Digitális technika 2
2012/2013

Készítette: Dudás Márton

Tartalomjegyzék

Valódi flip-flopok:

Tekintsük a következő, állapottáblájával adott hálózatot:

y	X_1X_2	00	01	11	10
A		C, 0	C, 0	A, 0	A, 0
B		B, 0	C, 0	B, 0	B, 0
C		A, 1	B, 1	C, 1	C, 1

Szinkron sorrendi hálózat, mely Moore modell szerint működik.
Kódoljuk az állapotokat hasraütésre és írjuk fel a kódolt állapottáblát.

A=00
B=01
C=11

A kódolt állapottáblának azonban 4 sora lesz:

y	X_1X_2	00	01	11	10
00		11, 0	11, 0	00, 0	00, 0
01		01, 0	11, 0	01, 0	01, 0
11		00, 1	01, 1	11, 1	11, 1
10		--, -	--, -	--, -	--, -

Egyszerűség érdekében használjunk D flip-flopot (ekkor a vezérlési tábla egy az egyben a kódolt állapottábla) és keressük meg a vezérlési függvényeket:

$Z=y_2$
$D_1= y_1*X_1 + /y_2*/X_1 + /y_1*/X_1*X_2$
$D_2=/y_1*/X_1 + y_2*X_2 + y_2*X_1$

A vezérlési tábla még tartalmaz dont care értékeket, de a megvalósított hálózatban ezeket az értékeket már rögzítjük. Ha a flip-flopok véletlenül 10 értéket vennének fel induláskor, akkor a hálózat olyan állapotba kerülne, ami nem specifikált működést okozhatna.

Megoldási javaslat:

1. Önbeálló hálózat:
A közömbös bejegyzéseket megfelelő módon rögzítve biztosíthatjuk, hogy a hálózat egy bizonyos állapotba kerüljön bekapcsolás után. Ez azonban elég kevés hálózatnál működik, vagy az állapotszám növekedéséhez vezet.

2. Valódi flip-flopok használata:
Valódi flip-flopokon a már megismert ki és bemenetek mellett van két aszinkron bemenet, a clear és a preset, melyek általában negált jellel működnek (azaz logikai 0 esetén aktívak), a clear 0, míg a preset 1 állapotba állítja a flip-flopot. Valódi flip-flopon a kimenet negáltja is rendelkezésünkre áll általában.
Amennyiben CL és PR vezérlőjel egyszerre éri a flip-flopot, a valós működés a gyártótól függ.
Lehetséges, hogy az egyik nagyobb prioritású mint a másik, és lehetséges, hogy ilyen esetben a kimenet és a negáltja is azonos értéket vesz fel.

A flip-flop nem azonnal reagál az órajelre, hanem technológiától függően valamilyen késleltetéssel.
Ezt a késleltetést propagation delay-nek nevezzük. A valóságban nagyjából 1-30 ns körüli érték.

Definiálnunk kell a nem szomszédos bemeneti változást.
Digitális technika 1-ből már definiáltunk setup és hold időket. Ezek azt mutatják meg, hogy az órajel megérkezése előtt illetve után mennyivel kell a bemenetnek stabilnak lennie. Gyakorlatban a hold idő elhanyagolhatóan alacsony, a setup idő jelentősebb szereppel bír.

MSI áramkörök

MSI áramkörök:

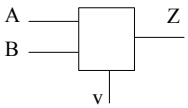
Egyszerű elemi áramkörök, amelyeket Digitális technika 1-ből már remekül ismerünk.
Ilyenek például a logikai kapuk vagy a flip-flopok.

Ezek szerepe, hogy a gyakran előforduló feladatokra kész építőelemeket használhassunk.
A technikai fejlődéssel egyre bonyolultabb kész elemeket gyárthatunk. Manapság olyan kész elemeket gyártanak, amelyek működése előre nem definiált, programozással határozhatjuk meg, hogy mit csináljanak.

Multiplexerek:

A multiplexerek a bemenet kiválasztására szolgáló áramkörök, amelyek segítségével akár kombinációs hálózatokat is megvalósíthatunk.

MP:

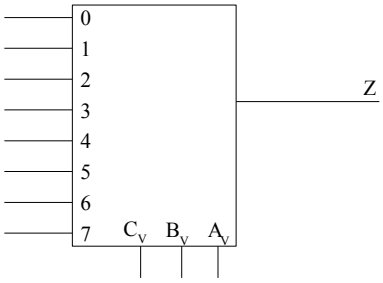


Tekintsünk egy egyszerű kombinációs hálózatot, amelynek 3 bemenete és 1 kimenete van, és a működése a következő: Az A és B bemenetei közül a harmadik, vezérlő bemenettel választhatjuk ki, hogy mely bemenet értéke jusson ki a kimenetre.

A kimenet függvénye a következő (A, B a normális bemenetek, v a vezérlő bemenet, Z a kimenet):
$Z= A*/v + B*v$

Amennyiben a vezérlő bemenet értéke 1, akkor az A*/v értéke 0, tehát a B határozza meg a kimenetet.
Amennyiben a vezérlő bemenet értéke 0, akkor a B*v értéke 0, tehát az A határozza meg a kimenetet.

Multiplexerek általánosítása:
Egy multiplexer 2^n normális és n darab vezérlő bemenettel rendelkezik (hiszen n bemeneten 2^n darab érték állítható elő).
Multiplexernek több kimenete is lehetséges. Ezt a következő módon tüntethetjük fel: választható bemenet / kimenet.
A fentebbi példa egy 2/1-es multiplexer működését írja le.
A multiplexerek vezérlő bemeneteit ABC betűivel jelöljük, ahol **az A változó az LSB!**



C változó az MSB!

Multiplexerek segítségével tetszőleges kombinációs hálózatot megvalósíthatunk, de hazárdmentesíteni nem tudunk!

MP:
Tekintsük a következő példát, melyet Karnaugh-táblájával adtunk meg:
Valósítsuk meg egy 8/1-es multiplexer segítségével.

A \ C,B	00	01	11	10
0	1	1	0	0
1	0	1	1	0
2	0	0	1	1
3	1	0	0	1

D

Szedjük két részre a Karnaugh-táblát D szerint:

D=0

C \ A,B	00	01	11	10
0	1	0	0	1
1	0	0	1	1

D=1

C \ A,B	00	01	11	10
0	1	1	0	0
1	0	1	1	0

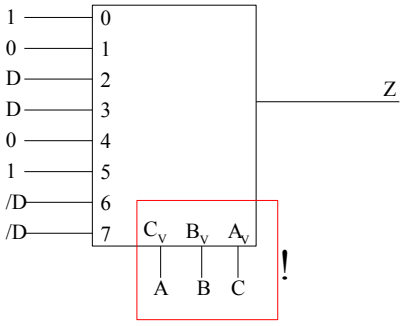
Minden cellánál nézzük meg, hogy az eredeti függvényünk mitől függ (3 lehetőség van: 0, 1 vagy D-től függő érték).
Ennek megfelelően az alábbi vezérlést kapjuk a MUX-ra:

Vezérlés:

C \ A,B	00	01	11	10
0	1	D	0	/D
1	0	D	1	/D

ABC bemenetet a MUX ABC vezérlő bemeneteire kell kötnünk, DE figyeljünk arra, hogy az eredeti ABC bemeneteknél az A az MSB, míg a MUX-on A az LSB, **így a sorrendet fel kell cserélnünk!**

Tehát a multiplexer bemeneteire 1, 0, D, D, 0, 1, /D, /D értékeket kell kötnünk.
Egy 8/1-es multiplexerrel megvalósítottunk egy 4 bemenetű kombinációs hálózatot.

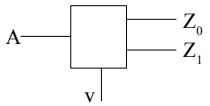


C változó az MSB!

Demultiplexer:

A demultiplexer a kimenet kiválasztására szolgáló áramkör.
Egy normális bemenetük és n darab vezérlő bemenetük van, ami segítségével a 2^n darab kimenet közül választhatjuk ki az aktívat.

MP:



Legyen 1 bemenetünk, és Z_1, Z_2 kimenetünk, a hálózat működése pedig legyen a következő:

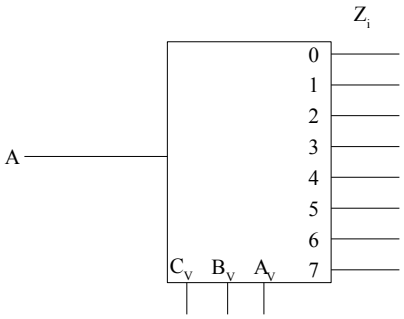
Ha v=0, akkor Z_0=A
Ha v=1, akkor Z_1=A

Z_0-et v=1 esetén nem definiáltuk, ahogy Z_1-t sem v=0 esetén.
Ezekhez az esetekhez rendeljünk 0-t.

Az igazságtábla:

A	v	Z_0	Z_1
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

Demultiplexerek esetén is léteznek nagyobb kimenetszámú áramkörök.
Ebben az esetben is n darab vezérlő bemenettel 2^n darab kimenetből választhatunk.
Demultiplexerek esetén is használhatjuk a multiplexerek esetén bevett jelölést:
Az alábbi példa egy 1/8-as demultiplexer.

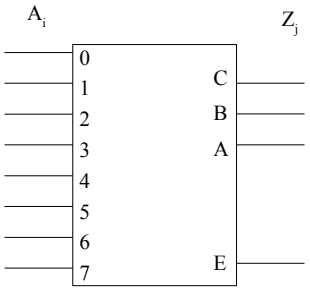


C változó az MSB!

Enkóder:

Az enkoder egy olyan hálózat, ami a kimenetein előállítja a sorszámát az egyetlen aktív bemenetének.
Az n darab kimeneten olyan bináris kód jelenik meg, ami megmutatja, hogy a 2^n bemenet közül melyiken érkezik 1-es.

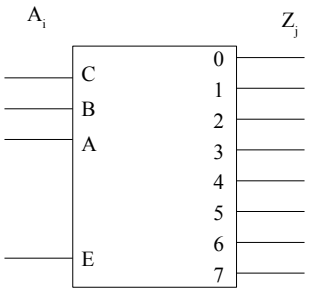
- Ha több bemeneten van 1-es, akkor definiálhatunk prioritást (általában a legmagasabb helyiérték a legmagasabb prioritású).
- Ha nincs 1-es, definiálhatunk egy E kimenetet, mely azt mondja meg, hogy a kimenet feldolgozandó-e.



C változó az MSB!

