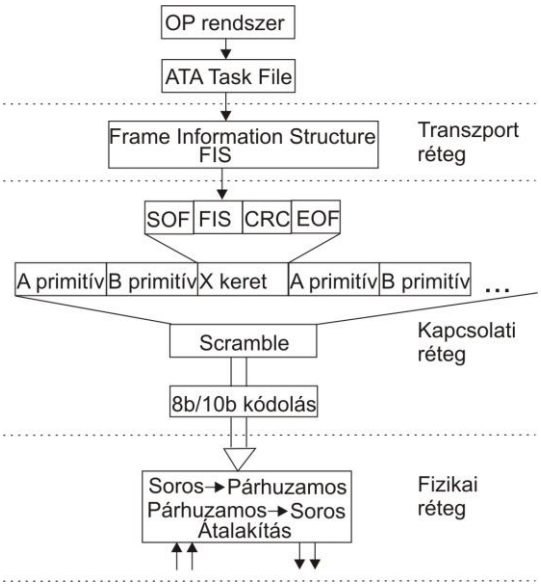


7.12 ábra – A soros ATA kommunikációs rétegei

Az egyes rétegek (almodulok) részletesebb funkcióit és ezek kapcsolatát a 7.13 ábrán láthatjuk (a rétegek részletesebb tárgyalására a következő fejezetekben térünk ki). A host a transzport almodullal egy regiszterinterfészen keresztül találkozik, amely ekvivalens a PATA hostadapter regiszterinterfészével. A SATA transzportvezérlő almodul az elvégzendő műveleteket akciók

Copyright © 2021 / BME AUT

sorozatára bontja le és ún. keretinformációs struktúrákban (Frame Information Structure = FIS) kommunikál az alatta található adatkapcsolati almodullal.



7.13 ábra – A SATA rétegek kapcsolata és funkciói

Az adatkapcsolati almodul a kapott struktúrát becsomagolja (Start of Frame, FIS, CRC, End of Frame) elhelyezi a kommunikációs primitívek között (ld. később), majd a szintén később elemzésre kerülő hálózatátviteli tulajdonságok javítása érdekében összekeveri (scramble) az információt, kódolja azt (8b/10b), és így adja át továbbításra a fizikai rétegnek. A soros-párhuzamos ill. párhuzamos-soros átalakításon és a fizikai továbbításon kívül a fizikai réteg kezeli az eszközök akár menet közbeni összekapcsolódását (Out Of Band = OOB jelek).

7.3.3 A fizikai réteg

A párhuzamos ATA esetében az interfész további fejleszthetőségét három tulajdonság akadályozta:

1) a sok vezetéket igénylő, szinkronizáció szempontjából nehezen kezelhető párhuzamos jelátvitel;
2) a TTL jelszint;
3) az aszimmetrikus jelátvitel használata.

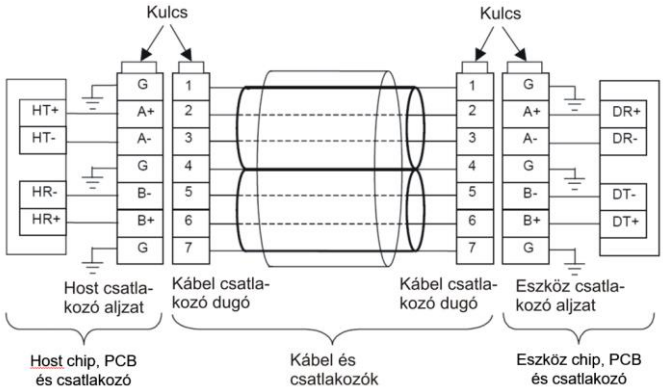
A SATA esetében mindhárom akadályt megszüntették. Az új megoldás legfontosabb tulajdonságai a következők:

1) 1,5 Gb/s, 3 Gb/s és 6 Gb/s adatátviteli sebesség NRZ kódolással;
2) kis jelszintű differenciális adatátvitel;

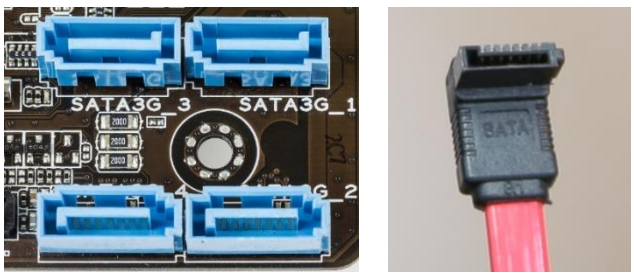
Copyright © 2021 / BME AUT

3) a 100 Ω-os hullámimpedanciájú kábel mindkét végén 100 Ω-mal szimmetrikusan lezárva;
4) a linkréteg 10, 20, 40 stb. bites párhuzamos kimenetét soros jellé alakítva továbbítja a vevő felé, illetve a vett soros adatokat ugyanilyen módon párhuzamosítva adja tovább a linkrétegnek;
5) a sávon kívüli (OOB) jelek vétele és adása;
6) tápfeszültség-sorrendiség kezelése (precharge) és sebességegyeztetés;
7) státuszjelzés a linkréteg felé: eszköz jelen, eszköz hiányzik, sikertelen egyeztetés.

A SATA interfész 7 pólusú kábel segítségével kapcsolja össze a host számítógépet az eszközzel (device). A kábel két differenciális érpárt tartalmaz, az egyiket az adás, a másikat a vétel számára, ezeken kívül 3 vezeték a jelföldeket köti össze. A feszültség alatti csatlakoztathatóság érdekében a csatlakozók jelföld-érintkezői kialakításukból adódóan hamarabb kerülnek elektromos kapcsolatba, mint a differenciális érpárok jelvezetékei. A kapcsolatot a 7.14 ábra szemlélteti, az ábrán alkalmazott rövidítések a következők: HT=host transmitter, HR=host receiver, DT=device transmitter, DR=device receiver. A csatlakozók fizikai megjelenését a 7.15 ábra szemlélteti.



7.14 ábra – Kábelcsatlakozás a host és az eszköz között

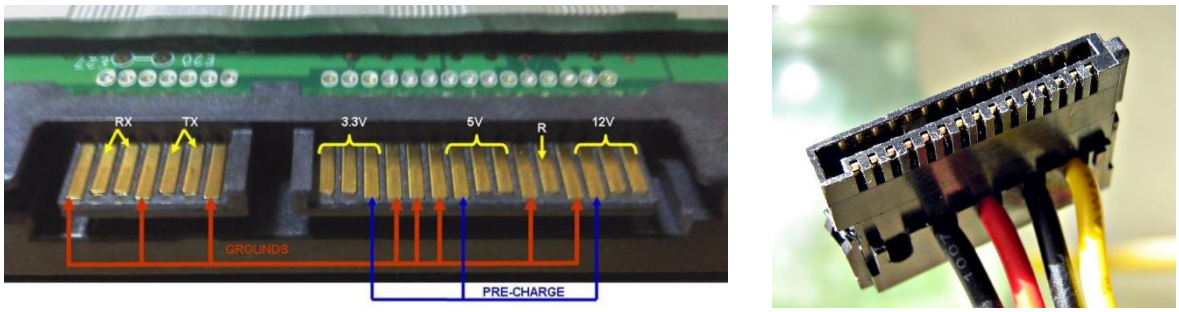


7.15 ábra – SATA adatkábel csatlakozók az alaplapon és a kábelen[7]

[7] https://hu.wikipedia.org/wiki/Serial_ATA (Letöltés: 2020.04.11.)

Copyright © 2021 / BME AUT

A SATA interfészes eszközökhöz az adatkábelen kívül egy külön tápkábel is csatlakozik. Ez egy 15 pólusú csatlakozón keresztül történik, az érintkezők kialakítása itt is a feszültség alatt történő csatlakoztathatóságot segíti (a tápfeszültség fizikailag hosszabb lábaihoz az eszközben egy ún. precharge-kapcsolás csatlakozik, amely hamarabb érzékeli a csatlakoztatást és megvédi az eszközt a különböző potenciálú pontok összezárásától). A 11-es csatlakozóláb az ún. késleltetett indítás jelzés/vezérlés, segítségével meggátolható, hogy több meghajtó esetén azok motorjai egyszerre pörögjenek fel (ilyenkor az áramfelvétel többszöröse az állandósultnak).



7.16 ábra – Tápellátás biztosítása a SATA eszközök számára[8]

A legfontosabb elektromos paramétereket a 7.3. táblázat foglalja össze.

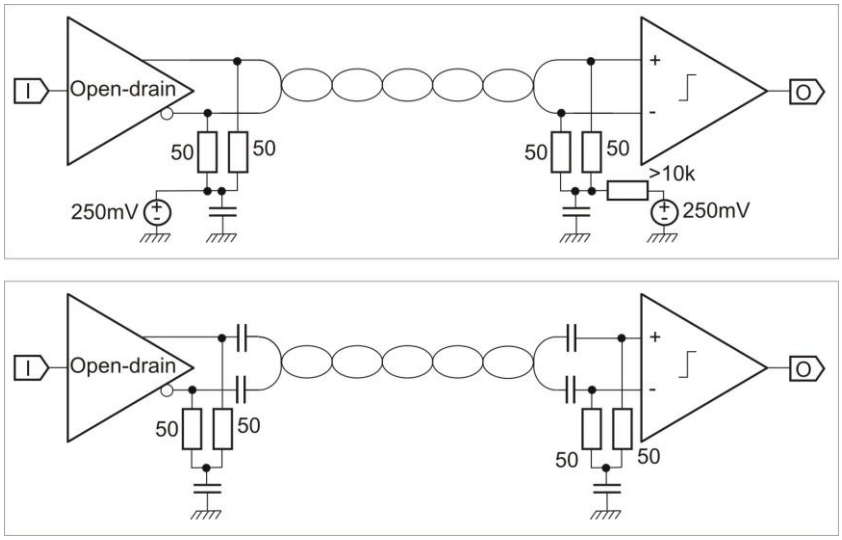
Paraméter	Egység	SATA 1			SATA 2			SATA 3		
		Min	Tip	Max	Min	Tip	Max	Min	Tip	Max
Csatorna sebesség	Gbps		1,5			3			6	
f_{baud}	GHz		1,5			3			6	
T_{UI} Unit Intervall	ps	670,23	666,67	666,43	333,12	333,33	333,22	166,56	166,67	166,61
f_{tol} Tx frekvencia pontosság	ppm f_{baud}	-350		+350	-350		+350	-350		+350
$V_{cm,DC}$ DC csatolt közös módusú fesz.	mV	200	250	450	Csak AC			Csak AC		
Z_{diff}	Ohm		100			100			100	
$Z_{diff,TX}$	Ohm	85		115	85		115	85		115
$V_{diff,TX}$	mV	400	500	600	400		700	240		900
$Z_{diff,RX}$	Ohm	85		115	85		115	85		115
U_{IOOB}	ps	646,67	666,67	686,67	646,67	666,67	686,67	646,67	666,67	686,67

7.3. táblázat – A SATA interfészek elektromos paraméterei

[8] http://e2e.ti.com/support/processors/f/791/t/552017?SATA-Cable-Connection-with-OMAP-L138-LCDK (Letöltés: 2020.04.11.)

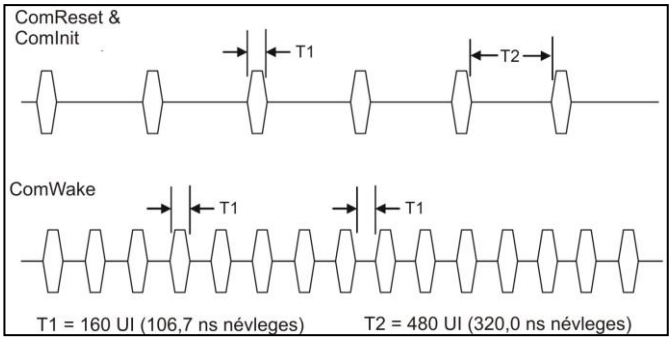
Copyright © 2021 / BME AUT

A differenciális jelátvitel névleges értéke 500 mV az adó és a vevő oldal 100 Ω-os szimmetrikus lezárása mellett. A SATA 1 szabvány a jelek DC és AC csatolását egyaránt megengedi, magasabb frekvenciákon (SATA 2 és SATA 3) már csak az AC csatolás megengedett. DC csatolás esetén a közös módusú eltolás névleges értéke 250 mV.



7.17 ábra – Adó és vevő lezárása, DC és AC csatolás

Külön meg kell emlékeznünk az ún. sávon kívüli (OOB) jelekről. A SATA interfészek 3 sávon kívüli jelet használnak, ezek a COMRESET, a COMINIT és a COMWAKE. Ezeket a jeleket minden esetben a fizikai alrendszer generálja és észleli, céljuk a rendszer alapállapotba hozása (COMRESET), inicializálása (COMINIT) és a rendszer csökkentett tápellátású módból normál üzemmódba való visszahozása (COMWAKE). A sávon kívüli jeleket mindig 160 időegységnyi (UI_{OOB}- Unit Intervall) $ALIGN_P$ primitíveket tartalmazó (ld. később) lökettel, és a löketek közötti nyugalmi állapottal (nincs differenciális jel) állítják elő.

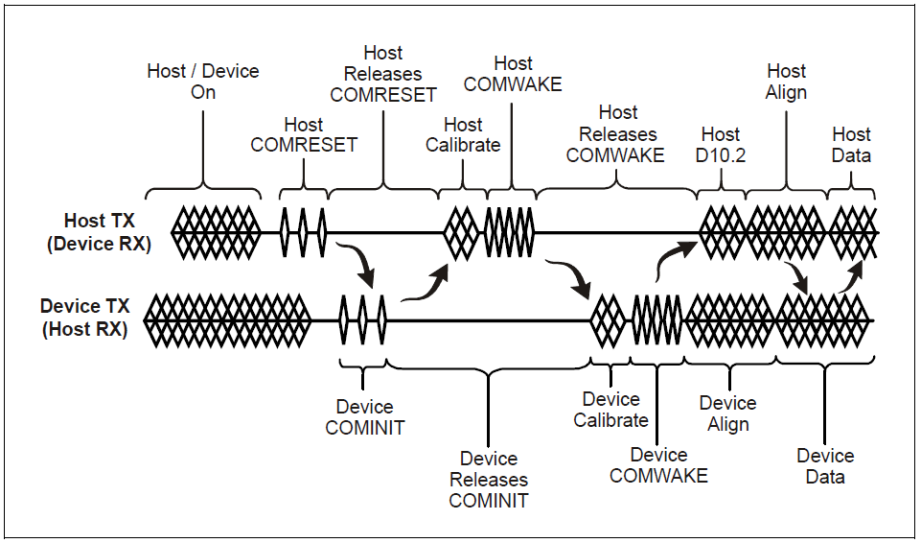


7.18 ábra – Sávon kívüli jelek (OOB)

A COMRESET jelet mindig a host generálja és ezzel resetet vált ki a csatlakoztatott eszközben. Erre az eszköz egy (teljesen hasonló felépítésű) COMINIT jellel válaszol. Hatására a host kalibrálja a TX kimenetét, majd COMWAKE jelet küld az eszköznek. Erre annak az adó és a

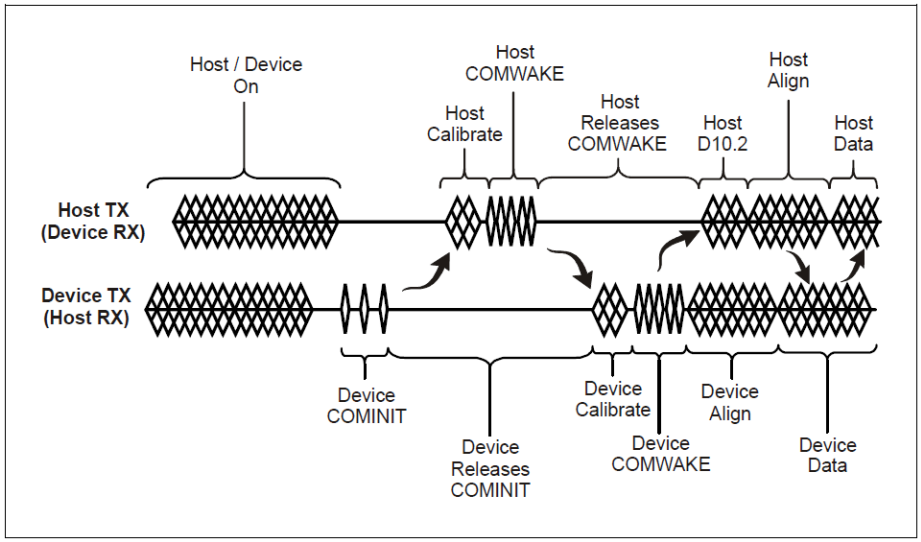
Copyright © 2021 / BME AUT

vevő frekvenciájának összehangolásához van szüksége, ezt követően ő küld COMWAKE jelet (6 db burst) a hostnak hasonló kalibráció lefolytatásához.



7.19 ábra – COMRESET szekvencia[9]

A COMINIT jelet mindig az eszköz küldi a host felé. Ez minden esetben a kommunikáció újrainicializálását váltja ki



7.20 ábra – COMINIT szekvencia

[9] Serial ATA Revision 3.0 - https://sata-io.org/

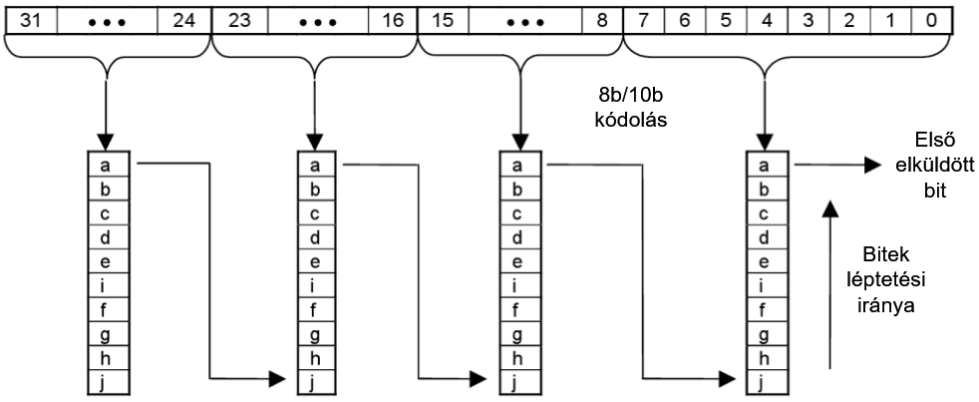
Copyright © 2021 / BME AUT

A sávon kívüli jelek az ALIGNP primitívből épülnek fel. Minden ALIGNP primitív 4 db 8b/10b kódolású karaktert tartalmaz a következő felépítésben:

RD = -1	RD = +1	Név	8b/10b név
0011111010	1100000101	Align1	K28.5
0101010101	0101010101	Align2	D10.2
0101010101	0101010101	Align3	D10.2
0010011100	1101100011	Align4	D27.3

7.4. táblázat – Az ALIGN primitív felépítése

Az első karakter a K28.5 minden más kódtól elkülönülő vezérlőkarakter, az ún. kommakarakter. Ezt kizárólag az ALIGNP primitív első karaktereként használják, ha egy vevő ezt a bitsorozatot észleli, biztos lehet benne, hogy ALIGNP primitív fog beérkezni hozzá. Az ezt követő két adatkarakter szemmel láthatóan ideálisan használható szinkronizáló karakterként (bitenként változnak benne a logikai jelszintek). Végül a lezáró karakter biztosítja azt, hogy az ALIGNP primitív „paritássemleges” legyen: pontosan ugyanannyi egyes található benne, mint amennyi 0. Ha még azt is figyelembe vesszük, hogy a 4 bájtból felépülő duplaszó soros kiküldésének előírása a 8b/10b szabvány szerint a legkisebb helyiértékű bit kiküldésével kezdődik (ld. 7.21 ábra), akkor láthatjuk, hogy a 40 bitben 10 vagy 12 db 0-1 jelszintváltás követi egymást. A sávon kívüli jelek időzítési előírása szerint minden löket legalább 4 db ALIGNP primitívet kell tartalmazzon egymás után (160 UI), a szinkronizálás céljára szolgáló COMWAKE jel pedig 6 ilyen csomagot küld ki egyenletes ütemezéssel.



7.21 ábra – Bit- és bájtsorrend a duplaszavak soros adásakor

A szabvány még számos olyan tulajdonságot definiál a fizikai réteg számára (pl. lezáró impedancia hangolása, beépített önteszt lehetősége stb.), melyek részletezésére itt terjedelmi okokból nem térünk ki.

Copyright © 2021 / BME AUT

7.3.4 Az adatkapcsolati réteg

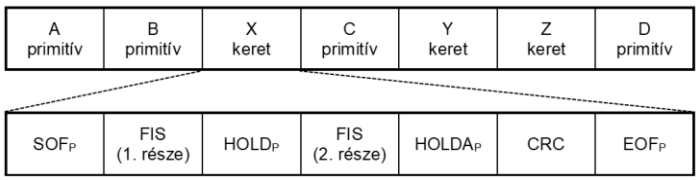
Az adatkapcsolati réteg megértéséhez a következő új fogalmakat vezetjük be:

1) Keretinformációs struktúra (FIS), amely összetartozó adat- és/vagy vezérlési információkat tartalmazó csomag több duplaszó (DWORD) formájában.
2) Primitív: egyetlen duplaszóból álló, a soros vonal vezérlésére vagy státuszára vonatkozó információ. A megkülönböztetésük érdekében valamennyi rövidítése után egy P indexet helyezünk el.
3) Átviteli keretstruktúra: minden továbbításra kerülő FIS-t ellátunk keret indító (SOF_P) primitívvel, egy a FIS tartalmára vonatkozó 32 bites CRC kóddal (Cyclic Redundancy Check, nem keverendő össze az egyszerű ellenőrző összeggel, a CheckSum-mal) és egy keret lezáró (EOF_P) primitívvel.

SOF_P (32 bit)	**FIS** (N x 32 bit)	**CRC** (32 bit)	**EOF_P** (32 bit)

7.22 ábra – Átviteli keretstruktúra legegyszerűbb formája

Az így létrejövő információfolyamban bármelyik két duplaszó között elhelyezhetők további primitívek, melyek a soros kommunikáció vezérlését szolgálják, és vételi oldalon még az átviteli keretstruktúra ill. a FIS feldolgozása előtt eltávolításra kerülnek. A primitívek akár az összetartozó FIS duplaszavak közé is beékelhetők, nem számítanak bele a CRC kód képzésébe és ellenőrzésébe, ily módon a soros adatfolyamba minimális késedelemmel elhelyezhetők szükség esetén (de csak két duplaszó között). Az adatfolyam egy összetettebb változatát mutatja a 7.23 ábra.



7.23 ábra – Átviteli keretstruktúra összetettebb formája

A 7.13 ábra szerint az adatkapcsolati réteg főbb feladatai a következők:

1) keretinformációs struktúrák (FIS) átvétele a transzport rétegből;
2) az átvett FIS átviteli keretstruktúrába ágyazása;
3) CRC számítása;
4) összekeverés (scrambling);
5) 8b/10b kódolás;
6) a keret elküldése és a nyugtázás vétele a cél adatkapcsolati rétegtől;

Copyright © 2021 / BME AUT

7) a transzportréteg értesítése a hibátlan vagy hibás adásról.

A primitívek mindegyike 4 bájtos duplaszó, első karakterük mindig egy vezérlőkarakter (ld. a 8b/10b kódolást tárgyaló fejezetben). Egyedül az $ALIGN_P$ primitív kezdődik a K28.5 kommakarakterrel, az összes többi primitívet a K28.3 vezérlőkarakter indítja. A lehetséges primitívek száma 18, a továbbiak ismertetésére terjedelmi okokból nem térünk ki.

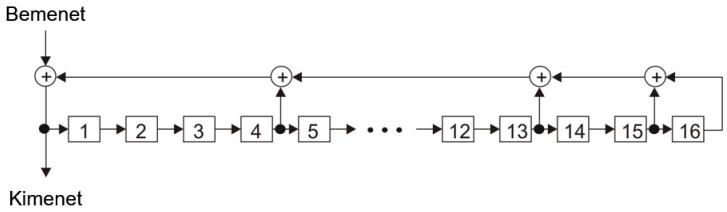
A CRC képzése a szokásos módon visszacsatolt léptetőregiszter és XOR kapuk segítségével történik. A szabvány által rögzített generátorpolinom a következő:

$$G(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1 \qquad (7.3)$$

Az összekeverés (scrambling) célja a SATA busz által kisugárzott EMI[10] csökkentése. A 8b/10b kódolás a jelvezetékek egyenáramú középértékének állandó értéken tartását és a jelváltások gyakoriságát hivatott biztosítani, azonban a többször egymás után ismétlődő forrásbájtokra kódolás után sok esetben ugyanazt a kódot állítja elő (pl. 23h adatra a D3.1 kódszó minden esetben az *abcdeifghj = 1100011001* bitsorozatnak felel meg). Amennyiben több ilyen bitsorozat követné egymást, jól érzékelhetően a *11000110011100011001110001100 1…* bitsorozatban megjelennének a bitfrekvencia felének és harmadának komponensei, méghozzá hosszabb sorozat esetén dominánsan. Ezt elkerülendő az összekeverés még a szisztematikus bitsorozatokban is véletlenszerűvé teszi a változásokat, ezáltal kisebb amplitúdójú, szélesebb spektrumú zaj keletkezik, mint az előző esetben. A SATA szabvány szerint az összekeverés generátorpolinomja a következő:

$$G(x) = x^{16}+x^{15}+x^{13}+x^{4}+1 \qquad (7.4)$$

A multiplikatív összekeverés hardver realizációját a 7.24 ábra mutatja.



7.24 ábra – Multiplikatív összekeverés visszacsatolt léptetőregiszterrel

[10] EMI: elektromágneses interferencián azt a szabad térben kibocsátott vagy sugárzott, táp- vagy jeltovábbító vezetékeken továbbított jelet értjük, amely veszélyezteti a rádiónavigációs rendszerek vagy egyéb biztonsági szolgáltatások működését, illetve jelentős mértékben rontja, zavarja, vagy időről időre megszakítja a rádió-távközlési szolgáltatást.

Copyright © 2021 / BME AUT

7.3.5 Transzportréteg

A transzportréteg a szabványos ATA adapter és az adatkapcsolati réteg között helyezkedik el, ezekkel kommunikál, cserél adatot. A szabványos ATA adapter árnyékregiszterek írásával és olvasásával kezelte az interfészt, a legfontosabb feladat ezek tartalmának az adatkapcsolati réteg számára kezelhető FIS-ekbe rendezése, ill. a kapott FIS-ek szétbontása és a regiszterek megfelelő részeibe töltése.

A SATA szabvány 9 FIS főtípust specifikál, melyek közül némelyik altípusokat is tartalmaz. Az egyes FIS típusoknak az azonosítása az első duplaszó legalacsonyabb helyiértékű bájtja (FIS-Type) alapján történik. A könnyebb érthetőség kedvéért egy ilyen FIS felépítését mutatja a 7.25 ábra.

0	Features(7:0)	Command	C R R R PM Port	FIS Type (27h)
1	Device	LBA(23:16)	LBA(15:8)	LBA(7:0)
2	Features(15:8)	LBA(47:40)	LBA(39:32)	LBA(31:24)
3	Control	ICC 7 6 5 4 3 2 1 0	Count(15:8)	Count(7:0)
4	Reserved (0)	Reserved (0)	Reserved (0)	Reserved (0)

7.25 ábra – Register FIS - Host to Device felépítése

Az egyes mezők (csak felsorolás jellegű) jelentése a következő:

- *FIS Type*: azonosítja a FIS-t, a Host to Device típus kódja 27h.
- *PM Port*: Port sokszorozó (Port Multiplier) használata esetén azonosítja az eszközt.
- *R, Reserved*: nem használt, foglalt.
- *Command*: az árnyékregiszter blokk parancsregiszterének tartalma.
- *Features*: az árnyékregiszter blokk tulajdonságregiszterének tartalma.
- *LBA*: a logikai blokkcím az árnyékregiszterek megfelelő helyeiről.
- *Device*: az árnyékregiszter blokk eszközregiszterének tartalma.
- *Count (7:0)*: az árnyékregiszter blokk szektorszám regiszterének tartalma.
- *Count (15:8)*: az árnyékregiszter blokk kiterjesztett címmező részének tartalma.
- *ICC*: (Isochronous Command Completion) - a host tudatja az eszközzel, hogy időkorlátos parancs végrehajtása van folyamatban.
- *Control*: az árnyékregiszter blokk Device Control regiszterének tartalma.

A kiragadott példa jól mutatja, hogy feladatok jelentős része a transzportrétegben az árnyékregiszterekben más elrendezésben megjelenő információk összerendelése a FIS struktúrákkal (oda-vissza), folyamatosan egyeztetve a két szomszéd réteggel.

Copyright © 2021 / BME AUT

				Register access operation	
	A2	**A1**	**A0**	**Read**	**Write**
CS 0 Active	0	0	0	Data Port	
	0	0	1	Error	Features
	0	1	0	Sector Count [15:8], [7:0]	
	0	1	1	LBA Low [31:24], [7:0]	
	1	0	0	LBA Mid [39:32], [15:8]	
	1	0	1	LBA High [47:40], [23:16]	
	1	1	0	Device / Head	
	1	1	1	Status	Command
CS 1 active	1	1	0	Alternate Status	Device Control

7.26 ábra – Egy egyszerű árnyékregiszter blokk felépítése az ATA rendszerben

A SATA szabvány által létrehozott FIS-típusokat (egy-egy mondatos értelmezéssel) a 7.5. táblázat mutatja be. Jól érzékelhető a táblázat alapján, hogy a FIS-típusok a meglévő árnyékregiszter-blokk elemeinek és az ATA szabvány üzemmódjainak a leképezését szolgálják a SATA transzportrétegében.

Típus	Név	Funkció
(27h)	Register FIS – Host to Device	Az ATA parancs- és vezérlőregiszterét képezi le.
(34h)	Register FIS – Device to Host	Az ATA parancs- és vezérlőregiszterét képezi le.
(39h)	DMA Activate FIS – Device to Host	Ezzel jelzi az eszköz, hogy készen áll DMA típusú adatfogadásra.
(41h)	DMA Setup FIS – Bi-directional	Ezzel programozható fel a First Party DMA
(46h)	Data FIS – Bi-directional	Mindkét partner küldheti. A merevlemezről olvasott, illetve az oda írt adatok átvitelére szolgál.
(58h)	BIST Activate FIS – Bi-directional	A beépített önteszt kiváltására szolgál. A küldőnek és a fogadónak egymástól eltérő tevékenységet kell végrehajtani az eredményes kooperáció érdekében.
(5Fh)	PIO Setup FIS – Device to Host	Az eszköz ezzel értesíti a hostadaptert az átviendő adatok számáról és a DRQ-blokk státuszáról PIO-adatfázis esetén.
(A1h)	Set Device Bits FIS – Device to Host	A hostadapter frissítésére szolgál, a BSY és a DRQ jelek törlése után az eszköz állapotáról és státuszáról informál.
(C7h) (D4h)	Vendor specific	Gyártóspecifikus adatok.

7.5. táblázat – FIS típusok a SATA szabványban

Copyright © 2021 / BME AUT

7.3.6 Native Command Queuing (NCQ)

A soros ATA szabvány egyik leglényegesebb új szolgáltatása a Natív Command Queuing (NCQ – természetes parancs-sorbaállítás). Az NCQ egy parancsprotokoll, amely lehetővé teszi az eszköz számára (tehát a merevlemezes tároló SATA interfésze számára), hogy több parancsot tároljon egy belső parancssorban. Ebben a sorban a parancsok sorrendje dinamikusan átrendezhető az optimális végrehajtás érdekében.

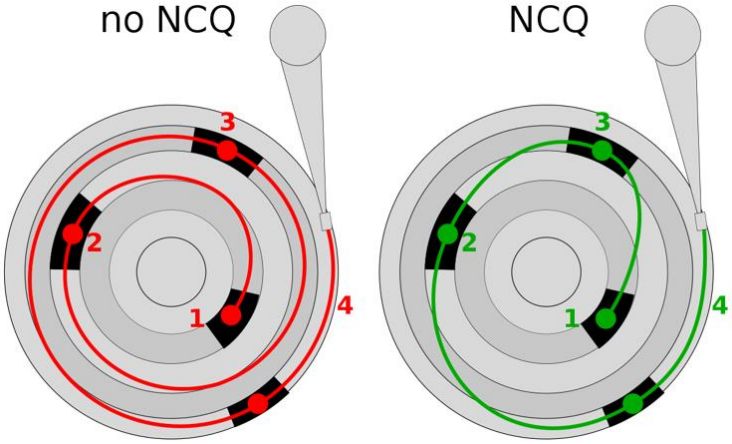
A korszerű operációs rendszerek (pl. a Windows, Linux) egyre jobban kihasználják a többszálú működés, a processzoralapú ún. Hyper-Threading technológiát és a többmagos processzorok által nyújtott párhuzamos végrehajtás lehetőségét. Ezek a működésmódok pedig egyidejűleg vagy gyors egymásutánban több parancsot is elküldhetnek a merevlemezes tárolók felé. Az NCQ alkalmazásával a diszkekhez való hozzáférés hatékonysága jelentősen növelhető.

A diszkekhez forduló alkalmazások ritkán akarnak olyan sorrendben hozzáférni az adatokhoz, amilyen sorrendben azok az olvasófej előtt sorban elhaladnak. Sokkal gyakoribb, hogy a diszkeken szétszórtan tárolt információkhoz kívánunk hozzáférni (ennek oka a véletlenszerű sorrendben elhelyezkedő kisebb és a szabad helyek vonatkozásában csak töredezetten elhelyezhető nagyobb állományokban keresendő). A lemez rendszeres töredezettség-mentesítése időt igényel, „fárasztja” a tároló médiumot és a mechanikusan fáradó alkatrészeket – de még akkor is csak a második problémán képes némileg segíteni, a tárolt fájlok későbbi hozzáférési sorrendjét előre nem tudhatjuk.

A merevlemezes tárolók mechanikai időtöbbletét (overhead) a léptetési és a forgási késleltetések okozzák, mindkettő akár több milliszekundum hosszúságú is lehet. A mindkét késleltetést minimalizáló legjobb ismert algoritmus az ún. forgásipozíció-sorrendezés (Rotational Position Ordering – RPO). Ezt használva a háttértároló a végrehajtandó parancsokat olyan sorrendben végzi el, hogy a hozzáférési időkésések minimális, a teljesítőképesség pedig maximális legyen, minimálisra csökkentve mindkét említett késleltetés-forrást.

A korai algoritmusok egyszerűen csak a sávok közötti léptetési (ún. keresési) utat minimalizálták. Egy rövid léptetés azonban hosszú forgási késleltetéssel járhat együtt, így a végeredmény nem optimális. A körülfordulási késleltetés akkor minimális, ha az eszköz olyan parancsot választ ki végrehajtásra, amelynek szektorai a forgás tekintetében a legközelebb vannak a fej pillanatnyi helyzetéhez. A keresési késleltetés akkor lesz minimális, ha a cilinderszám a lehető legközelebb esik a fej jelenlegi helyzetéhez. Értelemszerűen egy kétváltozós optimalizálási algoritmust kell tehát megoldani. Fentieken kívül az is előnye a parancs-sorbaállításnak, hogy a keresési és a körülfordulási késleltetések alatt, azzal átlapolva újabb parancsok átvételét is elvégezheti.

Copyright © 2021 / BME AUT



7.27 ábra – A Native Command Queuing működése[11]

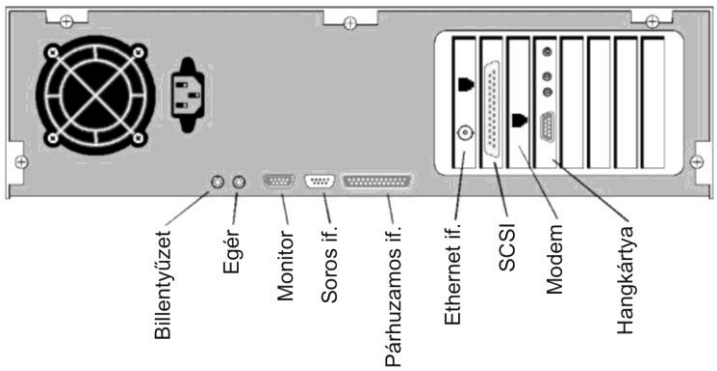
A 7.27 ábra az NCQ működését szemlélteti egy egyszerű példán. Az ábra baloldalán a tárolóegység a parancsok beérkezésének sorrendjében (1-2-3-4) hajtja végre az adatblokkok (szektorok, clusterek) hozzáférését. A 4 egység olvasása (vagy írása) a lemezek két körülfordulása alatt teljesíthető. Viszont ha az NCQ működni tud, és a meghajtó megkapja mind a 4 parancsot még az előzőek végrehajtása előtt, azok átrendezésével (1-3-2-4) a feladatot a lemezek egyetlen körülfordulása alatt is meg tudja oldani – ami ebben az esetben 100% sebességnövekedést jelent. A valóságban a hatékonyság nem ennyire nagy, hosszabb idejű reális feltételek mellett a tesztek kb. 10-20% sebességnövekedést mértek.

[11] https://en.wikipedia.org/wiki/Native_Command_Queuing (Letöltés: 2020.04.12.)

Copyright © 2021 / BME AUT

8 Universal Serial Bus

Az USB (Univerzális Soros Busz) interfészt a személyi számítógépek körül kialakult sokféle, nem szabványosított, különféle eszközmeghajtókat (drivereket) igénylő soros és párhuzamos interfészek kiváltására fejlesztették ki. A 8.1 ábra mutatja, hogy az egységesítés előtt meglehetősen kaotikus helyzet volt a számítógéphez kapcsolódó különféle eszközök csatlakozóinak (és ennek megfelelően az azokhoz kapcsolódó szabványok, specifikációk) vonatkozásában.



8.1 ábra – Tipikus személyi számítógép hátoldal még az USB interfész előtt

Periféria	Interfész	Csatlakozótípus
Billentyűzet	PS-2	PS-2 Connector
Egér	PS-2	PS-2 Connector
Monitor	VGA	D-SUB DB15E
Soros interfész	RS-232C	D-SUB DB9
Párhuzamos interfész	Centronics	D-SUB DB25
Hálózat (internet)	Ethernet	BNC, RJ-45
Külső háttértár	SCSI	D-SUB DB50
Modem (telefax, internet)	Telefonvonal	RJ-11
Hangkártya	Analóg (in-out), joystick	4-5 x 3,5 mm jack, D-SUB DB15F

8.1. táblázat – Perifériák és csatlakozóik

Összefoglalva az akkori helyzetet a következő hátrányos tulajdonságok voltak megállapíthatók:

- az üzem alatti csatlakoztathatóság (hot plug) hiánya;
- az automatikus konfigurálhatóság hiánya;
- az erőforrások korlátozott volta (I/O címek, megszakítások);
- sokféle csatlakozó- és kábeltípus, kusza kábelrengeteg;
- a megfelelő szabványok hiánya;

Copyright © 2021 / BME AUT

- sokféle eszközmeghajtó szükségessége;
- magas költség új periféria hozzáadásakor.

Ezeket a hiányosságokat felismerve hét vállalat (Compaq, Digital Equipment Corporation, IBM, Intel, Microsoft és Northern Telecom) összefogott, hogy kidolgozzon egy specifikációt, elsősorban a szokásos interfészek (PS-2, nyomtató, RS-232) egyetlen interfészbe foglalására. Ennek az eredménye lett az 1996. január 15-én megjelent USB interfész 1.0-s változatú specifikációja. Mint minden új technológia bevezetésekor itt is léptek fel olyan hibák, hiányosságok, melyeket elsősorban a különböző gyártók inkompatibilitásai okoztak. Ennek megoldására létrehozták az USB-IF (USB Implementers Forum) szervezet. Ennek köszönhetően jelent meg 1998. szeptember 23-án az USB 1.1-es specifikáció, amely lényegében a korábbi inkompatibilitásokhoz vezető időzítéseket tisztázta, de lényeges funkcionális továbbfejlesztést nem tartalmazott.

Az új interfész a klasszikus perifériákon (egér, nyomtató, billentyűzet) kívül egyéb eszközök esetében is (digitális kamera, mobileszközök, PDA, pendrive, stb.) közkedvelt illesztési felületnek bizonyult, és ezzel a személyi számítógépek világában a legsikeresebb interfésszé vált. Néhány évvel a kifejlesztését követően, csak a 2006-os évben több, mint 2 milliárd USB interfésszel ellátott eszközt adtak el. Ez a népszerűség természetesen folyamatos kihívások elé is állította fejlesztőket, az interfésznek egyre újabb változatai látnak napvilágot. Az utolsó (USB4) változat specifikációját 2019. augusztus 19-én adták ki, szakértők becslése szerint az implementálása 12-20 hónapot igényel, így az új változatot használó első eszközök 2021-től várhatók.

USB specifikáció	**USB 1.0**	**USB 1.1**	**USB 2.0**	**USB 3.0**	**USB 3.1**	**USB 3.2**	**USB4**
Módosított elnevezés				USB 3.2 Gen 1	USB 3.2 Gen 2	USB 3.2 Gen 2x2	Gen 3x2
Megjelenés éve	1996	1998	2000	2011	2014	2017	2019
Átviteli sebesség	1,5 Mbit/s	12 Mbit/s	480 Mbit/s	5 Gbit/s	10 Gbit/s	20 Gbit/s	40 Gbit/s
Elnevezés	Low Speed	Full Speed	High Speed	SuperSpeed	SuperSpeed+	SuperSpeed+	SuperSpeed+
Tx/Rx sebesség	1,5 Mbit/s	12 Mbit/s	480 Mbit/s	5 Gbit/s	10 Gbit/s	10 Gbit/s	20 Gbit/s
Sáv (lane)	1	1	1	1	1	2	2
Csatlakozó (standard)	Type-A, Type-B			Type-A, Type-B SuperSpeed			
Csatlakozó (új standard)					Type-C		
Csatlakozó (mini)			Mini-A Mini-B Mini-AB				
Csatlakozó (mikro)			Micro-A Micro-B Micro-AB	Micro-B SuperSpeed			

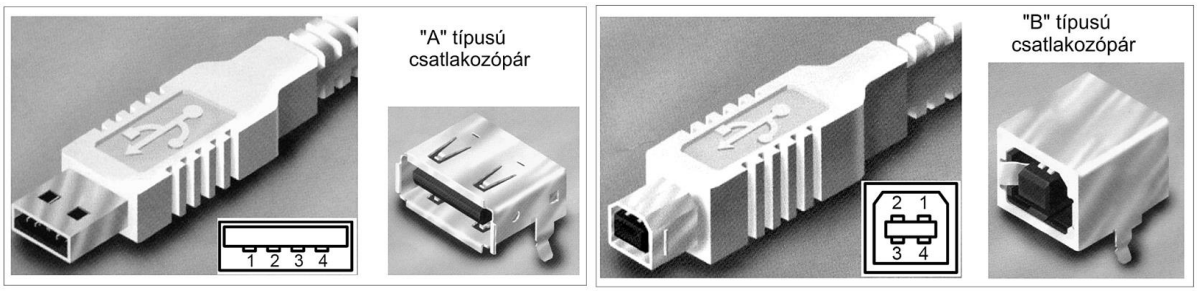
8.2. táblázat – Az USB interfész fejlődése

A fejlődés folyamatát a 8.2. táblázat mutatja. Ennek megértéséhez néhány kiegészítés:

Copyright © 2021 / BME AUT

- A már említett USB 1.0 és USB 1.1 változatok után a „nagy ugrást” az USB 2.0 specifikáció hozta. A 480 Mbit/s átviteli sebesség tekintélyt parancsolt az interfésznek a különböző célfeladatokra készült többi adatátviteli interfész között.
- A következő nagy lépés 2011-ben az USB 3.0 volt. Az USB-IF-nek lépnie kellett – mind a külső egységek növekvő adatsebesség-igényei, mind az Intel és az Apple által ekkor már aktívan fejlesztett Thunderbolt interfész (ezzel a következő fejezetekben foglalkozunk) megkövetelte a továbblépést. Az ugrás ismét számottevő volt: az adatátviteli sebesség a korábbi tízszeresére, 5 Gbit/s-ra növekedett.
- A következő évek egy folyamatos versenyfutásban teltek a Thunderbolt interfész fejlesztőivel. Az USB 3.1 változat megduplázta a sebességet (10 Gbit/s), majd egy új csatlakozó bevezetésével az eddigi egy sávos (egy Tx-Rx pár) átvitelt két sávosra növelték, így az USB 3.2 verzió már 20 Gbit/s sebességgel büszkélkedhetett.
- 2018-ban az Intel felszabadította a Thunderbolt specifikációt, szabadon felhasználhatta bárki licencdíj fizetése nélkül. Ez komoly előrelépést jelentett a hasonló célú és tulajdonságú interfészek egységesítése tekintetében, a 2019-ben bejelentett USB4 specifikáció már több tekintetben is integrálja a Thunderbolt, a PCI Express és a DisplayPort interfészek tulajdonságait. Az adatátviteli sebességet ismét megduplázták, így az már 40 Gbit/s-ra emelkedett.

Külön foglalkoznunk kell az USB interfész csatlakozóinak fejlődésével, ami általában azonnal elárulja a host vagy az eszköz teljesítőképességét. A szabvány először kétféle csatlakozót definiált, egy A és egy B típusút. Az A típus mindig a host felé irányult (ún. upstream csatlakozó), míg a B típus mindig a hosttól el (ún. downstream csatlakozó) mutatott. A két csatlakozótípus nem cserélhető fel (ld. a 8.2 ábrán), így kizárt, hogy pl. illegális visszahurkolást hozzunk létre.



8.2 ábra – USB Type-A és Type B csatlakozók

A számítógépek és a hubok (ld. később) eszközcsatlakozói tehát mind A típusúak, az eszközökön a számítógép felé mutató csatlakozók pedig mind B típusúak (egyes eszközök maguk is tartalmazhatnak további eszközcsatlakozási lehetőséget, akkor ezek megint A típusú csatlakozók lesznek). Az USB 2.0 szabvány ezeken kívül egy új csatlakozótípust is definiált, az USB mini csatlakozókat. Erre az igény a mobiltelefonok, digitális fényképezőgépek és kamerák, stb. oldaláról jelentkezett, ugyanis az eredeti Type-B csatlakozó túlságosan nagy volt ahhoz, hogy ezekbe az eszközökbe lehessen integrálni. További igényként jelent meg, hogy a mobileszközök (pl. egy okostelefon) korlátozott hostfunkciót is elláthassanak. Az USB 2.0 On-The-Go kiegészítése ennek a lehetőségét is specifikálta a mini A csatlakozó lehetőségével. Az

Copyright © 2021 / BME AUT

eszközök azonban továbbra is egyre kisebbek és laposabbak lettek, így szükségessé vált az USB micro (A és B) típusok bevezetése is (ld. 8.3 ábra).



8.3 ábra – USB standard, mini, és micro csatlakozók[1]

Az USB 3.0 szabvány gyökeres változást hozott a korábbiakhoz képest a nagysebességű sáv (lane, differenciális Tx-Rx jelvezeték-pár) bevezetésével. Olyan új csatlakozóra volt szükség, amely kompatibilis marad a korábbi USB 1.x, 2.0 kábelekkel, de a több jelvezetéket tartalmazó USB 3.0 eszköz is csatlakoztatható hozzá. Ezek lettek a Type-A és Type-B SuperSpeed csatlakozók (8.4 ábra). A csatlakozókon további 5 pólus jelent meg az új jelvezetékek csatlakozására, és jellegzetes kék színt kapott a belső részük az USB 3.x kompatibilitás jelzésére.



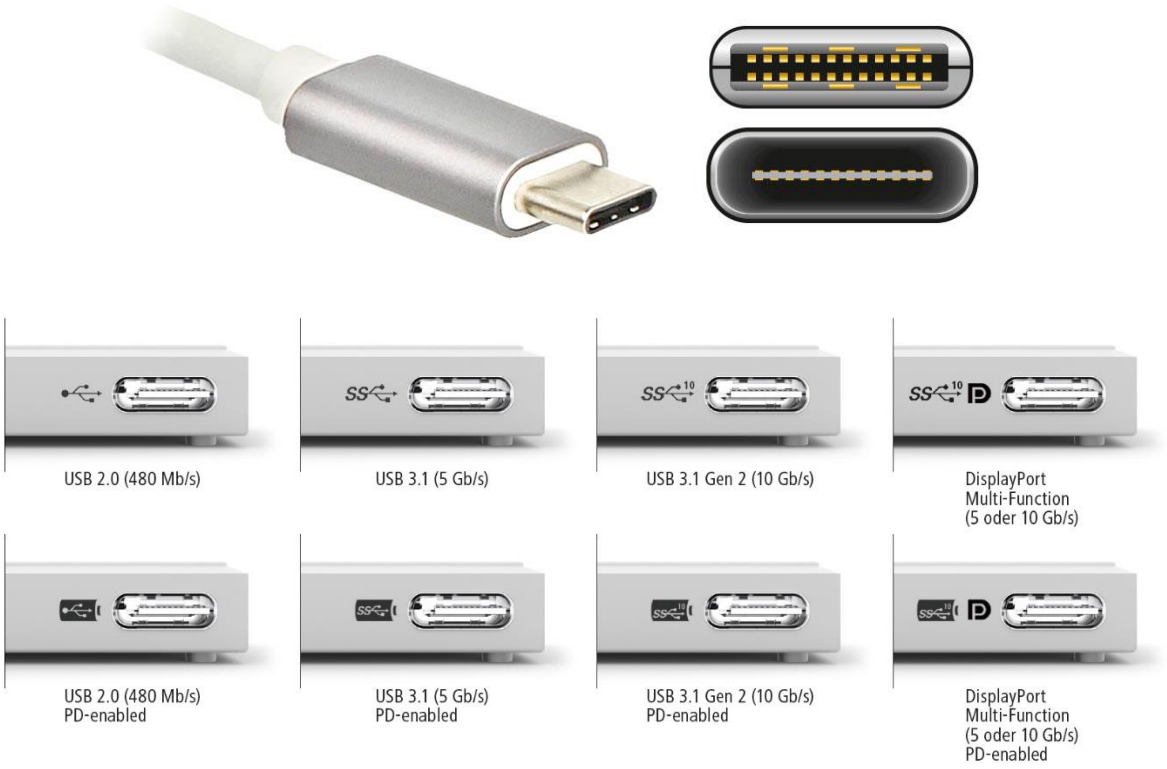
8.4 ábra – USB Type-A, Type-B és micro B SuperSpeed csatlakozók[2]

Az USB 3.2 szabványtól kezdve jelenik meg a dual-lane igénye, ismét csatlakozótípust kellett váltani. Az új USB Type-C csatlakozó már szimmetrikus felépítésű, azaz tengelye körül 180°-kal elforgatva „fejjel lefelé” is csatlakoztatható. A kétszerezett differenciális Tx-Rx jelvezeték-pár igénye csak ettől a szabványváltozattól kezdve jelenik meg, így ez a verzió, valamint az USB4 specifikáció már csak a 24 pólusú USB Type-C csatlakozó használatával lehetséges (8.5 ábra). Az ábra mutatja a számítógépeken alkalmazott jelölésrendszert is az egyes változatok

[1] https://www.reddit.com/r/coolguides/comments/5zrgdq/usb_types/ (Letöltés: 2020.04.17.)
[2] https://en.wikipedia.org/wiki/USB_3.0 (Letöltés: 2020.04.17.)

Copyright © 2021 / BME AUT

felismerhetőségére (a DisplayPort multifunkció megjelenésére a kommunikációs interfészek összehasonlító fejezetében térünk ki).



8.5 ábra – USB Type-C csatlakozó és jelölésrendszere[3]

A továbbiakban az egyes szabványváltozatok jelölésére a 8.3. táblázatban látható nemzetközileg elfogadott jelölésrendszert (elsősorban a rövidített elnevezést) használjuk. Az USB4 szabvány még annyira új, hogy alkalmazott jelölésrendszere sem alakult még ki.

USB specifikáció	**USB 1.0**	**USB 1.1**	**USB 2.0**	**USB 3.0**	**USB 3.1**	**USB 3.2**	**USB4**
Elnevezés magyarul	Alacsony sebesség	Teljes sebesség	Nagy-sebesség	Szuper-sebesség	Szuper-sebesség plusz	Szuper-sebesség plusz	USB4
Elnevezés angolul	Low Speed	Full Speed	High Speed	SuperSpeed	SuperSpeed+	SuperSpeed+	USB4
Rövidített elnevezés	LS	FS	HS	SS vagy SS^{5}	SS+ vagy SS^{10}	SS+ vagy SS^{20}	SS++ vagy SS^{40} (?)
Átviteli sebesség	1,5 Mbit/s	12 Mbit/s	480 Mbit/s	5 Gbit/s	10 Gbit/s	20 Gbit/s	40 Gbit/s

8.3. táblázat – Az USB interfész verzióinak elnevezései és jelölésrendszere az eszközökön

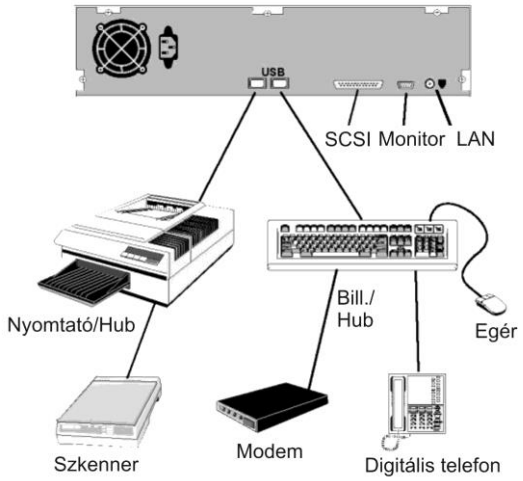
[3] https://www.delock.com/infothek/USB-3.1-USB-Type-C/usb-3.1-usb-type-c_e.html (Letöltés: 2020.04.17.)

Copyright © 2021 / BME AUT

8.1 Az USB rendszer architektúrája

Az USB interfész legfontosabb jellemzői a következők:

- **Alacsony költség.**
  Egy új periféria illesztése nem igényel bővítőkártyát és az interfész megvalósítása a perifériában a széleskörű elterjedtség miatt olcsó alkatrészekkel (kész USB kontrollerek) megvalósíthatók.
- **Feszültség alatti csatlakoztathatóság (hot plug).**
  A számítógép kikapcsolása nélkül csatlakoztathatók a perifériák, ill. bontható a kapcsolat.
- **Egyetlen interfész (és csatlakozó) típus.**
  Az összes periféria azonos csatlakozótípust használhat, a számítógép csatlakozói elosztókkal (hub) bővíthetők (ld. 8.6 ábra).



8.6 ábra – Perifériák csatlakoztatása USB interfésszel

- **127 db eszköz csatlakoztathatósága.**
  A rendszer teljes kiépítettségében 127 eszköz csatlakoztatását teszi lehetővé.
- **Tápellátás a kábelről.**
  A perifériák közvetlenül a kábelről is táplálhatók 5V-os feszültséggel a host/hub típusától függő áramfelvétel-határig (tip. 500 mA).
- **A rendszer erőforrásaira nincs szükség.**
  Az USB interfész nem igényel I/O címeket, megszakítási vonalakat, mint a PC buszokra (ISA, EISA, PCI) csatlakoztatott bővítőkártyák.
- **Hibafelismerés és javítás.**
  Az USB tranzakciók hibafelismerő mechanizmust tartalmaznak, és hiba esetén megismétlik a tranzakciót.

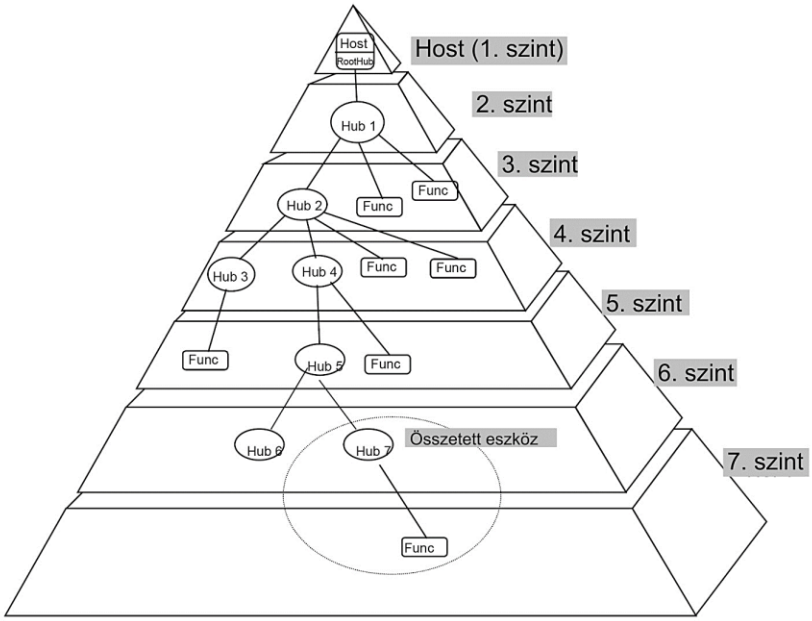
Copyright © 2021 / BME AUT

- **Energiatakarékosság.**
  Az USB eszközök 3 ms-os inaktív busz esetén automatikusan felfüggesztett (standby) üzemállapotba kerülnek, amelyben áramfelvételük nem lehet 500 µA-nél nagyobb.
- **Négyféle átviteli mód.**
  A különböző eszközök eltérő igényeinek megfelelően az USB négyféle átviteli módot támogat: nagy tömegű (bulk), kiszámítható időzítésű izoszinkron (=egyidejű, isochronous), megszakításos (interrupt) és vezérlő (control) adatátvitel.

Az USB interfész lehetővé teszi a host számítógép és a perifériák közötti mindkét irányú információcserét. A csatlakoztatott perifériák a host által ütemezett, token- (vezérjel) alapú protokoll szerint osztoznak az interfész sávszélességén. Azaz a rendszer egyetlen mestere a host, minden tranzakciót ő kezdeményez.

Busztopológia

Az USB interfész az eszközöket a hosttal köti össze. A fizikai kapcsolat többszintű csillagtopológia, minden csillag középpontjában egy hub (elosztó) található, és minden szegmens egy pont-pont kapcsolat a hub és egy eszköz, vagy egy hub és egy másik hub között (ld. 8.7 ábrát). A specifikáció öt egymást követő szinten engedi meg a hubok alkalmazását.



8.7 ábra – Az USB topológia szemléltetése

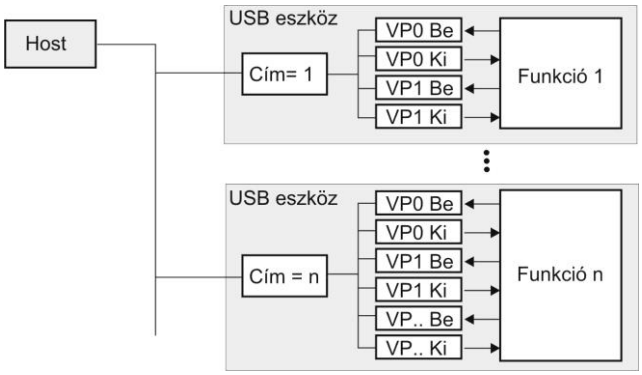
Minden USB rendszer egyetlen hostot tartalmazhat. A host számítógép és a fizikai USB kapcsolat közötti interfészegységet nevezzük hostvezérlőnek (HC = Host Controller). A hostrendszer részét képező gyökérhub egy vagy több csatlakozási ponttal rendelkezhet. A hubok öt szinten iktathatók be a rendszerbe, a maximálisan csatlakoztatható eszközök száma 127. Kezdetben a személyi számítógépek csak 1-2 USB portot tartalmaztak, és külső hubokkal

Copyright © 2021 / BME AUT

illetve összetett eszközökkel (klaviatúrába vagy nyomtatóba épített bővítő hub) lehetett a csatlakoztatandó eszközök számát növelni. A külső hubok növelték a költségeket, ezért ma már a PC-k sok, akár 6-8 USB portot is tartalmazhatnak. A modern asztali számítógépek az USB portok felét a homlokfelületükön tartalmazzák az átmenetileg csatlakoztatott eszközök (pl. pendrive, digitális fényképezőgép) számára, a laptopok hasonló okokból minden oldalukon tartalmaznak USB portokat.

Funkciók és végpontok

Funkcióknak nevezi a szabvány azt az USB eszközt, amely képes adatot továbbítani (adni és/vagy venni) az USB buszon. A host és funkciók kapcsolatát a 8.8 ábra szemlélteti.



8.8 ábra – Funkciók és végpontok

A funkcióknak egyedi címük van, továbbá a funkciók belül 16 bemeneti és 16 kimeneti, egyenként címezhető ún. végponttal (VP) rendelkezhetnek. Ezek a végpontok az adatforrások és az adatnyelők (a be- és a kimenet iránya mindig a host felől értendő). Ha például a szoftvermeghajtó egy adatcsomagot küld az 1-es funkció 1-es végpontjának, akkor a csomag az eszköz „VP1 Ki” pufferébe kerül, ahonnan az eszközt vezérlő processzor ki tudja azt olvasni (tipikusan értesítést is kap arról, hogy adat van a pufferben). Ha a funkció erre válaszolni akar, nem küldhet közvetlenül a buszra adatokat, mivel a buszt a host vezérli. Ezért neki az adatokat a „VP1 Be” pufferbe kell írnia, és azok mindaddig ott lesznek találhatók, ameddig a host egy IN csomagot nem küld, amely felszólítja a végpontot az adatok küldésére.

Minden eszköznek támogatnia kell a 0-s végpontot (VP0). Ez a végpont kapja meg a vezérlési és a státuszkérelmeket. A többi végpont támogatása opcionális.

Buszprotokoll

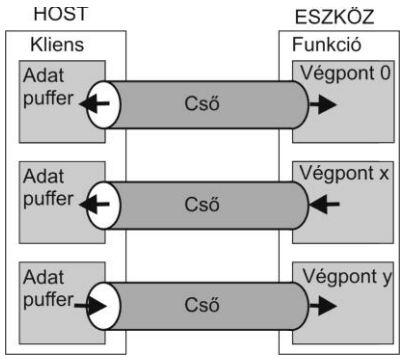
Az USB lekérdezéses (polled) busz. A tranzakciókat kizárólag a hostvezérlő kezdeményezheti. Minden busztranzakció 1, 2 vagy 3 csomagból áll. Minden tranzakció azzal kezdődik, hogy a hostvezérlő az aktuális ütemezésnek megfelelően egy olyan csomagot küld el, amely tartalmazza a tranzakció típusát és irányát, kijelöli a tranzakcióban résztvevő valamelyik eszközt és annak valamelyik végpontját (endpoint). Ez a csomag az ún. tokencsomag. Az USB eszközök dekódolják a tokencsomagban lévő címet, és csak az az USB eszköz válaszolhat,

Copyright © 2021 / BME AUT

amelyiknek a címét a csomag tartalmazta. Az adatátvitel irányát is a tokencsomag tartalmazza, az adatforrás ezután elküldi az adatokat, vagy jelzi, hogy nincsenek elküldendő adatok. Az adatok vevője kézfogásos csomaggal válaszolva jelzi az adatátvitel hibás vagy hibátlan voltát.

Csövek (pipes)

Az USB adatátviteli modelljét a host és az eszköz végpontja között csőnek (pipe) nevezik. Egy eszköz csatlakoztatásakor az USB meghajtó beolvassa az eszköz konfigurációs adatait, amelyek meghatározzák az adott funkcióhoz tartozó végpontok jellemzőit. Megállapítja, hogy rendelkezésre áll-e elegendő sávszélesség az összes végpont igényeinek megfelelően. Ha igen, akkor az USB meghajtó létrehoz egy kommunikációs csövet az igényelt sávszélességet tartalékolva az adott végpont számára, majd értesíti az eszközmeghajtót a cső létrejöttéről. A 8.9 ábra illusztrálja a csövek működésének elvét. A csövek a kliens memóriapufferei és az eszköz végpontjai között jönnek létre, lehetnek egy- és kétirányúak az adatforgalom szempontjából.



8.9 ábra – Kommunikációs csövek

A csövek mindaddig passzív állapotban maradnak, ameddig a kliensszoftver nem igényel átvitelt valamelyik csövön keresztül. Fontos megjegyezni, hogy az USB kliens eszközmeghajtója csak a kommunikációs csövekről és a végpontokról tud, és nem látja az USB alacsonyabb szintű adatátviteli mechanizmusát.

Kétféle cső létezik: folyam- (stream) és üzenet- (message) típusú. A folyamtípusúban nincs definiálva az adatstruktúra, míg az üzenettípusúban igen. A folyamcső egyirányú adatáramlást közvetít, az üzenetcső tipikusan kétirányút. Minden csőhöz hozzá van rendelve a sávszélessége és a kapcsolódó végpont tulajdonságai (irány, pufferméret stb.). A tranzakció ütemezése a folyamcsövek számára lehetővé teszi a folyam handshake-vezérlését. Ez hardver szinten akadályozza meg a pufferek alul- vagy túlcsordulását, a NAK (Not Acknowledged – nincs elfogadva) handshake-protokollt használva. A NAK vétele esetén a tranzakció a rendelkezésre álló buszidőben megismétlésre kerül.

Keretek (frames)

Az USB-n az adatátvitel keretekbe szervezve megy végbe. A gyökérhub minden ezredmásodpercben elküld egy SOF (Start of Frame – keret kezdete) csomagot. A két egymást

Copyright © 2021 / BME AUT

követő SOF-csomag közötti időtartamot keretidőnek nevezzük, ez LS és FS esetén 1 ms. A HS hubok ún. mikrokeretekbe (microframes) szervezik a működést, ezek időzítése 125 µs és a µSOF (Micro Start of Frame) csomagokkal indulnak. A SS változatok szintén megtartották a 125 µs-os időzítést, de a µSOF csomagok helyét ezeknél már az ITP (Isochronous Timestamp Packet) csomagok veszik át.

A keretekben megvalósítandó tranzakciókat adatfolyamtípus (ld. később) szerint ütemezi a rendszer. Így a keretekben rendelkezésre álló idő 90%-át az izoszinkron adatátvitel és a megszakításos átvitel foglalhatja el, a fennmaradó 10% a vezérlési és a tömegadat átvitelére használható.

Robusztusság

Az USB interfész robusztusságát több jellemző is támogatja:

- fizikai jelek szintjén a differenciális jelátvitel, valamint az árnyékolt kábelbusz;
- az adat- és a vezérlési mezők szintjén a CRC nyújt védelmet;
- biztosított az eszközök buszra csatlakozásának és lecsatlakozásának felismerése;
- a protokoll időkorláttal (timeout) igyekszik helyreállítani magát (self-recovery) az elveszett vagy korrumpálódott csomagokból;
- a folyamatvezérlés lehetővé teszi az azonosidejűséget (izoszinkronitást) és a hardverpufferek menedzselését.

Adatfolyam-típusok

Az USB interfész a host és az eszköz között funkcionális adat- és vezérlésátvitelt biztosít egy- és kétirányú csövekkel. Az adatátvitel a host szoftvere és az eszköz egyik végpontja között megy végbe (ezt az összerendelést neveztük csőnek). Egy adott csövön az adatátvitel független a többi csövön végbemenő adatátviteltől. Egy USB eszköz több csövet is tartalmazhat, ezek alapvetően az eszköz funkcionalitása szerint jönnek létre (pl. egy webkamera általában rendelkezik egy a képinformáció átvitelére szolgáló csővel és egy külön cső végzi a kamerába integrált mikrofon hangjelének átvitelét). A csövekben végbemenő adatátvitel négyféle lehet:

- **Vezérlésátvitel (Control Transfer):**
  Elsősorban az eszköz csatlakoztatásakor a konfigurálás céljaira használatos. Emellett egyéb eszközspecifikus célokat is elláthat, mint pl. az eszköz csöveinek a vezérlését.
- **Tömegadat-átvitel (Bulk Data Transfer):**
  Nagymennyiségű adat írását vagy olvasását jelenti nem kritikus időkorlát mellett (pl. nyomtató vagy szkenner illesztése esetén). Az adatátvitel szekvenciális, a megbízható adatátvitelt hardver szinten a hibafelismerés biztosítja, hiba esetén korlátozott számú ismétlés lehetséges. A tömegadat-átvitel számára rendelkezésre álló sávszélesség a busz egyéb aktivitásaitól függ.
- **Megszakításos átvitel (Interrupt Transfer):**
  Véletlenszerűen fellépő, minimális késleltetéssel átviendő kis mennyiségű adatok esetén használják (pl. egy pozícionáló eszköz által beviendő koordinátaértékeknél).

Copyright © 2021 / BME AUT

- **Izoszinkron adatátvitel (Isochronous Data Transfer):**
  Az USB sávszélességének előre egyeztetett részét (általában 90%-át) használja adott késleltetéssel. Az isochronous adat folytonos és valós idejű a keletkezésében, a továbbításában és a felhasználásában is (tipikus példák az audio- és videoadatok). Akár az adatátviteli sebesség kisebb a szükségesnél, akár késleltetve jut el az adat a célba, minőségromlás lép fel. Az izoszinkron átvitelnél a potenciális tranziens hibák javítása nem történik meg, azaz az elektromos adatátvitelben fellépő hibákat a hardver nem korrigálja a csomagok ismétlésével. A gyakorlatban az USB bithiba valószínűsége olyan kicsi, hogy ez általában nem okoz semmilyen nehézséget.

Spec.	Bitsebesség [Mbit/s]	Sáv (lane)	Frame/ µFrame [µs]	Keretidő 90%-a [µs]	Burst szám	Csomagok száma	Csomag mérete [Bájt]	Átvitt bájtok száma	Átvitt bitek száma	Idő-szükséglet [µs]
USB 1.0	1,5	1	1000	900	1	2	256	512	4 096	819
USB 1.1	12	1	1000	900	1	5	256	1 280	10 240	853
USB 2.0	480	1	125	112,5	1	6	1 024	6 144	49 152	102
USB 3.0	5 000	1	125	112,5	3	16	1 024	49 152	491 520	98
USB 3.1	10 000	1	125	112,5	8	16	1 024	131 072	1 081 344	108
USB 3.2	10 000	2	125	112,5	16	16	1 024	262 144	2 162 688	108

8.4. táblázat – Egyetlen keretben/mikrokeretben átvihető adatok száma izoszinkron adatátvitellel

A 8.4. táblázat mutatja az izoszinkron adatátvitellel egyetlen keret/mikrokeret idő alatt átvihető adatok mennyiségét. A bitek és a bájtok aránya az alkalmazott kódolástól is függ (az USB 3.0-ig nem használtak kódolást, utána pedig a 8b/10b, majd a 128b/132b kódolást alkalmazzák – ld. a fizikai réteg leírásánál). A valóságban az utolsó oszlop időszükségletei kicsit magasabbak a vezérlőkarakterek, CRC és a csomagok közötti szünetidők miatt. A burst szám (csak USB 3.0-tól, maximális csomagméret mellett), a csomagok száma és mérete paraméterezhető az átvitel számára – a lényeg az, hogy az így kiadódó időszükséglet ne haladja meg a rendelkezésre álló időt (a keretidő max. 90%-át). Az új USB4 szabványban ezek a fogalmak csak a korábbi verziókkal való kompatibilitás révén definiáltak (azonos értékekkel).

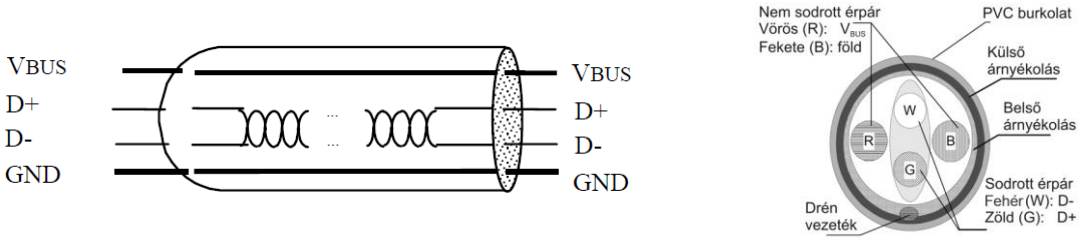
8.2 Elektromos jellemzők

Az USB 1.x és 2.0 szabványok esetében a fizikai interfész felépítése még azonos volt. Az USB 3.x szabvánnyal az interfész kibővült a SuperSpeed sebességű előbb egy, majd két sávval (angol elnevezéssel lane: Rx-Tx differenciális jelvezeték-pár), végül az USB4 szabvány már csak erre a két sávos változatra épül. A szabványok elektromos paraméterei eltérőek az egyes változatok esetében.

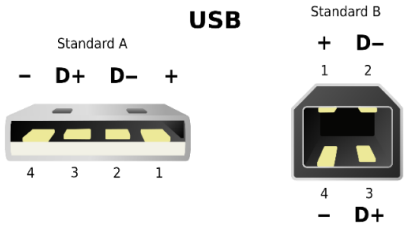
Copyright © 2021 / BME AUT

8.2.1 Az USB 1.1-es hub és eszközök kapcsolata

Az USB 1.0 és 1.1-es szabványok még két eszközsebességet ismertek: LS és FS. A vezérlő (host vagy hub) és az eszköz között 4 eres kábel teremt kapcsolatot, melyen mind az eszköz tápellátását (V_{BUS}, GND), mind a félduplex differenciális jelkapcsolatot (D+ és D-) megoldják. A kábel sematikus vázlatát a 8.10 ábra, a csatlakozók lábkiosztását a 8.11 ábra mutatja.

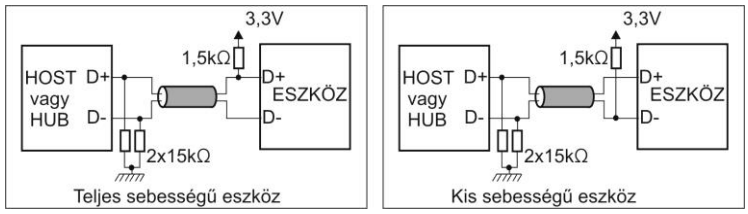


8.10 ábra – Az USB 1.x és 2.0 egységek jelvezetékei



8.11 ábra – Az USB 1.x és 2.0 csatlakozók lábkiosztása[4]

Az USB 1.1-es változatban a host és a hub (a továbbiakban már csak a hostot nevesítjük kettejük közül, de helyette minden esetben a hub is érthető) kapcsolata a 8.12 ábrán látható, működésük ez alapján érthető meg egyszerűen.



8.12 ábra – Az USB 1.x hub (host) és az eszköz kapcsolata

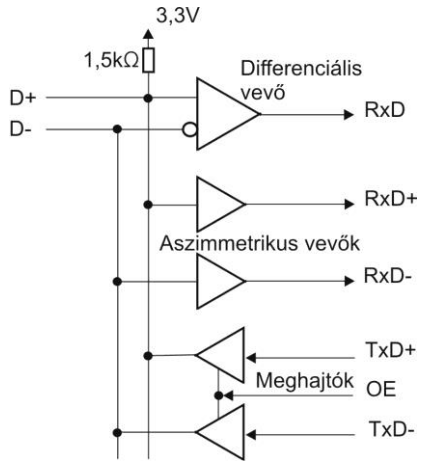
Amennyiben a host porthoz nem csatlakozik eszköz, akkor mindkét differenciális vonal (D+ és D-) földpotenciálon van az őket a hoston belül földpotenciálra húzó 15 kΩ-os ellenállások miatt. Ha a porthoz eszközt csatlakoztatunk, annak sebességétől függően vagy a D+ vezeték (FS eszköz) vagy a D- vezeték (LS eszköz) potenciálja emelkedik a tápfeszültség közelébe az

[4] https://en.wikipedia.org/wiki/USB (Letöltés: 2020.04.17.)

Copyright © 2021 / BME AUT

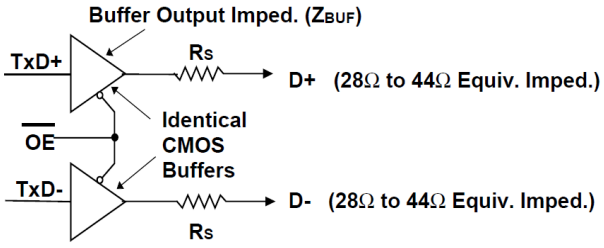
eszközben található 1,5 kΩ-os felhúzó ellenállás miatt. Ily módon a host azonosítani tudja az eszköz sebességét és ennek megfelelő adatátviteli sebességgel szólítja meg azt.

Egy teljes sebességű (FS) eszköznek a host felé mutató (upstream) kapuja a 8.13 ábra szerint épül fel. Jól látható a kapcsolási vázlaton, hogy a szimmetrikus (differenciális) adó- és vevőáramkörön kívül a vonalakra aszimmetrikus vevők (RxD+ és RxD-) is kapcsolódnak a vonalak jelszintjének külön-külön történő felismerhetőségének érdekében.



8.13 ábra – Egy teljes sebességű (FS) eszköz upstream kapuja

Az USB szabványok szerint a kábelek hullámimpedanciájának névleges értéke 90 Ω (ld. később az összefoglaló táblázatot). Ez már a 12 MHz-es adatátviteli sebesség (FS) mellett is figyelembe veendő, a nagyobb sebességek (HS, SS, SS+) esetében azonban kötelező számolnunk vele. Már a FS kimenetek esetében is kötelező jelleggel írja elő az USB 1.1-es szabvány, hogy a meghajtó áramkörök kimeneti ellenállásának a differenciális hullámellenállás felével (45 Ω) kell megegyeznie a 8.14 ábra szerint.



8.14 ábra – LS/FS kompatibilis USB meghajtó[5]

A kis (LS) és a teljes (FS) sebességnél a differenciális jelszintek egymásnak inverzei. Annak érdekében, hogy a sebességektől függetlenül lehessen kezelni a logikai jelszinteket, bevezették a J és a K állapotokat a következők szerint:

[5] Universal Serial Bus Specification Rev. 1.1

Copyright © 2021 / BME AUT

Buszállapot		Jelszint (vevő)
Differenciális „1”		(D+) – (D-) > 200 mV
Differenciális „0”		(D-) – (D+) > 200 mV
Adat J állapot	LS	differenciális „0”
	FS	differenciális „1”
Adat K állapot	LS	differenciális „1”
	FS	differenciális „0”

8.5. táblázat – A J és a K állapotok definíciója LS és FS sebességeknél

A jelszinteket és a további elektromos paramétereket a 8.7. táblázat alapján valamennyi sebességváltozatra közösen tekintjük át.

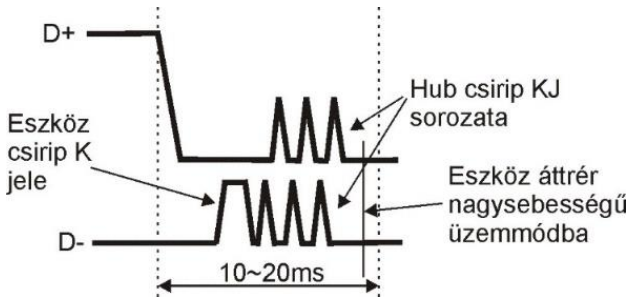
8.2.2 Az USB 2.0-s hub és eszközök kapcsolata

Nagysebességű (HS) eszközöknél az előző fejezetben bevezetett táblázat a következőképpen egészül ki:

Buszállapot		Jelszint (vevő)
Differenciális „1” (HS)		(D+) – (D-) > 300 mV
Differenciális „0” (HS)		(D-) – (D+) > 300 mV
Adat J állapot	HS	differenciális „1”
Adat K állapot	HS	differenciális „0”

8.6. táblázat – A J és a K állapotok definíciója HS sebességnél

Az USB eszközöknek azonosítaniuk kell magukat a host (hub) felé abban a tekintetben, hogy milyen sebességgel üzemelnek. Az 1.x szabvány szerinti LS és FS eszközök felhúzó ellenállások segítségével azonosítják magukat, ahogyan azt az előző fejezetben láttuk. A nagysebességű (HS) eszközök számára már másfajta azonosítási eljárást ír elő a szabvány. A HS eszközök teljes (FS) sebességű eszközként kapcsolódnak a buszra, azaz a D+ vonalat húzzák fel magas szintre. Majd egy ún. csiripeléses (chirp) módszerrel jelzik a host felé, hogy alkalmasak a nagy sebességű üzemmódra (ld. 8.15 ábra).

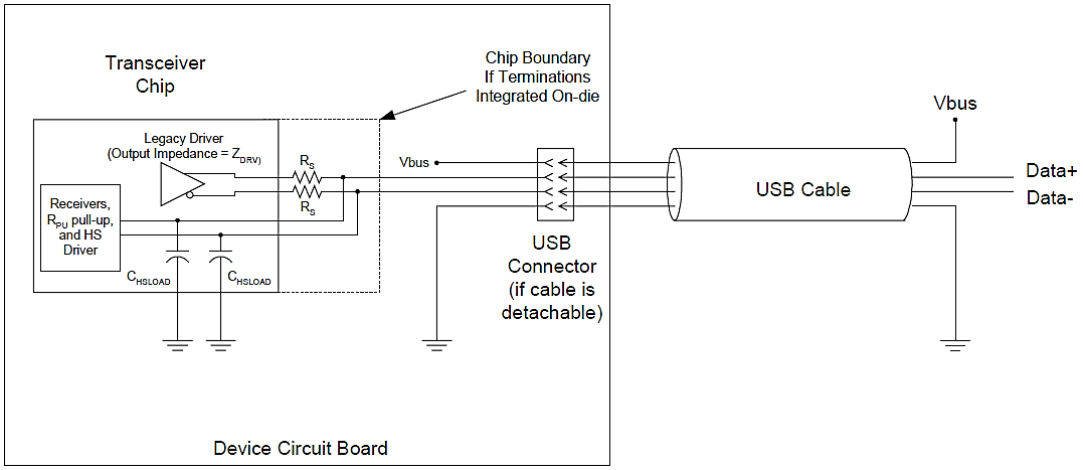


8.15 ábra – A csiripelés folyamata

Copyright © 2021 / BME AUT

A csiripelés lefolyása a következő:

- A **host** reset állapotot állít be a buszon (D+ és D- is GND potenciálra kerül).
- Az **eszköz** ezt érzékelve egy csirip K jelet ad ki 1..7 ms ideig.
- A **host** a csirip jelet észlelve váltakozó csirip K-J jeleket ad ki.
- Az **eszköz** 6 csirip jelet (3 db J-K párt) észlelve átkapcsol nagysebességű üzemmódra. Ehhez lekapcsolja a D+ vonalról a felhúzó ellenállást és engedélyezi a nagysebességű lezárást.



8.16 ábra – HS kompatibilis USB adó-vevő áramkör[6]

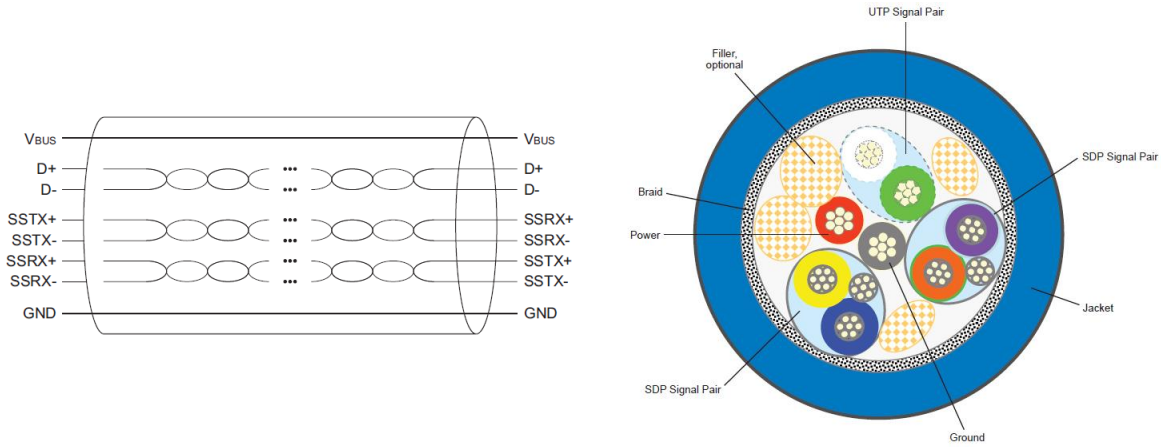
Az USB 2.0-s vonal mindkét végén 45 Ω-os impedanciával van lezárva. A vevő által érzékelt jelek vonatkozásában a szabvány egy szemábrával specifikálja a feszültségértékeket, az időzítéseket és ezek ingadozását (jitter). Egy tipikus HS kompatibilis port felépítését a 8.16 ábra mutatja.

8.2.3 Az USB 3.x-es hub és eszközök kapcsolata

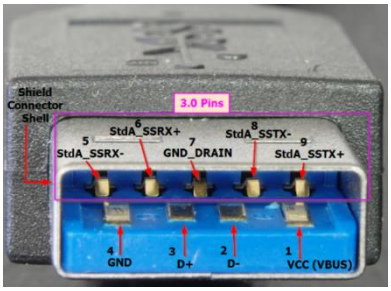
Az USB 3.0 szabvány hatalmas változást hozott: az eddigi 2 jelvezetéket (D+ és D-) megháromszorozták, két pár új, nagysebességű differenciális jelvezeték került a rendszerbe SSRX+, SSRX- ill. SSTX+ és SSTX- néven. A tápellátás céljára szolgáló VBUS és GND vonalak változatlanok maradtak (8.17 ábra). Az eddigi csatlakozóban további kontaktusok jelentek meg úgy, hogy a korábbi csatlakozókonstrukció változatlan maradt – az új típust a kék színű betét különböztette meg az eddigi fekete színűtől (8.18 ábra).

[6] Universal Serial Bus Specification Revision 2.0

Copyright © 2021 / BME AUT

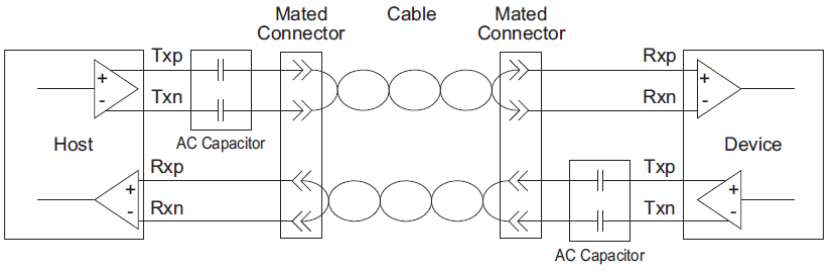


8.17 ábra – Az USB 3.0 és 3.1 egységek jelvezetékei[7]



8.18 ábra – Az USB 3.0 és 3.1 csatlakozók lábkiosztása[8]

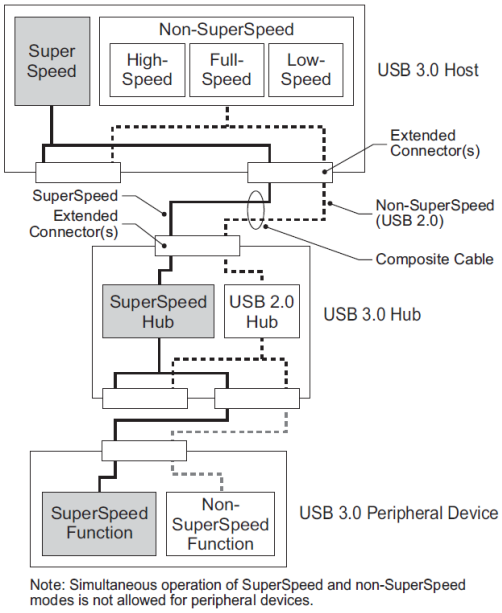
A korábbinál sokkal nagyobb sebességű (5 GHz) adó-vevő párosok kialakítása hasonló a korábbi fizikai interfészhez, de itt meghajtó oldalon kapacitív leválasztás biztosította az egységek DC elválasztását (a nagyobb sebesség miatt a differenciális jelszintek jóval alacsonyabbak, így a zajtávolságok is kisebbek lettek).



8.19 ábra – Az USB 3.x hub (host) és az eszköz kapcsolata[9]

[7] Universal Serial Bus 3.0 Specification Rev 1.0
[8] https://en.wikipedia.org/wiki/USB_3.0 (Letöltés: 2020.04.17.)
[9] Universal Serial Bus 3.0 Specification Rev 1.0

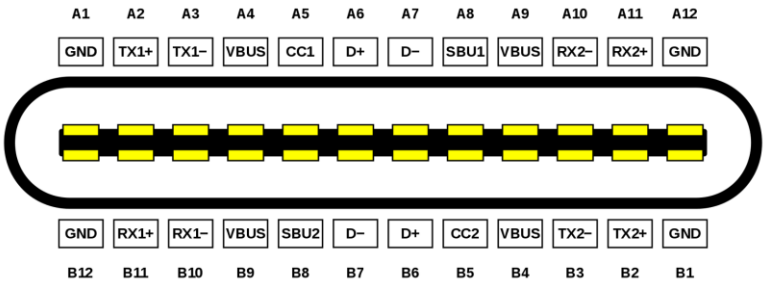
Copyright © 2021 / BME AUT



8.20 ábra – Az USB 3.0 rendszer kettős busz architektúrája[10]

Az USB 3.0 rendszerekben „két rendszer él párhuzamosan egymás mellett”: az eddigi HS-FS-LS üzemmódokat összefogó „non SuperSpeed” blokk mellé bekerült egy SuperSpeed (SS) blokk is, a csatlakoztatott eszköz teljesítményétől függően vagy az egyik (szaggatott vonal), vagy a másik (folytonos vonal) szolgálja ki azt. Az egyes egységek vagy SS, vagy „nem SS” módban működhetnek, egy egységben a két mód szimultán működtetésére nincs lehetőség, de egy SS hub két különböző portjára eltérő sebességű eszközök is csatlakoztathatók.

Az USB 3.1-es szabvány megkétszerezte az adatsebességet (10 Gb/s, Enhanced SuperSpeed vagy SS+), majd az USB 3.2 szabvány változatlan órajelfrekvencia megtartása mellett áttért az USB Type-C csatlakozó használatára (ld. 8.21 ábra), amely a korábbi Type-A, Type-B SuperSpeed csatlakozók egyetlen nagysebességű differenciális Tx-Rx csatornája (lane) helyett már két ilyen csatorna kiépítésére kínál lehetőséget.



8.21 ábra - Az USB Type-C csatlakozó lábkiosztása[11]

[10] Universal Serial Bus 3.0 Specification Rev 1.0
[11] https://en.wikipedia.org/wiki/USB (Letöltés: 2020.04.17.)

Copyright © 2021 / BME AUT

A 24 pólusú csatlakozó további nagy előnye szimmetrikus felépítése és póluskiosztása, aminek köszönhetően hossztengelye körül megfordítva is behelyezhető ellendarabjába. A két lane kihasználásával az adatátviteli sebesség változatlan órajel mellett is megduplázódott (20 Gb/s). Az USB-C szabvány a csatlakozóban található SBU1 és SBU2 (Sideband Use) jeleket kisebb sebességű oldalsávi jelátvitelre definiálja, először az USB4 szabvány kezdi el használni (sáv inicializálás, sáv engedélyezés/tiltás, USB4 port csatlakoztatás detektálása, alvó üzemmód kezelése). Hasonlóképpen a CC1 és a CC2 (Configuration Channel) jelek is ebben a szabványban definiáltak konfigurációs célokra (csatlakozó egyenesen vagy fordítva került behelyezésre, ill. host – eszköz (forrás-nyelő) felismerés is végezhető a segítségével).

8.2.4 Az USB4 szabvány elektromos jellemzői

Az USB4 szabvány rendkívül friss, jelenleg csak a szabvány előírásait ismerjük, tényleges eszközök megjelenése a piacon 2021-től várhatók. A szabvány az USB 3.2 tulajdonságaira épül, az adatátvitel frekvenciáját ismét megduplázva 20 Gb/s sebességű, kihasználva a Type-C csatlakozó már megismert lehetőségeit a 2 csatornán (lane) a teljes sávszélesség 40 Gb/s. Elektromos paramétereinek részletes elemzésével terjedelmi okokból itt nem foglalkozunk.

8.2.5 Az USB szabványok elektromos jellemzőinek összefoglalása

Az előző fejezetekben áttekintettük az USB szabvány különböző változatainak fontosabb elektromos tulajdonságait. Az egyes verziók között lényeges jelmegjelenítési/jelátviteli különbségek vannak, a 8.7. táblázatban a legfontosabb paramétereket foglaltuk össze a gyors áttekinthetőség érdekében.

			USB 1.0	USB 1.1	USB 2.0	USB 3.0	USB 3.1	USB 3.2
V_{IH}	V	Min	2,000	2,000				
V_{IL}	V	Max	0,800	0,800				
V_{OH}	V	Min	2,800	2,800	0,360			
		Max	3,600	3,600	0,440			
V_{OL}	V	Min	0,000	0,000	-0,010			
		Max	0,300	0,300	0,010			
$U_{diff,Tx}$	mV	Min				800	800	800
		Max				1 200	1 200	1 200
$U_{diff,Rx}$	mV	Min	200	200	300	30	30	30
Z_0	Ω	Nom	90	90	90	90	90	90
R_S	Ω	Nom	45	45	45	45	45	45
f_{DRATE}	Mb/s	Min	1,4775	11,97	479,76	4 998,50	9 997,00	9 997,00
		Max	1,5225	12,03	480,24	5 001,50	10 003,00	10 003,00
T_{FRAME}	µs	Min	999,50	999,50	124,94	124,94	124,94	124,94
		Max	1 000,50	1 000,50	125,06	125,06	125,06	125,06

8.7. táblázat – Az USB interfészek fontosabb elektromos paraméterei

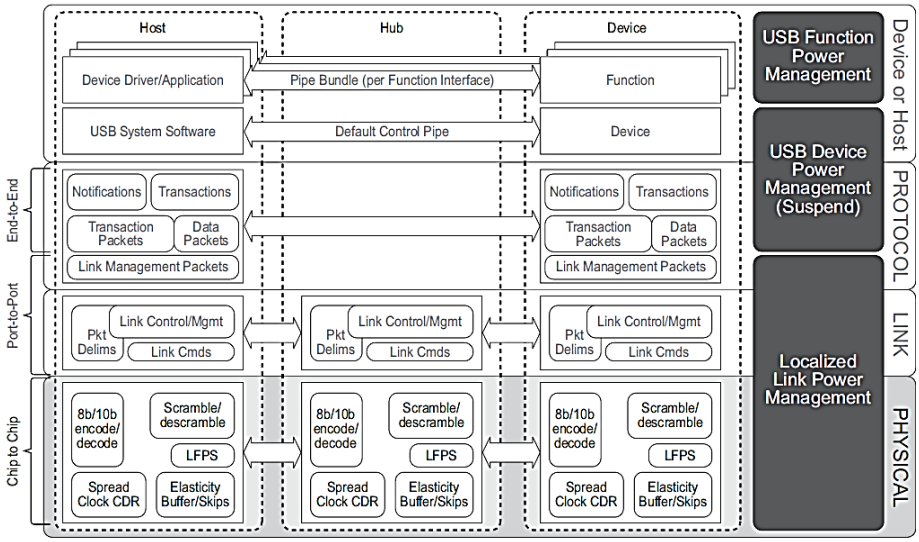
Copyright © 2021 / BME AUT

8.3 Logikai jellemzők

Az első USB szabványok még nem, de az USB 3.0-s szabvány már nagyából az OSI rétegszerkezet szerint tárgyalja a rendszerek felépítését (8.22 ábra). Ez horizontális irányban a rendszer fizikai elemeit (host – hub – eszköz), vertikálisan a működési rétegeket mutatja. Az ábra alapján jól látható az egyes rétegek (és azok funkcionális blokkjainak) előfordulása az egyes rendszerelemekben.

Az USB szabvány is (hasonlóan a korábban már megismert SATA szabványhoz) csak a fizikai, az adatkapcsolati és a transzport réteg definícióival foglalkozik, az e fölötti rétegeket már meghajtó (driver) és az operációs rendszer határozzák meg.

Jól látható, hogy a hub csak közvetítő szerepet játszik a rendszerben, kiemelve azt a 8.22 ábra középső blokkjából a két szélen elhelyezkedő host és eszköz egymással minden nehézség nélkül összekapcsolható. A hubban a 3 említett OSI réteg közül is csak a fizikai és az adatkapcsolati réteg elemei találhatók meg, a transzportréteg tulajdonságain a hub már nem változtathat, a tranzakciók, jelzések, csomagok általa csak változatlan formában továbbításra kerülnek.



8.22 ábra – Az USB 3.0 rendszerek rétegszerkezete[12]

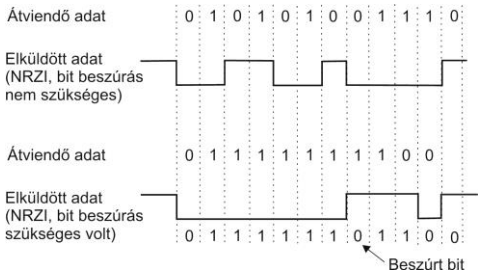
8.3.1 A fizikai réteg tulajdonságai

A fizikai réteg elektromos paramétereit, meghajtó és vevő áramköreinek tulajdonságait az előző fejezetekben már áttekintettük. Ebben a fejezetben csak azokkal a fontosabb tulajdonságokat foglaljuk, össze, melyek a fizikai réteg logikájának működéséhez tartoznak.

[12] Universal Serial Bus 3.0 Specification Rev 1.0

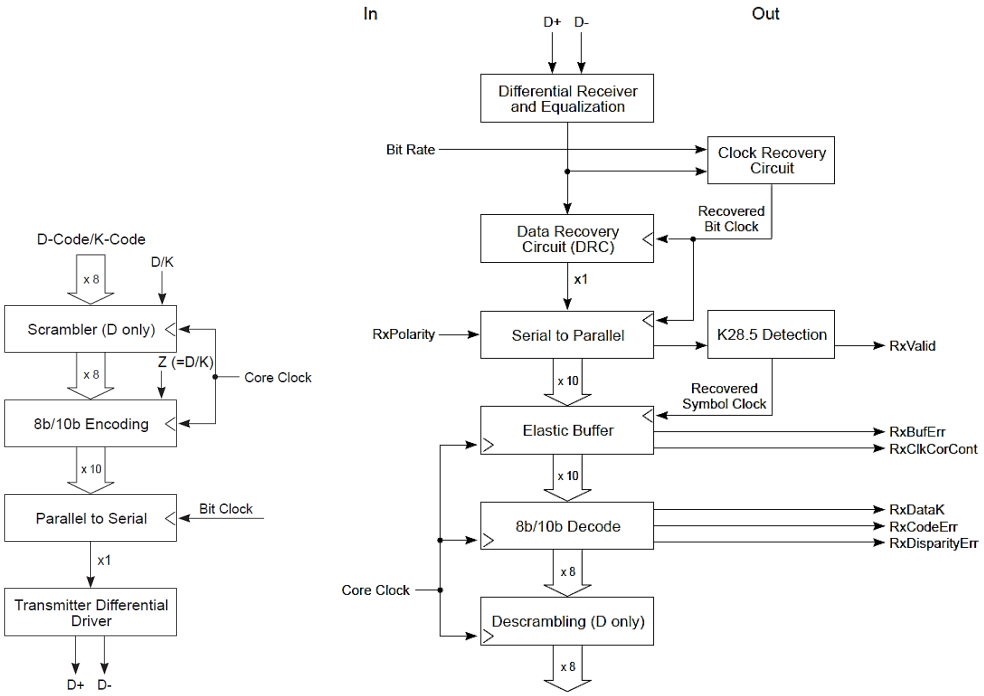
Copyright © 2021 / BME AUT

Az USB 2.0 verzióig a szabvány a kimenő adatok számára csak az NRZI-módszert írta elő (bitbeszúrással). NRZI kódolás esetén logikai „1” küldésekor nem változik a vonal állapota, minden „0” írásakor viszont ellentétesre vált. Sok 1-es esetén tehát nem változna a kimenőjel, aminek következtében a vevő kieshetne a szinkronizált állapotból (él nélkül nincs újraszinkronizálási lehetőség). Ezért hat egymást követő 1-es után az adó automatikusan beszúr egy 0-t, a vevő pedig a hat 1-est követő 0-t automatikusan „eldobja” (ld. 8.23 ábra).



8.23 ábra – NRZI kódolás bitbeszúrással

Az USB 3.0 szabvány fizikai réteg adójának és vevőjének blokkvázlatait a 8.24 ábra mutatja. A bitsebességgel már 5 GHz-nél járunk, tehát a biztonságos adatátvitelre már egészen más szabályok érvényesek, mint az alacsonyabb frekvenciák esetében.



8.24 ábra – Az USB 3.0 adó és a vevő fizikai rétegének blokkvázlata[13]

[13] Universal Serial Bus 3.0 Specification Rev 1.0

Copyright © 2021 / BME AUT

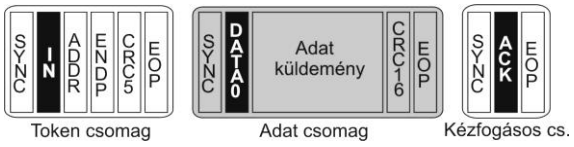
A blokkvázlatokban már megtaláljuk a SATA interfésznél megismert összekeverést (scrambler) és a 8b/10b kódolást. Már az adatkapcsolati réteg felől is érkezhetnek vezérlő („K") karakterek, ezeket természetesen mind az összekeverés, mint a kódolás figyelmen kívül hagyja (D/K bemenet). Elasztikus buffer egyenlíti ki az órajelingadozások miatti adatsebesség-ingadozásokat az adatkapcsolati réteg felé (ennek tulajdonságaival majd a PCI Express busznál foglalkozunk részletesebben).

Az USB 3.1, USB 3.2 és az USB4 szabvány ezen rétegei hasonló felépítésűek, azonban az adatátvitel hatásfokának növelése érdekében a 8b/10b kódolásról áttértek a hasonló tulajdonságú, de nagyobb adategységekben működő 128b/132b kódolásra. A 8/10=80%-os arány a hasznos adatátvitel szempontjából 20% veszteséget jelent, a 128/132=96,97%-os arány ezzel szemben csak kb. 3%-os veszteség.

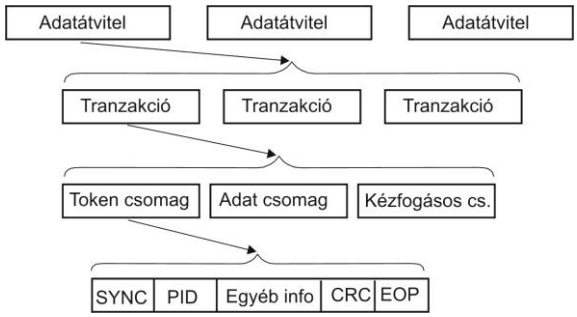
8.3.2 Az adatkapcsolati és a transzportréteg

Ezen rétegek részletes ismertetése (indoklással) egy tantárgy kereteit is kitöltené. Ebben a fejezetben pusztán szemelvényeket nyújtunk néhány fontosabb tulajdonságukból.

Az adatkapcsolati rétegben történik az adatcsomagok összeállítása. Az USB szabvány négyféle csomag-csoportot (token, adat, kézfogás, speciális) és azokon belül többféle csomagtípust (IN, OUT, DATAx, MDATA, ACK, ERR stb.) különböztet meg. Részteles ismertetésükre itt nem térünk ki. Bizonyos mennyiségű adat átvitele egy vagy több tranzakcióból, egy tranzakció egy, kettő vagy három csomagból, egy csomag pedig azonosító mezőből (PID = Packet Identifier), PID ellenőrző bitekből, s bizonyos járulékos információból áll.



8.25 ábra – Példák csomagtípusokra (IN, DATA0, ACK)



8.26 ábra – Az adatátvitel elemei az USB buszon

Az egyes csomagokat méretüktől függő hosszúságú (és „biztonságú") CRC kódok védik: a legrövidebbeket 5 bites, a közepeseket 16, és a nagyobb adatcsomagokat 32 bites CRC.

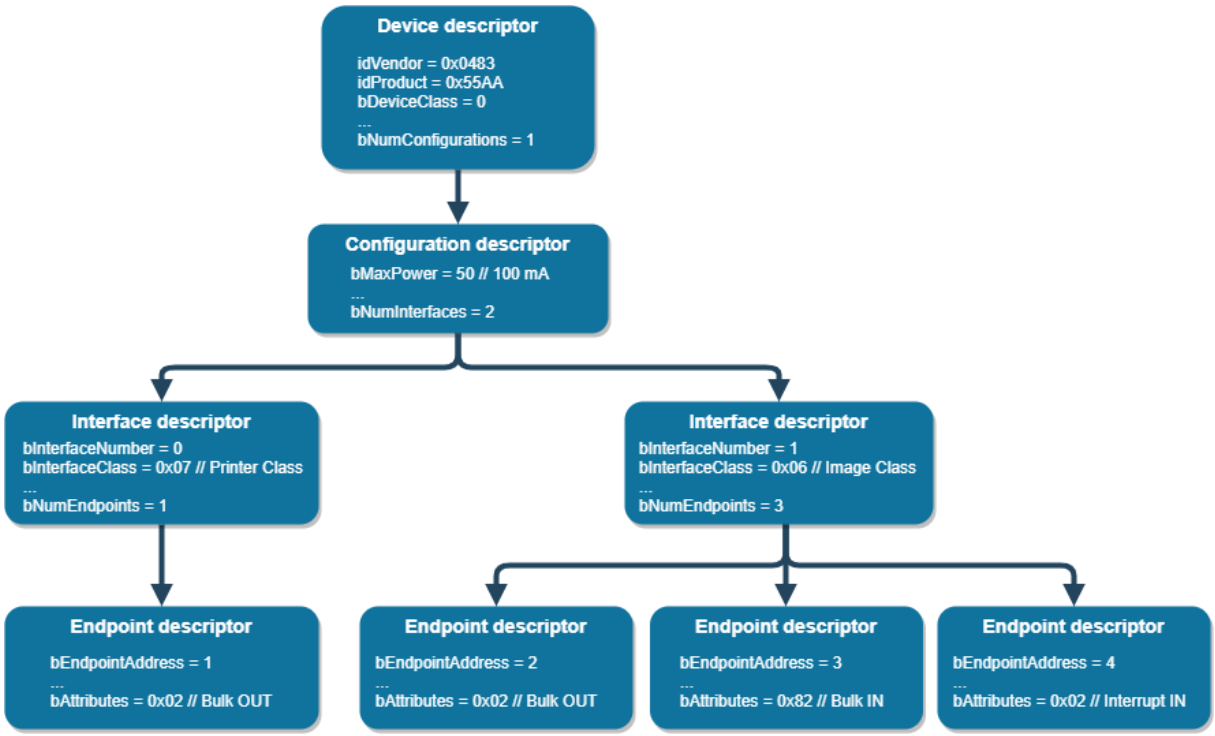
Copyright © 2021 / BME AUT

8.4 Szoftver (driver) modell[14]

8.4.1 USB leírók

Egy USB eszközt csatlakoztatva a buszra a rendszerszoftvernek fel kell ismernie a csatlakoztatott eszköz típusát, végpontjainak jellemzőit stb. annak érdekében, hogy a megfelelő eszközmeghajtót betölthesse, és a tranzakciókat megfelelően ütemezhesse. Ezt a folyamatot konfigurálásnak nevezzük, az angol szakirodalom gyakran az enumeration (számbavétel, felsorolás) kifejezést is használja. Az eszköznek a konfiguráció során beolvasott jellemzőit a korábban már említett 0-s végponton található ún. eszközleírók tartalmazzák.

A leírók előre meghatározott struktúrájú adatcsomagok, melyek az USB eszköz típusát és funkcióit írják le, melyek hierarchikus rendszerbe szerveződnek, erre láthatunk egy példát a 8.27 ábrán egy multifunkciós nyomtató és szkenner esetére.



8.27 ábra – Az eszközleírók hierarchikus rendszere

8.4.1.1 Eszközleíró (Device descriptor)

Egy USB eszköznek egy ilyen leírója létezik, ebben található az eszköz USB verziója, USB osztály típusa, a nullás végpont maximális csomagmérete és a konfiguráció leírók száma. Ezeken kívül itt találhatók az eszköz gyártójának és típusának azonosítói is (VendorID és

[14] A fejezet kidolgozásában nyújtott segítségéért szerzők köszönetüket fejezik ki Csorvási Gábor tanársegédnek.

Copyright © 2021 / BME AUT

ProductID), melyekre az operációs rendszernek a megfelelő eszközillesztő szoftverek betöltéséhez lesz szüksége. A példánkban az osztály típusa 0-s kódú, ami azt jelenti, hogy az egyes interfészek határozzák meg az osztályokat.

8.4.1.2 Konfigurációleíró (Configuration descriptor)

Egy USB eszköznek többféle konfigurációja is létezhet, így akár több konfiguráció leírója is. Ez a leíró tartalmazza az eszköz maximális áramfelvételét és az interfészleírók számát az adott konfigurációban. Több leíróra abban az esetben van szükség, ha például az eszköznek lehet kis- és nagyfogyasztású állapota is (pl. különböző módú táplálása lehetséges buszról, vagy külső tápegységről). A mindennapi gyakorlatban ritkán van egynél több leírója egy eszközöknek. A példánk esetében is csak egyetlen ilyen leírót definiálunk 100 mA-es maximális fogyasztással.

8.4.1.3 Interfészleíró (Interface descriptor)

Az interfészleíró a végpontokat csoportosítja funkcionális egységekbe, a nullás végpont kivételével. A 8.27 ábrán az látható, hogy az egyik interfész egy nyomtatót definiál a szabványos eszközosztályból, a második interfész pedig egy képolvasó eszközé az USB Image Class osztályból. A konfigurációkkal ellentétben interfészekből egyszerre több is lehet aktív állapotban.

8.4.1.4 Végpontleírók (Endpoint descriptor)

Ez a leíró tartalmazza az egyes végpontok típusát (bulk, interrupt, isochronous), azok irányát, maximális csomagméretét, és a lekérdezés gyakoriságát. Egy nyomtató esetén elegendő egy tömegadat átviteli csatorna, amin a nyomtatandó adatokat továbbítjuk. Az állapot visszakérdezése a vezérlő interfészen történik.

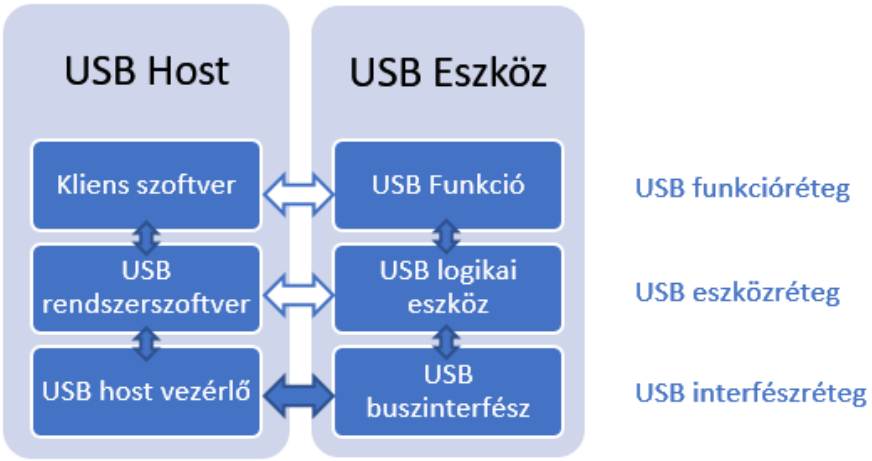
8.4.1.5 Egyéb leírók

Ide tartoznak a szöveges leírók (String descriptor), amik különböző szöveges információk közvetítésére szolgálnak. Ilyenek lehetnek a gyártó neve, az eszköz típusa, sorozatszáma stb. Bizonyos eszközosztályok további leírókat definiálhatnak.

8.4.2 Az USB host-szoftver

Az USB host-szoftver az USB specifikációra épülő réteges keretrendszer. Az egyes rétegek logikai kapcsolatot jelentenek a host és az eszközök azonos rétegei között, és elrejtik az alsóbb rétegek által használt struktúrákat és paramétereket. Korábban már említettük, hogy az első USB szabványok még nem, de az USB 3.0-s szabvány már nagyából az OSI rétegszerkezet szerint tárgyalja az USB rendszerek felépítését. Szoftver szempontból a host és az eszköz kapcsolatát a 8.28 ábra mutatja.

Copyright © 2021 / BME AUT



A kék színű nyilak fizikai, a fehér színűek logikai kapcsolatot jelölnek

8.28 ábra – A host és az eszköz réteges kapcsolata

8.4.2.1 Az USB interfészréteg

Ezt a réteget korábban már végigelemeztük, a host-szoftver szempontjából a fizikai réteget, az adatkapcsolati réteget és részben a transzportréteget foglalja magában. Itt zajlik a tényleges kommunikáció a host és az eszközök között. USB 1.0 megjelenése után két típusú hostvezérlőt használtak (oHCI – Open Host Controller Interface, uHCI – Universal Host Controller Interface). Erre reagálva az USB-IF (USB Implementers Forum) az USB 2.0-tól egységes interfészt írt elő (eHCI – Enhanced Host Controller). Az USB 3.0 esetén a SuperSpeed átvitelt külön vezérlő intézi (xHCI).

8.4.2.2 Az USB eszközréteg

Az USB host oldalán az eszközrétegben található az USB operációs rendszer szintű illesztése. Felépítése operációs rendszer függő, de alapvetően ennek a rétegnek a feladata az eszközök címének kiosztása és a megfelelő konfiguráció kiválasztása a rendelkezésre álló sávszélesség és tápellátás alapján. Továbbá itt történik az adatok feldarabolása az eszköz által meghatározott méretre. Az eszközréteg a megfelelő leírókon keresztül (eszköz- és konfigurációs leíró) látja az USB eszközt, a konkrét fizikai átvitelt nem ismeri. Ez a réteg nyújt szolgáltatást a funkciórétegnek az adatok átvitelére a csővezetékek (pipes) és folyamok (streams, messages) segítségével.

8.4.2.3 Az USB funkcióréteg

Ennek a rétegnek csak az USB eszköz interfészéről és a végpontokról vannak ismeretei. Az adatok továbbítása az eszközrétegen keresztül történik. Itt találhatóak az USB osztályok eszközvezérlői, melyeket az operációs rendszer szolgáltat, például egy tárolóeszköz (USB Mass Storage), vagy egy beviteli eszköz (USB HID) esetén. Továbbá itt futnak az általános USB eszközmeghajtók, melyek a kliensprogramoknak nyújtanak átviteli szolgáltatásokat (WinUSB,

Copyright © 2021 / BME AUT

libusb, libusbK). A gyakorlatban ezek két részből állnak: egy kernel szintű driverből (Linux esetén ez a megfelelő USB kernel modul, Windows esetén WinUSB.sys, libusb0.sys, libusbK.sys driverek egyike), és erre épülnek a felhasználói szintű szoftver könyvtárak, melyek segítségével lehet egy általános célú programon belül kommunikálni az USB eszközökkel.

Libusb

A libusb egy C nyelven írt felhasználói szintű szoftverkönyvtár. Támogatja az összes jelenlegi USB változatot. A libusb platformfüggetlen, elérhető a legtöbb operációs rendszerben. Két különböző programozói modellt használhatunk, egy szinkron és egy aszinkron interfészt.

Szinkron I/O

Az egyszerűbb szinkron interfészen keresztül elérhető a három legfontosabb átviteli mód (control, interrupt, bulk). Egyszerű használni, az általános programozási elveket követi (a függvények blokkolnak és hibakóddal térnek vissza, melyek jelzik a hívások sikerességét). Ebből következik a legnagyobb problémájuk is, mert nagyobb üzenet továbbítása jelentős időbe telhet, addig viszont a programunk nem tud reagálni a felhasználói beavatkozásra. Ezen kívül nincs lehetőség az átvitelek idő előtti leállítására.

Aszinkron I/O

Az aszinkron interfészen keresztül lehetőségünk van mind a négy átviteli módot használni. Felépítése bonyolultabb, viszont sokkal nagyobb rugalmasságot nyújt. Ebben az esetben az üzeneteket úgynevezett transzferekbe csomagoljuk, amit továbbítunk az operációs rendszer felé. A függvények azonnal visszatérnek, és az átadott függvénymutatókon keresztül (callback hívás) értesülünk az alkalmazás az átvitel sikerességéről. Az események kezelését az alkalmazásnak kell kezdeményeznie, ezt megtehetjük a programunk főciklusában, vagy egy külön szálon is.

Copyright © 2021 / BME AUT

9 A PCI buszcsalád

A személyi számítógépek kezdeti korszakában (megjelenésüktől a 90-es évek elejéig) a külső perifériákat az ISA buszra lehetett illeszteni. Ennek a busznak a megtervezésekor a perifériák sebessége még igen korlátos volt, ez azonban rohamosan változni kezdett. Megjelentek a gyors grafikus kártyák, az egyre nagyobb sebességű kommunikációs felületek, a gyors külső tárolók. Ezek nem csak gyorsabb processzorokat, hanem a rendszer központi eleme és a perifériák között is gyorsabb adatátvitelt igényeltek. Ennek következtében a magát nagyon sokáig „tartó" ISA busz már olyan szűk keresztmetszetté vált, ami a teljesítőképesség növekedését gátolta.

Egy számítógépben a leggyorsabb busz a lokális busz, melynek jeleit közvetlenül a processzor lábairól vezéreljük. A PC tervezők számára az ISA buszos számítógépek teljesítőképességének növeléséhez a legkézenfekvőbb megoldás lett volna az, hogy a gyors perifériákat közvetlenül a lokális buszra illesszék. Sok olyan számítógép-változat készült, amelyben a grafikus vezérlőt vagy más nagysebességű perifériavezérlőt közvetlenül az alaplapon helyezték el a processzor lokális buszához csatlakoztatva.

Jóllehet ez a megoldás biztosította a kívánt sebességet, két komoly hátránnyal járt együtt. Az egyik az volt, hogy a processzorhoz ily módon közvetlenül illesztett perifériák, I/O alrendszerek állandóan áttervezésre szorultak, ha a processzor megváltozott. Márpedig a gyorsabb és újabb processzorok megjelenése egyre gyakoribbá vált. Másrészt a felhasználók számára is igen rugalmatlanok voltak az így kialakított rendszerek, hiszen hiába jelent meg például egy új, nagyobb tudású grafikus alrendszer (processzor), az nem volt csatlakoztatható a régi rendszerhez.

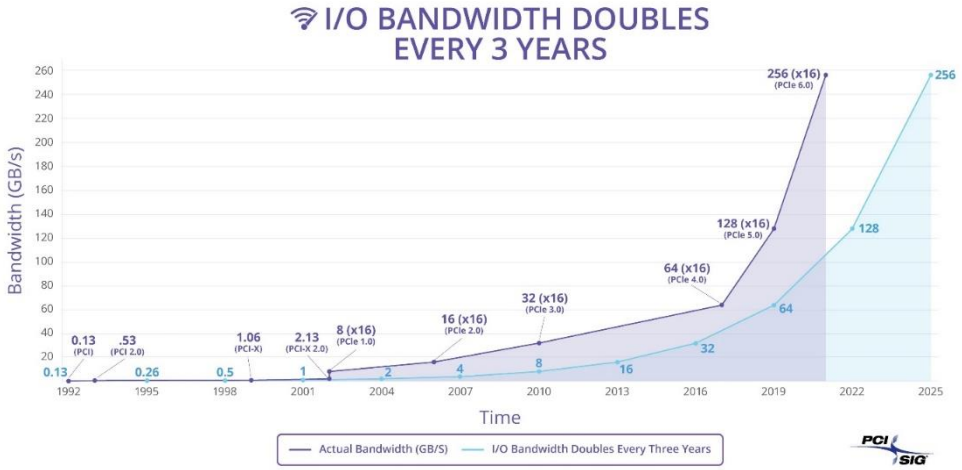
A megoldás a következő lett: a grafikus vezérlőt egy bővítőkártyán helyezték el, hogy leegyszerűsítsék az egyre nagyobb teljesítményű processzorokra való váltás (upgrade) lehetőségét. A különböző gyártók azonban eltérő módon alakították ki a csatlakoztatás módját, ezért az csak korlátozott mértékben tette lehetővé a cserét, a bővítést. Ezt a nehézséget kiküszöbölendő de facto szabványok kezdtek kialakulni: 1987-ben az IBM bejelenti az MCA (Micro Channel Architecture) interfész létrejöttét, amely az ISA 8 MB/sec sebességével szemben a kecsegtető 66 MB/s sávszélességet kínálta. Ez elegendő is lett volna az aktuális igények számára, de használatát az IBM licencdíjhoz kötötte, ami erősen mérsékelte a gyártók lelkesedését alkalmazására. Ezzel párhuzamosan fejlesztették az EISA ingyen használható buszrendszert (1988), viszont ez a max. 33 MB/s sávszélességével már nem volt elegendő a grafikus processzorok számára. 1992-ben jelenik meg a VESA Local Bus (létrehozó: Video Electronics Standards Association), amely az i486 processzorok buszrendszerének tulajdonságaira épült (100-160 MB/s), azonban a busz nem lett hosszú életű, mivel a Pentium processzorok megjelenésével azok átalakult buszrendszeréhez nehezen volt alkalmazható. 1998-tól vezeti be az Intel az AGP buszrendszert a grafikus processzorok kezelésére, amelynek 1.0-s változata 266 MB/s, későbbi 3.0-s változata 2133 MB/s sávszélességével minden igényt kielégített. Ez a busz sokáig volt jelen a PC-k alaplapján a grafikus kártyák fogadására, háttérbe szorítani csak a jelen fejezet második felében tárgyalt PCI Express busz tudta.

Copyright © 2021 / BME AUT

A grafikus kártyák mellett a többi perifériakártya is egyre nagyobb sávszélességet igényelt. A VESA lokális busz nem volt hosszú életű, hamar kiszorította a minden szempontból előnyösebb tulajdonságokkal rendelkező PCI (Peripheral Component Interconnect) lokális busz. Először 1991-ben tettek javaslatot ennek a lokális busznak a szabványosítására az Intel Technical Forumon, a specifikációk első változatát pedig 1992 júniusában tették közzé. A specifikációt az Intel, a Compaq, a DEC, az IBM és az NRC dolgozta ki. A PCI nyitott ipari szabvánnyá vált, amelyet a hamarosan megalakított PCI-SIG (PCI Special Interest Group) felügyel. 1993 áprilisában már ez a felügyelet adta ki a PCI Rev. 2.0 változatot.

A PCI busz első változata 32 bites multiplexelt adat- és címbuszon 33 MHz-es órajellel elvben 133 MB/s adatátviteli sebességet tett lehetővé. Ez óriási előrelépés volt az ISA 16 bites 8 MB/s sebességéhez képest. Az újabb változatokban a busz szélessége és az órajel is növekedett, 64 bites adatbuszon 66 MHz-es órajellel a busz elérte az 533 MB/s sebességet (PCI 2.1).

Még a PCI busz végső változata előtt megjelent a PCI-X busz. Ez a meglévő PCI architektúrán alapult, teljes mértékben kompatibilis volt vele, azaz a PCI buszra tervezett kártyák a PCI-X buszon is üzemeltethetők és fordítva. Ennek az első változata két frekvenciát specifikált, a 66 MHz-et és a 133 MHz-et, utóbbi adatátviteli sebessége elérte az 1,066 GB/s értéket. A végső változatban 133 MHz-es órajelfrekvencia mellett órajelenkét 4 adatátviteli ciklussal ez a sebesség 4,266 GB/s-re nőtt. Ez a sebesség igen nagy volt már, viszont nagyon drága áramköri megoldásokat igényelt, így csak a nagyteljesítményű szerverekben terjedt el.



9.1 ábra - A PCI buszok sávszélességének alakulása[1]

A párhuzamos buszokon a jelenlegi technológia mellett a PCI-X közel került a megvalósíthatóság felső határához. Új utat kellett keresni az adatátviteli sebesség növelhetősége és az ezzel azonos mértékben fontos alacsonyabb költségek elérése érdekében. A megoldást egy soros busz, a PCI Express (PCIe) hozta meg. Ez a busz valójában egy pont-pont összeköttetést valósít meg a két végpontja között, azaz valójában nem is busz. A kétirányú soros adatátvitel az első változatokban 2,5 GHz-es órajelet használt, ez a későbbi 2.0-s

[1] https://pcisig.com/newsroom (Letöltés: 2020.04.18.)

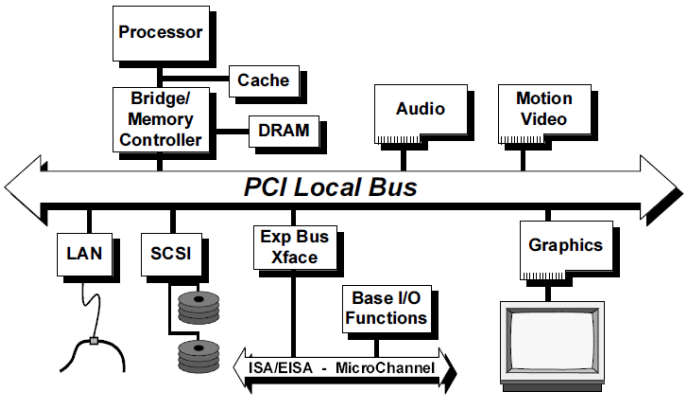
Copyright © 2021 / BME AUT

változatban már 5 GHz-re emelkedett. Az adatátviteli sebesség a két pont közötti adatátviteli sávok (lanes) számától függ, 16 sáv esetén elérte a 16 GB/s értéket. 2019 júniusában jelentette be a PCI SIG a PCIe 6.0-s változatát (várhatóan 2021-ben jelenik meg), melynek sebessége 16 csatorna esetén 256 GB/s lesz. A felügyelet büszkén hirdeti, hogy előbbre járnak, mint a korábban megfogalmazott (Moore törvényéhez hasonló) becslés, amely szerint az interfészek sávszélessége 3 évente megduplázódik (9.1 ábra).

9.1 A PCI busz

A PCI buszcsalád legelső tagja tehát a PCI busz. Az egymást követő változatok során sebességében és szolgáltatásaiban is bővült, ezek közül a legfontosabbak a következők:

- processzorfüggetlenség;
- 256 funkcionális eszköz kiszolgálásának lehetősége buszonként;
- hierarchikus rendszerben 256 PCI busz működhet együtt;
- kis fogyasztás;
- blokkos adatátvitel támogatása;
- 33 MHz vagy 66 MHz működési frekvencia;
- 32 vagy 64 bites adatbusz;
- blokkátvitelnél az írási és az olvasási idő is 1 órajel, nem blokkátvitel esetén az írási ciklus 2, az olvasási 3 órajel;
- konkurens buszműködés;
- busmester támogatás;
- rejtett buszarbitráció;
- tranzakció integritásának ellenőrzése;
- három címtartomány: memória (4 GB, opcionálisan 64 GB), I/O (64 kB, opcionálisan 4 GB) és konfigurációs;
- különböző méretű bővítőkártyák lehetősége.



9.2 ábra – PCI rendszer blokkvázlata[2]

[2] PCI Local Bus Specification Rev 2.2

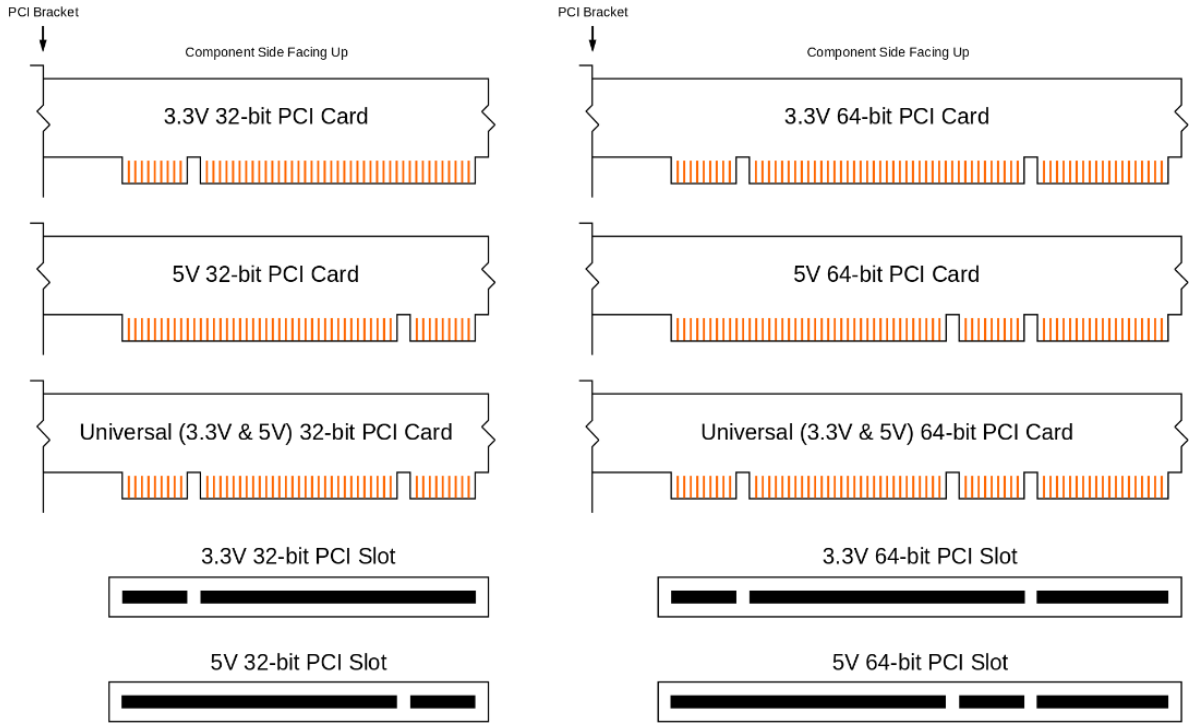
Copyright © 2021 / BME AUT

9.1.1 Mechanikai jellemzők

A PCI buszra csatlakoztatható kártyáknak két változata van: a szabványos hosszúságú és a rövid kártya. A rövid változat rögzített és változó magasságú is lehet, ezt azért vezették be, hogy a bővítőkártyákat egyszerűbb funkciók esetén minél kisebb költséggel meg lehessen valósítani. A kártyák azonban nem csak a költségek csökkentését teszik lehetővé, de kisebb fogyasztású és kisebb fizikai méretű rendszer megvalósítását is.

A PCI buszt és a csatlakoztatandó kártyákat mind 32, mind 64 bites rendszerre specifikálták. A 32 bites csatlakozója 120 érintkezős, a 64 bites ezt 184 érintkezőre egészíti ki. Mindkettőben lehetséges az 5 voltos és a 3,3 voltos jelszintű eszközök csatlakoztatása is. Egy adott jelszintű kártyának a másik jelszintű csatlakozóba való behelyezését reteszek akadályozzák meg a csatlakozókban. Másrészt a 32 bites busszal rendelkező kártyák behelyezhetők (és működnek) a 64 bites csatlakozókban is, sőt, a 64 bites kártyák is behelyezhetők a 32 bites csatlakozókba – de ilyenkor 32 bites üzemre kell konfigurálni őket (9.3 ábra).

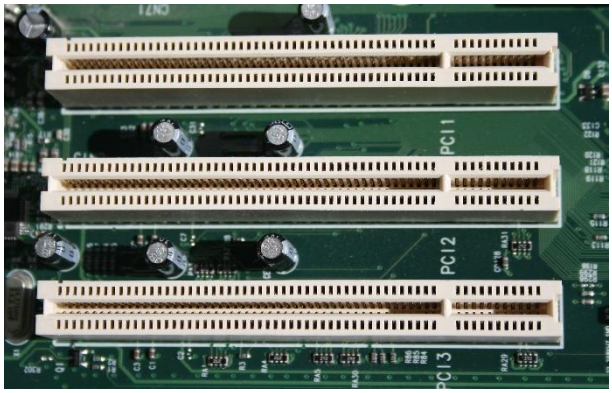
A PCI busz csatlakozói kétféleképpen helyezkedhetnek el az alaplapon. Ha az alaplapba építik be őket, akkor a kártyák szokásos módon az alaplapra merőlegesen helyezkednek el. De ún. kiemelőkártya (riser board) alkalmazására is van lehetőség, ilyenkor a kiemelőkártya áll merőlegesen és a bővítőkártyák párhuzamosak lesznek az alaplappal.



9.3 ábra – 32/64 bites, 3.3/5V-os kártyák és csatlakozóik[3]

[3] https://en.wikipedia.org/wiki/Conventional_PCI (Letöltés: 2020.04.18.)

Copyright © 2021 / BME AUT



9.4 ábra – PCI csatlakozók egy PC alaplapon (5V, 32 bit)[4]

9.1.2 Elektromos jellemzők

A PCI szabvány az elektromos jellemzőket teljes mértékben definiálja, biztosítva ezzel a kompatibilitást az alaplap-gyártók és a PCI modulok gyártói között. A busznak két olyan sajátossága van, amelyek döntően meghatározzák az elektromos jellemzőit és a buszon használható adó- és vevőáramkörök karakterisztikáit:

- egyrészt a PCI busz ún. CMOS busz az alkalmazott alkatrészek gyártástechnológiájának köszönhetően;
- másrészt működése alapján az ún. visszavert hullámra kapcsoló (reflected wave switching) busz.

Mindkét jellemző domináns a fogyasztás csökkentése és az ún. „zöld architektúra" szempontjából. A fogyasztás csökkentése pl. a mobil és a hordozható eszközökben (pl. laptop) alapvető fontosságú.

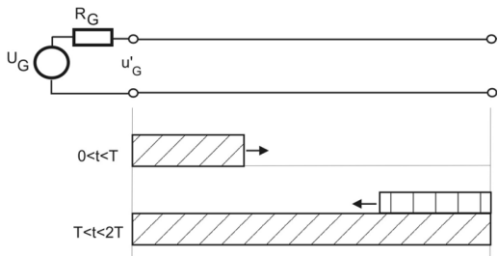
A fogyasztás csökkenése egyrészt abból adódik, hogy a CMOS technológián alapuló buszon állandósult állapotban (miután a kapcsolási tranziensek lecsengtek) az áram minimális, a meghajtóáramokat lényegében csak a felhúzóellenállások fogyasztják. A fogyasztás csökkenése másrészt a visszavert hullámra kapcsoláshoz szükséges lezáratlanság következménye. A visszavert hullámra kapcsolás azt jelenti, hogy a buszmeghajtók csak a szükséges jelszint felével hajtják meg a jelvezetéket, a teljes jelszint a lezáratlan ($R_T \approx \infty$) busz végén közel 100%-osan visszaverődő hullám hatására áll be. Ideális esetben, azaz végtelen gyors kapcsolást feltételezve, és ha a buszmeghajtó kimeneti ellenállása (R_G) megegyezik a buszvonal hullámellenállásával (Z_0), akkor

- kapcsoláskor a buszmeghajtó üresjárási feszültségének a felével azonos $U_G = V_{OH}/2$ értékű feszültséghullám és $I = V_{OH}/(2*Z_0)$ értékű áramhullám indul el a busz végpontja felé;
- a végponton a feszültséghullám +1, az áramhullám –1 értékű reflexiós tényező (ld. távíró egyenletek) miatt pozitív, ill. negatív értékkel verődik vissza;

[4] https://en.wikipedia.org/wiki/Conventional_PCI (Letöltés: 2020.04.18.)

Copyright © 2021 / BME AUT

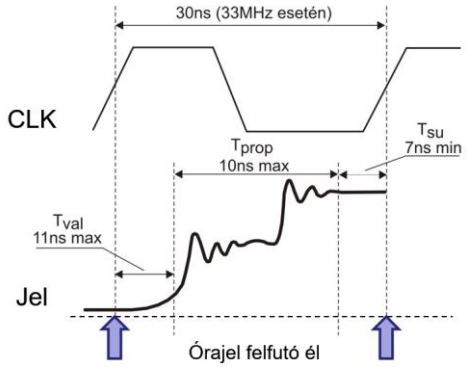
- így a buszon visszafelé haladó hullám a jelvezeték feszültségét U_G értékre, áramát pedig 0 értékre állítja be (9.5 ábra);
- amikor a hullámok elérik a buszmeghajtót, az ideális illesztés miatt nincs reflexió, a tranziens befejeződik.



9.5 ábra – Bekapcsolási (jelváltási) folyamat távvezetéken

A gyakorlatban természetesen a viszonyok eltérnek az ideálistól, végesek a kapcsolási idők, a meghajtóoldali illesztés nem tökéletes, útközben koncentrált paraméterű csatlakozók kapcsolódnak a buszra, stb., ezért a valóságos jelalakok eltérnek az ideálistól.

A 9.6 ábra egy jel alacsony-magas jelváltását szemlélteti a fontosabb időzítési paraméterek feltüntetésével. Jól látható, hogy a kívánt jelszint két lépésben alakul ki. Az első hullám hatására a jelszint kb. a feszültség végértékének a felére áll be, majd a visszavert hullám állítja be annak a teljes értékét. A PCI buszon a váltások (és a beolvasások) mindig az órajel felfutó élére történnek, az ábrán jól látható a három fontos dinamikus paraméter: T_{val} a kimenet késleltetése, T_{prop} a terjedési idő, ez alatt kell a jelnek a végértéket felvennie. (Emlékezzünk az interfész blokk bevezetőjében elmondottakra: kb. $0{,}8 \cdot c_0$ jelterjedési sebességet feltételezve az előírt max. 10 ns terjedési idő alatt a jel kb. 120 cm-t tud megtenni oda/vissza !) Végül a T_{su} az előkészítési idő (setup time), az órajel éle előtt ebben az időtartamban a jel már nem változhat.



9.6 ábra – Egy valóságos L-H jelszintváltás a PCI buszon

Fentiek miatt a szabvány a meghajtók és a vevő áramkörök számára nem csak a statikus (DC-) paramétereket specifikálja ($V_{IH} \geq 2{,}0V$, $V_{IL} \leq 0{,}8V$, $V_{OH} \geq 2{,}4V$, $V_{OL} \leq 0{,}55V$ stb.), hanem az ún. dinamikus (AC-) paramétereket is U/I jelleggörbék formájában. A példaként megadott értékek 5V-os interfészekre vonatkoznak, 3,3V-os jelekre hasonló definíciók adottak.

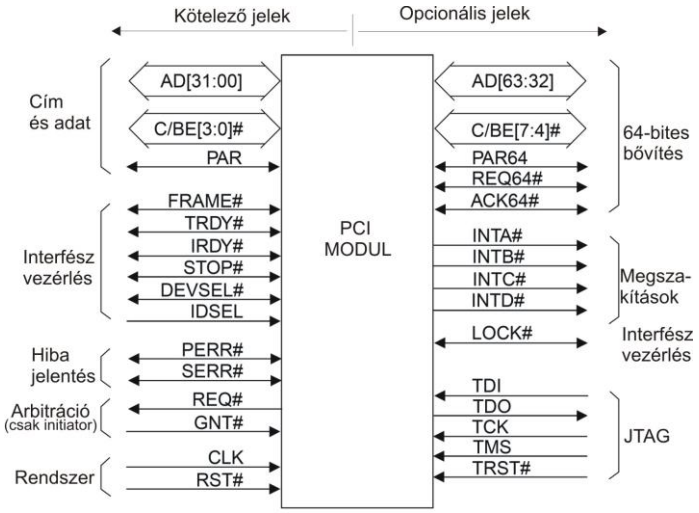
Copyright © 2021 / BME AUT

9.1.3 Logikai jellemzők

A PCI busz logikai jellemzői magukban foglalják a buszjelek értelmezését, a buszparancsok rendszerét, az információátviteli protokollt, a tranzakciókat és a konfigurációs rendszert.

9.1.3.1 A PCI busz jelei

A PCI busz rendszerében a mester-szolga szerepkörök elnevezése initiator (kezdeményező) és target (cél, célpont). Ezek rövidítései az I és a T betűk szerepelnek bizonyos jelek neveiben, mutatva, hogy az adott jel melyik szereplő jellemzője, státusza. A PCI rendszerében az initiator szerepet ellátó eszközöknek 49, a target-szerepű eszközöknek 47 jelet kötelező kezelniük. A kötelező és az opcionális jelek a 9.7 ábrán láthatók; a jelek irányánál feltételeztük, hogy initiator és target szerepkört egyaránt ellátó eszközről van szó. Az egyes jelek végén a „#” jelzés azt mutatja, hogy az adott jel aktív állapota a logikai 0 szint.



9.7 ábra – A PCI busz jelei

A jelek tárgyalásakor típusukra a következő jelöléseket használjuk:

- **in** szabványos bemeneti jel;
- **out** push-pull kimenet, szabványos aktív meghajtó;
- **t/s** háromállapotú, általában kétirányú be- és kimeneti jel;
- **o/d** nyitott drain típusú jel, amely megengedi huzalozott VAGY kapcsolat létrehozását. Az inaktív állapotot a központ felhúzással tartja fent;
- **s/t/s** fenntartott háromállapotú jel (sustained tri-state). Amelyik eszköz ezt a jelet alacsony szinten meghajtja, annak a nagyimpedanciás állapotba vezérlése előtt legalább egy órajelig magas szinten kell tartania. A következő meghajtó eszköz pedig csak a nagyimpedanciás állapotba vezérlést követően legalább egy órajel elteltét követően hajthatja meg a vonalat. A vonal felhúzása a központ részéről kötelező.

Copyright © 2021 / BME AUT

Tekintsük át az egyes jeleket, típusukat és funkciójukat – ezek ismerete elengedhetetlen a busz működésének megértéséhez.

Rendszerjelek		
CLK	in	*(Clock)* A PCI busz tranzakcióinak időzítésére szolgál, minden PCI eszköznek bemenőjele. Minden buszjel mintavételezése (az RST# és az INTx# jelek kivételével) az órajel felfutó élére történik.
RST#	in	*(Reset)* A PCI busz eszközeinek alaphelyzetbe állítására szolgál.

Cím- és adatjelek		
AD[31::0]	t/s	*(Address & Data)* Multiplexelt cím- és adatvonalak. Egy busztranzakció egy cím- és egy vagy több adatfázisból áll. Ezek a vonalak címfázis alatt az adat címét (egyetlen adatfázis esetén), vagy az adatblokk kezdőcímét (több adatfázis esetén) tartalmazzák, adatfázisban pedig az adatok átvitelére szolgálnak.
C/BE[3::0]#	s/t/s	*(Bus Command & Byte Enables)* Multiplexelt vonalak a busz parancs és a bájtengedélyezés számára. Címfázisban a busz parancsot, adatfázisban az egyes bájtok engedélyezését tartalmazzák.
PAR	t/s	*(Parity)* Egyenes (páros) paritásbit. Képzése: az AD, a C/BE és a PAR jelekben az egyesek száma mindig páros.

Interfészvezérlő jelek		
FRAME#	s/t/s	*(Frame)* A busztranzakció kezdetét jelzi. Az initiator vezérli aktív állapotba, és az utolsó adat átvitele előtt vált passzív állapotba.
TRDY#	s/t/s	*(Target Ready)* A target jelzése arra, hogy kész a jelenlegi adatfázis végrehajtására. Egy adatfázis akkor fejeződik be, ha IRDY# és TRDY# egyaránt aktívak. Olvasáskor az adat érvényességét jelzi, íráskor pedig azt, hogy kész az adatokat fogadni. Az initiator mindaddig várakozási ciklusokat iktat be, ameddig az IRDY# és a TRDY# egyaránt aktívak nem lesznek.
IRDY#	s/t/s	*(Initiator Ready)* Az initiator jelzése arra, hogy kész a jelenlegi adatfázis végrehajtására. Egy adatfázis akkor fejeződik be, ha IRDY# és TRDY# egyaránt aktívak. Íráskor az adat érvényességét jelzi, olvasáskor pedig azt, hogy kész az adatokat fogadni. Az initiator mindaddig várakozási ciklusokat iktat be, ameddig az IRDY# és a TRDY# egyaránt aktívak nem lesznek.
STOP#	s/t/s	*(Stop)* A jelenlegi target jelzi az initiator felé, hogy fejezze be az aktuális tranzakciót (nem képes valamiért lebonyolítani azt).
DEVSEL#	s/t/s	*(Device Select)* Aktív állapota azt jelzi, hogy az eszköz, mint a jelenlegi tranzakció targete, dekódolta a címet.
IDSEL	in	*(Initialization Device Select)* Konfigurációs üzemmódban az eszköz kiválasztására szolgál. Az eszközök egyedi chip-select jeleként működik, a szabvány csak ajánlásokat tartalmaz a kezelésére (pl. geografikus címzés, ld. később).

Copyright © 2021 / BME AUT

Hibajelek		
PERR#	s/t/s	*(Parity Error)* Az adatparitási hiba jelzésére szolgál a speciális ciklus (tranzakció típus, ld. később) kivételével. Az adatot fogadó eszköznek kell meghajtania két órajellel az adat vétele után. Mint valamennyi fenntartott jelet, ezt is egy órajel időtartamáig magas állapotba kell vezérelni a nagyimpedanciás állapot előtt.
SERR#	o/d	*(System Error)* A speciális ciklus (tranzakció típus, ld. később) cím- vagy adatparitás hibáját, vagy más rendszerhibát jelez.

Arbitrációs jelek (csak busmester esetén)		
REQ#	t/s	*(Request)* Az eszköz buszkérelmét jelzi az arbiter felé. Pont-pont típusú jel, tehát minden initátornak saját REQ# jele van, melyet aktív RST# (reset) állapot esetén nagyimpedanciás állapotban kell tartani.
GNT#	t/s	*(Grant)* Az arbiter az eszköz felé jelzi, hogy megkapta a busz vezérlésének a jogát. Ez is egy pont-pont típusú jel, tehát minden initátornak saját GNT# jele van, melyet aktív RST# (reset) állapotban figyelmen kívül kell hagyni.

64 bites busz bővítés (opcionális)		
AD[63::32]	t/s	*(Address & Data)* Multiplexelt vonalak, a cím és az adatvonalakat 64 bitesre bővítik. Ha a parancs DAC (Dual Address Command = duális cím parancs) és a REQ64# vonal aktív, akkor a 64 bites cím felső 32 bitjét tartalmazza a címfázisban. Az adatfázisban, ha REQ64# és ACK64# egyaránt aktívak, akkor a 64 bites adat felső 32 bitjét tartalmazza.
C/BE[7::4]#	t/s	*(Bus Command & Byte Enables)* Multiplexelt vonalak a busz parancs és a bájtengedélyezés számára. Címfázisban a busz parancsot, adatfázisban az egyes bájtok engedélyezését tartalmazzák.
PAR64	t/s	*(Parity Upper DWORD)* A felső duplaszó paritásbitje. PAR64 teljesen hasonlóan működik a felső duplaszóra, mint PAR az alsó 32 bitre. Képzése: az AD[63::32], a C/BE[7::4] és a PAR64 jelekben az egyesek száma mindig páros.
REQ64#	s/t/s	*(Request 64-bit Transfer)* Az aktuális initiator ezzel jelzi, hogy 64 bites átvitelt kíván végezni.

Megszakításkérő jelek (opcionális)		
LOCK#	s/t/s	*(Lock)* Atomi műveletet jelez a híd felé, amely több tranzakció osztatlan elvégzését jelenti. A szabvány külön mellékletben foglalkozik ennek a jelnek a kezelésével eltérő hardver környezetekben.
INTA#, INTB# INTC#, INTD#	o/d	*(Interrupt x)* Megszakításkérő jelek.

Copyright © 2021 / BME AUT

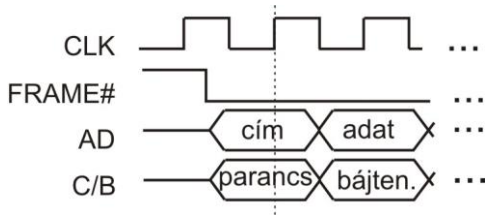
JTAG / peremfigyeléses teszt jelei (opcionális)		
TDI	in	*(Test Data Input)* Az IEEE 1149.1 JTAG szabvány (Test Access Port and Boundary Scan Architecture) szerinti tesztadatok beléptetésére szolgáló bemenet. A szabvány előírása szerint az eszközök TDO-TDI lábait láncolni kell a működtetéshez, amennyiben az adott kártya nem használja a teszt lehetőségét, a TDO-TDI lábakat össze kell kötnie egymással.
TDO	out	*(Test Data Output)* A JTAG szabvány szerinti tesztadatok kimenete az adott target egységről. A szabvány előírása szerint az eszközök TDO-TDI lábait láncolni kell a működtetéshez, amennyiben az adott kártya nem használja a teszt lehetőségét, a TDO-TDI lábakat össze kell kötnie egymással.
TCK	in	*(Test Clock)* A JTAG szabvány szerinti órajel az állapotinformáció és a tesztadatok be- és kiléptetésének ütemezésére.
TMS	in	*(Test Mode Set)* A JTAG szabvány szerinti TAP-vezérlő állapotának vezérlőjele.
TRST#	in	*(Test Reset)* A JTAG szabvány szerinti TAP-vezérlő aszinkron inicializálása.

Járulékos jelek (nem szerepelnek a kötelező és az opcionális jelek felsorolásában)		
PRSNT[1::2]#	in	*(Present)* A bővítőkártyák jelei. Ezzel jelzik az alaplap felé, hogy fizikailag jelen vannak a buszon, és hogy mekkora a tápáram-igényük. Amennyiben mindkét vonal nyitott, az a kártya hiányára utal, a két vezetéken történő L-szintek 3 kombinációban 7.5, 15 vagy 25W max. teljesítményfelvételt jeleznek.
CLKRUN#	in, o/d, s/t/s	*(Clock Running)* Opcionális jel. Eszközinputként az órajel státusza (van órajel), kimenetként az órajel elindítása és gyorsítása kérhető vele.
M66EN	o/d	*(66 MHz Enable)* Az eszközök felé jelzi, hogy a busz 66 vagy 33 MHz-es frekvenciával működik, az eszköz L szinttel tudja jelezni, hogy nem képes 33 MHz felett működni.
PME#	o/d	*(Power Management Event)* Opcionális jel, amellyel az eszköz a tápellátás állapotának megváltoztatását kérheti (wake-up funkció).
3.3Vaux	s/t/s	(Auxiliary 3.3V) Opcionális 3.3V-os tápfeszültség a tápellátás-menedzsmentet használó kártyák számára, ha a fő tápellátás ki van kapcsolva. Részletesen ld. a *PCI Bus Power Management Interface Specification* dokumentumban.

Copyright © 2021 / BME AUT

9.1.3.2 Buszparancsok

Az initiator buszparanccsal közli a target eszközzel, hogy milyen típusú tranzakciót kíván végrehajtani. A tranzakció első fázisa mindig a címfázis, amelynek kezdetét a FRAME# jel aktív állapotba váltása jelzi. Ekkor a multiplexelt AD-vonalak a target eszköz címét, a C/BE vonalak pedig a kódolt parancsot tartalmazzák (9.8 ábra).



9.8 ábra – Címfázis és buszparancs

A C/BE[3::0]# vonalakon kódolt buszparancsok a következők lehetnek:

C/BE[3::0]#	Parancstípus	Leírás
0000	Megszakítás elfogadás	A megszakítási vektor beolvasása a rendszer megszakításvezérlője számára az adatvonalakon.
0001	Speciális ciklus	Egyszerű üzenetközvetítési ciklus broadcast üzenetek átvitelére (a szabvány „logikai vezetéknek” hívja az egyes eszközök felé).
0010	I/O olvasás	I/O tartományban működő egység olvasása.
0011	I/O írás	I/O tartományban működő egység írása.
…		
0110	Memória olvasás	Memória tartományban működő egység olvasása.
0111	Memória írás	Memória tartományban működő egység írása.
…		
1010	Konfig. olvasás	Az eszköz konfigurációs tartományának olvasása.
1011	Konfig. írás	Az eszköz konfigurációs tartományának írása.
1100	Memória olvasás, többszörös	Hasonló a memóriaolvasáshoz, de azt jelzi, hogy a mester több olvasási ciklust kíván egymás után végrehajtani.
1101	Duális címciklus	64 bites cím átvitele két 32 bites ütemben, ha olyan eszközt címezünk, amely rendelkezik 64 bites címtartománnyal, és a cím nem az alsó 4 GB tartományban van.
1110	Memóriaolvasás cache határig	Hasonló a memóriaolvasáshoz, de azt jelzi, hogy a mester egy komplett cache sort kíván beolvasni egymás után több ciklussal, megszakítás nélkül.
1111	Memóriaírás és érvénytelenítés	Hasonló a memóriaíráshoz, de azt jelzi, hogy a mester egy komplett cache sort kíván végig írni a targetben megszakítás nélkül.

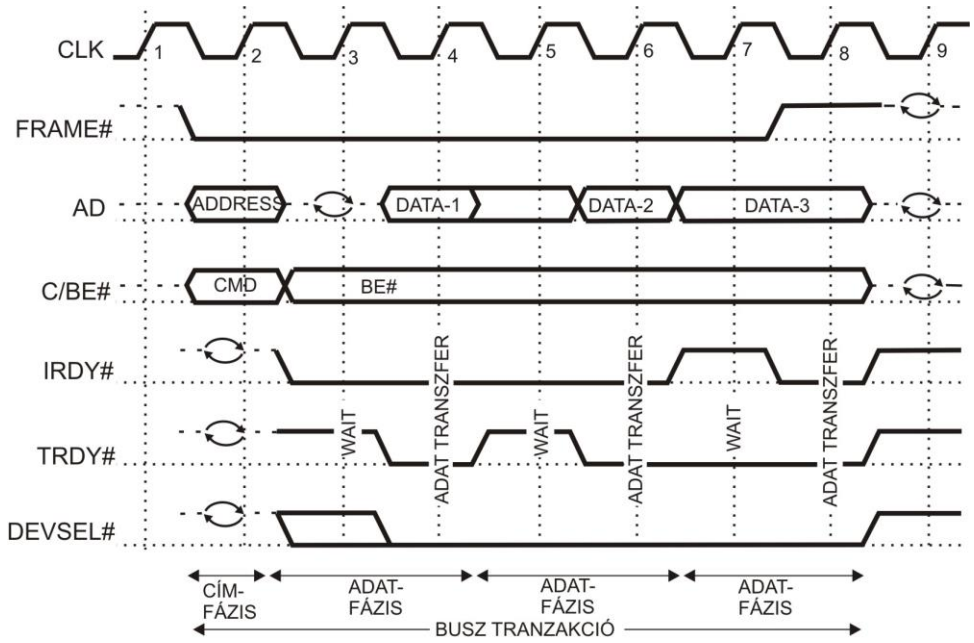
Copyright © 2021 / BME AUT

9.1.3.3 Tranzakciók

A következőkben néhány tipikus tranzakciót mutatunk be a PCI busz működésének szemléltetésére. Nem ismertetjük az összes tranzakciótípust, ezek részletes leírása a szabványban megtalálható. Az idődiagramokon a következő konvenciót használtuk:

- a folytonos vonallal jelölt jeleket az initiator vagy a target aktívan meghajtja;
- a szaggatott vonalakkal rajzolt jelek nincsenek aktívan meghajtva. Ennek ellenére stabil értékűek, ha a szaggatott vonal magas logikai jelszinten van;
- a háromállapotú jelek definiálatlan értékűnek tekintendők, ha a szaggatott vonaluk a magas és az alacsony jelszint között van (pl. az AD vonalak).

A PCI busz alapvető adatátviteli mechanizmusa a blokkátvitel. Egy blokk egy címfázisból és egy vagy több adatfázisból áll. A jelek mintavételezése mindig az órajel felfutó élére történik (a továbbiakban ezt rövidítve egyszerűen csak „órajel”-nek hívjuk). A tranzakciók vezérlésére alapvetően 3 jel, a FRAME#, az IRDY# és a TRDY# jelek szolgálnak. A busz szabad állapotában mindhárom jel passzív (felhúzott). A FRAME# jel aktívvá válása után az első órajel a címfázist jelöli ki, ekkor megy végbe a cím és a buszparancs átvitele. Az ezt követő órajelek már adatfázisokat jelölnek ki, ha az IRDY# és a TRDY# egyaránt aktív. Az adatfázisokba várociklusok is beszúrhatók az IRDY# vagy a TRDY# passzív állapotba vezérlésével.



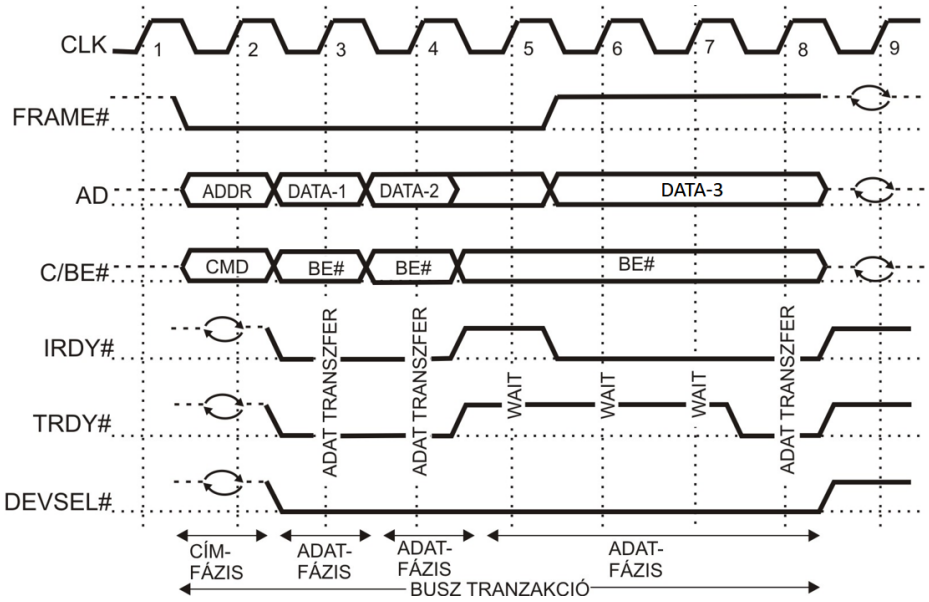
9.9 ábra – Olvasás tranzakció a PCI buszon

Egy olvasás tranzakció idődiagramját a 9.9 ábra mutatja. A tranzakció a FRAME# jel aktívvá válását követő első órajelre a címfázissal indul (2. sz. órajel). Az AD[31::0] a target címét, a C/BE[3::0]# pedig a buszparancsot tartalmazza. A target a DEVSEL# jellel jelzi, hogy a címből magára ismert és a tranzakciót kiszolgálja. Olvasási tranzakció lévén a címfázist követően megfordul az adatáramlás iránya az AD vonalakon. Hogy véletlenül se történhessen összehajtás

Copyright © 2021 / BME AUT

(eddig a mester hajtotta a vonalakat, mostantól a targetnek kell aktívan meghajtania), egy órajel periódusig a target nem hajthatja meg a vonalakat (ún. turnaround-cycle, az ábrán ezeket oda-vissza kettős nyilak jelölik). A target ezt kötelezően a TRDY# passzív állapotával kényszeríti ki.

Az adatfázis(ok)ban a C/BE# vonalak már a bájtonkénti engedélyezőjeleket tartalmazzák. A példában 3 adatátviteli fázis követi egymást, az első után a target szúr be egy váróciklust (TRDY# passzív), a második után az initiator (IRDY# passzív). A C/BE# vonalak akár ciklusonként változhatnának is, mindig egy ciklussal az adatátvitel előtt. A FRAME# jelet a mesternek az utolsó adat átvitele (8. sz. órajel) előtt meg kell szüntetnie. A példában a FRAME# passziválása már a 6. sz. órajel után meg kellene történjen, azonban a mester tudja, hogy késleltetni fogja az adatfázist (IRDY# passzív), ezért nem passzíválhatja még ekkor a FRAME# jelet.

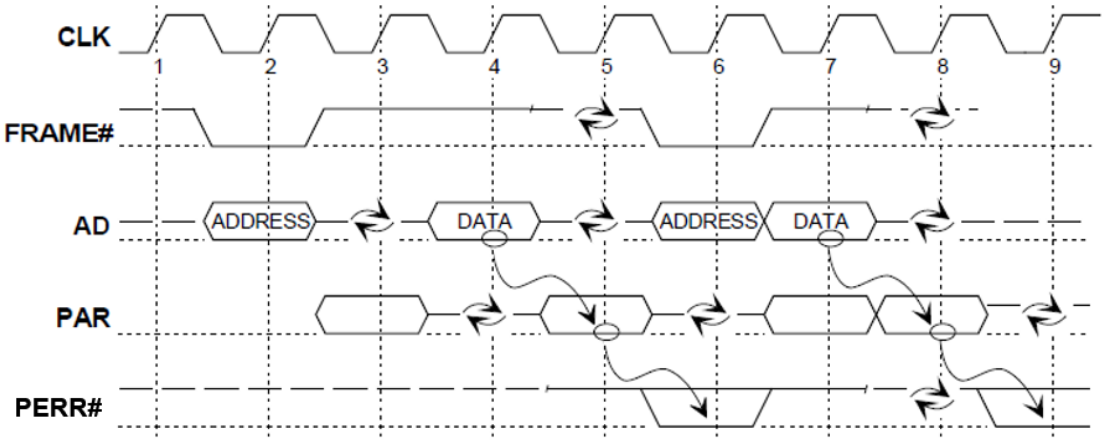


9.10 ábra – Írás tranzakció a PCI buszon

Az írás folyamatát mutatja a 9.10 ábra. Az idődiagram nagyon hasonló az olvasáséhoz, azzal az eltéréssel, hogy lévén írási tranzakcióról van szó, nincs fordulás az AD vonalak meghajtásában (a címet is a mester hajtotta meg, az adatot is ő fogja), így nincs szükség egy ciklus kihagyására: a 2. sz. órajel (címfázis) után azonnal következhet az adatfázis (3. sz. órajel). A harmadik adatfázist a mester 1, a target 3 váróciklussal késlelteti (5.-7. sz. órajelek), a mester hiába vette vissza a FRAME# jelet passzív állapotba (itt is az IRDY# után, egyébként ezt már egy órajellel hamarabb is megtette volna), nem tudhat viszont előre arról, hogy a target tovább fogja késleltetni az adatátvitelt.

A PAR jel minden adatátvitelnél az adatátvitelt követő órajel alatt aktív. Paritáshiba esetén ennek megfelelően a vevő a PERR# jel aktiválásával két órajel késéssel jelezheti a hibát az adónak. A 9.11 ábrán előbb egy olvasási, majd egy írási tranzakciót láthatunk paritáshiba jelzéssel. A paritás ellenőrzése minden résztvevő számára kötelező.

Copyright © 2021 / BME AUT



9.11 ábra – Paritáshiba jelzése a PCI buszon[5]

9.1.3.4 A rendszer konfigurálása

A személyi számítógépek korai buszaira illesztett bővítőkártyák konfigurálása (pl. címtartománybeli elhelyezkedése) mikrokapcsolókkal történt. Ezekkel ki lehetett (kellett) választani a kártya működési paramétereit, mint például a memória vagy az I/O címtartományt, a megszakítási vonalakat, a DMA csatornát, stb. Egy ilyen kártya konfigurálása komoly szakértelmet követelt meg a PC hardverről, a helytelen konfigurálás pedig konfliktusokat okozhatott más kártyákkal (címütközések, azonos megszakítási vonalak, stb.).

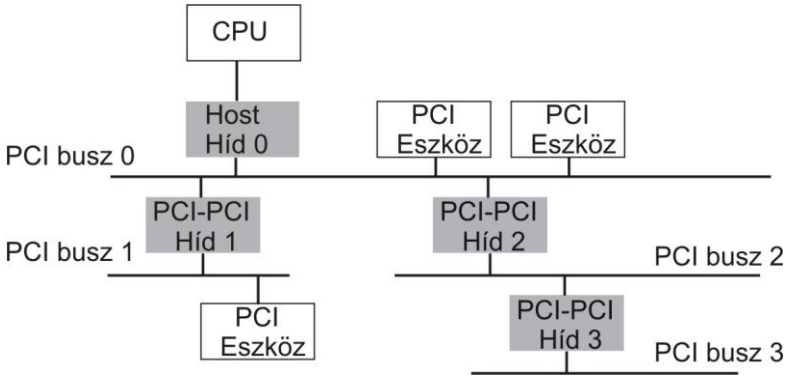
A PCI interfész egyik kulcsfontosságú tulajdonsága, ami megkülönbözteti az olyan korai buszoktól, mint pl. az ISA-busz, a rendszer dinamikus konfigurálhatósága az erőforrások közötti konfliktusok elkerülésére. Népszerű nevén ezt „Plug-and-Play” (helyezd be és működik) konfigurálhatóságnak is nevezik.

Dinamikus – más néven szoftvervezérelt – konfiguráláskor a bővítőkártyák tartalmazzák mindazt az információt, ami nekik a helyes működésükhöz szükséges: mennyi memória és I/O címtartományt igényelnek, szüksége van-e az eszköznek megszakításra, DMA átvitelre stb. A konfigurációs szoftver a rendszer betöltésekor valamennyi bővítőkártyáról beolvassa ezeket az információkat – ide értve akár a gyártót, az eszköz pontos típusát, aminek segítségével még a saját meghajtóprogramja is pontosan azonosítható és letölthető a gyártótól. Ezt követően a konfiguráló szoftver úgy osztja ki az egyes kártyák működési paramétereit, hogy a működésük során ne legyenek erőforrás-ütközések. Így pl. a kártyák saját regisztereibe írják be, hogy hol legyen a báziscímük, melyik megszakítást használják, stb. Ezeket a célokat szolgálja a PCI rendszerben az ún. konfigurációs címtartomány.