Dekóder:

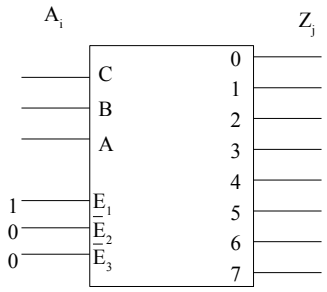
A dekodernek n darab bemenete és 2^n kimenete van.
A bemenetein előálló bináris számnak megfelelő sorszámú kimenetén 1-es értéket ad.
Itt is lehet pluszban engedélyező bemenet.



C változó az MSB!

Dekoder/demultiplexer áramkör:

Van 3 darab engedélyező bemenete, amelyen ha az 100 kombináció jelenik meg, akkor az a kimenete 0 (tehát a kimenetek negált logikával működnek), amelynek sorszáma megjelenik a CBA bemeneteken.

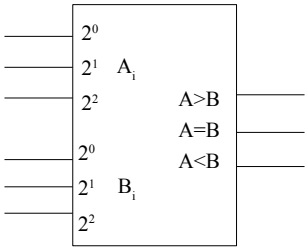


C változó az MSB!

Memóriaillesztésnél sokat fogjuk használni ezt a fajta áramkört.

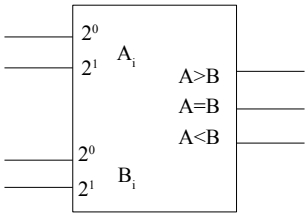
Komparátorok:

Bináris számok összehasonlítására szolgáló hálózatok, melyek 3 darab kimenettel, és az összehasonlítandó számok méretétől függő számú bemenettel rendelkeznek.



MP:
2 bites komparátor:
2 bites bináris számok összehasonlítására szolgáló hálózat, mely 2*2 bemenettel, és 3 darab kimenettel rendelkezik.

A 2*2 bemeneten 2 darab bináris számot fogad helyiérték szerint.



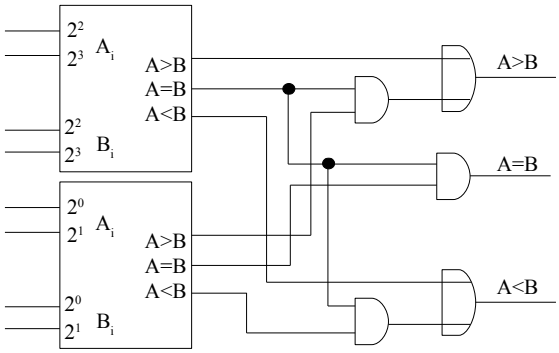
Nézzük meg a példa kedvéért az = kimenet függvényét:

$$A=B : F_{A=B} = \overline{A_1 \oplus B_1} * \overline{A_0 \oplus B_0}$$

A másik két kimenet is könnyen megkonstruálható.

Léteznek/megvalósíthatók nagyobb komparátorok is, csak a bemenetszámok és a logikai függvények bonyolódnak. Nagyobb komparátorok megvalósíthatóak kisebb komparátorok felhasználásával.

MP:
4 bites komparátor megvalósítása:
2 darab 2 bites komparátort használhatunk, amennyiben tervezünk hozzá egy megfelelő kombinációs hálózatot.
A=B: ha a két komparátor = kimenetein egyenlő értékek vannak.
A>B: ha a magasabb helyiértéknél A>B, vagy ha a magasabb helyiértékeknél A=B és a kisebb helyiértéknél A>B.
A<B: ha a magasabb helyiértéknél B>A, vagy ha a magasabb helyiértékeknél A=B és a kisebb helyiértéknél B>A.

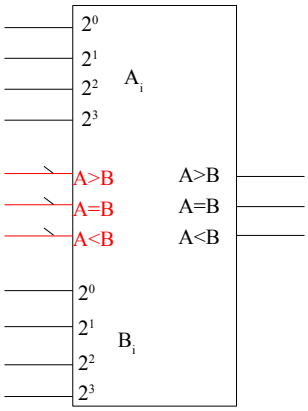


Figyeljünk a helyiértékekre és a ponált illetve negált be és kimenetekre!

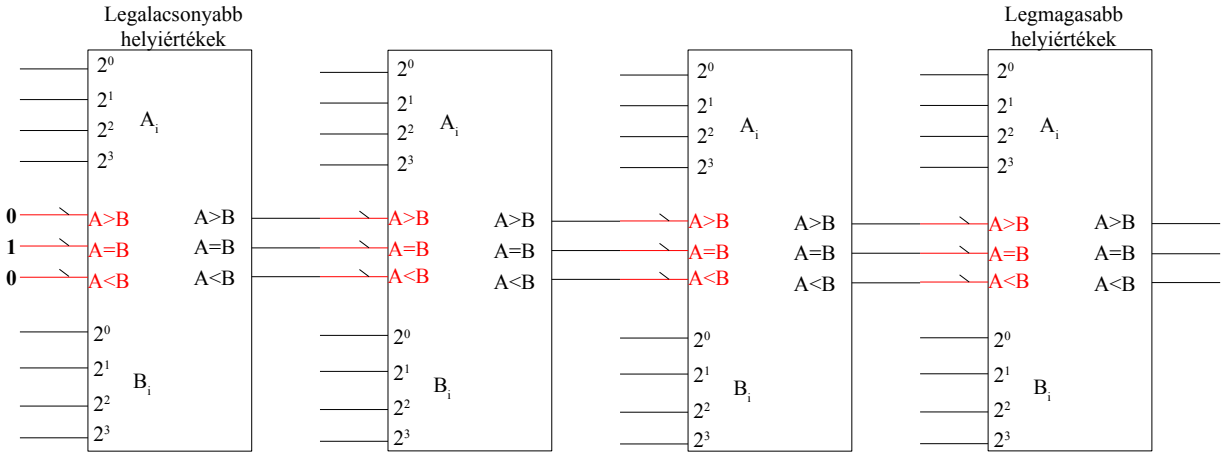
Kaszkádosított komparátorok:

Kaszkádosításnak nevezzük azt a folyamatot, amikor erre felkészített építőelemeket úgy kötünk össze, hogy nagyobb, azonos működési elvű egységet valósítsunk meg.

Kaszkádosított 4 bites komparátor:
Előző komparátortól fogadja az alacsonyabb bitek összehasonlításának eredményét.
Amennyiben nem n/1 kódot kap (a kaszkád bemenetek közül nem pontosan 1 darab aktív), különböző specifikációknak megfelelően működik.
A kaszkád bemenetek ezen az ábrán pirossal vannak feltüntetve.



Ilyen kaszkádosított komparátorokat sorba köthetünk, de az első alacsonyabb biteket fogadó bemenetére kössük oda az A=B jelet (az első komparátor a legalacsonyabb helyi értéket vizsgálja)!
Gyakran a kaszkádbemenetekre kapcsolt 010 kombináció nélkül is működnek a hálózatok, de a biztos megoldás, ha ezeket a bemeneteket mi biztosítjuk.



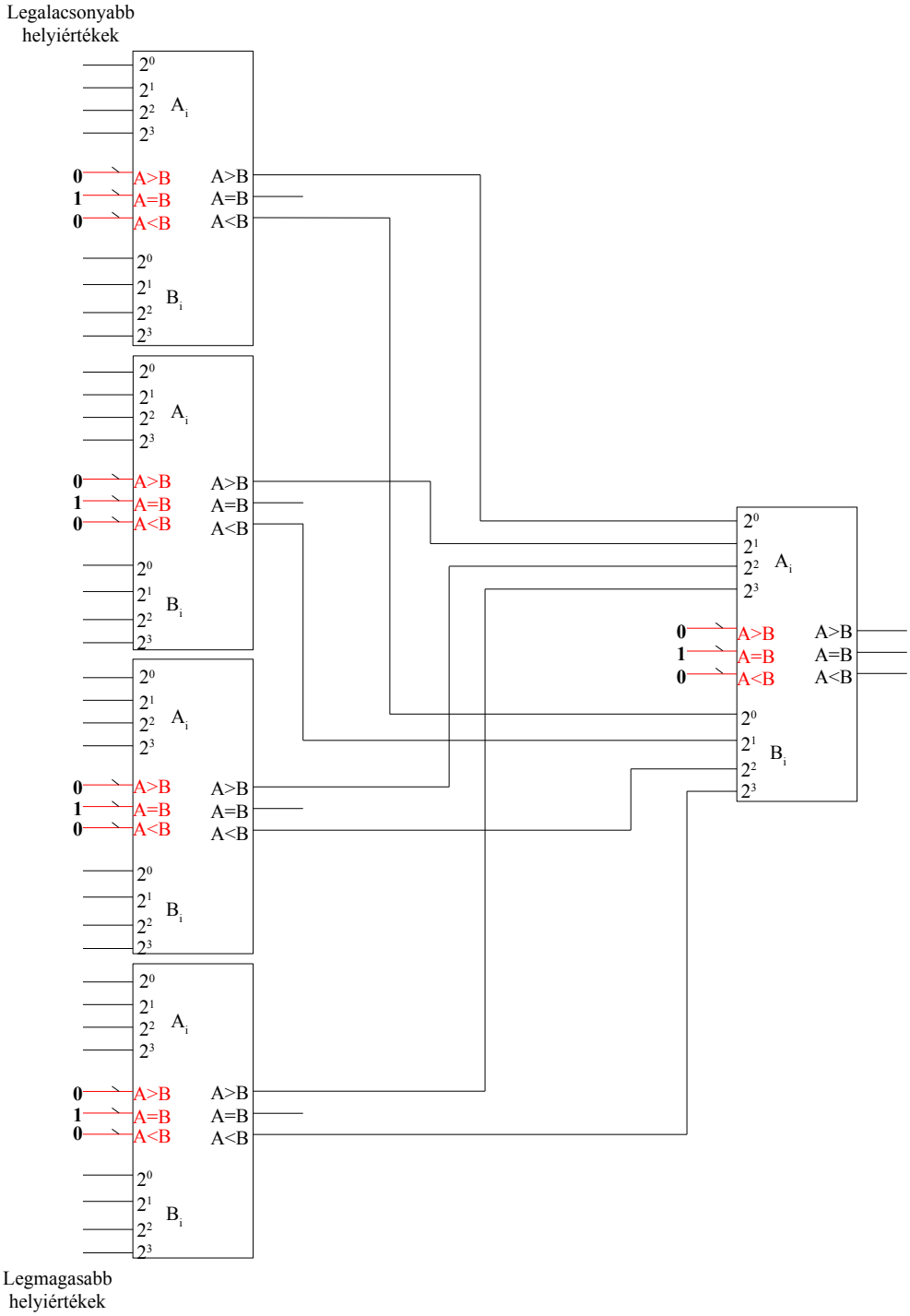
A komparátor működési ideje nem 0. Sorba kapcsolt esetben a döntési idő n-szeresére nő!