A PCI rendszer alkalmazási céljaiból adódó sajátossága, hogy az egyetlen PCI buszra csatlakoztatható eszközök száma alacsony. Ezt a korlátot a PCI-rendszer a hierarchikus rendszerben összekapcsolt PCI-buszokkal (szegmensekkel) szünteti meg, az egyes szegmenseket ún. PCI-PCI hidak kapcsolják össze (9.12 ábra).

[5] PCI Local Bus Specification Rev 2.2

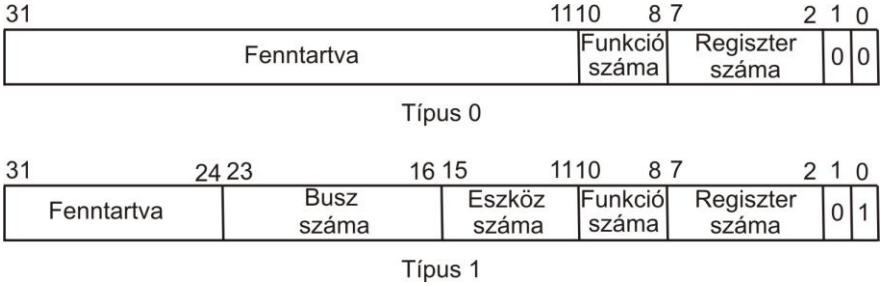
Copyright © 2021 / BME AUT



9.12 ábra – Hierarchikus PCI buszrendszer

Ebben a rendszerben egy PCI-PCI hídnak tudnia kell, hogy a tranzakciós hozzáféréseket – ide értve a konfigurációs hozzáférést is - mikor kell továbbítania a híd mögötti eszközök felé. A bekapcsolás utáni (konfigurálás előtti) állapot annyiban különbözik a későbbi állapottól, hogy az eszközök (és a hidak) még nem ismerik a rendszer teljes konfigurációját, nem kapták meg a báziscímeiket – „nincs mit dekódolni”, nem tudni, hogy milyen kezdeményezésekre és címekre kell válaszolni. Emiatt külön vizsgáljuk a rendszer konfigurációs fázisát, bármilyen furcsa is, a normál üzem a működési paraméterek ismeretében ennél már egyszerűbb lesz.

Induljunk onnan, hogy a hierarchiának megfelelően a PCI rendszer kétféle konfigurációs tranzakciót, illetve címtípust használ – fizikailag ezek a bitek kerülnek továbbításra a címfázisban az AD vonalakon (ld. 9.13 ábra):



9.13 ábra – Konfigurációs címtípusok

A használt fogalmak a következők:

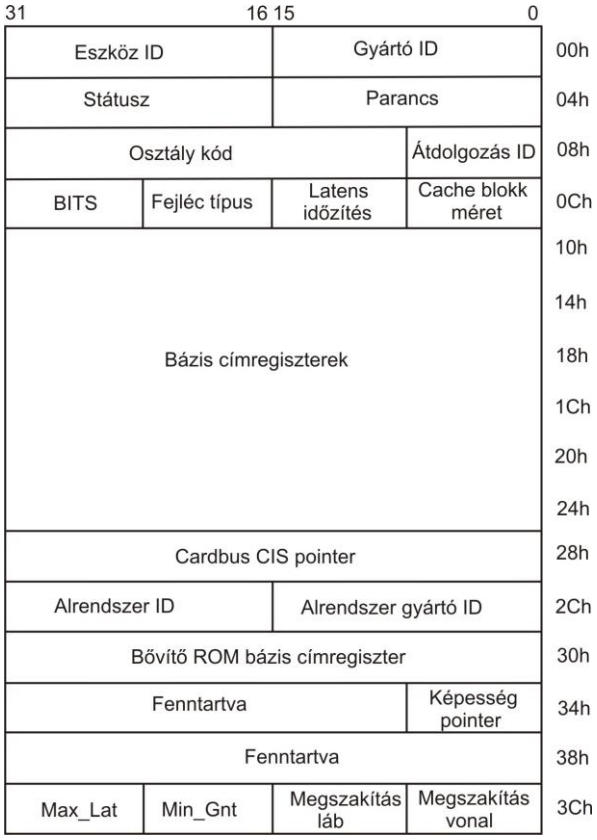
- *Regiszter száma*: a kiválasztott target konfigurációs címtartományában (256 bájt) található duplaszó címe. Lévén a címzés csak duplaszavanként lehetséges, az alsó két címbit értéke mindig 0.
- *Funkció száma*: minden multifunkcionális eszköz max. 8 funkciót tartalmazhat, a funkció száma ezek közül jelöli ki valamelyiket.
- *Eszköz száma*: az adott lokális buszon a lehetséges 32 eszköz egyikének a címe.
- *Busz száma*: a rendszerben lehetséges 256 busz egyikét azonosítja.

Copyright © 2021 / BME AUT

Ha egy hídnak 0-s típusú konfigurációs tranzakciót kell generálnia, akkor az eszköz száma alapján választja ki, hogy melyik IDSEL vonalat aktíválja ehhez. A 0-s típusú tranzakciók a lokális PCI szegmensen mennek végre, nem terjednek át a hidakon keresztül alsóbb szegmensekre és a lokális szegmens valamelyik eszközének kell azt fogadnia. Az 1-es típusú tranzakcióra csak a hidak reagálhatnak. Ezek dekódolják a busz számát és ez alapján eldöntik, hogy a busz „mögöttük van-e”. Ha nem, akkor figyelmen kívül hagyják a tranzakciót, ha igen, akkor elfogadják azt. Amennyiben ez a buszszám nem az adott híd után van közvetlenül, akkor a tranzakciót változtatás nélkül továbbengedik. Ha azonban a buszszám közvetlenül a híd mögötti lokális buszt specifikálja, akkor a híd a tranzakciót 0-s típusúvá alakítja és az eszközszámot dekódolva az egységet a megfelelő IDSEL jel aktív állapotba vezérlésével választja ki (geografikus címzés, hiszen az adott IDSEL már fix elhelyezkedésű kártyához csatlakozik).

Az IDSEL jelek előállításának módját nem írja elő pontosan a szabvány. Történhet ez

- a hídvezérlő ezt a célt szolgáló digitális kimeneteinek közvetlen vezérlésével (ez max. 32 kimenetet jelent);
- az előző kimenetek akár kódoltak is lehetnek, mivel egyszerre két különböző eszközre konfigurációs tranzakció nem történhet (így már csak max. 5 kimenetet jelent);
- a célra (néhány külső kapuval) akár az AD vonalak is felhasználhatók, hiszen a 0-s típusú tranzakcióban az AD vonalak 31..11 bitjei nincsenek használatban (21 vezeték).

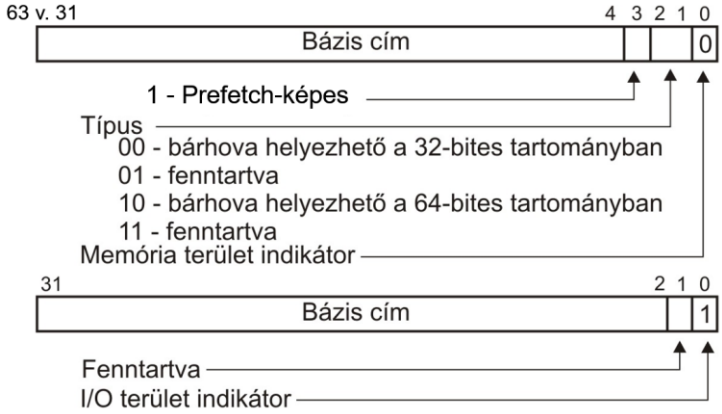


9.14 ábra – A konfigurációs tartomány (0-s típus)

Copyright © 2021 / BME AUT

A konfigurációs címtartomány 256 bájtját minden PCI buszra csatlakoztatott egységnek tartalmaznia kell. Ennek az első 64 bájtját (16 duplaszó, 00..3Fh) definiálja a szabvány (9.14 ábra). Pontosabban több ilyet is definiál (ld. „Fejléctípus” mező), gyártótól függően a mező 10h-tól kezdődő tartománya máshogyan is felépíthető. A tartomány egyes mezőinek pontos jelentését terjedelmi okokból nem tárgyaljuk, a mezőnevek önmagukban is beszédesek a tartalmuk vonatkozásában.

A dinamikus konfiguráláshoz elengedhetetlen a báziscímek megadásának lehetősége, melyeket a fejezet bevezetőjében már tárgyaltunk. A konfigurációs tartomány 10h-tól kezdődő területe 6 duplaszó számára foglal helyet, ezek pontos felhasználására kötelező előírást nem tartalmaz a szabvány. Előírja viszont a következőket az egyértelműsítés érdekében:



9.15 ábra – Memória és I/O báziscímregiszter[6]

- 10h-tól kezdve a báziscímregisztereknek sorban követniük kell egymást;
- 2 duplaszó (DWORD) összevonható egyetlen 64 bites szóvá (QWORD);
- memória báziscím 32 vagy 64 bites lehet, I/O báziscím csak 32 bites;
- a memória báziscímek 16-tal, az I/O báziscímek 4-gyel oszthatók kell legyenek – így báziscímek alsó 4 ill. 2 bitje mindenképpen 0, amit a 9.15 ábra szerint kell kitölteni.

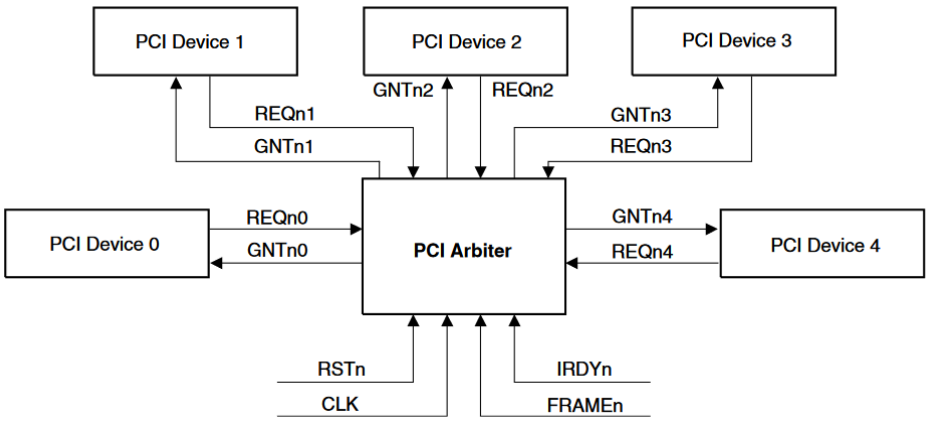
9.1.3.5 Arbitráció

A PCI buszon több mester képességű eszköz is előfordulhat. Amennyiben egy buszon több mester résztvevő lehetséges, rendelkezni kell a busz vezérlésének jogáról. Mielőtt egy mester elindítana bármilyen tranzakciót a buszon, meg kell szereznie a vezérlés jogát. A folyamatot arbitrációnak, a folyamat vezérlőjét arbiternek nevezzük. A PCI busz ún. független kérelmeket kiszolgáló központi arbitrációs rendszert használ. A PCI vezérlők egy része nem is foglalkozik ezzel a kérdéssel, az arbitrációs feladatokat önálló vezérlőegységre bízzák, amely néhány PCI rendszervezérlő jel (CLK, RST#, FRAME#, IRDY#) ismeretében képes is ellátni ezt a feladatot. Az arbitráció „rejtett”, vagyis a következő tranzakció arbitrációja lebonyolítható az

[6] „Prefetch-képes” paraméter alatt olyan memóriastruktúrát kell érteni, amely a fizikai hozzáféréstől nem változtatja a címét vagy a tartalmát. A konvencionális memóriák ilyen tulajdonságúak, de pl. egy FIFO-tár nem.

Copyright © 2021 / BME AUT

aktuális tranzakció alatt: a beérkező aktív REQ# igényekre a kiválasztott egység GNT# jelének aktiválásával ítélhető oda a busz vezérlésének a joga (minden kártyahely saját REQ# és GNT# jelekkel rendelkezik). A kiválasztott vezérlőnek meg kell várnia az aktuális tranzakció befejeződését, amit a FRAME# és az IRDY# jelek együttes passzív állapota jelez.

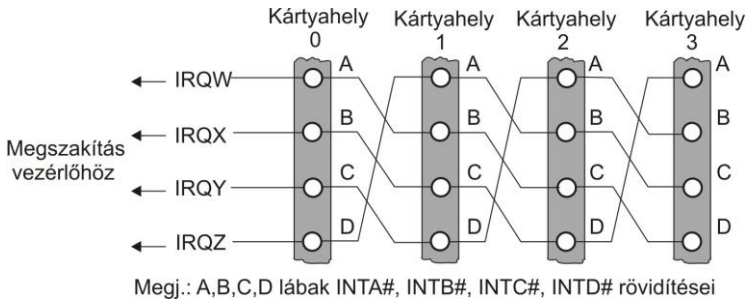


9.16 ábra – PCI arbiter felépítése[7]

Az arbitrálás algoritmusára a szabvány nem tartalmaz előírást, az „igazságos” (nem kiéheztető, pl. Round-Robin) algoritmusokat ajánlják, de bizonyos alkalmazásokban az egyszerű prioritásos arbitráció is elképzelhető lehet.

9.1.3.6 A megszakítási rendszer

A PCI rendszerben a megszakítás opcionális lehetőség. A buszon 4 jel szolgál erre: az INTA#, INTB#, INTC# és az INTD# jelek. Az egyfunkciós egyközök csak az INTA# vonalat, a többfunkciósak akár mind a négyet használhatják. A PCI megszakítások szintérzékeny, alacsony aktív szintű jelek, a megszakítást kérő eszköznek mindaddig aktív szinten kell tartania a kérést, ameddig annak nyugtázása meg nem történt. A megszakítási vonalakat O/D (open-drain) meghajtókkal kell aktiválni, így egy vonalon több egység is megszakítást kérhet.



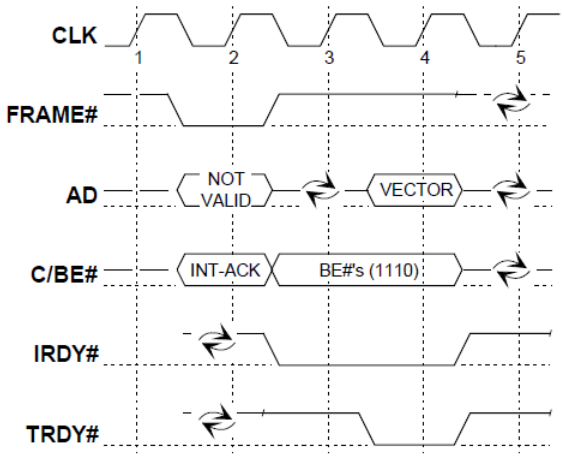
9.17 ábra – A megszakítási vonalak javasolt vezetése

[7] https://datasheet.octopart.com/COREARBITER-AR-Actel-datasheet-166046.pdf (Letöltés: 2020.04.19.)

Copyright © 2021 / BME AUT

A specifikáció nem írja elő, hogy a megszakítási vonalak hogyan csatlakozzanak a központi megszakításvezérlőhöz, de a 9.17 ábrán szereplő bekötést ajánlja. Ezzel az egyfunkciós (csak INTA#-t használó) eszközök mindegyike önálló megszakítást kap (4 eszközig egy buszon).

A megszakítások nyugtázása buszparancs segítségével történik, ilyenkor kerül továbbításra az eszköz által szolgáltatott 8 bites megszakításvektor is (9.18 ábra).



9.18 ábra – Megszakítást nyugtázó tranzakció a PCI buszon

9.2 A PCI-X busz

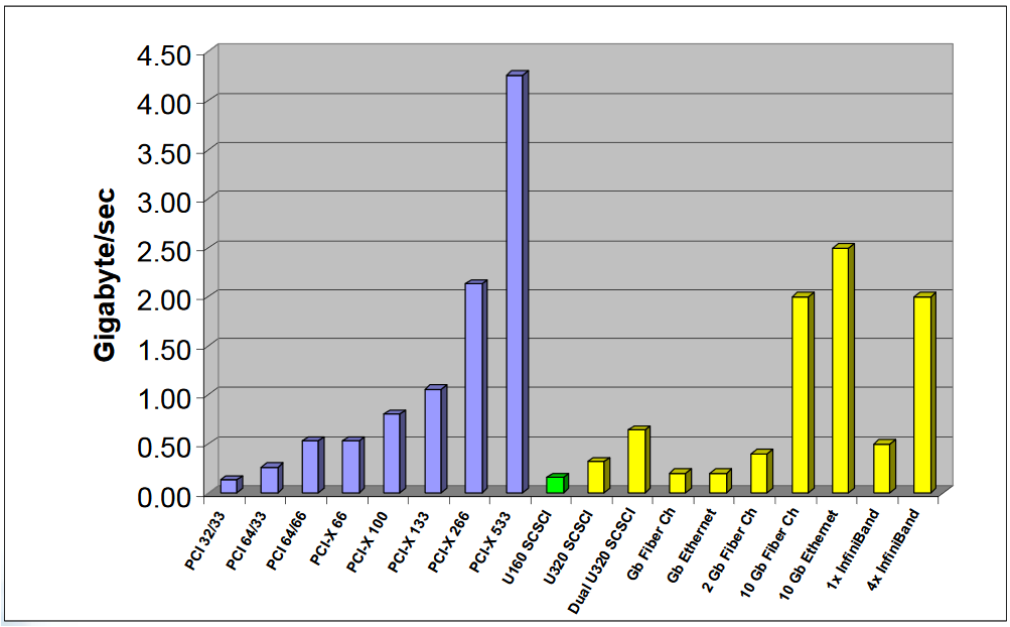
A PCI-X busz (Peripheral Component Interconnect Extended – nem keverendő össze a későbbi PCI Express busz nevében szereplő X betűvel) teljesítőképességében a PCI busz felett, de a PCI Express busz alatt helyezkedik el. A buszt az IBM, a HP és a Compaq fejlesztette ki elsősorban a nagyobb teljesítményű szervereik számára, és nyújtotta be szabványosításra 1998-ban. Azért merült fel az igény a PCI busz teljesítményét (533 MB/s) meghaladó buszra, mert a Gigabit Ethernet, a Fibre Channel és az Ultra-SCSI kártyák adatátviteli sebességigénye meghaladta a PCI által nyújtani tudott sávszélességet (9.19 ábra).

Először a PCI busz frekvenciáját duplázták meg két lépcsőben (100, majd 133 MHz), majd a buszhozzáférések megkétszerezésével (DDR = Double Data Rate), végül a megnégyszerezésével (QDR = Quad Data Rate) megnövelték a teljesítményt egészen 4,26 GB/s-ig. A busz órajele kompatibilitási okokból mindvégig 133 MHz maradt, azonban a belső dupla, majd négyszeres buszsebesség miatt elterjedt a 266 és az 533 MHz elnevezés is (mivel ilyen frekvenciával történik az adatok átvitele). A busz fejlődését és legfontosabb paramétereinek alakulását a 9.1. táblázat mutatja.

A PCI-X busz a hagyományos PCI csatlakozókat és jeleket használja, mindössze egyetlen érintkező definícióját változtatta meg (az is kompatibilis maradt a korábbi PCI eszközökkel). A B38-as csatlakozólábat – eredetileg jelföld (GND) – alakították át PCIXCAP (PCI-X képesség) jellé a 9.20 ábra szerint. A PCI vezérlő híd ellenőrzi ezt a jelet a tápfeszültség bekapcsolása után az M66EN jellel együtt annak érdekében, hogy megállapítsa a legkisebb

Copyright © 2021 / BME AUT

frekvencia-képességgel rendelkező eszközt. Ha a jel magas szinten van, akkor csak 133 MHz-es PCI-X eszközök csatlakoztak a buszra. Ha ez a jelszint valahol a magas és az alacsony szint között középtájon helyezkedik el, akkor kisebb képességű (66 MHz-es) PCI-X eszközt csatlakoztattak a buszra. Végül, ha ez a jel földpotenciálon van, akkor hagyományos PCI eszköz is található a buszon (ezeken a kártyákon ez a láb korábban is a GND felülethez csatlakozott), amely az M66EN jel állapotától függően 66 vagy 33 MHz-es is lehet. Természetesen a PCI kártya jelenléte a PCI-X buszon nagymértékben lecsökkenti a teljesítőképességet, így kerülni kell a használatát.



9.19 ábra – Sávszélesség: PCI, PCI-X és az igények[8]

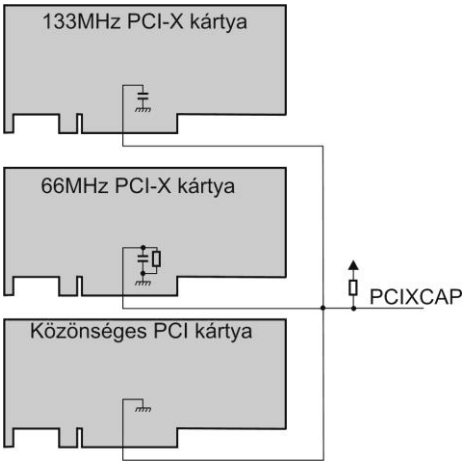
Változat	Busz-szélesség	Órajel	Tulajdonságok	Sávszé-lesség
PCI-X 66	64 bit	66 MHz	3,3V, 4 kártyahely, 256 bájt konfigurációs tartomány	533 MB/s
PCI-X 133	64 bit	100 MHz	3,3V, 2 kártyahely, 256 bájt konfigurációs tartomány	800 MB/s
PCI-X 133	64 bit	133 MHz	3,3V & 1,5V, 1 kártyahely, 256 bájt konfigurációs tartomány	1,06 GB/s
PCI-X 266	64 bit (opc. 16 bit)	133 MHz DDR	3,3V & 1,5V, ECC-támogatás*,1 kártyahely, 4 kBájt konfigurációs tartomány	2,13 GB/s
PCI-X 533	64 bit (opc. 16 bit)	133 MHz QDR	3,3V & 1,5V, ECC-támogatás*, 1 kártyahely, 4 kBájt konfigurációs tartomány	4,26 GB/s

* ECC = Error Correcting Code

9.1. táblázat – A PCI-X busz fejlődése

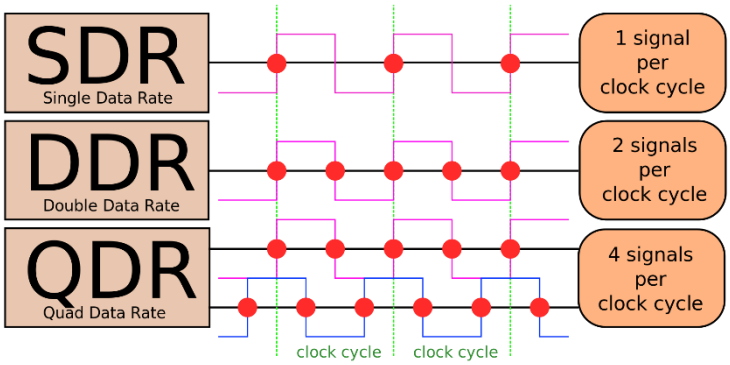
[8] https://pcisig.com/newsroom (Letöltés: 2020.04.19.)

Copyright © 2021 / BME AUT



9.20 ábra – a PCIXCAP jel az egyes kártyatípusokon

Míg a PCI busz órajelének maximális értéke 66 MHz, a PCI-X buszon ez a duplája, 133 MHz is lehet. Minden órajel felfutó élére végbemegy egy adatfázis, így ez megduplázódott adatsebességet jelent. A PCI-X 2.0-s változata a DDR és a QDR technológiák bevezetésével további kétszerezést illetve négyszerezést jelentett a sávszélességben. A technológia már korábban is ismert volt a memóriák világában (DDR SDRAM – a fejlődés folytatódik a DDR2, DDR3, DDR4 memóriákkal), a duplasebességű eszközök az órajel mindkét élét aktív adatátvitelre használják, a négyszerező egységek pedig egy belső frekvenciaduplázó áramkör segítségével előállított második, fázisában az első órajelhez (C) képest 90°-kal eltolt segédórajel (CQ) mindkét élét is felhasználják a folyamatban. (9.21 ábra).

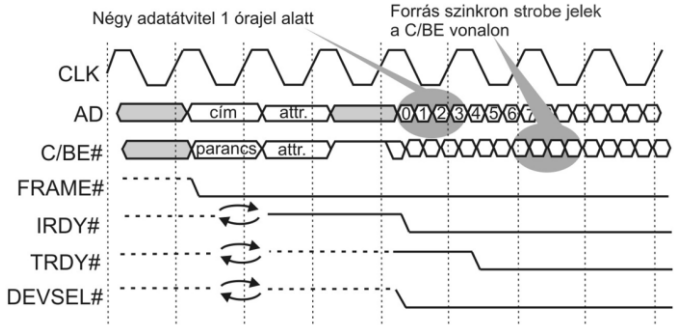


9.21 ábra – az SDR, a DDR és a QDR adatátvitel elve[9]

Az Intel minden Pentium 4 és Xeon processzoránál ezt a megoldást használja a front-side buszokon. Az már egy elvi vita tárgya lehetne, hogy ily módon mekkora órajellel is működik a rendszer – a jelváltásokból adódó szórt spektrum biztosan a négyszerese lesz az órajelenként egy élre történő SDR adatátvitelnek.

[9] https://en.wikipedia.org/wiki/Double_data_rate (Letöltés: 2020.04.22.)

Copyright © 2021 / BME AUT

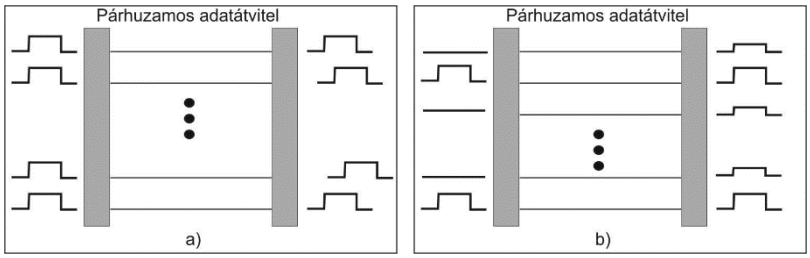


9.22 ábra – Adatátvitel a QDR technológiával

9.3 A PCI Express busz

A PCI Express busz a PCI buszcsalád legmodernebb tagja. Amíg a korábbi tagok között alapvetően evolúciós (fejlődés-jellegű) eltérések voltak, addig itt a váltás revolúciós. A párhuzamos adatátvitelt felváltotta a soros adatátvitel, ami a korábbi változatok továbbfejlődésének a legnagyobb akadálya volt. Ezt a fejlődést megfigyelhettük már az ATA-SATA interfészek világában is, mi lehet az indoka ennek a váltásnak ?

Mind a belső buszoktól, mind a külső interfészektől egyre nagyobb adatátviteli sebességet várunk el, ez az órajelfrekvencia és ehhez kapcsolódóan a jelsebesség növekedését igényli. Logikusan azt gondolhatnánk, hogy a párhuzamos adatátvitel sokkal nagyobb adatsebességet biztosít, hiszen egyidejűleg nem egy, hanem több (8, 16, 32, 64) bitet visz át. Ez a kijelentés akkor igaz, ha az összehasonlított soros és párhuzamos megoldás frekvenciája azonos. Ám a soros megoldások sokkal magasabb frekvenciával működhetnek. A PCI busz frekvenciája 66 MHz-es, a PCI-X buszé 133 MHz-es frekvenciánál „torpant meg” – innen kezdve ennek a megoldásnak a továbblépése már komolyabb jelátviteli problémákat vet fel, és csak lényegesen költségesebb megoldások jöhetnek szóba. Miért ?



9.23 ábra – Jelcsúszás (a) és áthallás (b) a párhuzamos buszokon

A párhuzamos buszokon a sebesség növelését több tényező is nehezíti, ezek közül a két legdominánsabb a jelcsúszás (angolul skew), és az áthallás (angolul crosstalk). Ezek hatását szemlélteti a 9.23 ábra. A jelcsúszás annak a következménye, hogy az egyes adatutak geometriai (fizikai jelvezetési) és elektromos paraméterei soha nem azonosak. Ennek során nem csak a buszcsatlakozók összeköttetéseit és a buszvezérlőhöz történő csatlakozásukat kell figyelembe vennünk, hanem a kártyamodulokon a csatlakozóktól a mintavételező pufferekig is

Copyright © 2021 / BME AUT

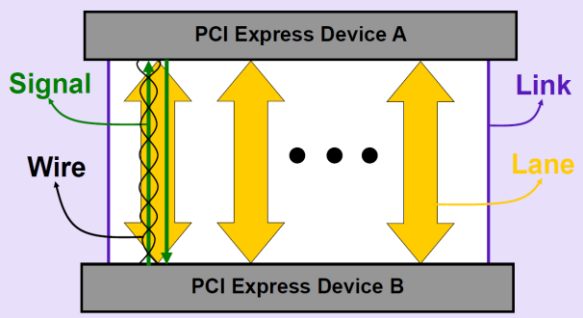
figyelnünk kell a jelutakat. Az egyes jelutak eltérő volta eltérő induktív és kapacitív viszonyokat is eredményez minden egyes jel vonatkozásában. Az áthallás pedig a párhuzamosan (és általában nagyon szorosan) futó vezetékek közötti kölcsönös induktivitás és kapacitás következménye. Az áthallás jelensége a frekvencia növekedésével nő.

Ezen elektromos problémák mellett még ne feledkezzünk meg arról a tényről sem, hogy a sok vezeték járulékos költségeket jelent: sokvezetékes kábelekre, sokpólusú csatlakozókra van szükség, bonyolultabb a nyomtatott áramköri kártyák kialakítása is. A párhuzamos buszok félduplex üzemmódban működnek („csak egy adatbusz van”), a soros buszokon párhuzamosan futó adó-vevő vonalak ezzel szemben duplex kapcsolatot biztosítanak.

9.3.1 A PCI Express rendszer architektúrája

A PCI Express nagy sebességű, pont-pont jellegű, skálázható soros busz. Szigorúan nézve nem nevezhetnénk busznak, hiszen a busz egy olyan adatút, amire egyidejűleg több eszköz is csatlakozhat, és azt megosztva használhatják. Ennek ellentmond a pont-pont kapcsolat, amely két eszközt kapcsol össze, harmadik eszköz nem osztozhat ezen a kapcsolaton. Ugyanakkor a PCI Express egy buszként kifejlődött család tagja, így a szakirodalommal összhangban a továbbiakban is busznak nevezzük. Szokásos rövidítése még a PCIe.

A busz sávszélessége skálázhatóan növelhető újabb adó-vevő kapcsolatok, úgynevezett lane-ek (sávok) hozzáadásával két eszköz között. A specifikáció x1, x2, x4, x8, x16 és x32 jelöléssel ennyi sáv kialakíthatóságát specifikálja. Mielőtt téves gondolatokra jutnánk, feltétlenül tisztázzuk: több sáv alkalmazása nem jelenti a busz „párhuzamosítását”. A sávok önálló szinkronizálással és az egymás közötti csúszás kiegyenlítésére szolgáló speciális áramkörökkel rendelkeznek (ld. később), a megoldás semmiképpen sem azonos egy párhuzamos busz felépítésével ! A pontos terminológiát mutatja a 9.24 ábra: két eszköz között kapcsolatról (link) beszélünk, ezen belül lehet több sáv (lane), egy-egy jel (signal) tipikusan két jelvezetékből (wire) áll, kábelben sodrott érpárként, nyomtatott áramkörön a lehető legszorosabban egymás mellett kialakított vezetékekként.



9.24 ábra – A link, a sáv, a jel és a vezeték fogalma[10]

[10] PCI_Express_Basics_Background (https://pcisig.com/ – Letöltés: 2020.04.18.)

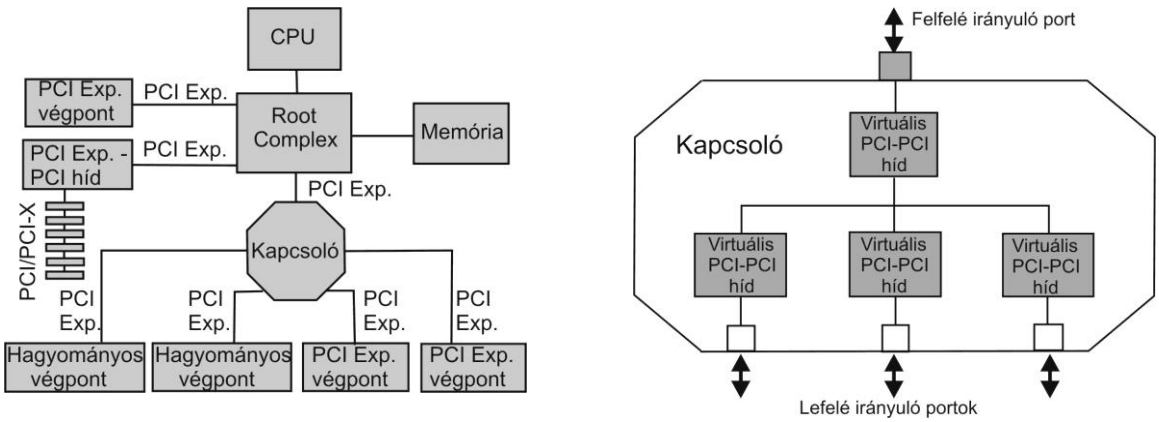
Copyright © 2021 / BME AUT

Az alapváltozat sebessége 2,5 GHz (szokjunk hozzá a GT/s, a gigatranszfer/másodperc mértékegységhez, ami az adatátvitelek számát jelöli időegységenként – az órajel nem feltétlenül kell ezzel megegyezzen, az adatmennyiség pedig függ az alkalmazott sávok számától. A sávszélességnél figyelembe kell vennünk, hogy 1 bájt a kódolás miatt nem 8 bit átvitelét jelenti a 8b/10b, majd a magasabb verziószámoknál a „takarékosabb” 128b/130b kódolás miatt – továbbá azt, hogy a tényleges adatátviteli sebesség az alkalmazott szinkronizáló és vezérlőkarakterek, az ellenőrző, és hibajavító kódok miatt még ennél is kisebb (ld. 9.2. táblázat).

PCI Express verzió	Bevezetés éve	Kódolás	Adat-sebesség	Sávszélesség				
				x1	x2	x4	x8	x16
1.0	2003	8b/10b	2.5 GT/s	0.250 GB/s	0.500 GB/s	1.00 GB/s	2.0 GB/s	4.0 GB/s
2.0	2007	8b/10b	5.0 GT/s	0.500 GB/s	1.000 GB/s	2.00 GB/s	4.0 GB/s	8.0 GB/s
3.0	2010	128b/130b	8.0 GT/s	0.985 GB/s	1.969 GB/s	3.94 GB/s	7.88 GB/s	15.75 GB/s
4.0	2017	128b/130b	16.0 GT/s	1.969 GB/s	3.938 GB/s	7.88 GB/s	15.75 GB/s	31.51 GB/s
5.0	2019	128b/130b	32.0 GT/s	3.938 GB/s	7.877 GB/s	15.75 GB/s	31.51 GB/s	63.02 GB/s
6.0 (tervezett)	2021	128b/130b	64.0 GT/s	7.877 GB/s	15.754 GB/s	31.51 GB/s	63.02 GB/s	126.03 GB/s

9.2. táblázat – A PCI Express busz fejlődése

A PCI Express busz helyének és szerepének bemutatására tekintsük a 9.25 ábrát. A vázolt szerkezet nagyon hasonlít a hálózatok szerkezetére. Eszközökből és pont-pont kapcsolatokból áll, egy Root Complexnek nevezett memóriavezérlő hub képezi a rendszer gyökerét, melyhez eszközök, kapcsolók és hidak csatlakozhatnak. A kapcsolók belül a szoftver kompatibilitás fenntartása érdekében PCI-PCI hídként működnek. A rendszerben megengedett a PCI Express-PCI/PCI-X híd is.

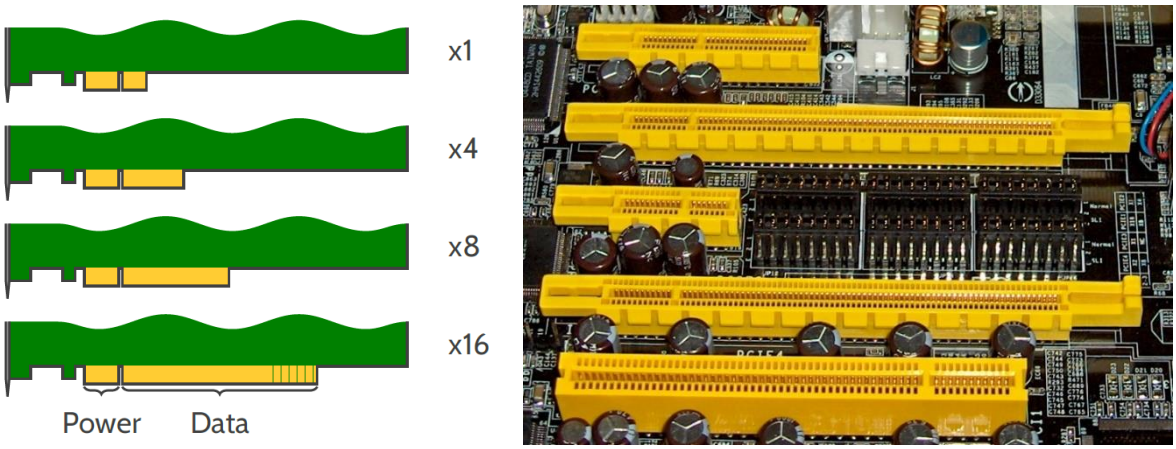


9.25 ábra – PCI Express alapú topológia és kapcsoló

A PCI Express kártyák jellegzetes, könnyen felismerhető csatlakozókkal rendelkeznek. Az egyértelmű csatlakoztatást retesz biztosítja, amely a minden kártya esetében jelenlévő fix jeleket (tápellátás, reset, ébresztés, JTAG jelek stb.) elválasztja a sávok számától függő

Copyright © 2021 / BME AUT

adatvezetékektől (9.26 ábra). A sávonként 2 adó és 2 vevő jel mindig a kártya ellentétes oldalán helyezkedik el, a túloldalon pedig árnyékoló földvezeték található a vezetékek közötti áthallás csökkentése érdekében.

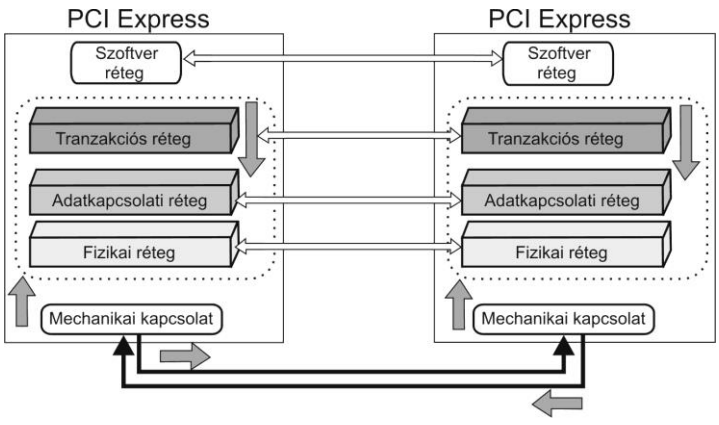


Fentről lefelé: x4, x16, x1, x16, hagyományos PCI (32 bit, 5V)

9.26 ábra – A PCI Express kártyák csatlakozói[11]

9.3.2 A PCI Express rétegszerkezete

A PCI Express busz esetében is definiálja a szabvány az OSI rétegszerkezetet. Ez lehetővé teszi a különböző funkcionális területek elszigetelését, valamint az egyes rétegek egymástól független módosíthatóságát, megújítását. Pl. egy új tranzakció bevezethető a tranzakciós (transzport) rétegben anélkül, hogy ahhoz az adatkapcsolati vagy a fizikai réteget módosítani kellene.



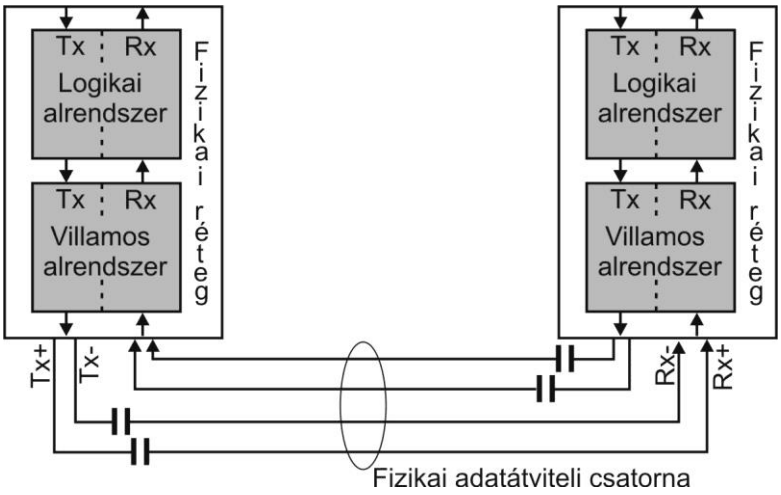
9.27 ábra – A PCI Express rétegszerkezete

[11] https://en.wikipedia.org/wiki/PCI_Express (Letöltés: 2020.04.26.)

Copyright © 2021 / BME AUT

9.3.2.1 A fizikai réteg

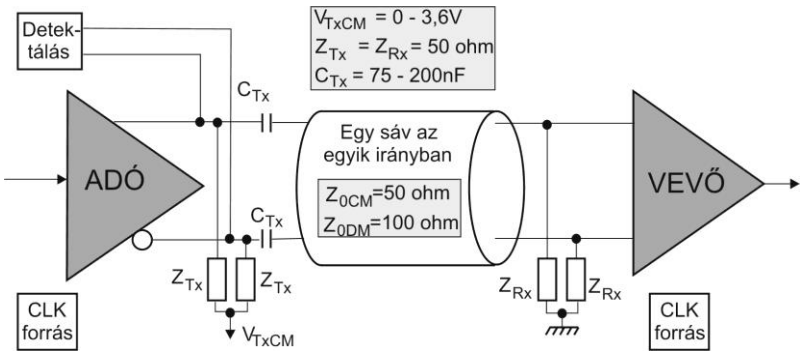
A fizikai réteg a rétegszerkezet legalsó tagja, villamos és logikai alrendszerből áll. A két eszköz egymástól csatolókondenzátorokkal van elválasztva a közös módusú feszültségek kialakulásának elkerülése érdekében. A villamos alrendszer az adó és a vevő, illetve a kapcsolódó jelek fizikai tulajdonságait írja le.



9.28 ábra – A fizikai réteg két alrendszere

Az elektromos alrendszer

Az adó és a vevőáramkörök csatlakoztatását, fontosabb fizikai paramétereit a 9.29 ábra szemlélteti. A csatorna közös módú hullámimpedanciája 50 Ω, differenciális hullámimpedanciája pedig 100 Ω. A reflexiók elkerülése érdekében a szabvány ±20% pontosságon belül írja elő a lezáró ellenállások értékét.



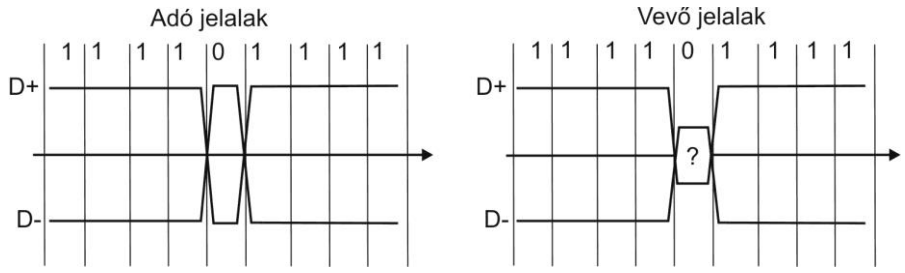
9.29 ábra – Adó és vevőáramkörök csatlakoztatása

A vevő csatlakoztatásának detektálása a következő módszeren alapszik: reset vagy tápfeszültség bekapcsolása esetén az adó stabil közös módusú feszültséggel meghajtja a vonalat. Bizonyos idő eltelte után megváltoztatja ezt a feszültséget. Ehhez a változtatáshoz a leválasztókapacitások áttöltése miatt csatlakoztatott vevőnél sokkal hosszabb időre van

Copyright © 2021 / BME AUT

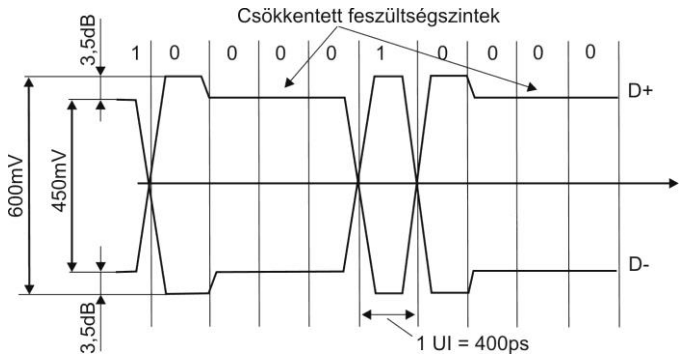
szükség, mintha nincs vevő csatlakoztatva a túloldalon. Ezt az időt mérve dönti el az adó, hogy van-e csatlakoztatott eszköz a túloldalon.

A szabvány számol a *karakterek közötti egymásra hatás* (inter-symbol interference) jelenségével. Minél nagyobb frekvencián üzemel egy adatátviteli csatorna, annál kisebb az egy bit átvitelére jutó idő, ezt az időtartamot UI-nak (unit interval) nevezzük. A csatlakozók, a hozzávezetések és a csatorna viszont bizonyos kapacitással rendelkezik, ezen kapacitások eredőjének a töltését kell rövid idő alatt az ellenkezőjére változtatni, ha az átviendő jel ellentétesre változik. Hosszabb változatlanság esetén a kapacitás jobban feltöltődik, és kevésbé lesz sikeres az „azonnali” váltás jelváltozáskor, ami akár hibás vételhez is vezethet (9.30 ábra).



9.30 ábra – Karakter-egymásrahatás

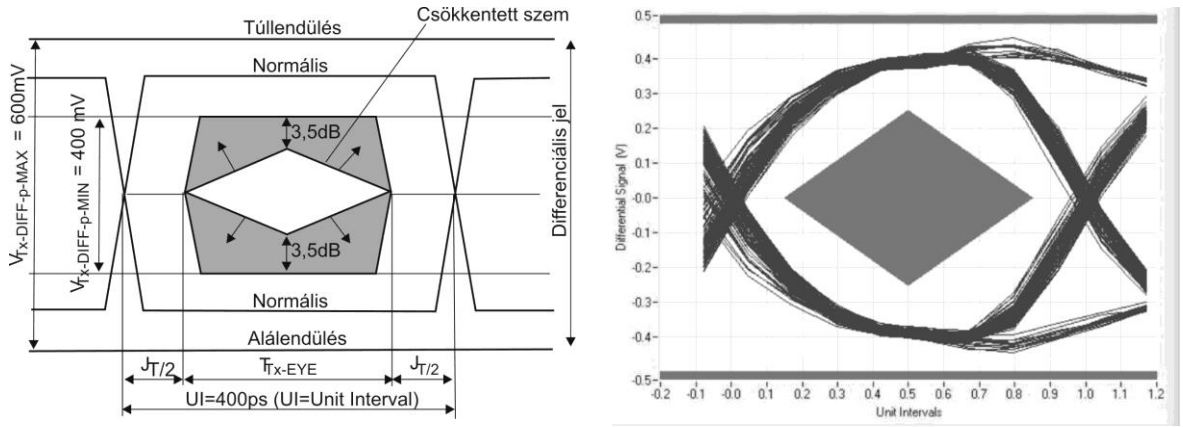
Ha több azonos bit követi egymást a jelfolyamban, akkor az első bit kivételével a többit csökkentett jelszinttel kapcsolja a csatornára az adó (alulhangsúlyozás). Hasonló hatást érünk el, ha az azonos bitek közül az elsőt megemelt szinttel, a többit normál szinttel hajtjuk meg (túlhangsúlyozás). Ezeket mutatja arányaiban a 9.31 ábra.



9.31 ábra – Túl- és alulhangsúlyozás

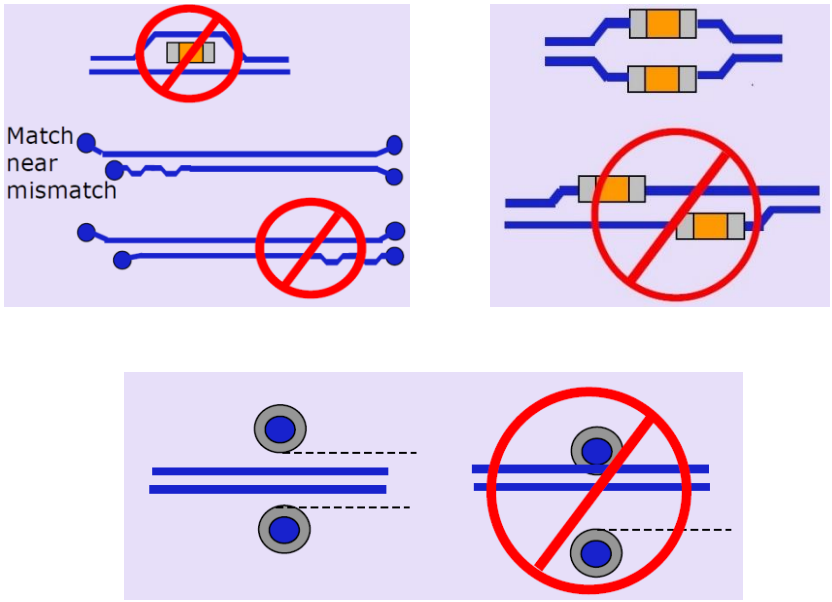
A kommunikációs csatornákon továbbított jelek definiálására és vizsgálatára is használják az ún. szemábrákat (eye diagram). Ezek segítségével jól vizsgálható, hogy a jel megfelelő jelszint és időzítési paraméterekkel rendelkezik-e, a karakter-egymásrahatás következtében fellép-e jeltorzulás. Nagy utánvilágítási idejű oszcilloszkópok segítségével tudunk sok jelet egymás után megjeleníteni, miközben a jelet az adat órajelével triggereljük. Ilyen szemábra definíciót mutat a 9.32 ábra bal oldali része, a valóságban mért értékeket pedig a jobb oldali felvétel mutatja.

Copyright © 2021 / BME AUT



9.32 ábra – Az adó szemábrája definíció szerint és a valóságban (PCIe 1.0)

Szigorú előírások vonatkoznak ilyen magas frekvenciákon a nyomtatott áramkörök vonalvezetésére és az alkatrészek elhelyezkedésére. Egy-két ilyen ajánlást (és tiltást) láthatunk a 9.33 ábrán.



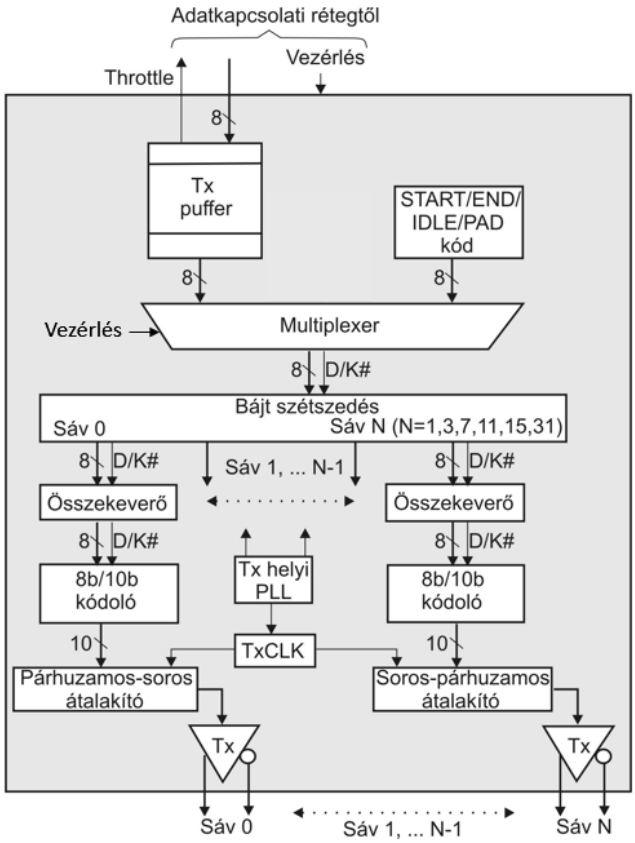
9.33 ábra – Ajánlások a nyomtatott áramkörök kialakításához[12]

[12] PCI Express 1.1 PHY Design Considerations - https://pcisig.com/newsroom (Letöltés: 2020.04.26.)

Copyright © 2021 / BME AUT

A logikai alrendszer

Az adó logikai alrendszerének blokkvázlatát a 9.34 ábrán láthatjuk.



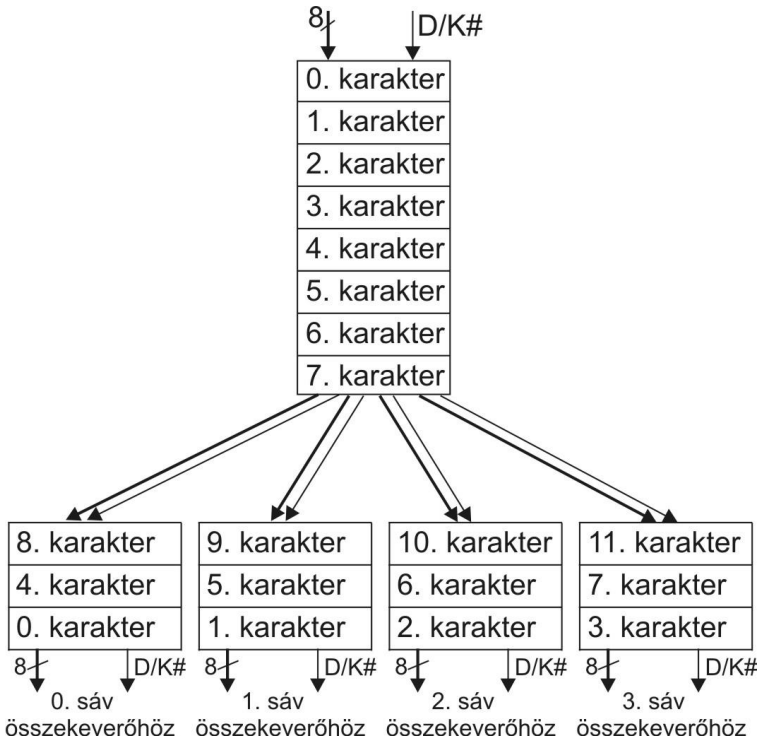
9.34 ábra – Az adó egység blokkvázlata

Az adó puffer veszi át az adatkapcsolati rétegtől a továbbítandó csomagokat. Vezérlő jel jelzi a csomagok kezdetét és végét, a throttle (lassít, zár) jel azt jelzi vissza, hogy a puffer telítődni fog, az adatátvételt lassítani kell. Az adó pufferben még alapvetően adatbájtok (kódolás után majd Dx.y karakterek) találhatók.

Ezek elé és mögé kerülnek a multiplexer segítségével azok a vezérlő karakterek (Kx.y), melyek nem téveszthetők össze az adatokkal és egyértelműen behatárolják a csomagokat (STP/SDP/END – ld. később).

Ha egy link több sávon is kapcsolódik a túloldalhoz, akkor következik a bájt-szétszedés (byte stripping) folyamata. Ez a csomagban lévő bájtokat sorban egymás után az egymást követő sávoknak adja át, a bájt-szétszedő után annyi párhuzamos ágon haladunk tovább, ahány sáv található az adott kapcsolatban. A bájt-szétszedőnek kezelnie kell, hogy a csomagolás során már a láncba került vezérlő karaktereket (10 bit) azonos módon kezelje az adatokkal (8 bit), melyek a kódolás során szintén át lesznek alakítva 10 bites adatkarakterekké (a K karakterek már nem). A bájt-szétszedés folyamatát a 9.35 ábra mutatja.

Copyright © 2021 / BME AUT

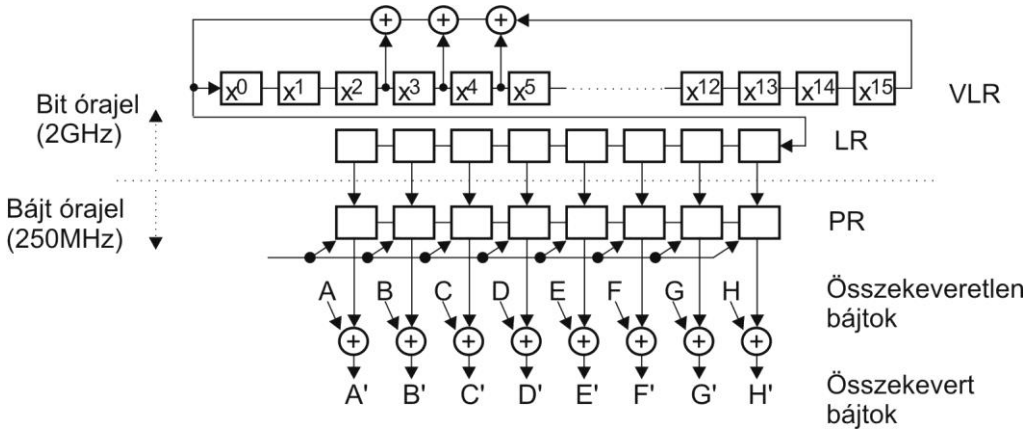


9.35 ábra – A bájt-szétszedés folyamata

Az egyes sávokon a következő lépés az összekeverés (scrambling). Ennek fontosságával már a SATA interfésznél foglalkoztunk. A szabvány az összekeveréshez a

$$G(x) = x^{16}+x^5+x^4+x^3+1 \qquad (9.1)$$

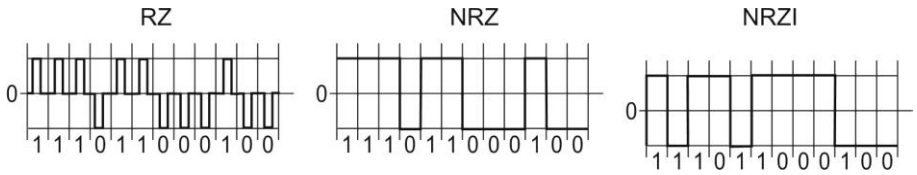
generátor polinomot írja elő. Ennek egy lehetséges megvalósítása a 9.36 ábrán látható.



9.36 ábra – Az összekeverő egység sávonként

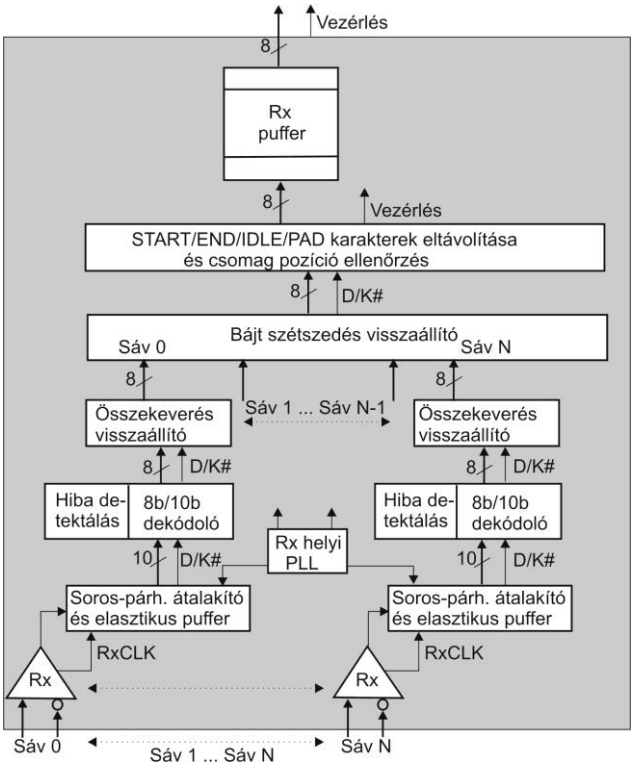
Copyright © 2021 / BME AUT

Innen kezdve már csak a 8b/10b kódolás van hátra (a vezérlő karakterek természetesen nem kerülnek kódolásra !), majd a differenciális adómeghajtóhoz értünk. A PCI Express adóegysége a vonalat normál NRZ (Non Return to Zero) kódolással továbbítja. NRZ kódolásnál szokásos módon az 1-esek képviselik az egyik, a 0-k a másik jelszintet, ez a megszokott hozzárendelés nem keverendő össze sem az RZ, sem az NRZI kódolással, melyek az NRZ-től eltérő eredményeket szolgáltatnak (9.37 ábra).



9.37 ábra – Az RZ, NRZ és az NRZI kódolások

Vételi oldalon sem egyszerűbb a helyzet a fent elmondottaknál – sőt. A vevőnek valamennyi átalakítási folyamatot visszafelé végig kell csinálni, de ezen felül még az ő feladata a szinkronizáció megoldása is. A vevő blokkvázlata a 9.38 ábrán látható.

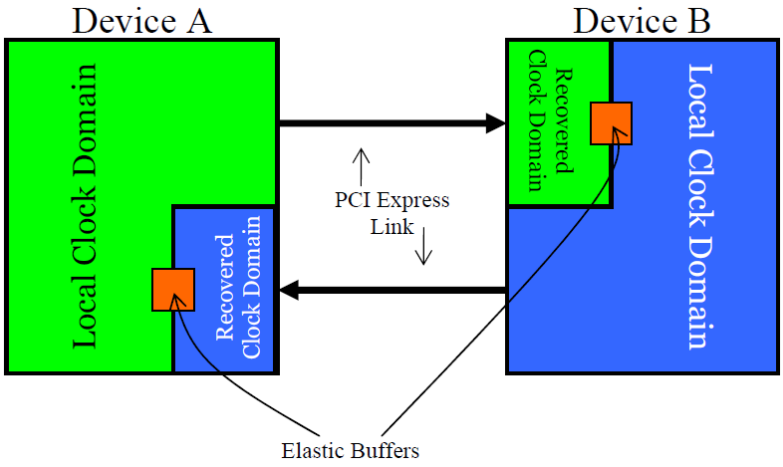


9.38 ábra – A vevő egység blokkvázlata

Aszinkron rendszer lévén az adó és a vevő soha nem működnek pontosan megegyező órajellel. Előfordulhat, hogy kismértékben az adó „siet” (magasabb az órajelfrekvenciája), de az is lehet, hogy „késik” (akkor viszont alacsonyabb). Az adatok ráadásul több sávon indulhatnak el az adó felől, az egyes sávok között útközben időeltérések keletkezhetnek (jelcsúszás). Amikor a vevő

Copyright © 2021 / BME AUT

fogadja a beérkező adatfolyamot, neki a beérkező élekhez igazodva – akár sávonként eltérő módon – szinkronizálódnia kell az adott sáv adójának órajeléhez (ezt nevezzük forráshoz szinkronizált időzítésnek az aszinkron átvitelek között). A szinkronizálódás lehetőségét az teremti meg, hogy a 8b/10b kódolás révén megfelelő sűrűséggel garantált a jelváltások megléte a vett jelben. Ily módon vételi oldalon keletkezik egy olyan „sziget” (duplex kapcsolat esetén mindkét oldalon), amely időzítését tekintve aszinkron működik az egység többi logikájától. Ezen két logika között képeznek átmenetet az elasztikus bufferek (ld. 9.39 ábra).



9.39 ábra – Elasztikus bufferek elhelyezkedése forráshoz szinkronizált időzítésnél

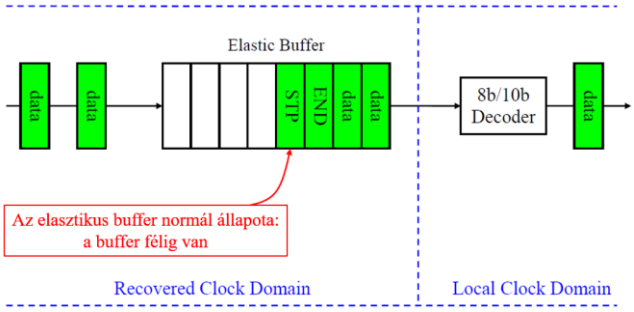
Amennyiben az adó lassabban ad, mint ahogyan a karaktereket a vevő logika (saját órajele szerint) fogadná, nincs nagyobb baj, csak „ki kell várni” esetleg órajel kihagyással a következő karakter megérkezését. Ha viszont az adó „siet” a vevő időzítéséhez képest, akkor torlódás alakulna ki a két logika határán – és ilyenkor karakter veszhetne el. Nyilvánvalóan ez ellen tennünk kell valamit.

A specifikáció szerint az órajelek értéke max. ±300 ppm-mel térhet el a névleges értéktől, ez azt jelenti, hogy szélsőséges esetben pl.

$$f_{adó} = f_{névl} + 300 \text{ ppm}, f_{vevő} = f_{névl} - 300 \text{ ppm}, \text{tehát } \Delta f = 600 \text{ ppm}$$

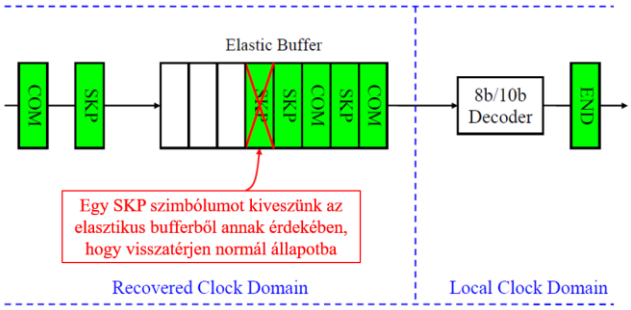
Ez az adó és a vevő lehetséges eltérésének a maximuma is egyben, legrosszabb esetben $N = 1/(600 \cdot 10^{-6}) = 1\,667$ bitenként pontosan 1 bitnyi elcsúszás lehet az adó és a vevő között (lévén arányokból indultunk, mindez igaz tetszőleges adategységek, így pl. a 10 bites karakterek szintjén is: 1 667 karakterenként maximum 1 karakternyi csúszás léphet fel az adó bájt-szétszedő és a vevő bájt-összerakó áramkörei között). Mint már említettük, a nagyobb problémát az okozza, hogy a nagyobb sebességű adótól a karakterek gyorsabban érkeznek, mint ahogy azt a vevő bájt-összerakója el tudná szedni – emiatt torlódnak az adatok a két egység határán. Puffert kell tehát építenünk ezen adatok számára. Az adó minden 1180..1538 karakter után küld egy szabadon rövidíthető/törölhető vezérlőkarakter-sorozatot (COM, SKIP, SKIP, SKIP = K28.5, K28.0, K28.0, K28.0), aminek az a célja, hogy a vevőnek legyen mit „eldobnia”, ha torlódás lépne fel az előbb említett átmeneti pufferben.

Copyright © 2021 / BME AUT

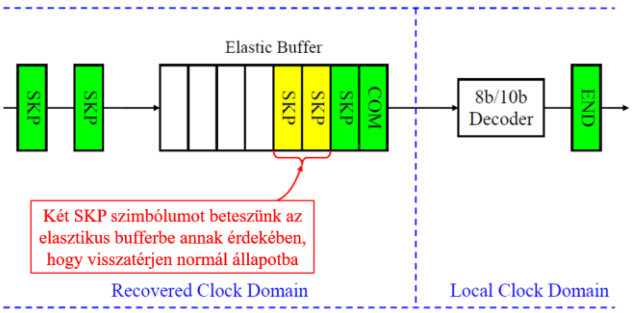


9.40 ábra – Elasztikus buffer normál állapota

Az elasztikus buffer vezérlését úgy alakítjuk ki, hogy folyamatosan félig telt állapotban tartsa ezt a FIFO jellegű tárolót. Normál állapotot és adatfolyamot mutat a 9.40 ábra. A puffer félig van, éppen két adatcsomag határán (END: az előző csomag vége, STP: a következő csomag kezdete – a tranzakciós csomagok felépítését ld. később).



9.41 ábra – Adósebesség magasabb: karakter kivétele a bufferből



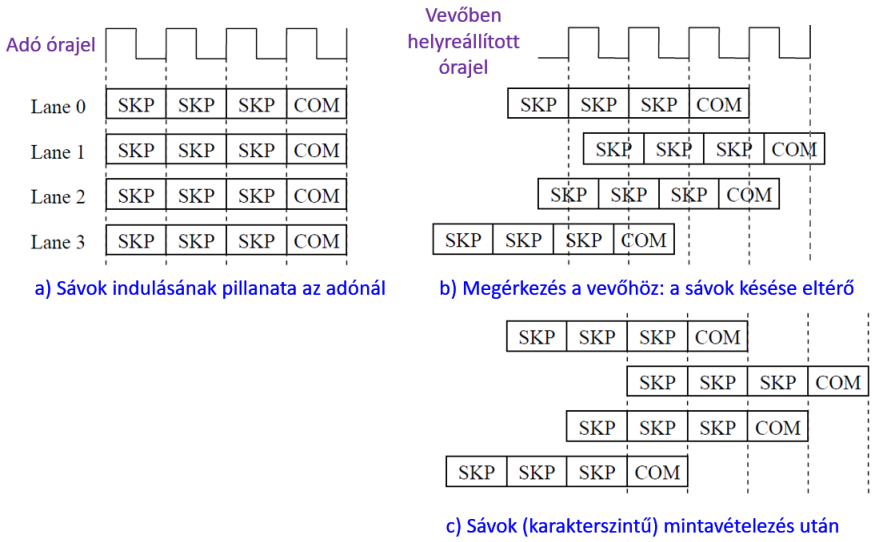
9.42 ábra – Adósebesség kisebb: karakterek beszúrása a bufferbe

A 9.41 ábra és a 9.42 ábra már eltérő adó és vevő sebességek esetén mutatja az elasztikus buffer működését. A SKIP karakterek csak vezérlési feladatot látnak el, így szabadon kivehetők vagy beszúrhatók a bufferbe annak érdekében, hogy azt normál állapotban (félig telve) tartsuk. A vezérlő karaktereket a vevőoldali logika el fogja távolítani az adatok közül, így azok semmi kárt nem fognak okozni az adatfolyamban.

Copyright © 2021 / BME AUT

A helyzetet tovább bonyolítja, ha az adó és a vevő között nem egyetlen adatkapcsolati sáv (lane) található, hanem több. Ilyet már láthattunk az USB 3.2 és az USB 4 szabványokban is, még hangsúlyosabban jelentkezi ez a PCI Express interfész esetében, ahol akár 32 párhuzamos sáv is kiépíthető. Minden sávon szükséges egy adott idő, ameddig a jel az adótól a vevőig ér, viszont az egyes sávokon ez a késleltetés eltérő lehet, azaz az 1. sáv Δt_1, a 2. sáv Δt_2 stb. idővel tér el (késik) az ideális állapothoz képest. Mindez nem is lenne baj, de ily módon a vevő egyes sávjai már külön-külön is aszinkron futnak majd egymásról, márpedig valamennyi független sáv végén el fogunk jutni a bájt-szétszedő vevő oldali ellenpárjához (nevezzük bájt-összerakónak), amely már a vevő logika egyetlen közös órajelével kell kezelje az egyes sávokról érkező adatinformációkat.

Az adó sávok órajele közös, így az indítási időpont mindig közel azonos, de az elcsúszások időben ingadozni fognak. Az adó minden sávban minden 1180..1538 karakter után minden sávban küldi a szabadon rövidíthető/törölhető vezérlőkarakter-sorozatot (COM, SKIP, SKIP, SKIP), ezek azonban eltérő időpontokban érkeznek be a vevő (adóhoz szinkronizálódó) blokkjaiba. A fizikai vevők után először következik a soros-párhuzamos átalakítás, majd a 9.43 ábra mutatja az indulás, az érkezés és a karakterszintű mintavételezés után látható állapotot az egyes sávokban (szándékosan a beszúrt vezérlőkarakter-sorozatot ábrázolva).



9.43 ábra – Sáv-elcsúszás (lane skew)

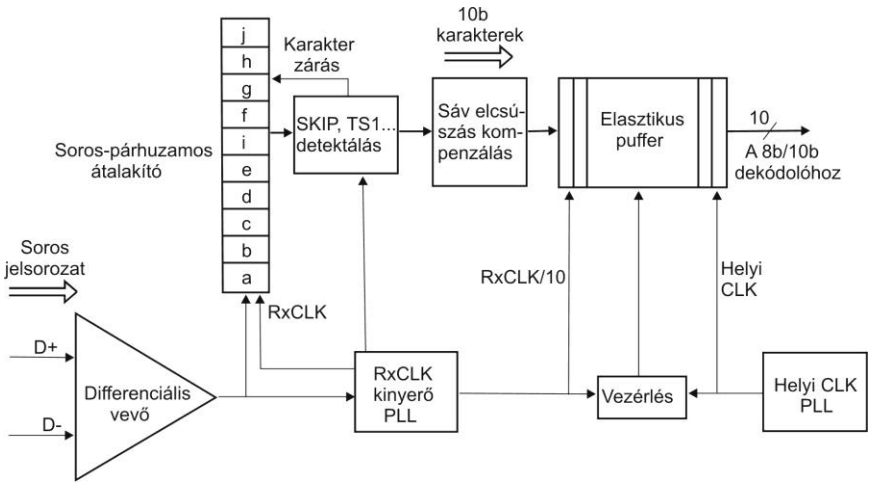
Természetesen vevőoldalon a bitszintű jelváltások alapján történik az órajel visszanyerése, de a soros-párhuzamos átalakítás után a bitcsoportok (karakterek) egymáshoz képesti elcsúszása megmarad mindaddig, amíg a sávok szinkronizálása meg nem történik.

Annak érzékeltetésére, hogy a csúszások hogyan viszonyulnak a bit- ill. karakteridőkhöz, néhány adat a szabványból (PCIe 2.0 – 5 GT/s, UI = 200 ps):

- Tx oldalon 2 sáv közötti maximális csúszás: 500 ps + 4 UI (= 6,5 UI = 1300 ps)
- Rx oldalon 2 sáv közötti maximális csúszás: 8 ns (= 40 UI = 4 karakter)

Copyright © 2021 / BME AUT

Nézzük kicsit részletesebben a vevő blokkvázlatában a 8b/10b dekódoló előtti logikai egység („soros-párhuzamos átalakító és elasztikus puffer”) részletezését (9.44 ábra).



9.44 ábra – A vevő bement logikája

A vevőnek fel kell ismernie a 10 bites karakterhatárokat a soros-párhuzamos átalakítás után. A felismerést a COM (komma, K28.5) karakterek segítik, melyek semmilyen más adattal vagy vezérlőkarakterrel sem összetéveszthetők és kizárólag ezt a funkciót látják el. Azt már rögzítettük, hogy a vezérlőkarakterek bitjei soha nem kerülnek összekeverésre (scrambling), így az adott bitsorozat a vett adatok között egyértelműen felismerhető. Ha a vevő érzékel egy COM karaktert, biztos lehet abban, hogy a soros-párhuzamos átalakító karakterhatárhoz ért, azaz alaphelyzetbe állítható. Ezt az állapotot hívjuk karakterzárásnak.

Többsávos rendszerekben az egyes sávok egymáshoz képest is elcsúsznak. Az elcsúszás mértéke nem igazodik sem a bitidőhöz, sem a karakterek léptetésének idejéhez (=10 bitidő), a 9.43 ábra jobb felső részén látható véletlenszerűen elcsúszott karakterek a bájt-összerakás előtt már a vevő logika minden sávra közös órajelével mintavételezve kezdenek „szinkronizálódni” (ld. az ábra alsó részét). Ha ezeket az 1-2 karakternyi csúszásokat nem kompenzálná a vevő, akkor a bájt-összerakás nem működne helyesen. A karakterek az ún. elasztikus pufferekbe kerülnek (minden sávban), ahol az adó által beszúrt – egyébként felesleges – SKIP karakterek törlésével az elasztikus puffer a sávokat „ki tudja egyenlíteni”, és innen kezdve már teljes szinkronban történhet a sávokon a 8b/10b dekódolás majd a bájt-összerakás folyamatának az elvégzése.

Az adatok tranzakciósréteg-csomagokban (TLP, ld. később) kerülnek továbbításra, a fent említett szinkronizálásra a csomagok között van lehetősége az adónak és a vevőnek. Kérdés: egy adatcsomag átvitele alatt maximálisan hány karakternyi csúszás léphet fel az adó és a vevő között, azaz hány „eldobható karaktert” kell beépíteni az adónak ahhoz, hogy az átvitel után szinkronban tudjon maradni a két egység ? (Arra gondoljunk, hogy a gyorsabb adó „feltölti” a lassabb vevő pufferét, és egy idő után a beérkező karaktereket nem lenne hova tenni.) Ez nyilvánvalóan függ az átviendő adatok mennyiségétől a (9.2) összefüggés szerint:

Copyright © 2021 / BME AUT

$$Max\_SKIP = \frac{\frac{Adathossz + Overhead}{Sávok\ száma} + 1538}{1667} \tag{9.2}$$

Az elasztikus bufferben annyi tartalék kell legyen található, amennyi legrosszabb esetben az adó-vevő szinkron fenntartásához szükséges (a legrosszabb órajel-arány esetén is). A szabvány előírása szerint az adó 1180..1538 karakterenként kíséreli meg a beszúrást, de ha pont a felső határon elkezdődik egy adatcsomag (TLP) továbbítása, akkor meg kell várnia annak a végét. A számlálóban így az 1538-hoz hozzá kell adnunk a „kivárandó" karakterek maximális számát, ami az adatblokk hossza + a TLP csomag felépítéséhez szükséges járulékos karakterek száma (overhead), ha az átvitel több sávon történik, akkor ezt oszthatjuk a párhuzamos sávok számával. Ezt kell viszonyítanunk a max. 1 karakter csúszást jelentő 1667 karakternyi időhöz.

Egy egyszerű példa: 4096 bájtos adatcsomag és 28 bájtnyi „csomagolás" (1 bájt START, 2 bájt sorszám, 16 bájt header, 4 bájt ECRC, 4 bájt LCRC, 1 bájt END – jelentésüket ld. a következő alfejezetben) és 1 sáv esetén *Max_SKIP = 3.4*, azaz 4 karakter beszúrásával a csúszás kiegyenlíthető, ez indokolja a 4 karakteres SKIP karaktercsoportok használatát. Természetesen a beszúrandó karakterek száma függ a választott adatcsomag hosszától (ld. 9.3. táblázat):

Adatcsomag hossza [bájt]	128	256	512	1024	2048	4096
Beszúrandó karakterek száma	1.02	1.09	1.25	1.55	2.17	3.4

9.3. táblázat – SKIP karakterek száma az adatcsomag hosszától függően

A fizikai réteg az adatkapcsolati rétegtől kapott csomagok átvitelébe csak akkor avatkozik bele, ha a fizikai kapcsolat valamilyen hangolást, kiegészítést igényel. Erről a kényszerről a fizikai réteg feletti rétegek nem tudnak és nem is kell tudjanak. Ezek a beavatkozások az ún. rendezett csoportok (ordered set) segítségével történnek, ilyen pl. a már megismert késleltető (SKIP) funkció.

A rendezett csoportok nagyon rövid áttekintése a következő:

- **1-es tréningsorozat (TS1)**: Egy COM és tizenöt TS1 karakterből áll, lekérdezéskor, konfiguráláskor, hibaállapot megszüntetésekor használja a rendszer (szinkronizálásra);
- **2-es tréningsorozat (TS2)**: Egy COM és tizenöt TS2 karakter, szerepe hasonló TS1-hez;
- **SKIP**: Egy COM és három SKIP karakterből áll – a frekvenciakülönbség és a jelcsúszás kiegyenlítéséhez használja az adó és a vevő;
- **Tétlen állapot (IDLE)**: Egy COM és három IDL karakter. Ha nincs elküldhető adat, ilyen csomagok küldésével tartják fent adó és vevő között a szinkronizált állapotot;
- **Gyors tréningsorozat (FTS)**: Egy COM és három FTS karakterből áll, gyorsabb változata a TS1 és a TS2 tréningsorozatoknak.

Copyright © 2021 / BME AUT

9.3.2.2 A tranzakciós és az adatkapcsolati rétegek

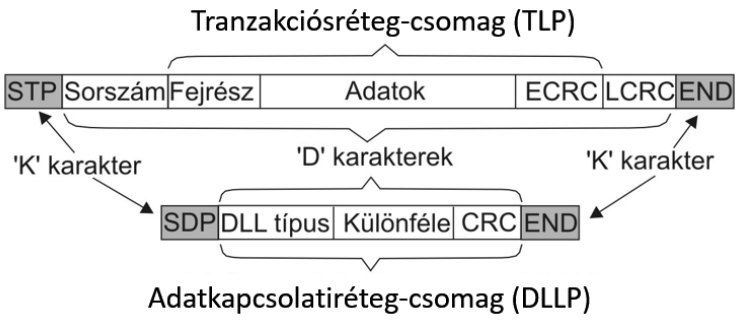
Az OSI rétegszerkezet hatalmas előnye, hogy a fizikai szintről feljebb lépve elfelejthetjük mindazt a bonyolult elektronikai-logikai rendszert, amely elvégzi az adatok hibátlan átvitelét, adó és vevő szinkronizálását, a sebességkülönbségük kiegyenlítését. Ettől a rétegtől kezdve már arra koncentrálhatunk, hogy iniciátor és target milyen adatokat kívánnak cserélni egymással.

A PCI Express rendszert a SATA rendszerhez hasonlóan olyanra alakították ki, hogy a korábbi PCI rendszerek vezérlő szoftverei lehetőleg ne vegyenek semmit észre az átalakulásból, a PCI vagy a PCI-X rendszer lecserélődéséből PCI Express rendszerre. Biztosították a korábbi szoftverrendszerekkel való kompatibilitást – ami legyünk őszinték – tovább növelte a PCI Express interfész komplexitását.

A PCI Express csatornáin keresztül minden információ csomagokban kerül továbbításra. Ez alól kivételt csak a fizikai réteg rendezett csoportjai képeznek (ezeket kvázicsomagoknak nevezzük, korábban már tárgyaltuk őket). A csomagok tartalmukat tekintve memória, I/O, valamint konfigurációs írási/olvasási tranzakciókat, továbbá üzenet tranzakciókat közvetítenek, valósítanak meg. A csomagok két fő osztálya:

- tranzakciósréteg-csomag (TLP) – ezek valójában az adatcsomagok;
- adatkapcsolatiréteg-csomag (DLLP) – vezérlési, ellenőrzési, ismétlési funkciók.