Jegyezzük meg, hogy a magasabb helyiértéknek szüksége van információra az alacsonyabb helyiértéken történt döntésről, így az a legalacsonyabb helyiérték, amelynek mi magunk állítjuk be a 010 értékekkel a kaszkád bemeneteit.

Csökkenthetjük a késleltetést a következő módon:

Csináljunk 2 szintű hálózatot az előzőleg 4 szintű hálózatból úgy, hogy párhuzamosan használjuk a 4 komparátort az első szinten, majd a második szinten ugyan ebből a komparátorból használunk még egyet.

Az A>B értékeket helyiérték szerint kössük a komparátor A bemeneteire, az A<B értékeket kössük a komparátor B bemeneteire helyiérték szerint (legnagyobb helyiértéket számláló komparátor a legnagyobb helyiértékre).
Ezzel a késleltetést n*t helyett 2*t-re csökkentettük egy darab plusz komparátor felhasználásával.



Ebből a hálózatból 20bites komparátort is csinálhatunk, ha még egy komparátor kimenetét a második szinten lévő komparátor kaszkád bemeneteire kötjük. Ekkor a kaszkádbemenetekre kötött számláló lesz a legalacsonyabb helyiérték.

Kapuk kimenete:

A valóságban 3 -féle kapukimenet létezik és ezek használata eltérő.
Nézzük a számunkra jelenleg érdekes tulajdonságok közt a fő különbséget:

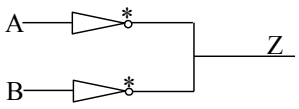
Totem-pole kimenet mindig közel rövidzár, ezért TP kimenetek nem köthetőek össze!
Nyitott kollektoros (OC) kimenetek feltétel nélkül összeköthetőek, ekkor huzalozott függvénykapcsolat jön létre.
Az OC kimenetet egy * fogja jelölni a kimeneteken.
Háromállapotú (Three-state) kimenetek csak akkor köthetőek össze, ha az összekötött kimenetek közül pontosan egy darab aktív, az összes többi úgynevezett magas impedanciás állapotban van. Az ilyen kimenetet tartalmazó áramkörökön engedélyező bemenet is van, amellyel biztosítani lehet több összekötése esetén a pontosan egy aktív kimenetet.

Huzalozott (wired) függvény kialakítás:
Ilyen függvényeket OC kimenetű elemek kimenetének összekötésével alkothatunk.

MP:
Kössük össze 2 darab OC kimenetű inverter kimeneteit.
Így egy 2 bemenetű NOR kaput valósítottunk meg huzalozott függvények kialakításával.

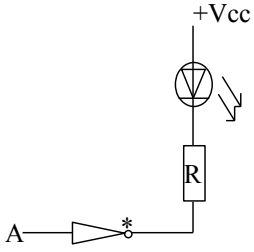
Ha bármely kimenet 1-es értéket kap, akkor a kimenete 0 értéket vesz fel, ekkor a kimenetet az inverter föld potenciálra húzza, és így a kimenet 0 értéket mutat.
Ha mindkét inverter bemenete 0, akkor a kimeneteik 1 értéket vesznek fel, tehát az OC kimenetnek megfelelően egy felhúzó ellenállás 1 szinten tartja a kimenetet.
Ez a működés pontosan megfelel a NOR kapu működésének.

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0



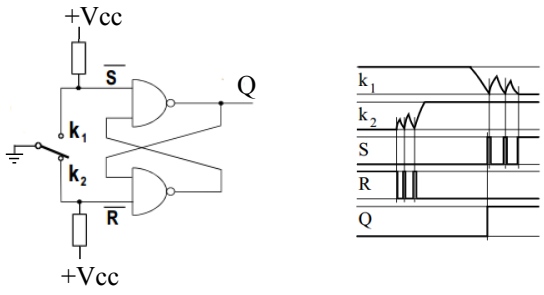
LED meghajtás OC kimenetű inverterrel:

LED-eket is működtethetünk OC kimenetű inverter segítségével.
Amennyiben az inverter bemenete 1 értékű, az inverter kimenete föld potenciálon van. Mivel az OC kimenetet felhúzó ellenállás húzza 1-es értékre, így ha az ellenállást megfelelően méretezzük, és a LED-et megfelelően kötjük, az inverter bemenete pont úgy működteti, ahogy azt szeretnénk.

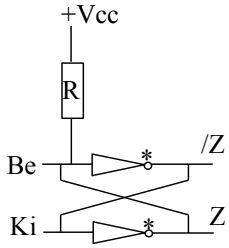


Pergésmentesítés:

A pergésmentesítést kevés gondolkodással SR flip-floppal oldhatjuk meg.



Pergésmentesítést sokkal egyszerűbben is megoldhatjuk, tekintsük a következő hálózatot:

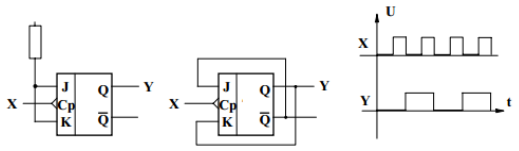


Be állásnál 1 van a felső inverteren, tehát a kimenetén 0 érték jelenik meg, ez van visszacsatolva, így az alsó inverteren 0 a bemenet, tehát a kimenete 1 (és a felhúzó ellenállás és a tápfeszültség egyébként is 1 értéken tartja a hálózatot).
Ki állásban az alsó inverter 0 kimenetet ad, így földre kötve Z kimenetet.
A két állás között mindkét inverteren 0 van, így az alsó inverter kimenete 1, ezzel tartva az értéket.

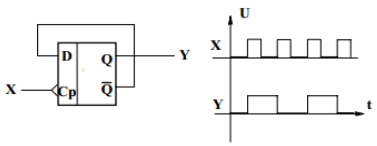
Frekvenciaosztás:

Frekvenciaosztót tudunk készíteni JK flip-flop felhasználásval.
Nézzük meg, hogy reagál a JK flip-flop, ha mindkét bemenetére 1 értéket kötünk.
Specifikáció szerint ekkor a flip-flopnak állapotot kell váltania. Ezt mindig akkor teheti meg, amikor működésétől függően felfutó/lefutó élt kap. Mivel a másik élre nem válthat állapotot, ezért a kimenetén pont az órajel frekvenciájának megfelelő órajel áll elő.

Ugyan ezt a működést érhetjük el akkor is, ha a J bemenetre /Q, a K bemenetre pedig a Q kimenetet csatoljuk vissza.



Felfutóél vezérelt D flip-flop /Q kimenetének visszacsatolásával is megvalósíthatjuk a fentebbi feladatot.



Ez a hálózat is teljesen megegyező elven működik, mint a fentebb mutatott két hálózat.

Komparátorok kettes komplemensben adott számokhoz:

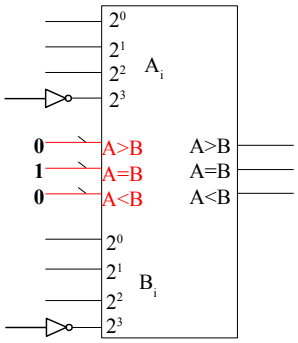
Vizsgáljuk meg, hogyan állnak elő a kettes komplemens beli számok.
A pozitív-pozitív és a negatív-negatív számok összehasonlítása hibátlanul működik.

MP: -1 4 biten ábrázolva kettes komplemensben: 1111
-3 4 biten ábrázolva kettes komplemensben: 1011

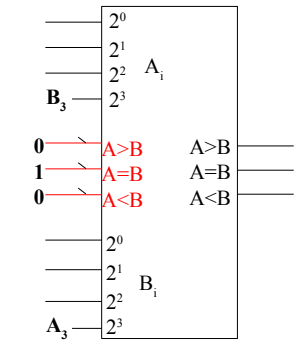
A -1 valóban nagyobb mint a -3, ahogy az 1111 is nagyobb mint az 1011.
Probléma csak akkor van, ha pozitív és negatív számot hasonlítunk össze.

Probléma akkor lép fel, ha negatív-pozitív számokat akarunk összehasonlítani, ugyanis a pozitív számok első, legmagasabb helyiértékű bitje 0, míg a negatív számok legmagasabb helyiértékű bitje 1, tehát a komparátor ebben az esetben rossz eredményt adna.

Megvalósíthatnánk a komparátort úgy, hogy egy XOR kapuba vezetjük a legmagasabb helyiértékeknek megfelelő biteket. Ennek az értéke 0, ha mind a két szám pozitív, vagy negatív. Ha az így előálló értéket és a kimeneteken előálló értéket AND kapukba vezetjük, akkor csak pontos értéket kaphatunk.
Nézzünk azonban egy sokkal egyszerűbb és kézenfekvőbb megoldást:
Jól látható, hogy a negatív számok nagyobbak mint a pozitívak, mivel a legmagasabb helyiérték a negatív számoknál 1.
Kézenfekvő a megoldás, hogy invertáljuk a legmagasabb helyiértéket. Így egy kettes komplemens esetén tökéletesen működő komparátort kapunk.



Nem annyira triviális, de a harmadik, és egyben végleges megoldás még a másodiknál is egyszerűbb, és jól működő megoldást kaphatunk, ha az első biteket egyszerűen fordított sorrendben kötjük be. Tehát az A szám legmagasabb helyiértékét a legmagasabb helyiértékű B bemenetre kötjük és fordítva.

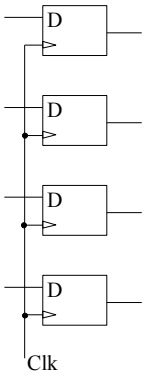


Ha mindkét szám negatív volt, vagy mindkét szám pozitív volt, akkor nem történik változás.
Azonban ha az egyik szám pozitív, a másik negatív volt, akkor így pontosan az inverterek funkcióját valósítottuk meg a lehető legegyszerűbb módon.

Valós komparátorok:
A már megszokott lábak mellett rendelkeznek egy Vcc és egy GND lábbal is.
Csak 2 kimenete van (A=B és A>B), és ezek negált logika szerinti kimenetek.
Az A<B kódot úgy állíthatjuk elő, hogy a kimeneteket invertáljuk, és egy AND kapuba vezetjük.

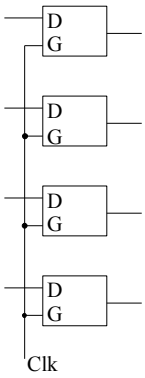
Regiszter:

Regisztert használunk arra, hogy egy n bites bináris számot eltároljunk ideiglenesen.
Egy n bites regisztert n darab D flip-flop valósít meg, melyek közös órajelet kapnak, így a bemenetet az órajel idejéig (vagy kikapcsolásig) meg tudjuk jegyezni.



Az órajel ebben az esetben gyakran nem a megszokott órajel lesz, hanem mi magunk állítjuk majd elő úgy, hogy a regiszter addig őrizze az értékeket, amíg számunkra az szükséges, és csak akkor olvasson be új értékeket, amikor mi azt szeretnénk.

Latch:
DG flip-floppal is megvalósíthatunk regiszterhez hasonló hálózatot úgy, hogy a G bemeneteket egyszerre vezéreljük. De ezt a fajta megoldást Latch-nek nevezzük.

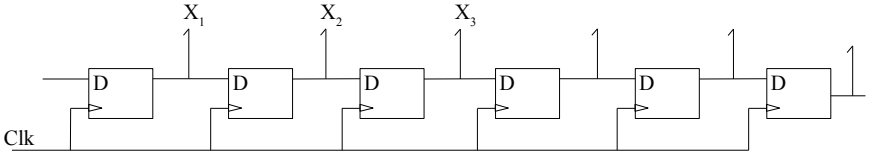


A regiszter és a latch közt a következő különbségeket tapasztalhatjuk:

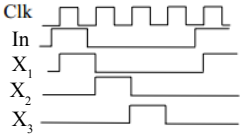
- A regiszter felfutóélre mintát vesz és a bemenetét egyből a kimenetre is másolja.
- Latch esetén amíg G=1, addig a flip-flop egyből másolja a bemenetét a kimenetre, és a lefutó élnél egyből rögzíti a kimenetét a következő felfutó élig.

Latch alkalmazásával a követő hálózatnak több ideje van az információ felhasználására és így gyorsabb működést kaphatunk. Akkor érdemes latchet használni, ha a bemenetet előállító hálózat az órajel 1 értéke alatt állítja elő az értéket, és ezt mi minél hamarabb fel akarjuk dolgozni.

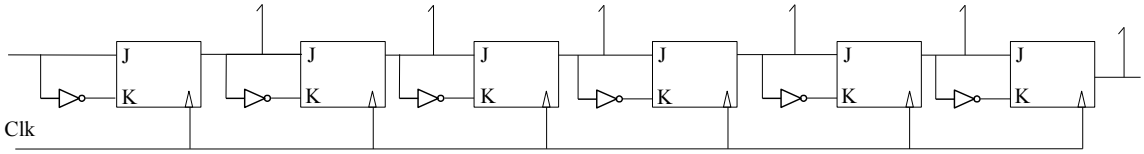
Léptető regiszter (shift regiszter):
Sorba kötött D flip-flopok, melyek azonos órajelet kapnak.
Mivel a kimenet előállítása időbe telik, ezért a következő flip-flop még az előző bemenetből vesz mintát, így egy jel végiglépked a flip-flopokon.



A shift regiszter működésére egy példát láthatunk a következő idődiagrammon:

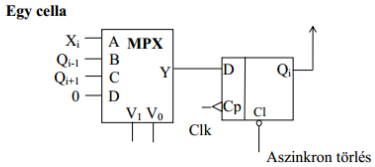


JK flip-floppal is megvalósíthatjuk a J=D és K=/D bekötéssel.



Általános regiszter:
Egészítsük ki a regiszterünk bemenetét egy 4 bemenetű multiplexerrel, és meg is kapjuk az általános shiftregisztert.

V_0	V_1	Funkció
0	0	Tölt
0	1	Balra léptet
1	0	Jobbra léptet
1	1	Töröl

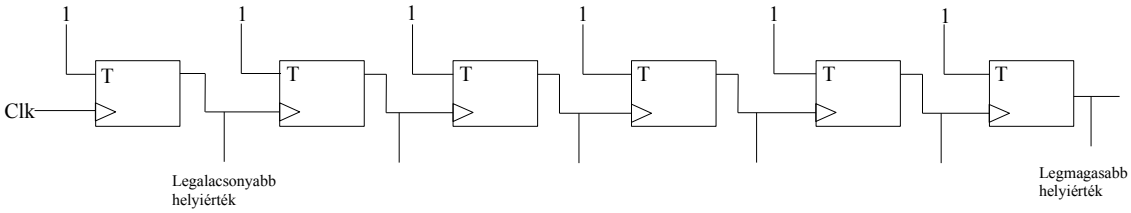


Aszinkron számláló:

Kössünk sorba T flipflopokat úgy, hogy minden T bemenetre 1-es értéket kapcsolunk, az első kapja meg az órajelet, a többi órajele legyen az előző flip-flop kimenete.
Ez a hálózat egy órajelfelezgető áramkör lesz. Minden flip-flop egyszer felezi az órajelet.

Felfelé és lefelé számláló hálózatra is szükségünk lehet:

- Ha kivezetjük a kimeneteket, és az utolsó flipflopot tekintjük a legmagasabb helyiértéknek, akkor egy visszafele számláló hálózatot kapunk.
- Amennyiben a negált kimeneteket vezetjük ki, és azokat használjuk, a hálózat felfelé számlál.
- Ha órajelként a Q kimenet helyett /Q visszük tovább szintén egy felfelé számláló számlálót kapunk.



Az ábra egy lefelé számláló számláló elvi logikai rajzát mutatja.
Az alaphelyzetbe állítás a /CL és /PR bemenetek használatával lehetséges.

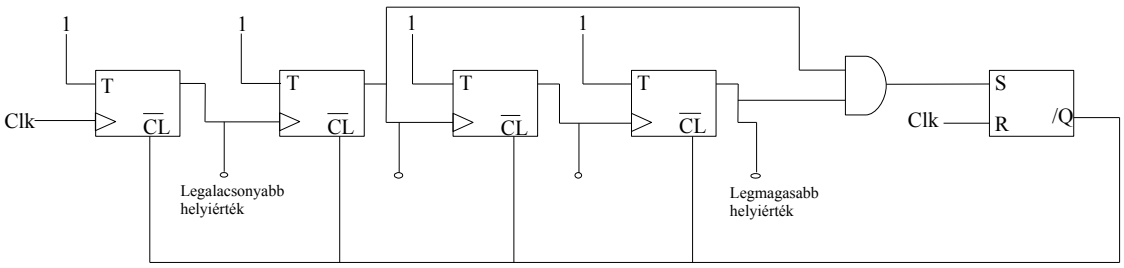
Azért nevezzük aszinkronnak az ilyen kapcsolást, mert a helyiértékek nem egyszerre változnak, a jelnek az összes flip-flopon végig kell haladnia, és minden helyiérték késleltetve változik.

Decimális aszinkron számláló:

Készíthetünk decimális számlálót is, ha módosítjuk a fentebbi számlálónkat.

A 0-9 tartományban számláláshoz egy 0-15 tartományban működő, azaz 4flipflopból álló számlálót kell módosítanunk. A 10-es értéket kell észlelnünk a kimeneteken, és amint ezt érzékeljük, a /CL bemenetek segítségével a számlálót 0-ba kell állítanunk.
10-es érték: 1010
A 2^3 és a 2^1 helyiértékek egyszerre ebben az esetben veszik fel először az 1-es értéket, tehát ezeket egy NAND kapuval a /CL bemenetekre kötve (NAND, hiszen a /CL negált logikával működik), a számláló nullázza magát.

A /CL bemenet használatakor problémát jelenthet a késleltetés. Lehet, hogy nem sikerül minden flip-flopot kinullázni, mert az egyik túl gyorsan nullázódik, a kapu kimenete 1-re vált, és a többi flip-flopot nem sikerül kinullázni.
A probléma kiküszöböléséhez vegyünk egy SR flip-flopot. Az S bemenetre kössük a már fentebb tárgyalt helyiértékeket egy AND kapuval. Az S bemeneten tehát akkor lesz 1-es, amikor megjelenik az 1010 kombináció, azaz a 10-es érték. Ekkor a flip-flop kimenete 1-esre változik, és a negált kimenetet használva a flip-flopok nullázódnak. Kössük az R bemenetre az órajelet, ez fogja biztosítani, hogy egy fél órajelperiódus után az SR flip-flopunk negált kimenete újra 1 értéket vesz fel, így a törlő jel megszűnik. A fél periódus kellő széles kell, hogy legyen az összes flip-flop nullázására.



Szinkron számlálók:

Tervezzünk úgy egy szinkron számlálót, hogy Z=y
Egyszerű a feladat, egyből felvehetjük kódoltan is az állapottáblát egy engedélyező bemenettel:

y_1	y_2	y_3	E=0			E=1		
0	0	0	0	0	0	0	0	1
0	0	1	0	0	1	0	1	0
0	1	0	0	1	0	0	1	1
0	1	1	0	1	1	1	0	0
1	0	0	1	0	0	1	0	1
1	0	1	1	0	1	1	1	0
1	1	0	1	1	0	1	1	1
1	1	1	1	1	1	0	0	0

Ha D flip-flopot használunk, akkor ez megfelel a vezérlési táblának.
Felírhatjuk a 3 darab vezérlési függvényt a Karnaugh-táblák alapján, majd az elvi logikai rajzot is.
Ha felvesszük a függvényeket, láthatjuk, hogy elég összetettek.

Lehetséges, hogy T flip-floppal egyszerűbb vezérlési függvények adódnak, ehhez azonban fel kell vennünk először a vezérlési táblát:

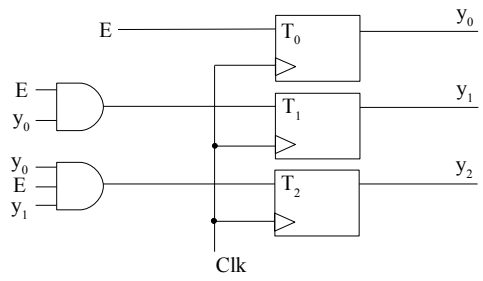
y_1	y_2	y_3	E=0			E=1		
0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	1	1
0	1	0	0	0	0	0	0	1
0	1	1	0	0	0	1	1	1
1	0	0	0	0	0	0	0	1
1	0	1	0	0	0	0	1	1
1	1	0	0	0	0	0	0	1
1	1	1	0	0	0	1	1	1

Vegyük fel a 3 karnaugh táblát a T_0, T_1, T_2 flip-flopokhoz és grafikus minimalizálással határozzuk meg a függvényeket:

$T_0 = E$
$T_1 = y_0 * E$
$T_2 = y_0 * y_1 * E$

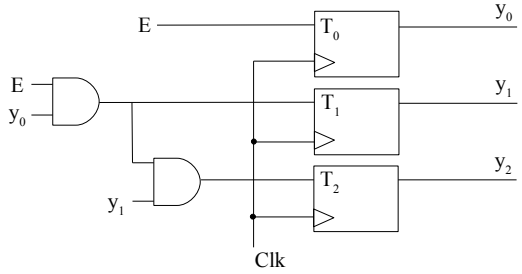
Ha megcsináltuk D flip-flopokhoz is a függvényeket, láthatjuk, hogy sokkal egyszerűbb eredményt kaptunk most.