A tranzakciós réteg és az adatkapcsolati rétegek szintjén az adatok bájt (=8 bit), szó (=16 bit) vagy duplaszó (=32 bit) egységekben kezelődnek. A fizikai rétegben a bájtok 8b/10b kódolást és a már említett vezérlőkarakterek beszúrását követően már 10 bites vezérlő (K) és adat (D) szimbólumokból állnak. Minden csomag elejére és végére a fizikai rétegben csomagkezdet-jelző és csomagvég-jelző karaktereket helyezünk el a csomagok határainak egyértelmű felismerhetősége érdekében (Csomagkezdet-jelzők: STP = tranzakciósréteg-csomag kezdete, SDP = adatkapcsolatiréteg-csomag kezdete, csomagvég-jelző: END). Az így kiegészített csomagok felépítését mutatja a 9.45 ábra.



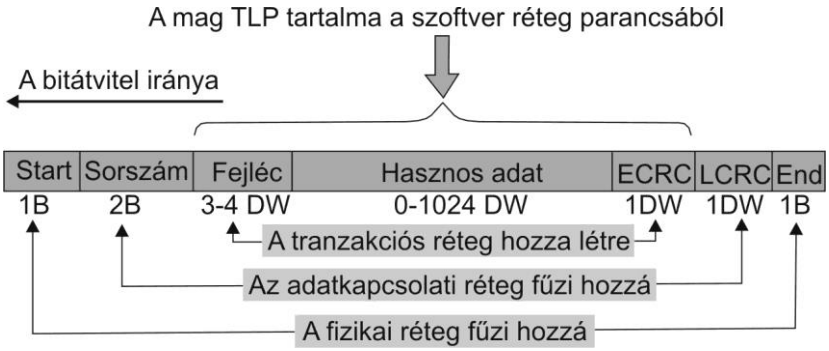
9.45 ábra – Transzport és adatkapcsolati réteg csomagok START és END karakterekkel

A két réteg feladatait közösen tekintjük át a teljesség igénye nélkül. Az egyes fejlécek, csomag- és mezőelemek részletes ismertetése a PCI Express szabványban megtalálható.

Copyright © 2021 / BME AUT

Tranzakciósréteg-csomag

A hasznos adat áramlását a PCI Express egyes rétegei között a 9.46 ábra mutatja. Az adó egységben a tranzakciós réteg a szoftverrétegtől kapott kérelmet egy tranzakciósréteg-csomaggá alakítja. Ez ebben a rétegben még csak egy TLP fejlécből, az azt követő adatokból és opcionálisan egy End-to-End CRC (ECRC) ellenőrző kódból áll. Ezt a csomagot a lejjebb lévő rétegek járulékos információkkal egészítik ki. Az adatkapcsolati réteg elöl sorszámot, hátul egy ellenőrző kódot (LCRC) fűz a csomaghoz, amelyet végül a fizikai réteg START és END karakterekkel keretez be. A fizikai csatornán már végül ez a csomag kerül továbbításra (B=bájt=8 bit, DW=duplaszó=32 bit).



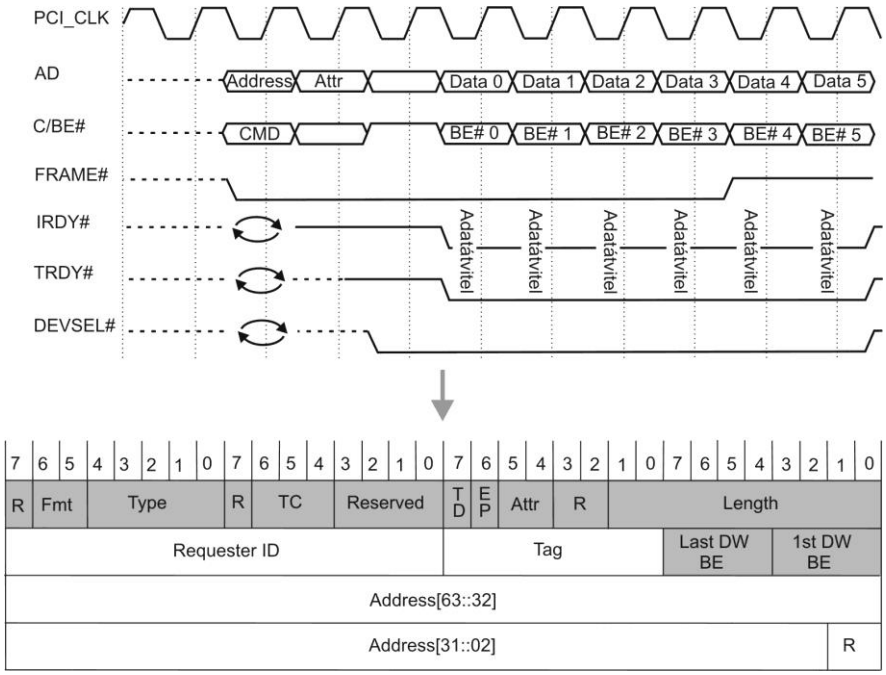
9.46 ábra – A teljes TLP struktúra

A TLP protokoll egyes lépései a következők:

1) A szoftver a tranzakciós kérelmet a PCI Express hardverinterfészének adja át. A szabvány nem specifikálja ennek a pontos módszerét, ezért ez eszközfüggő lehet. A kérelem általában a következő információkat tartalmazza:
    a. a végrehajtandó PCI Express parancs, a target címe vagy ID-je;
    b. a tranzakció típusa (memória olvasás/írás, I/O olvasás/írás, konfigurációs ciklus stb.);
    c. az átviendő küldemény hossza, és az adatok (ha vannak);
    d. virtuális csatorna, forgalomosztály információ;
    e. az átvitel attribútumai.
2) A fenti információ alapján az adó tranzakciós rétege felépíti a TLP fejlécet (ld. 9.47 ábra), a küldeménytartalmat és az opcionális toldalékot.
3) Az adatkapcsolati réteg a TLP-t sorszámmal és Link CRC-vel egészíti ki.
4) A fizikai rétegben megtörténik a bájt-szétszedés, az összekeverés, kódolás. A csomag eleje és vége egy STP és egy END karakterrel egészül ki.
5) A vevőbe megérkezett csomagot a vevő fizikai rétege visszaalakítja párhuzamos karakterekké, majd a csomagot átadja az adatkapcsolati rétegnek.

Copyright © 2021 / BME AUT

6) A vevő adatkapcsolati rétege CRC-t számol, összehasonlítja a számított és a vett értékeket. Ellenőrzi a csomag sorszámát. Ha nincs hiba, átadja a csomagot a tranzakciós rétegnek, az pedig dekódolva a csomagot továbbítja azt a szoftverrétegnek. Ezen kívül a vevő adatkapcsolati rétege egy DLLP ACK vagy NAK csomaggal értesíti az adó adatkapcsolati rétegét a hibátlan vagy hibás vételről. NAK esetén az adó megismétli az adást.



9.47 ábra – A TLP fejléc felépítése – összevetve a PCI tranzakció elemeivel

A mezők jelentése a TLP fejlécben az egyszerűbb érthetőség kedvéért (csupán áttekintés jelleggel):

- **Length**: a küldemény hossza duplaszavakban (0..1024 DW = 4 kB);
- **Attr**: PCI/PCI-X fogalmak, nem részletezzük;
- **EP**: átvitt adat hibás;
- **TD**: opcionális toldalék az adatok után;
- **TC**: forgalomosztály;
- **Type**: a tranzakció típusa (memória, I/O, konfiguráció, írás/olvasás);
- **Fmt** (Format): a fejléc hossza duplaszavakban;
- **1st DW BE**: bájtszintű adatmegadás – első bájtban;
- **Last DW BE**: bájtszintű adatmegadás – első bájtban (közbenső DW-k nem bonthatók);
- **Tag**: elküldött, de még nem teljesített kérelem azonosító;
- **Requester ID**: egyértelműen azonosítja a kérelmezőt;
- **Address**: az átviendő adatok kezdőcíme.

Copyright © 2021 / BME AUT

10 Korszerű soros buszok kapcsolódása

Az előző fejezetekben áttekintettük, mekkora fejlődésen mentek keresztül a személyi számítógépek külső és belső interfészei. Láthattuk, hogy mind a belső, mind a külső kapcsolatok napjainkra soros adatátvitelekké váltak, megadtuk ennek indokait is. A sok különféle egyedi célokra fejlesztett interfészkapcsolat mára néhány meghatározó résztvevőre szűkült, ezek igen nagy elterjedtséget és népszerűséget mondhatnak maguknak. A „néhány" (és nem egyetlen) indoklására két fontos szempontot hozunk fel:

- Az informatikai eszközök piaca hatalmas. Több „mamutcég" fejleszti és gyártja a számítógépek processzorait, fontosabb egységeit (háttértárak, megjelenítők, egyéb perifériák) és magukat a komplett számítógépeket (desktopok, laptopok, szerverek, munkaállomások, routerek stb.), de ma már nyugodtan ide vehetjük a személyi számítógépek teljesítőképességével versenyző mobileszközöket is. A vezető óriások sok esetben nem hajlandók engedni saját elképzeléseikből, piacpolitikai okokból nem kívánják alkalmazni a konkurens cég esetleg hasonló célokra fejlesztett megoldásait.
- A jelenleg még ténylegesen létező külső interfészek többnyire eltérő célokat valósítanak meg és szolgálnak ki (perifériák kapcsolatait, hálózati célokat, megjelenítési célú video- és audiokapcsolatot, lokális eszközök kapcsolatait), ennek megfelelően sok esetben nem is összevonhatók, nem tudnák egymást helyettesíteni.

Fentiek ellenére megfigyelhető az a törekvés, hogy a legnagyobb gyártók az interfészük fizikai tulajdonságaiban (felépítés, sebesség, jelszintek stb.) alapvetően hasonló kapcsolatokat kezdik oly módon összevonni, hogy azonos fizikai réteg használata mellett a belső logikai és szoftver rétegek egymás mellé integrálásával képessé tesznek egy-egy interfészt akár több fentebb felsorolt feladat ellátására is – csökkentve ezzel a szükséges csatlakozási pontok, kábelek, csatlakozók számát. Ily módon egy-egy interfésztípus több helyen is használhatóvá válik, ami növeli az azonos alkalmazások számát, nagyobb számban alkalmazhatóvá teszi a komplex interfészvezérlőket, csatlakozókat, az előre konfekcionált interfészkábeleket – csökkentve ezzel az árukat, ami az eladhatóságukat javítja és ezzel az eladási számokat még magasabbra emeli.

Az előző fejezetekben megismerkedtünk részletesebben már az USB és a PCI Express interfészekkel. Ehhez még néhány, ma elterjedten használt interfésztípust csak felszínesen megemlítünk, majd áttekintjük ezek jelenleg létező kapcsolódásait.

10.1 A DisplayPort interfész

A DisplayPort interfészt megjelenítőeszközöknek (monitor, TV) személyi számítógépekhez történő csatlakozására fejlesztették ki, nagysebességű soros interfész, melyet a VESA (Video Electronics Standards Association) gondoz és fejleszt mind a mai napig. Jogdíjmentesen felhasználható bármely eszközön, ami hatalmas előny a hasonló céllal létrejött interfészekkel (pl. HDMI) szemben. Azzal a nem titkolt céllal hozták létre, hogy a már akkor is igen korosodó

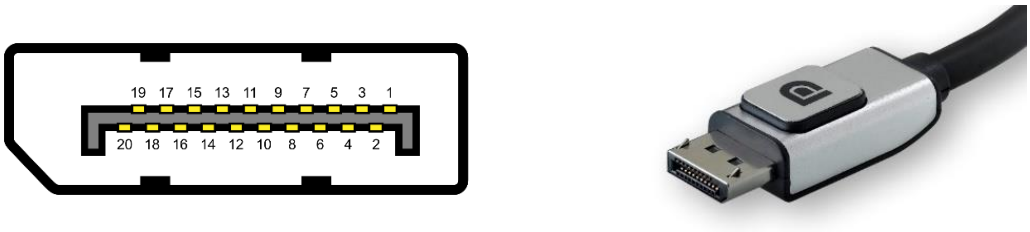
Copyright © 2021 / BME AUT

VGA szabványt leváltsa. Érdekesség, hogy a HDMI-t létrehozó 6 cégből csupán a Hitachi és a Philips nem tagja a VESA-nak. Főbb paramétereit már láthattuk a 6. fejezetben, itt csak ismétlésként emeljük ki az ott látott összetettebb táblázatból, néhány paraméterrel kibővítve azt (10.1. táblázat):

Év	Név	Bit/s	Specifikáció	Adatátviteli mód	Adatátvitel sávonként	Felhasználás	Fejlesztő
2006	DisplayPort 1.0	**10.8 Gbit/s**	1,35 GB/s	RBR, HBR	1,62 Gbit/s	PC környezet	VESA
2007	DisplayPort 1.1	**10.8 Gbit/s**	1,35 GB/s	RBR, HBR	2,7 Gbit/s	PC környezet	VESA
2010	DisplayPort 1.2	**21.6 Gbit/s**	2,7 GB/s	HBR2	5,4 Gbit/s	PC környezet	VESA
2014	DisplayPort 1.3, 1.4	**32,4 Gbit/s**	4,05 GB/s	HBR3	8,1 Gbit/s	PC környezet	VESA
2019	DisplayPort 2.0	**80 Gbit/s**	10 GB/s	UHBR	20 Gbit/s	PC környezet	VESA

10.1. táblázat – A DisplayPort fejlődése és főbb paraméterei

Az interfész alapértelmezésben 4 sávot (lane) használ az adatok továbbítására, a fenti táblázatban a Gbit/s értékek a négy sáv „együttes” teljesítményét mutatják.



10.1 ábra – A DisplayPort szabványos csatlakozója[1]

Asztali számítógépeken a DisplayPort 10.1 ábrán látható csatlakozóját találjuk, az Apple saját számítógépei (elsősorban laptopjai) számára kifejlesztette ennek a mini változatát is. A 2.0-s szabvány már lehetővé teszi az USB Type-C csatlakozó használatát is (dual-lane), megnyitva ezzel az USB interfészbe integrálás lehetőségét.

Kijelzők száma	Rövidítés	Felbontás	Refresh rate [Hz]	Színmélység [bpc]	Színtér	Tömörítés
1	16K	15 360 x 8 640	60	10	RGB/YCbCr 4:4:4	DSC
	10K	10 240 x 4 320	60	10	RGB/YCbCr 4:4:4	-
2	8K	7 680 x 4 320	120	10	RGB/YCbCr 4:4:4	DSC
	4K	3 840 x 2 160	144	8	RGB/YCbCr 4:4:4	-
3	10K	10 240 x 4 320	60	10	RGB/YCbCr 4:4:4	DSC
	4K	3 840 x 2 160	90	10	RGB/YCbCr 4:4:4	-

10.2. táblázat – A DisplayPort 2.0 konfigurációs lehetőségek

[1] https://hu.wikipedia.org/wiki/DisplayPort (Letöltés: 2020.04.27.)

Copyright © 2021 / BME AUT

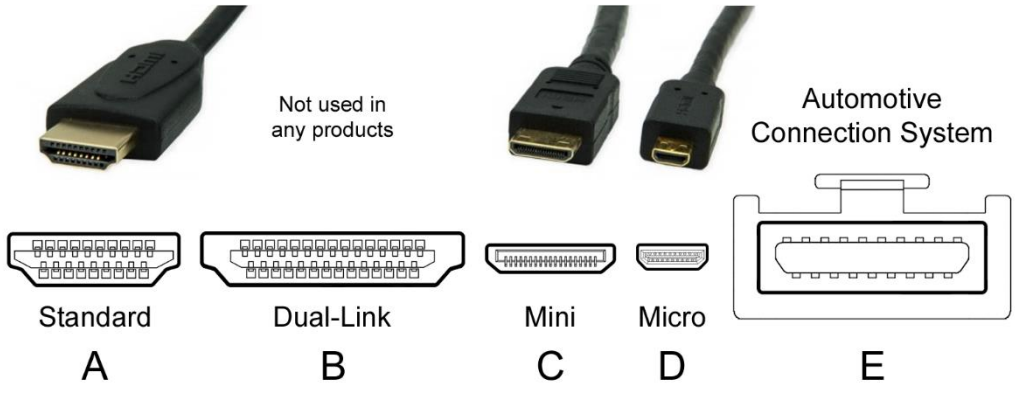
10.2 A HDMI interfész

A HDMI (High Definition Multimedia Interface) a másik korszerű csatlakozófelület képi és hanginformációk átvitelére. Tömörítetlen adatátvitelt céloz meg nagyfelbontású eszközök számára/között, ezen eszközök az interfészek témablokk bevezető fejezetében bemutatott hatalmas mértékű fejlődése kényszerítette ki a kapcsolódó interfészek fejlesztését is. Hét, az audio- és videoeszközök gyártásában közismert cég (Philips, Lattice, Maxell, Panasonic, Sony, Technicolor és Toshiba) alapították meg azt a konzorciumot, amely létrehozta és sokáig gondozta a HDMI szabványokat. Ma a Silicion Image leányvállalata, a HDMI Licensing LLC (Limited Liability Company) birtokolja a HDMI szabvány specifikációjának módosítási és lincencelési jogait, ami azt jelenti, hogy amennyiben bármely gyártó használni kívánja a szabványt egy új eszközön, jogdíjat köteles fizetni (komoly hátrány a DisplayPort interfésszel szemben). Az interfész fejlődését a 10.3. táblázat foglalja össze.

Év	Név	Bit/s	Specifikáció	Felhasználás	Fejlesztő
2002	HDMI 1.0, 1.1, 1.2	**4.95 Gbit/s**	627 MB/s	PC környezet	Hitachi, Panasonic, Philips, Silicon Image, Sony, Thomson, RCA és Toshiba
2006	HDMI 1.3, 1.4	**10.2 Gbit/s**	1,275 GB/s	PC környezet	
2013	HDMI 2.0	**18 Gbit/s**	2,25 GB/s	PC környezet	
2017	HDMI 2.1	**48 Gbit/s**	6 GB/s	PC környezet	

10.3. táblázat – A HDMI interfész fejlődése és fontosabb paraméterei

A HDMI interfész is saját csatlakozótípust használ, amely nem cserélhető fel semelyik másik interfész csatlakozójával. Laptopjaink népszerű videokimenete (pl. projektorok felé), amely mostanra teljesen leváltotta a VGA csatlakozófelületet. A szabvány az eszköz méretétől függően többféle csatlakozótípust (standard, mini, mikro) is alkalmaz.



10.2 ábra – A HDMI interfész csatlakozói[2]

[2] https://en.wikipedia.org/wiki/HDMI (Letöltés: 2020.04.27.)

Copyright © 2021 / BME AUT

A HDMI három különböző kommunikációs csatornát használ:

- **DDC (Display Data Channel)**: ez egy, az I^2C buszhoz hasonló kommunikációs csatorna. Célja a forrás és az eszköz kommunikációja annak érdekében, hogy az eszköz milyen videoformátumokat tud venni (E-DDC). Adatátviteli sebessége legalább 100 kbit/s, de üzemszerűen használják a normál 400 kbit/s sebességet is.
- **TMDS (Transition-Minimized Differential Signaling)**: ezen a csatornán történik 8b/10b kódolást követően a digitális kép- és hangadatok átvitele. Az adatátvitel csomagrendszerű, 3 különböző csomagtípus kerül továbbításra: videoadatok (Video Data Period), audioadatok (Data Island Period) és vezérlési információk (Control Period). A korábbi katódsugárcsöves kijelzők számára megalkotott videojel-felépítést követve az audioadatok és a vezérlési információk (némítás, színmélység, stb.) továbbítása a sorvisszafutási idő alatt történik.
- **CEC (Consumer Electronics Control)**: egyvezetékes kétirányú soros busz, amely az AVlink protokollon alapul. Lehetővé teszi, hogy a két összekapcsolt eszköz (TV, set-top-box, DVD lejátszó stb.) között olyan információk továbbítása történhessen meg, amelyet a felhasználó csak kezelőszervek vagy távirányító segítségével tudna elvégezni. Alkalmazása a HDMI szabványban is opcionális.

A HDMI interfész célja sok tekintetben eltér azon interfészekétől, melyekkel eddig foglalkoztunk. Ennek az interfésznek kifejezetten a video- és hangeszközök összekapcsolása és a felhasználó kép- és hangélményének stúdióminőségű kiszolgálása a célja, ideértve az akár több forrásból származó kép- és hangélményt (ARC: Audio Return Channel), vagy az internetről vett videostream lehetőségét 100 Mbit/s sebességgel (HEC: HDMI Ethernet Channel). Mind a céljai, mind a licencjogok szabályozása miatt megmaradt az interfészek között „magányos medvének” és kevésbé integrálódik bármelyik másik interfész belsejébe.

10.3 A Thunderbolt interfész

Különös figyelmet érdekel az interfészek áttekintése során az Intel által (az Apple céggel együttműködésben) kifejlesztett Thunderbolt (angolul villámcsapás) interfész. Induláskor Light Peak néven fejlesztették és forgalmazták, 2009-ben mutatták be először.

A felületet eredetileg kizárólag optikai jelátvitelre tervezték, optikai szálas kábelek felhasználásával. 2010. május 4-én Brüsszelben az Intel bemutatta a Light Peak csatlakozóval ellátott laptopot, demonstrálva: a technológia már oly mértékben miniatürizálódott, hogy beleférjen egy ilyen kisméretű eszközbe is. Két laptop egyidejű HD videostreamet küldött egymásnak, ekkor még az Intel fejlesztői azt állították, hogy a hardver gyártása 2010 végén elindul. A technológiát úgy mutatták be, hogy annak kezdeti sebessége 10 Gbit/s volt műanyag optikai kábeleken keresztül, a végső megcélzott sebesség azonban a 100 Gbit/s volt.

Az Intel fejlesztői ekkor még azt állították, hogy a vállalat „az optikai szál rézhuzallal való kombinálásán dolgozik, hogy a Light Peak interfész képes legyen a számítógéphez csatlakoztatott eszközök tápellátására is.” A cél tehát hasonló volt, mint az USB interfész

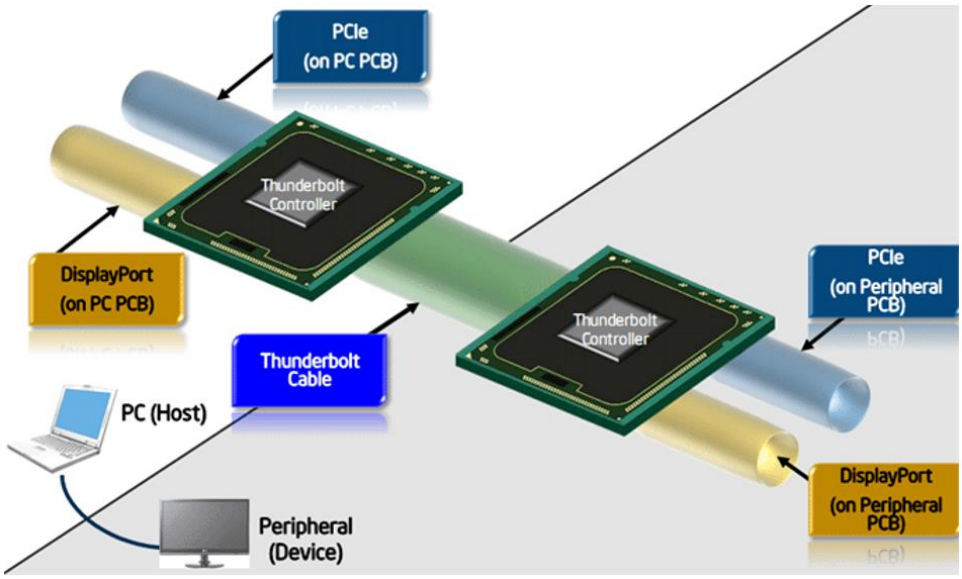
Copyright © 2021 / BME AUT

esetében, azonban a Light Peak víziója az optikai technológia bevezetése volt az informatika hétköznapi fogyasztói világába.

2011-ben az Intel fejlesztője már arról tájékoztatott, hogy a Thunderbolt névre átkeresztelt interfész első megvalósítása rézhuzalokon alapul. „A réz nagyon jó volt, meglepően jobb, mint amit gondoltunk.” Egyik fő előnye, hogy energia továbbítására képes, márpedig a Thunderbolt szabvány minden porton 10 W DC tápenergia-ellátást ír elő. (Ez az USB esetében ekkor még tipikusan 0,5..2,5 W.)

Az Intel és fejlesztő partnerei azonban továbbra is fejlesztették a Thunderbolt optikai változatát – aminek a legnagyobb ellenfele a megfelelő kábel volt. Az optikai kábeleknek legalább tíz méter hosszon át kellett volna vinniük az információt és ez technológiailag nem egyszerű feladat. Az első ilyen optikai Thunderbolt kábelt a Sumitomo Electric Industries nevű japán cég tudta gyártani 2013-ban 10, 20 és 30 méter hosszúságban. Az áruk kb. 20-30-szor volt magasabb, mint a rézből készült Thunderbolt kábeleké. Egy német, majd egy másik japán cég is beszállt a fejlesztésbe, de a megcélzott eredményeket ők sem tudták elérni. Bár a Thunderbolt 3-as szabvány 2016 óta létezik, mind a mai napig nem kapható a piacon optikai Thunderbolt 3 kábel. 2019. áprilisában a japán Corning cég bemutatott ugyan egy optikai Thunderbolt 3 kábelt, a sériaforgalmazás dátumát azonban még nem jelentették be. 2020 januárjában a CES (Consumer Electronics Show) világkiállításon mutatták be a Thunderbolt 4 változatot.

A Thunderbolt interfész célja az USB-hez hasonlóan a külső perifériák egységes csatlakoztatásának lehetősége. Az interfész multiplexeli a PCI Express és a DisplayPort jeleit és két duplex sávon továbbítja azokat, majd a céloldalon megtörténik ezek szétválasztása (ld. 10.3 ábra). Megcélozták ezen kívül a 10 Gbit/s Ethernet kapcsolat kiválthatóságát is.

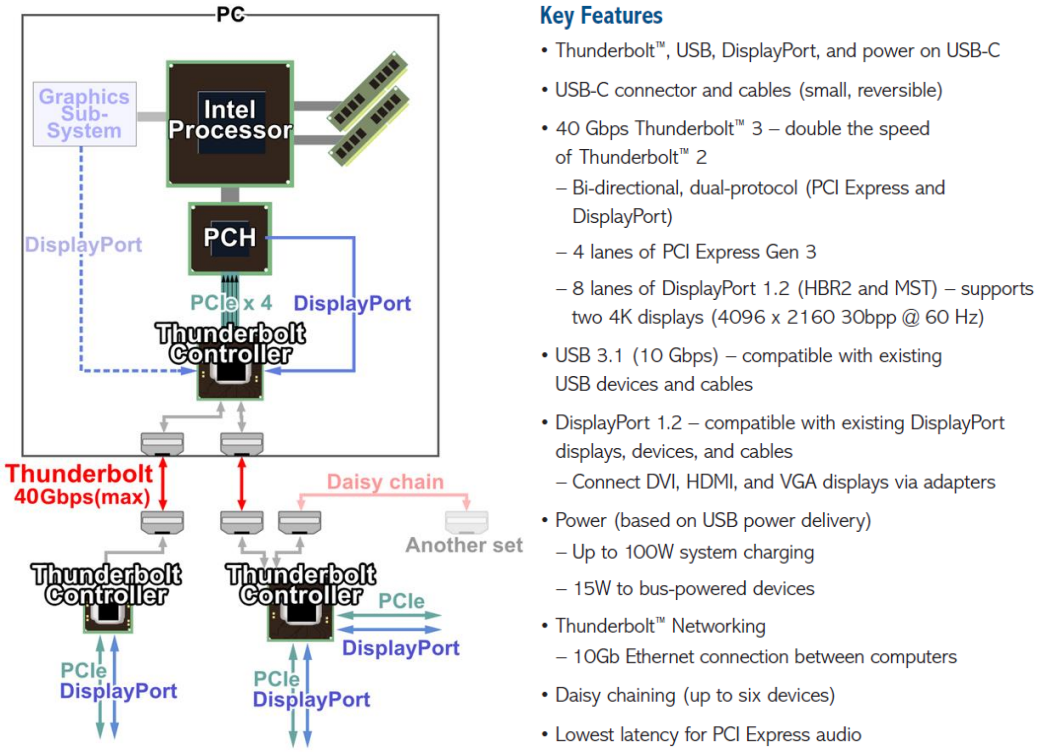


10.3 ábra – A Thunderbolt interfész alapgondolata[3]

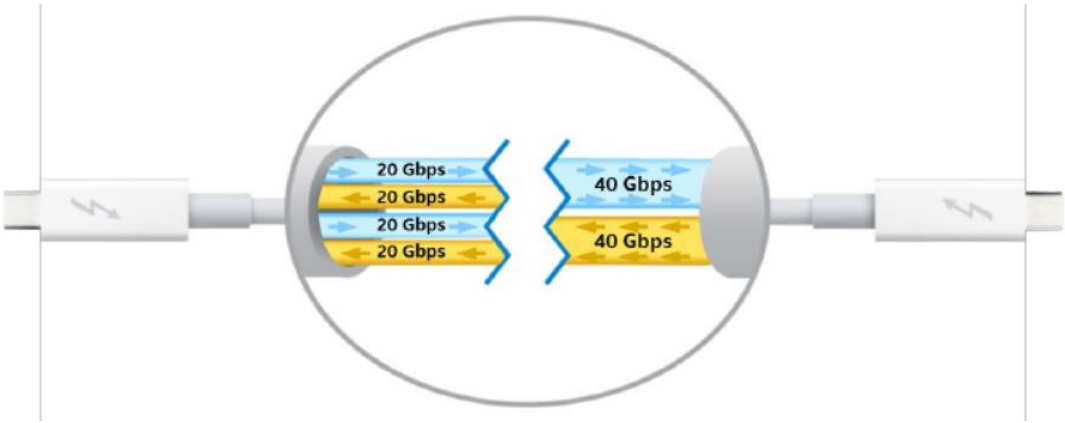
[3] https://www.intel.com/content/www/us/en/products/docs/io/thunderbolt/ (Letöltés: 2020.04.27.)

Copyright © 2021 / BME AUT

A cél az volt, hogy a számítógép és a külvilág között egy olyan nagysebességű (duplex) adatcsatornát hozzanak létre, amit utána tetszőleges protokollal „feltöltve” a szükséges interfészek valamennyi adatátviteli feladatát képes legyen ellátni. „One port can deliver all this” hangzott a cég reklámszlogenje.



10.4 ábra – A Thunderbolt interfész blokkvázlata és célkitűzései [4]

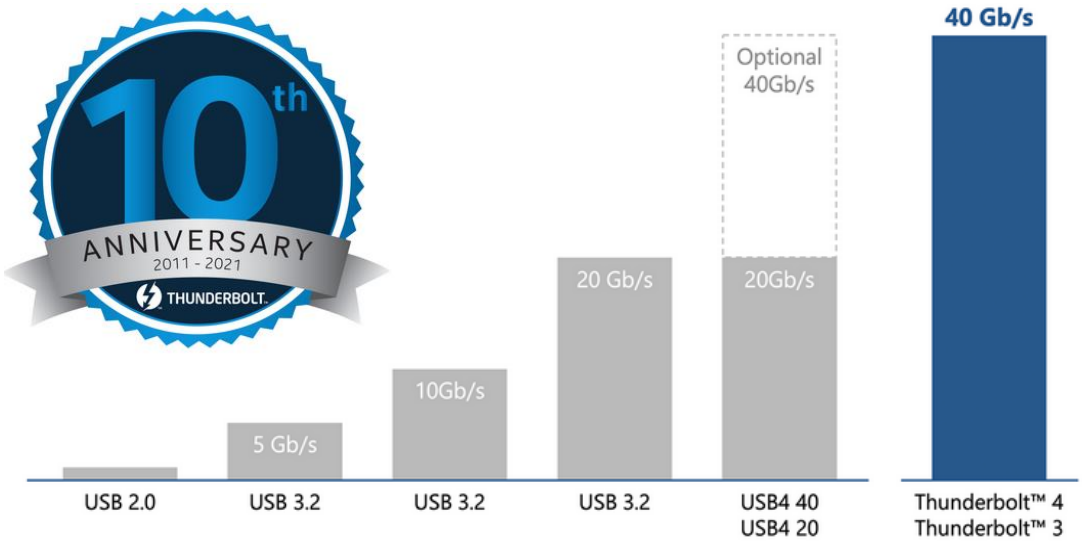


10.5 ábra – a Thunderbolt interfész dual-lane szerkezete[5]

[4] https://www.intel.com/content/www/us/en/products/docs/io/thunderbolt/ (Letöltés: 2020.04.27.)
[5] https://www.intel.com/content/www/us/en/products/docs/io/thunderbolt/ (Letöltés: 2020.04.27.)

Copyright © 2021 / BME AUT

Az optikai megoldások lassan haladtak, de a réz alapú változat gyorsan fejlődött. Az első változat 10 Gbit/s, a második 20 Gbit/s, majd mind a Thunderbolt 3, mind a jelenleg aktuális Thunderbolt 4 változat 40 Gbit/s adatátviteli sebességre képes. Mindez két duplex adatsávon (ld. 10.5 ábra), az egy sávon elérhető bitsebesség tehát megfelel az USB 3.0, 3.1 és a 4.0 sebességeinek (ld. 10.6 ábra). A két interfész egymás riválisaként fej-fej mellett haladt mindvégig, az általános célú eszközök nagyobb számban használták az USB interfészt, a Thunderbolt fő felhasználója az Apple maradt.



10.6 ábra – Az USB és a Thunderbolt interfészek fejlődése[6]

A Thunderbolt kezdetben a DisplayPort minicsatlakozóját használta, a 3-as változattól azonban áttért az USB Type-C csatlakozó használatára. Ezzel a két interfész lényeges lépést tett egymás felé, amit tovább fokozott, hogy az Intel 2017-ben bejelentette: 2018-tól felszabadítja a Thunderbolt 3 szabvány licencét, a gyártók térítésmentesen integrálhatják azt chipjeikbe.



10.7 ábra – Thunderbolt kábel USB Type-C csatlakozóval

Jogos kérdésként merülhet fel, hogy a Thunderbolt rendszer blokkvázlatában szereplő PCIe x4 Gen 3 csatorna (32 Gb/s) és 2 db DisplayPort 1.2 darabonként 4 sávos interfész (2 * 21.6 Gb/s = 43.2 Gb/s) teljes igényelt sávszélességége nagyobb, mint a Thunderbolt interfész maximális

[6] https://thunderbolttechnology.net/# (Letöltés: 2021.04.15.)

Copyright © 2021 / BME AUT

adatátviteli kapacitása. Az interfész leírásában meg is említik ezt a helyzetet a következő megállapítások mellett:

1) Valószínűtlen esetnek tartják a fejlesztők, hogy mindkét kijelző csatorna és az adatátviteli csatorna éppen a maximális sávszélességet igényelné. A Thunderbolt interfész menet közben is dinamikusan tudja változtatni az egyes csatornák sávszélességét, így a telítődés csak meglehetősen worst case helyzetben fordulhat elő.
2) A Thunderbolt interfész csomagfelépítése és az alkalmazott kódolási eljárás az overhead szempontjából kedvezőbb, mint az említett interfészeké, így a hasznos adatforgalom az egyszerűen összeadott értékeknél nagyobb lehet.
3) Ha mégis előfordulna, hogy a rendelkezésre álló sávszélesség kevés lenne, a Thunderbolt chip prioritást ad a kijelző információinak továbbítására és lassítja a PCIe csatorna adatátviteli sávszélességét annak érdekében, hogy az átvitt videojel információn az összkapacitás elérése ne legyen semmilyen formában sem érzékelhető. (Részlet a Thunderbolt 3 Technology Brief dokumentumból[7]: ”*So if a consumer stresses a specific Thunderbolt port, and attempts to use dual simultaneous displays at high resolution and additional PCI Express data, the Thunderbolt 3 silicon will prioritize the display traffic first and throttle the (PCI Express) data traffic.*”)

10.4 Kapcsolódások

Ha végiggondoljuk az előző fejezetekben áttekintett interfészek fontosabb tulajdonságait, a következő összegző megállapításokra jutunk:

1) A számítógépek körüli (illetve azon belüli) interfészek meghatározó mértékben átalakultak párhuzamos jellegből soros jellegbe. Ennek indokait korábban már áttekintettük, a folyamatot végignézhettük az ATA – SATA, SCSI – SAS, PCI – PCI Express interfészek esetében is. A párhuzamos interfészek már csak speciális, nagyteljesítményű alkalmazásokban maradtak meg, itt nagyobb teljesítményüket a sorosnál lényegesen drágább megoldásokkal tudják fenntartani.
2) A soros interfészek mindegyike kb. azonos adatátviteli sebességgel rendelkezik, amennyiben ez az adott alkalmazáshoz nem lenne elegendő, párhuzamos sávok (lanes) kiépítésével bővítik azokat. A többsávos rendszerek nem tévesztendők össze a párhuzamos rendszerekkel, mivel előbbiek differenciális jelátvitellel, sávonkénti szinkronizálással, jelcsúszás-kompenzációval rendelkeznek, ily módon kiküszöbölik a párhuzamos rendszerek sebességnövelésekor jelentkezett akadályokat.

A megismert interfészek közül fentiek figyelembevételével 4 fő interfészcsalád maradt a látóterünkben:

[7] https://thunderbolttechnology.net/sites/default/files/Thunderbolt3_TechBrief_FINAL.pdf (Letöltés: 2021.04.20.)

Copyright © 2021 / BME AUT

1) A **PCI-Express interfész** szaámítógépek belső és külső buszaként, várhatóan a nem túl távoli jövőben a SATA interfészt is kiszorítja (különös tekinettel a félvezető alapú SSD háttértárak igényeire, melyeket a SATA már ma sem tud teljesíteni).
2) A **DisplayPort interfész** a számítógép körüli megjelenítőeszközök, monitorok, video- és audiojelek kiszolgálására.
3) Az **USB interfész** a számítógépek univerzális külső illesztőegységeként;
4) A **Thunderbolt interfész** szintén univerzális külső illesztőegységként, az Intel által még fel nem adott optikai jelátvitel elvben magasabb adatátviteli frekvencia felső határt és nagyobb hatótávolságot ígér, mint a rézvezetékek esetében.
5) Ötödikként meg kell említenünk a széles körben használt **HDMI interfészt** is. Fontos azonban hangsúlyoznunk, hogy ez az interfész erősen a félprofesszionális otthoni stúdiótechnika irányába fejlődik, így céljai lényegesen eltérnek az előző 4 interfészcsaládétól. Teljestőképessége hasonló, de eltérő céljai miatt a továbbiakban nem foglalkozunk vele.
6) Hasonló módon el kell különítsük az elsődlegesen hálózati kapcsolatok céljára kifejlesztett **Ethernet interfészt** is a többiektől.

Az első 4 fent említett interfész adatátvitel szempontjából legfontosabb paramétereit a 10.4. táblázat foglalja össze.

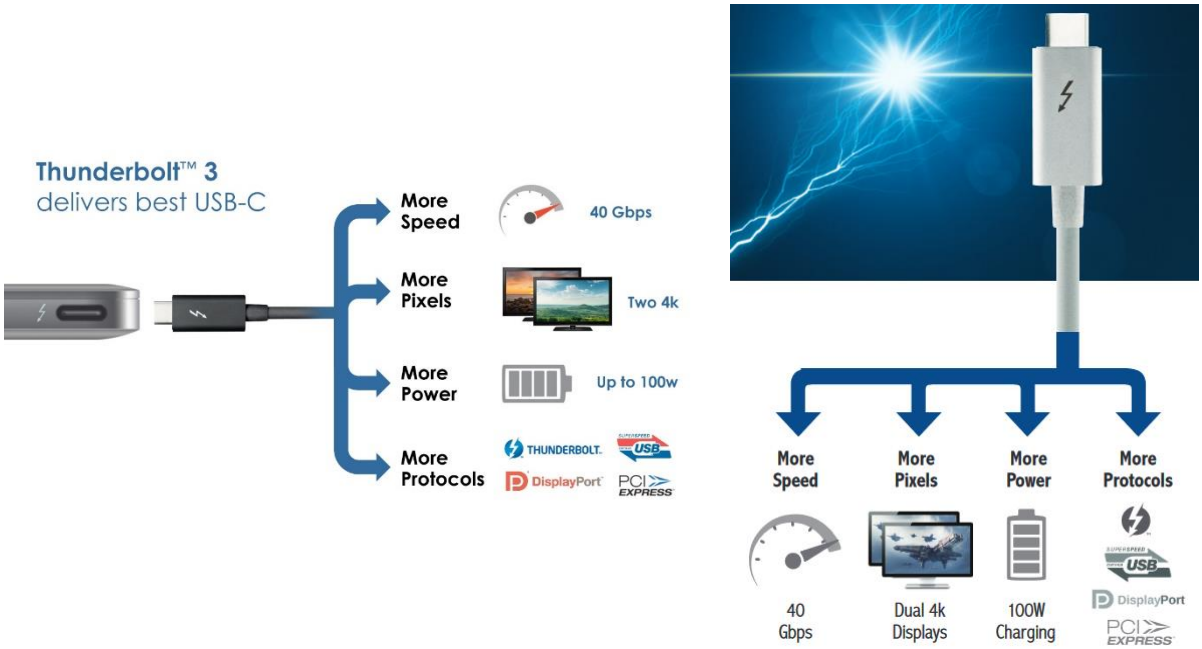
Interfész	Verzió	Sebesség	Sáv	Csúcs
PCi Express	5.0	32 GT/s	1-2-4-8-16-32	126 GB/s
DisplayPort	2.0	20 GT/s	2-4	10 GB/s
USB	4	20 GT/s	1-2	5 GB/s
Thunderbolt	4	20 GT/s	2	5 GB/s

10.4. táblázat – A legfontosabb interfészek főbb paraméterei

A közel azonos adatátviteli tulajdonságokkal rendelkező interfészeken a különböző „tulajdonságú” információk átvitele már csak protokol kérdése. Láttuk több interfész esetében is, hogy a fizikai réteg feletti rétegek már csak a digitális információ csomagolásával, az adatátvitel vezérlésével (nyugtázás, hibaellenőrzés, ismétlés, stb.) foglalkoznak, könnyű elképzelni, hogy a különböző interfészeken más adatprotokolok alkalmazásával bármelyik fenti interfész adatai átvihetők. Az USB-C (vagy USB Type-C) csatlakozótípus elfogadása és elterjedése miatt ez már az eszközök csatlakoztathatóságában sem jelentene problémát.

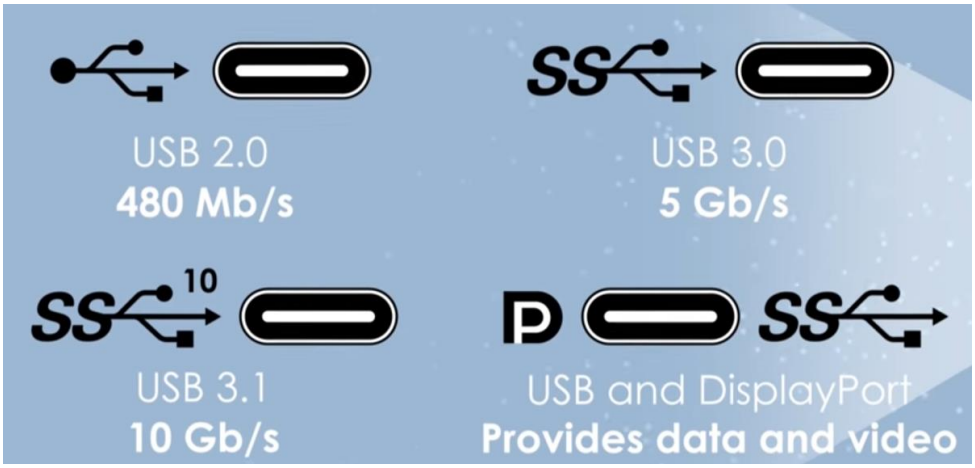
Nézzünk erre az állításra néhány példát ! A Thunderbolt interfész megjelenése óta hirdeti, hogy képes egy PCI Express és egy DisplayPort interfész összefogására, továbbítására, majd a céleszközben ezek szétválasztására (10.3 ábra). Hivatalos reklámanyagában ennél is többet ígér: a Thunderbolt, az USB, a DisplayPort és a PCI Express interfészek integrálását a felhasználás szükségletének megfelelően (10.8 ábra).

Copyright © 2021 / BME AUT



10.8 ábra – A Thunderbolt interfész reklámjai[8]

Hasonlóan az USB interfészt hirdető reklámanyag már az interfész verzióin túl a DisplayPort integrálását is mutatja, egyes forgalomban lévő laptopokon (elsősorban Apple) ez már megtalálható (10.9 ábra).

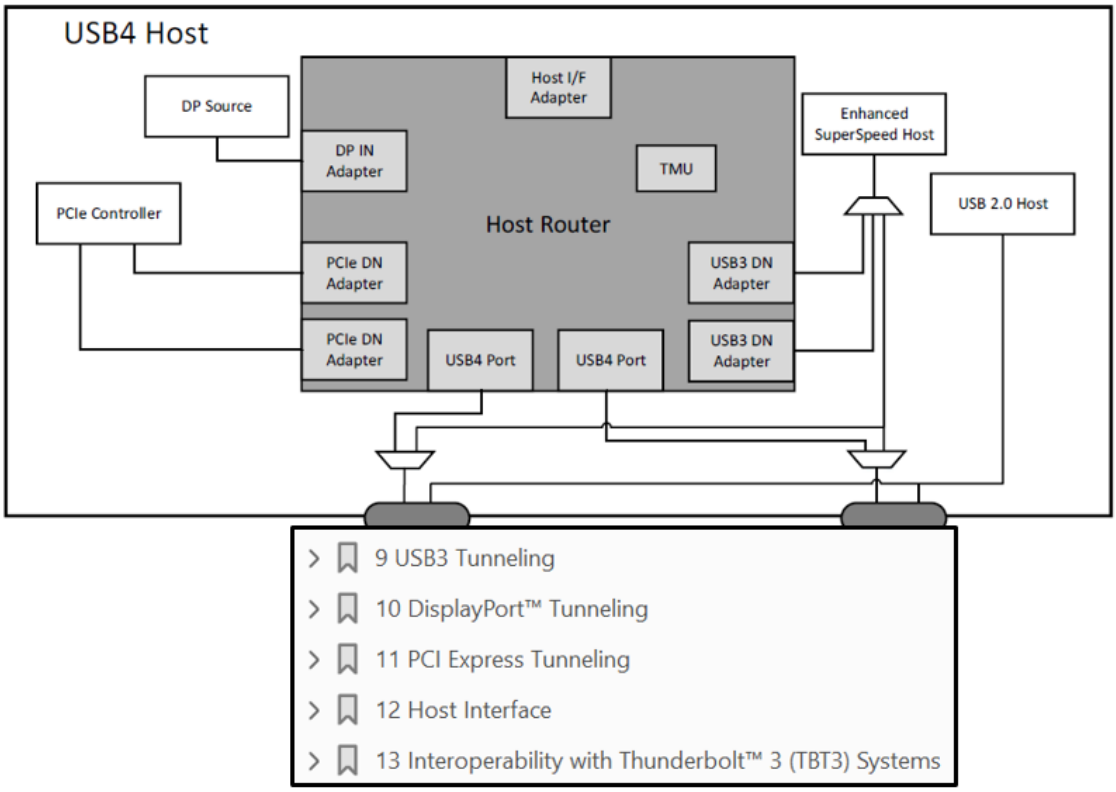


10.9 ábra – USB interfész reklámanyag

Végül bemutatjuk az USB4 szabvány egyik blokkvázlatát, amely hasonló módon a különböző protokolok integrálásának lehetőségét mutatja (10.10 ábra).

[8] https://www.intel.com/content/www/us/en/products/docs/io/thunderbolt/ (Letöltés: 2020.04.27.)

Copyright © 2021 / BME AUT



10.10 ábra – Az USB4 szabvány blokkvázlata és belső felépítése[9]

[9] Universal Serial Bus 4 (USB4™) Specification

Copyright © 2021 / BME AUT

Nagyteljesítményű mikrokontrollerek és interfészek

Elektronikus jegyzet

III. rész

System-on-Chip rendszerek

Tárgyfelelős: **Dr. Tevesz Gábor**
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117 Budapest, Magyar tudósok körútja 2.
Q ép. B szárny II. em. B216.
Tel: 463-2881
E-mail: tevesz@aut.bme.hu

Szerző(k): **Dr. Gál Tibor, Dr. Nagy Ákos**
BME Automatizálási és Alkalmazott Informatikai Tanszék

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki mesterszak (MSc)
Számítógép alapú rendszerek specializáció hallgatói

Utolsó módosítás: 2021. 04. 22.

Verzió: 2.0

Copyright © 2021 / BME AUT

COPYRIGHT

Jelen dokumentum a BME Villamosmérnöki és Informatikai Kar hallgatói számára készített elektronikus jegyzet. A dokumentumot a Nagyteljesítményű mikrokontrollerek és interfészek c. tantárgyat (BMEVIAUMA07) felvevő hallgatók jogosultak használni, és saját céljukra 1 példányban kinyomtatni. A dokumentum módosítása, bármilyen eljárással részben vagy egészben történő másolása tilos, illetve csak a szerző(k) előzetes engedélyével történhet.

Copyright © 2021 / BME AUT

Tartalom

11 SoC rendszerek általános tulajdonságai 5

11.1 Történeti áttekintés 5
11.2 Alapvető fogalmak 7
11.3 IP-k leírása 10
11.3.1 IP típusok 10
11.3.2 RTL és HDL leírás 11
11.3.3 A Verilog nyelv 12
11.3.4 IP-k beszerzése 21

12 FPGA alapú SoC rendszerek 23

12.1 Xilinx FPGA családok 23
12.2 FPGA áramkörök általános felépítése 24
12.3 Xilinx 7-es FPGA család 24
12.3.1 Logikai blokkok 25
12.3.2 Blokk RAM memóriák 30
12.3.3 DSP blokk 31
12.3.4 I/O blokkok 32
12.3.5 Általános összekötések, órajelhálózat 33
12.4 Szintézis FPGA esetén 33
12.5 Legújabb fejlesztések 35
12.5.1 Xilinx SSI technológia 35
12.5.2 Intel Agilex FPGA 36
12.5.3 Xilinx Versal platform 37
12.5.4 FPGA alapú gyorsítókártyák 38
12.6 Különböző platformok összehasonlítása 39
12.6.1 A teljesítmény és a fogyasztás kapcsolata 39
12.6.2 Esettanulmányok 40
12.7 Magas szintű szintézis (HLS) 42
12.7.1 Xilinx Vivado HLS 42

13 Lapkán belüli kommunikációs buszok 50

13.1 Buszrendszerek általános felépítése 50
13.2 ARM AMBA buszcsalád 51
13.2.1 AHB busz 52
13.2.2 APB busz 56
13.2.3 AXI busz 58

Copyright © 2021 / BME AUT

14 Processzormagok SoC rendszerekben 61

14.1 Soft processzorok 61
14.1.1 ARM Cortex-M IP-k 61
14.1.2 Xilinx PicoBlaze 62
14.1.3 Xilinx MicroBlaze 64
14.1.4 Xilinx perifériák 72
14.1.5 Intel Nios II 74
14.1.6 További gyártók soft processzorai 76
14.1.7 Akadémiai kutatások 78
14.2 Hard processzorok 81
14.2.1 Xilinx történeti áttekintés 81
14.2.2 Intel Agilex SoC 85
14.2.3 Microsemi SmartFusion SoC 85
14.2.4 Cypress PSoC 86

15 Xilinx Zynq PSoC 87

15.1 Zynq család tagjai 87
15.1.1 Zynq-7000 87
15.1.2 Zynq UltraScale+ MPSoC 88
15.1.3 Zynq UltraScale+ RFSoC 89
15.2 Zynq-7000 SoC 90
15.2.1 Összehasonlítások 90
15.2.2 Zynq-7000 PS felépítése 94
15.2.3 Zynq-7000 PL felépítése 111
15.2.4 Zynq-7000 indulási folyamata 113
15.3 Zynq kártyák 113
15.3.1 Digilent Zybo 114
15.3.2 Digilent PYNQ 115
15.3.3 MYC-CZU3EG SoM kártya 116
15.3.4 Xilinx ZCU102 fejlesztőkártya 116
Felhasznált és ajánlott irodalom 118

Copyright © 2021 / BME AUT

11 SoC rendszerek általános tulajdonságai

Az elektronika fejlődése során mindig kiemelt cél volt az integráltság mértékének növelése. A gyártástechnológia fejlődésének köszönhetően egyre kisebb méretben lehet egyre bonyolultabb feladatokat megoldani. Ennek a folyamatnak egy jelentős állomása volt, amikor teljes rendszereket tudtak egyetlen nyomtatott áramkörön megvalósítani. Ezt követőn pedig egy másik fontos lépés volt, amikor már egyetlen integrált áramköri lapka is elegendő volt a feladat megoldására. Ezeket az áramköröket *egylapkás*, angolul *System-on-Chip (SoC)* rendszereknek nevezik.

Ebben a fejezetben átvesszük az egylapkás rendszerek legfontosabb tulajdonságait, bemutatjuk a területen használt lényegesebb fogalmakat, és tervezési stratégiákat.

11.1 Történeti áttekintés

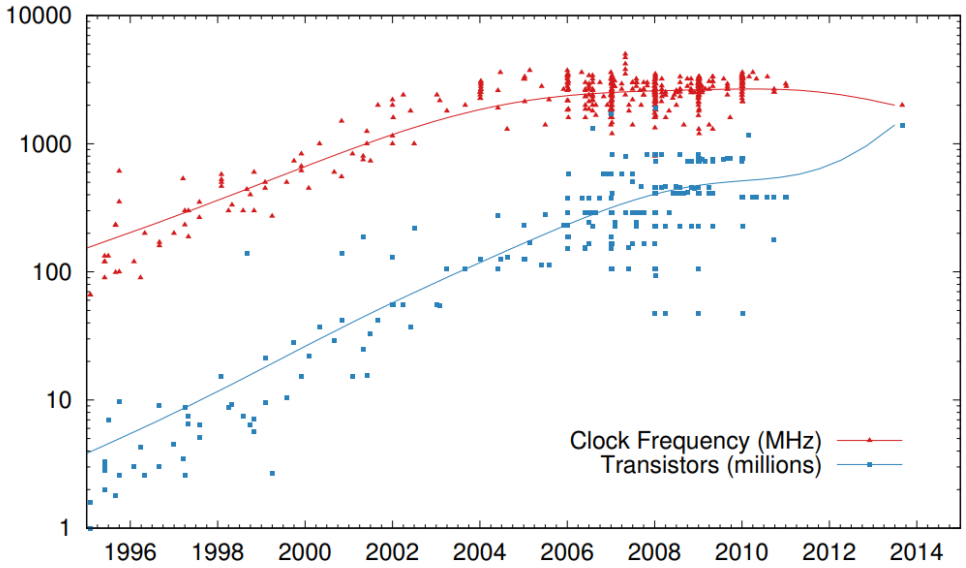
Ahogyan a második fejezetben említettük, a Moore-törvény az elektronikai rendszerek fejlődését sok szempontból jól leírja. A fejlődés első időszakában, kb. a 2000-es évek közepéig a tranzisztorok száma és az órajelfrekvenciája is megfelelt a Moore-törvény által becsült kétévente történő duplázásnak. Ebben az időszakban az egyszálú számítási teljesítmény növelése volt a fő célkitűzés, és ennek köszönhetően a processzorok disszipációja is folyamatosan emelkedett. Viszont ahogyan az 11.1. ábrán látható, a 2000-es évek közepétől az órajelfrekvencia növekedése megállt, és a tranzisztorok számának növekedése is ideiglenesen lelassult.

A folyamat megértéséhez vizsgáljuk meg a CMOS alapú tranzisztorok disszipációját. A CMOS áramkörök dinamikus disszipációját leíró korábbi képletet (2.8) egészítsük ki az alábbi módon:

$$P_D = \alpha \cdot f \cdot C_{pd} \cdot V_{CC}^2$$

Az eredeti (2.8) egyenlet az α, aktivitásnak nevezett taggal bővült ki, amely azt írja le, hogy a vizsgált áramkör az idő mekkora részében aktív. Az integrált áramkörök fejlődésének első fázisában a C_{pd} kapacitás és a V_{CC} magfeszültség folyamatosan csökkenése ellensúlyozta az f órajelfrekvencia növekedését. Ebben az időben az úgynevezett *90/10-es szabály* volt érvényes az α értékére, amely kimondja, hogy a tranzisztorok 90%-a aktív egy időben, és csupán a maradék 10% inaktív.

Copyright © 2021 / BME AUT



11.1. ábra – A tranzisztorok száma és az elért órajelfrekvencia változása[1]

A 2000-es évek után viszont a fejlődés korábbi üteme megtört. Egyrészt a magfeszültség már nem csökkenthető tovább, valamint az áramkörök statikus fogyasztása kezd jelentőssé válni. Emiatt az elérhető órajelfrekvencia nem növelhető tovább, és habár az egy lapkán elhelyezett tranzisztorok száma növekszik, ezek nem tudnak gyorsabb frekvencián működni, és egy időben nem lehetnek bekapcsolva.

A probléma megoldása érdekében elsőként többmagos rendszereket kezdtek el fejleszteni. E rendszerek, habár alacsonyabb órajelfrekvenciát használtak, de képesek voltak nagyobb számítási teljesítményt elérni adott disszipációs korlát mellett, amennyiben az elvégzendő feladat párhuzamosítható. Viszont sok esetben nincs lehetőség a feladat párhuzamosítására, így ez a megközelítés csak korlátozottan alkalmazható.

A mobil, és más beágyazott rendszerek elterjedésével ráadásul a számítási kapacitás növelése helyett az energiahatékonyság lett a fő célkitűzés. Ennek következtében speciális célhardvereket integráltak az áramkörökben, amelyek egy adott feladatot tudtak elvégezni (kisebb energiafelhasználással, mint egy általános egység). Ezen részegységek csupán az adott feladat elvégzéséig aktívak, az idő többi részében nincsenek bekapcsolva, hogy a rendszer fennmaradó részei ki tudják használni az elérhető energiamennyiséget. Ennek hatására a korábbi 90/10-es szabályt a *10x10-es filozófia* váltja fel, amely azt mondja ki, hogy a tranzisztorok csupán egy kis része lehet aktív egy időben (például 10%-a). Az áramkör azon részeit, amelyek az energiafelhasználás korlátja miatt nem lehetnek aktívak *dark silicon*-nak nevezik.

A bemutatott új megközelítés hatására speciális célhardverekből álló heterogén áramkörök alakultak ki, amelyeket egylapkás (SoC) áramköröknek nevezünk.

[1] Forrás: https://www.cs.columbia.edu/~paolo/pubs/mantovani_thesis.pdf (2020. 05. 18.)

Copyright © 2021 / BME AUT

11.2 Alapvető fogalmak

Az előző részben megmutattuk, hogy az integrált áramkörök fejlődése során hogyan jutottunk el az egylapkás rendszerekig. Fontos kiemelni, hogy a bemutatott gondolatmenet csak egy fajta megközelítés, más módon is be lehet vezetni az SoC rendszereket. Ebben az alfejezetben pedig az egylapkás rendszerek témakörében előforduló legfontosabb fogalmakat tekintjük át.

Egylapkás rendszerek (SoC)

Azokat az integrált áramköröket nevezzük egylapkás (SoC) rendszereknek, amelyek egy teljes számítógép alapú rendszert egyetlen szilíciumlapkára integrálva tartalmaznak. Manapság az SoC elnevezésnek különböző változatai is léteznek, habár ezek használata nem teljesen konzisztens a gyártók részéről.

A *PSoC* a programozható egylapkás rendszereket jelöli (*Programmable System-on-Chip*), amelyek esetén lehetőség van az áramkör működését valamilyen módon konfigurálni. Általában olyan SoC áramkörökről van szó, amelyek tartalmaznak mikroprocesszort és programozható logikát is. A Cypress cég egy konkrét termékcsaládot is bevezetett PSoC néven 2002 óta (lásd 11.2. ábra).

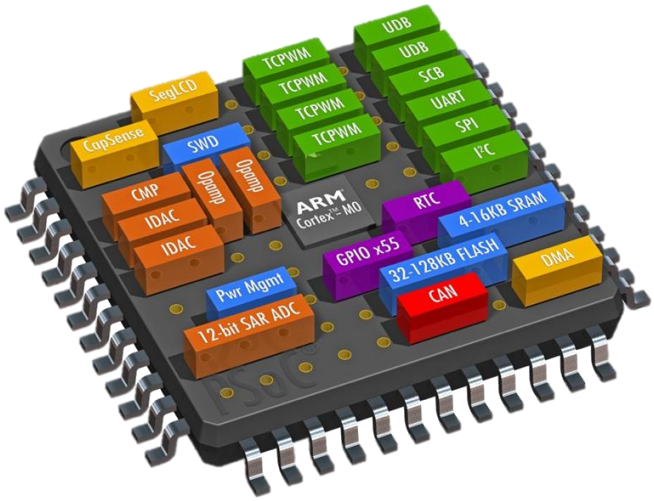
Az *MPSoC* megnevezést pedig a több mikrovezérlőt tartalmazó egylapkás rendszerek esetén használják (*Multi-Processor System-on-Chip*). Például a Xilinx cég használja az MPSoC megnevezést a Zynq termékcsaládja esetén. Ezekben az eszközökben a programozható logikán kívül ARM Cortex processzormagok is találhatók.

Intellectual Property (IP)

Ahogyan már említettük, az egylapkás rendszerek különböző feladatok elvégzésére alkalmas komponensekből állnak. Ezeket az elemeket *IP* (Intellectual Property) blokkoknak vagy magoknak nevezik. Az IP blokkokat tekinthetjük az SoC rendszerek építőelemeinek is (lásd 11.2. ábra).

A 11.2. ábrán látható *TCPWM* blokkok időzítő (timer) blokkokat jelölnek, amelyek képesek PWM jelek előállítására, és inkrementális adók fogadására is. Az *IDAC* egységek programozható árammeghajtások. Az *UDB* pedig a konfigurálható logikai blokkokat jelöli. Az ábrán szereplő többi blokkal az előtanulmányok során már minden bizonnyal megismerkedhetett az olvasó.

Copyright © 2021 / BME AUT



11.2. ábra – A Cypress PSoC4 IP alapú felépítése[2]

Az egylapkás rendszerek IP blokkjai hasonlóak a programoknál alkalmazott szoftverkönytárakra, csak ebben az esetben egy hardvert írnak le. Ahogy a szoftverkönyvtárak esetén, az IP-k esetén is fontos szempont az *újrafelhasználhatóság*, mivel egy komplex áramkör megtervezését nagy mértékben gyorsítja, ha bizonyos blokkokat nem kell minden alkalommal megtervezni.

Az IP alapú fejlesztés hatására szabványos buszok alakultak ki, és IP blokkokat tervező cégek jelentek meg. Ehhez kapcsolódóan az IP blokkok formátuma, védelme, jogi háttere is kialakult. Általánosságban elmondható, hogy az IP-k ára igen magas tud lenni (akár a tíz millió Ft nagyságrendje), és számos IP licenszdíja használatalapú, *royalty* összeget is tartalmaz.

Az IP blokkok készítésével foglalkozó cégek közül a legnagyobb az *ARM Holdings*, amely leginkább processzormagjairól ismert, de számos egyéb IP magot fejleszt. Szintén gyakran alkalmazzák az általuk kifejlesztett AMBA buszcsaládot, amelyet a 13.2. fejezetben ismertetünk. A *Synopsys Inc.* és a *Cadence Design Systems Inc.* tartozik még a jelentősebb IP fejlesztők közé. Mindkét cég kínál tervezőprogramokat, szimulátorokat is az IP alapú fejlesztéshez.

Network-on-Chip (NoC)

Ahogyan az SoC rendszerek integráltsága növekszik, egyre komplexebb belső felépítésű áramkörök keletkeznek, és az IP blokkok közötti buszrendszerek sok esetben túlságosan bonyolulttá válnak. Erre jelent megoldást, ha a blokkok közötti buszkommunikációt hálózat alapú megvalósítással helyettesítjük. Ezeket az áramköröket *Network-on-Chip (NoC)* rendszereknek nevezzük. Ebben az esetben az IP-ket összekapcsoló hálózaton csomagok

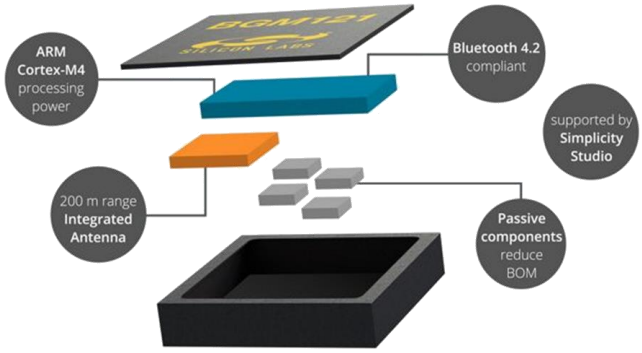
[2] Forrás: https://static.electronicsweekly.com/news/wp-content/uploads/sites/16/2015/02/PSoC-4-M-Series-Lego1-300x300.jpeg (2020. 05. 18.)

Copyright © 2021 / BME AUT

közlekednek, és a klasszikus hálózati megközelítésben használt útválasztó (router) egységek is megjelennek.

System-in-Package (SiP)

Az SoC rendszerekhez kapcsolódik a *System-in-Package (SiP)* elnevezés is. Ebben az esetben nincs olyan szoros kapcsolat a különböző modulok között, mint egy SoC esetén, mivel a komponensek nem egy lapkára vannak integrálva, hanem csak egy tokozáson belül találhatóak. A tokozásban található lapkák elhelyezkedése alapján megkülönböztetünk horizontális és vertikális felépítést is.



11.3. ábra – Silicon Labs BGM121 Bluetooth SiP[3]

A 11.3. ábrán látható egy SiP alapú áramkör, amelyeknél egy ARM Cortex-M4 processzormagon kívül Bluetooth antennát és passzív komponenseket is egy tokozáson belül valósított meg a gyártó.

System-on-Module (SoM)

A *System-on-Module (SoM)* megoldások esetén még SiP áramkörökhöz képest is lazább a különböző komponensek közötti kapcsolat. Ebben az esetben egy nyomtatott áramkörön valósítják meg a számítógépes rendszer összetevőit. A rendszer központi eleme általában valamilyen mikroprocesszort tartalmazó SoC, amely mellett memória egységek, tápáramkörök, és külső csatlakozók is helyett kapnak. SoM modult viszonylag sok gyártó készít, például a MicroChip (SAMA5D27 CPU kártya), az Olimex (A20-SOM), és még számos egyéb gyártó.

A megoldás egyik legfőbb előnye, hogy úgy tudjuk felhasználni a SoM által nyújtott lehetőségeket, hogy nem szükséges a meglehetősen bonyolult tokozású áramköröket tartalmazó nyomtatott áramkört megterveznünk, legyártanunk és beültetnünk, hanem kész modulként megkapjuk a teljes rendszert.

[3] Forrás: https://www.silabs.com/products/wireless/bluetooth/bluegecko-bluetooth-sip-module-intro (2020. 05. 18.)

Copyright © 2021 / BME AUT

11.3 IP-k leírása

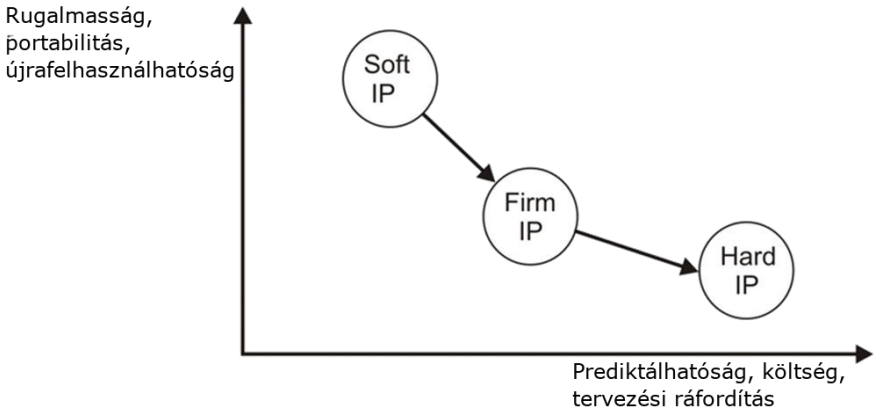
Az előző részben bemutattuk, hogy az egylapkás rendszerek legfontosabb elemei az IP blokkok. Ebben az alfejezetben pedig részletesen megismerkedünk az IP magok leírásával, és az IP alapú fejlesztéssel.

Az IP alapú megközelítés alkalmazásával óriási termelékenység-növekedés érhető el. Ahelyett, hogy az SoC tervezőjének ezeket a komponenseket külön-külön meg kellene valósítania, elegendő csak integrálni kell őket egy lapkán, ami lényegesen rövidebb idő alatt elvégezhető.

Az újrafelhasználható IP magok között megtaláljuk a processzorokat, memória blokkokat, interfész elemeket, analóg blokkokat, valamint különböző alkalmazás-specifikus funkciókat megvalósító elemeket. Az egyes blokkokhoz kapcsolódó szoftver komponensek szintén rendelkezésre állhatnak.

11.3.1 IP típusok

Alapvetően az IP magoknál megkülönböztetünk analóg és digitális blokkokat. Napjainkban rengeteg féle digitális IP blokk létezik, amelyeket a következő három csoportba sorolhatunk: *soft IP* blokkok, *hard IP* blokkok, és *firm IP* blokkok. A három csoport viszonyát a 11.4. ábrán láthatjuk.



11.4. ábra – A digitális IP magok osztályozása

Soft IP

A soft IP blokkok megadása általában valamilyen *hardver leírónyelv* (*HDL*) segítségével történik, amely független az SoC gyártási technológiájától. Így a soft leírás egy sokkal rugalmasabb módszer, mintha az IP-t megvalósító áramköri kapukat írnánk le. Ennek a formának az előnye a rugalmasságon kívül még a portabilitás és újrahasználhatóság is. Hátránya viszont, hogy nem garantáltak az időzítések és a tápellátási igények, így a különböző technológiai eljárásokban való megvalósítások különböző minőséget eredményeznek.

Copyright © 2021 / BME AUT

Hard IP

A hard IP blokkok fix layout-tal rendelkeznek, és nagymértékben optimalizáltak egy adott alkalmazásra egy specifikus gyártástechnológiai eljárásban. Nagy előnyük, hogy megjósolhatók az időzítési, fogyasztási, és egyéb paramétereik. Ez azonban járulékos ráfordítással és költségekkel jár, és a portabilitás hiánya korlátozza alkalmazási területeiket.

Firm IP

A firm IP blokkok parametrizált áramköri leírással rendelkeznek, ezért a tervező speciális céljainak megfelelően optimalizálhatja őket. Rugalmas paramétereik lehetővé teszik, hogy a megtervezett rendszer viselkedése a soft IP blokkokhoz képest nagyobb mértékben megjósolható legyen. A firm IP blokkok a soft és a hard mogok közötti kompromisszumot jelentik, rugalmasabbak és portábilisabbak, mint a hard IP blokkok, ugyanakkor jobban megjósolható a viselkedésük, mint a soft blokkoké.

Analóg IP

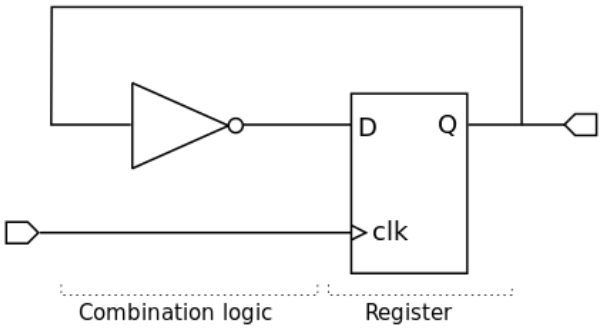
Az egylapkás rendszerek digitális IP magokon kívül számos analóg blokkot tartalmaznak. Gondoljunk csak az analóg-digitális, digitál-analóg átalakítókra, műveletei erősítőkre, fáziszárt hurok (PLL) áramkörökre, vagy egy áramkörön belüli feszültségszabályozó (pl. LDO) blokkra. Gyakran használnak úgynevezett analóg/vegyes jelű, angolul *analog/mixed-signal (AMS)* IP magokat is, amelyek mind digitális, mind analóg részt tartalmaznak.

11.3.2 RTL és HDL leírás

A továbbiakban leginkább digitális IP blokkokkal fogunk foglalkozni. Elsőként azt nézzük meg, hogy miként lehet egy digitális IP magot leírni.

A szinkron digitális logikák leírására gyakran használják a *regiszter-átviteli szintű*, angolul *register-transfer-level* (RTL) absztrakciót, amely során a rendszert tárolókkal (másnéven regiszterekkel), és a tárolók közötti adatokon végzett logikai műveletekkel modellezik. A logikai műveleteket rendszerint *kombinációs hálózatokkal* valósítják meg. A 11.5. ábrán látható egy példa az RTL leírásra. A példában a tároló egy *D flip-flop*-ot jelent, a kombinációs hálózat pedig csupán egy inverter-ből áll.

Copyright © 2021 / BME AUT



11.5. ábra – Egy egyszerű áramkör RTL leírása[4]

A digitális áramkörök fejlesztése leggyakrabban valamilyen *hardver leírónyelv*, angolul *hardware description language* (HDL) segítségével történik. A HDL és az RTL leírások közötti megfeleltetés általában könnyedén megtehető. Fontos, hogy a HDL nyelveket elsősorban párhuzamos, egyidejű leírásra használjuk, nem sorrendi leírásra, mint a szoftveres világban megismert programozási nyelveket. A két leggyakrabban használt hardver leírónyelv a *Verilog* és a *VHDL*. Ezek közül a Verilog-ot részletesen is ismertetjük a következő részben.

A folyamatot, amely egy HDL leírásból előállítja az áramkört leíró kapukat és a kapuk közötti összeköttetést, *szintézisnek* nevezzük. A szintézis kimenete a *netlist*, amely tartalmazza a szükséges kapukat és összeköttetéseket. A szintézist megvalósító *szintézer* program megfelel a szoftveres világban a fordítóprogramnak.

11.3.3 A Verilog nyelv

A Verilog nyelvet 1984-ben kezdték fejleszteni a Gateway Design Automation Inc. cégnél (mára a Cadence cég tulajdonában van). Kezdetben a nyelvet csak a hardverterv leírására és szimulálására használták, hardvert szintetizálni csak később lehetett vele. Ebből is adódik, hogy a Verilog nyelv bizonyos elemeiből nem generálható hardver, ezen elemek nem szintetizálhatóak (pl. fájlkezelés, osztályok). Alapvetően a nyelv szintakszisa hasonló a C nyelvhez (pl. kommentek, operátorok esetén).

11.3.3.1 Nyelvi elemek

A Verilog alapvetően hierarchikus felépítésű, egymásba ágyazott modulokból épül fel. Minden modul portok segítségével kapcsolódik egymáshoz. A legfelső, *top module*-nak nevezett modul pedig a külvilághoz kapcsolódik. A modulok belső részei a portok közötti logikai kapcsolatot írják le. Portok esetén megkülönböztetünk bejövő (input), kimenő (output), és két irányú (inout) portokat. A következő rövid kódrészlet egy 2:1 multiplexert valósít meg:

[4] Forrás: https://en.wikipedia.org/wiki/Register-transfer_level (2020. 05. 18.)

Copyright © 2021 / BME AUT

```
module mux_using_case(
    input wire  din_0,        // Mux elso bemeneti portja
    input wire  din_1,        // Mux masodik bemeneti portja
    input wire  sel,          // Kivalaszto port
    output reg  mux_out       // Mux kimeneti port
);
    always @(*)
    begin
        case (sel)
            1'b0: mux_out <= din_0;
            1'b1: mux_out <= din_1;
        endcase
    end
endmodule
```

Legfontosabb adattípusok

Ahogy a fenti kódrészletből is látható, alapvetően két adattípust használunk a nyelvben. A *wire* típus segítségével kombinációs logikát tudunk leírni, és alapvetőn vezeték keletkezik a wire típusból. Ebben az esetben folyamatosan biztosítanunk kell a változó meghajtását, a változó nem tárolja az állapotát.

A *reg* típus alkalmazásával pedig szekvenciális és kombinációs logika is leírható, tehát ebben az esetben tárolókat is le tudunk írni. Ezenkívül számos más adattípus is létezik, de ezeket lényegesen ritkábban használják, ezért nem térünk ki rájuk.

Bitértékek

A fenti kódban látható a konstans 0 és 1 bitérték használata (a b jelölés azt jelenti, hogy bináris konstansról van szó). Ezenkívül a nyelv definiálja a *z* és *x* bitértéket is. Az *x* érték azt jelenti, hogy ismeretlen, vagy nem releváns (don't care) az adott bitérték. Ezt leginkább szimulációnál használják, de valós hardvernél is használható (később látunk rá példát). A *z* pedig a nagyimpedanciás értéket jelöli.

Operátorok

A Verilog nyelvben alkalmazható operátorok nagyrészt megegyeznek a C nyelvből ismert operátorokkal. Az operátorok közül fontos szerepe van a *bitösszefűzés (konkatenálás)* operátornak, amelyet a következő rövid példa mutat be:

```
reg [7:0] reg8 = 8'h80;

always @(posedge clk)
begin
    reg8[7:0] <= {reg8[6:0], reg8[7]};
end
```

Copyright © 2021 / BME AUT

A példában a kapcsos zárójel közötti két változót fűzi össze a kód a bal oldalon található változóba. A példában az is látható, hogy miként lehet tömböt létrehozni, és hogyan tudunk hexadecimális konstanst megadni.

Értékadások

A Verilog nyelvben többféle módon tudunk értéket adni egy változónak, és ennek hatására más és más hardver keletkezik a kódból. A wire típus esetén az *assign* utasítással lehet értéket adni, aminek hatására memóriamentes, logikai függvényt írunk le, tehát kombinációs logika keletkezik. Ezt mutatja be a következő rövid példa:

```
wire a, b, c, d, o;
assign o = ((a & b) | (c ^ d));     //komb. logika
```

Reg típus esetén *always* blokk segítségével tudunk értéket adni a változónak. Az always blokkban megadott logika az érzékenységi listájában megadott feltételek esetén folyamatosan kiértékelődik. A következő példa ugyanazt a hardvert eredményezi, mint az előző példa (tehát kombinációs logika keletkezik), viszont itt reg típust használunk always blokkal:

```
reg o;
always @(a or b or c or d) begin
    o <= ((a & b) | (c ^ d));     //komb. logika
end
```

Az always blokk utáni érzékenységi listában minden résztvevő változó fel van sorolva, tehát bármelyik megváltozása esetén kiértékelődik a logika. A következő példában annyit változtatunk, hogy az érzékenységi listát módosítjuk:

```
reg o;
always @(posedge clk) begin
    o <= ((a & b) | (c ^ d));     //komb. logika + tároló
end
```

Látható, hogy jelen esetben a kimenet csak a clk-nak nevezett jel felfutó élére változhat meg. Ez azt jelenti, hogy most egy kombinációs logikát, és egy hozzá kapcsolódó tárolót (sorrendi hálózat) írtunk le.

A fenti két példában látható, hogy az o változó beállításához a <= szintakszist használtuk, míg az assign esetén a = jelölést. Az előbbit nem-blokkoló értékadásnak nevezik, míg az utóbbit blokkolónak. A kettő közötti különbség megértéséhez tekintsük a következő példát:

```
always @(posedge clk)    //1 ff
begin
    b = a;
    c = b;
end
```

Ebben a példában a blokkoló értékadást használtuk, amely azt eredményezi, hogy az utasítások sorrendben hajtódnak végre (hasonlóan a szoftveres világban használt utasításokhoz), így a

Copyright © 2021 / BME AUT

második sorban már az első sorban található utasítás érvényre jut. Emiatt a leírásból egy darab tároló keletkezik (persze egyszerűbben is létrehozható egy tároló). A következő példában ugyanezt a kódot láthatjuk a nem-blokkoló értékadással:

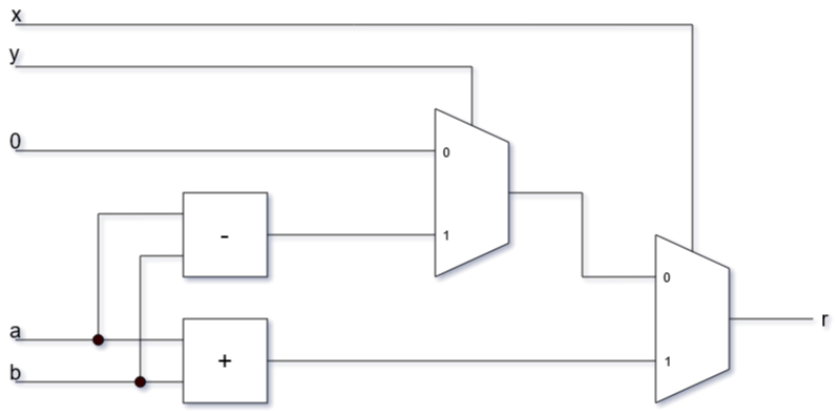
```
always @(posedge clk)    //2 ff
begin
    b <= a;
    c <= b;  //b korabbi erteke -> c
end
```

Ebben az esetben a nem-blokkoló értékadás azt jelenti, hogy párhuzamosan kiértékelődnek a jobb oldalon található kifejezések, és egyszerre jutnak érvényre a bal oldalon. Ennek hatására két tároló keletkezik a fenti példában, és az első tároló kimenete kapcsolódik a második tároló bemenetére. A nem-blokkoló értékadás jobban hasonlít a hardveres valósághoz, így célszerűbb alkalmazni.

If utasítás

Az *if* utasítás leírása hasonlít a C nyelvben megismert if utasításhoz. Lényeges kiemelni, hogy ilyenkor olyan hardver születik a kódból, amely a leírásnak megfelelő prioritást biztosítja. Ezt szemlélteti a következő kódrészlet, és a hozzá tartozó hardver:

```
if (x)
    r <= a + b;
else if (y)
    r <= a - b;
else
    r <= 8'h00;
```



11.6. ábra – A fenti kódrészlethez tartozó megvalósítás.

Az ábra alapján látható, hogy a két multiplexer kialakítása miatt az x jel az if szerkezetnek megfelelően előnyt élvez az y jelhez képest. Amennyiben az x értéke 1, akkor az r jel mindenképp a + b lesz, függetlenül az y értékétől.

Copyright © 2021 / BME AUT

Case utasítás

Az *if* utasításhoz hasonló a *case* utasítás is. Ebben az esetben is igaz, hogy prioritás van a különböző esetek között. Fontos még kiemelni, hogy létezik *casez* és *casex* változat is. Különbség akkor van köztük, ha használjuk az x, z bitértékeket is. Ezt mutatjuk be az alábbi példákban.

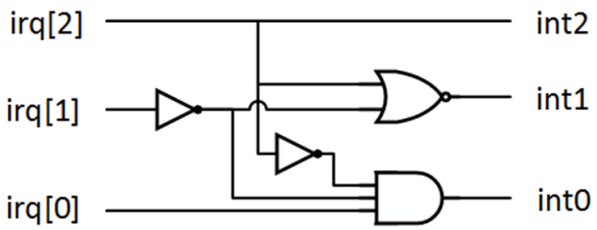
Amennyiben case-t használunk, és egy feltételnél szerepel az x vagy z érték, akkor az adott feltétel csak akkor jut érvényre, ha az adott helyiértéken az x vagy z értéket kapunk. Ennek főleg szimulációnál van jelentősége, hiszen egy valós bemeneten nem mérhetünk x vagy z értéket, mindig 0-át vagy 1-et mérünk. Tehát a következő példában szereplő harmadik eset soha nem jut érvényre valós hardver esetén (mux_out soha nem lesz mux_in[2]):

```
case (sel)
    2'b00: mux_out <= mux_in[0];
    2'b01: mux_out <= mux_in[1];
    2'b1x: mux_out <= mux_in[2];     //Nem jut ervenyre valos hw eseten
    default: mux_out <= mux_in[3];
endcase
```

Ezzel szemben a casez használatakor a feltételben szereplő z esetén érvényre jut a feltétel 0 vagy 1-es érték esetén is. Viszont a feltételben szereplő x esetén csak a feltétel x értéke esetén jut érvényre az adott ág. Casex során a z értéken kívül az x értéknél is érvényre jut a feltétel 0 vagy 1-es érték esetén is.

Ennek alapján a következő kódrészlethez az alatta található megvalósítás tartozik (fontos, hogy a case esetén is prioritás van a feltételek között):

```
always @(irq)
begin
   {int2, int1, int0} <= 3'b000;
   casex (irq)
       3'b1xx: int2 <= 1'b1;
       3'b01x: int1 <= 1'b1;
       3'b001: int0 <= 1'b1;
       default: {int2, int1, int0} <= 3'b000;
   endcase
end
```



11.7. ábra – A fenti kódrészlethez tartozó megvalósítás

Copyright © 2021 / BME AUT

A példában szereplő *default* ág akkor jut érvényre, ha egyik fenti ágban lévő feltétel sem teljesül. A casex feletti utasítás pedig a három kimeneti változó alapértékét adja meg.

Állapotgép leírása

Digitális logika tervezése közben gyakran akarunk állapotgépet készíteni, hogy szekvenciális logikát írjunk le. Többféle megoldás létezik rá, de általában case szerkezetet használunk erre a célra. A következő példában egy két állapotból álló állapotgépet láthatunk:

```
parameter ST_A = 2'b01, ST_B = 2'b10;

always @(posedge clk)
begin
    if (rst == 1'b1) begin
        state <= ST_A;
        gnt_0 <= 0;
    end else begin
        case(state)
            ST_A:
                if (req_0 == 1'b1) begin
                    state <= ST_B;
                    gnt_0 <= 1;
                end else begin
                    gnt_0 <= 0;
                end
            ST_B:
                if (req_0 == 1'b1) begin
                    state <= ST_A;
                end else begin
                    gnt_0 <= 1;
                end
            default: state <= ST_A;
        endcase
    end
end
```

A kódrészlet elején, a két állapothoz konkrét értéket rendelünk a *parameter* nyelvi elemmel. Ezt nevezik állapotkódolásnak, amely jelentősen befolyásolja a kialakuló hardvert. A szintézer képes felismerni az állapotgépeket, és akár is megváltoztathatja az állapotok kódolását. Habár a példában egy always blokkal valósítottuk meg a logikát, gyakran külön always blokkba kerül a következő állapot meghatározása vagy a kimenet beállítása, de ez csak leírásmódban különbözik, a keletkező hardver azonos mindegyik esetben.

Véletlen latch

Fontos kiemelni, hogy amennyiben egy if vagy case utasítás esetén nem teljesen specifikáljuk a logikát, latch keletkezik, amelyet FPGA-ra történő fejlesztéskor kerülni kell (ennek indokát hamarosan látjuk). Arra, hogy mit jelent a nem teljesen specifikált logika, tekintsük a következő példát:

Copyright © 2021 / BME AUT

```
always @(in) begin
    if (sel)
        out_r <= in;
end
```

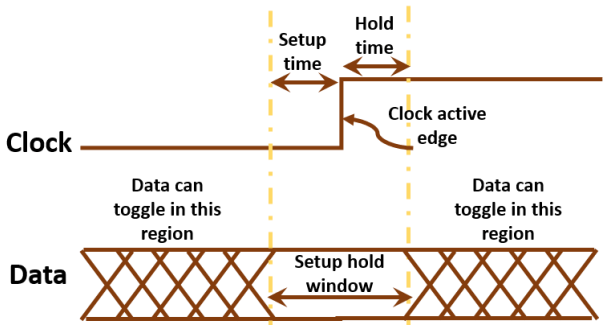
A példában nincs else ág, így amennyiben a sel változó értéke 0, tartani kell az out_r változó értékét, tehát tárolóról van szó. Viszont az érzékenységi lista alapján nem élvezérlt tárolóról van szó, így szintvezérelt tároló (latch) keletkezik.

Időzítések

Az előbbi példában bemutatott latch az időzítések miatt kerülendő az FPGA alapú fejlesztések során. Egyrészt előfordulhat, hogy az FPGA olyan elemekből tudja felépíteni a latch-et, amelyek jelentős késleltetést okoznak. Másrészt a szintézer nehezen tudja meghatározni az időzítéseket a latch-et tartalmazó jelutak mentén, és így előfordulhat olyan időzítési probléma, amelyet nem ismer fel a szintézer.

Az időzítési feltételek megértéséhez vizsgáljuk meg a következő leegyszerűsített modellt. Tegyük fel, hogy minden kombinációs logika rendelkezik egy t_{comb} *jelterjedési idővel* (angolul *propagation delay*). Ennyi idő telik el a bemenet megváltozása és a kimenet megváltozása között. Az egyszerűség kedvéért tegyük fel, hogy a magasról alacsony, és alacsonyról magas jelváltásnál azonos idő telik el. Ezenkívül a tárolók esetén is legyen egy t_{CQ} késleltetés, amely az órajel megváltozása és a kimenet megváltozása között telik el.

A késleltetéseken kívül két feltételt is be kell tartani a tárolók esetén, amelyeket a következő ábra szemléltet:



11.8. ábra – Setup és hold idők a tárolók esetén[5]

Az ábra alapján látható, hogy az órajel megváltozása előtt (t_{setup}) és után (t_{hold}) egy bizonyos ideig nem változhat meg a tároló adat bemenete. Amennyiben ezeket a feltételeket nem tartjuk be, úgynevezett *metastabil* állapotba kerülhet a tároló. Ez az állapot azt eredményezi, hogy a

[5] Forrás: https://electronics.stackexchange.com/questions/347499/setup-and-hold-time-output-when-violated (2020. 05. 18.)

Copyright © 2021 / BME AUT

tároló kimenete folyamatosan vált az alacsony és magas állapotok között, majd egy idő után véletlenszerűen valamelyik stabil állapotba kerül.

Most pedig vizsgáljuk meg, hogy ez a két feltétel milyen korlátozásokat eredményez az eddig bevezetett időmennyiségekre. A tároló órajelének periódusidejét jelölje t_{clk}, és tekintsük a következő ábrát:



11.9. ábra – Az időzítési feltételekhez használt hálózat

Az ábra alapján látható, hogy órajelváltás esetén, a bal oldali tároló kimenetének megváltozása $t_{comb} + t_{CQ}$ idő után jelentkezik a jobb oldali tároló bemenetén. Ennek az időnek a t_{hold} időnél nagyobbnak kell lennie, különben megsértenénk a jobb oldali tároló hold idejét (túl korán változna az adat). Másrészt a $t_{comb} + t_{CQ}$ időnek kisebbnek kell lennie, mint $t_{clk} - t_{setup}$, hogy a következő órajelciklushoz tartozó setup időt ne sértsük meg a jobb oldali tároló esetén (túl későn váltana az adat). Ezt a két feltételt formálisan a következő két egyenlőtlenséggel írhatjuk le:

$$t_{hold} < t_{comb} + t_{CQ} < t_{clk} - t_{setup}$$

Látható, hogy a bal oldali egyenlőtlenség alapvetően a t_{comb} jelterjedési időt korlátozza (alsó korlátként), hiszen a másik két mennyiségre nincsen hatásunk, ezeket az adott gyártástechnológia meghatározza. A jobb oldali egyenlőtlenséget pedig át lehet rendezni, hogy a t_{clk} időre kapjunk egy alsó korlátot:

$$t_{comb} + t_{CQ} + t_{setup} < t_{clk}$$

Ez az összefüggés az elérhető maximális órajelfrekvenciára ($f_{clk} = 1/t_{clk}$) határoz meg egy felső korlátot, amelynél nagyobb frekvenciával nem működhet a tároló:

$$f_{clk} < \frac{1}{t_{comb} + t_{CQ} + t_{setup}}$$

Ahhoz, hogy egy adott terv esetén nagyobb frekvenciát érjünk el, a t_{comb} értékét kell csökkentenünk (másik két paraméter továbbra is fix). Ezt elsősorban *csővezeték (pipeline)* használatával tehetjük meg. Ilyenkor az adott kombinációs hálózatot több részre bontjuk tárolók segítségével, és a tárolók között már csak töredéke lesz a t_{comb} idő. Például egy tároló alkalmazásával, felére csökkenthető a kombinációs hálózat jelterjedési ideje. A megoldása hátránya, hogy a késleltetést egy órajel cikklussal növeltük.

Copyright © 2021 / BME AUT

11.3.3.2 SystemVerilog

A *SystemVerilog* a Verilog nyelv kiegészítéseként jött létre 2002-ben. Elsősorban verifikációhoz készítették, és nem valós hardver szintetizálásához. Később, 2009-ben viszont összevonták a két nyelvet, és az összevont nyelv neve SystemVerilog lett. Az aktuális Verilog szabvány az IEEE 1800-2017.

A SystemVerilog nyelv egy jelentős része továbbra sem szintetizálható. Például megjelentek a nyelvben az objektumorientált programozásból ismert osztályok, dinamikus tömbök, garbage collector (GC), és szemaforok. A nyelv ebből a szempontból jobban hasonlít egy programozási nyelvre, mint egy HDL-re.

Az összevont SystemVerilog azonban számos szintetizálható résszel is bővült. Ezek közül bemutatunk néhány újdonságot:

- Megjelent egy új adattípus, a *logic*, amely az esetek jelentős részében felváltja a reg és a wire típusokat. Azon jelek esetében, ahol több meghajtó tartozik egy jelhez, továbbra is a wire típust kell használni.
- Megjelent három változata az always blokknak, amelyek egyértelműsítik, hogy milyen hardvert hozunk létre. Kombinációs logika esetén használható az *always_comb*, flip-flop esetén az *always_ff*, és a latch esetén az *always_latch*.
- Megjelentek a más nyelvekből ismert *struct*, *enum*, *union* kifejezések.
- Komplexebb tervek esetén nagyon sok port kapcsolódik a modulok között. Ezen segít az *interface* nyelvi elem, amely lehetővé teszi, hogy bizonyos jeleket csoportosítsunk, és ezt követően elég a létrehozott csoportra hivatkozni.

11.3.3.3 Verilog AMS

A Verilog nyelvnek létezik analóg (Verilog-A) és analóg/vegyes jelű (Verilog-AMS) kiegészítése is. A Verilog esetében alapvetően eseményalapú szimulációkat használunk, ezt cserélik le az analóg jeleket is leíró nyelvek egy folytonosidejű szimulációval.

A Verilog-AMS nyelvben megjelenik az integrálás, deriválás operátor, és lehetőségünk van különböző zajmodelleket is felhasználni. A következő egyszerű példa egy induktivitást ír le Verilog-AMS nyelv segítségével:

```
module ind(a, b);
    electrical a, b;
    parameter L = 0.001;    //1 mH

    analog
        V(a,b) <+ L*ddt(I(a,b));
endmodule
```

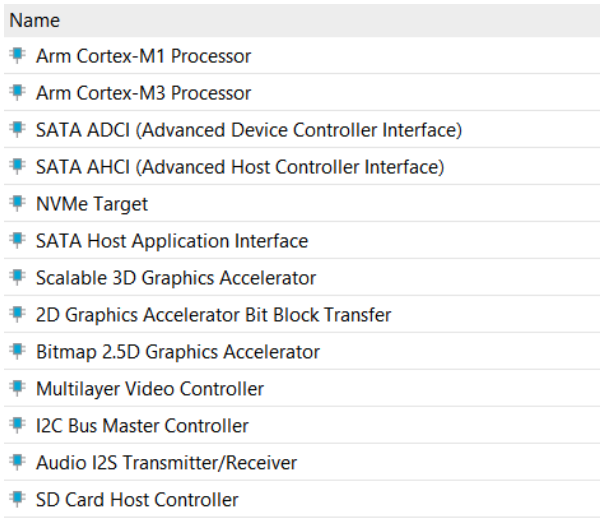
Copyright © 2021 / BME AUT

11.3.4 IP-k beszerzése

Ahogyan már említettük, az IP alapú fejlesztés egyik legfontosabb tulajdonsága az újrafelhasználhatóság. Ebből adódóan felmerül a kérdés, hogy honnan tudunk IP-ket beszerezni egy készülő hardverterv esetén. Nyilvánvalóan saját magunk is elkészíthetjük az IP magokat, de sok esetben a fejlesztési idő miatt erre nincs lehetőség.

Gyártói IP-k

Amennyiben FPGA alapú rendszert tervezünk, az FPGA gyártója számos IP-t a rendelkezésünkre bocsát. Itt egy rövid részlet, hogy például a Xilinx cég milyen IP-ket biztosít az FPGA áramköreihez:



11.10. ábra – Xilinx FPGA-hoz tartozó IP-k egy részlete[6]

Látható, hogy viszonylag sokféle IP-t kínálnak, beleértve az ARM processzormagokat, kommunikációs interfészeket (SATA, I2C, I2S, NVMe), és grafikus gyorsítókat. Ezen IP-k előnye, hogy a gyártó megadja a szükséges erőforrásokat az IP-k megvalósításához, és az IP teljes mértékben ki tudja használni az FPGA által nyújtott lehetőségeket.

Harmadik féltől származó IP-k

Ahogy a bevezetőben már említettük, számos cég foglalkozik IP-k készítésével. Példaként tekintsük a 11.11. ábrát, amelyen a Synopsys cég IP portfóliójának egy részét, az USB interfészhez tartozó listát láthatjuk. Az ábrán látható, hogy a gyártó a hard IP magjaihoz az elérhető gyártástechnológiát is felsorolja.

[6] Forrás: Xilinx Vivado 2019.2

Copyright © 2021 / BME AUT

Nyílt forráskódú IP-k

Ahogyan a szoftveres világban megszoktuk, a hardveres IP-k világában is számos nyílt forráskódú IP-t találhatunk. Nyilvánvalóan ezek megbízhatósága és dokumentáltsága sok esetben nem megfelelő, és a nyílt forráskódú IP-k szinte kizárólag soft IP-k.

Léteznek olyan weboldalak is, ahol a közösség által elkészített IP-ket összegyűjtik, rendszerezik. Erre példa a https://www.librecores.org/ és a https://opencores.org/ oldal.

Interface IP												
USB	Process Technologies										Controllers/ Features	Verification IP
	65 nm	55 nm	40/45 nm	28 nm	22 nm	20 nm	14/16nm FinFET	12nm FinFET	10nm FinFET	7nm FinFET		
USB 3.2										✓	Device	✓
USB 3.1							✓	✓		✓	Dual-Role Device (Device & Host)	✓
USB-C 3.1							✓	✓		✓	Dual-Role Device (Device & Host)	✓
USB-C 3.1/ DisplayPort 1.4							✓	✓	✓	✓	Device, Host, DisplayPort Tx, Subsystem Solution for 16FFC	✓
USB-C 3.1/ DisplayPort 1.3							✓	✓	✓	✓	Dual-Role Device (Device & Host), DisplayPort Tx	✓
USB 3.0	✓	✓	✓	✓	✓	✓	✓	✓			Dual-Role Device, Device, Host, HSIC	✓
USB-C 3.0				✓	✓		✓	✓			Device, Host	✓
USB 2.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Device, Host, HSIC, Dual-Role Device	✓
USB-C 2.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Device, Host, Dual-Role Device	✓
eUSB2										✓	Device, Host, Dual-Role Device	✓
HSIC			✓	✓		✓	✓		✓		Device, Host	✓

11.11. ábra – Synopsys cég USB IP blokkjainak listája[7]

[7] Forrás: https://www.synopsys.com/dw/doc.php/ds/o/product_overview.pdf (2020. 05. 18.)

Copyright © 2021 / BME AUT

12 FPGA alapú SoC rendszerek

Az előző fejezetben már többször említettük az FPGA-kat, mint az egylapkás rendszerek egy lehetséges platformját. Ebben az fejezetben kicsit részletesebben bemutatjuk az FPGA-kat, elsősorban a Xilinx cég 7-es FPGA sorozatát.

Programozható eszközök viszonylag régóta léteznek, a 60-as évek vége felé már megjelentek olyan áramkörök (pl. Motorola XC157), amikkel egyszerű logikai függvényeket le lehetet írni. Később megjelentek a *CPLD (Complex Programmable Logic Device)* eszközök, amelyek már sok szempontból hasonlítanak az FPGA-kra. CPLD-ket a mai napig több gyártó is árul, és sok helyen használnak az iparban.

A *Field-Programmable Gate Array (FPGA)* eszközök a 80-as évek elején jelentek meg. Nagyjából egy időben két amerikai gyártó is piacra vitte saját megoldását (*Altera Corporation, Xilinx Inc.*), akik azóta is a piac 90%-át birtokolják (azóta az Altera-t az Intel, míg a Xilinx-ot az AMD vette meg).

12.1 Xilinx FPGA családok

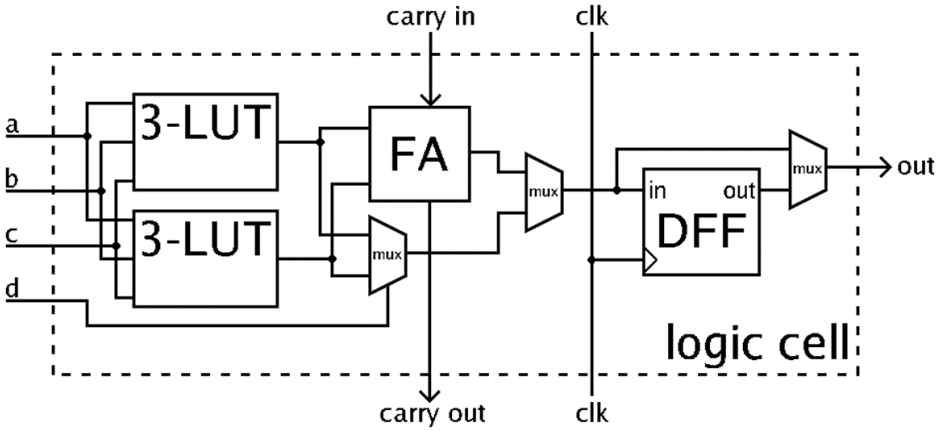
A Xilinx jelenleg a legnagyobb FPGA gyártó a piacon. Mielőtt részletesen bemutatjuk az egyik FPGA generációjuk felépítését, röviden ismertetjük a cég legfontosabb FPGA eszközeit:

- Spartan család: A *Spartan* elnevezésű FPGA eszközök alacsony árral és korlátozott teljesítménnyel rendelkeznek. A Spartan tekinthető a belépőszintű FPGA sorozatnak. A régebbi Spartan 6-os generációt továbbra is gyártja a Xilinx 2027-ig, habár a jelenleg elérhető fejlesztőeszköz nem használható ehhez a termékhez. A később bemutatott 7-es szériához tartozó Spartan 7 már elérhető a legújabb fejlesztőeszközben is.
- 7-es széria: A 2010-ben megjelent 7-es szériájú FPGA eszközök 28 nm-es gyártástechnológiával készülnek, de ennek ellenére széles tartományban kínálnak FPGA alapú megoldásokat. A széria legkisebb, Spartan 7-es tagja a modern belépőszintű FPGA eszközöket jelenti, viszont a *Virtex 7*-es család a piac legnagyobb FPGA eszközeihez tartozik. A Virtex és a Spartan között található két további szint is, Artix és Kintex néven.
- UltraScale/UltraScale+: 2010 óta megjelent a Virtex és a Kintex eszközök újabb generációja 20 és 16 nm-es gyártástechnológiával. Ezek egyértelműen a nagyobb teljesítményű, költségesebb alkalmazásokhoz készültek. A generáció legolcsóbb tagjai is több száz ezer Ft-ba kerülnek, és a legdrágább eszközök a több tíz millió Ft-ot is elérik.

Copyright © 2021 / BME AUT

12.2 FPGA áramkörök általános felépítése

Az FPGA áramkörök legfontosabb része a programozható logikai blokk. A következő ábrán egy egyszerű felépítésű, általános logikai blokkot mutatunk be:



12.1. ábra – Egy egyszerű felépítésű logikai blokk[1]

Az ábrán látható *LUT (Lookup table*) blokkokkal bármilyen 3 bemenetű és 1 kimenetű logikai függvényt le lehet írni. Az *FA (full adder)* pedig egy 2 bites teljes összeadót jelöl, amihez átvitel logika is tartozik. A DFF pedig egy élvezérelt D flip-flopot jelent. Látható az ábrán, hogy az előbb felsorolt blokkok közötti multiplexer (MUX) áramkörökkel tudunk különböző struktúrákat kialakítani. Fontos megjegyezni, hogy alapvetően a logikai blokk bemenetei és kimenetei horizontálisan kapcsolódnak a szomszédos blokkokhoz, viszont a tárolóknál lévő órajelnek dedikált órajelhálózata van a legtöbb FPGA esetén (clk jelek az ábrán). Szintén dedikált vonala van az átvitel hálózatnak a logikai blokkok között (carry in és out az ábrán).

12.3 Xilinx 7-es FPGA család

Most pedig részletesen megnézzük a Xilinx cég 2010-ben megjelent 7-es FPGA családját. Habár a 7-es sorozat óta megjelentek újabb generációs FPGA-k is, a 7-es család a mai napig népszerű maradt, és az újabb generációs eszközök nem sokban térnek el a 7-es sorozattól.

Általánosságban igaz, hogy az FPGA-k a következő fontosabb részekből épülnek fel:

- Programozható logikai blokkok
- Összekötések, órajel hálózat
- Dedikált blokkok (hard IP-k):
    - Blokk memóriák

[1] Forrás: https://hu.wikipedia.org/wiki/Field-programmable_gate_array (2020. 05. 18.)

Copyright © 2021 / BME AUT

- DSP blokkok
- I/O blokkok (nagysebességű soros átvitellel)
- Egyéb hard IP-k (pl. PCI Express vezérlők)

12.3.1 Logikai blokkok

A Xilinx a logikai blokkokat *CLB*-nek *(Configurable Logic Block)* nevezi. Minden CLB-t felosztanak további két szeletre, amit *SLICE*-nak neveznek. A két SLICE között vannak kisebb eltérések, de az egyszerűség kedvéért most csak az egyik típust vizsgáljuk (SLICEM változat).

A 12.2. ábrán látható a teljes SLICEM felépítése. A felépítés nagyjából megfelel a 12.1. ábrán látott egyszerűsített blokknak. Az egyik eltérés, hogy jelen esetben 4 db 6 bemenetű és 2 kimenetű LUT-ot tartalmaznak a blokkok (a 12.2. ábra bal oldalán). A tárolók tekintetében kétszer 4 db D flip-flop található a SLICEM-ben. Az ábra jobb szélén látható 4 db tároló latch-ként is tud működni, míg a középen található tárolók csak élvezérelt módon. Habár dedikált teljes összeadót nem tartalmaz a SLICE (LUT-ból kell megvalósítani), viszont a vertikális átvitel logika megtalálható, amely az aritmetikai műveletek esetén szükséges.

12.3.1.1 Lookup table (LUT)

Ahogyan már említettük a LUT-ok segítségével logikai függvényeket tudunk leírni. A konkrét SLICEM esetén annyi megkötés van, hogy 6 bemenet használata esetén csak 1 kimenetet használhatunk. Kevesebb bemenetű logikai függvény esetén mind a két kimenet használható. Mivel a SLICEM több LUT-ot is tartalmaz, ezért lehetőség van ezek összekapcsolásával akár 8 bemenetű logikai függvényt leírni. Az összekapcsolás a dedikált MUX-ok segítségével lehetséges. Fontos megjegyezni, hogy LUT-ok jelterjedési késleltetése független a megvalósított függvénytől.

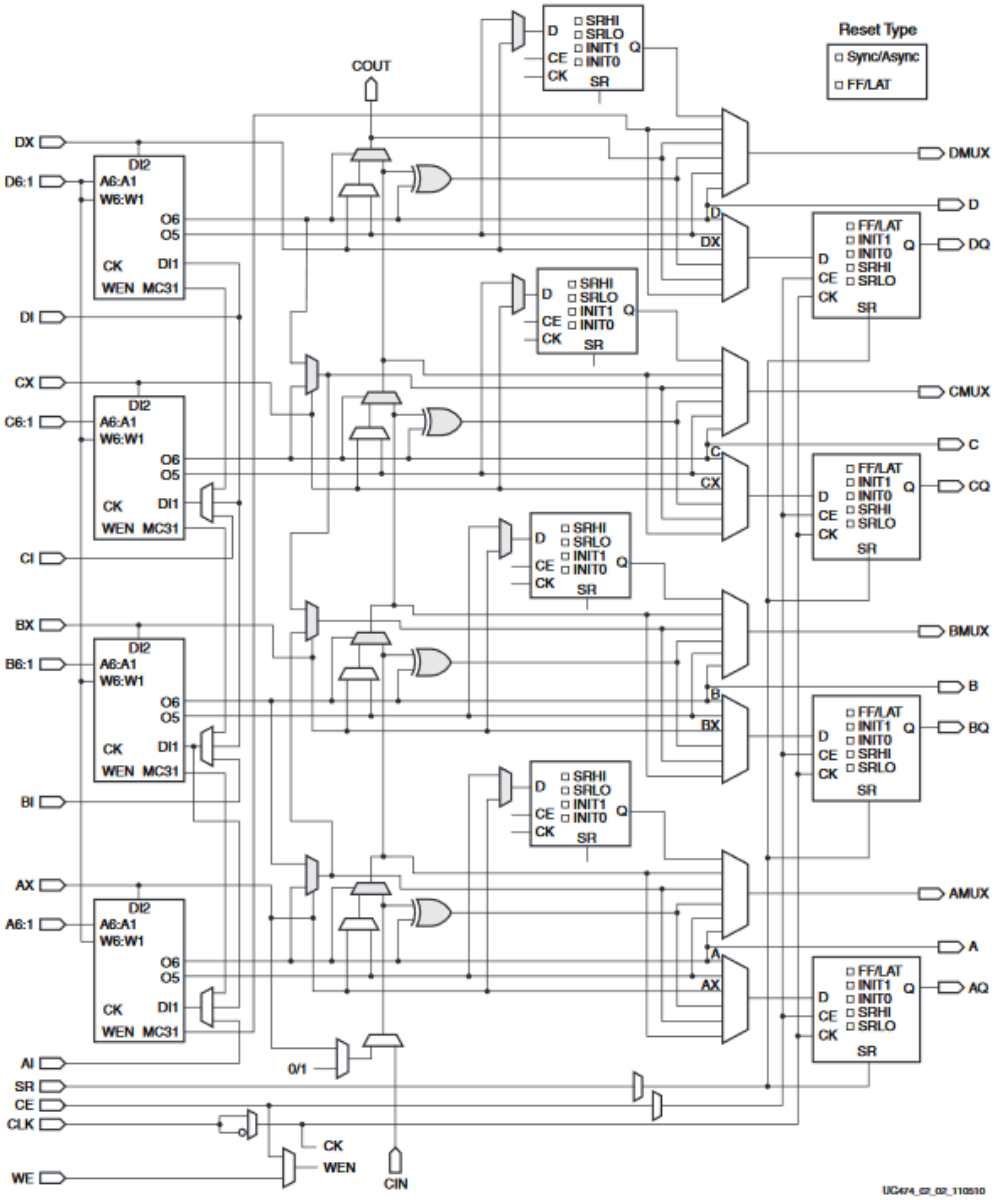
A SLICEM-ben található LUT-ok segítségével memóriaelemet is létre tudunk hozni, akár RAM, akár ROM memóriát. Ennek az előnye, hogy mivel a LUT-okat sokféleképpen lehet egymáshoz kapcsolni, sokféle struktúrájú memóriát ki tudunk alakítani. Megadható a memória szélessége, mélysége, és a hozzáférési portok száma is.

A 12.3. ábrán egy 2 bit széles, 32 elemű, 4 portos LUT-okból felépülő memória látható (LUT RAM). Az ábra jobb oldalán található 4 adatporton (DOA-DOD) keresztül olvasni tudunk csak, ezekhez bal oldalt találhatók a címportok. Írás esetén a LUT-ok WA bemeneteire jutó azonos címet írjuk a DID bemenetekkel.

A 12.4. ábrán pedig egy 1 bit széles, 128 elemű 1 portos memória látható, amelyet két LUT-ból tudunk előállítani. Ennél a példánál az is látható, hogy a SLICE-ban található tárolót felhasználhatjuk opcionálisan a memória kimenetéhez (gondoljunk bele, hogy az időzítéseknél elmondottak alapján ez miért előnyös).

Copyright © 2021 / BME AUT

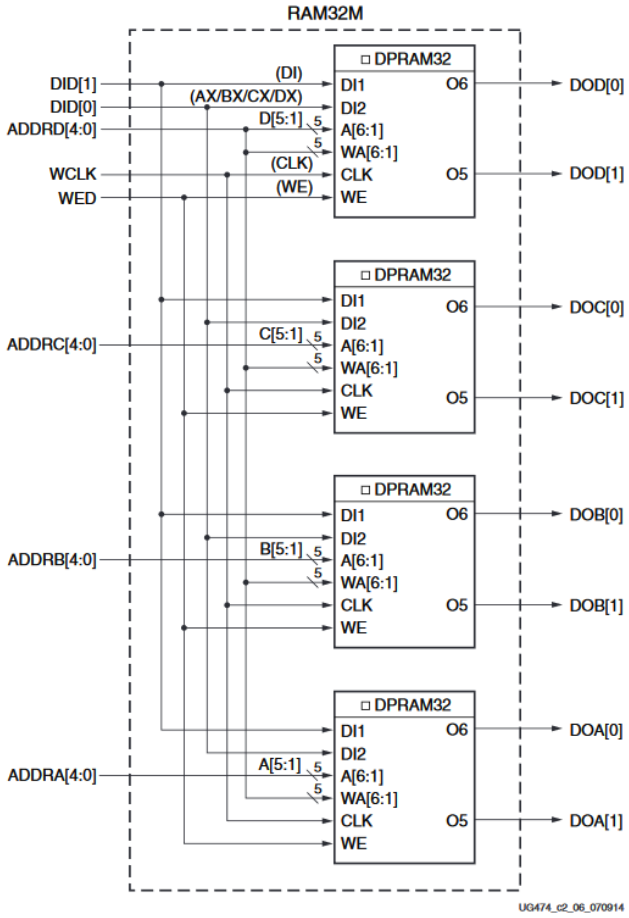
Fontos kiemelni, hogy a SLICEM-ben található LUT-ok rendelkeznek olyan speciális portokkal is, amelyek alapvetően egy LUT-hoz nem szükségesek, de más célokra (pl. memória létrehozása) hasznosak. Ezek közé tartozik a LUT CLK és WE portja is.



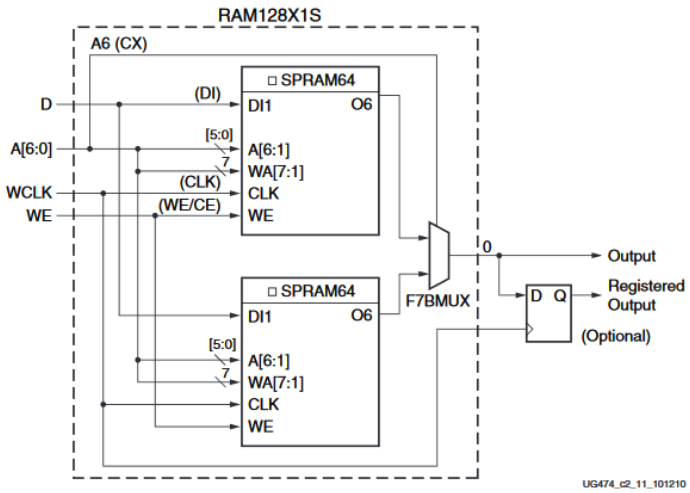
12.2. ábra – Xilinx 7-es sorozat esetén egy SLICEM blokk felépítése[2]

[2] Forrás: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf (2020. 05. 18.)

Copyright © 2021 / BME AUT



12.3. ábra – Egy 4 portos 32x2 bites LUT memória[2]

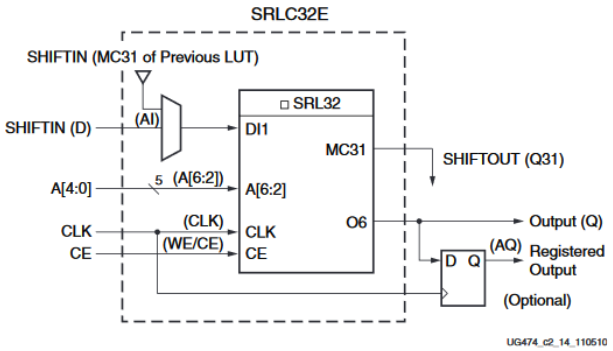


12.4. ábra - Egy 1 portos 128x1 bites LUT memória opcionális regiszter kimenettel[2]

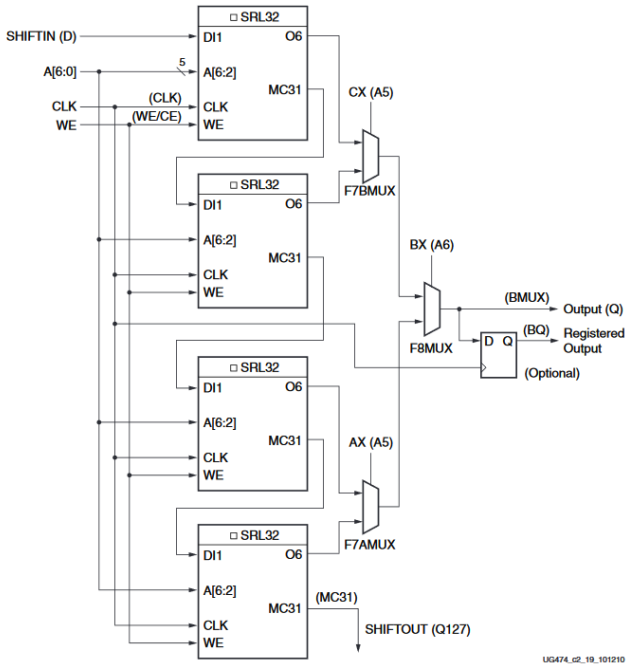
Copyright © 2021 / BME AUT

A SLICEM-ben található LUT segítségével léptető regisztert (shift regisztert) is létre tudunk hozni. Shift regisztereket például szűrők esetén használunk. Alapvetően egy LUT-tal 32 bites shift regiszter alakítható ki. A LUT ilyen esetben is címezhető (A porton keresztül), hogy egy adott helyiértéken lévő bitet megkapjunk, és így 32 bitnél kisebb léptetés kialakítható. A címzéshez tartozó kimenet a O6-os port. Ahogy a 12.5. ábrán látszik, a DI1 bemeneten tudjuk a léptető regiszter kezdőbitjét megadni, és az MC31-es porton kapjuk meg a kilépő bitet. A DI1 és MC31 portok, a CLK és WE jelekhez hasonlóan speciális portok, amelyek a LUT eredeti funkciójához nem szükségesek.

Mivel a DI1 bemeneten található egy MUX, így több LUT-ból kialakítható nagyobb léptető regiszter is. Például a 12.6. ábrán látható egy 128 bites léptető regiszter, ahol látszik, hogy a LUT-ok DI1 bemenetei és a MC31 kimenetei összekapcsolódnak. Szintén látható, hogy ilyenkor is van lehetőség címezni a shift regisztert.



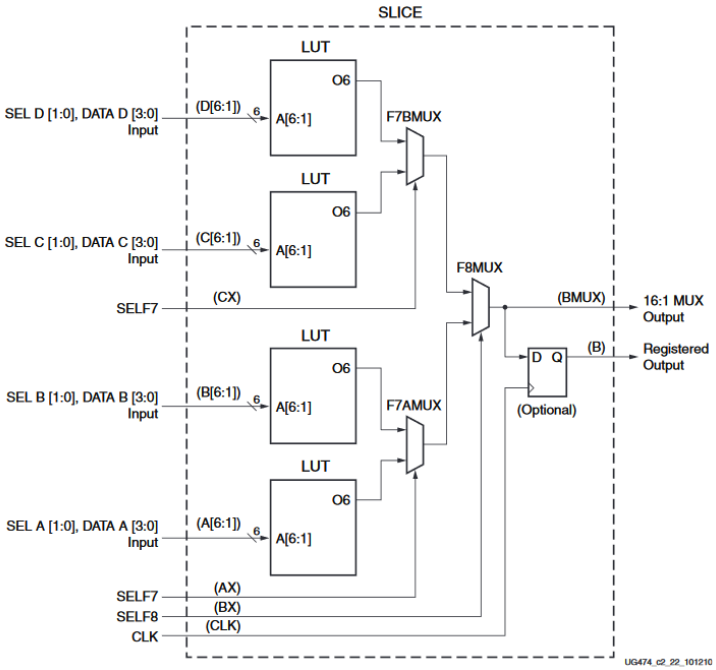
12.5. ábra – 32 bites shift regiszter kialakítása LUT-ból[2]



12.6. ábra – 128 bites shift regiszter kialakítása 4 db LUT-ból[2]

Copyright © 2021 / BME AUT

A LUT-ból multiplexer (MUX) is könnyedén kialakítható, amit nagyon gyakran használunk digitális logikáknál. A 12.7. ábrán egy 16:1-es MUX látható, tehát itt 16 bemenetből 1 db-ot választunk ki. Ahogy látható az ábrán, ehhez felhasználjuk a SLICE-ban található mind a 4 db LUT-ot, és még 3 dedikált 2:1 MUX-ot is. A MUX-hoz tartozó 4 bites kiválasztójel alsó 2 bitje megy a LUT-ok *A* portjára, a felső 2 bit pedig a dedikált MUX-okhoz (SELF7, SELF8). A 16 db adatvezeték szintén a LUT-ok A portjára megy.

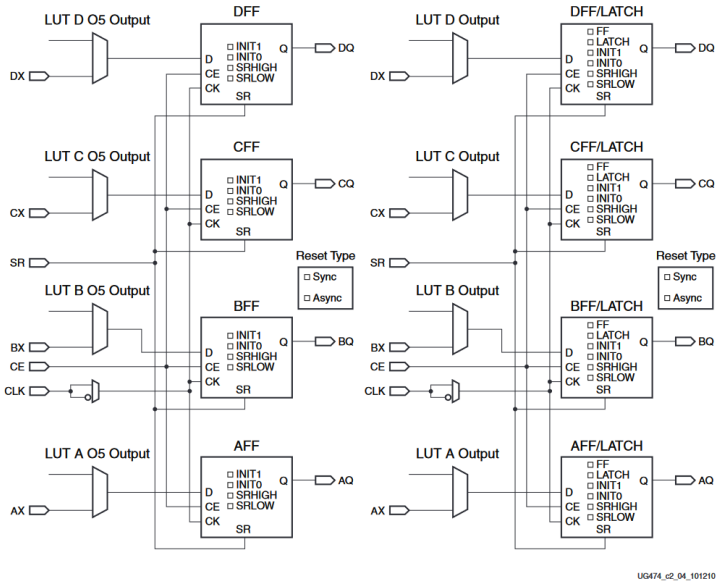


12.7. ábra – 16:1-es MUX kialakítása LUT-ból[2]

12.3.1.2 Tároló elemek

Ahogy a bevezetőben is említettük, összesen 8 darab tárolót tartalmaz egy SLICEM blokk (lásd 12.8. ábra). Minden tároló tartalmaz egy adatbemenetet (D), egy órajel bemenetet (CK), egy engedélyező jelet (CE), és egy konfigurálható SR alaphelyzetbe állító vezérlőjelet. Az SR esetén konfigurálható, hogy magas (SRHIGH) vagy alacsony (SRLOW) legyen a kimenet a vezérlőjel hatására. Az is megadható, hogy az SR jel az órajelre szinkron vagy aszinkron legyen. Az ábrán látható INIT0, INIT1 konfigurációs bitekkel pedig megadható, hogy mi legyen az adott flip-flop kezdőértéke.

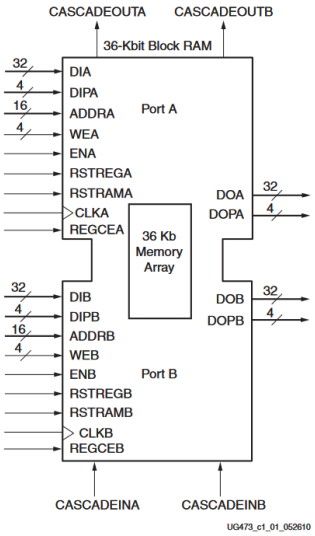
Copyright © 2021 / BME AUT



12.8. ábra – A SLICEM-ben található tároló elemek[2]

12.3.2 Blokk RAM memóriák

Az FPGA-ban kialakítható memória a CLB egységektől függetlenül is. Ezeket *blokk RAM* moduloknak nevezik, és minden modul 36 Kb méretű, így elsősorban a nagyobb memória blokkokhoz célszerű használni őket. A Xilinx leírása alapján 16 bitnél szélesebb esetben célszerű blokk RAM-ot használni, alatta a korábban bemutatott LUT RAM-ot.



12.9. ábra – A blokk RAM modul interfésze[3]

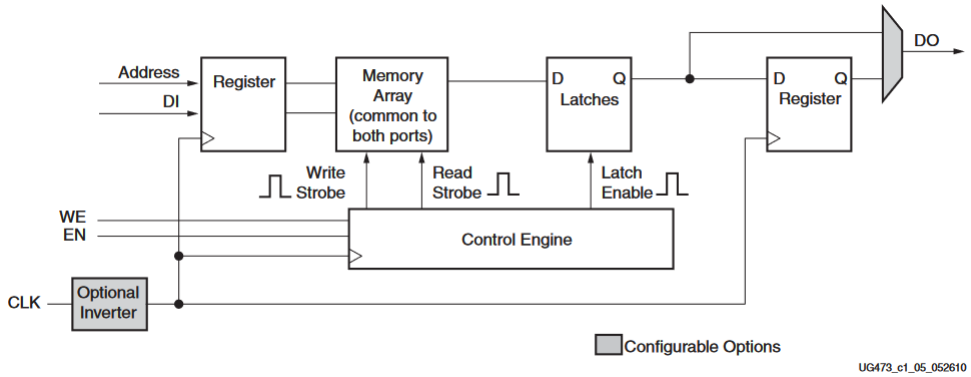
[3] Forrás: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf (2020. 05. 18.)

Copyright © 2021 / BME AUT

Ahogy a 12.9. ábrán látható, a blokk RAM-ok dualport memóriák, teljesen független órajel bemenetekkel (CLKA/B), engedélyező jelekkel (ENA/B), írás vezérlőjelekkel (WEA/B), cím buszokkal (ADDRA/B), és adat buszokkal (DIA/B, DOA/B). Az ábrán az is látható, hogy minden adatport mellett található egy 4 bites port is, amely alapvetően hibaellenőrzésre szolgáló paritásbiteket tartalmaz.

A blokk memóriákat rugalmasan lehet konfigurálni, beleértve a memória szélességét, mélységét, és a portok számát (1 vagy 2 port). Például megtehető, hogy a két 32 bites adatbuszt és a hozzájuk tartozó paritásbiteket összevonjuk, és így egy 72 bit széles, egy portos memóriát kapunk (ilyenkor persze a memória mélysége lecsökken). Nagyobb memória blokkokat is ki lehet alakítani az ábrán látható kaszkád be- és kimenetekkel.

A blokk RAM mindkét kimenetén található egy-egy opcionális regiszterszint (lásd 12.10. ábra), amellyel az időzítések javíthatók egy plusz órajelciklus késleltetéssel. Ezekhez a regiszterekhez külön reset és engedélyező jelek tartoznak. A 12.9. ábrán látható RSTREGA/B és REGCEA/B feliratok ezeket a jeleket jelölik. A regiszterek előtt található egy latch is, amelyhez szintén elérhető egy alaphelyzetbe állító jel (RSTRAMA/B).



12.10. ábra – Blokk RAM egy portjának felépítése[3]

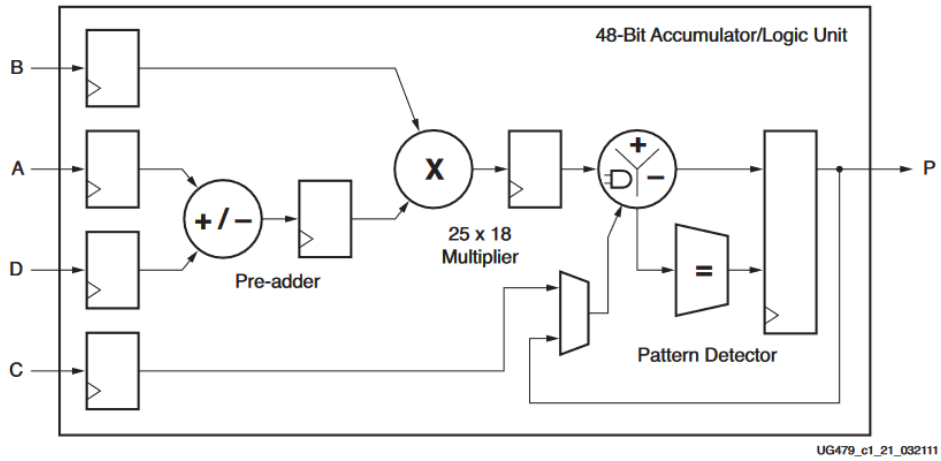
FPGA fejlesztés során viszonylag gyakran használunk *First-In-First-Out (FIFO)* tárolókat. Emiatt előnyös, hogy a 7-es FPGA családba tartozó blokk RAM egységeket FIFO tárolókként is tudjuk használni, anélkül, hogy ehhez egyéb logikai erőforrást fel kellene használni (pl. státusz és számláló jelekhez). Tehát a FIFO pufferhez tartozó státuszjeleket hardveresen megvalósítja az FPGA. Fontos, hogy a FIFO használata esetében is független órajeleket használhatunk a puffer két oldalán.

12.3.3 DSP blokk

Habár a CLB blokkban lévő logikai elemekkel ki tudunk alakítani különféle aritmetikai egységeket, nyilvánvalóan jobb megoldás dedikált aritmetikai blokkokat használni. A 12.11. ábrán látható *DSP blokk* a CLB egységektől függetlenül használható aritmetikai műveletekhez. Ahogy az ábrán látható, a DSP blokk tartalmaz egy 25 bit x 18 bites szorzóegységet, egy

Copyright © 2021 / BME AUT

előösszeadó blokkot, és egy többfunkciós egységet (jobb oldali, három részre bontott kör az ábrán). A többfunkciós egységgel megvalósíthatunk 48 bites akkumulátort, logikai függvényeket (pl. AND, OR, XOR műveletek), és SIMD műveleteket (pl. 2x24, 4x12 bites felosztásban) is. Az ábrán látható mintaillesztő rész, adott bitminta esetén jelez, amely például kerekítések, túlcsordulás jelzése esetén hasznos.



12.11. ábra – DSP blokk felépítése: a B port 18 bit, az A port 30 bit, a D port 25 bit, a C port és az eredményt tartalmazó P port pedig 48 bit széles[4]

A DSP blokkokat gyakran használják jelfeldolgozás esetén, pl. FIR szűrőhöz, Fourier transzformációnál. A blokk kialakítása támogatja az ilyen jellegű felhasználásokat. Ahogy az előző FPGA blokkok esetén, itt is tudjuk kaszkádosítani a DSP magokat, amelynek az előnye, hogy dedikált vezetékekkel tudjuk a DSP-ket összekötni.

A DSP blokk tekinthető egy egyszerű felépítésű ALU-nak is, hiszen az elvégzendő műveletet dinamikusan tudjuk változtatni. A 7-es FPGA család legnagyobb tagjai 12 000 DSP blokkot tartalmaznak, amely igen nagy számítási kapacitást jelent.

12.3.4 I/O blokkok

AZ FPGA az *I/O blokkok* segítségével tartja a kapcsolatot a külvilággal. Az I/O lábak alapvetően széles jeltartományt képesek használni, kezdve az 1,2 V-os jelektől, egészen a 3,3 V-os jelekig. Az FPGA mind a bemenő, mind a kimenő jelek esetén támogatja a differenciális meghajtást is, hiszen ez sok korszerű interfész esetén szükséges (pl. USB, PCI Express).

[4] Forrás: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf (2020. 05. 18.)

Copyright © 2021 / BME AUT

A nagysebességű jelek esetén akár a meghajtás impedanciáját is tudjuk vezérelni, ezzel helyettesítve a külső soros és párhuzamos lezárásokat. Ezt a dedikált *Digitally Controlled Impedance (DCI)* modulok segítségével tehetjük meg.

Az I/O blokkok tartalmaznak soros-párhuzamos, és párhuzamos-soros átalakító modulokat (*SERDES, DESERDES* blokkok). Ezek nagysebességű soros átvitelekhez használhatóak, és így az adatátvitel külső órajele lecsökkenthető az FPGA belsejében. Szintén a soros átviteleknél lehet szükség a különböző jelutak időben való eltolására. Erre szolgál a DELAY blokk, amellyel akár pikoszekundum nagyságrendben tudjuk az adott jelet időben eltolni.

A GT/s-es soros átvitelek támogatására dedikált I/O blokkokat tartalmaz az FPGA, és így plusz erőforrás nélkül tudjuk ezeket az interfészeket használni (pl. SATA, PCI Express, Ethernet, USB esetén). Például a tanult 8b/10b-es átalakításokat is elvégzik ezek a hard blokkok.

12.3.5 Általános összekötések, órajelhálózat

Az eddig bemutatott blokkok között számtalan összeköttetés található az FPGA-n belül. Ezek segítségével tudjuk a CLB logikai elemeit, a blokk RAM portjait, vagy a DSP jeleit összekötni. Néhány esetben dedikált utakat is találunk (pl. DSP, blokk RAM esetén), amelyek segítségével kisebb késleltetéssel tudunk összekötni blokkokat. Ebbe a kategóriába tartoznak a különböző blokkok órajelei is. Ráadásul a 7-es szériában található I/O-k esetén is megkülönböztetjük az órajelképes jeleket, amelyek képesek az FPGA-ban található órajelhálózatra rákapcsolódni.

Az FPGA tartalmaz PLL blokkokat is, amelyek segítségével képesek vagyunk különböző frekvenciájú, fázisú órajeleket előállítani. A beérkező órajel frekvenciája igen széles tartományban változtatható ezen PLL blokkok segítségével.

12.4 Szintézis FPGA esetén

A bevezető részben már említettük, hogy a szintézer az a program, amely elkészíti a HDL kódból az adott hardver leírását. Most röviden azt vesszük át, hogy FPGA esetén hogyan történik ez a folyamat. A szintézer ebben az esetben az előző részben bemutatott építőelemekre (pl. LUT-okra, tárolókra, blokk RAM-okra stb.) bontja le a kapott HDL kódot, és meghatározza a blokkok közötti összekötéseket is.

A szintézis folyamata után a következő lépés, hogy elhelyezzük a különböző blokkokat az FPGA-n belül. Ez azt jelenti, hogy például egy LUT esetén meg kell határozni, hogy pontosan melyik CLB, melyik LUT-ja feleljen meg az adott LUT-nak. Ezt a folyamatot nevezik elhelyezésnek, angolul *place*-nek. Ezenkívül a blokkok közötti huzalozást is ki kell alakítani, amelyet angolul *route*-nek neveznek. Ezt a két folyamatot összefoglalva implementációnak nevezik a Xilinx FPGA-k esetén, amelyet természetesen automatikusan elvégez a fejlesztőkörnyezet.

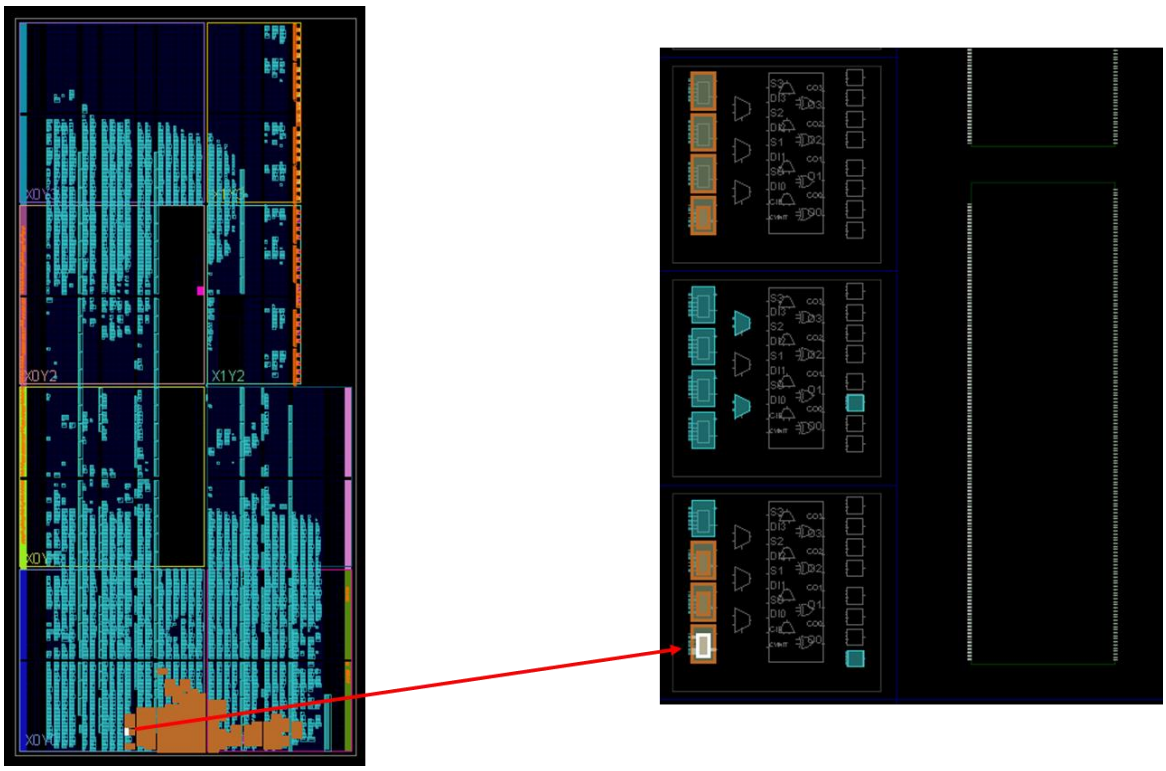
Az implementáció során a rendszer ellenőrzi az időzítések alakulását, és jelzi, hogy ha egy adott jel esetén nem teljesül valamilyen feltétel. Az implementáció során többféle szempont alapján

Copyright © 2021 / BME AUT

eljárhat a rendszer, például megpróbálhatja a legkisebb területre elhelyezni a blokkokat, optimalizálhat az alacsonyabb fogyasztásra, vagy a nagyobb teljesítményre (elérhető órajelre). Ezt a célt a fejlesztőkörnyezetben meg tudjuk adni.

A terv számos állapotában lehetőségünk van szimulációt készíteni a leírt HDL kódhoz. Akár a szintetizálás előtt, után, vagy az implementációt követően is. A folyamat minél korábbi állapotában készítünk szimulációt, annál gyorsabban végrehajtható, viszont ezzel párhuzamosan pontatlanabb is lesz a szimuláció a valósághoz képest. Például a szintetizálás előtti esetben csak a funkcionális leírás áll a szimulátor rendelkezésre, így a kapuk, jelek késleltetését nem tudja figyelembe venni. Implementációt követően viszont már lényegesen több információval rendelkezik, emiatt lesz pontosabb, de lassabb is a szimuláció.

A 12.12. ábrán látható, hogy egy adott terv esetén az FPGA mely erőforrásait használtuk fel, és kiemelve látszik egy konkrét LUT is. Miután az implementációt követően elkészült a végleges terv, fel kell programozni az FPGA-t, hogy a tervnek megfelelő működést kapjuk.



12.12. ábra – Egy elkészült FPGA terv által lefoglalt blokkok, kiemelve egy adott LUT: a lefoglalt erőforrás kék színnel láthatóak, és a sárga színnel jelzett erőforrások a terv egy adott modulját jelölik[5]

[5] Forrás: Xilinx Vivado 2019.2

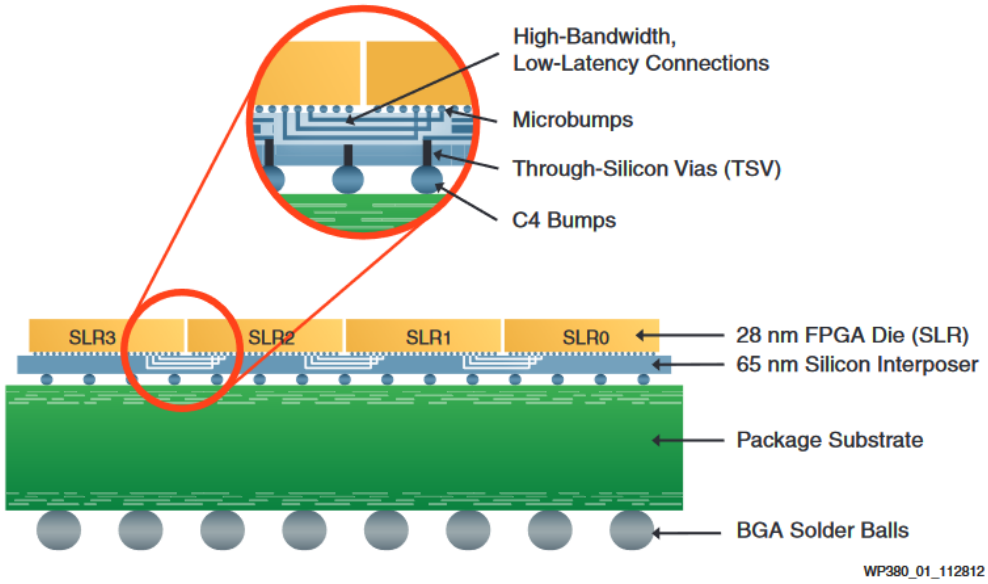
Copyright © 2021 / BME AUT

12.5 Legújabb fejlesztések

Most pedig átnézzük a közelmúltban megjelent legújabb fejlesztéseket és irányokat az FPGA alapú SoC rendszerek világából.

12.5.1 Xilinx SSI technológia

A Xilinx 7-es széria esetében a legnagyobb FPGA eszközöket úgynevezett 3D-IC technológia segítségével gyártják le. A Xilinx-nál a konkrét technológiát *Stacked Silicon Interconnect*-nek, rövidítve SSI-nek nevezik. Az elrendezés lényegét a 12.12. ábrán láthatjuk. Az SSI megoldásnál a szilícium hordozóra több FPGA lapkát is elhelyeznek egy köztes hordozó segítségével. A technológia előnye, hogy így a teljes költség alacsonyabb, mint ha egy nagy FPGA-t hoznának létre, amely az SSI-n található több FPGA kapacitásának megfelelő komplexitással rendelkezne.

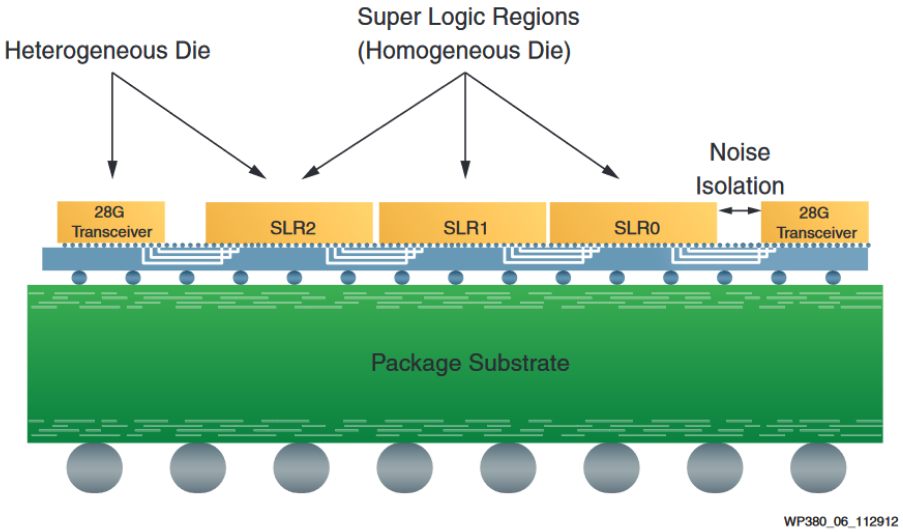


12.13. ábra – Xilinx SSI technológia[6]

Az SSI technológia segítségével nemcsak FPGA áramköröket lehet egy tokozáson belül létrehozni, hanem készítenek olyan heterogén IC-ket is, amely esetében például nagysebességű soros átviteli blokkokat helyeznek el (lásd 12.14. ábra). Ebben az esetben a zajelválasztás miatt is előnyös ez a kialakítás.

[6] Forrás: https://www.xilinx.com/products/silicon-devices/3dic.html (2020. 05. 18.)

Copyright © 2021 / BME AUT



12.14. ábra – SSI technológiával kialakított heterogén FPGA[6]

HBM memóriák

A heterogén FPGA-k esetében a GPU-knál már megismert *High Bandwith Memory (HBM)* modulokat is el tudnak helyezni egy tokozáson belül. A HBM memóriák előnye, hogy lényegesen nagyobb sávszélességű kapcsolat alakítható ki az FPGA maggal, mint a klasszikus DDR memóriák esetén. Ennek az az oka, hogy a rövidebb összeköttetés miatt kisebb a parazita kapacitások és induktivitások hatása.[7]

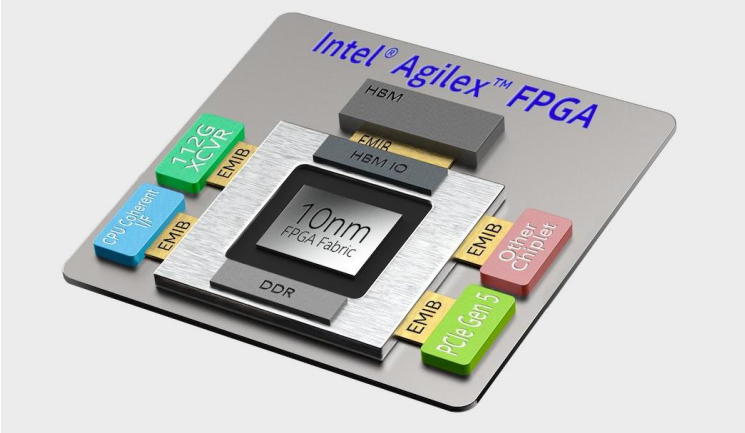
Mind a GPU-k, mind az FPGA-k esetében a memória sávszélességének növekedése elmaradt a többi perifériához viszonyítva (pl. az Ethernet tíz év alatt legalább egy nagyságrend növekedést ért el, míg a DDR memóriák esetén csupán kétszeres a növekedés). Emiatt fontos a HBM memóriáknál látott megoldás a sávszélesség növelésére.

12.5.2 Intel Agilex FPGA

A Xilinx-ot követő legjelentősebb FPGA gyártó, az Intel is készít hasonló, heterogén IC-ket. A legújabb, *Agilex*-nek nevezett termékük esetén szintén megjelennek a HBM memóriák, és a nagysebességű soros átviteli blokkok is (a 12.15. ábrán 112G XCVR jelöléssel látható a soros átviteli blokk).

[7] Forrás: https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf

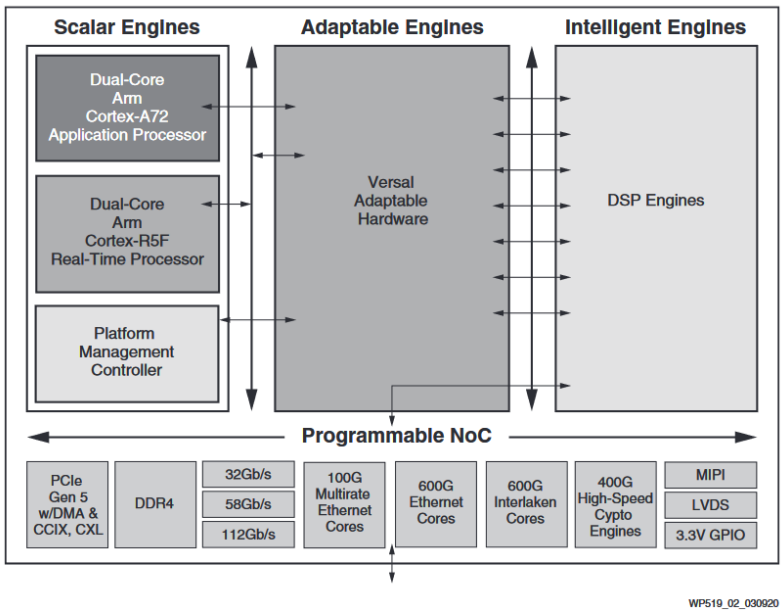
Copyright © 2021 / BME AUT



12.15. ábra – Az Intel Agilex FPGA megoldása[8]

12.5.3 Xilinx Versal platform

A Xilinx 2019-ben bemutatott legújabb fejlesztése szintén a 3D-IC technológiára épül, viszont itt már annyira különböző részekből épül fel az áramkör, hogy a gyártó nem is FPGA-nak nevezi azt, hanem bevezet egy új fogalmat, az *Adaptive Compute Acceleration Platform (ACAP)* megnevezést. A Versal ACAP esetében ráadásul a NoC elnevezést használják már az SoC helyett.



12.16. ábra – Xilinx Versal architektúra egy megvalósítása[9]

[8] Forrás: https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex.html (2020. 05. 18.)
[9] Forrás: https://www.xilinx.com/support/documentation/white_papers/wp519-versal-premium-intro.pdf (2020. 05. 18.)

Copyright © 2021 / BME AUT

A 12.16. ábrán látható a Versal platform egyik megvalósítása, amely alapvetően négy részből áll. A bal oldalon található egy több skalár processzorból álló rész, ARM Cortex-A72 és Cortex-R5F magokkal. A középső részen látható Versal Adaptable Hardware egy FPGA-s blokk, míg a jobb oldalon található DSP Engines részben vektoros végrehajtó egységeket találunk (GPU-hoz hasonló felépítés). Az ábrán látható alsó részben pedig különböző nagysebességű perifériák találhatók (pl. PCI Express 5, 112 Gb/s sávszélességű soros interfész).

12.5.4 FPGA alapú gyorsítókártyák

Az elmúlt években az FPGA alapú megoldásokat a különböző adatközpontok is elkezdték alkalmazni (pl. Microsoft, Amazon). Ezen a területen az FPGA-k elsősorban az alacsonyabb fogyasztásuk miatt előnyösebbek a CPU és GPU alapú megoldásokhoz képest. Ennek megfelelően az FPGA gyártók és külső cégek is FPGA alapú gyorsítókártyákat készítettek, amelyek segítségével egyszerűen gyorsítható egy meglévő számítógépes rendszer.

A 12.17. ábrán látható a Xilinx által fejlesztett *Alveo* elnevezésű gyorsítókártya. A termék PCI Express csatlakozóval kapcsolódik a számítógéphez, de sok esetben Ethernet porttal is rendelkeznek a kártyák. Szoftveres oldalról a megközelítés hasonlít a GPU-k világában megismert megoldáshoz. A PC-s program PCI Express porton leküldi az adatokat a kártyának, majd szintén ezen a porton kapja vissza az eredményt. Csak ebben az esetben a gyorsítást egy FPGA végzi a GPU helyett.



12.17. ábra – Xilinx Alveo gyorsítókártya[10]

[10] Forrás: https://www.designlinxhs.com/technology-support (2020. 05. 18.)

Copyright © 2021 / BME AUT

12.6 Különböző platformok összehasonlítása

A gyorsítókártyák esetén már előkerült, hogy az FPGA alapú rendszerek milyen előnyökkel rendelkeznek a CPU és GPU alapú platformokhoz képest. Most részletesebben megvizsgáljuk ezen megoldások közötti különbségeket, és azt, hogy mikor, melyik platformot érdemes használni.

Általánosságban elmondható, hogy amennyiben a nyers számítási teljesítményt vesszük alapul, akkor a GPU megelőzi mind a CPU, mind az FPGA teljesítményét. Nyilvánvalóan az eredmény függ az adott algoritmustól, és előfordulhat, hogy egy FPGA alapú megoldással jobb eredményt érünk el. Például abban az esetben, ha a feladat speciális struktúrát igényel, amelyhez a CPU vagy GPU architektúrája nem illeszkedik, az FPGA előnyösebb választás lehet a rugalmasabb kialakítása miatt.

Különböző mérések alapján kijelenthető, hogy egy CPU esetén a felhasznált energia kb. csupán 10%-a megy el az aktuális végrehajtásra, a maradék 90%-ot a processzor egyéb erőforrásai (angolul overhead) viszik el (pl. gyorsítótár, szuperskalár csővezeték stb.). Ezzel szemben egy hardveres megoldás esetén lényegesen jobb arány érhető el. Ez alapján általánosságban is kijelenthető, hogy az FPGA alapú megoldások alacsonyabb fogyasztást eredményeznek a CPU és GPU-t használó rendszerekhez képest.

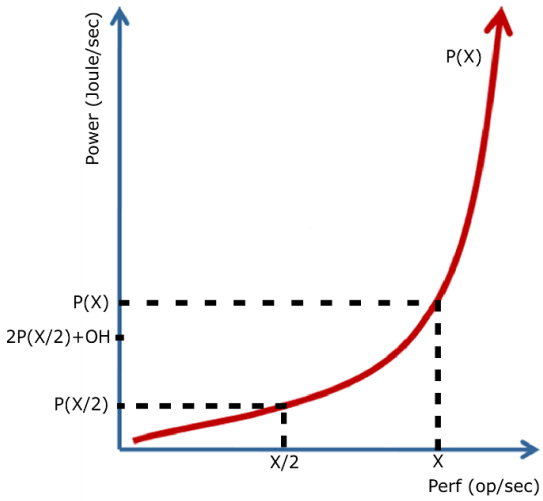
A processzoros rendszerek architektúrája más szempontból is hátrányosabb egy FPGA-s megoldáshoz képest. Egy processzoros rendszer esetén a bemenet megváltozásától a hozzátartozó kimenet megváltozásáig eltelt késleltetés nagyobb lesz, mintha ugyanezt a funkciót egyedi hardverrel oldanánk meg. GPU esetén ráadásul valamilyen módon el kell juttatni a feldolgozandó adatokat a GPU-hoz, majd az eredményt vissza kell juttatni a CPU-hoz. Ez a késleltetés elsősorban valós idejű rendszereknél fontos (gondoljunk például egy autóipari alkalmazásra).

Összességében az FPGA alapú rendszerek azokban az esetekben lehetnek előnyösek, ha a feladat speciális kialakítással rendelkezik, vagy fontos a rendszer késleltetése, illetve energiafelhasználása. Ezzel szemben a hardveres megoldások legnagyobb hátránya a lényegesen nagyobb fejlesztési idő. Emiatt próbálnak az FPGA gyártók olyan fejlesztőeszközöket kialakítani, amelyek esetén hardveres tapasztalat nélkül tudunk hardvert tervezni. Erre a következő részben látunk is majd példát.

12.6.1 A teljesítmény és a fogyasztás kapcsolata

Habár az előző részben a számítási teljesítményt és a fogyasztást külön vizsgáltuk, a két mennyiség nem választható szét, mivel nem függetlenek egymástól. A 12.18. ábrán látható a teljesítmény és fogyasztás kapcsolatára jellemző függvény (P(X)). Az ábra alapján megállapítható, hogy ha egy adott számítási teljesítményt szeretnénk elérni (például az ábrán jelölt X pont), ahhoz mekkora energiaigény tartozik. Viszont amennyiben a feladat párhuzamosítható, akkor megtehetjük, hogy az X/2 ponthoz tartozó végrehajtóegységet használunk két példányban. Ennek az előnye, hogy a függvény alakjából adódóan így kisebb energiát igényel a megoldás az X ponthoz képest, ugyanolyan számítási teljesítmény esetén.

Copyright © 2021 / BME AUT



12.18. ábra – A teljesítmény és a fogyasztás kapcsolata (P(X) függvény): az X egy adott teljesítményszintet jelöl, az OH pedig a párhuzamossítással járó overhead-et.

Az előző gondolatmenetet befolyásolja a párhuzamosítással járó esetleges többletenergia (overhead). Ennek hatását is figyelembe tudjuk venni, ha a kialakuló $2 \cdot P(X)$ energiaigényhez hozzáadunk egy konstans OH energiát. Ebben az esetben azt tettük fel, hogy a többletenergia független a többi vizsgált mennyiségtől.

A teljesítmény és a fogyasztás kapcsolata arra is rávilágít, hogy a többmagos processzoroknak az az egyik legnagyobb előnye, hogy a teljes rendszer fogyasztása így alacsonyabb lesz az egy magos megoldáshoz képest. Fontos kiemelni, hogy ez csak a párhuzamosítható feladatok esetén igaz.

A másik fontos következmény, hogy a különböző architektúrák vizsgálatánál nem elég csupán az egyik mennyiséget vizsgálni, hanem mindkét tényezőt számításba kell venni. Ennek hatására alakult ki a *teljesítmény/watt* mérőszám, amely azt mutatja meg, hogy adott számítási teljesítmény eléréséhez mekkora energia szükséges. Habár a watt használata az energiafelhasználásra kicsit félrevezető, a szakirodalomban mégis ez a fogalom terjedt el.

12.6.2 Esettanulmányok

Most pedig nézzünk meg néhány konkrét tanulmányt, amelyekben az FPGA, CPU és GPU alapú megoldásokat hasonlították össze.

Nvidia GTX285 GPU vs Xilinx Virtex 5

Az első, 2010-es cikkben[11] a szerzők egy Nvidia GTX285 GPU-t és egy Xilinx Virtex-5 alapú FPGA-s kártyát hasonlítanak össze. A cikkben elsősorban alapvető aritmetikai műveleteket (pl. mátrixszorzás, vektorok összeadása) vizsgálnak. Az elért eredmények szerint a GPU minden

[11] http://cas.ee.ic.ac.uk/people/ccb98/papers/DavidFPL10.pdf

Copyright © 2021 / BME AUT

esetben nagyobb számítási teljesítményt ér el, mint az FPGA-s megoldás. Például 32 bites vektorok összeadása esetén közel háromszor nagyobb teljesítményt ért el a GPU, mint az FPGA, de 64 bites vektorok összeadásánál már csak másfélszeres a különbség.

CPU, GPU vs FPGA

Egy másik 2010-es cikkben[12] hasonló összehasonlításokat találunk. Viszont ebben az esetben kétféle GPU-t vizsgáltak, és egy Virtex-6 FPGA-val hasonlították össze őket. Ráadásul a szerzők azt is megadták, hogy egy egyedi célhardverrel (ASIC) mekkora várható teljesítmény érhető el. Elsőként a mátrixszorzás algoritmusát nézték meg különböző platformon:

Intel Core i7	Nvidia GTX285	ATI R5870	Xilinx Virtex 6	ASIC
1,14	6,78	9,87	3,62	50,73

12.1. táblázat – Mátrixszorzás esetén elért teljesítmény/watt eredmények GFLOPS/J mértékegységben

A táblázatból látható, hogy a célhardverrel nagyságrenddel jobb eredményt érhetünk el a többi megoldáshoz képest. A második és harmadik helyen pedig a két GPU végzett. Amennyiben a gyors Fourier transzformáció (FFT) algoritmust nézzük, eltérő eredményeket kapunk:

Intel Core i7	Nvidia GTX285	ATI R5870	Xilinx Virtex 6	ASIC
0,71	4,2	-	6,5	90

12.2. táblázat – FFT esetén elért teljesítmény/watt eredmények GFLOPS/J mértékegységben

Látható, hogy ennél a feladatnál az FPGA jobb eredményt ér el, mint a GPU, de az ASIC továbbra is egy nagyságrenddel jobb a többi megoldásnál.

A cikk egyik konklúziója, hogy a legjobb eredmények a heterogén architektúrákkal érhetők el. Vélhetően hasonlóan gondolkodtak a Xilinx mérnökei is, amiker elkészítettek a legújabb Versal platformjukat, amely szintén heterogén architektúrára épül

Beágyazott CPU, GPU vs FPGA

Egy 2019-es mérésnél[13] pedig azt vizsgálták meg, hogy beágyazott környezetben milyen teljesítményt érhetünk el CPU, GPU és FPGA esetén. A vizsgált rendszer feladata különböző objektumok felismerése az úgynevezett YOLOv2 neurális hálózat segítségével.

[12] http://users.ece.cmu.edu/~jhoe/distribution/2010/micro10.pdf
[13] https://haltian.com/news/fpga-vs-gpu/

Copyright © 2021 / BME AUT

Platform	CPU	GPU	FPGA
Eszköz	ARM Cortex-A57	Nvidia Jetson TX2	Zynq Ultrascale+
Órajelfrek. [Hz]	1,9 GHz	1,3 GHz	0,3 GHz
Memória	32 GB	8 GB	21,1 Mb
FPS	0,23	1,48	35,71
Fogyasztás [W]	4	7	4,5
Telj. [FPS/W]	0,057	0,211	7,93

12.3. táblázat – YOLOv2 rendszer esetén elért teljesítmény/watt eredmények fps/W mértékegységben

Az eredmények alapján a Xilinx Zynq Ultrascale+ architektúra lényegesen jobb eredményt ért el, mint a beágyazott GPU. Az összehasonlítás alapja az elérhető képfrissítési frekvencia a fogyasztást is figyelembe véve. Tehát ez a mérték is megfelel az előző részben bevezetett teljesítmény/watt mérőszámnak.

12.7 Magas szintű szintézis (HLS)

A különböző platformok összehasonlításánál már említettük, hogy az FPGA-k egyik legnagyobb hátránya a többi architektúrához képest a hosszabb fejlesztési idő. Ez alapvetően abból adódik, hogy egy HDL kód segítségével alacsony szinten tudjuk leírni a megoldandó feladatot. Nagyjából ez annak felel meg, mintha szoftveres rendszereket assembly nyelven kellene megírnunk. A problémán egyrészt segít az IP alapú tervezés, valamint, ha magasabb szintű nyelven írjuk le a készítendő hardvert. Ez utóbbit nevezik *magas szintű szintézisnek*, angolul *High-Level Synthesis*-nek (*HLS*).

HLS nyelvek és eszközök fejlesztésével már a 90-es évek óta próbálkoznak. Általában igaz, hogy már meglévő, ismert programozási nyelvekből indulnak ki a fejlesztők (pl. C/C++), és nem saját nyelvet készítenek a magasabb szintű leíráshoz. Előfordul az is, hogy nem egy szöveges leírás alapján tudunk hardvert generálni, hanem valamilyen vizuális modell alapján. Erre jó példa a *MathWorks* cég *Matlab HDL Coder* eszköze, amely során lehetőségünk van Matlab függvényekből és Simulink modellekből HDL kódot generálni.

A két legnagyobb FPGA gyártó is rendelkezik saját HLS eszközzel. Az Intel saját eszközét *Intel HLS*-nek, a Xilinx eszközét pedig *Vivado HLS*-nek nevezik. Mindkettő esetében igaz, hogy C/C++ nyelvből tud HDL leírást generálni. Ebben a részben a Xilinx HLS eszközét nézzük át részletesebben.

12.7.1 Xilinx Vivado HLS

A Xilinx 2011-ben vette meg az AutoESL Design Technologies HLS technológiával foglalkozó céget, majd ezt követően jelent meg a saját HLS eszközével. A Vivado HLS alapvetően

Copyright © 2021 / BME AUT

C/C++/SystemC nyelvű forrásokat támogat, de a legújabb verzióban már az OpenCL nyelvet is használhatjuk.

12.7.1.1 Modulok

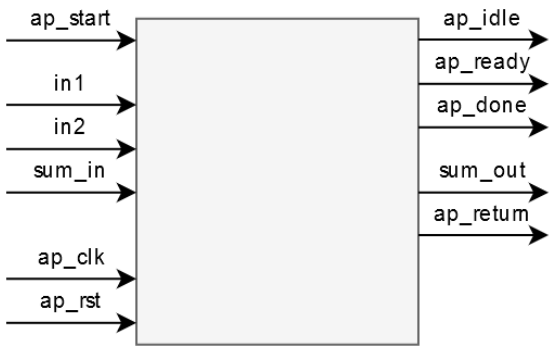
A Vivado HLS esetén alapvetően ugyanolyan C/C++ nyelvű függvényeket kell írnunk, mint a klasszikus, szekvenciális programozás során. Fontos, hogy minden függvényből a HLS egy külön hardver modult generál. A következő példában egy egyszerű összeadó függvényt láthatunk:

```
int adders(int in1, int in2, int *sum)
{
    int temp;

    *sum = in1 + in2 + *sum;
    temp = in1 + in2;

    return  temp;
}
```

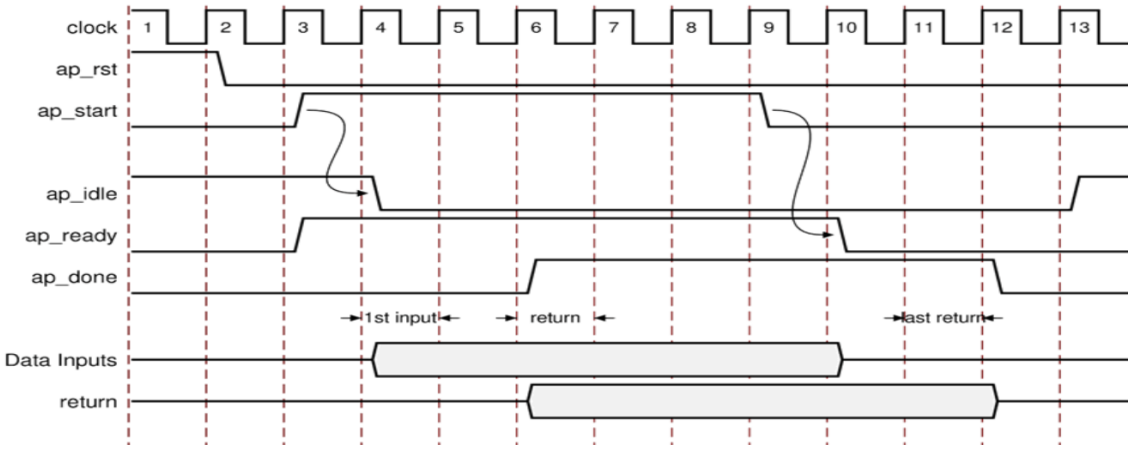
A kód alapján látható, hogy a modulnak alapvetően két bemenete és egy kimenete van, és rendelkezik egy pointer típusú változóval, amely egyszerre bemenet és kimenet is. Fontos kiemelni, hogy a HLS esetén a pointer típusokat alapvetően a példában látható célra használhatjuk, és például tömbök címzésére nem alkalmasak, ahogy a klasszikus programozásnál megszokhattuk.