Az elvi logikai rajzot is megadhatjuk ezek alapján:



A működési frekvenciát az előzőekkel ellentétben egy flip-flop és egy AND kapu késleltetése határozza meg.

Rajzoljuk kicsit át a hálózatot, mert n bit esetén n bemenetű AND kapura volna szükség.
Használjuk az előző szint AND kapujának kimenetét, hiszen az utolsó szinten lévő AND kapu 2 bemenetére pont ezt kötöttük.



Ekkor kiküszöböltük azt, hogy nagy bemenetszámú kapukat kelljen használnunk, azonban a késleltetés megnő, hiszen a jelnek több AND kapun kell áthaladnia.

Tízszeres órajelosztás:

Tervezzünk egy hálózatot, amely tizedére osztja az órajelet.

Ha minket csak az érdekel, hogy mikor jön felfutóél, akkor a fentebb tárgyalt 10-ig számláló hálózat tökéletesen megfelel, amennnyiben csak a legmagasabb helyiértéket vezetjük ki.
Ekkor azonban 8 időegységig lesz 0 a kimeneten, és csak két időegységig lesz 1 az órajel értéke.
Ugyan a felfutó élünk jó helyen lesz, hiszen két felfutó él között pontosan 10 időegység telik el, de a kitöltési tényező csak 1/5.

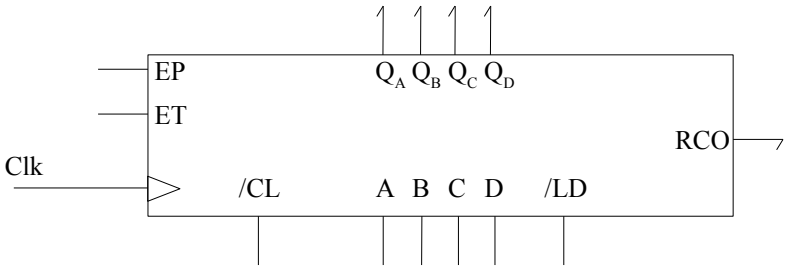
Tervezzünk egy hálózatot, amely tizedére osztja az órajelet úgy, hogy a kitöltési tényező ½, azaz az órajel pontosan ugyan annyi időegységig vesz fel 0 értéket, mint 1 értéket.

A megoldást az jelenti, ha először előállítunk egy ötödére osztott órajelet a következő módon:
A fentebb tárgyalt 10-ig számláló hálózatnak valósítsuk meg az egyel kevesebb bitet tartalmazó verzióját, amely csak 5-ig számlál. Ekkor kaptunk egy az előbbihez elég hasonló órajelet, aminek a felfutó élei a számunkra megfelelő helyen vannak.
Ha ezzel az órajellel vezérlünk egy felére osztó hálózatot (T flip-flop, melynek bemenete 1), akkor pontosan a specifikációknak megfelelő áramkört kapjuk, ami a Q és a /Q kimeneten is a számunkra megfelelő órajelet adja, azonban eltérő fázisban.
Ha az abszolút fázis fontos a követőhálózatok számára, mérlegelnünk kell, hogy az adott esetben melyik fázis előnyösebb számunkra.

Számlálók gyakorlatban:

Jegyezzük meg, hogyha számláló kimeneteit negáljuk, a számlálás irány változik.
A nagy késleltetés miatt az aszinkron számlálók kimenetét a valóságban nehéz számlálási feladatokra használni, órajel leosztásra viszont egy kimenetet tekintve alkalmasak, bár az előálló leosztott órajel az eredeti órajelhez képest más fázisban van, ami egyes alkalmazásoknál nem megengedett.

A valóságban egy számlálón jóval több bemenet és kimenet található, mint az elvi rajzokon.
Nézzük ezeket sorra:



Ahogy azt már megszoktuk, itt is találunk egy clear bemenetet, mely lehet szinkron és aszinkron is attól függően, hogy mire szeretnénk használni a számlálónkat. Természetesen a /CL bemenet itt is negált logikával működik, azaz akkor lesz aktív, ha 0 értéket kapcsolunk rá.

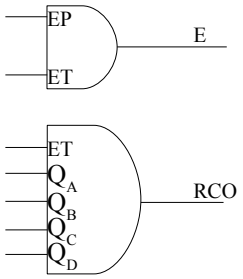
- Szinkronra akkor van szükségünk, ha adott érték után 0-tól szeretnénk folytatni a számlálást a következő órajelre.
- Aszinkronra akkor van szükségünk, ha a számlálástól teljesen független esemény hatására szeretnénk nullába állítani a számlálónkat.

Szeretnénk tetszőleges értéktől is számlálni. Ehhez 4 bemenet és a hozzájuk tartozó load vezérlő bemenet áll rendelkezésünkre. Ez is negált logikával működik, tehát az ABCD bemeneteken fennálló bináris számot akkor tölti be a flip-flopba, ha az /LD-re 0 értéket kapcsolunk. (D változó az MSB)
A /LD vezérlő bemenet mindig szinkron módon működik!

Az /LD és a /CL között prioritást kell definiálnunk arra az esetre, ha mindkettőre egyszerre kötnénk 0 értéket.
Aszinkron /CL esetén a /CL azonnal töröl.
Szinkron /CL esetben általában a /CL élvez nagyobb prioritást.

A számláló a kaszkádosításhoz tartalmaz egy RCO (ripple carry out) kimenetet, melyen 1 érték jelenik meg akkor, ha a számláló elérte a maximális értékét.

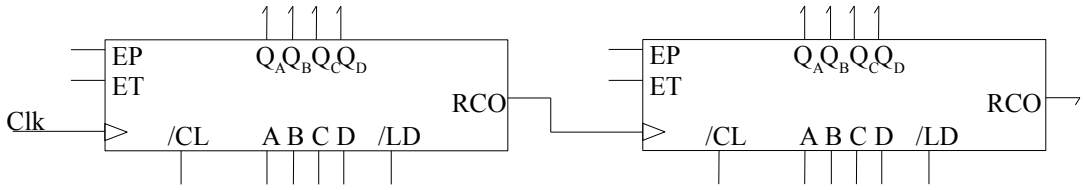
Elvárás lehet, hogy megmondhassuk a számlálónknak, hogy éppen nem akarunk számolni.
Ezen funkció eléréséhez engedélyező bemenetek állnak rendelkezésre EP és ET néven, melyek a következő módon működnek:



ET-nek és a Q_i-knek mindnek 1-nek kell lennie, hogy az RCO értéke 1 legyen.
EP-nek és ET-nek 1-nek kell lennie, hogy a számláló számoljon.
Ennek miértjére mindjárt választ is adunk.

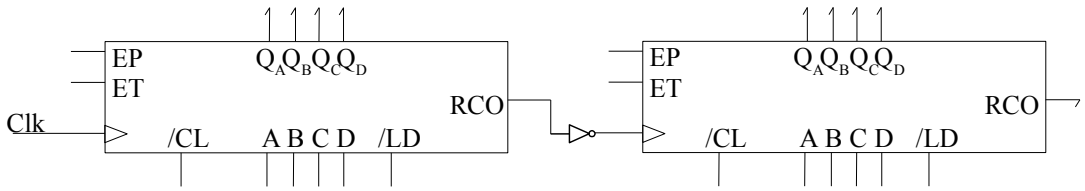
Nagyobb számlálók megvalósítása kisebbek segítségével:

a.)



Első ötletünk az volna, hogy kössük sorba a számlálókat úgy, hogy az RCO kimeneteket kötjük be, mint órajel.
Ekkor azonban a számláló úgy számolna, hogy a 15 után 31 érték jelenne meg, és csak utána a 16, 17 …

b.)



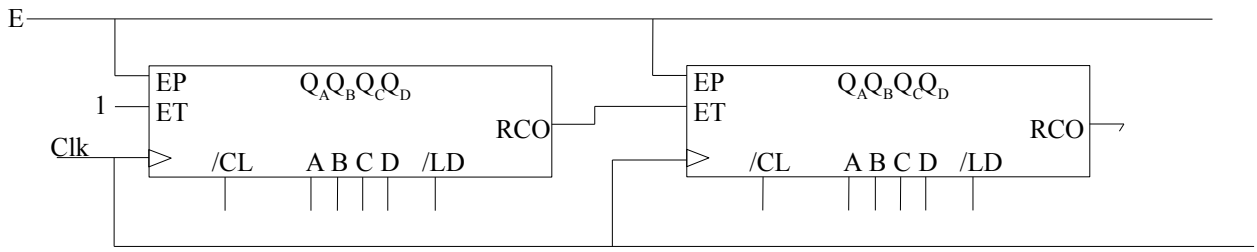
Mi történne, ha a negált RCO értéket vinnénk tovább órajelként. Ekkor egy aszinkron számlálót kapnánk, de a számlálás már helyes volna. Problémába ütköznénk azonban nullázásnál, mivel a második számláló nem kapna órajelet, így nem tudnánk nullázni, a számlálás nullázás után újra rossz volna.

Még nem használtuk ki a két engedélyező bemenet és a működésük által nyújtott előnyöket.

Ezek a megoldások rosszak!

c.)