12.19. ábra – Az előző HLS kódhoz tartozó modul

A kódrészlethez tartozó modult a 12.19. ábrán láthatjuk. Itt az előbb tárgyalt portokon kívül megjelenik egy órajel (*ap_clk*), és egy reset (*ap_rst*) bemenet is, valamint egyéb vezérlő jelek is. Ezeket a jeleket a Xilinx HLS automatikusan létrehozza. A járulékos jeleket a következő ábra segítségével mutatjuk be:

Copyright © 2021 / BME AUT



12.20. ábra – HLS modul vezérlése[14]

Az ábrán láthatjuk, hogy miután a modulhoz tartozó reset jel inaktív lesz, az *ap_start* bemeneti jellel tudjuk jelezni a modulhoz tartozó végrehajtás kezdését. Erre reagál a modul *ap_idle* kimenete, amellyel jelzi a modul, hogy elkezdte a végrehajtást. Az *ap_ready* jellel pedig azt jelzi a modul, hogy képes új bemeneti adatot feldolgozni. Ez a jel a mostani példában az *ap_start* aktív szintje alatt végig magas, tehát ebből is látható, hogy csővezetékes végrehajtás szerint működik a létrehozott modul (nem várja meg az első adat feldolgozását). Az *ap_done* kimenet a kimeneti adatok érvényességét mutatja.

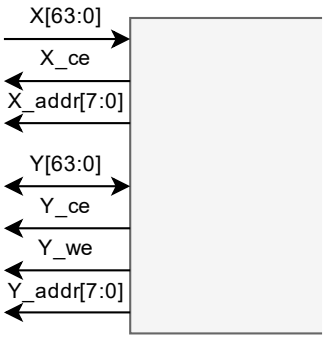
A 12.20. ábrán látható vezérlőjelektől eltérő megvalósítás is lehetséges, mi most csak egy lehetséges opciót mutattunk be. A bemeneti adatportok esetén viszont megnézzük még azt az esetet, amikor tömböt szeretnénk átadni a modulnak. Ehhez tekintsük a következő egyszerű példát:

```
#define N (256)
void foo(double X[N], double Y[N])
{
    for (int i=0; i<N; i++)
    {
        Y[i] = X[i] + Y[i];
    }
}
```

A kód alapján látható, hogy az X tömb esetén csak olvasni szeretnénk, viszont az Y tömböt írni is szeretnénk. A függvényhez tartozó portokat a 12.21. ábrán láthatjuk. Az egyszerűség kedvéért számos vezérlőjeleket az ábrán nem tüntettük fel.

[14] Forrás: http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Overview.pdf (2020. 05. 19.)

Copyright © 2021 / BME AUT



12.21. ábra – A foo függvényhez tartozó adatportok

Látható, hogy mivel double változót használtunk, így 64 bites adatbuszok alakultak ki, és mivel 256 elemű tömböket hoztunk létre, 8 bites címbusz keletkezett az adatokhoz. Az ábra alapján a tömbökhöz engedélyezőjelek is tartoznak (írásra külön vezérlőjel).

A fenti kódrészletben használt tömbök esetén alapvetően blokk RAM modult hoz létre a HLS szintézer, de lehetőség van FIFO használatára is. FIFO esetén természetesen nem jön létre címbusz.

A példában azt is láttuk, hogy a double típushoz 64 bit széles változót generált a rendszer. Felmerülhet a kérdés, hogy mit tudunk tenni, ha nekünk a C/C++ nyelvben nem szereplő bitszélességű változókra van szükségünk. Ebben az esetben az *ap_int* típust használhatjuk, ahogy a következő példában látható:

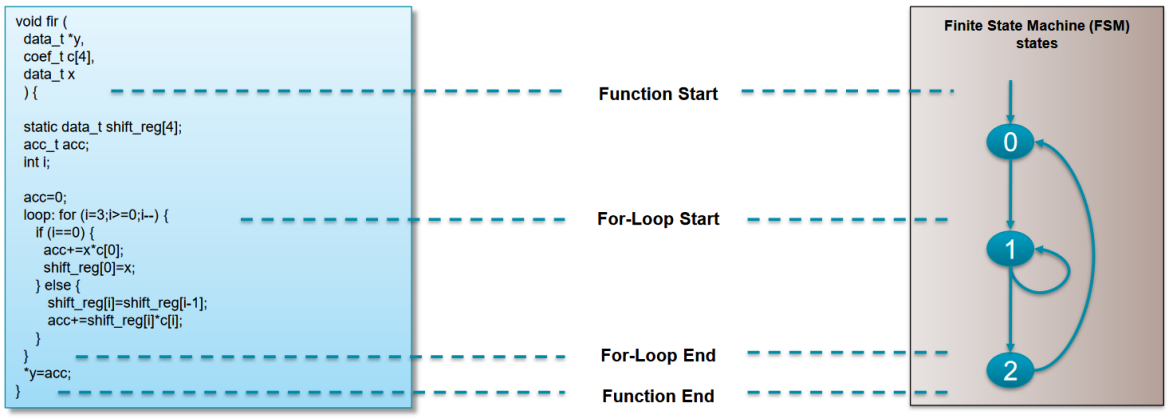
```
   ap_int<53> var;  //53 bites valtozo
```

12.7.1.2 Vezérlés

Miután megvizsgáltuk, hogy a létrejövő modulhoz milyen portok tartoznak, nézzük meg, hogy a modul belseje hogyan épül fel. Elsőként a modulhoz tartozó főbb részek vezérlését vizsgáljuk meg.

Alapvetően a Vivado HLS egy állapotgépet készít a kódban lévő utasítások, ciklusok alapján. Az állapotgépnek köszönhetően tudja a rendszer a szekvenciális forráskódot hardver segítségével leírni. Egy konkrét példát mutat be a következő ábra az állapotgép és forráskód kapcsolatára:

Copyright © 2021 / BME AUT



12.22. ábra – Egy függvényhez tartozó állapotgép felépítése[14]

Ahogy látható az ábrán, a for ciklushoz egy állapot tartozik. Tehát alaphelyzetben a ciklusokat a HLS nem változtatja meg, ugyanúgy ciklusként kezeli őket.

12.7.1.3 Ütemezés és megfeleltetés

A vezérlés kérdéséhez tartozik, hogy a C/C++ utasításokat hány órajel alatt végezze el a rendszer. Ezt nevezzük *ütemezésnek*, angolul *scheduling*-nek a HLS szintézis esetében. A *megfeleltetés*, angolul *binding* pedig azt vizsgálja, hogy milyen erőforrásokkal hajtjuk végre az utasításokat. A kérdést a következő rövid kódrészlet esetén vizsgáljuk meg:

```
void foo(...)
{
    t1 = a * b;
    t2 = c + t1;
    t3 = d * t2;
    out = t3 - e;
}
```

Látható, hogy alapvetően négy műveletet végzünk a fenti kódban. Alapvetően ezeket a műveleteket a modulhoz tartozó órajellel szinkron végezzük el. Az elsőként vizsgáljuk meg azt az esetet, hogy ha minden utasítást egy órajel alatt végzünk el. Ezt mutatja a következő ábra:



Ebben az esetben megtehetjük, hogy egy szorzó egységet használunk az első és a harmadik utasításnál lévő szorzás művelethez, ekkor viszont a szorzó előtt egy multiplexerrel kell eldönteni, hogy melyik konkrét műveletről van szó. Az összeadás és kivonás esetében is több lehetőségünk van. Használhatunk olyan aritmetikai egységet, amely mindkét utasítást el tudja végezni, de akár külön dedikált összeadó és kivonó áramkört is.

Copyright © 2021 / BME AUT

Az ütemezésre egy másik lehetőség, hogy egy órajel ciklus alatt mondjuk három műveletet is elvégzünk. Ezt mutatja a következő ábra:



Ebben az esetben mindenképp két szorzó egységet kell használnunk, viszont az összeadás és kivonás külön órajelben történik, így ott továbbra is választani tudunk a közös és a dedikált végrehajtók között.

A fent említett tervezési döntéseket makrók segítségével a forráskódból is befolyásolni tudjuk, de a Vivado HLS magától is meghozza ezeket a döntéseket. Alaphelyzetben arra törekszik a HLS szintézer, hogy a rendszer késleltetését minél alacsonyabb szinten tartsa. Természetesen mi megadhatjuk, hogy inkább az erőforrásokat szeretnénk minimalizálni, vagy esetleg a rendszer adat-áteresztőképességét maximalizálni.