Szinkron számlálóink vannak, tudjuk, hogy minden számlálóegység egyszerre kell, hogy kapja az órajelet.
Nézzük meg, hogy soros kaszkádosítással milyen megoldást kapunk, és mi lesz a probléma:

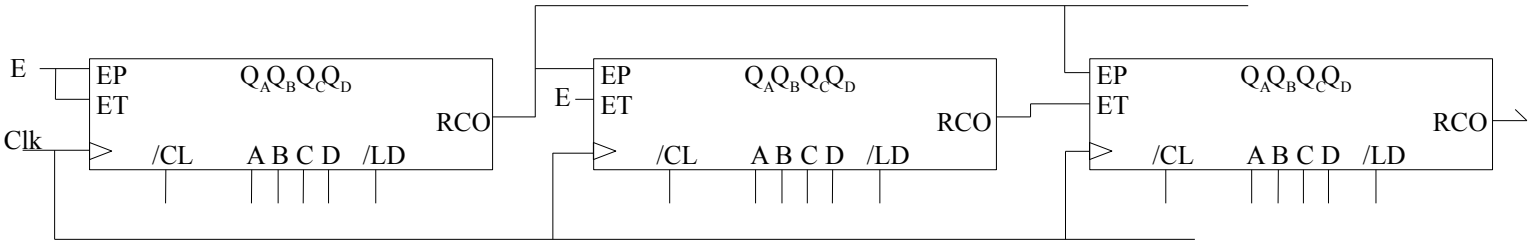


Az E engedélyező jelet kössük az EP bemenetekre.
Első ET-t engedélyezzük, a többi ET bemenet mindig legyen az előtte levő RCO kimenetére kötve.

A számlálás helyes, a törlés működik, a megoldással viszont van egy probléma.
Az ET és az RCO jel mögött minden egységben egy AND kapu húzódik meg. Így abban az esetben, amikor a kisebb helyiértékeket előállító számlálók mindegyike át kell, hogy billenjen (például a 255 → 256 átmenet esetén) az RCO jel késleltetése n-1 darab AND kaputól függ, ami nagyban befolyásolja a működési frekvenciát, és ez számunkra már nem is elegendő, így meg kell alkotnunk a 4. megoldást is.

d.)

Úgynevezett **soros-párhuzamos kaszkádosítással** sokkal gyorsabb hálózatot kaphatunk.
Időt nyerhetünk, ha az első számlálót rendesen engedélyezzük, azonban a második számlálótól az EP-kre kötjük az első RCO jelét, és a soros kaszkádosításhoz hasonlóan ET-kre kötjük az előző RCO kimenetet.



Nagy előnye ennek a megvalósításnak, hogy így 16 órajelperiódus ideje van az első RCO kimenet jelének, hogy elérje az utolsó számlálót is, ami annyi idő, hogy lehetővé teszi, hogy ilyen felépítéssel nagyjából 200bit-es számlálót is megvalósíthassunk.

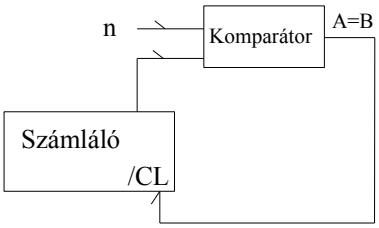
Képzeljük el ismét a 255 → 256 átmenetet, melyhez 3 darab 4 bites számláló szükséges.
Az előző megoldás szerint ekkor a második számláló a 15-ös érrtéken áll, így az RCO jele 1-es értékű lehetne, amennyiben az ET-n engedélyezve volna az RCO kimenet. Amikor a legkisebb helyiérték eléri a 15-ös értéket, akkor az RCO kimenetén megjelenik az 1-es, engedélyezi az egyel nagyobb helyiérték ET jelét, így annak RCO-ja 1-es értéket felveheti. Így a 3. számláló egy AND kapukésletetéssel később kapja meg az engedélyt a számlálásra.

A soros-párhuzamos kaszkádosítás esetén viszont, mivel az ET engedélyezve van a második számlálónál, így az RCO jel folyamatosan terjed. Így amikor a számláló eléri a 240 értéket, akkor a második számláló összes kimenete már 1-es értéket vett fel, és mivel az RCO jel engedélyezve van, ezért a 3, magasabb helyiértékeket előállító számláló 240-es érték elérésétől folyamatosan tudja, hogy neki számolnia kell majd.
Amikor eléri a számlálólánc a 255-ös értéket, akkor az első, legalacsonyabb helyiértéket előállító számláló RCO kimenete egyből engedélyezi a többi számlálót, így a működési sebesség sokkal nagyobb lehet.

Több komponenst kötve párhuzamosan még nagyobb bitszámmal valósíthatjuk meg számlálónkat.

0 … n érték közti számlálás:

Valósítsunk meg 0 ... n értékek közt számlálót:



Szinkron /CL szükséges a biztos működéshez!

Aszinkron számláló esetén a késleltetések miatt például a 011 → 100 átmenetnél egészen biztosan 010 → 000 köztes átmenet adódik. Szinkron számláló esetén tranziensben nem tudjuk megmondani egy pillanatig a kimenetet, ekkor előfordulhat véletlenül egy olyan érték, amit a komparátor A=B-nek érzékel, és törli a számlálót. Ez tehát rossz működést eredményezhet.

Szinkron számláló esetén aszinkron törlést nem használunk, kivéve ha a külvilágtól érkező jellel akarunk törölni, ami a számlálástól független.

Hogy oldhatjuk mégis meg?
A komparátor A=B kimenetét az /LD bemenet engedélyezésére használjuk mely minden esetben szinkron, és a 0 értéket töltsük be a számlálóba.

Számlálók csoportosítása a gyakorlatban:

a.)
- Bináris számlálók
- BCD számlálók
- Gray számláló:
  - Minden órajelperiódusban szomszédos kódot ad ki. Ezzel a követőhálózatban kikerülhetjük a funkcionális hazárdot.

b.)
- Felfelé számlálók
- Lefelé számlálók
- Két irányú számlálók
  - RCO értelmezése picit más, ekkor MAX/MIN-nek nevezik általában.

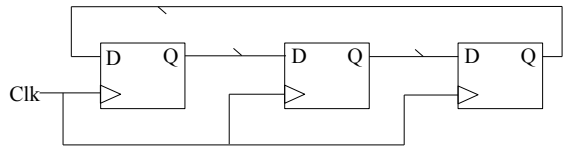
c.)
- Szinkron /CL bemenettel rendelkező számlálók
- Aszinkron /CL bemenettel rendelkező számlálók

Jegyezzük meg jól:
Aszinkron törlő bemenetet tilos a kimenetek és kombinációs hálózat segítségével előállított törlőjellel vezérelni. Tranzienben a szinkron számláló kimenetén bizonytalan érték található, így előfordulhat, hogy két szám közti váltás esetén a kombinációs hálózat pont olyan értéket kapna a bemenetein, amivel a számlálót törölné.

Speciális számlálók:

Gyűrűs számláló:

Tekintsük a következő hálózatot:
Kössünk sorba 3 darab D flip-flopot úgy, hogy közös órajelet kapnak, és minden bemenetre kapcsoljuk az előtte levő flip-flop bemenetét, így az első bemenetére az utolsó kimenetét kell kötnünk.



Kis gondolkodás után rájöhetünk, hogy a hálózat az 1-es értéket lépteti körbe:

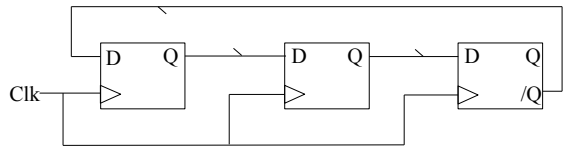
100
010
001
100
010
…

Bekapcsoláskor azonban nem garantált a flip-flopok megfelelő állapota így a /PR és /CL bemenetekkel kell biztosítanunk az 100 értéket egy START jellel.

Itt könnyedén önbeálló hálózatot is készíthetünk, ha az első D-re $Q_1 * Q_2$ értéket kötünk.
Ekkor a hálózat véges lépésben 100 állapotba kerül, ahonnan megfelelően kezd el számlálni.

Móbius vagy Johnson számláló:

Amennyiben a gyűrűs számlálót úgy módosítjuk, hogy az utolsó flip-flop Q kimenete helyett a /Q kimenetét csatoljuk vissza az első bemenetére, akkor szintén periodikus számlálót kapunk, azonban a következő módon számlál:
Tegyük fel, hogy 000 értékről indulunk: 000, 100, 110, 111, 011, 001, 000 …

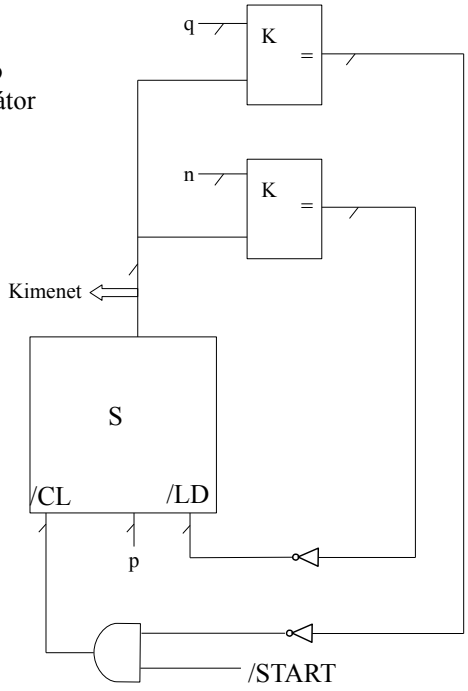


A kezdeti érték beállításához itt is a /PR és /CL bemeneteket kell használnunk.

MP:
Tervezzünk ciklikus számlálót, mely a következő módon számlál:
0 … n → p … q → 0 … n

Készítsünk egy félig blokksémát:

S: számláló
K: komparátor



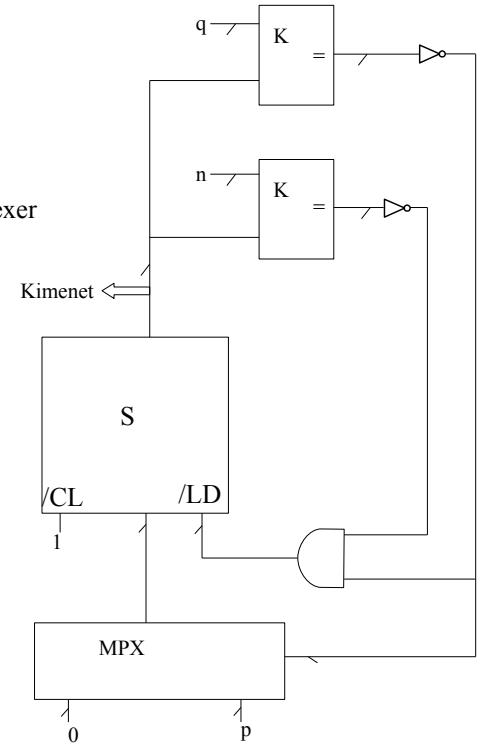
Nézzük a működést:

A számláló elindul, majd ha a kimenet eléri az n értéket, akkor az alsó komparátor 1-et ad ki a kimenetén, ezt invertálnunk kell, hiszen a load negált logikával működik, és ekkor betöltődik a p érték, és innen folytatódik a számlálás.
Majd a számláló eléri a q értéket, ekkor a felső komparátor kimenetén jelenik meg 1 érték.
Ezzel közvetlenül vezérelhetjük a szinkron /CL bemenetet.
Ekkor önbeálló hálózatot kapunk.