12.7.1.4 Ciklusok szervezése

Ahogy a vezérlési állapotgépnél láttuk, alapvetően a Vivado HLS nem változtatja meg a ciklusok struktúráját. Ez azt jelenti, hogy a ciklus iterációi egymás után futnak le, ugyanazokat az erőforrásokat felhasználva. Viszont a ciklusmagban szereplő utasításokat párhuzamosan hajtja végre a rendszer, ha azok függetlenek egymástól. Ezt a ciklusszervezést feltekert, angolul *rolled* struktúrának nevezzük. Ezt a felépítést a következő példa szemlélteti:

```
void fir(float* y, float c[4], float x)
{
    static float shift_reg[4];
    float acc = 0;

    loop: for(int i=3; i>=0; i--)
    {
        if (i==0)
        {
            acc += x * c[0];
            shift_reg[0] = x;
        }
        else
        {
            shift_reg[i] = shift_reg[i-1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

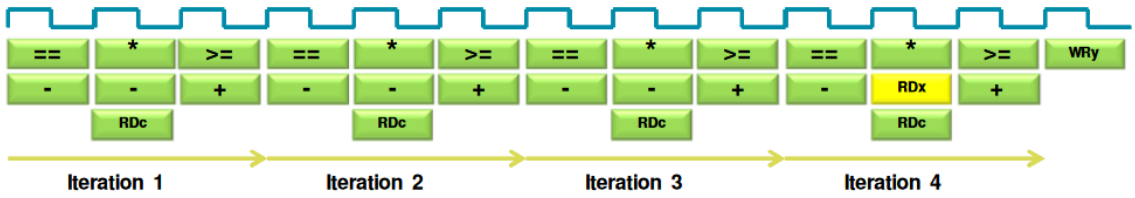
A fenti C-nyelvű függvény egy harmadfokú *Finite Impulse Response (FIR)* szűrő egy iterációját valósítja meg. Formailag a következő képletet oldjuk meg a kódrészlet segítségével:

Copyright © 2021 / BME AUT

$$y[n] = c_0x[n] + c_1x[n-1] + c_2x[n-2] + c_3x[n-3]$$

A kód egy adott n időpont és x bejövő jel esetén számolja ki az y értékét. A szűrésnél használt c_1, c_2, c_3 és c_4 együtthatók állandók.

A forráskóddal kapcsolatban fontos kiemelni, hogy a bemeneti *c* változót blokk RAM-ban tárolja a rendszer, viszont a *shift_reg* tömböt regiszterekkel valósítja meg. A kódhoz tartozó ütemezést a következő ábrán láthatjuk:



12.23. ábra – A *fir* függvényhez tartozó HLS megoldás rolled esetben[14]

A 12.23. ábrán látható, hogy milyen műveletekre képezte le a Vivado HLS a *fir* függvényt. Látható, hogy mind a négy iteráció ugyanazzal a két, párhuzamosan végrehajtott művelettel kezdődik. Ezek az *i* ciklusváltozóhoz tartozó 0-val való komparálás és dekrementálás műveletei. Az iterációk végén található >= művelet szintén az *i* ciklusváltozóhoz tartozik.

Az első három iteráció második órajelében történik meg a *c* tömb adott értékének beolvasása és szorzása a shift regiszter megfelelő tagjával (a shift regiszter is ebben a ciklusban shift-elődik). Látható, hogy az utolsó iterációban olvassa be az *x* változót a rendszer (RDx), és ez alapján történik meg a szorzás. A kimeneti *acc* változó frissítése minden iteráció utolsó órajelében történik meg, és a teljes ciklus után írjuk ki az *acc* értékét az *y* változóba.



12.24. ábra – A fir függvényhez tartozó HLS megoldás unrolled esetben[14]

Megadhatjuk a HLS szintézernek, hogy a ciklust bontsa szét, és ezáltal a benne található utasításokat amennyire lehetséges párhuzamosan hajtsa végre. Nyilván ez a megoldás lényegesen nagyobb erőforrással jár, viszont kevesebb órajelciklus alatt kapunk eredményt, és

Copyright © 2021 / BME AUT

nem kell a ciklusváltozóhoz tartozó műveleteket elvégezni, mivel nem lesz ciklus. Ezt a ciklusszervezést kitekert, angolul *unrolled* esetnek nevezzük.

Ha unrolled szervezést használunk, abban az esetben a 12.24. ábrán látható megoldást kapjuk a *fir* függvényhez. Látható, hogy így két órajel alatt a teljes függvényt végrehajtja a rendszer, és eltűntek az *i* ciklusváltozóhoz tartozó műveletek.

Felmerül a kérdés, hogy vajon miért szükséges két órajel, miért nem lehet a feladatot akár egy órajel alatt végrehajtani. A válasz az, hogy mivel a rendszer a *c* tömböt alapvetően duál port blokk RAM modulokkal valósította meg, így egy órajel alatt csupán 2 eleméhez fér hozzá. Emiatt egy órajel alatt csak két *RDc* utasítást tudunk végrehajtani.

Abban az esetben, ha megadjuk a szintézernek, hogy négy porton szeretnénk hozzáférni a *c* tömbhöz a következő megoldást kapjuk (ebben az esetben párhuzamosan, két blokk RAM-ot készít a tömbhöz):



12.25. ábra – A fir függvényhez tartozó HLS megoldás unrolled esetben, 4 portos RAM esetén[14]

Látható az ábrán, hogy most már minden utasítást végre tud hajtani egy órajel alatt az elkészült hardver. Természetesen emiatt az erőforrásigény magasabb lett, hiszen egy ciklus alatt négy szorzást és három összeadást is el kell végezni. Fontos megjegyezni, hogy azért tudja a rendszer egy órajel alatt végrehajtani a feladatot, mert a FIR algoritmus párhuzamosítható.

Azt is fontos megjegyezni, hogy abban az esetben, ha egymásba ágyazott, több szintű ciklusokat használunk, akkor ezeket a Vivado HLS alaphelyzetben kitekeri, és egyszintű ciklussá alakítja át. Ennek az az előnye, hogy a vezérlési állapotgépben így kevesebb állapotátmenet lesz.

Copyright © 2021 / BME AUT

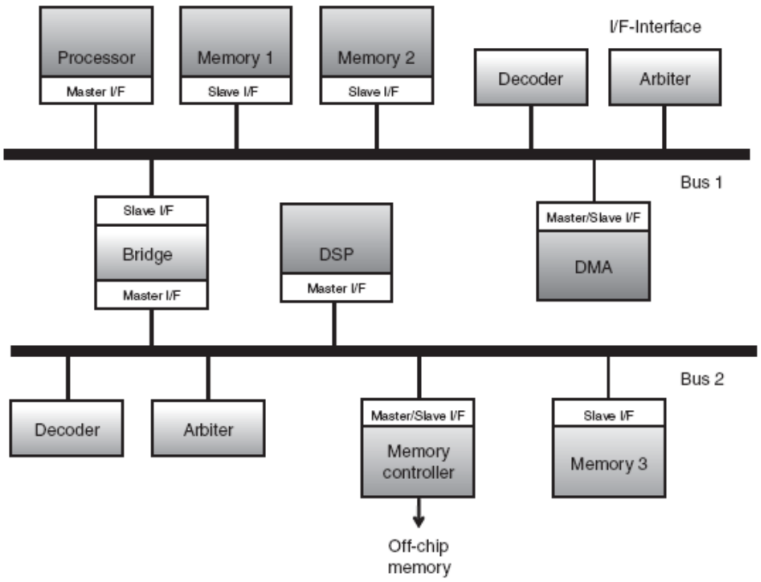
13 Lapkán belüli kommunikációs buszok

Az előző részekben többször említettük, hogy az IP alapú tervezés egyik legfontosabb motivációja az újrafelhasználhatóság. Viszont csak abban az esetben tudunk egy adott IP-t rövid idő alatt beüzemelni, ha a blokk csatolófelülete szabványos. Amennyiben a rendszerünkben található különböző IP-k más és más interfésszel rendelkeznek, akkor jelentős fejlesztési időt vesz el a különböző interfészek összekapcsolása. Emiatt alakultak ki szabványos interfészek és kommunikációs buszok az egylapkás rendszereken belül.

Ebben a fejezetben a kommunikációs buszokat vizsgáljuk meg, és bemutatjuk a leggyakrabban használt, *ARM Microcontroller Bus Architecture (AMBA)* buszcsaládot.

13.1 Buszrendszerek általános felépítése

Az általunk vizsgált buszok alapvetően adatvezetékekből, címvezetékekből és vezérlőjelekből épülnek fel. A vezérléshez tartozó jelek lehetnek például kérések, nyugtázások, átviteli mérettel kapcsolatos jelek.



13.1. ábra – Egy általános busz főbb komponensei[1]

A buszon található egységek között megkülönböztetünk *master* és *slave* modulokat. Minden esetben a master egység kezdeményezi a kommunikációt, és alaphelyzetben egy master és egy

[1] Forrás: https://www.engr.colostate.edu/~sudeep/publications/on-chip-communication-architectures-system-on-chip-interconnect/ (2020. 05. 19.)

Copyright © 2021 / BME AUT

slave egység kommunikál a buszon egy időben. Ezenkívül azt, hogy melyik master-nek van joga kommunikációt kezdeményezni, az úgynevezett *arbiter* modul dönti el valamilyen prioritás alapján. A megfelelő slave egységet pedig a *decoder* egység köti rá a buszra (sok busznál a decoder része a slave-nek). Különböző buszok között a *bridge* modul teremti meg a kapcsolatot, amely az egyik oldalon master-ként, a másik buszon slave egységként értelmezhető. A felsorolt egységeket mutatja be a 13.1. ábrán látható példa.

Az egylapkás rendszerekhez számos cég készít saját buszrendszert. Sok esetben ezek a specifikációk rugalmasak, bizonyos tulajdonságokat az IP integrátora maga dönthet el (pl. arbitrációs szabály, bitszélesség, órajelfrekvenciák). A legjelentősebb buszok, buszcsaládok felsorolás szintjén a következőek:

- ARM Microcontroller Bus Architecture (AMBA)
- Altera (Intel) Avalon
- IBM CoreConnect
- STMicroelectronics STBus
- OpenCores Wishbone
- Intel QuickPath
- AMD HyperTransport

A listán szereplő ARM AMBA a leggyakoribb kommunikációs buszcsalád, amelyet a következőben részletesebben is megvizsgálunk.

13.2 ARM AMBA buszcsalád

Az ARM által kifejlesztett AMBA buszcsalád tagjai közé számos busz tartozik, különböző alkalmazásokhoz. A teljesen nyílt AMBA családnak az 1996-os bevezetése óta öt verziója jelent meg, és kezdetben csupán két konkrét busz tartozott az AMBA-hoz. Minden verzió esetén igaz, hogy a konkrét időzítések és feszültségviszonyok nem jelennek meg a szabványokban, hiszen ezen tulajdonságok erősen függnek az alkalmazott gyártástechnológiától. Az is igaz az összes AMBA busz esetén, hogy nincsenek nagyimpedanciás jelek, hiszen gyakran erre nem ad lehetőséget a platform (pl. FPGA-k esetén csak I/O jelek lehetnek nagyimpedanciás meghajtásúak).

Fontos megemlíteni, hogy azonkívül, hogy az AMBA nyílt, tehát a specifikációja szabadon hozzáférhető, royalty-t sem kell a használata után fizetni.

Ebben a részben nem vesszük át a teljes AMBA szabványt, csupán néhány, gyakrabban alkalmazott buszt tekintünk át a családból.

Copyright © 2021 / BME AUT

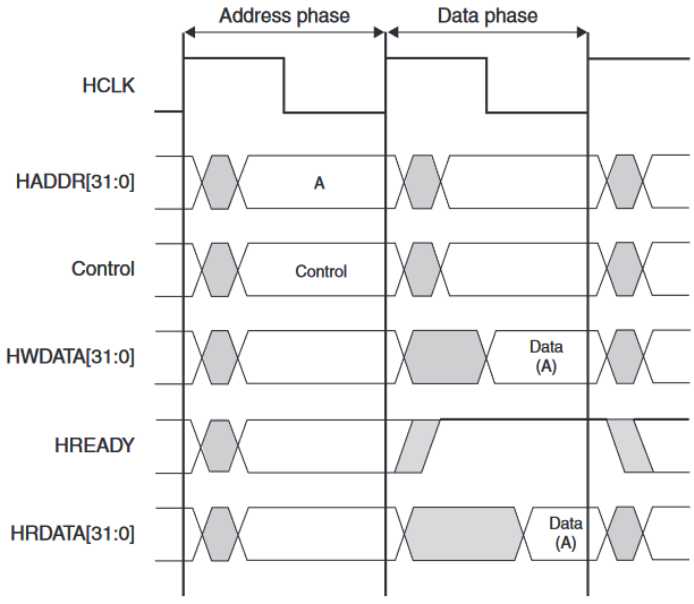
13.2.1 AHB busz

Az AMBA családba tartozó egyik legkorábbi nagy teljesítményű busz az *Advanced High-performance Bus (AHB)* volt. Habár az AHB-nél valamivel korábban jelent meg az *Advanced System Bus (ASB)* busz, az ARM az ASB-t ma már nem tanácsolja használni, így mi is inkább az AHB-t vesszük át elsőként.

Az AHB esetén az órajel (*HCLK* jelöléssel) mellett találunk címvezetéket (*HADDR*), master felől a slave felé érkező adatvezetékeket (*HWDATA*), slave felől a master felé tartó adatvezetékeket (*HRDATA*), és számos egyéb vezérlőjelet, amelyek közül csak a lényegesebbeket mutatjuk be. A jelek bitszélessége rugalmas, az adatvezetékek ajánlott minimális szélessége 32 bit, és ez egészen 1024 bitig megnövelhető. Az adatok bájtsorrendje sem kötött, mind little-endian, mind big-endian megoldás lehetséges. A címzéssel kapcsolatban fontos megemlíteni, hogy az AHB memóriába ágyazott busz, tehát a különböző egységeket azonos címtartományban érjük el (pl. memória egységet, perifériákat).

13.2.1.1 AHB tranzakciók

A legegyszerűbb esetben az írási vagy olvasási tranzakció az AHB buszon két órajelciklus alatt végbemegy. Ez látható a 13.8. ábrán. Fontos kiemelni, hogy az ábra mind az írási, mind az olvasási tranzakciót mutatja, de a valóságban ezek egyszerre nem mehetnek végbe, csak az egyszerűség kedvéért jelennek meg egy ábrán.



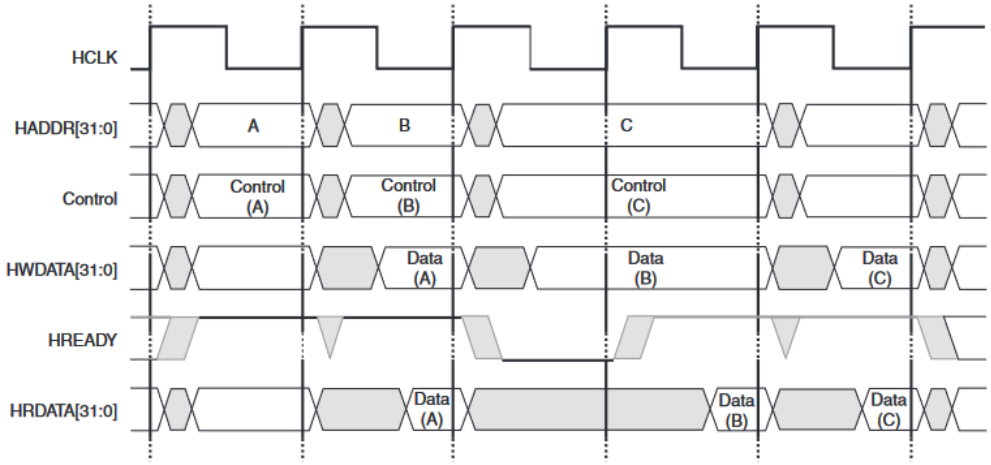
13.2. ábra – AHB átvitel írás/olvasás esetén (a valóságban nem lehet egyszerre írás és olvasási művelet)[1]

Az első órajelciklusban történik a slave egység címzése (címzési fázis), és a vezérlőjelek átvitele. A második ciklusban pedig a master elküldi az írni kívánt adatot a HWDATA porton, vagy a slave az olvasott adatot a HRDATA porton (adatátviteli fázis). Minden esetben a felfutó

Copyright © 2021 / BME AUT

élnél történik a jelek vezérlése és a mintavételezés is (természetesen nem ugyanannál a felfutó élnél). A slave egység a HREADY vezérlőjellel tudja késleltetni az adatok átvitelét (WAIT ciklus).

A 13.3. ábrán láthatunk is egy példát, amiben a slave egység késlelteti az átvitelt egy órajelciklussal. Ebben az esetben az is látható, hogy a különböző átvitelekhez tartozó címzési és adatátviteli fázisok egymást átlapolhatják, így csővezetékes módon minden órajelciklusban befejeződik egy átvitel. Az ábrán a *C* átvitel címzési fázisában láthatunk egy WAIT ciklust.



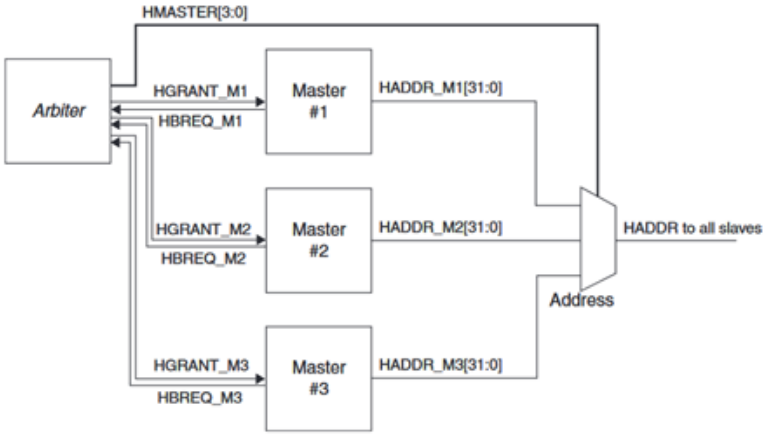
13.3. ábra – AHB csővezetékes átvitel WAIT ciklussal (a valóságban nem lehet egyszerre írás és olvasási művelet)[1]

13.2.1.2 AHB arbitráció, burst átvitel

Ahogy a bevezetőben említettük, az arbitráció folyamatában dönti el az arbiter egység, hogy a buszon található master-ek közül melyik kapja meg a buszvezérlés jogát. Az AHB busz esetén dedikált arbitrációs vezérlőjelek találhatók a master-ek és az arbiter között (lásd 13.4. ábra). Minden master-hez tartozik egy HBREQ_Mx vezeték, amelyen a buszvezérlési jogot tud kérni, valamint egy HGRANT_Mx jel, amelyen az arbiter jelez a master felé, hogy megkapja-e a jogot. Az AHB szabvány azt nem rögzíti, hogy milyen módszer alapján választ az arbiter a különböző master-ek között, ez implementációfüggő.

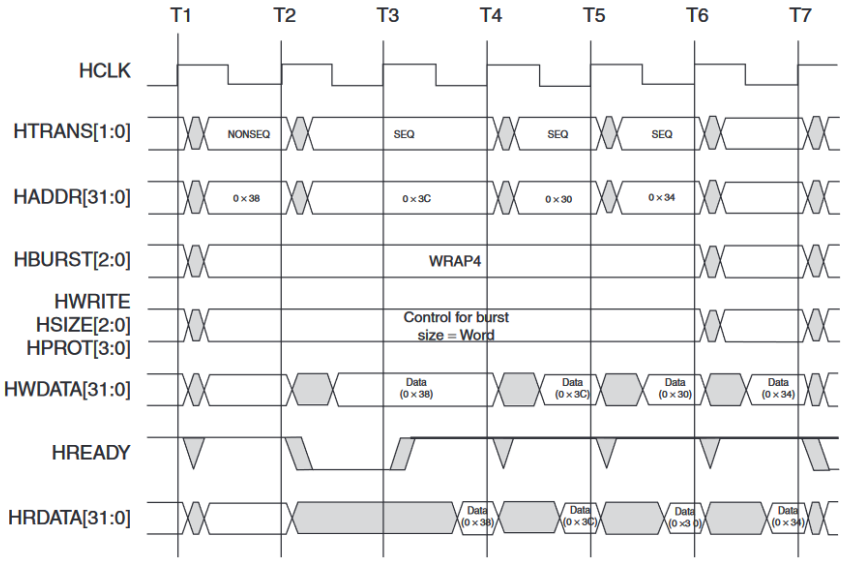
Mivel az AMBA családhoz tartozó buszoknál nincs nagyimpendanciás meghajtás, a különböző vezetékeket (pl. cím, adat, vezérlés) multiplexerek segítségével kell eljuttatni a különböző master egységektől a slave egységekig. Ezeket a multiplexereket az arbiter vagy a decoder vezérli. A 13.4. ábrán látható a címvezetékek esetén alkalmazott multiplexelés az arbiter segítségével.

Copyright © 2021 / BME AUT



13.4. ábra – AHB busz esetén az arbitrációhoz tartozó vezérlőjelek[1]

Mivel az arbitráció folyamata akár több órajelig is eltarthat minden tranzakció előtt, ez jelentős overhead-et jelent az átvitelek során. Emiatt találták ki a burst átvitelt az AHB esetén, amely során több adatátvitel is végbemegy egy tranzakció során. Fontos kiemelni, hogy alaphelyzetben a burst átvitelnél is megkaphatja más master a buszt, amennyiben magasabb prioritással rendelkezik. Ez alapján a burst átvitel akkor jelent előnyt, ha magasabb prioritású master nem kívánja megszerezni a buszt. Ez alól az úgynevezett nem megszakítható burst átvitelek jelentenek kivételt, mert ekkor semmilyen más master nem kaphatja meg a buszvezérlés jogát az átvitel során.



13.5. ábra – Burst átvitel AHB busz esetén (a valóságban nem lehet egyszerre írás és olvasási művelet)[1]

A burst átvitelre egy példát a 13.5. ábrán láthatunk. A HTRANS jel azt adja meg, hogy a burst átvitel első (NONSEQ), vagy azt követő (SEQ) ciklusáról van szó. A HWRITE vezérlőjel jelzi, hogy írási vagy olvasási átvitelről van szó (jelen esetben az ábra megint csak mindkettőt

Copyright © 2021 / BME AUT

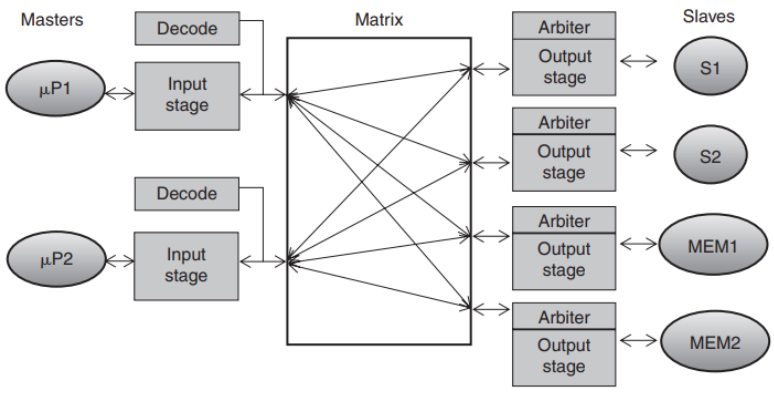
mutatja), a HSIZE az átvitt adatok méretét (pl. 32 bites széles HWDATA esetén is írhatunk csupán 8 bitet), és a HPROT egyéb, jelen esetben nem tárgyalt információt szolgáltat az átvitelről.

A 13.5. ábrán látható a HBURST port is, amely a burst típusát adja meg. Alapvetően használhatunk INCR4/8/16 típusokat, amelyeknél a megadott számú átvitel történik, a címek megfelelő növelésével. Az ábrán is szereplő WRAP4/8/16 esetén a címzés – a burst mérettől és a HSIZE értékétől függően – egy adott szóhatárnál átfordul. Az ábrán szereplő példa esetén (4 darab átvitel, 4 bájt-os HSIZE) a 16 bájtos szóhatárnál történik meg az átfordulás, és emiatt követi a 0x3C címet a 0x30-as cím.

A WRAP átvitelnek például a gyorsítótárak esetén van jelentősége. Képzeljük el, hogy egy adott master-nek (pl. processzor) szüksége van egy adott blokkra, amely nem található meg a gyorsítótárban. Emiatt a gyorsítótár betölti a megadott blokkot a memóriából, és annak környezetét (gyorsítótár sort). Tegyük fel, hogy a keresett elem a gyorsítótár sor utolsó eleme. Emiatt normál burst átvitel esetén a master-nek meg kellene várnia a teljes sor betöltését, hogy megkapja az elemet. Viszont WRAP átvitel esetén a gyorsítótár kezdheti a keresett elemmel a beolvasást, és így a master rögtön megkaphatja az elemet. Viszont ebben az esetben is betöltődik a teljes sor a gyorsítótárba, tehát hátránnyal nem jár a WRAP átvitel. A példában azt feltételeztük, hogy mind a master és a gyorsítótár között, mind a gyorsítótár és a memória között AHB busz található.

13.2.1.3 AHB topológiák

Az AHB buszt többféle topológia szerint ki lehet alakítani. Kialakíthatunk egy buszt, amelyhez csatlakoznak a master és slave egységek. Lehetőség van hierarchikus, több szintű buszra is, és bonyolultabb mátrix topológiára is. Ahogy a 13.6. ábrán is látható, a mátrix kialakítás előnye, hogy a két master egység egyidőben tud kommunikálni két különböző slave modullal.

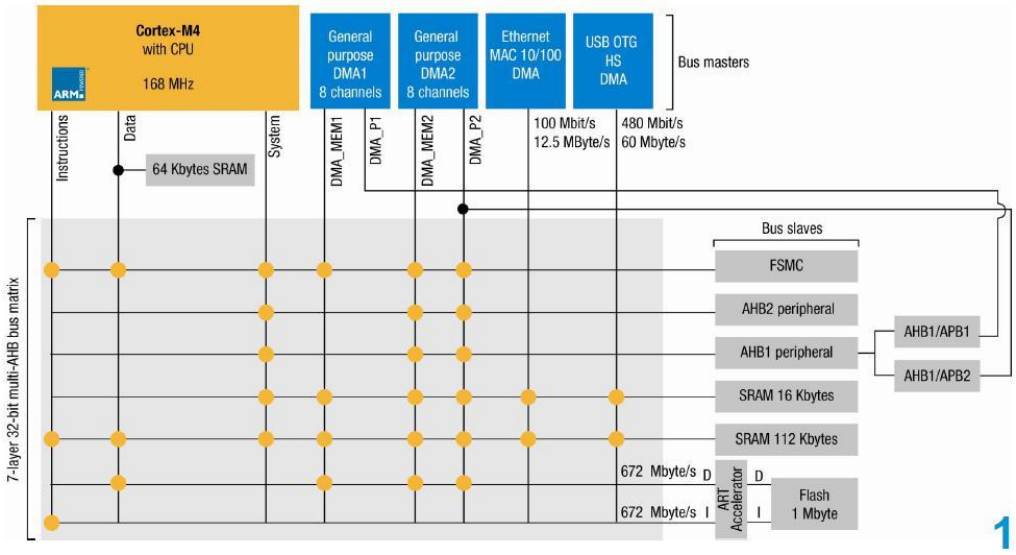


13.6. ábra – Két párhuzamos AHB busz mátrix topológiával[1]

Az AHB buszt gyakran alkalmazzák mikrovezérlők esetén, például az STMicroelectronics cég STM32 sorozatánál. A 13.7. ábrán látható az STM32F4-es mikrovezérlő család AHB mátrix

Copyright © 2021 / BME AUT

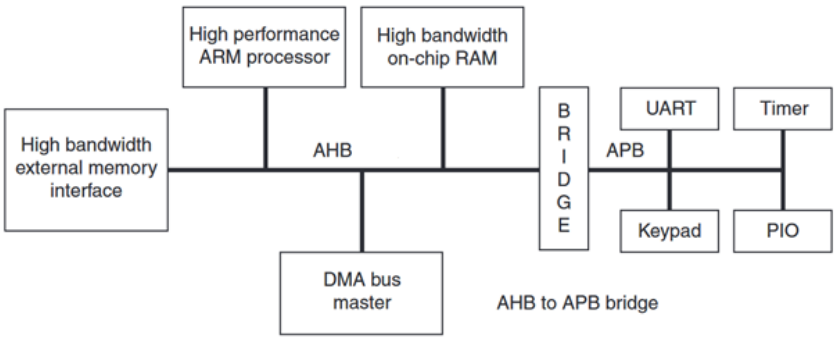
topológiája. Látható, hogy ebben az esetben is több master és slave egység csatlakozik a busz mátrixra.



13.7. ábra – STM32F4 mikrovezérlő AHB busz mátrix topológiája[2]

13.2.2 APB busz

Ahogy említettük, az AHB busz a nagy sebességű egységek összekötéséhez készült. A lassabb perifériák illesztéséhez az ARM egy külön buszt fejlesztett az AMBA család részeként, *Advanced Peripheral Bus (APB)* néven. A 13.8. ábrán látható egy példarendszer, amelyben az AHB a nagysebességű egységeket köti össze, az APB busz pedig az egyszerűbb perifériákat illeszti az AHB-hez egy bridge egységen keresztül.



13.8. ábra – Példarendszer AHB és APB buszokkal[1]

Az APB busz az AHB-hoz képest lényegesen egyszerűbb felépítésű, minden esetben egy master egység vesz részt az átvitelben, és nincs csővezetékes vagy burst átvitel. A master egység sok

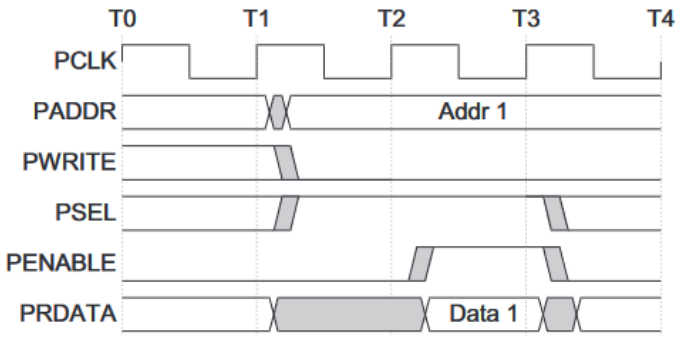
[2] Forrás: https://www.aut.bme.hu/Course/VIAUAC06 (2020. 05. 19.)

Copyright © 2021 / BME AUT

esetben az AHB-APB bridge, amely az AHB felől érkező átviteleket továbbítja az APB számára. A következőkben az APB busz legfontosabb jeleit vesszük át, amelyek a legelső APB verzió óta jelen vannak a szabványban.

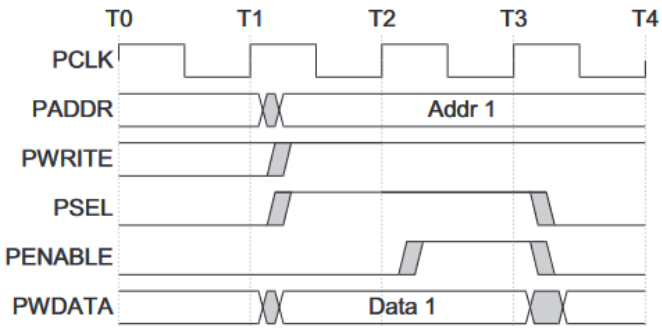
Az APB busznál használt jeleket a 13.9. ábrán látható olvasási átvitel során mutatjuk be. Az APB master a *PADDR* buszon jelzi az olvasni kívánt címet, és a *PWRITE* jelen adja meg, hogy olvasási átvitelről van szó. Az APB buszon lévő összes slave egységhez tartozik egy *PSEL* jel, és ezeken keresztül választja ki a master a megfelelő slave-et. Az olvasás esetén a slave egység a *PRDATA* buszra teszi ki az adatot, és a *PENABLE* jelen jelzi a tranzakció érvényességét.

Az APB átvitel során alapvetően három állapotot különböztetünk meg. Az *IDLE* állapotban nem történik kommunikáció, ez a 13.9. ábrán a T0 órajelciklusban látható. A T1-es ciklust *SETUP* állapotnak nevezik, amelyik során kiválasztásra kerül a slave, és a master kiteszi a buszra az átvitelhez tartozó címet. Az ezt követő, T2-es ciklusban a slave befejezi az átvitelt, ezt *ENABLE* állapotnak nevezi a szabvány. A jelen példában az ENABLE fázis egy órajelig tart, de ez akár több órajelet is igénybe vehet.



13.9. ábra – APB olvasási átvitel[1]

Az olvasáshoz nagyon hasonló az írási átvitel az APB esetén (lásd 13.10. ábra). Az egyik különbség, hogy a PWRITE jel magas állapota jelzi, hogy írásról van szó, és itt a slave a PENABLE jellel jelzi, hogy fogadta a master által elküldött adatot.



13.10. ábra – APB írási átvitel[1]

Copyright © 2021 / BME AUT

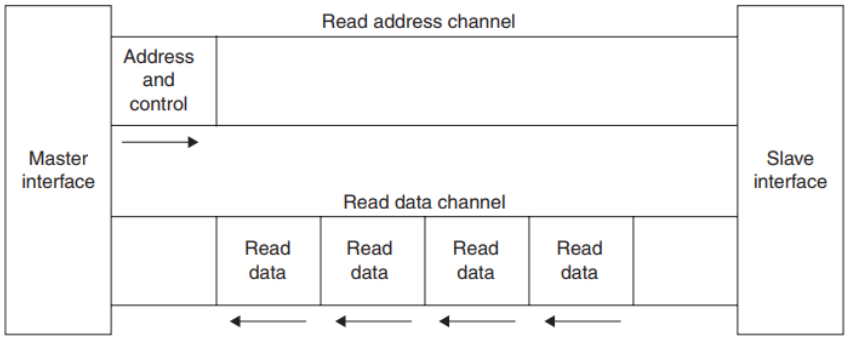
13.2.3 AXI busz

Az AMBA szabvány 3-as verziójában jelent meg egy új busz, *Advanced eXtensible Interface (AXI)* néven, amelyet az AHB busznál is nagyobb sávszélességű átvitelekhez készítettek. Az AXI busz felépítése az AHB-hez képest is rugalmasabb, így amellett, hogy nagyobb sebességre képes, sok esetben egyszerűbb infrastruktúrát igényel.

Az AXI egy pont-pont kapcsolatot valósít meg a master és slave egységek között. Az átvitelek alapvetően különböző csatornákon keresztül történnek. Az olvasási átvitelhez tartozik egy olvasási címcsatorna (vezérléseket is beleértve), és az adatok átviteléhez egy adatcsatorna. Az írási átvitel esetén három csatorna is használható, mivel a cím-, és adatcsatornákon kívül egy írási válaszcsatorna is megjelenik a szabványban. Fontos megemlíteni, hogy mivel az AXI felépítése rugalmas, így sok esetben összevonnak bizonyos csatornákat (pl. a két címcsatornát).

13.2.3.1 AXI átvitel

A master és slave közötti lehetséges olvasási átvitelt mutatja be a 13.11. ábra. Látható az olvasási címcsatorna és adatcsatorna. Valamint az ábra alapján az is megfigyelhető, hogy egy adott címcsomag hatására több adatcsomag is érkezhet a slave egységtől (a példa esetében négy csomag). Ezt az AHB-hez hasonlóan burst átvitelnek nevezzük.



13.11. ábra – AXI busz esetén az olvasási címcsatorna és adatcsatorna[1]

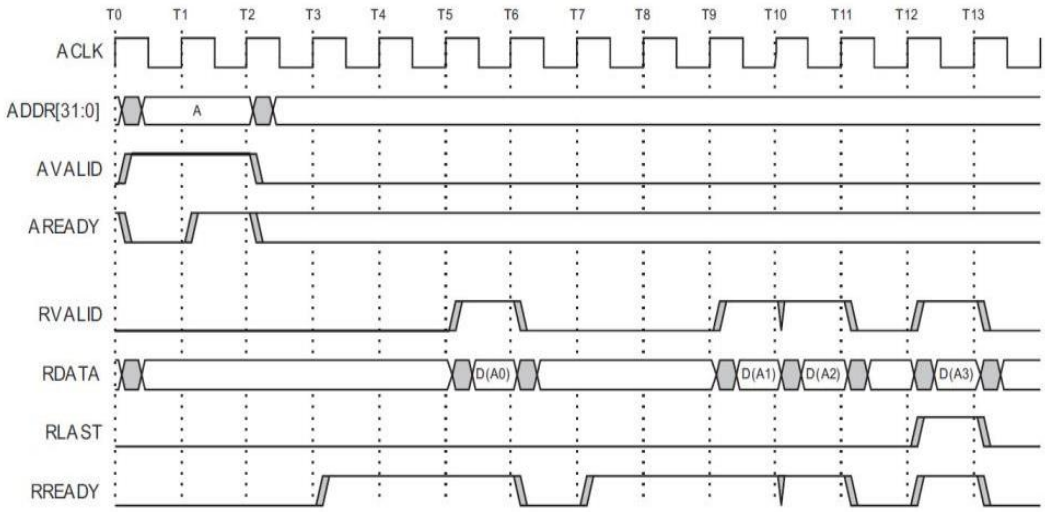
Az AXI olvasás átvitelt részletesebben a 13.12. ábra mutatja be. Fontos megemlíteni, hogy ez az ábra sem mutatja az összes jelet, csupán az átvitel során használt fontosabb jeleket.

Általánosságban igaz, hogy mindegyik csatorna esetén a *VALID* jel jelzi az adott csatornán átküldött információ érvényességét, és minden csatorna esetén társul a VALID jelhez egy *READY* vezeték, amely azt mutatja meg, hogy a másik fél képes fogadni az információt. Az átvitel akkor lesz sikeres, ha azonos órajelciklusban magas a VALID és a READY jel értéke. A példában az olvasási címcsatorna esetén a T1 ciklusban lesz érvényes a cím átvitele (a cím az ADDR buszon látható). A leírtakból következik, hogy bizonyos esetben a master, bizonyos esetben a slave hajtja meg a VALID, READY jeleket.

A 13.12. ábrán látható olvasási adatcsatorna nagyban hasonlít a címcsatornára, csak ebben az esetben cím helyett az adatokat látjuk, és a master helyett a slave az információ forrása. Az

Copyright © 2021 / BME AUT

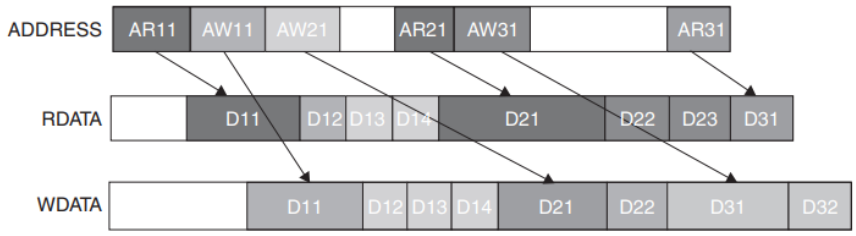
ábrán látható, hogy ebben az esetben megjelenik egy RLAST átvitel, amely a burst jellegű átvitel utolsó tagját jelzi. Ez minden adatcsatorna esetén megjelenik.



13.12. ábra – AXI olvasási átvitelnél szereplő fontosabb vezetékek[1]

Habár a 13.12. ábrán egy egyszerű átvitelt láthattunk, bonyolultabb esetek is lehetségesek az AXI busz esetében. A 13.13. ábrán látható egy példa, amely esetén szimultán írási és olvasási átvitelek követik egymást. Ráadásul ebben az esetben közös címcsatornát használ a példa. Az is látható, hogy az AHB-hoz hasonlóan itt is megjelenik a csővezetékes átvitel, tehát egy adott átvitel befejezését megelőzően már elindul egy következő átvitel.

A példában az is látszik, hogy különböző azonosítószámok, ID-k tartoznak az átvitelekhez, amellyel azonosítani lehet az átvitt információkat. Ugyanaz az ID jelenik meg a cím- és adatcsatornán az átvitelek során.



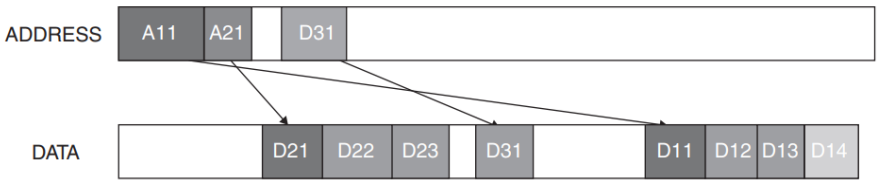
13.13. ábra – AXI szimultán írási és olvasási átvitel[1]

13.2.3.2 Speciális AXI képességek

Az AXI számtalan eszközt biztosít, amellyel növelhető a busz sávszélessége, vagy csökkenthető az átvitelek késleltetése. A késleltetés mértékét csökkenti a 13.14. ábrán látott sorrenden kívüli átviteli mód, angolul *out-of-order (OoO)* átvitel. A megoldás lényege, hogy amennyiben egy tranzakcióhoz tartozó adat átküldéséig jelentős időre van szükség, a slave-nek

Copyright © 2021 / BME AUT

lehetősége van a következő átvitelekhez tartozó adatokat átküldeni. Emiatt is lényeges, hogy ID-val lássuk el az átviteleket, hogy a master azonosítani tudja, hogy melyik átvitelhez tartozó adatot kapta meg (a sorrend az OoO átvitel miatt nem elég).



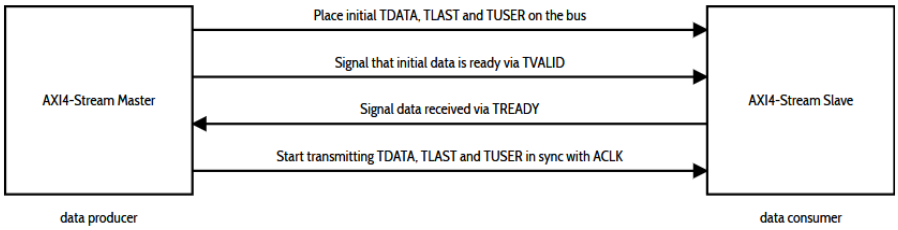
13.14. ábra – AXI out-of-order (OoO) átvitel[1]

Hasonlóan az AHB buszhoz, az AXI-nál is megoldható a nem megszakítható burst átvitelhez hasonló tranzakció. Az AXI esetén ez úgy történik, hogy a slave egység visszajelez a master számára, hogy az átvitelek közben történt-e másik master egységgel kommunikáció. Amennyiben a slave nem kommunikált más master-rel, akkor sikeres volt a tranzakció, különben sikertelen. Ezt az eljárást *szemafor átvitelnek* nevezik az AXI szabványban.

13.2.3.3 AXI változatok

Az AXI szabványnak létezik olyan változata, amely bizonyos korlátozások mellett, kevesebb erőforrást igényel. Ezt *AXI-Lite*-nak nevezik, és a legfontosabb korlátozás, hogy minden átvitel egy csomagból áll, tehát nincs burst tranzakció. Ezenkívül az átvitel során az adatbusz maximális bitszélességével megegyező méretben tudunk csak adatot küldeni, míg az AXI esetében lehetőség van kisebb szélességet használni. Az AXI-Lite más megkötéseket is tartalmaz az AXI-hoz képest, de ezekre most nem térünk ki.

Egy másik, gyakran használt változata az AXI busznak az *AXI-Stream*. Ebben az esetben már nem memóriába ágyazott buszról beszélünk, hanem adatfolyam jellegű átvitelről, címzés nélkül. Ennek megfelelően az AXI-Stream megfelel az AXI írási adatcsatornájának, ahol korlátlan hosszú burst átvitellel tudunk adatokat átvinni (lásd 13.15. ábra). Az AXI-Stream busz jól alkalmazható például kamerás rendszerek esetén, a szenzortól érkező pixelinformációk továbbítására. Ebben az esetben a különböző perifériák AXI-Stream-en fogadják a pixeleket, elvégeznek valamilyen műveletet rajtuk, majd egy másik AXI-Stream-en továbbítják a módosított pixeleket. A későbbiekben erre még látunk példát.



13.15. ábra – AXI-Stream átvitel két egység között: a TUSER-en tetszőleges kiegészítő információt tudunk küldeni, a többi vezeték megfelel az AXI-nál bemutatott jeleknek[1]

Copyright © 2021 / BME AUT

14 Processzormagok SoC rendszerekben

Korábbiakban már említettük, hogy állapotgépek segítségével tudunk szekvenciális logikát leírni. Például kommunikációs interfészek (pl. UART, I2C, SPI) esetén találkozunk gyakran ezzel a megoldással. Bonyolultabb eljárások esetén viszont adódik a lehetőség, hogy egy processzormagot használjunk a túlságosan komplex állapotgépek helyett. Ennek előnye lehet a rövidebb fejlesztési idő is, hiszen sok esetben egy szoftveres megoldással gyorsabban meg tudjuk oldani a feladatot, főleg, ha rendelkezésünkre állnak kész szoftveres könyvtárak.

Ebben az fejezetben megvizsgáljuk, hogy milyen módszerekkel tudunk processzormagot elhelyezni egy FPGA alapú SoC rendszerben, és megnézzük a legfontosabb processzor IP megvalósításokat.

14.1 Soft processzorok

A 11.3.1. fejezetben bemutattuk, hogy alapvetően hard és soft IP típusokat különböztethetünk meg. A processzor IP-k esetén is beszélhetünk soft megvalósításról, amikor is egy HDL leírást kapunk a processzorhoz, és ezt az FPGA általános építőelemeivel (pl. LUT, FF, blokk RAM) valósítjuk meg. A hard processzorok esetében viszont az áramkör dedikált processzormagot tartalmaz, amely nem foglalja el az FPGA általános erőforrásait. Egy mikrovezérlő szintén hard processzor IP-t tartalmaz (leggyakrabban ARM által tervezett magot), a különböző periféria és memória egységek mellett.

Viszonylag sokféle soft CPU mag létezik, kezdve az FPGA gyártók által fejlesztett IP-ktől, egészen a nyílt forráskódú akadémiai processzorokig. Ebben a részben először a leggyakrabban használt, ARM által fejlesztett processzormagokat nézzük át, majd áttérünk az FPGA gyártók processzor megoldásaira, és végül néhány nyílt forráskódú CPU IP-t is bemutatunk.

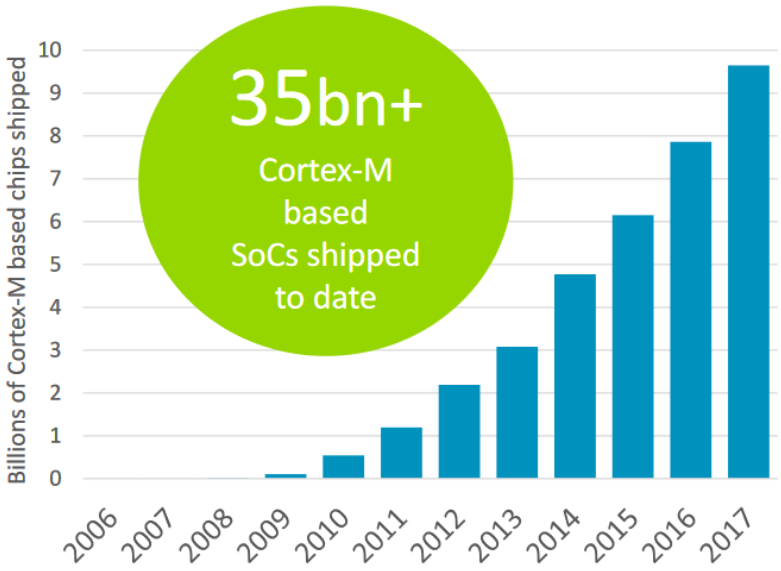
14.1.1 ARM Cortex-M IP-k

Az *ARM Cortex-M* processzorcsalád több tagját is használhatjuk FPGA alapú fejlesztés során. Ezeket a CPU magokat évente több milliárd eszközben használják (lásd 14.1. ábra), így számtalan szoftver és segédeszköz áll a rendelkezésünkre a fejlesztéshez.

A legkisebb processzor, a Cortex-M0 esetén az ARM ingyen elérhetővé tette az IP-hez tartozó Verilog forráskódot[3], amelyet prototípus fejlesztéshez ingyen igénybe vehetünk. Amennyiben célhardvert készítünk a processzormag felhasználásával, akkor már nem ingyenes az IP.

[3] Megjegyzés: A Verilog forráskódot úgy alakította át az ARM, hogy az nem alkalmas arra, hogy megértsük a processzor belső működését, csupán arra használható, hogy felhasználjuk a tervünkben.

Copyright © 2021 / BME AUT



14.1. ábra – ARM Cortex-M processzorok eladási számai[4]

A Cortex-M0-nál nagyobb teljesítményű Cortex-M1 és Cortex-M3 processzorok esetén pedig a Xilinx és az ARM közötti együttműködés alapján férhetnek hozzá ingyen az IP-hez a Xilinx FPGA-k felhasználói. Ezeknél az eszközöknél nem kapjuk meg a forráskódot, hanem grafikus módon tudjuk beállítani az IP-t. A beállítások tekintetében viszonylag kevés opciónk van, például a megszakítások számát és a megszakításhoz tartozó prioritások számát adhatjuk meg, vagy a hibakeresési funkciókat tudjuk beállítani.

14.1.2 Xilinx PicoBlaze

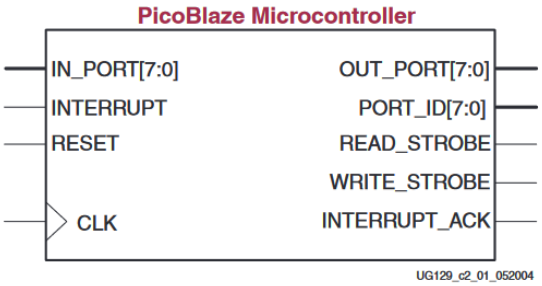
Az ARM CPU magok esetén azt láttuk, hogy alapvetően nincs sok lehetőségünk az IP konfigurációjára. Amennyiben a rugalmas kialakítás fontos számunkra, előnyösebb választás lehet az FPGA gyártója által készített soft processzor megoldásokat használni. A Xilinx esetén időrendben az első soft processzor a *PicoBlaze* volt.

A PicoBlaze egy 8-bites RISC architektúrára épülő processzor, amely könnyedén használható, és a belső felépítése is egyszerű. Az egyszerűsége miatt akár még az FPGA eszközöknél egyszerűbb felépítésű CPLD-k esetén is használható. Egy 7-es sorozatú Xilinx FPGA esetén mindössze 26 db SLICE és 1 db blokk RAM erőforrást igényel.

A PicoBlaze tipikus interfésze a külvilág felé a 14.2. ábrán látható. Az órajel (*CLK* jelöléssel) és a reset bemeneten kívül láthatunk egy megszakítás vonalat (*INTERRUPT*) és egy hozzá tartozó nyugtázó jelet (*INTERRUPT_ACK*). Ezenkívül egy-egy 8 bites be- és kimeneti porttal rendelkezik a processzor (*IN_PORT* és *OUT_PORT*), amelyekhez egy címport is tartozik (*PORT_ID*). A portokhoz tartozik egy írási és olvasási vezérlőjel is (*READ_STROBE, WRITE_STROBE*).

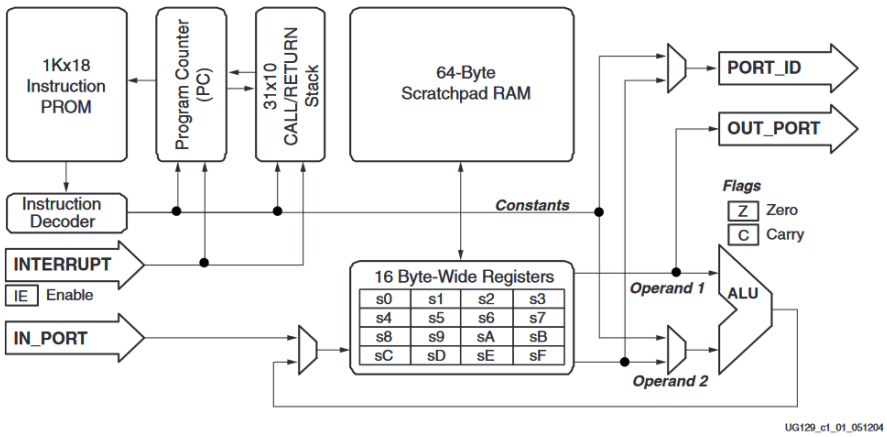
[4] Forrás: Xilinx

Copyright © 2021 / BME AUT



14.2. ábra – A PicoBlaze IP interfésze[5]

A PicoBlaze belső felépítése a 14.3. ábrán látható. A processzor utasításai minden esetben 18 bitesek, így az ábrán látható 1Kx18 utasítástárolóban összesen 1024 utasítás fér el (bizonyos konfigurációban kibővíthető az utasítástároló).



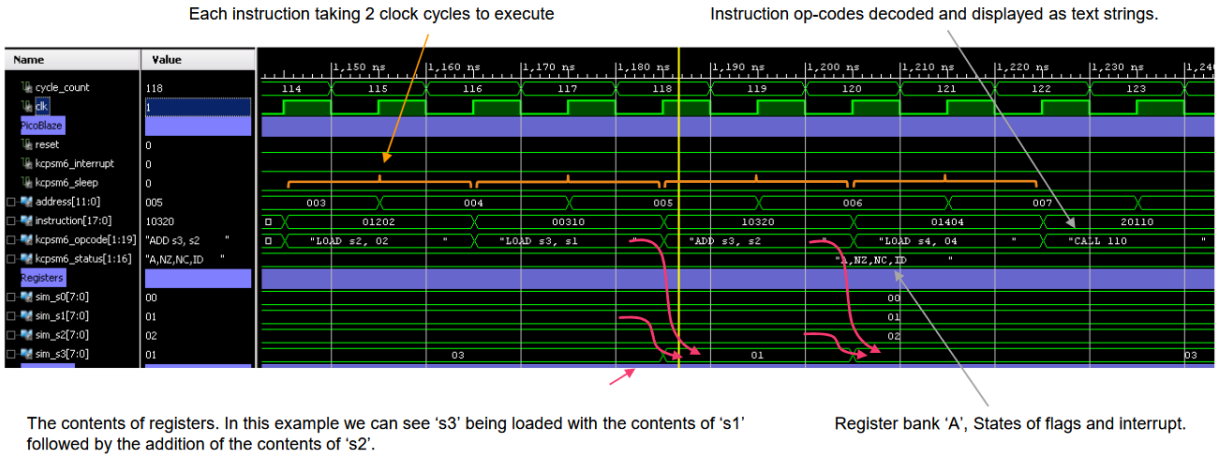
14.3. ábra – A PicoBlaze belső felépítése[5]

Minden utasítás végrehajtása két órajelciklust igényel, és az utasítások összesen 16 db általános célú regisztert használhatnak. A RISC architektúrának megfelelően minden utasítás felhasználhatja bármelyik regisztert. A regisztereken kívül felhasználható a processzorban lévő 64 bájt nagyságú RAM is.

Az elérhető maximális órajelfrekvencia a PicoBlaze esetén kb. 250 MHz. Ez nagyjából 100 MIPS maximális teljesítménynek felel meg. A 8 bites CPU-k között ez a teljesítmény nem rossz érték, de egy 32 bites ARM Cortex-M3 CPU a 100 MIPS értéket már 100 MHz frekvencia esetén eléri.

[5] Forrás: https://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf (2020. 05. 20.)

Copyright © 2021 / BME AUT



14.4. ábra – A PicoBlaze utasításainak szimulálása[6]

A PicoBlaze esetén általában assembly nyelvű kódot készítünk, az általánosabb, magasabb szintű nyelveket (pl. C) nem támogatja a processzor. Az assembly kódból a processzorhoz tartozó PC-s alkalmazás segítségével készíthetjük el az utasítástároló tartalmát, és így az FPGA felprogramozása során az általunk írt program belekerül az utasítástárolóba.

Arra is lehetőségünk van, hogy az elkészült programot szimuláció segítségével kipróbáljuk. A 14.4. ábrán látható szimulációs diagrammon látszik, hogy milyen utasításokat hajt végre a processzor, és hogyan alakul ezek hatására a regiszterek tartalma.

14.1.3 Xilinx MicroBlaze

A PicoBlaze-zel kapcsolatban a legfőbb kritika az alacsony teljesítmény lehet, hiszen egy nagyobb FPGA esetén bőven van elegendő erőforrás nagyobb teljesítményű CPU-ra is. Ezt felismerve a Xilinx fejlesztett egy 32 bites, RISC soft processzort, *MicroBlaze* néven. A PicoBlaze-hez hasonlóan, a MicroBlaze-t is testre tudjuk szabni a saját igényeink szerint, de léteznek konkrét összeállítások is az egyszerűbb használat érdekében. Az előre elkészített konfigurációk hasonlítanak az ARM-nál látott Cortex verziókra (pl. Microcontroller, Real-Time, Application).

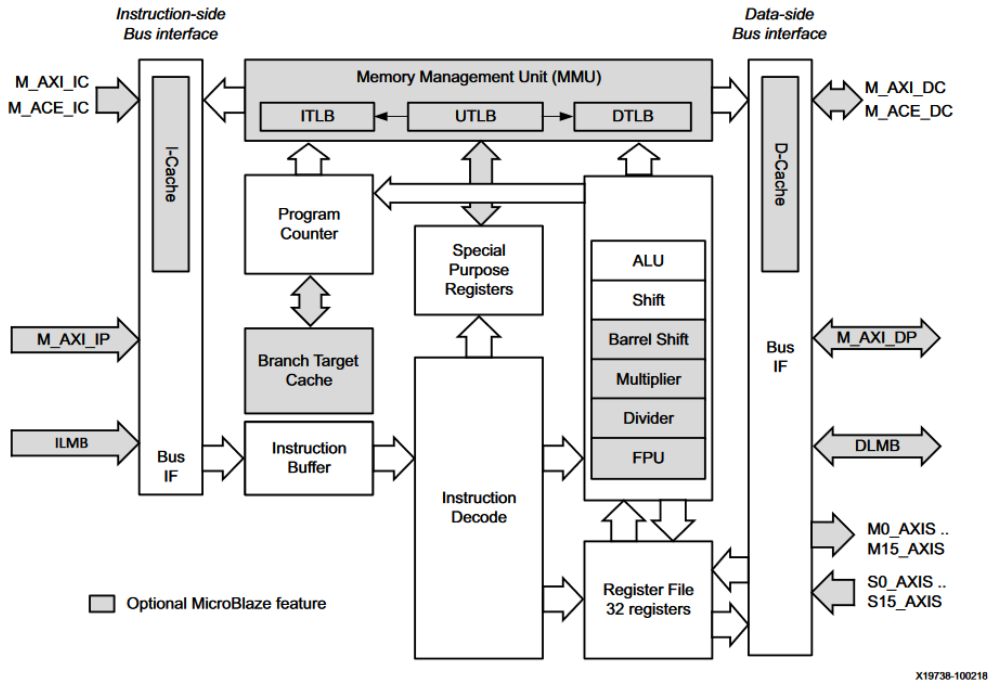
14.1.3.1 Belső felépítés

A 14.5. ábrán látható a MicroBlaze belső felépítése. Mivel a legtöbb komponens opcionális, emiatt tudjuk az IP-t az egyszerűbb mikrovezérlőtől a Linux operációs rendszert futtató komolyabb CPU-ig bezárólag használni. Szintén megfigyelhető, hogy viszonylag sokféle busz csatlakozhat a CPU különálló adat és utasítás interfészéhez. Ezeket a buszokat a következőkben még részletesen tárgyaljuk.

[6] Forrás: Xilinx KCPSM6 User Guide

Copyright © 2021 / BME AUT

A 14.5. ábrán, az interfészek között található egy opcionális memóriavezérlő egység. Ennek a blokknak a legfontosabb feladata, hogy a virtuális memóriacímeket fizikai címekké alakítsa át. Az ábra többi részegységével a korábbi fejezetekben bemutatott processzoroknál már találkoztunk, csupán a *Branch Target Cache* modul *Branch Target Buffer* (BTB) néven szerepelt. Ennek az egységnek a feladata a feltételes ugró utasítások kezelése, a korábban bemutatott *elágazásjövendölés*.



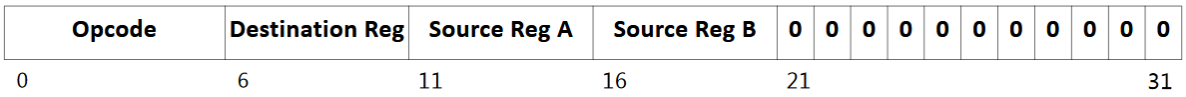
14.5. ábra – A MicroBlaze architektúrája[7]

Kezdetben a MicroBlaze-t csak 32 bites változatban lehetett használni, viszont később 64 bites változata is megjelent. Ennek megfelelően a processzor interfészei és regiszterei lehetnek 32 vagy 64 bitesek is, viszont az utasítások hossza mindkét esetben 32 bit.

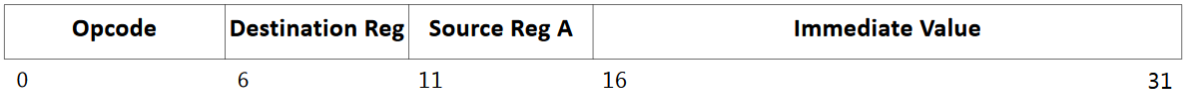
Az utasítások alapvetően két típusba sorolhatók. Az *A típusú* utasítások esetén két forrás-, és egy célregisztert tudunk használni, ahogy az a 14.6. ábrán látható. A *B típusú* utasítások esetén az egyik forrás regiszter helyett egy 16 bites konstans operandust használhatunk. A B típus esetén egy külön utasítással (IMM utasítás) megoldható, hogy akár 32 bites konstans értéket is használjunk.

[7] Forrás: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug984-vivado-microblaze-ref.pdf (2020. 05. 21.)

Copyright © 2021 / BME AUT



14.6. ábra – A MicroBlaze A típusú utasításainak felépítése[7]



14.7. ábra – A MicroBlaze B típusú utasításainak felépítése[7]

A következő táblázatban látható néhány konkrét utasítás felépítése:

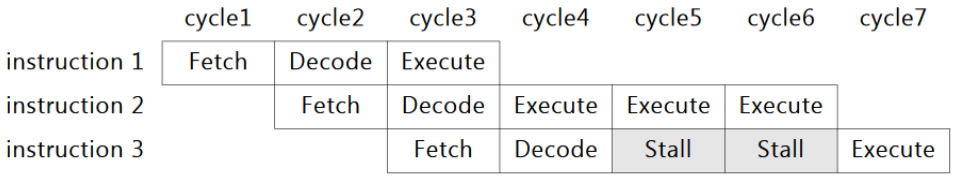
Utasítás	0-5. bit	6-10. bit	11-15. bit	16-20. bit	21-31. bit	Leírás
ADD Rd,Ra,Rb	000000	Rd	Ra	Rb	0...0	Rd := Rb + Ra
RSUB Rd,Ra,Rb	000001	Rd	Ra	Rb	0...0	Rd := Rb + /Ra
ADDC Rd,Ra,Rb	000010	Rd	Ra	Rb	0...0	Rd := Rb + Ra + C
RSUBC Rd,Ra,Rb	000011	Rd	Ra	Rb	0...0	Rd := Rb + /Ra + C
ADDI Rd,Ra,Imm	001000	Rd	Ra	Imm		Rd := Imm + Ra
RSUBI Rd,Ra,Imm	001001	Rd	Ra	Imm		Rd := Imm + /Ra + 1

14.1. táblázat – Néhány MicroBlaze utasítás felépítése

A táblázatban szereplő Ra/Rb/Rd értékek regisztereket jelölnek, a /Ra a regiszter bitenkénti negálását jelöli, az Imm a 16 bites konstans operandust, és a C az átvitel flag-et jelöli. A táblázat első négy eleme A típusú, az utolsó két utasítás pedig B típusú utasítás.

A MicroBlaze processzor esetén az utasítások nagy része egy órajelciklus alatt végrehajtódik, ez alól csak a lebegőpontos műveletek és az osztás utasítása jelent kivételt. Ehhez a teljesítményhez csővezetékes végrehajtás szükséges. A processzor IP esetén megadható, hogy három-, öt-, vagy nyolcfokozatú csővezetéket szeretnénk használni. A háromfokozatú csővezeték (lásd 14.8. ábra) első fázisában az utasítás felhozatal történik (*fetch* fázis), a másodikban pedig az utasítás dekódolás (*decode*), a harmadik fázisban pedig az utasítás végrehajtás (*execute*). Ahogy az ábrán látható, bizonyos esetekben előfordulhatnak várakozó, *STALL* ciklusok. Ezeket a processzor a hosszabb utasítások, vagy a különböző hazárdok elkerülése miatt szúrja be a csővezetékbe. A felmerülő hazárdokat a 2. fejezetben már tárgyaltuk.

Copyright © 2021 / BME AUT



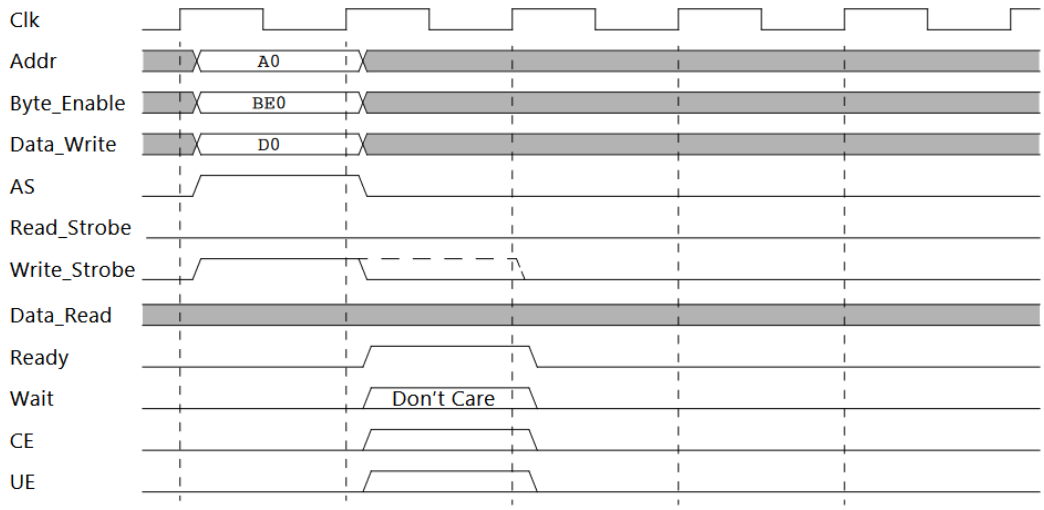
14.8. ábra – A MicroBlaze csővezetéke három fokozatú esetben[7]

Az öt- és nyolcfokozatú csővezeték a memóriához történő hozzáférés és az adatok visszaírása fázisokkal egészül ki. Az ötfokozatú csővezeték esetén egy fázis tartozik a memóriahozzáféréshez, a nyolcfokozatúnál viszont már négy fázis.

14.1.3.2 LMB busz

Korábban említettük, hogy viszonylag sok buszhoz illeszthető a MicroBlaze processzor. Fontos megjegyezni, hogy régebbi MicroBlaze verzióknál más buszokat is lehetett használni, mint az aktuális verzió esetén. Ebben a részben az aktuális, *v11.0* verziójú MicroBlaze IP-t ismertetjük.

A 14.5. ábrán látható buszok közül elsőként az ILMB és a DLMB buszokat nézzük át. Mindkettő esetén az úgynevezett *Local Memory Bus (LMB)* buszról van szó, az ILMB az utasítások, míg a DLMB az adatok eléréséhez használható. Az LMB busz elsősorban az FPGA-ban található blokk RAM-ok illesztéséhez használható, de akár saját perifériát is illeszthetünk a segítségével.

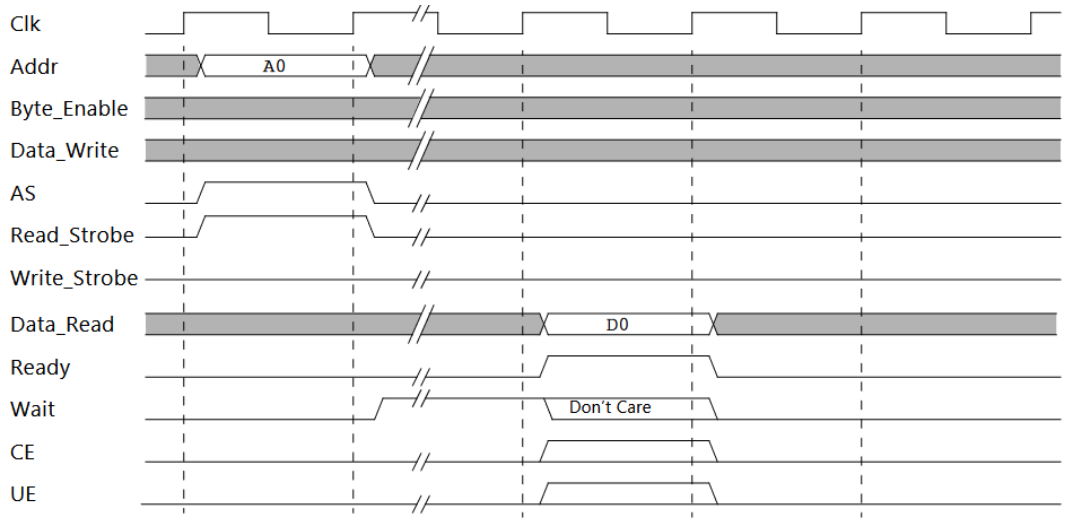


14.9. ábra – LMB írási átvitel[7]

A 14.9. ábrán látható egy írási tranzakció az LMB buszon. Az első ciklusban megjelenik a cím (*Addr* jelzéssel az ábrán), és a maximum 32 bites adat (*Data_Write*). A cím érvényességét az *AS* vezeték, az írás műveletet a *Write_Strobe* jelzi az átvitel során. Az adathoz tartozik egy *Byte_Enable* jel is, amellyel megoldható, hogy kevesebb bájtot írjunk a periféria számára (egy, két és három bájtos átvitel is lehetséges). A második ciklusban a periféria jelzi az átvitel sikeres végrehajtását a *Ready* jel segítségével. Az ábrán látható *Wait* jellel pedig időben eltolható az

Copyright © 2021 / BME AUT

átvitel, amennyiben erre szüksége van a perifériának. A *CE, UE* jelek hibajelzésre szolgálnak, ezeket most nem vizsgáljuk.



14.10. ábra – LMB olvasási átvitel WAIT ciklussal[7]

Az olvasás nagyon hasonlóan történik az íráshoz az LMB buszon (lásd 14.10. ábra). Ebben az esetben a *Read_Strobe* jel magas értéke jelzi az olvasást, és ennek megfelelően az írási adat nem jelenik meg a buszon. A következő órajelnél a *Wait* jelzés aktív, tehát a periféria még nem áll készen az adat továbbítására ebben a példában. A *Ready* jel későbbi magas értéke jelzi, hogy az adat rendelkezésre áll a buszon a *Data_Read* porton.

14.1.3.3 AXI buszok

A 14.5. ábrán látható, hogy AXI buszok is illeszthetők a MicroBlaze IP-hez. Az *M_AXI_DP/IP* portok segítségével perifériák illeszthetők a processzorhoz. Mivel az AXI busz memóriába ágyazott kommunikációt tesz lehetővé, így az általunk illesztett periféria saját címtartományt kap a használathoz. Az LMB-hez hasonlóan ebben az esetben is külön utasítás (M_AXI_IP) és külön adat (M_AXI_DP) busszal rendelkezik a CPU, és mindkét esetben master AXI-Lite portról van szó a processzor oldalán.

A soft processzorok esetén felmerülhet az igény, hogy saját, egyedi utasítással egészítsük ki a processzort. A MicroBlaze esetén erre nincs lehetőség, viszont a CPU rendelkezik 16-16 AXI-Stream porttal. A 14.5. ábrán az M[x]_AXIS a master, míg a S[x]_AXIS a slave portokat jelöli. Ezen portok segítségével címzés nélküli hardveres gyorsítókkal tudjuk kiegészíteni a processzort. A megoldás előnye az egyedi utasításokkal szemben az, hogy a processzor belső felépítésére nincs hatással a hardveres gyorsítóegység (pl. a csővezeték kialakítását nem zavarja meg), viszont az AXI-Stream miatt nagysebességű kommunikációt tesz lehetővé a külső egységgel. Az adatfolyamokhoz külön *get* és *put* utasítás tartozik. A get esetében két órajel, a put esetében egy órajel alatt végrehajtódik az átvitel.

Copyright © 2021 / BME AUT

14.1.3.4 Gyorsítótár interfészek

A MicroBlaze processzor opcionálisan rendelkezhet direkt leképezéses (1 utas) gyorsítótárral. Egy gyorsítótár sor hossza 4-től 16 szóig terjedhet, és kétféle busz segítségével is illeszthető. Egyrészt használhatjuk az újabb AMBA verzióknál megjelenő *AXI Coherency Extensions (ACE)* megoldást, amely a gyorsítótáraknál lévő koherencia problémát busz szinten megoldja, így nem szükséges szoftveres oldalról kezelni a gyorsítótárak közötti koherenciát. Másrészt AXI buszt is használhatunk a gyorsítótár illesztéséhez. Mindkét esetben a processzor oldal lesz a master, és külön utasítás és adat gyorsítótárat használhatunk a MicroBlaze-nél. Ennek megfelelően a 14.5. ábrán a következő jelölések tartoznak a gyorsítótár interfészekhez: *M_AXI_IC/DC és M_ACE_IC/DC.*

A MicroBlaze gyorsítótár komponense tartalmaz egy *stream buffer*-nek nevezett egységet, amelybe az aktuális utasítást követő címekről tölt be utasításokat spekulatívan a processzor. Így az egymást követő utasítások végrehajtása gyorsabb lesz, hiszen a gyorsítótár eléri a következő gyorsítótár sort a stream buffer-ből, és nem kell a memóriához fordulnia.

A stream buffer mellett egy másik gyakran használt megoldást is tartalmaz a MicroBlaze. Az úgynevezett *victim cache* lényege, hogy amennyiben egy sor kikerül a gyorsítótárból, nem kerül törlésre, hanem átkerül a victim cache-be. Így amennyiben a jövőben mégis szükség van az adott sorra, gyorsabban elérhető, mintha a memóriából kellene újra betölteni. Ezeket a megoldásokat a 4.3.2. fejezetben már láthattuk a gyorsítótárak tárgyalásánál.

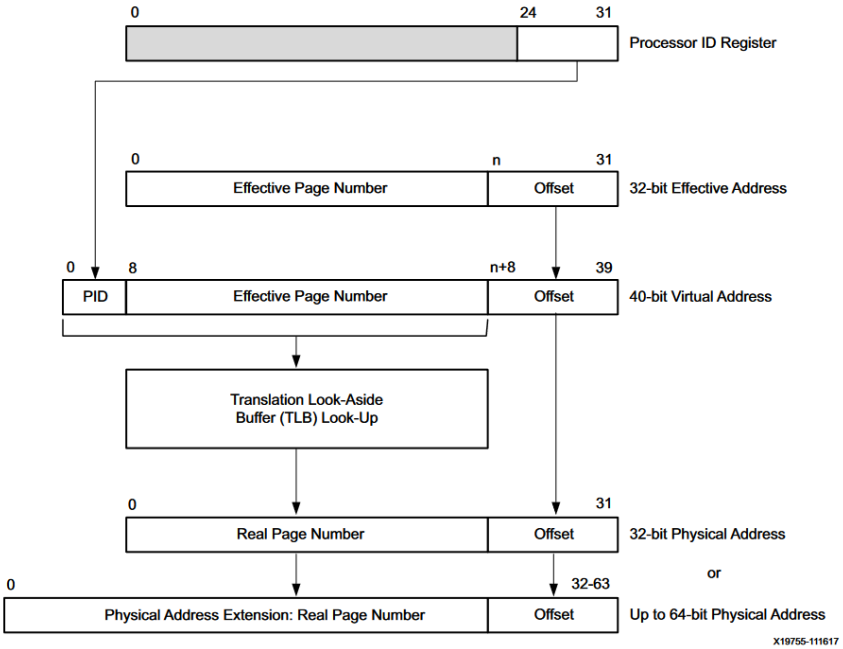
14.1.3.5 Memóriavezérlő egység

A MicroBlaze esetén akár komplexebb operációs rendszer (pl. Linux) is használható az alkalmazások futtatásához. A legtöbb operációs rendszer viszont megköveteli, hogy a processzor rendelkezzen memóriavezérlő egységgel, angolul *Memory Management Unit-tal (MMU)*. Az MMU legfontosabb feladata, hogy a processzor által használt címeket (ezeket *virtuális címeknek* nevezzük) felosztja adott nagyságú tartományokra (a tartományokat *lapnak* nevezzük ebben a terminológiában). A virtuális címek egyik része a lapokat azonosítja, a másik része pedig a lapon belüli ofszet értékét határozza meg. Az MMU számontartja, hogy melyik lap, milyen fizikai címen található meg a memóriában, ezáltal egy címtranszformációt végez a virtuális és a fizikai címek között. A megoldásnak számtalan előnye van: adott esetben az alkalmazások nem férnek hozzá a másik program memóriaterületéhez (memóriavédelem), a folytonos virtuális címtartomány leképezhető a memória különböző részeire (memória töredezettség esetén), és az alkalmazások összességében nagyobb memóriaterületet használhatnak, mint amekkora tényleges memóriával a rendszer rendelkezik (memory swapping).

A MicroBlaze esetén a címtranszformáció alapvetően két módban használható. A *valós mód* esetén nem történik transzformáció, a processzor által használt cím lesz a fizikai cím is. A *virtuális mód* esetén a 14.11. ábrán látható transzformáció történik. A 32 bites effektív cím kiegészül az adott alkalmazáshoz tartozó 8 bites azonosítószámmal (*PID* szám). Az így keletkező 40 bites cím alsó része adja meg a lapszámot, míg a felső része a lapon belüli ofszet érték lesz. A transzformáció az úgynevezett *Translation Look-Aside Buffer (TLB)* segítségével

Copyright © 2021 / BME AUT

történik, amely eltárolja a legutóbb használt virtuális címekhez tartozó fizikai címeket, ezáltal gyorsítva a címtranszformációt. A kapott cím a MicroBlaze kialakításától függően lehet 32 vagy 64 bites fizikai cím is.



14.11. ábra – Címtranszformáció MicroBlaze MMU esetén. Kissé szokatlan módon a legkisebb helyiértékű bit a bal oldalon található az ábrán. [7]

14.1.3.6 MicroBlaze erőforrásigény és teljesítmény

A MicroBlaze előző részekben bemutatott legtöbb komponense választható, ebből adódóan a hardver kialakítása nagyon széles tartományban változtatható. A processzor teljesítményét ráadásul nagyban befolyásolja az alkalmazott FPGA eszköz is a CPU beállításain kívül. A 14.12. ábrán egy adott FPGA esetén (XC7K325T) látható az előre elkészített konfigurációk erőforrásigénye és az elérhető órajel frekvenciája. Látható, hogy például a felhasznált flip-flop-ok számát tekintve a konfigurációk kb. 200 és 9000 között szórnak, amely elég széles tartomány, tekintve, hogy ugyanarról a CPU-ról van szó. Ennél a konkrét FPGA-nál az elérhető maximális órajel kb. 400 MHz, az újabb FPGA-k esetében viszont 500 MHz feletti frekvenciára is képes a MicroBlaze. A legnagyobb elérhető teljesítmény pedig kb. 500 DMIPS a legújabb FPGA technológia esetén.

Copyright © 2021 / BME AUT

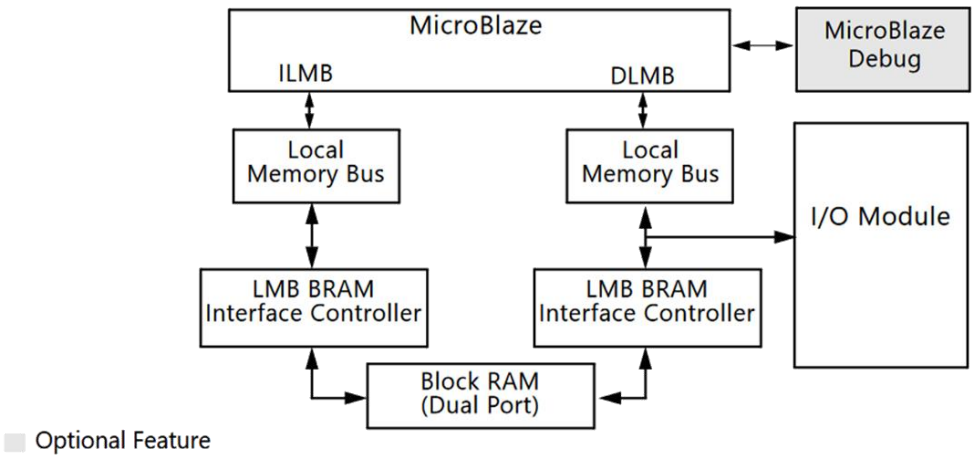
Table A-3: **Device Utilization - Kintex-7 FPGAs (XC7K325T ffg900-3)**

Configuration	Device Resources							
	32-bit				64-bit			
	LUTs	FFs	BRAMs (36K)	F_{max} (MHz)	LUTs	FFs	BRAMs (36K)	F_{max} (MHz)
Microcontroller Preset	1176	811	0	318	2129	1226	0	287
Real-time Preset	2477	2121	6	246	3792	3151	6	220
Application Preset	4368	3779	19	214	6479	4899	19	174
Minimum Area	637	234	0	398	1146	398	0	330
Maximum Performance	4129	3207	19	222	6816	4778	20	171
Maximum Frequency	908	553	0	398	1817	862	0	330
Linux with MMU	3507	3149	11	206	5088	4493	16	205
Low-end Linux with MMU	2986	2537	7	213	4521	3708	10	202
Typical	2017	1679	6	257	3404	2496	8	252
Frequency Optimized	6004	5874	14	263	9425	8765	15	172

14.12. ábra – A MicroBlaze erőforrás szükséglete és elérhető órajelfrekvencia Kintex-7 FPGA esetén[7]

14.1.3.7 MicroBlaze Micro Controller System (MCS)

Abban az esetben, ha viszonylag gyorsan akarunk összerakni egy processzort tartalmazó rendszert, akkor előnyös lehet a *MicroBlaze Micro Controller System (MCS)* IP-t használni. Ennek az IP-nek a célja, hogy egy teljes MicroBlaze alapú mikroprocesszoros rendszert kapjunk, memóriával és alapvető perifériákkal. Az egyszerűség miatt a MicroBlaze ebben az esetben csak kétféle konfigurációban használható (területre vagy teljesítményre optimalizált változat).



14.13. ábra – MicroBlaze MCS felépítése[8]

[8] Forrás: https://www.xilinx.com/support/documentation/ip_documentation/microblaze_mcs/v3_0/pg116-microblaze-mcs.pdf (2020. 05. 21.)

Copyright © 2021 / BME AUT

A 14.13. ábrán látható a MicroBlaze MCS IP blokkvázlata. A MicroBlaze CPU egységen kívül található a rendszerben egy blokk RAM modul, amelynek két portjához kapcsolódik az ILMB és a DLMB busz. Ezenkívül a DLMB buszra kapcsolódik egy I/O modul, amely tartalmaz időzítőt, UART vezérlőt, GPIO portokat, és megszakításvezérlőt is.

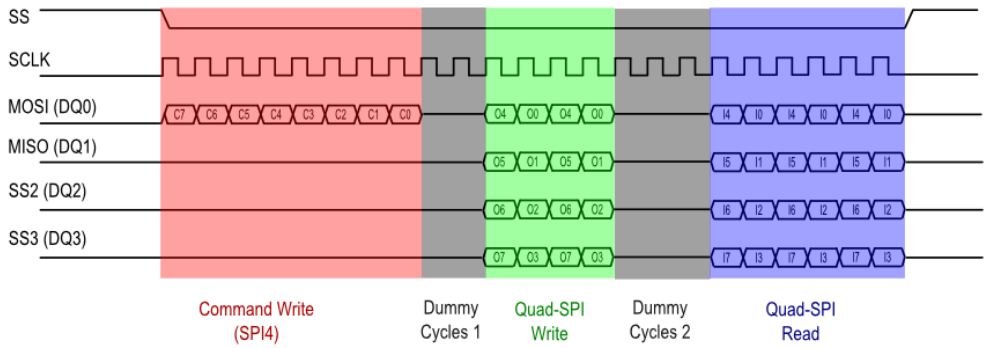
14.1.4 Xilinx perifériák

A Xilinx számos perifériát készített a MicroBlaze rendszerhez, amelyeket könnyedén tudunk a processzor IP-vel használni. Ezek közül néhány már említés szintjén szerepelt az MCS IP-nél, és most áttekintünk még néhány gyakran használt perifériát.

14.1.4.1 Xilinx SPI periféria

A *Serial Peripheral Interface (SPI)* egy széles körben használt szinkron, soros kommunikációs interfész két egység között (master, slave). Alapvetően az interfész az órajelen kívül egy kiválasztó jelet (SS: Slave Select), és két adatvezetéket tartalmaz (MOSI: Master Output Slave Input, MISO: Master Input Slave Output). A Xilinx által készített SPI IP esetében az úgynevezett *quad SPI* módra is lehetőségünk van, amelynél a nagyobb sávszélesség érdekében a két adatvezeték kiegészül még két vezetékkel (lásd 14.14. ábra). Ezt például SPI flash memóriáknál használják.

Az SPI IP-t mind master, mind slave módban tudjuk használni, és a processzorhoz AXI és AXI-Lite busszal is csatlakoztatható a periféria. A memóriába ágyazott AMBA buszokon keresztül, regiszterek segítségével tudjuk beállítani a kívánt működést. Szoftveres oldalról nézve pedig kész illesztőprogramot találunk az IP-hez.



14.14. ábra – Quad SPI kommunikáció[9]

A periféria egy speciális, úgynevezett *XIP (eXecute In Place)* móddal is rendelkezik, amelyet elsősorban flash memóriák illesztéséhez használnak. Ebben az esetben az AXI-Lite porton lehet az IP konfigurációját elvégezni, az AXI porton pedig csak olvasható módon tudunk a külső SPI

[9] Forrás: https://electronics.stackexchange.com/questions/286457/quad-spi-serial-or-parallel (2020. 05. 24.)

Copyright © 2021 / BME AUT

egységhez hozzáférni. A XIP mód segítségével közvetlenül külső SPI flash memóriából tudunk programot végrehajtani a CPU magon.

14.1.4.2 Xilinx Ethernet periféria

Az SPI-nál bonyolultabb kommunikációs interfészeket is használhatunk a MicroBlaze esetén. Például különböző Ethernet IP-k is a rendelkezésünkre állnak. Az alacsonyabb, 10 Mb/s és 100 Mb/s közötti Ethernet kommunikáció megvalósításához az *Ethernet Lite IP*-t használhatjuk, amely a *Media Access Controller (MAC)* rétegnek felel meg az OSI modellben. Ebben az esetben úgy tudjuk használni a kommunikációt, ha az FPGA áramkörön kívül elhelyezünk egy fizikai (PHY) réteget megvalósító áramkört az Ethernet csatlakozó és az FPGA között.

Ez az IP is AXI vagy AXI-Lite buszon keresztül illeszthető a MicroBlaze processzorhoz, és természetesen itt is rendelkezésünkre áll szoftveres illesztőprogram a perifériához.

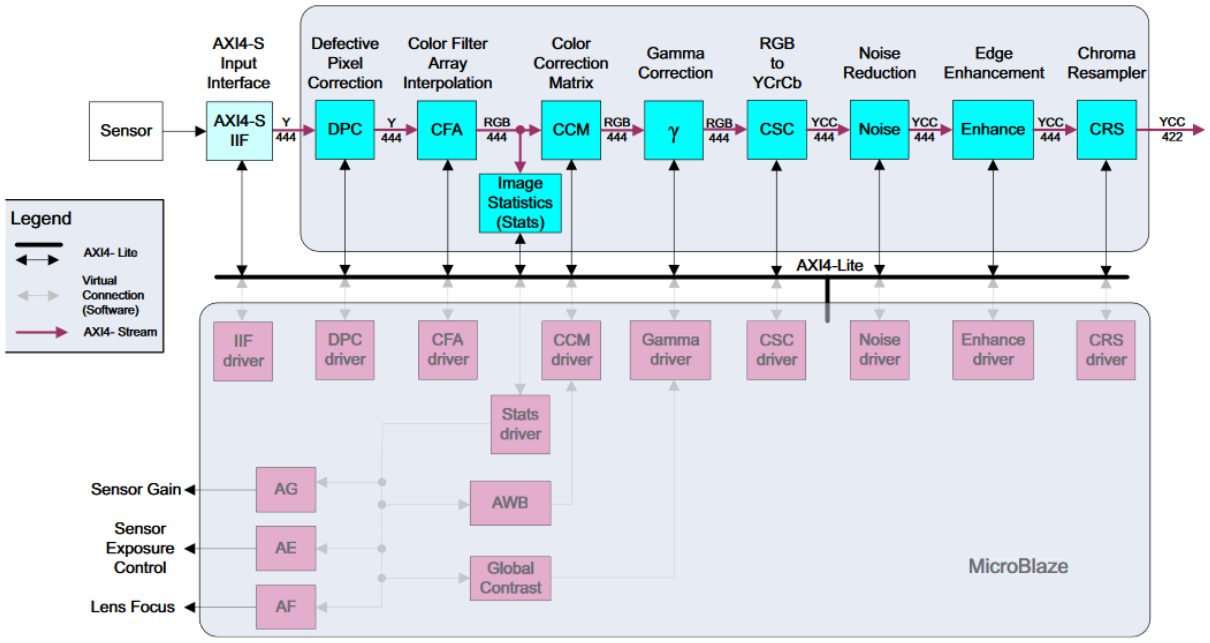
14.1.4.3 Xilinx PCI Express periféria

Az Ethernet-hez hasonlóan a PCI Express esetében is lehetőségünk van külső PHY egységgel kialakítani a kommunikációt. Viszont a Xilinx 7-es sorozatú FPGA eszközei óta már hard PCI Express magot is tartalmazhatnak az FPGA eszközök, amelyekhez nem szükséges semmilyen külső áramkör. Ebben az esetben a PCI Express-nél tanult összes feladatot (pl. 8b/10b kódolás, dekódolás) megoldja a hard IP mag.

A legújabb hard IP megoldások a PCI Express 4-es szabványát támogatják, és akár bizonyos esetekben 16 sávos kommunikációt is lehetővé tesznek. Az sávok sebessége a 16 GT/s értékét is elérheti, mivel az IP felhasználja az FPGA-ban található nagysebességű soros átviteli blokkokat is. A PCI Express AXI-Stream interfész segítségével illeszthető az FPGA többi részéhez (pl. MicroBlaze-hez).

A 13.2.3. fejezetben már említettük, hogy az AXI-Stream használható például kamerás rendszerek esetén a pixelinformációk átviteléhez. A 14.15. ábrán látható egy kamerarendszer megvalósítása AXI-Stream és MicroBlaze alapokon. Az ábra felső részén különböző képmanipuláló IP blokkok láthatóak. Mindegyik blokk AXI-Stream buszon kapja meg a bejövő pixeleket, és szintén AXI-Stream buszon küldi tovább a módosított adatokat. A blokkok feladata, hogy a nyers pixeleket korrigálja és kompenzálja valamilyen szempontból. Például kiszűrje a hibás pixeleket, meghatározza a helyes színinformációt minden pixel esetén, színtér konverziót végezzen stb. Az IP blokkokhoz sokféle beállítási lehetőség tartozik, amelyet a példában AXI-Lite buszon lehet megtenni (lásd a 14.15. ábra alsó fele). Minden blokkhoz tartozik szoftveres oldalról egy-egy illesztőprogram (angolul *driver*), amelyek segítségével be tudjuk állítani az adott IP regisztereit. Az ábrán az is látható, hogy a MicroBlaze alapú szoftver a kamera egyéb paramétereit is szabályozhatja (pl. erősítés, exponálási idő, fókusz beállítás).

Copyright © 2021 / BME AUT



14.15. ábra – Kamerarendszer MicroBlaze processzorral és AXI-Stream busszal[10]

14.1.5 Intel Nios II

Az Intel is rendelkezik saját soft CPU megoldással az FPGA eszközeihez, amelyet *Nios* néven érhetünk el. Az első verzió még egy 16 bites processzor volt, az aktuális Nios II pedig már a MicroBlaze-hez hasonlóan egy 32 bites RISC processzor.

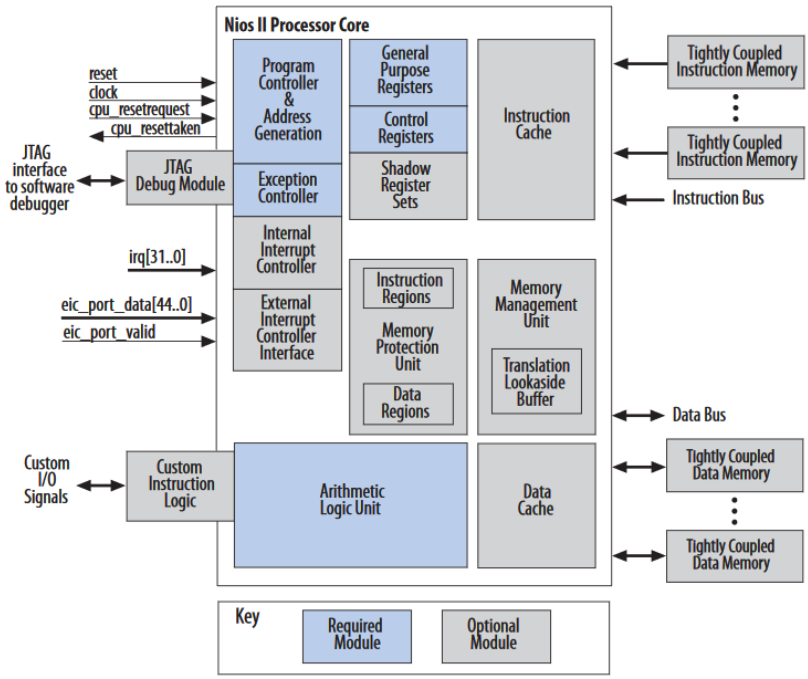
A Nios II felépítése a 14.16. ábrán látható. Ahogy ezt már a MicroBlaze-nél tapasztaltuk, ebben az esetben is a legtöbb modul opcionális. Ráadásul a processzor legtöbb elemével már megismerkedtünk a Xilinx soft CPU megoldása esetén. A legnagyobb különbség, hogy az Intel a saját Avalon buszcsaládját használja a Nios esetén. Habár AXI busz nem használható a processzorhoz, viszont létezik Avalon-AXI bridge IP, amely segítségével megoldható az áttérés.

A 14.16. ábrán az is látható, hogy a Nios esetén lehetőségünk van egyedi utasítással kiegészíteni a processzort. Ezt jelzi a *Custom Instruction Logic* modul az ábra bal oldalán. A Xilinx esetén nem volt meg ez a lehetőségünk. A Nios-nál több különböző egyedi utasítás típust használhatunk, de ezeket jelen esetben nem részletezzük.

A blokkdiagram alapján különbséget jelent az is, hogy míg a MicroBlaze MMU modulja tartalmazza a memóriavédelmi egységet is, az Intel ezt egy külön blokkba szervezte ki. Szintén apróbb különbség, hogy a Nios II fixen hatfokozatú csővezetékkel rendelkezik.

[10] Forrás: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf (2020. 05. 24.)

Copyright © 2021 / BME AUT



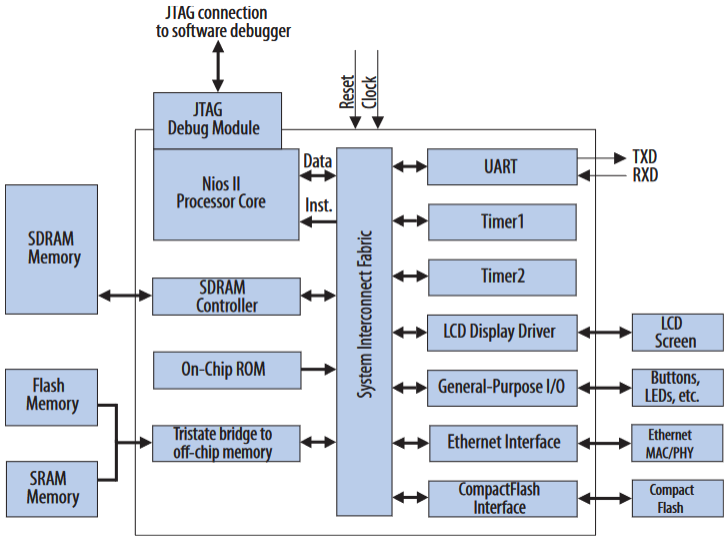
14.16. ábra – Intel Nios II felépítése[11]

Az Intel Nios alapvetően két konfigurációban használható. A *fast* esetén egy nagyteljesítményű processzort kapunk, míg az *economy* beállításnál kevesebb erőforrást használ a rendszer, és kevesebb beállítási lehetőséggel rendelkezünk.

Az Intel esetében is működik az IP alapú tervezés, amelyet a Xilinx esetében már láthattunk. A leggyakrabban használt perifériák ugyanúgy rendelkezésre állnak, csak most az AMBA családba tartozó buszok helyett az Avalon-hoz tartozó buszokat használhatjuk. Az Avalon esetében is találunk memóriába ágyazott és adatfolyam jellegű buszokat is.

[11] Forrás: Intel NIOS II Processor Reference Guide (2020. 04. 24.)

Copyright © 2021 / BME AUT



14.17. ábra – Intel Nios alapú rendszer blokkvázlata[11]

A 14.17. ábrán látható egy példarendszer, amelynél egy Nios II processzorhoz illesztettek különböző perifériákat, például statikus és dinamikus memóriavezérlőket, időzítőket, Ethernet interfészt.

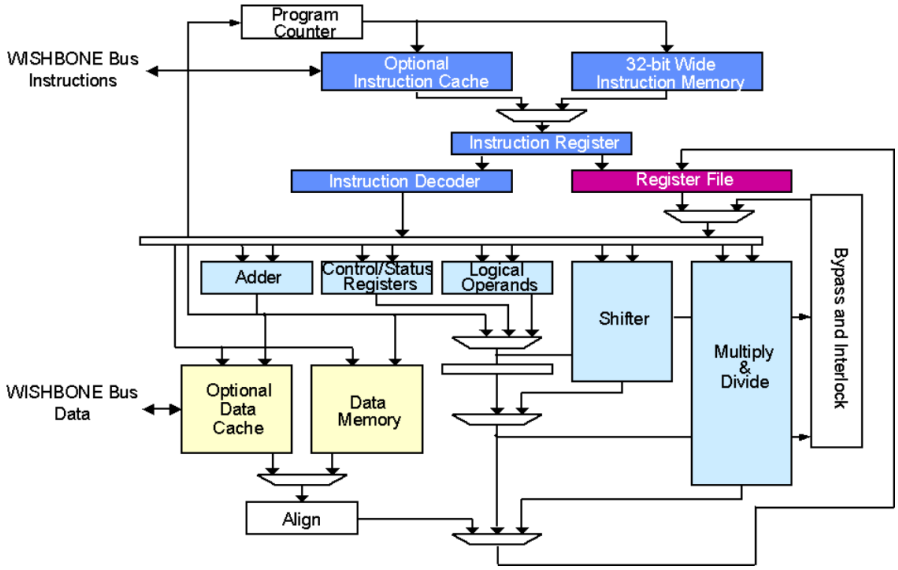
14.1.6 További gyártók soft processzorai

Most pedig röviden megnézzük, hogy más iparági szereplők milyen soft processzorokkal rendelkeznek.

14.1.6.1 Lattice LatticeMico processzorok

A Lattice cég a kisebb FPGA gyártók közé tartozik az Intel és a Xilinx után. A nagyobb gyártókhoz hasonlóan a Lattice is rendelkezik saját, 32 bites RISC soft processzorral, *LatticeMico32* néven. Emellett létezik 8 bites változata is *LatticeMico8* néven, amely a PicoBlaze-hez hasonló komplexitású soft processzor.

Copyright © 2021 / BME AUT



14.18. ábra – Lattice 32 bites RISC soft processzora[12]

A 14.18. ábrán látható a LatticeMico32 felépítése. A processzor legtöbb elemét már a korábbiakban láthattuk más CPU-k esetében. A legnagyobb különbséget az jelenti, hogy ennél a processzornál a *Wishbone* buszt használhatjuk a külső egységek illesztéséhez. Egy másik lényeges különbség a LatticeMico32 esetén az, hogy ezt az IP-t nem csak a Lattice FPGA-k esetén használhatjuk, hanem a konkurens gyártók termékei esetén is, mivel a LatticeMico32 nyílt forráskódú, royalty mentes IP.

A blokkdiagram alapján az is látható, hogy a Lattice processzora egyszerűbb felépítésű, mint a MicroBlaze vagy a Nios. Például nem látunk MMU egységet vagy lebegőpontos aritmetikát. Viszont ez a processzor is rendelkezik hatfokozatú csővezetékes végrehajtással és elágazásjövendölő egységgel.

14.1.6.2 Altium TSK51x

Az Altium céget elsősorban a nyomtatottáramkör-tervező programjai miatt ismerhetjük. Habár FPGA áramköröket nem fejlesztenek, FPGA-hoz köthető szoftvereket fejlesztettek régebben.

A fejlesztőeszközökön kívül saját 8 bites soft processzort is készítettek, *TSK51x* néven. A TSK51x különlegessége, hogy az utasításkészlete kompatibilis a jól ismert Intel 8051-es processzor utasításkészletével. Az utóbbi években viszont megszűnt a TSK51x CPU támogatottsága, mivel az Altium kiszállt az FPGA iparágból.

[12] Forrás: LatticeMico32 Processor Reference Manual

Copyright © 2021 / BME AUT

14.1.6.3 ESA LEON

Az Európai Űrügynökség *LEON* néven fejlesztett saját soft processzort elsősorban űripari alkalmazásokhoz. Viszont bizonyos megkötések mellett szabadon felhasználható az IP. A processzor a Sun Microsystems SPARC V8 32 bites RISC architektúrájára épül. Az űripari alkalmazások miatt úgy alakították ki a CPU-t, hogy nagyobb sugárzás esetén is működőképes legyen (angolul *radiation-tolerant* eszközöknek nevezik ezeket az alkatrészeket). Ez a gyakorlatban leginkább a redundanciák növelését jelenti a processzor architektúrájában.

14.1.7 Akadémiai kutatások

Az eddig bemutatott soft processzor megoldások minden esetben a hardvertervezési iparág valamely vállalatához köthetők. Viszont számtalan nyílt forráskódú, gyakran egyetemi kutatáshoz kapcsolódó soft IP-vel is találkozhatunk a szakirodalomban. Ezek közül mutatunk be most néhány példát.

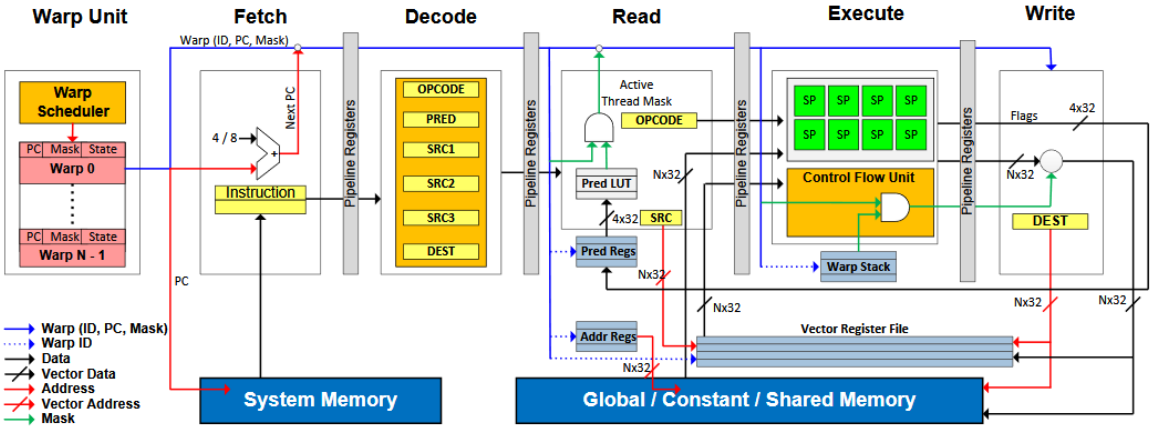
14.1.7.1 FlexGrip GPGPU

A 2013-ban kifejlesztett *FlexGrip* processzor[13] különlegessége, hogy *Nvidia Cuda* utasításokat képes futtatni. Ebből adódóan inkább általános célú GPU-nak (GPGPU) tekinthető, mint CPU-nak. A szerzők a megvalósítás során az Nvidia G80-as architektúráját vették alapul, és egy Xilinx Virtex 6 FPGA-ban készítették el a megoldásukat. Ebből adódóan a GPGPU épít az FPGA-ban található blokk RAM és DSP egységekre.

A FlexGrip úgynevezett Streaming Multiprocessor (SM) blokkokból áll, amelyekkel már a 5. fejezetben találkoztunk. Az SM blokkok vázlata a 14.19. ábrán szerepel. Az ábra alapján látható, hogy minden SM nyolc darab skalár végrehajtó egységből, angolul Scalar Processor (SP) blokkból áll, és a végrehajtáshoz ötfokozatú csővezetéket használnak, Fetch, Decode, Read, Execute és Write fázisokkal.

[13] Forrás: http://www.ecs.umass.edu/ece/tessier/andryc-fpt13.pdf (2020. 05. 25.)

Copyright © 2021 / BME AUT



14.19. ábra – FlexGrip GPGPU Streaming Multiprocessor blokkvázlata[13]

A FlexGrip fejlesztői által közölt cikk alapján a GPGPU akár 30-szor gyorsabb lehet a MicroBlaze processzorhoz képest a vizsgált FPGA esetén. Hozzá kell tenni, hogy ezt az eredményt úgy érték el, hogy a FlexGrip lényegesen több erőforrást igényel, mint a MicroBlaze. Például LUT-ből több, mint 200 ezer darabot igényel, míg az összehasonlításban szereplő MicroBlaze csak 3 ezret.

14.1.7.2 iDEA CPU

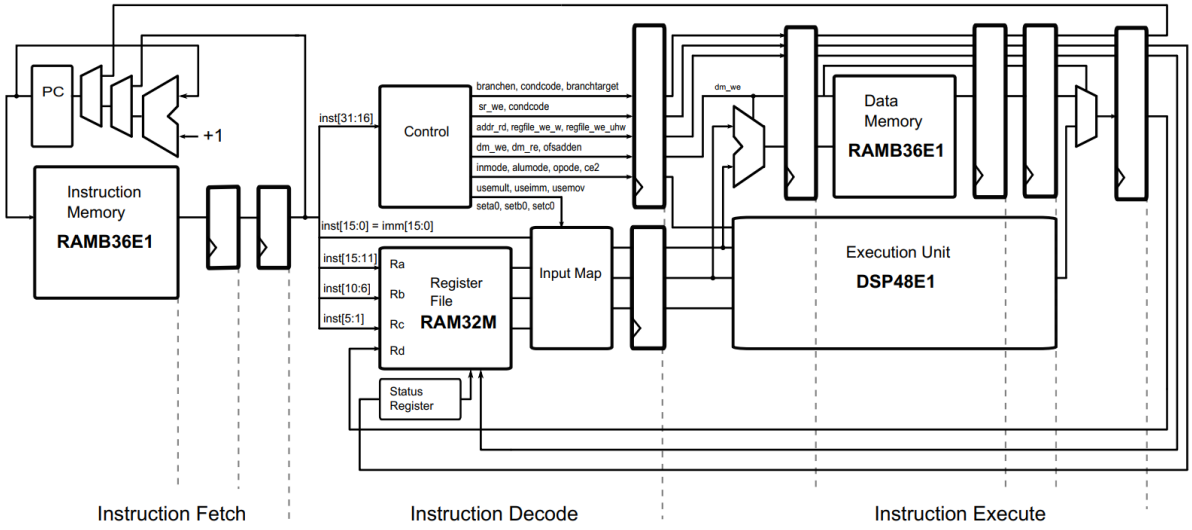
A FlexGrip-hez hasonlóan a *DSP Extension Architecture (iDEA)* soft CPU is akadémiai kutatásból származik[14]. Ebben az esetben a megoldás különlegessége az, hogy a processzor aritmetikai egysége teljes mértékben a Xilinx DSP blokknak felel meg. Mivel a DSP blokk által végrehajtott utasítás dinamikusan konfigurálható, így lehetőség van programozható módon használni ezeket a blokkokat. Ezt használja ki az iDEA CPU.

A CPU háromfázisú végrehajtó egysége a 14.20. ábrán látható. Az ábra jobb oldalán, az Execute fázisban látható a DSP blokk. Az ábra alapján az is megállapítható, hogy a Xilinx DSP blokkjain kívül a blokk RAM egységre is erősen építkezik az architektúra, hasonlóan a FlexGrip GPGPU-hoz.

Mivel a DSP blokk viszonylag kevés utasítással rendelkezik egy CPU-hoz képest, így az iDEA utasításkészlete is korlátozott. Viszont a felépítésből adódóan az erőforrásigénye is alacsony. A szerzők által vizsgált Xilinx 7-es FPGA eszközök esetén alacsonyabb erőforrásigényt mértek egy hasonló képességű MicroBlaze IP-hez képest. Ráadásul a vizsgált MicroBlaze csupán 200 MHz órajelfrekvenciát tudott elérni, míg az iDEA a 400 MHz-es frekvenciát is elérte. Számítási teljesítményben viszont az esetek többségében a MicroBlaze győzött a szerzők mérései alapján.

[14] Forrás: https://warwick.ac.uk/fac/sci/eng/staff/saf/papers/trets2014-cheah.pdf (2020. 05. 25.)

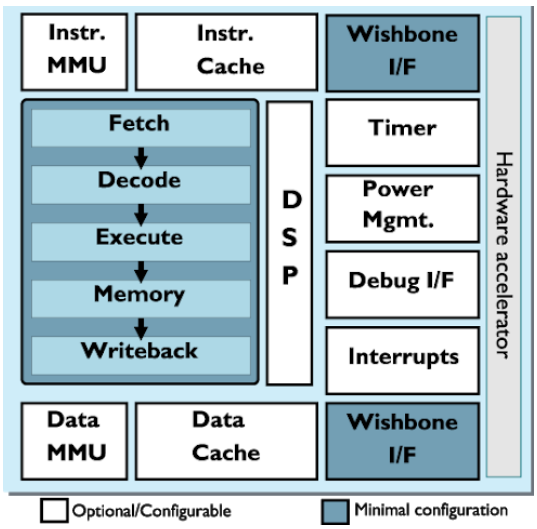
Copyright © 2021 / BME AUT



14.20. ábra – iDEA CPU végrehajtó egysége[14]

14.1.7.3 OpenRISC 1000 CPU

A korábbiakban már említettük a Wishbone buszt, amelyet a Lattice használt a saját soft CPU megoldása esetén. A Wishbone-t létrehozó közösség (OpenCores) számos egyéb nyílt forráskodú IP-t készített, beleértve egy RISC processzort is, az *OpenRISC 1000* CPU-t.



14.21. ábra – Az OR1200 CPU főbb egységei[15]

[15] Forrás: https://elinux.org/images/e/e2/Openrisc_talk_2016.pdf (2020. 05. 25.)

Copyright © 2021 / BME AUT

Az OpenRISC fejlesztése 2000-ben kezdődött, és viszonylag részletes leírás készült az architektúráról[16]. Az évek során számos projektben használták az OpenRISC CPU-t, kezdve a Samsung televíziós eszközeitől, egészen a NASA TechEdSat programjáig[17].

Az OpenRISC CPU-hoz több implementáció is készült. Az első Verilog alapú implementáció *OR1200* néven jelent meg, és a 14.21. ábrán látható a felépítése. Az ábra alapján látható, hogy ebben az esetben is a legtöbb blokk opcionális, az eddig látott komolyabb soft processzorokhoz hasonlóan.

Az OR1200 esetében természetesen az OpenCores közösség által létrehozott Wishbone kommunikációs buszt használhatjuk. Az OR1200 processzor rendelkezik szoftveres támogatottsággal is, például GCC alapú fordítóprogram, GDB hibakereső, szimulátor is használható, és akár Linux operációs rendszert is futtathatunk a soft processzoron.

14.2 Hard processzorok

Ebben a részben azokat az egylapkás rendszereket vizsgáljuk meg, amelyek hard processzor magot is tartalmaznak. Ahogy a korábbiakban, most is elsősorban az FPGA alapú SoC rendszerekre fókuszálunk.

14.2.1 Xilinx történeti áttekintés

A Xilinx nagyteljesítményű FPGA családjában, a Virtex sorozatban 2002 óta találunk olyan programozható logikákat, amelyek mellett hard processzormag is található. A Virtex II volt az első FPGA a sorozatban, amely esetén hard CPU magot tudtunk használni. Ebben az FPGA-ban két darab *PowerPC 405* CPU blokkot helyeztek el.