A példában azonban egy START jelet is definiáltunk, amivel 0-ról indíthatjuk a számlálónkat. Ha a /CL bemenetet akarjuk vezérelni, akkor a második komparátor kimenete 1, ekkor az AND kapu bemenete 0, tehát a /CL aktív.
A /START jelet ha nullázzuk, akkor a hálózat 0-ról indul.

Mi történik, ha **aszinkron /CL** bemenetünk van:
Ekkor egy multiplexerrel kell választanunk, hogy mely értéket töltsük be (0 vagy p).
A /LD-t a két komparátor negált kimeneteinek AND kapcsolatával vezéreljük.

S: számláló
K: komparátor
MPX: multiplexer



Nézzük a működést:

A számláló elindul, majd egyszer csak az egyik komparátor jelzi, hogy elértük az értékét. Ekkor a negált érték 0-ra vált, az AND kapu kimenete 0-ra vált, és a /LD aktív lesz, és betölti a kiválasztott értéket, melyet egy multiplexer választ. Vagy 0-t vagy p-t tölti be, ehhez megfelelő multiplexer szükséges.

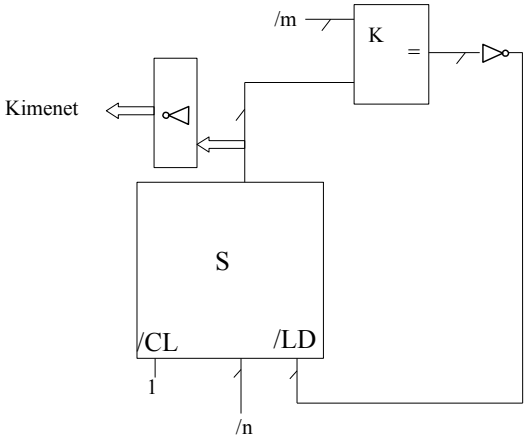
A multiplexert a q-t, azaz a felső határt érzékelő komparátor negált kimenetével vezéreljük, tehát:
$0 = /K_q$ és $p = K_q$

A START jelet most a /CL bemenetre egyszerűen ráköthetjük, hiszen az aszinkron működésű.

MP:
Tervezzünk ciklikusan visszafele számlálót, mely az n … m tartományon visszafele számlál:

A megoldás nagyon egyszerű: tervezzünk egy felfele számlálót, ami /n -től /m -ig számlál, és a felhasználónak mutassuk meg a bitenként invertált kimenetet, hiszen ha egy felfelé számláló kimenetét bitenként invertáljuk, pont egy visszafelé számlálót kapunk.

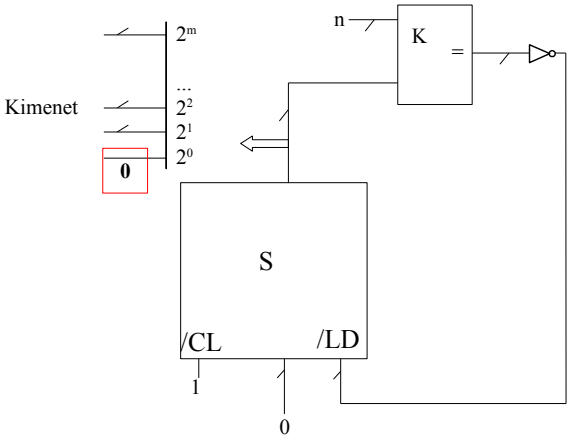
Elméleti háttér: visszafele is pontosan ugyan úgy bitenként változnak a számok. Ha n-től számolunk vissza m-ig, akkor n nagyobb mint m, tehát /n kisebb mint /m, azaz /n-től kell előrefele számolnunk /m-ig.



Az inverterrel jelölt doboz a bitenkénti negálást végzi. Bitek számának megfelelő számú inverterből áll.

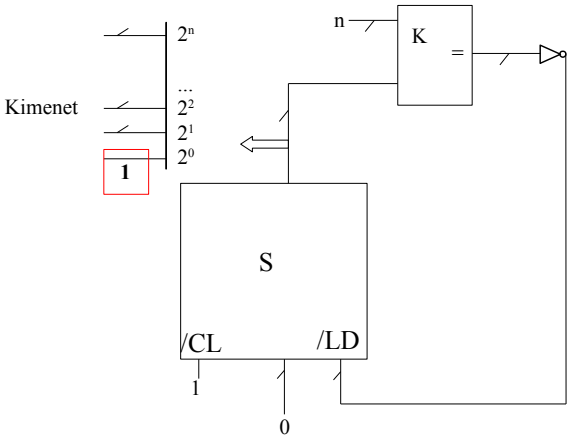
MP:
Tervezzünk számlálót, mely a 0 ... 2n tartományban párosával számol:

A megoldás rendkívül egyszerű itt is, a páros számokról tudjuk, hogy utolsó bitjük 0-s.
Csúsztassuk odébb a biteket egyel, és az utolsó helyiérték helyére kössünk fixen 0 értéket.



MP:
Tervezzünk számlálót, mely a 1 … 2n+1 (tehát a páratlan számokon) tartományban párosával számol:

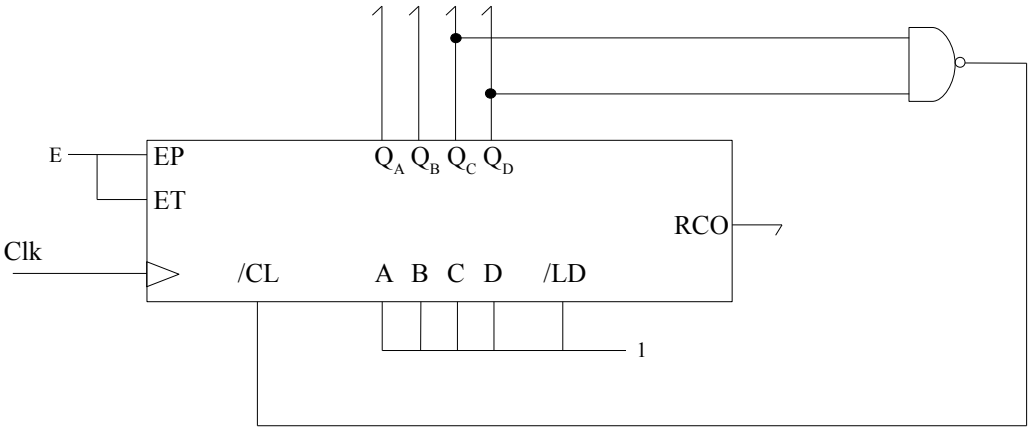
A megoldás itt is rendkívül egyszerű, az előző megoldásunk tökéletes, ha az utolsó bitet nem 0-ra, hanem 1-re fixáljuk.



Ugyan ezt a sémát használhatjuk akkor is, ha menet közben szeretnénk eldönteni, hogy páros, vagy páratlan számokkal szeretnénk számolni. Egy flipflopra és egy vezérlésre lesz csupán szükségünk. Erre a legegyszerűbb példa egy D flip-flop és egy v vezérlőjel (v=0 → páros, v=1 → páratlan).

MP:
Valósítsunk meg egy 0 … 12 tartományban ciklikusan számláló áramkört az adott szinkron load és szinkron clear bemenetekkel rendelkező, 4 bites bináris számláló segítségével és minimális kiegészítő hálózattal.
Figyeljünk arra, hogy a bekötetlen bemenetek hibapontot vonnak maguk után, és a minimális kiegészítőhálózat legyen valóban minimális.

Eddig egyszerű komparátorral jelöltük a kiegészítő hálózatot, itt azonban egy NAND kapu elég, ami a két legmagasabb kimenetet figyeli, hiszen a 12 4 biten ábrázolva az 1100 érték, ami kisebb számoknál nem fordul elő.
Szinkron clear bemenetünk van, tehát használhatjuk azt, figyeljünk arra, hogy az 0-ban aktív!



A /LD 0 esetén aktív, így rá 1 értéket kell kötnünk.
ABCD bemenetekre mindegy mit kötunk, hiszen /LD már nem aktív.
EP és ET-t engedélyeznünk kell, hogy számolhassunk.

Aszinkron /CL esetén a /LD-t kéne vezérelnünk, és az ABCD bemeneteken 0-t kéne betöltenünk.

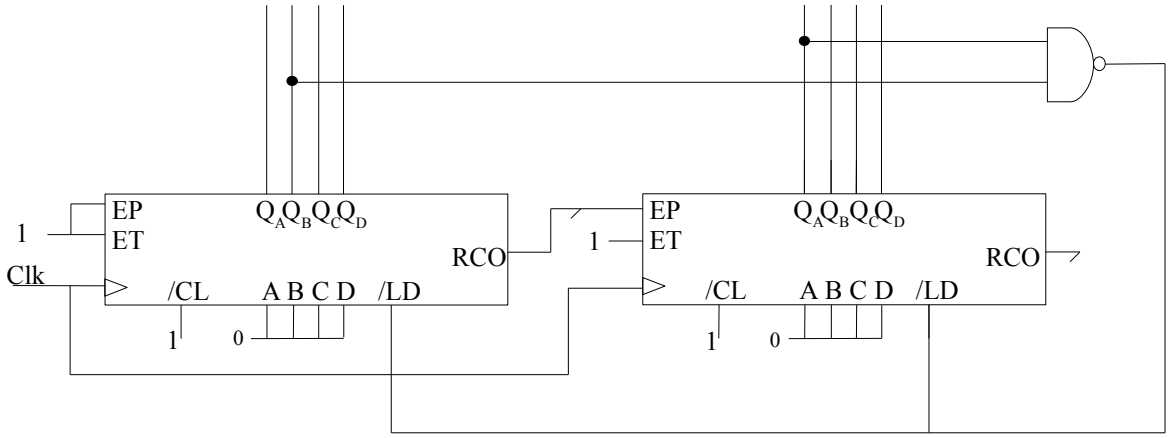
MP:
Valósítsunk meg egy 0 … 12 tartományban ciklikusan számláló áramkört az adott szinkron load és aszinkron clear bemenetekkel rendelkező, 4 bites BCD számláló segítségével és minimális kiegészítő hálózattal.
Figyeljünk arra, hogy a bekötetlen bemenetek hibapontot vonnak maguk után, és a minimális kiegészítőhálózat legyen valóban minimális.