A PowerPC egy RISC felépítésű processzor, amely az Apple, IBM, és a Motorola közös fejlesztéseként indult a 1990-es évek elején. A Virtex II-ben található *PowerPC* 300 MHz feletti órajelfrekvenciával működött, kb. 600 DMIPS-re volt képest egy mag, tartalmazott hardveres szorzó- és osztóegységet, valamint MMU blokkot. A processzor az IBM által fejlesztett *CoreConnect* busszal volt kompatibilis.

A Xilinx még a Virtex 5 FPGA esetén is a PowerPC magokat használta, amelyek ebben az időben már *1100 DMIPS* teljesítményre voltak képesek magonként. Összehasonlításképpen a MicroBlaze a Virtex 5 esetén kb. 240 DMIPS teljesítményt ért el. Ebben az FPGA-ban található *PowerPC 440* mag már szuperskalár csővezetékkel, sorrenden kívüli végrehajtással is rendelkezett.

A 2009-ben megjelent Virtex 6 esetében viszont már semmilyen hard CPU magot nem helyeztek el a Xilinx mérnökei. Ebben az időben csak soft IP blokkok segítségével lehetett

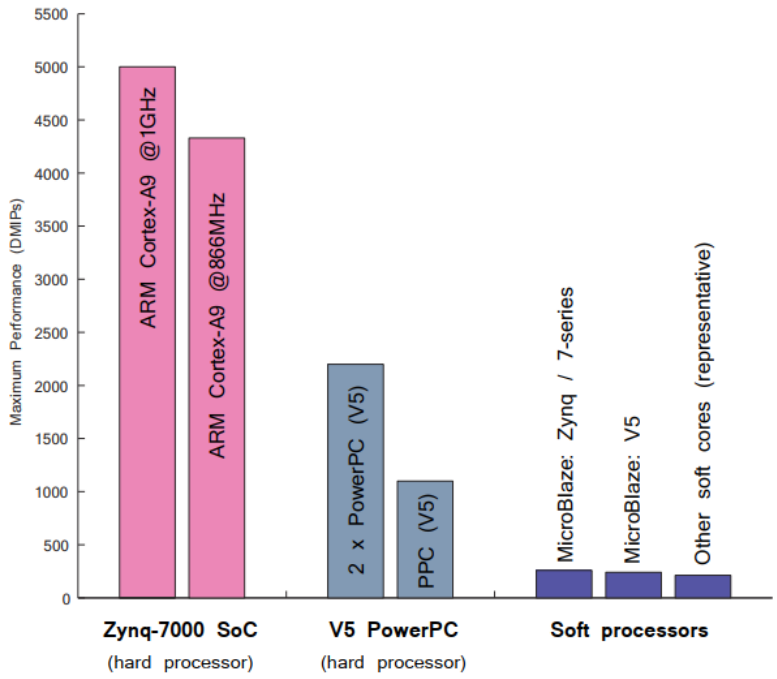
[16] Forrás: https://openrisc.io/architecture (2020. 05. 25.)
[17] Forrás: https://en.wikipedia.org/wiki/OpenRISC (2020. 05. 25.)

Copyright © 2021 / BME AUT

processzort használni a Virtex 6 FPGA-n belül. Nyilvánvalóan a Xilinx a MicroBlaze-t preferálta, amely abban az időben a PowerPC-hez hasonlóan a CoreConnect buszt használta.

14.2.1.1 Xilinx Zynq

A 2010-ben megjelent 7-es FPGA család esetén visszatértek a hard CPU magok a Xilinx FPGA áramkörökbe. Egy teljesen új, *Zynq-7000* elnevezésű egylapkás áramkör jelent meg, amely a programozható logikán kívül ARM Cortex-A9-es magokat is tartalmazott, valamint számtalan perifériát. A 14.22. ábrán látható a Zynq-7000 processzoros egysége által elért teljesítmény a Virtex 5-ben szereplő PowerPC magokhoz és a MicroBlaze soft CPU magokhoz viszonyítva. A Zynq felépítését és működését a 15. fejezetben részletesen megvizsgáljuk.



14.22. ábra – Különböző platformok által elért DMIPS teljesítmény[18]

A 2010 óta megjelent újabb generációs FPGA családokkal párhuzamosan a Zynq család is fejlődött, és a legújabb *Zynq Ultrascale+* SoC-k belsejében már ARM Cortex-A53 és Cortex-R5 CPU magokat, illetve ARM Mali GPU magokat is találunk a programozható logikán kívül.

14.2.1.2 Xilinx Versal

Ahogy korábban említettük, a Xilinx 2019-ben egy új eszközcsaládot jelentett meg *Versal* néven. A Versal esetén már nem az FPGA kifejezést, hanem az Adaptive Compute Acceleration Platform (ACAP) elnevezést használják. A névváltás leginkább abból adódik, hogy az

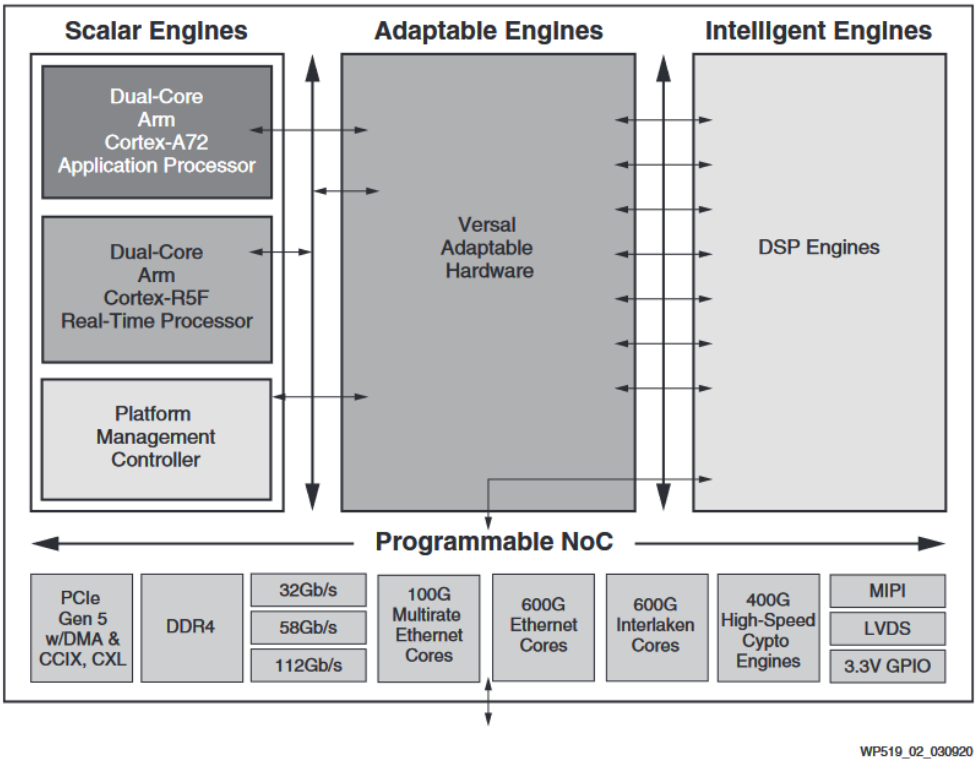
[18] Forrás: http://www.zynqbook.com/ (2020. 05. 25.)

Copyright © 2021 / BME AUT

áramköben jelentős szerepet kapnak a különböző hard IP magok, nem a programozható logika a domináns.

A 14.23. ábrán látható a Versal Premium áramkör blokkvázlata. Az ábra középső, Versal Adaptable Hardware blokkja jelöli a programozható logikai részt, amely megfelel az általunk vizsgált FPGA-k felépítésének. Emellett az áramkör ARM Cortex-A72 és Cortex-R5F CPU magokat is tartalmaz, mint skalár végrehajtó egységek. Az ábrán látható harmadik nagyobb egység, az *Intelligent Engines*, a korábban bemutatott DSP blokkokat tartalmazza. A DSP blokkok tekinthetők vektor végrehajtó egységként is, amely nagy mennyiségű adatot tud párhuzamosan feldolgozni.

A Versal áramkör a három különböző végrehajtó egységen (skalár, vektor, egyedi programozható logika) kívül számos hard IP perifériát tartalmaz, amelyek elsősorban nagysebességű kommunkációt tesznek lehetővé. Az ábrán például megfigyelhetők a PCI Express, Ethernet és DDR4 vezérlő hard IP blokkok. A különböző részegységek között nagysebességű, AXI alapú buszhálózat található. Ezt jelöli a *Network-on-Chip (NoC)* megnevezés az ábrán.



14.23. ábra – Versal Premium ACAP felépítése[19]

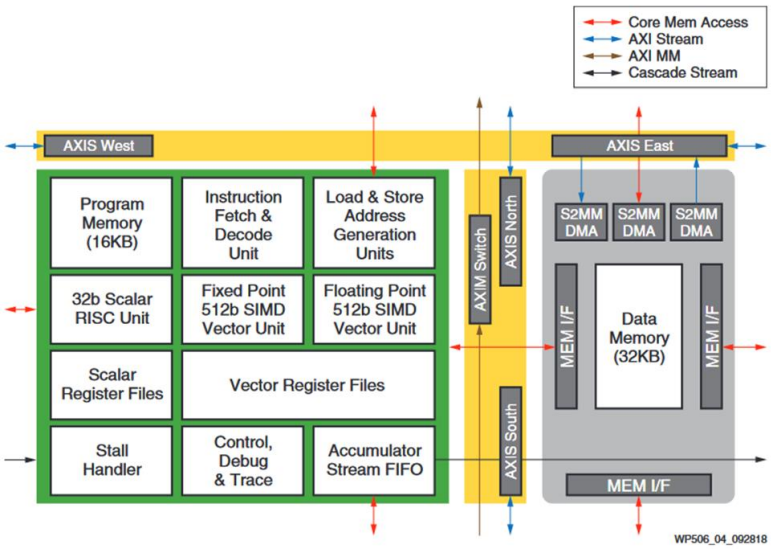
[19] Forrás: https://www.xilinx.com/support/documentation/white_papers/wp519-versal-premium-intro.pdf (2020. 05. 25.)

Copyright © 2021 / BME AUT

Bizonyos Versal eszközök esetén az Intelligent Engines nem DSP blokkokat, hanem úgynevezett *Artificial Intelligence (AI) Engine* magokat tartalmaz. Ezek az egységek szintén elsősorban vektoros végrehajtásra készültek a DSP blokkokhoz hasonlóan. A 14.24. ábrán látható, hogy pontosan milyen részekből épül fel az AI Engine. Található a blokkban egy 32 bites skalár RISC CPU mag, két 512 bites SIMD vektoros végrehajtó egység, valamint adat és program memóriák. Ezekből a blokkokból akár 400 db is lehet egy Versal áramkörön belül, és a blokkok között AXI buszon tudunk kommunikálni.

A Xilinx mérései alapján, ha az AI Engine funkcionalitását általános programozható logika (FPGA) segítségével építenénk fel ugyanakkora területen, akkor az AI Engine blokkok akár nyolcszor nagyobb számítási kapacitást érnének el. Ezzel párhuzamosan az energia igénye is fele lenne az AI Engine-nek az FPGA alapú megoldással szemben.

A Xilinx által készített mérések alapján a Versal az Nvidia V100-as GPU-ját is felül tudja múlni számítási teljesítményben. Az általuk vizsgált feladat egy, a gépi tanuláshoz kapcsolódó eljárás volt GoogLeNet neurális hálóval, és a Versal kb. háromszor nagyobb teljesítményt ért el a GPU-hoz képest.



14.24. ábra – Versal AI Engine blokkvázlata[20]

Az AI Engine végrehajtó egységei (14.24. ábra zöld részén található egységek) VLIW utasításokat hajtanak végre. Minden utasítás magába foglal két skalár műveletet, egy vektor műveletet, két vektor betöltést (load), és egy vektor mentést (store). A VLIW utasítások teszik lehetővé az utasításszintű párhuzamosítást, míg az adatszintű párhuzamosítás a SIMD végrehajtással lehetséges.

[20] Forrás: https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf (2020. 05. 26.)

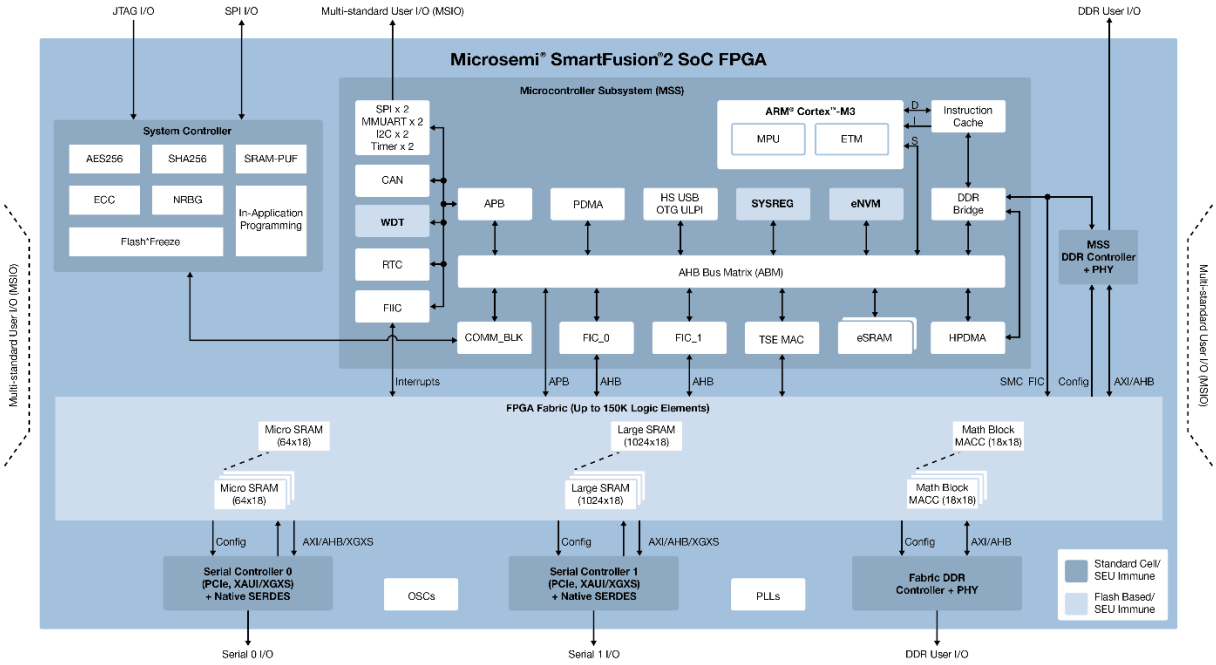
Copyright © 2021 / BME AUT

14.2.2 Intel Agilex SoC

Az Intel legújabb Agilex FPGA családját már a 12.5. fejezetben röviden bemutattunk. Arról eddig nem esett szó, hogy az Agilex-nek is létezik olyan változata, amely esetén ARM Cortex-A processzormagokat is integráltak az áramkörbe. A Zynq Ultrascale+ áramkörhöz hasonlóan, itt is ARM Cortex-A53 hard CPU magot találunk, és ebben az áramkörben is találunk számos perifériát a processzormag körül (pl. USB, Ethernet stb.)

14.2.3 Microsemi SmartFusion SoC

A kisebb FPGA gyártók közé tartozik a Microsemi cég. Korábban önállóan, ma már a Microchip cég részeként működik. A Microsemi viszonylag sokféle FPGA eszközt gyárt, és több esetben CPU magokat is találunk a programozható logikán kívül. A 14.25. ábrán látható a *SmartFusion 2* elnevezésű egylapkás eszközük, amely egy ARM Cortex-M3 processzor magot tartalmaz a programozható logikán kívül.



14.25. ábra – Microsemi SmartFusion 2 felépítése[21]

Az ábrán látható, hogy ennél az áramkörnél is az ARM AMBA buszcsaládja dominál. Láthatjuk a középen megjelenő AHB busz mátrixot, és a jobb oldalon található AXI buszt a DDR vezérlőnél. A programozható logika az ábra alsó részén foglal helyet.

Habár a Microsemi FPGA megoldásait nem tárgyaltuk, az ábrán sok ismerős modult azonosíthatunk. A különböző SRAM blokkok megfelelnek a Xilinx blokk RAM moduljainak,

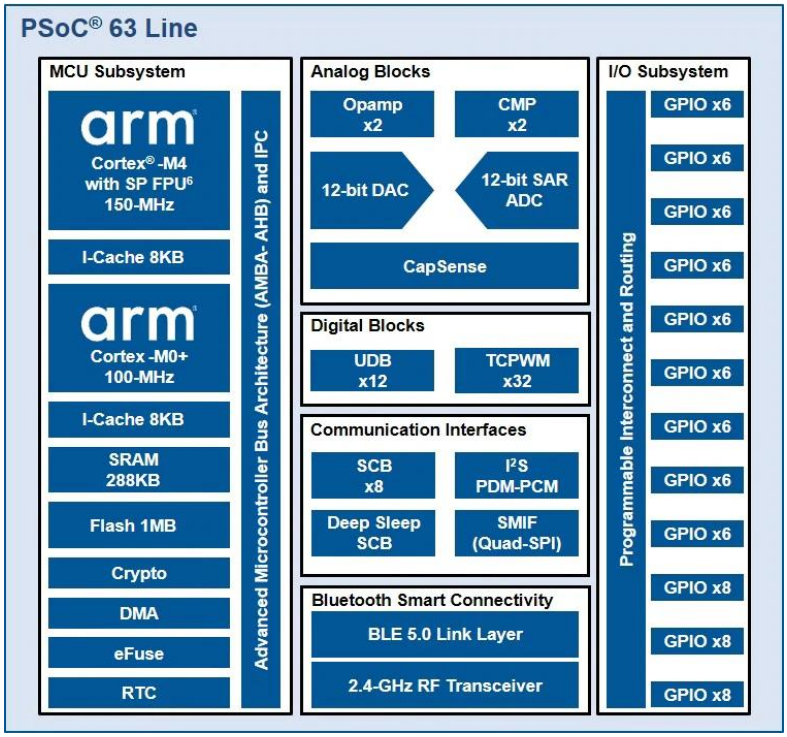
[21] Forrás: https://www.microsemi.com/product-directory/soc-fpgas/1692-smartfusion2 (2020. 05. 26.)

Copyright © 2021 / BME AUT

a *Math Block* elnevezés pedig a DSP blokkoknak. Az ábra alsó részén pedig nagysebességű soros átviteli blokkokat látunk, amely a Xilinx és az Intel megoldásainál is előfordul.

14.2.4 Cypress PSoC

A Cypress cég PSoC termékét említettük már a 11.2-es fejezetben. A 14.26. ábrán pedig a PSoC legújabb, 6-os változatának blokkvázlata látható. A Microsemi megoldásához hasonlóan, itt is ARM Cortex-M magokat láthatunk, viszont ebben az esetben két hard CPU mag is helyet kapott az áramkörben. Hasonlóan az eddig látott eszközökhöz, ebben a termékben is az ARM AMBA buszcsalád teremt kapcsolatot az eszközön belüli IP-k között. A perifériák közül említésre méltó a Bluetooth 5-ös rádiós IP blokk. A programozható logikai blokkot pedig *Universal Digital Block (UDB)* néven láthatjuk az ábrán.



14.26. ábra – Cypress PSoC 63 blokkvázlata[22]

[22] Forrás: https://www.cypress.com/products/32-bit-arm-cortex-m4-cortex-m0-psoc-63-connectivity-line (2020. 05. 26.)

Copyright © 2021 / BME AUT

15 Xilinx Zynq PSoC

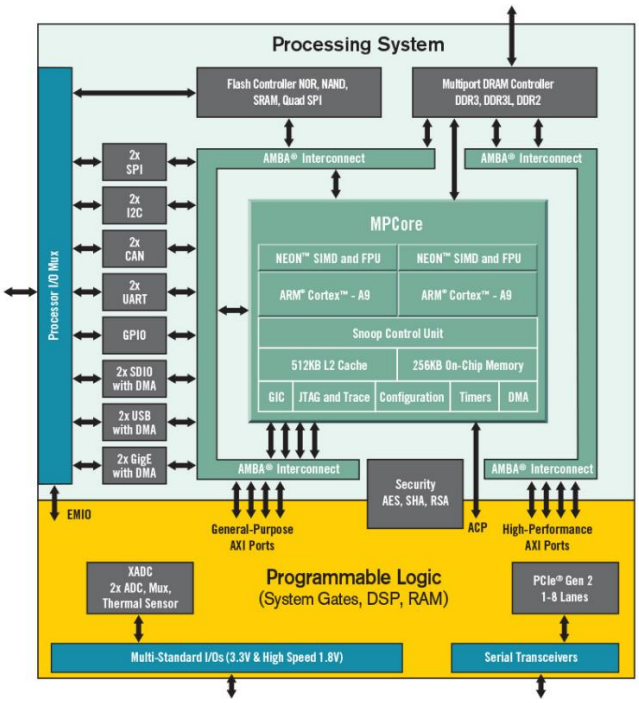
Az előző részben már röviden bemutattuk a Xilinx Zynq SoC rendszert. Ebben az fejezetben pedig áttekintjük a különböző Zynq eszközöket, és részletesen bemutatjuk a legelső Zynq-7000 áramkör architektúráját.

15.1 Zynq család tagjai

A 2011-ben megjelent első Zynq eszköz óta már számos változat jelent meg, amelyek mind a CPU mag, mind a programozható logika terén fejlődést jelentettek az eredeti architektúrához képest.

15.1.1 Zynq-7000

A Zynq család legelső tagja a *Zynq-7000*-es áramkör volt. Ahogy a korábbiakban említettük, ebben két darab ARM Cortex-A9 CPU mag található, a programozható logika pedig a 7-es sorozatú FPGA-nak felel meg. A rendszer blokkvázlata a 15.1. ábrán látható.



15.1. ábra – Zynq 7000 blokkvázlata[1]

[1] Forrás: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (2020. 05. 26.)

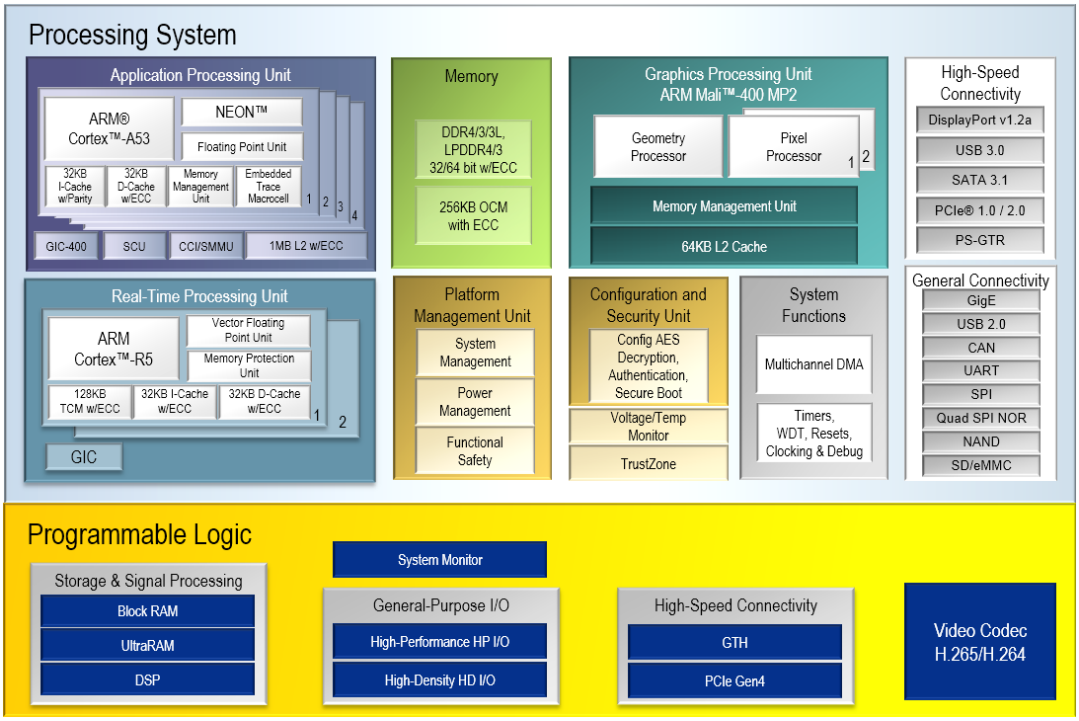
Copyright © 2021 / BME AUT

Az ábrán *MPCore* megnevezés alatt találjuk a CPU magokat, az alsó *Programmable Logic (PL)* elnevezés pedig az FPGA részt jelöli. Az MPCore körül számos periféria csatlakozik a CPU magokhoz AMBA buszok segítségével. A perifériákat és az MPCore-t együttesen *Processing System (PS)*-nek nevezi a Xilinx. Ezek az elnevezések a későbbi architektúrákban is megmaradtak. A PS-ben található perifériákon kívül a PL is tartalmaz hard IP magokat, mint például PCI Express-t, vagy analóg-digitális átalakítót (XADC néven).

15.1.2 Zynq UltraScale+ MPSoC

A legújabb Xilinx FPGA család, az UltraScale+ megjelenésével a Zynq eszközök is fejlődtek. Alapvetően két részre bontották a legújabb Zynq áramköröket. Az általános célú változat *Multi-Processor System-on-Chip (MPSoC)* jelöléssel jelent meg, míg az elsősorban rádiófrekvenciás alkalmazásokhoz készült változat *Radio Frequency System-on-Chip (RFSoC)* névvel jelent meg.

A Zynq UltraScale+ MPSoC blokkvázlata a 15.2. ábrán látható. A PS részben már négy darab ARM Cortex-A53 magot találunk, és mellettük megjelent két, valós-idejű alkalmazásokhoz fejlesztett ARM Cortex-R5 blokk is. Emellett a legnagyobb Zynq UltraScale+ MPSoC áramkörökben egy kétmagos ARM Mali GPU is helyet kapott, mint vektoros végrehajtó egység.



15.2. ábra – Zynq UltraScale+ MPSoC blokkvázlata[2]

[2] Forrás: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html (2020. 05. 26.)

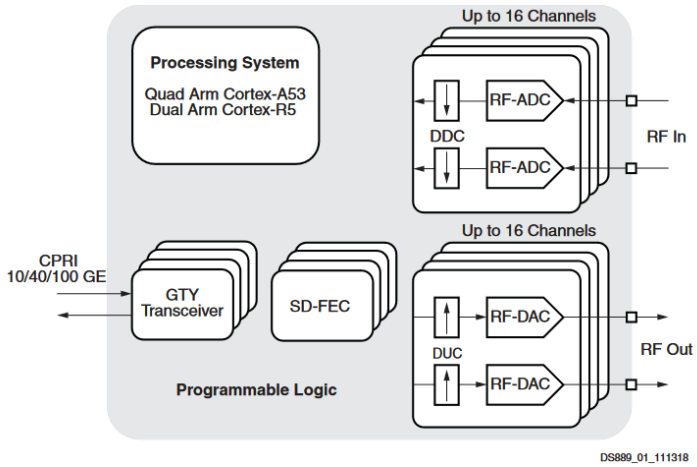
Copyright © 2021 / BME AUT

Az SoC perifériaegységei is fejlődtek, és megjelent például H.264/265 videó kódoló és dekódoló hard IP az áramkörben. Így a Zynq segítségével már bonyolult multimédiás alkalmazásokat meg lehet valósítani minimális kiegészítő hardver segítségével.

15.1.3 Zynq UltraScale+ RFSoC

Ahogy fent említettük, a Zynq UltraScale+ RFSoC eszközök rádiós alkalmazásokhoz készültek. Ennek megfelelően nagysebességű digitál-analóg (DAC) és analóg-digitál (ADC) átalakítókat tartalmaznak (lásd 15.3. ábra). Az ADC átalakítók 5 GSPS (GigaSample Per Second) sebességre képesek, míg a DAC átalakítók akár 10 GSPS sebeségre. Mindkét esetben 14 bites átalakításról van szó.

Az ADC, DAC átalakítókon kívül úgynevezett Soft-Decision Forward Error Correction (SD-FEC) modulok is láthatók a 15.3. ábrán. Az SF-FEC blokk lényege, hogy hiba detektálást és hiba korrekciót tesz lehetővé zajos környezetben átvitt adatok esetén. A módszert a telekommunikációs rendszerek (pl. 5G hálózat) esetén használják, és azért nevezik előremutató hibajavításnak, mivel nincs szükség az átvitt adatok újraküldésére a hibajavításhoz.



15.3. ábra – Zynq UltraScale+ RFSoC blokkvázlata[3]

A Zynq RFSoC alkalmas – rádiós alkalmazásokon kívül – például lézer alapú távérzékeléshez, angolul Light Detection and Ranging (LIDAR) eszközök készítésére is. A LIDAR elv lényege, hogy úgy mérünk távolságot, hogy lézerfényt bocsátunk ki, és mérjük a kibocsátás és a visszaérkezés között eltelt időt. Amennyiben a szenzort egy adott tengely mentén forgatjuk, úgy a síkban található összes tárgy távolságát meg tudjuk mérni. A komolyabb szenzorok több lézersugárral mérnek, és akár a körülöttük lévő teljes háromdimenziós teret letapogatják.

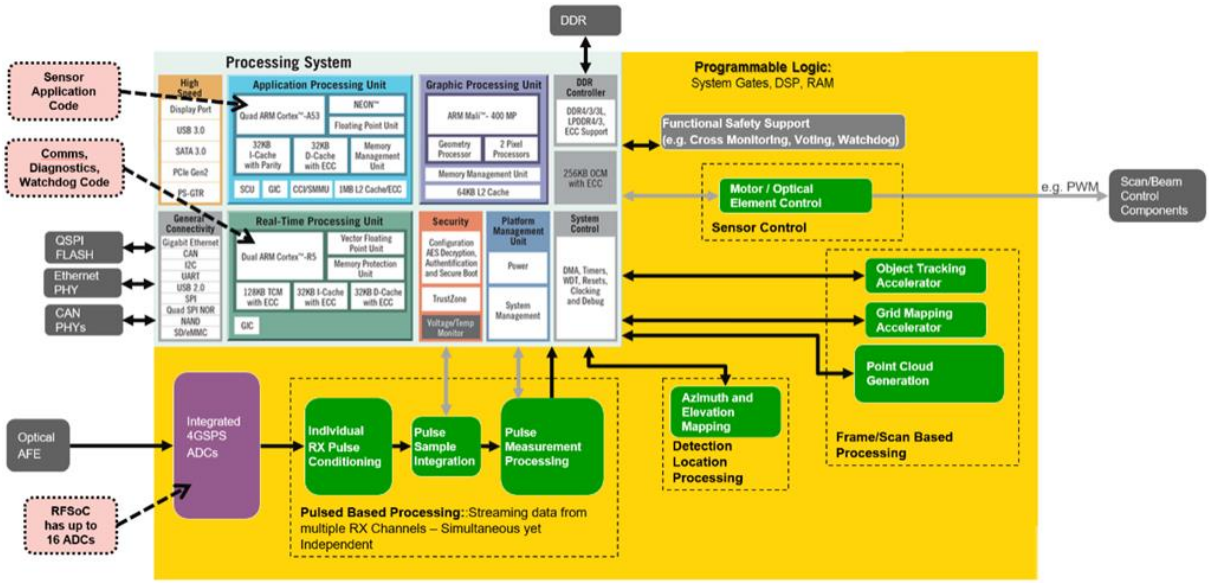
A 15.4. ábrán látható egy Zynq RFSoC alapú példarendszer, amely egy LIDAR mérőeszköz feldolgozóegységét mutatja be. A példa abból a szempontból életszerű, hogy a drágább LIDAR

[3] Forrás: https://www.xilinx.com/support/documentation/data_sheets/ds889-zynq-usp-rfsoc-overview.pdf (2020. 05. 26.)

Copyright © 2021 / BME AUT

szenzorok esetén valóban alkalmaznak FPGA eszközöket. Az ábrán látható rendszer elsősorban a Zynq nagysebességű ADC átalakítóit használja ki.

Az eljárás lényege, hogy a lézersugár kibocsátásakor elindítunk egy számláló egységet, és ezzel párhuzamosan folyamatosan digitalizáljuk a bejövő lézersugarat (megfelelő analóg jelfeldolgozás után). A beérkező jelet összehasonlítjuk egy tárolt referencia jellel, és amennyiben a jel korrelációja a referencia jellel egy bizonyos értékét elér, megállítjuk a számlálót. A számláló aktuális értéke arányos lesz az eltelt idővel. Ezzel a módszerrel, és a Zynq RFSoC áramkörben lévő 4 GSPS-es ADC átalakítóval 10 cm alatti mérési pontosságot tudunk elérni.



15.4. ábra – LIDAR megvalósítása Zynq UltraScale+ RFSoC áramkörrel[4]

15.2 Zynq-7000 SoC

A továbbiakban a legelső Zynq eszközt, a Zynq-7000-t fogjuk részletesen megvizsgálni. Mivel a 12.3. fejezetben a Xilinx 7-es szériás FPGA-k felépítését már tárgyaltuk, ezért elsősorban a processzormagokat tartalmazó PS-re fogunk fókuszálni.

15.2.1 Összehasonlítások

Mielőtt az architektúra részleteibe belemennénk, hasonlítsuk össze a Zynq-7000 teljesítőképességét különböző egyéb eszközökkel.

[4] Forrás: https://developer.xilinx.com/en/articles/lidar-pulse-detection-accelerator-model-based-design-targeting-vitis-and-rfsoc.html (2020. 05. 26.)

Copyright © 2021 / BME AUT

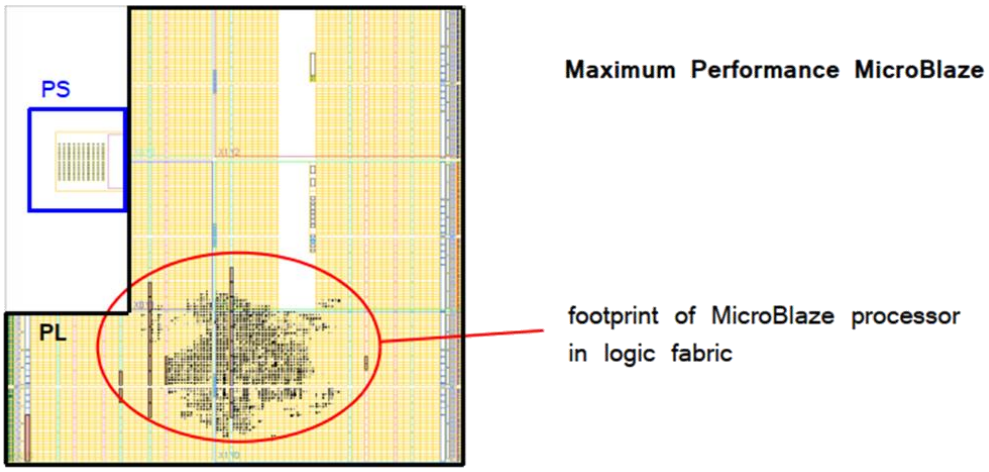
15.2.1.1 A Zynq-7000 összehasonlítása a MicroBlaze soft CPU-val

A soft processzoroknál részletesen bemutattuk a MicroBlaze IP-t, így adódik a lehetőség, hogy elsőként hasonlítsuk össze a Xilinx két CPU megoldását. A következő táblázatban elsősorban a két termékben található processzoros részt hasonlítjuk össze:

	Teljesítmény	Max. órajel	Terület	Rugalmasság	Szoftver
MicroBlaze	Kb. 500 DMIPS	Kb. 500 MHz	1-4 egység	Maximális	GCC, Linux
Zynq-7000 (ARM Cortex-A9)	Kb. 2500 DMIPS/mag	Kb. 1 GHz	1 egység	Korlátozott	

15.1. táblázat – Xilinx Zynq-7000 és a MicroBlaze összehasonlítása

A táblázat alapján látható, hogy a Zynq lényegesen nagyobb teljesítményre képes, ráadásul kevesebb helyet foglal a hard IP kialakítás miatt (a táblázat terület oszlopában lévő egységet relatív egységként kell értelmezni). Mivel a MicroBlaze teljesítménye és területe is implementációfüggő, így fontos kiemelni, hogy ebben az esetben a teljesítményre optimalizált MicroBlaze konfigurációt vizsgáltuk. A terület alakulása a két IP esetén a 15.5. ábrán is látható. Kékkel szerepel a PS, míg piros kiemeléssel látható a MicroBlaze.



15.5. ábra – Zynq-7000-ben implementált MicroBlaze IP[18]

A rugalmasság szempontjából a MicroBlaze mindenképp felülmúlja a Zynq-et a soft kialakítás miatt. Ebből a szempontból kiemelendő, hogy a Zynq esetén is lehetőségünk van a belső AMBA buszrendszerrel rugalmas kapcsolatokat kialakítani az SoC részegységein keresztül, valamint a fogyasztás minimalizálása miatt ki is tudjuk kapcsolni a nem használt blokkokat.

Szoftveres oldalról nézve, a két eszköznek hasonló támogatottsága van. Mindkettő esetén rendelkezésünkre áll GCC alapú programcsomag a fejlesztéshez, és Linux operációs rendszer

Copyright © 2021 / BME AUT

is. Mivel az ARM processzorok lényegesen elterjedtebbek, így ott azért valamivel több szoftveres megoldás áll a rendelkezésünkre.

15.2.1.2 A Zynq összehasonlítása dedikált mikrovezérlőkkel

A dedikált mikrovezérlők manapság nagyon népszerűek a beágyazott rendszerek világában. Sok gyártó fejleszt és árul különböző felépítésű vezérlőket, de a legtöbb esetben ARM Cortex-M maggal rendelkező eszközöket találunk. A következő táblázatban az STMicroelectronics és az AMD cég termékeivel hasonlítjuk össze a Zynq SoC-t:

	Teljesítmény	Max. órajel	Költség
STM32H7 ARM Cortex-M7	Kb. 3200 CoreMark	480 MHz + 240 MHz	Min. 2 000 Ft
STM32F4 ARM Cortex-M4	Kb. 600 CoreMark	180 MHz	Min. 1 500 Ft
Xilinx Zynq-7000 (ARM Cortex-A9)	Kb. 6000 CoreMark	Kb. 1 GHz	Min. 15 ezer Ft
Xilinx Zynq UltraScale+ (ARM Cortex-A53)	Kb. 16 000 CoreMark	Kb. 1,2 GHz	Min. 70 ezer Ft
AMD Ryzen 1700X	Kb. 310 000 CoreMark	Kb. 3,8 GHz	Min. 60 ezer Ft

15.2. táblázat – Xilinx Zynq és néhány dedikált mikrovezérlő összehasonlítása

A táblázatban szereplő teljesítményértékek ebben az esetben nem DMIPS, hanem az úgynevezett *CoreMark* mértékegységben szerepelnek, mivel az ST mikrovezérlők esetén ezeket adják meg.

A táblázat utolsó eleme egy asztali CPU, amely nem mikrovezérlő, de jól mutatja a különbséget egy nagyteljesítményű, asztali processzor és a beágyazott eszközökben szereplő megoldások között, és utal valamennyire a CoreMark mérőszám nemlinearitására is.

A táblázatban szerepel a legújabb Zynq eszköz is, hogy látható legyen a fejlődés a Zynq sorozaton belül. Valamint az összehasonlítás így korrekt, mert az ST processzorok esetén is szerepel a legújabb M7-es sorozat.

A táblázatban elsősorban a teljesítményt és az áramkör árát hasonlítottuk össze, mivel a terület összehasonlításnak nincs értelme, mert mindegyik esetben hard IP-ről van szó, és a szoftveres támogatottsága mindegyik eszköznek azonos az ARM mag miatt.

Az értékek alapján látható, hogy az ST processzorok számítási teljesítményben nem maradnak el lényegesen a legelső Zynq eszközben található processzortól, ráadásul ár szempontjából mindenképp előnyösebbek. A legújabb Zynq teljesítményben nagy ugrást jelent az előző generációhoz képest, viszont árban is jelentősen drágább. A legújabb Zynq kezdő ára már több

Copyright © 2021 / BME AUT

mint egy nagyságrenddel nagyobb, mint a Cortex-M processzorokat tartalmazó ST mikrovezérlőké.

15.2.1.3 A Zynq-7000 összehasonlítása más technológiai megoldásokkal

Az összehasonlításokat azzal zárjuk, hogy megnézzük, más technológiákhoz képest milyen előnyei, hátrányai vannak a Zynq heterogén kialakításának. A 15.6. ábrán látható összehasonlítást a Xilinx készítette, és alapvetően négy különböző technológiát vizsgál. Az *Application-Specific Integrated Circuit (ASIC)* elnevezés az egyedi fejlesztésű célhardvereket jelöli, az *Application-Specific Standard Product (ASSP)* elnevezés pedig a készen kapható, standard célhardvereket. Az ábrán megjelenő *2 Chip Solution* pedig azt jelöli, amikor külön áramkörben helyezkedik el a processzormag és az FPGA.

A 15.6. ábra alapján megállapítható, hogy habár az ASIC eszközök teljesítményben és fogyasztásban mindenképp kiemelkednek a többi megoldáshoz képest (valamennyire a Zynq-hez képest is), viszont egy ASIC áramkör kifejlesztése és gyártása nagyon sok időt és pénzt emészt fel szemben a többi technológiával. Emellett egy elkészült ASIC esetén már módosítani sem tudunk, így a rugalmassága mindenképp alacsonyabb a többi technológiához képest.

Az ASSP megoldás esetén a fejlesztés és gyártás hátrányai nem jelennek meg, hiszen már valaki más kifejlesztette és legyártotta az eszközöket, mi már csak használjuk az elkészült technológiát. Viszont a rugalmasság ezeknél az eszközöknél sem biztosított, mert az eszköz belső működése nem módosítható.

	ASIC	ASSP	2 Chip Solution	Zynq
Performance	+	+	■	+
Power	+	+	–	+
Unit Cost	+	+	–	■
Total Cost of Ownership	■	+	+	+
Risk	–	+	+	+
Time to Market	–	+	+	+
Flexibility	–	–	+	+
Scalability	–	■	+	+

+ positive, – negative, ■ neutral

15.6. ábra – A Zynq összehasonlítása más technológiákkal[5]

A külön processzor és külön FPGA áramkör esetén már a Zynq-hez hasonló rugalmasságot kapunk, és a fejlesztés költsége és ideje is alacsonyabb az ASIC technológiához képest. A

[5] Forrás: Xilinx Video Tutorials (2020. 05. 27.)

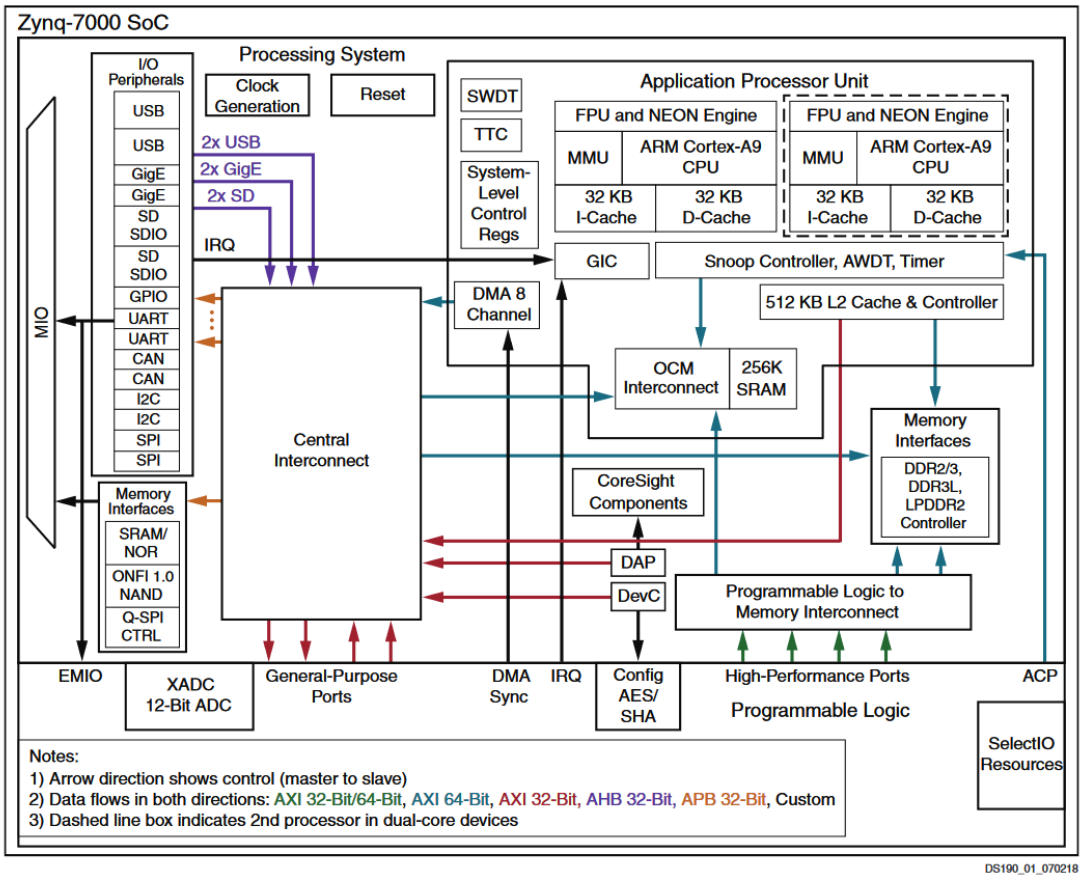
Copyright © 2021 / BME AUT

megoldás hátránya lehet, hogy a két dedikált áramkör beszerzési költsége viszont magasabb, mint a célhardverek költsége. Ezenkívül a két áramkörös megoldás biztosan alacsonyabb teljesítményre és magasabb fogyasztásra lesz képes, mint a célhardveres megoldások.

A teljesítmény és fogyasztás szempontjából a Zynq is biztosan elmarad az ASIC, ASSP technológiákhoz képest, annak ellenére, hogy a Xilinx által készített táblázatban ez nem jelenik meg. Valamint a Zynq másik hátránya a viszonylag magas ára lehet. Amennyiben nagy darabszámú eszközről van szó, a célhardverek lényegesen kedvezőbb egységárat tudnak elérni.

15.2.2 Zynq-7000 PS felépítése

A Zynq-7000 részletes ismertetését a processzoros rész (PS) tárgyalásával kezdjük. A PS elemei megfelelnek egy dedikált processzor áramkörben található blokkoknak (pl. processzormag, perifériák), csak ebben az esetben a blokkok közötti összeköttetés rugalmasabban alakítható.



15.7. ábra – Zynq-7000 SoC belső felépítése[6]

[6] Forrás: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf (2020. 05. 27.)

Copyright © 2021 / BME AUT

A 15.7. ábrán látható az egylapkás rendszer részletes blokkvázlata. Az ábrán különböző színnel jelölték a különböző AMBA buszokat, és az ábra alsó részén látható, hogy melyik szín pontosan milyen buszt jelöl.

Az ábra felső része mutatja a PS felépítését, az alsó részen pedig a PL-ben található egységeket látjuk. A blokkvázlat jobb felső sarkában lévő *Application Processor Unit (APU)* tartalmazza a CPU magokat, valamint a processzorhoz tartozó legfontosabb blokkokat. Ezenkívül kiemelendő még a *Central Interconnect* elnevezésű blokk, amely a különböző egységek rugalmas összekötését teszi lehetővé, valamint a bal oldalon található I/O perifériák. Az I/O perifériák az úgynevezett *Multiplexed I/O (MIO)* egységen keresztül kapcsolódnak az áramkör I/O lábaihoz.

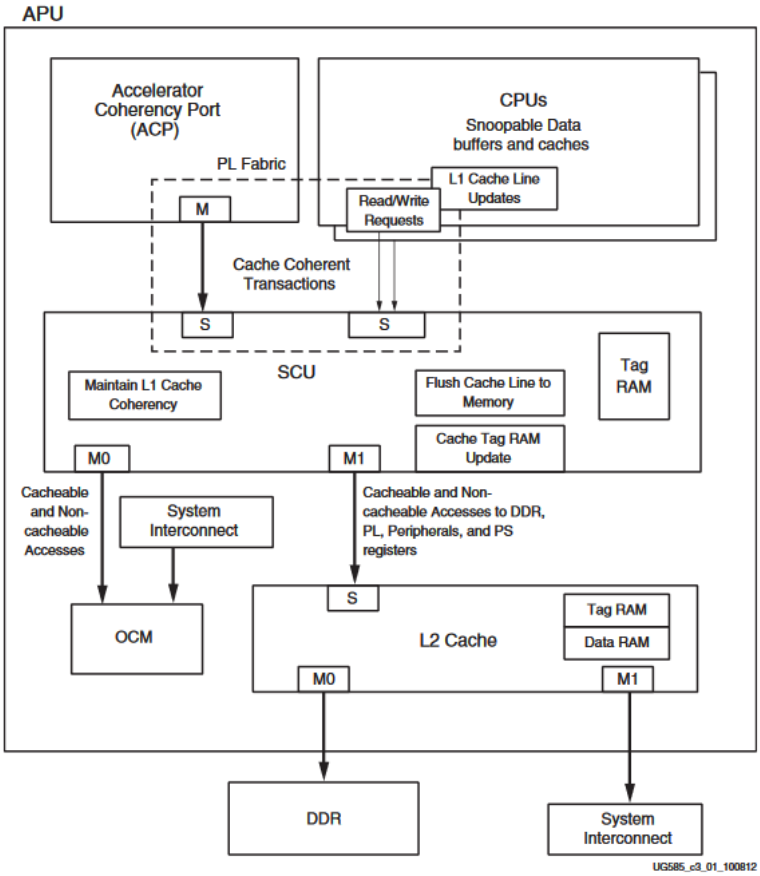
15.2.2.1 Application Processor Unit (APU)

Az APU blokk legfontosabb egységei az ARM Cortex-A9 CPU magok (lásd 3.3.2. fejezet). A processzormagok legfeljebb 1 GHz frekvenciával képesek működni, tartalmaznak dupla pontosságú FPU végrehajtót, NEON SIMD végrehajtót, és MMU egységet is. Ahogy a 15.7. ábrán látható, mindkét CPU-hoz tartozik saját utasítás és adat gyorsítótár is (4-utas csoportos asszociatív), amelyek L1 szintű gyorsítótárnak felelnek meg. Az L1 gyorsítótárral szemben az L2 gyorsítótárból már csak egy található az APU-ban, és a gyorsítótárak kezeléséhez tartozik a *Snoop Controller* egység is, amelyről már az 4. fejezetben esett szó. A memória egységekhez tartozik még egy *On-Chip Memory (OCM)* elnevezésű RAM blokk is, amelyet még a későbbiekben megvizsgálunk.

A processzormagokon kívül az APU tartalmaz még néhány CPU-hoz szorosan kötődő perifériát, például időzítőket (Triple Timer Counters (TTC) az ábrán), Watchdog áramköröket (System Watchdog Timer (SWDT) az ábrán), és megszakítás vezérlőt (Generic Interrupt Controller (GIC) az ábrán).

A 15.8. ábrán látható, hogy miként épül fel az APU gyorsítótár rendszere. Az L1 szintű tárak a *Snoop Controller Unit (SCU)* egységen keresztül érik el az L2 szintet, és az OCM memóriát is. Az L2 gyorsítótár pedig eléri az DDR memóriákhoz tartozó vezérlőt, és a központi összekötő blokkot is. A központi összekötő ezen az ábrán *System Interconnect* néven szerepel, míg a korábbi 15.7. ábrán *Central Interconnect* elnevezéssel szerepelt.

Copyright © 2021 / BME AUT



15.8. ábra – APU-ban található AXI buszok[6]

15.2.2.2 Accelerator Coherency Port

A 15.8. ábrán szereplő SCU egységhez kapcsolódik az Accelerator Coherency Port (ACP) is. Ennek az ARM által készített IP-nek a feladata, hogy gyorsítótár koherens kapcsolatot teremtsen a Zynq PS és PL része között. A PS és PL között több AXI busz is található, de csak az ACP port segítségével tudunk úgy átvinni adatokat, hogy az L1 és L2 gyorsítótárak tartalma mindvégig koherensek maradnak. Az ACP, a PL oldalán AXI master portként használható, és 64 bites adat tud átvinni. Mivel az SCU hozzáfér az L2 vezérlőn keresztül a központi összekötőhöz, így a PS-ben található perifériákat is elérjük a PL-ből.

Az ACP gyakorlati használatára tekintsük a következő egyszerű példát:

1. A CPU előkészíti a PL számára a feldolgozandó adatot
2. A CPU jelez a PL számára pl. általános célú AXI porton (lásd később)
3. A PL kiolvassa az adatokat az ACP-n keresztül
4. A PL elvégzi a megfelelő műveletet az adatokon, és visszaírja őket az ACP-n keresztül
5. A PL jelez a CPU számára, hogy befejezte az adatok feldolgozását pl. megszakítás vonalon keresztül

Copyright © 2021 / BME AUT

6. A CPU használhatja a gyorsítótár koherens adatokat, nem szükséges érvényteleníteni a gyorsítótárakat

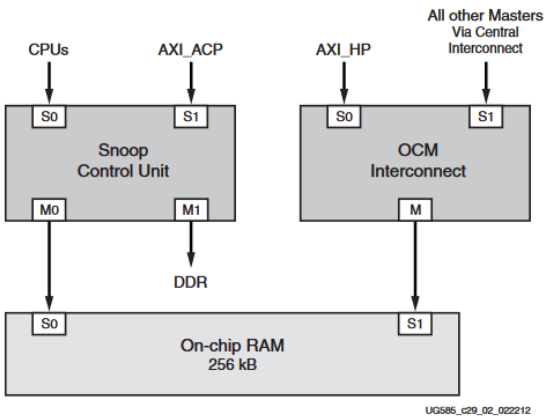
A példa alapján a legfontosabb előnye az ACP portnak, hogy egy külső AXI master egységet (ami jelen esetben a PL-ben található) gyorsítótár koherens módon kapcsol a PS többi eleméhez.

15.2.2.3 Memória interfészek

Az APU esetén már említettük, hogy többféle memóriához is hozzáférnek a CPU magok. Most ezeket a memória interfészeket vizsgáljuk meg.

OCM interfész

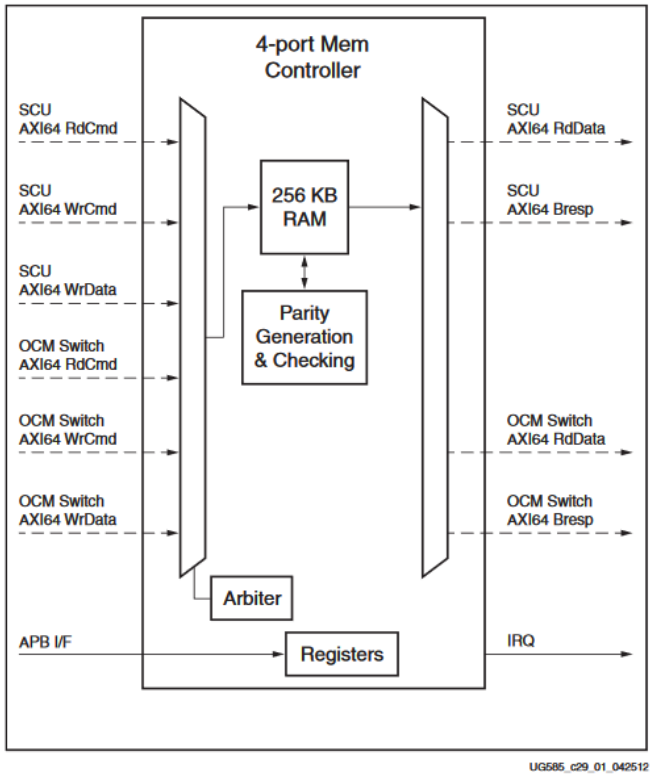
Ahogy a korábbiakban említettük, az On-Chip Memory (OCM) belső RAM memóriát jelent, amely az SCU egységen keresztül kapcsolódik a CPU magokhoz. Ezenkívül az OCM memóriának van egy dedikált interfésze is, amelyen keresztül bekapcsolódik a központi összekötőbe. Ezek a kapcsolatok láthatók a 15.9. ábrán is. Az ábrán látható, hogy az *OCM Interconnect* egység az AXI_HP porttal is kapcsolatban áll a központi összekötőn kívül. Az AXI_HP a PL-ben található AXI port, amelyről még a későbbiekben szó lesz. Ezek alapján a PL-ből mind az ACP porton mind az AXI_HP porton keresztül elérhető az OCM.



15.9. ábra – A Zynq OCM memória portjai[6]

Az OCM memória belső felépítése a 15.10. ábrán látható. A memória 256 kB RAM-mal rendelkezik, és a két AXI porthoz összesen 10 db csatorna tartozik (ezek láthatók az ábrán), mivel az AXI busz alapvetően öt csatornából áll (lásd 13.2.3. fejezet). Fontos kiemelni, hogy a 256 KB RAM-on kívül található 128 KB ROM is az OCM egységben, amelyet nem mutat az ábra. Ez a ROM nem elérhető a felhasználó számára, és a boot folyamathoz tartozó programot tartalmazza (BootROM).

Copyright © 2021 / BME AUT



15.10. ábra - Az OCM memória belső felépítése[6]

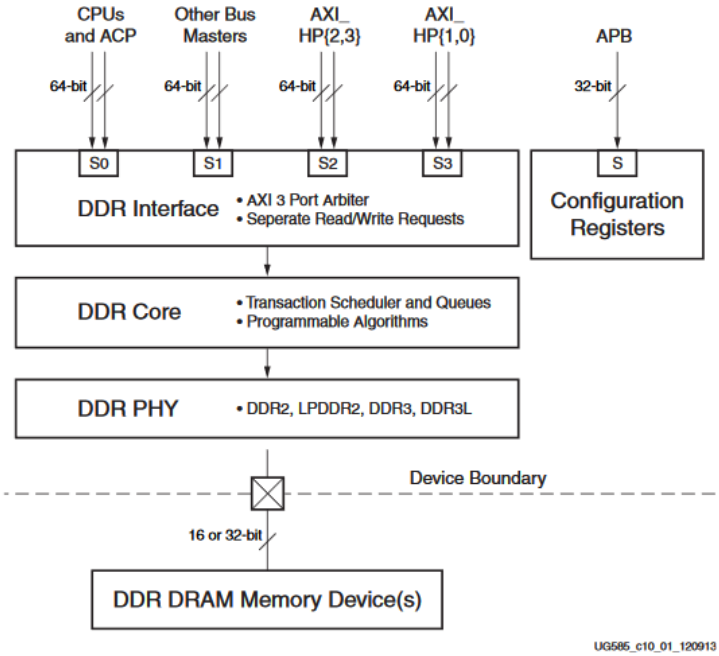
A 15.10. ábra alapján látható, hogy a két AXI busz miatt arbitráció is történik a portok között, valamint a memória paritást is számol és ellenőriz a memória műveletekhez. Az OCM memória támogatja az ARM fejezetben bemutatott TrustZone kiegészítést is.

DDR interfész

Az OCM memória viszonylag korlátozott mennyiségű adatot képes tárolni, így sok esetben külső memóriát is kell illeszteni a Zynq-hez. A PS-ben található DDR vezérlő egységgel különböző DDR2 és DDR3 memóriákat tudunk használni. A vezérlő a memóriákhoz tartozó PHY blokkot is tartalmazza, így nem szükséges külső áramkört elhelyezni a memóriák kezeléséhez.

A DDR vezérlő a Zynq többi részegységéhez négy db AXI porton keresztül kapcsolódik, és egy APB porttal is rendelkezik a vezérlő beállításához. A DDR vezérlő IP a 15.11. ábrán látható. Az AXI portok közül egy csatlakozik a CPU-khoz az SCU egységen keresztül, egy port a központi összekötőhöz tartozik, és két AXI_HP port pedig a PL-hez kapcsolódik.

Copyright © 2021 / BME AUT



15.11. ábra – A Zynq PS DDR interfésze[6]

Az AXI busz vizsgálatánál, a 13.2.3. fejezetben bemutattuk, hogy a busz képes sorrenden kívül végrehajtani az átviteleket (OoO átvitel). A DDR vezérlő képes kihasználni ezt a képességet, és más sorrendben végrehajtani a kapott kéréseket, ha a DDR memória szempontjából ez előnyösebb. Az OoO átvitelen kívül burst átviteleket is támogat a DDR vezérlő, maximum 16 hosszú átvitelt képest fogadni.

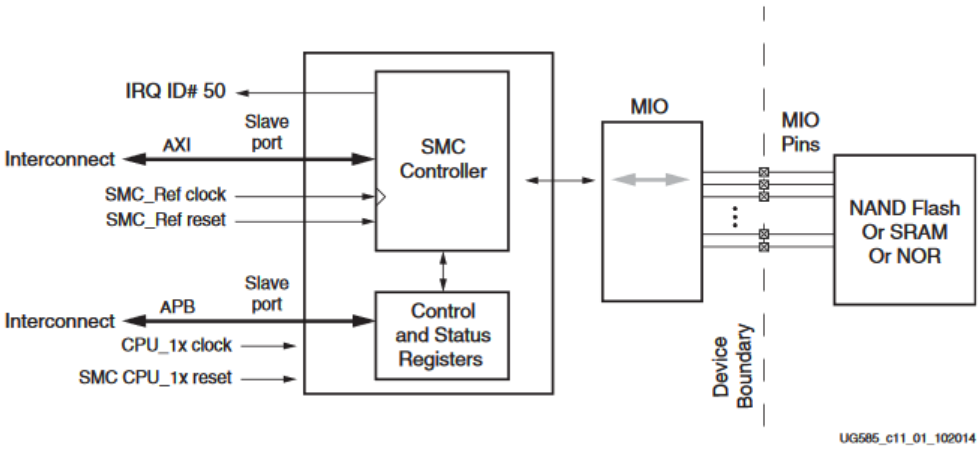
Az OCM memóriához hasonlóan a DDR vezérlő is tartalmaz arbitert, amely eldönti, hogy a négy port közül melyiktől fogad kéréseket. A döntésnél figyelembe veszi azt is a vezérlő, hogy mióta várakozik az adott AXI master.

Statikus memória vezérlő (SMC)

Az előző részben megnéztük, hogy dinamikus memóriákat hogyan tudunk a Zynq-hez illeszteni. Viszont sok esetben használunk más típusú memóriákat is beágyazott környezetben. A Zynq PS részében található *Static Memory Controller (SMC)* egység képes statikus RAM (SRAM) és flash memóriákat (NOR and NAND típust) is kezelni.

Az SMC blokkvázlata a 15.12. ábrán látható. A vezérlő a Zynq PS részének bal oldalán helyezkedik el a 15.7. ábrán látott áttekintő blokkvázlatban, és a MIO egységen keresztül kapcsolódik a Zynq-hez illesztett memóriához. A Zynq-en belül pedig a központi összekötőn keresztül érhető el az SMC. Ahogy a DDR esetén láttuk, itt is egy független APB buszon keresztül lehet beállítani a vezérlőt, de az adatok átvitele AXI buszon keresztül történik.

Copyright © 2021 / BME AUT



15.12. ábra – SMC vezérlő[6]

Quad-SPI vezérlő

A MicroBlaze esetén már bemutattuk az SPI protokollt, amely kis lábszámú, SPI flash memóriaegységek illesztéséhez használható, és quad SPI módban többszörös sávszélesség is elérhető a normál SPI-hoz képest.

A Zynq PS egysége is rendelkezik quad-SPI vezérlővel. A vezérlő felépítése megfelel az SMC vezérlő felépítésének. Itt is a MIO porton tudjuk a külső flash memóriát illeszteni, és a Zynq-en belül egy AXI és egy APB busz áll rendelkezésre, ahogy az SMC esetén is volt.

A quad-SPI vezérlőnél érdemes megemlíteni, hogy alapvetően három módban használható. Az *SPI mód* normál SPI átviteleket tesz lehetővé, tehát ebben az esetben nem használja ki a vezérlő, hogy az SPI kommunikáció másik oldalán flash memória található, így akár más SPI eszközt is kezelhetünk. Ezzel ellentétben a *lineáris címzési módban* memória olvasási parancsokat fogad a vezérlő az AXI buszon keresztül, és teljesen elrejti a külső flash memória kezelését a felhasználó elől. A két mód között helyezkedik el az *I/O mód*, amely esetén a flash memória kezelésével szükséges parancsokat (pl. formázás) a vezérlőt használó programnak kell kiadnia.

15.2.2.4 Kommunikációs interfészek

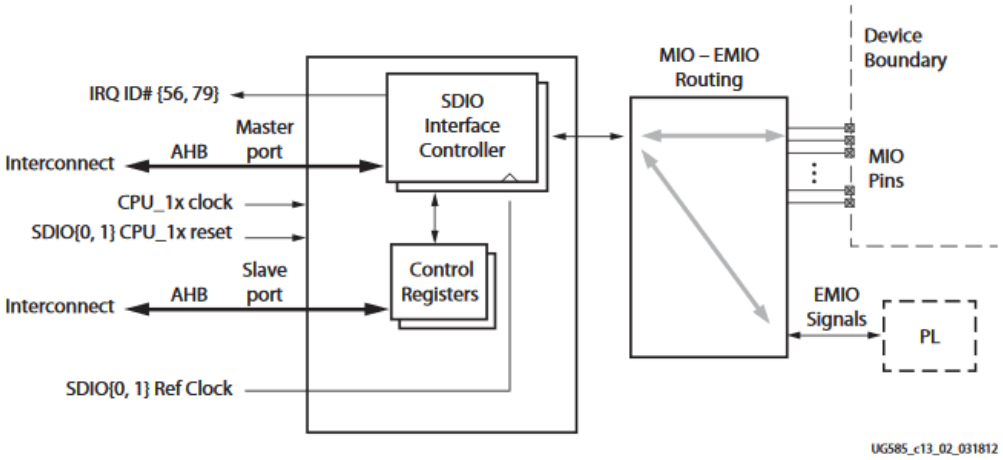
Most pedig röviden áttekintünk még néhány fontosabb kommunikációs perifériát a Zynq PS-ben, amelyek segítségével különböző eszközök illeszthetők az SoC-hez.

SD/SDIO vezérlő

A memória interfészekhez kapcsolódik, hogy *Secure Digital (SD)* memóriakártyákat is tudunk illeszteni a rendszerünkhöz. Ehhez dedikált SD vezérlő modul található a PS-ben. A *Secure Digital Input Output (SDIO)* szabványnak köszönhetően a memóriakártyákon kívül egyéb eszközöket is lehet illeszteni a vezérlőhöz. Például GPS, Wifi, Bluetooth és számtalan más eszköz használható az SDIO segítségével.

Copyright © 2021 / BME AUT

A 15.13. ábrán látható az SD/SDIO vezérlő blokkvázlata. Az ábra alapján látható, hogy ebben az esetben két AHB buszon keresztül érjük el a modult. A másik érdekesség, hogy a perifériát a PL-n keresztül is kivezethetjük a Zynq megfelelő lábaihoz, amennyiben a MIO-hoz tartozó lábak már mind foglaltak. Ezt az *Extended Multiplexed I/O (EMIO)* blokk segítségével tehetjük meg.



15.13. ábra – SD/SDIO vezérlő[6]

USB vezérlő

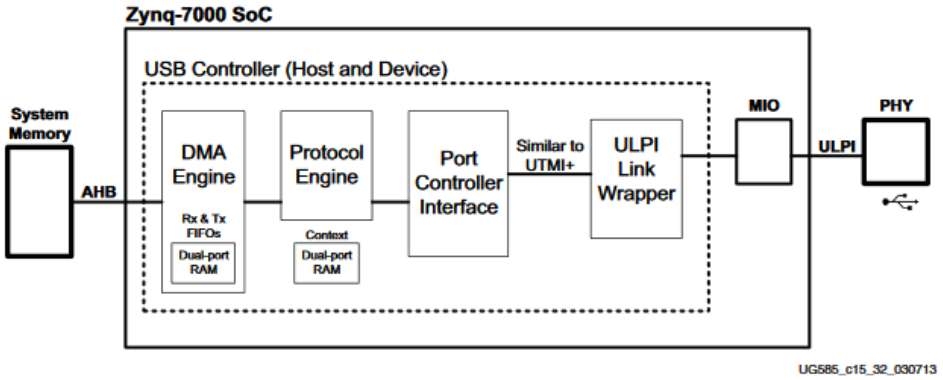
A 8. fejezetben bemutatott USB interfész a beágyazott eszközök világában is elterjedté vált. Ennek megfelelően a Zynq-7000 esetén is találunk két darab USB 2.0 vezérlőt.

A vezérlők képesek host módban külső periféria eszközt kezelni, de lehetőségünk van arra is, hogy külső host eszközhöz illesszük a Zynq-et, mint device-t. A két módot ötvözi az úgynevezett USB On-the-Go (OTG) mód, amely esetén a fenti két mód menet közben átváltható. Az OTG-t mobiltelefonok esetén is gyakran használják, például pendrive-ot csatlakoztathatunk a telefonhoz, de a telefon is használható adattárolóként egy PC mellett.

Az USB használatához külső áramkör szükséges, amely megvalósítja a fizikai (PHY) rétegnek megfelelő logikát. A Zynq USB vezérlője és a PHY között az *UTMI+ Low Pin Interface (ULPI)* protokoll teremt kapcsolatot (lásd 15.14. ábra).

A Zynq oldalán, az SD/SDIO vezérlőhöz hasonlóan, ebben az esetben is az AHB buszra kapcsolódik a vezérlőegység. A 15.14. ábra alapján látható, hogy az ULPI jelek csak a MIO-n keresztül használhatók, viszont egyéb vezérlési jelek az EMIO segítségével kerülnek kivezetésre.

Copyright © 2021 / BME AUT

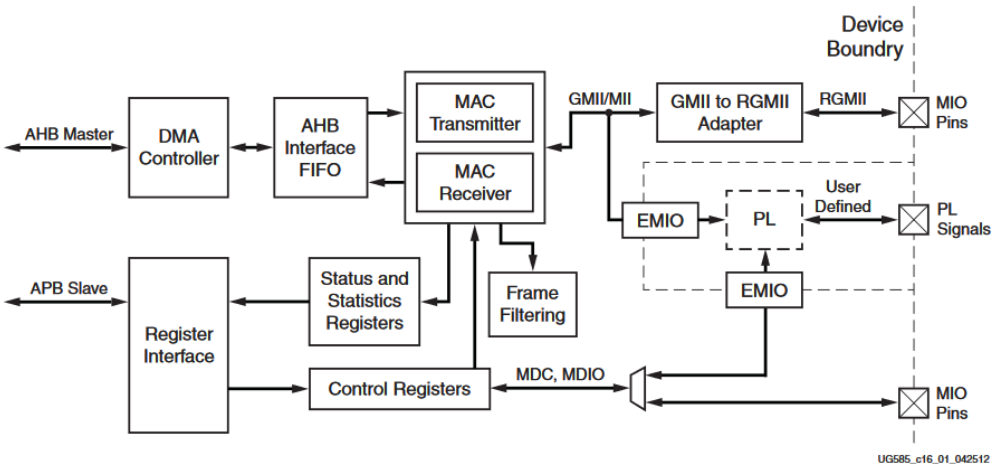


15.14. ábra – USB vezérlő[6]

Ethernet vezérlő

A 14.1.3. fejezetben röviden ismertettük, hogy FPGA esetén hogyan lehet Ethernet MAC interfészt használni az FPGA általános erőforrásait felhasználva. A Zynq viszont rendelkezik két darab hard Ethernet MAC vezérlővel is a PS-ben. Ezek az egységek akár 1 Gb/s átvitelű kapcsolatot is lehetővé tesznek, külső PHY áramkör segítségével.

Az Ethernet vezérlőhöz tartozó blokkvázlat a 15.15. ábrán látható. Ahogy azt már más perifériáknál is láttuk, az adatátvitel ebben az esetben is AHB buszon, a konfiguráció APB buszon keresztül történik. A PHY egységgel a *Media-independent interface (MII)* protokoll teremt kapcsolatot, amelynek több fajtáját is használhatjuk a Zynq-en belül. A MIO-n keresztül egy egyszerűsített változat, a Reduced Gigabit MII (RGMII) megoldást használhatjuk, míg a PL-n keresztül saját interfészt is kialakíthatunk, amely MII-n keresztül tart kapcsolatot a MAC vezérlővel (lásd 15.15. ábra).



15.15. ábra – Ethernet vezérlő[6]

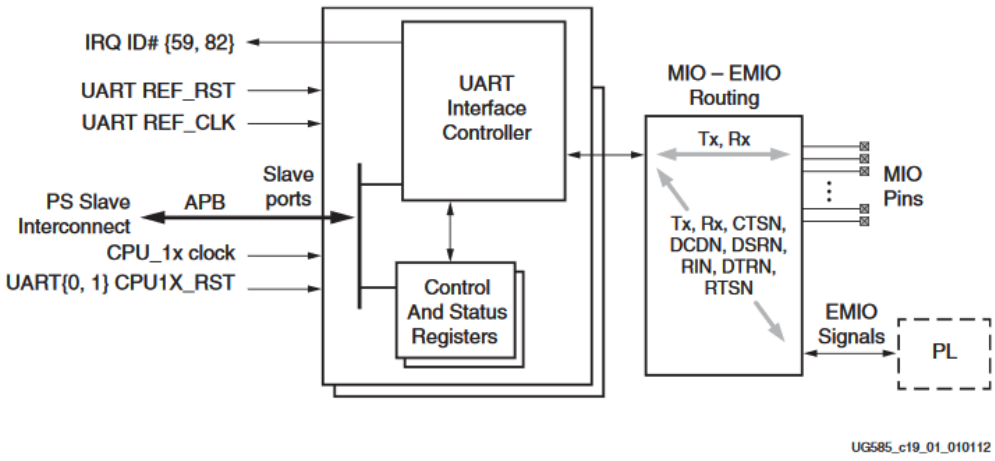
Copyright © 2021 / BME AUT

Egyéb I/O perifériák

Az eddig bemutatott perifériákon kívül még néhány egyéb I/O perifériát is tartalmaz a Zynq-7000 SoC. Ezeket a 15.7. ábra bal oldalán találjuk.

A már bemutatott quad-SPI modulon kívül találunk két darab normál SPI perifériát is, és két darab *Controller Area Network (CAN)* perifériát is. Ezekkel a perifériákkal számtalan áramkör illeszthető a rendszerhez. A PS oldalán mindkét interfész FIFO puffereken keresztül kapcsolódik az APB buszra. A másik oldalon használhatjuk a MIO-t, vagy lehetőségünk van a PL-n keresztül is csatlakoztatni a külső egységeket a perifériákhoz. Az SPI vezérlővel maximum 50 MHz-es átvitellel tudunk kommunikálni, a CAN esetén 1 Mb/s a maximális adatátvitel.

A beágyazott rendszerekben gyakran alkalmazzák az *Inter-Integrated Circuit (I2C)* és a *Universal Asynchronous Receiver-Transmitter (UART)* kommunikációs interfészeket is. Hasonlóan az SPI és CAN perifériákhoz, ebben az esetben is 2-2 vezérlőt találunk a Zynq-ben. Továbbá itt is az APB buszon keresztül érjük el az egységeket, és a másik oldalon a MIO vagy a PL is elérhető. Az UART vezérlő esetén a 15.16. ábrán láthatjuk az előbb felsorolt lehetőségeket.



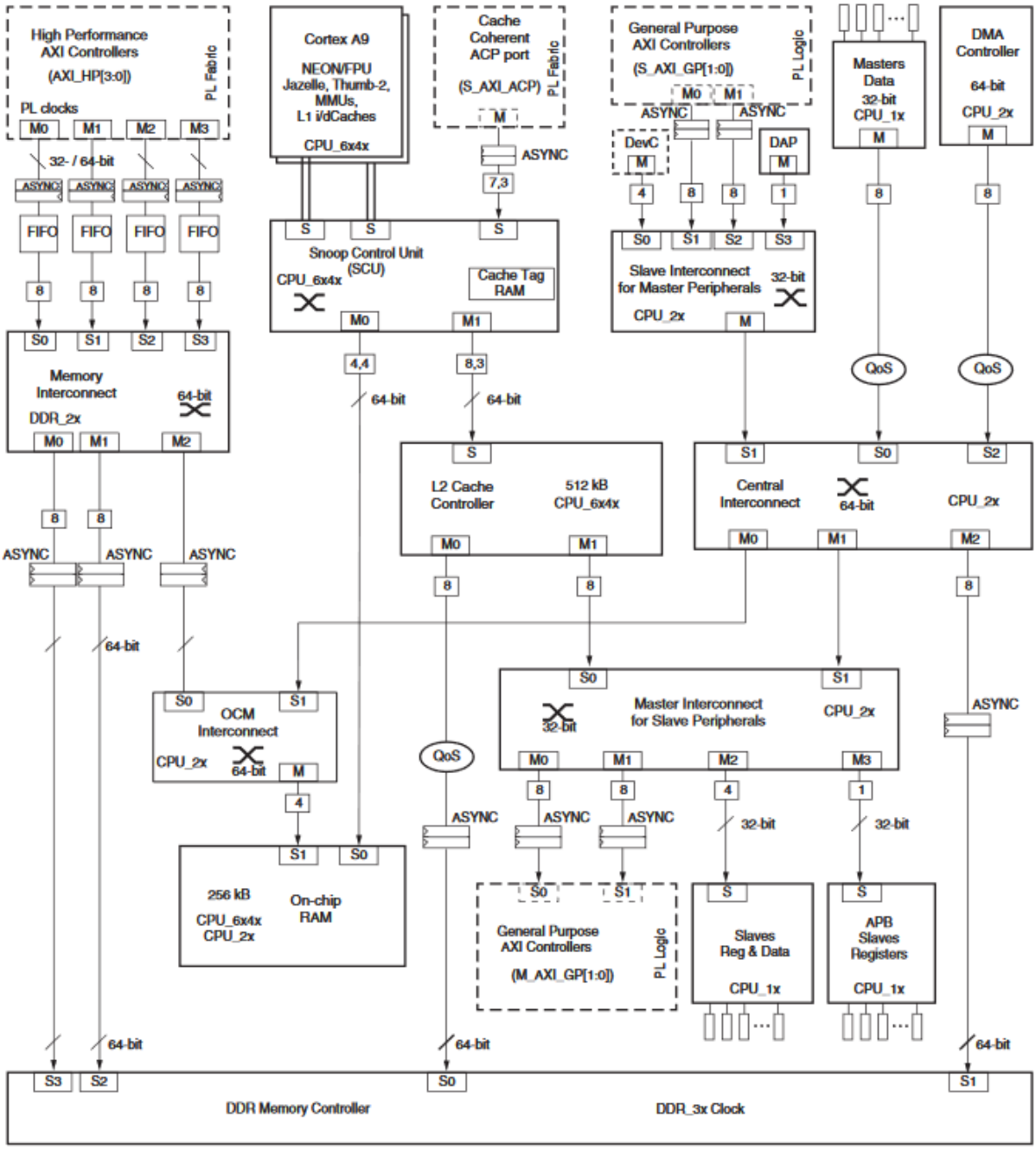
15.16. ábra – UART vezérlő[6]

15.2.2.5 Összeköttetések

A Zynq-7000 rendszerben viszonylag bonyolult a különböző blokkok közötti kapcsolat. Valamelyest segíti az eligazodást a 15.17. ábra, amely a teljes rendszer buszhálózatát mutatja. A szaggatott vonallal jelzett blokkok a PL-ben, a folytonos vonallal jelzett blokkok pedig a PS-ben található egységek.

Copyright © 2021 / BME AUT

Az ábrán szereplő *async* jelölések azt mutatják, hogy a kapcsolatban álló blokkok független, aszinkron órajelet használnak. Ez azt jelenti, hogy a két órajel közötti fázist előre nem ismerjük, ha a jelek frekvenciáját tudjuk is. Emiatt mindenképp szinkronizációs logikára van szükség, hogy elkerüljük a metastabil állapotot (lásd 11.3.3.1. fejezet). A Zynq buszrendszere megoldja a szinkronizációt az async modulok között, így felhasználói oldalról nem szükséges ezt kezelni.



15.17. ábra – A Zynq-7000 buszhálózata[6]

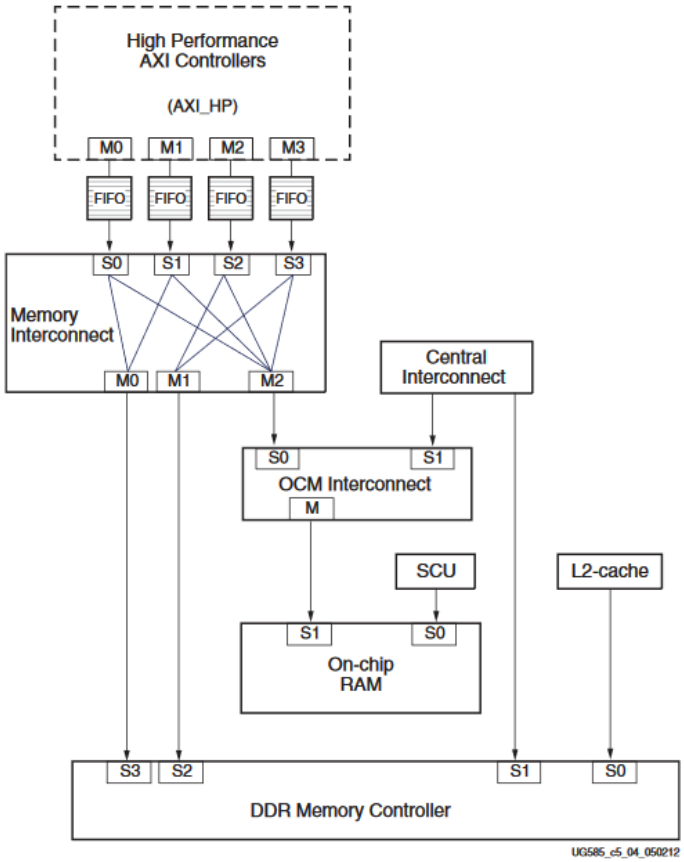
Copyright © 2021 / BME AUT

Az ábrán a blokkok között láthatunk egy vagy két számot is bekeretezve. Ezek azt jelzik, hogy az adott buszon hány darab olvasási és írási parancsot küldhet ki a master egymás után. Például az alul található DDR vezérlő esetén 8 db egymást követő írási vagy olvasási parancsot küldhetünk az AXI portokon.

A Zynq-7000 összekötő hálózata alapvetően három részre osztható. A processzorokhoz szorosan kapcsolódik az SCU és az L2 gyorsítótár (L2 Cache Controller néven). Ezek a processzorhoz hasonlóan ARM IP-k. Ezenkívül a legtöbb általános célú összekötő blokkot az ARM NIC-301-es IP megoldása biztosítja. Az ARM NIC-301 segítségével bonyolult buszhálózatot lehet kiépíteni akár 128 master és 64 slave egység között. A buszhálózat az AMBA család különböző buszait támogatja. A Zynq esetében ez az IP valósítja meg az összes olyan blokkot, amelyben az *interconnect* megnevezés szerepel, valamint az AHB, APB buszhoz köthető bridge blokkokat is. A Zynq buszhálózatának harmadik része a PL és PS közötti összeköttetések, amelyet a következőkben ismertetünk.

A PL és a PS közötti összeköttetések

Az ACP portról már korábban esett szó, amely gyorsítótár koherens kapcsolatot teremt a PL és a PS között. Ebben az esetben a PL-ben AXI master port áll a rendelkezésünkre.



15.18. ábra – AXI_HP kapcsolódási lehetőségek[6]

Copyright © 2021 / BME AUT

Említés szintjén már kitértünk az *AXI High Performance (AXI_HP)* kapcsolatra, amely nagy sebességű kapcsolatot teremt a PS és PL között. A 15.18. ábrán látható, hogy a négy darab AXI_HP port segítségével milyen egységeket érhetünk el a PS-ben. Látható, hogy a DDR és az OCM memóriát tudjuk az AXI_HP segítségével elérni. Ahogy az ábrán is látható, ezek a kapcsolatok pufferelt átvitelt tesznek lehetővé, és mind a négy AXI porton 32 vagy 64 bit széles adatokat küldhetünk. Habár a 15.18. ábra nem mutatja, ebben az esetben is használhatunk független órajeleket a kapcsolatok két oldalán.

Amennyiben az adatátvitel sebessége nem lényeges, mert például egy PS-ben található blokkot akarunk konfigurálni a PL-ből, akkor használhatjuk az *AXI General Purpose (AXI_GP)* portokat. Ehhez két darab master port, és két darab slave port áll a rendelkezésünkre a PL-ben (lásd 15.17. ábra alsó és felső szaggatott blokkját). Ebben az esetben a portok 32 bitesek, és nincs pufferelés, szemben az AXI_HP portokkal. Viszont itt is különböző lehet a csatornák két oldalán használt órajel.

A 15.19. ábrán láthatjuk összefoglalva a különböző PS-PL közötti átviteleket. Minden egyes átvitel esetén látható az adott port sávszélessége is. Ez viszonylag egyszerűen kiadódik a port bitszélességéből és a használt órajelfrekvenciából. Az is látható az AXI portok esetén, hogy a PL-ben master vagy slave portot kell illesztenünk az adott interfészhez. Például az AXI_HP esetén a PL-ben AXI master található (lásd 15.18. ábra), így ehhez egy slave egységet kell illesztenünk.

Interface	Type	Bus Width (bits)	IF Clock (MHz)	Read Bandwidth (MB/s)	Write Bandwidth (MB/s)	R+W Bandwidth (MB/s)	Number of Interfaces	Total Bandwidth (MB/s)
General Purpose AXI	PS Slave	32	150	600	600	1,200	2	2,400
General Purpose AXI	PS Master	32	150	600	600	1,200	2	2,400
High Performance (AFI) AXI_HP	PS Slave	64	150	1,200	1,200	2,400	4	9,600
AXI _ACP	PS Slave	64	150	1,200	1,200	2,400	1	2,400
DDR	External Memory	32	1,066	4,264	4,264	4,264	1	4,264
OCM	Internal Memory	64	222	1,779	1,779	3,557	1	3,557

15.19. ábra – PS-PL között interfészek sávszélessége[6]

15.2.2.6 Órajelhálózat

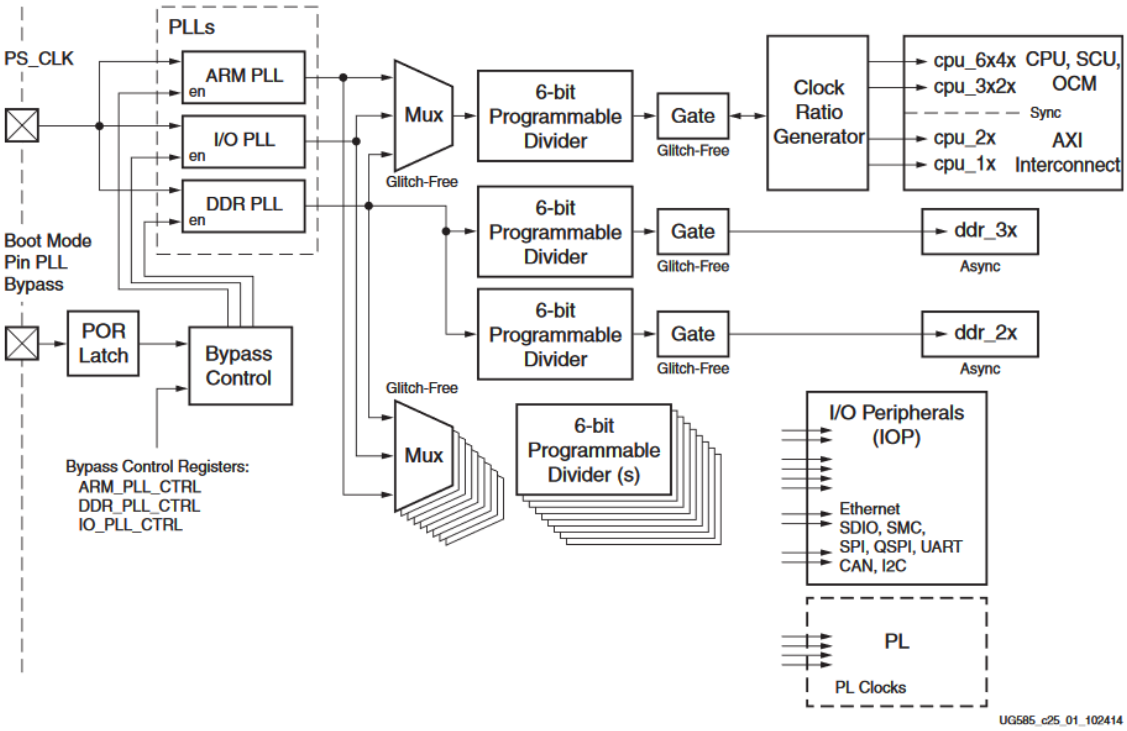
Az előző részben már említettük, hogy bizonyos blokkok között szinkronizálás szükséges az egymástól független órajelek miatt. Most a Zynq-7000 áramkörben előforduló különböző órajeleket vizsgáljuk meg.

Copyright © 2021 / BME AUT

A buszhálózatot korábban bemutató 15.17. ábrán is szerepelnek már az órajelek. Minden blokk esetén a téglalapba írt megnevezés az adott blokk órajelét adja meg (pl. DDR_2x, CPU_6x4x, PL clocks). Ezeket az órajeleket a PS állítja elő, egy külső *PS_CLK* órajel alapján.

Az előállítás a 15.20. ábrán látható. A beérkező PS_CLK jel három PLL áramkörre csatlakozik, amelyek három független órajelet állítanak elő az ARM IP-k (CPU, összekötő IP-k), az I/O-k, és a DDR vezérlő számára. A PLL áramkörök kihagyhatók, ha alacsonyabb fogyasztást szeretnénk elérni. Ezt jelöli az ábrán látható *Bypass Control* blokk, amely akár külső I/O láb, akár a program alapján is vezérelhető.

A 15.20. ábra alján látható, hogy a PLL-ek segítségével előállított órajelek közül, különböző leosztás után, négy darab a PL számára is rendelkezésre áll. Emellett a PL-ben használhatunk a 12.3.5. fejezetben bemutatott PLL modulokat is.



15.20. ábra – Órajelek a Zynq-7000 esetén[6]

I/O órajelek

A PL órajeleihez hasonlóan az I/O perifériák számára is meg kell adnunk a használni kívánt órajeleket (IOP jelölés a 15.20. ábrán). Gyakran szerepelt referencia órajel a korábban bemutatott perifériák esetén, ezeket kell előállítanunk a periféria működéséhez. Például a 15.16. ábrán az UART vezérlő esetén az *UART REF_CLK* megnevezés jelöli a referencia órajelet.

Copyright © 2021 / BME AUT

Bizonyos perifériák esetén viszont külső forrásból származik a referencia órajel (pl. USB vezérlő), vagy előfordulhat az is, hogy csak a cpu_1x órajelet használja a periféria, és nincs referencia órajel.

DDR órajelek

A 15.20. ábrán látható két DDR órajel is. Ezek az órajelek egymástól függetlenek, a *ddr_2x* az AXI_HP-hoz tartozó *Memory Interconnect* blokkhoz tartozik (lásd 15.17. ábra), míg a *ddr_3x* órajelet a tényleges DDR vezérlő használja.

CPU órajelek

Az ARM részhez a *cpu_* kezdetű órajelek tartoznak. A különböző számok az adott órajelnév végén az egymáshoz tartozó viszonyt írják le. Ezt mutatja be a 15.21. ábrán látható táblázat. Alapvetően két módot különböztet meg a rendszer, amelyet két oszlopban mutat a táblázat. A két mód között az egyik különbség az, hogy a *cpu_1x* órajel különböző frekvenciájú (133 MHz és 150 MHz). A másik különbség, hogy a cpu_3x2x és cpu_6x4x jelek viszonya különbözik a cpu_1x-hez képest. A táblázat első oszlopához tartozó módban 3-szoros és 6-szoros a frekvencia a cpu_1x-hez képest, a másik oszlopban viszont 2-szeres és 4-szeres. Emiatt bonyolultabb az első két sorban látható cpu órajelek elnevezése. A két módot, kicsit szerencsétlenül, 6:2:1 és 4:2:1 néven találjuk a Zynq dokumentációjában. Az arányok a cpu_6x4x, cpu_2x, és a cpu_1x jelek arányát adják meg.

A buszhálózatot bemutató 15.17. ábrán látható, hogy a különböző cpu órajelek között nincs async blokk, mivel ezek az órajelek azonos forrásból származnak, és a frekvenciájuk aránya egész szám.

CPU Clock	6:2:1	4:2:1	Clock Domain Modules
CPU_6x4x	800 MHz (6 times faster than CPU_1x)	600 MHz (4 times faster than CPU_1x)	CPU clock frequency, SCU, and OCM arbitration, NEON, L2 cache memory
CPU_3x2x	400 MHz (3 times faster than CPU_1x)	300 MHz (2 times faster than CPU_1x)	APU timers
CPU_2x	266 MHz (2 times faster than CPU_1x)	300 MHz (2 times faster than CPU_1x)	I/O peripherals, central interconnect, master interconnect, slave interconnect, and OCM RAM
CPU_1x	133 MHz	150 MHz	I/O peripherals AHB and APB interface busses

15.21. ábra – Az ARM órajelek viszonya[6]

15.2.2.7 Reset logika

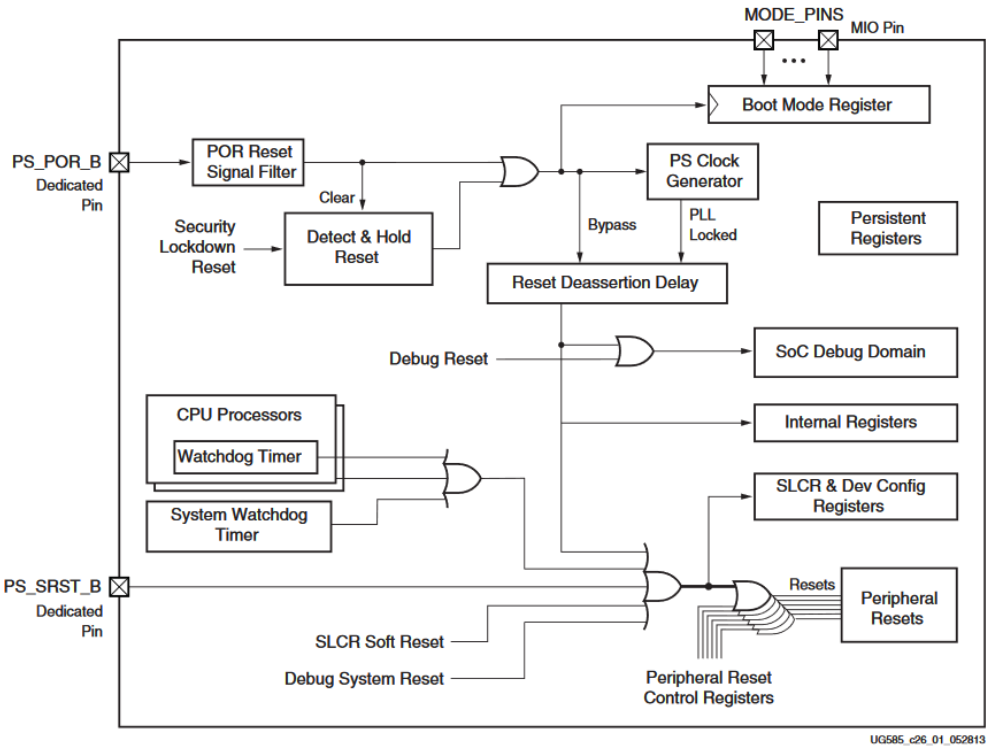
Az órajelhálózathoz hasonlóan dedikált hálózaton keresztül történik a különböző modulok alaphelyzetbe állítása (reset). A reset hálózat főbb részeit a 15.22. ábra mutatja.

Copyright © 2021 / BME AUT

A *PS_POR_B* külső láb úgynevezett *Power-On-Reset (POR)* alaphelyzetre használható. Ameddig a tápfeszültség nem éri el a működéshez szükséges szintet, ez az alaphelyzet folyamatosan fennáll. A POR reset a teljes rendszert alaphelyzetbe állítja, beleértve perifériákat, hibakereséshez használt és egyéb belső regisztereket. Folyamatos POR reset-et állít elő a Security Lockdown modul is (lásd 15.22. ábra), ha biztonsági problémát lát. Ebből az állapotból csak egy külső PS_POR_B reset hozza ki a rendszert. A biztonsági problémák okát nem részletezzük ebben a jegyzetben.

Az úgynevezett *System Reset* ezzel szemben az előbb említett hibakeresési és belső regisztereket nem törli, így például a hibakeresésnél a töréspontok megmaradnak. System Reset-nek több forrása is lehet (lásd 15.22. ábra):

- A PS_SRST_B láb (külső forrás)
- Az SLCR regiszteren keresztül a program is kezdeményezheti
- Watchdog időzítők
- Debug reset



15.22. ábra – Zynq-7000 reset hálózata[6]

A hibakereséshez és programozáshoz használt *Joint Test Action Group (JTAG)* interfészen keresztül is alaphelyzetbe állítható a rendszer. Ezt *Debug Reset*-nek nevezik, és ahogy az ábrán látható, kétféle reset-et eredményezhet. Egyrészt, ahogy az előző felsorolásban is szerepel,

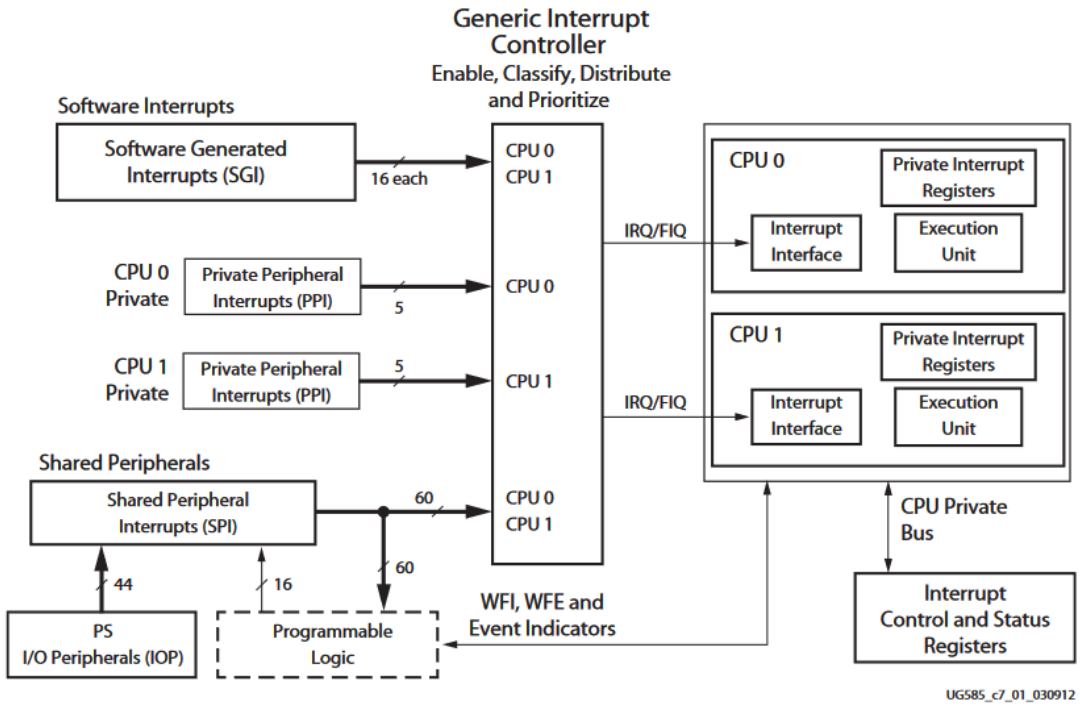
Copyright © 2021 / BME AUT

jelenthet System Reset-et (*Debug System Reset* az ábrán), másrészt alaphelyzetbe állíthatja csupán a rendszer debug részeit is (*Debug Reset* az ábrán).

15.2.2.8 Megszakítások

Az APU felépítése esetén már említettük, hogy a rendszer külön megszakításvezérlő IP-vel rendelkezik (lásd 15.2.2.1. fejezet). A *Generic Interrupt Controller (GIC)* fő feladata a különböző forrásból származó számos megszakítási eseményt kezelni (prioritásokat, maszkokat), és a legmagasabb prioritásút eljuttatni a megadott CPU maghoz. A forrásokat mutatja a 15.23. ábra. Látható, hogy maga a CPU magokon futó program is kezdeményezhet megszakítást (SGI az ábrán), és ezenkívül a különböző perifériák tudnak megszakítást kérni a CPU magoktól. A perifériákhoz tartozik a PL-ben elhelyezkedő logika is.

A 15.23. ábrán látható az is, hogy az úgynevezett *Wait For Interrupt (WFI)* és *Wait For Event (WFE)* események kapcsolatban állnak a PL-lel. A WFI és WFE állapotok lényege, hogy a WFI vagy WFE utasítások hatására a CPU mag *alvó (sleep)* módba kapcsol mindaddig, amíg egy megszakítás vagy esemény be nem érkezik a CPU-hoz. A PL egyrészt megszakítás segítségével fel tudja ébreszteni a processzort ebből az állapotból, másrészt látja a CPU mag állapotát is.



15.23. ábra – GIC megszakításvezérlő[6]

Copyright © 2021 / BME AUT

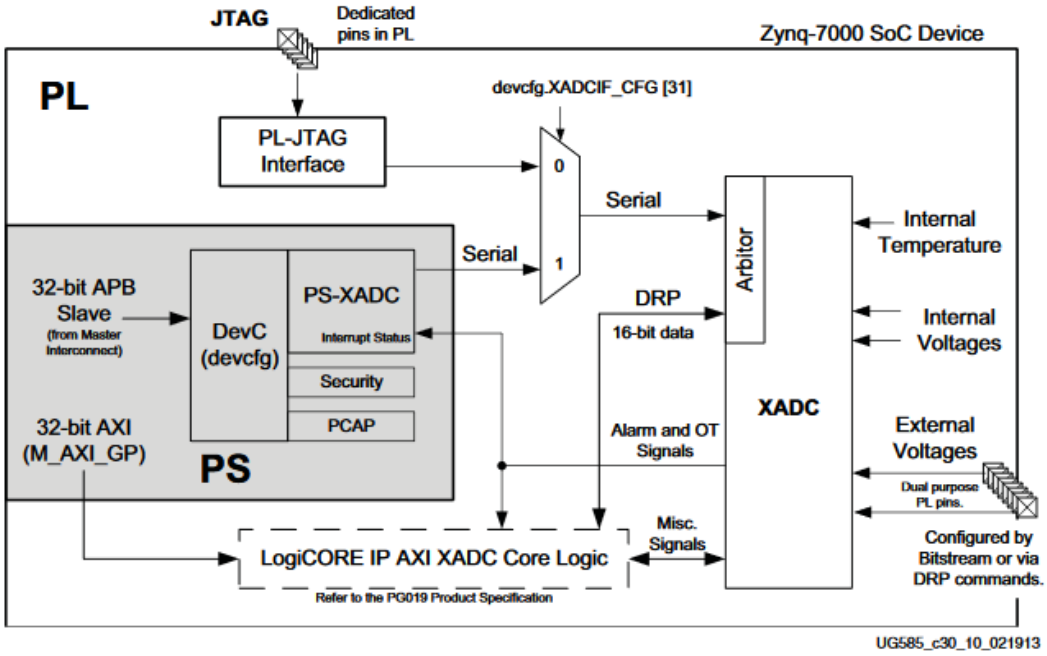
15.2.3 Zynq-7000 PL felépítése

Ahogy már többször említettük, a Zynq SoC-k legfontosabb tulajdonsága, hogy az ARM alapú processzoros részegységen kívül programozható logikát is tartalmaznak. A Zynq-7000 esetén a PL a 12.3. fejezetben bemutatott elemeket tartalmazza (7-es sorozatú FPGA). A különböző Zynq-7000 áramkörökben különböző mennyiségű programozható logikát találunk, és ennek megfelelően az áramkörök ára is széles tartományban változik.

15.2.3.1 XADC

A Zynq-hez hasonló SoC rendszerek gyakran tartalmaznak analóg-digitális átalakítókat (ADC). A Zynq-7000 esetén a PL-ben található két 12 bites ADC, amelyek 1 MSPS sebességre képesek. Az ADC-k felépítése a 15.24. ábrán látható. Az XADC többféle interfészen keresztül elérhető. Egyrészt a PL-ben használhatunk egy soft XADC vezérlő IP-t, amely az AXI_GP portra kapcsolható. Ezenkívül a PS is tartalmaz egy XADC vezérlőt, amely viszont APB porton keresztül érhető el a központi összekötőn keresztül. Ráadásul az XADC vezérelhető JTAG interfészen is.

A PL-ben található XADC segítségével az áramkör belső hőmérsékletét és különböző feszültségeit is megmérhetjük. A külső analóg jelek esetén riasztási eseményeket is tudunk generálni a mérésekhez (lásd 15.24. ábra). Az ábrán szereplő *OT* jel pedig hőmérsékleti riasztást jelent (*over temperature*).



15.24. ábra – XADC felépítése[6]

Copyright © 2021 / BME AUT

15.2.3.2 PS-PL összehasonlítás

Felmerülhet a kérdés, hogy egy adott feladat megoldásához mikor érdemes a PL, és mikor a PS erőforrásait igénybe venni. Általánosságban igaz, hogy az egyedi hardvernek a következő előnyei vannak a szoftveres megoldással szemben:

- Párhuzamosítás: A korábbi részekben már rámutattunk, hogy a műveletek párhuzamos végrehajtása milyen előnyökkel járhat. Ezt leginkább a HLS esetén vizsgáltuk meg részletesen a 12.7.1. fejezetben. Viszont fontos megjegyezni, hogy a PL esetén elérhető órajel biztos, hogy alacsonyabb lesz, mint amit a PS-ben tudunk használni.
- Fogyasztás: A különböző platformok vizsgálatánál rámutattunk, hogy a legtöbb esetben az egyedi hardver kevesebb energiát igényel a szoftveres megoldással szemben (lásd 12.6. fejezet). A Xilinx saját mérései alapján a PL-ben megvalósított logika egytől két nagyságrendig tartó energiamegtakarítást jelenthet a PS-ben megírt programhoz képest. A fogyasztással kapcsolatban érdemes megjegyezni azt is, hogy az áramkör belső erőforrásai mindig kevesebb energiát igényelnek, mint az áramkörön kívüli elemek. Például a DDR memória elérése mindig nagyobb energiát igényel, mint a belső OCM memória használata.
- Késleltetés: Szintén a 12.6. fejezetben vizsgáltuk meg, hogy a bemenet megjelenésétől a kimenet változásáig tartó késleltetés a hardveres megoldások esetén lényegesen alacsonyabb, mint egy szoftver esetén.
- Determinisztikus végrehajtási idő: Egy szoftver esetén sok esetben nehezen meghatározható az adott eljárás végrehajtási ideje. Ennek oka lehet, hogy az operációs rendszer más taszkot is beütemezhet a feladat végrehajtása közben, de akár a különböző megszakítások is befolyásolhatják a futási időt. Ezzel szemben egy hardveres megoldás végrehajtási ideje egyértelmű.

A fenti előnyökön kívül hátrányként megjelenik a PS és a PL közötti kommunikáció. Valahogyan el kell juttatni a feldolgozandó adatot a PL-hez, és a feldolgozás után vissza is kell jutnia az adatnak a PS-hez az esetek többségében. A Xilinx saját mérései szerint 4 kB feletti adatméret esetén ez a negatív hatás már nem jelentős.

A PL használatánál korlátozást jelenthet az erőforrások száma is. A párhuzamosítás mértékét mindenképp korlátozza, hogy egy adott Zynq áramkör csak adott mennyiségű erőforrással rendelkezik (pl. DSP blokkokból).

15.2.3.3 PL konfigurációja

A PL megfelelő működéséhez be kell állítani az erőforrások konfigurációját, például a különböző elemek közötti huzalozásokat, a flip-flop-ok alapállapotát, vagy a hard IP-k beállításait. Minden FPGA terv esetén létezik egy úgynevezett *bitstream* fájl, amely a rendszer konfigurációját tartalmazza. Mivel a Xilinx FPGA-k és a Zynq sem rendelkeznek megfelelő méretű nemfelejtő memóriával, így ezt a fájlt a legtöbb esetben külső tárolóban tároljuk. A külső memóriából történő betöltés történhet a Zynq indulási folyamata során (*bootolás*), vagy akár a későbbi felhasználói program is betöltheti a megfelelő bitstream fájlt a PL-be. Arra is

Copyright © 2021 / BME AUT

lehetőségünk van, hogy már egy betöltött PL konfigurációt a későbbiekben átprogramozzunk, ezáltal más és más feladathoz különböző hardvert használjunk a PL-ben.

A PS oldaláról az úgynevezett *Processor Configuration Access Port (PCAP)* interfész teszi lehetővé a PL konfigurációját. Az interfész AXI buszt használ a PS oldalán. Ezenkívül a PL és a PS független JTAG portokkal rendelkezik, amelyen keresztül szintén fel tudjuk programozni mind a PS-t, mind a PL-t.

15.2.4 Zynq-7000 indulási folyamata

Minden programozható áramkör esetén lényeges kérdés, hogy miként történik az áramkör elindulása, azaz a *bootolás* folyamata. A Zynq esetén a helyzet abból a szempontból speciális, hogy a CPU magokon kívül FPGA részt is tartalmaz, amelyet konfigurálni kell.

A különböző szintű bootloader programokra azért van szükség, mert a betöltendő szoftver (pl. operációs rendszer) nem feltétlenül tudja, hogy milyen inicializálásokat kell az adott hardveren elvégezni.

Az indulási folyamat első lépéseként (a POR reset után) az OCM-ben található BootROM program hajtódik végre. Ennek a programnak a feladata az alapvető hardverek inicializálása, és egy másik program betöltése az OCM memóriába, amely aztán folytatja a boot folyamatot. A BootROM program minden esetben az egyik CPU magon fut, a másik CPU mag eközben alvó állapotban van. A BootROM által betöltött programot *First-Stage Boot Loader*-nek (FSBL) nevezik, amely a rendszerben található külső tárolók bármelyikében elhelyezkedhet (pl. quad-SPI, NAND és NOR flash-ek).

A FSBL már vagy a felhasználói alkalmazást tölti be egy külső tárból a DDR memóriába, vagy bizonyos esetekben egy *Second-Stage Boot Loader*-nek (SSBL) nevezett következő szintű betöltő alkalmazást másol a DDR memóriába. SSBL program lehet például a Linux esetén használt *U-Boot*. Az U-Boot ezt követően már be tudja tölteni a Linux operációs rendszer kernelét.

A PL konfigurációjára vagy az FSBL futása idején van lehetőség, vagy a későbbi felhasználói program tudja felprogramozni a PL-t.

15.3 Zynq kártyák

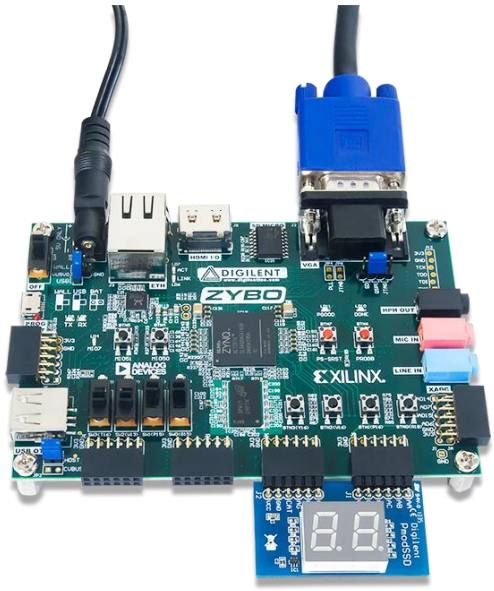
A Zynq eszközök alkalmazását megkönnyíti a különböző készen kapható modulok használata. Ezeknek a SoM kártyáknak a legnagyobb előnye, hogy egy nyomtatott áramkörön helyezik el a Zynq áramkört és a hozzátartozó legfontosabb külső egységeket is. Az elhelyezett egyéb áramkörök közé tartozhat DDR memória, periféria PHY áramkörök, periféria csatlakozók (pl. USB, Ethernet), és a tápellátást szolgáló áramkörök is. Egy SoM modul használatával nem nekünk kell a teljes nyomtatott áramkört megterveznünk, és a gyakran bonyolult tokozású integrált áramköröket beültetnünk.

Copyright © 2021 / BME AUT

Ebben a részben néhány tipikus kártyát mutatunk be, amelyek más és más feladatokhoz használhatóak.

15.3.1 Digilent Zybo

A Digilent cég által gyártott *Zybo* elnevezésű kártya egy egyszerűbb felépítésű Zynq-7000 áramkört tartalmaz. Ennek következtében az ára is – a többi Zynq alapú kártyához képest – alacsonyabb, kb. 60 ezer Ft. A Zynq áramkör mellett 1 GB DDR3L memóriát tartalmaz, és ahogy a 15.25. ábrán látható, viszonylag sok perifériát használhatunk a kártyán. Láthatunk például Ethernet, USB, VGA és HDMI csatlakozókat. Amennyiben saját eszközt szeretnénk illeszteni a Zynq-hez, a kártyán elhelyeztek egy Pmod-nak nevezett bővítőcsatlakozót (a kártya alsó részén látható), amelyen keresztül elérjük a Zynq I/O lábait.

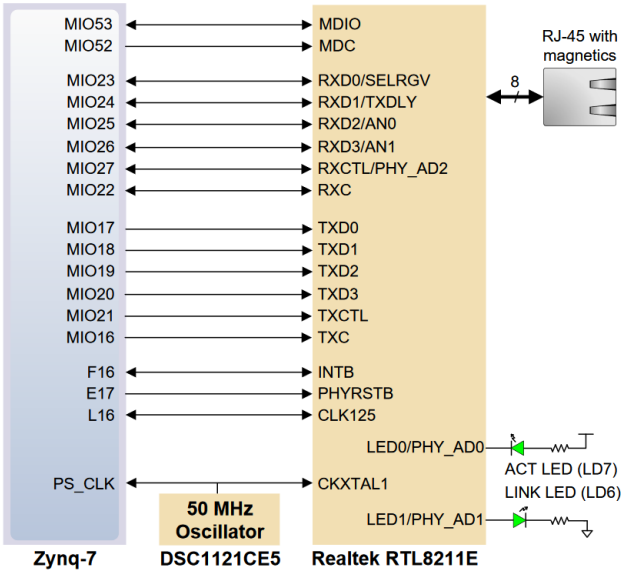


15.25. ábra – Digilent Zybo fejlesztőkártya[7]

Az Ethernet periféria esetén már említettük a 15.2.2.4. fejezetben, hogy külső fizikai réteget megvalósító áramkört kell illeszteni a Zynq-hez. Ehhez kapcsolódóan mutatja a 15.26. ábra, hogy a Zybo esetén hogyan történt a PHY áramkör bekötése a Zynq-hez. Látható, hogy a MIO portokon keresztül csatlakozik a Realtek cég által gyártott PHY, és a PHY-hez csatlakozik a tényleges Ethernet csatlakozó, valamint két jelző LED is.

[7] Forrás: https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual (2020. 06. 01.)

Copyright © 2021 / BME AUT

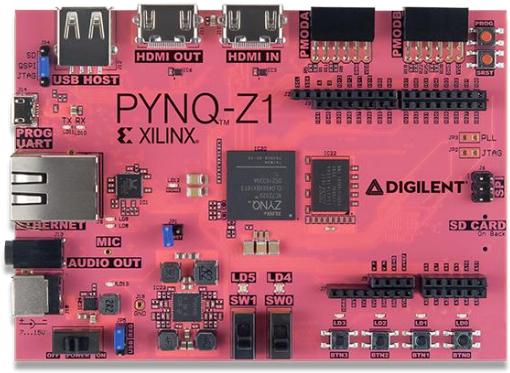


15.26. ábra – Ethernet interfész a Zybo kártya esetén[7]

15.3.2 Digilent PYNQ

Az előző, Zybo kártyához nagyon hasonlóak a *PYNQ-Z1 és PYNQ-Z2* elnevezésű kártyák. Ezeket is a Digilent cég gyártja, és az áruk is megegyezik a Zybo-val. A legfontosabb különbség, hogy a kártyákhoz tartozó szoftveres támogatottság lehetővé teszi, hogy a népszerű Python programozási nyelv segítségével használjuk őket. Ehhez egy teljes Python alapú keretrendszer áll a rendelkezésünkre, így akkor is ki tudjuk kihasználni a Zynq által nyújtott hardveres lehetőségeket, ha nem rendelkezünk hardvertervezői ismeretekkel.

A PYNQ kártyák perifériái hasonlóak a Zybo-hoz, viszont a PYNQ-Z1 rendelkezik Arduino, a PYNQ-Z2 pedig Raspberry Pi bővítőcsatlakozóval.



15.27. ábra – Digilent PYNQ-Z1 fejlesztőkártya[8]

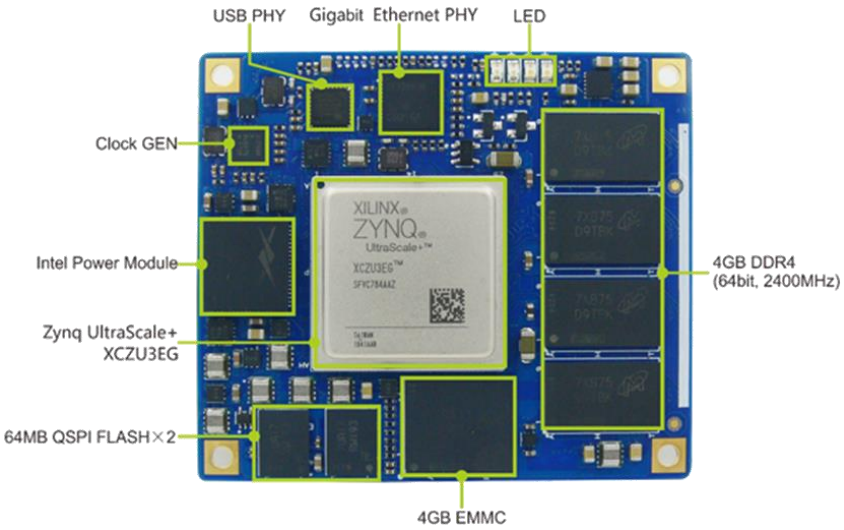
[8] Forrás: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start (2020. 06. 01.)

Copyright © 2021 / BME AUT

15.3.3 MYC-CZU3EG SoM kártya

Az előző két Digilent kártya elsősorban egy termék fejlesztési fázisában, esetleg tanulási (pl. egyetemi oktatás) célokra használhatók. Beágyazott rendszerekhez ezek a kártyák méretükből adódóan a legtöbb esetben nem alkalmasak.

Ezzel szemben a MYIR cég által fejlesztett különböző SoM kártyák már késztermékekhez is használhatók. A 15.28. ábrán látható egy konkrét típus, a MYC-CZU3EG, amely egy kisebb teljesítményű Zynq UltraScale+ SoC-t tartalmaz. Ennek a modulnak az ára már 120 ezer Ft-ot is eléri. A perifériák tekintetében a modul rendelkezik 4 GB DDR4 memóriával, 4 GB eMMC flash memóriával, Ethernet és USB PHY áramkörrel. A kártya hátsó oldalán pedig egy sokpólusú csatlakozót találunk, amely segítségével egyedi nyomtatott áramkört illeszthetünk a SoM-hoz.



15.28. ábra – MYC-CZU3EG SoM[9]

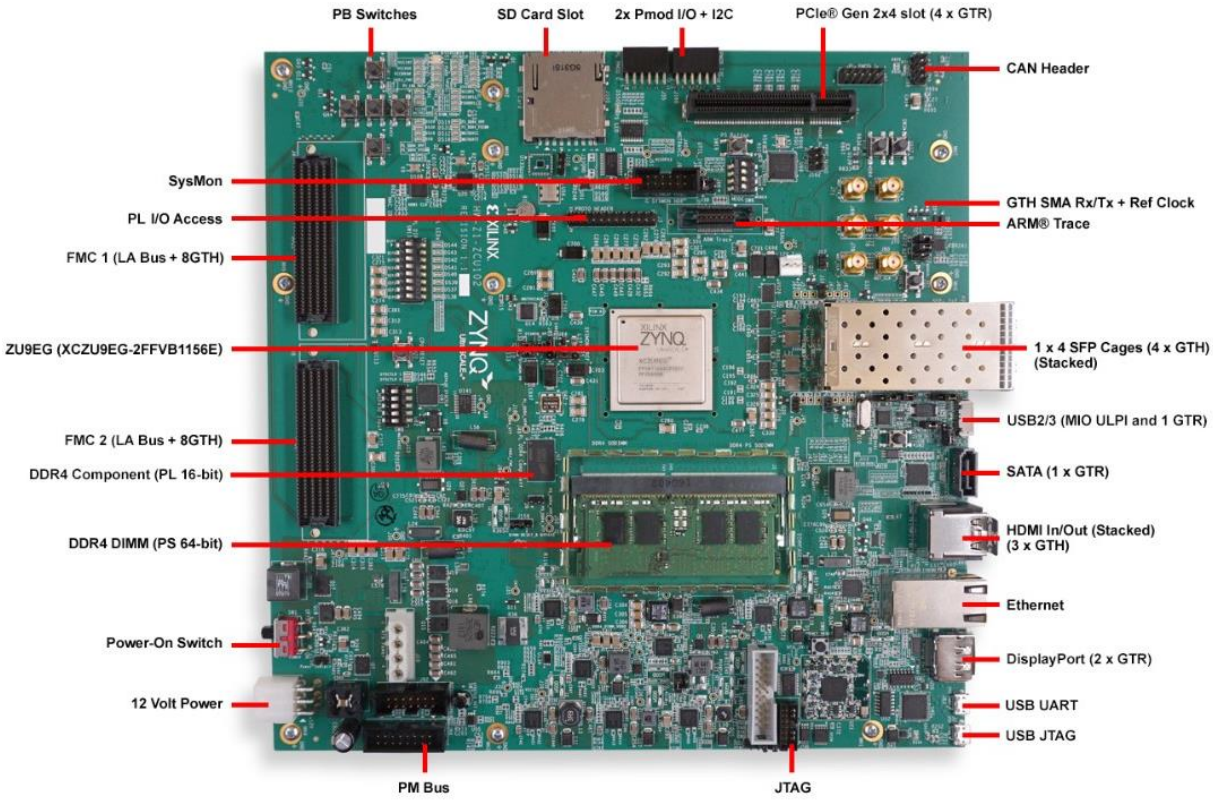
15.3.4 Xilinx ZCU102 fejlesztőkártya

A Xilinx saját maga is készít kártyákat, hogy bemutassa különböző áramköreinek képességét. A kártyák méretükből adódóan inkább prototípusfejlesztéshez alkalmasak, mint késztermékhez.

A 15.29. ábrán látható egy nagyobb teljesítményű Zynq UltraScale+ alapú fejlesztőkártya. A nagyobb teljesíményű Zynq miatt a kártya ára meghaladja az 1 millió Ft-ot is. A kártyán számtalan csatlakozó található, például PCI Express csatlakozó, DDR4 foglalat, és többféle Ethernet interfész. A kártyán találhatók úgynevezett *FPGA Mezzanine Card (FMC)* csatlakozók is, amelyek segítségével további perifériákkal lehet bővíteni a rendszert.

[9] Forrás: https://hu.mouser.com/new/myir-tech/myir-myc-czu3eg-som/ (2020. 06. 01.)

Copyright © 2021 / BME AUT



15.29. ábra – Xilinx ZCU102 fejlesztőkártya[10]

[10] Forrás: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html#hardware (2020. 06. 01.)

Copyright © 2021 / BME AUT

Felhasznált és ajánlott irodalom

[1] M. Paolo, Scalable System-on-Chip Design, Columbia University, 2017.

[2] S. Pasricha és N. Dutt, On-Chip Communication Architectures (System on Chip Interconnect), Morgan Kaufmann, 2008.

[3] Louise Crockett et al. The Zynq Book.
https://www.zynqbook.com/

[4] Wikipedia. FPGA.
https://hu.wikipedia.org/wiki/Field-programmable_gate_array

[5] Wikipedia. System-on-Chip.
https://en.wikipedia.org/wiki/System_on_a_chip

[6] Wikipedia. LEON.
https://en.wikipedia.org/wiki/LEON

[7] Wikipedia. LIDAR.
https://hu.wikipedia.org/wiki/LIDAR

[8] Wikipedia. Register-transfer level
https://en.wikipedia.org/wiki/Register-transfer_level (Hozzáférés: 2020. 06. 01.)

[9] ARM Technologies. AMBA.
https://developer.arm.com/architectures/system-architectures/amba

[10] ARM Technologies. DesignStart FPGA.
https://www.arm.com/resources/designstart/designstart-fpga (Hozzáférés: 2020. 06. 01.)

[11] Xilinx. MicroBlaze Processor Reference Guide.
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug984-vivado-microblaze-ref.pdf

[12] Xilinx. Zynq-7000 SoC Technical Reference Manual.
https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[13] Xilinx. Intro to Versal ACAPs.
https://www.xilinx.com/support/documentation/white_papers/wp519-versal-premium-intro.pdf

[14] Xilinx. AI Engine White Paper.
https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf

[15] Xilinx. LIDAR Pulse Detection Accelerator Model Based Design
https://developer.xilinx.com/en/articles/lidar-pulse-detection-accelerator-model-based-design-targeting-vitis-and-rfsoc.html

[16] Intel. Nios II Processor Reference Guide.
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf

Copyright © 2021 / BME AUT