BCD-ben számolunk, ahol a 12 8 bittel van leírva, tehát két darab számlálóra lesz szükségünk.
Az RCO-t az EP bemenetre kell továbbkötnünk. Figyeljünk arra, hogy az első számláló állítja elő a legkisebb helyiértéket, azaz a számok sorrendje pont a fordítottja a számlálók sorrendjének.

Az első számlálót egyszerűen engedélyeznünk kell. A második számláló ET bemenetét engedélyezzük.
Aszinkron a /CL, így azt nem használhatjuk. Kössük fixen 1 értékre.
A load bemeneteket kell használnunk a 0-zásra, kössük őket 0-ra, hogy ezt az értéket töltsék be.

Beszéljünk kicsit a működésről is, ugyanis nem biztos, hogy midnenki számára magától értetődő a BCD számláló haszálata. A bináris számláló 15-ig számol, és ekkor jelzi az RCO kimeneten, hogy a következő számláló (a magasabb helyiérték) léphet 1-et. A BCD számláló ezt már a 9-es érték után megteszi.
Az első számláló elkezd számolni: 0, 1 … 9, ekkor a második számláló végig 0 értéken áll, tehát mi 00, 01 … 09 értékeket látunk. Amint elérte a 9-et, jelzi a következő számlálónak, hogy léphet egyet. Ekkor az lép 1-et, és az előző számláló kezdi elölről a számlálást, tehát mi a 10, 11 … 19 értékeket látjuk.

Nézzük meg a 12-est BCD számként:
12 = 0001 0010
Egyértelmű, hogy 12-ig a 4. bit csak 10, 11, 12-nél 1-es. Ekkor a 0001 0000, 0001 0001, 0001 0010 értékeket látjuk. Jól látható tehát, hogy elég a két darab 1-est figyelni, és ha azok értéke együttesen 1-es, akkor kell nulláznunk a számlálóink.
Az A változó az LSB, a D változó az MSB. A bal oldali az alacsonyabb helyiérték!



MP:
Valósítsunk meg egy 21 … 39 majd majd 72 … 101 tartományban ciklikusan számláló áramkört az adott szinkron load és szinkron clear bemenetekkel rendelkező, 4 bites bináris számláló segítségével és minimális kiegészítő hálózattal. Figyeljünk arra, hogy a bekötetlen bemenetek hibapontot vonnak maguk után, és a minimális kiegészítőhálózat legyen valóban minimális.

101-ig akarunk elszámolni, látható, hogy egy számláló nem elég. Két kaszkádosított számlálóval 256-ig is el tudunk számolni, tehát két darab már elég és szükséges is.

Nullázásra nincs szükségünk, a /CL bemeneteket kössük is 1-es értékre, hiszen nem fogjuk őket használni.

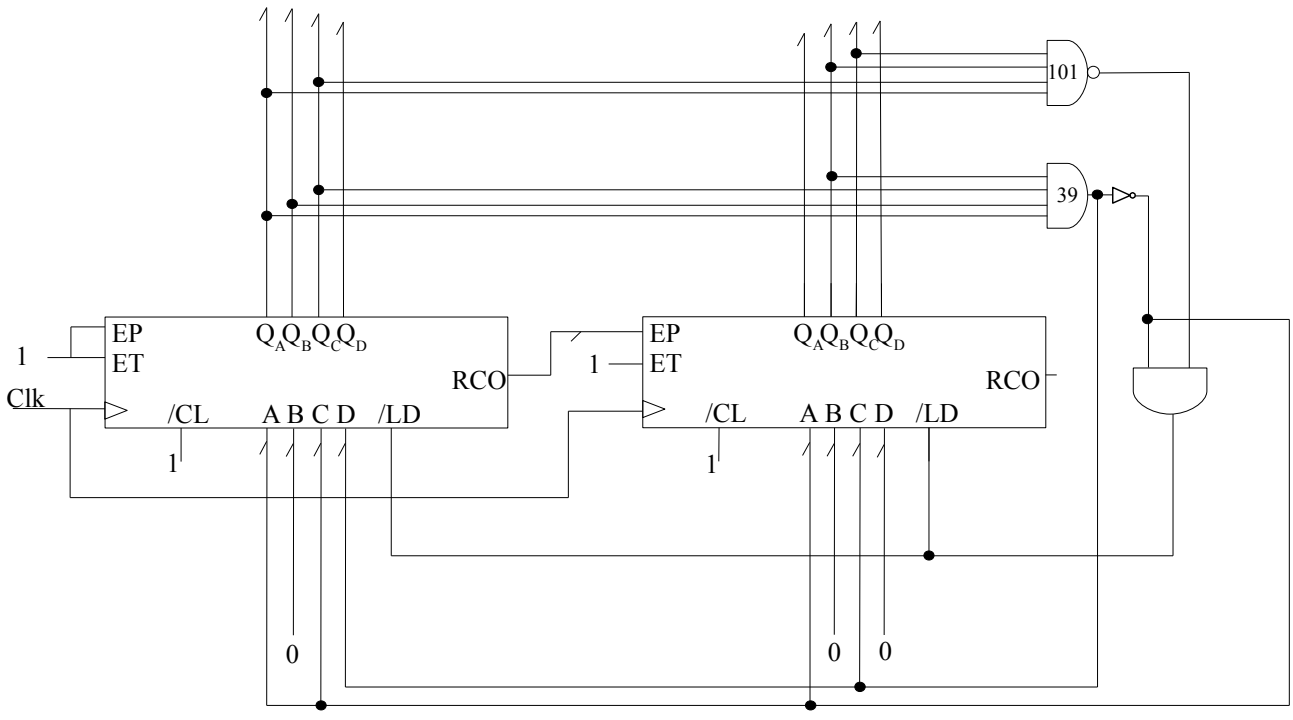
Ezt is megvalósíthatnánk komparátorokkal és multiplexerrel, de minimális kiegészítő hálózatot kértek.
Vizsgáljuk meg a számainkat helyiérték szerint, hogy ténylegesen minimális kiegészítőhálózatot készíthessünk:
Figyeljünk arra, hogy visszafele írtuk fel a helyiértékeket, mivel a rajzon is így szerepelnek!

	$2^0 = 1$	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$	$2^5 = 32$	$2^6 = 64$	$2^7 = 128$
21	1	0	1	0	1	0	0	0
39	1	1	1	0	0	1	0	0
72	0	0	0	1	0	0	1	0
101	1	0	1	0	0	1	1	0

Induljunk 21-től, és nézzük mit kell figyelnünk, ha elérjük a 39-et. A 39-es legnagyobb helyiértéke a 6. .
Amikor a 7. helyiértéket léptetjük akkor 103-at kapunk, ami már kinn van a számolási tartományainkból, tehát elég a 39-es 1-es értékeit vizsgálnunk egy NAND kapuval (=AND + inverter) ismét.

Mit kell vizsgálnunk a 101-nél: egészen bizonyosak lehetünk abban, hogy nem fog előfordulni a számolás során még egyszer, mivel a következő érték amiben szerepel, az a 229, ami a tartományon kívül van bőven.

Mit kell a load bemenetekre kötnünk (ne használjunk multiplexert, gondolkozzunk):
Vannak olyan helyiértékek, amiket beköthetünk, hiszen akár 21-et, akár 79-et kell betöltenünk, azok a helyiértékek nem változnak. Ezeket sárgával jelöltük a táblázatban.
Nézzük mit kell kötnünk például a legkisebb helyiértékre:
2^0 : Ha a 39-et érzékeljük, akkor 0-t kell ide kötnünk. Ha nem a 39-et érzékeljük, akkor ide 1-est kell kötnünk.
Tehát használhatju a 39-es kapu inveter utáni jelét, ugyanis az pont az elvártak szerint működik.
Az összes többi helyiérték hasonló módon áll elő, van ahova az inverter előtti jelet kell kötnünk.

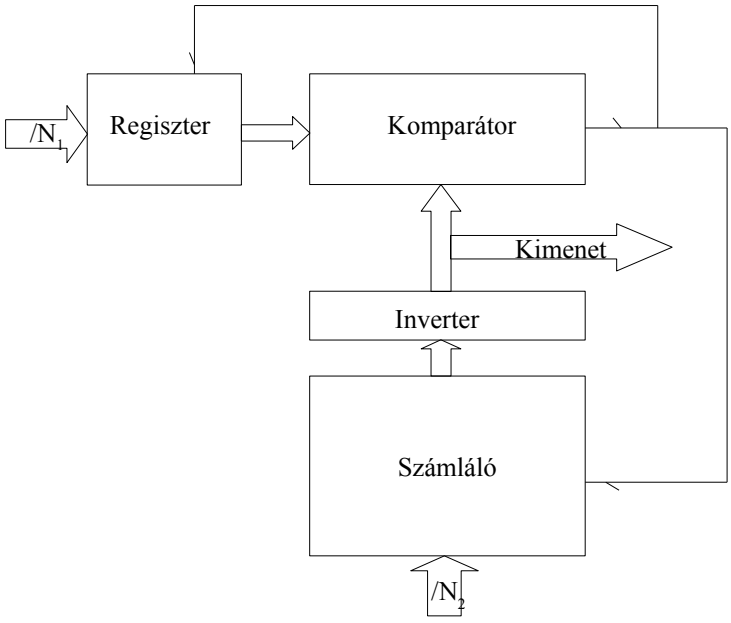


MP:
Valósítsunk meg egy N_2 … N_1 tartományban 5 biten ciklikusan visszafelé páratlan számokkal számláló áramkört az adott elemek és minimális kiegészítő hálózat felhasználásával.
N_2 és N_1 kívülről megadhatóak, ciklus közben nem módosíthatóak, és biztosítva van, hogy $N_1<N_2$ és mind a kettő páratlan.
Figyeljünk arra, hogy a bekötetlen bemenetek hibapontot vonnak maguk után, és a minimális kiegészítőhálózat legyen valóban minimális.

Adott építőelemek:
74163-as számláló melyről tudjuk, hogy szinkron clear és szinkron load bemenetekkel rendelkező 4bites bináris számláló.
4 bites komparátor.
8 bites regiszter.
JK flip-flop.

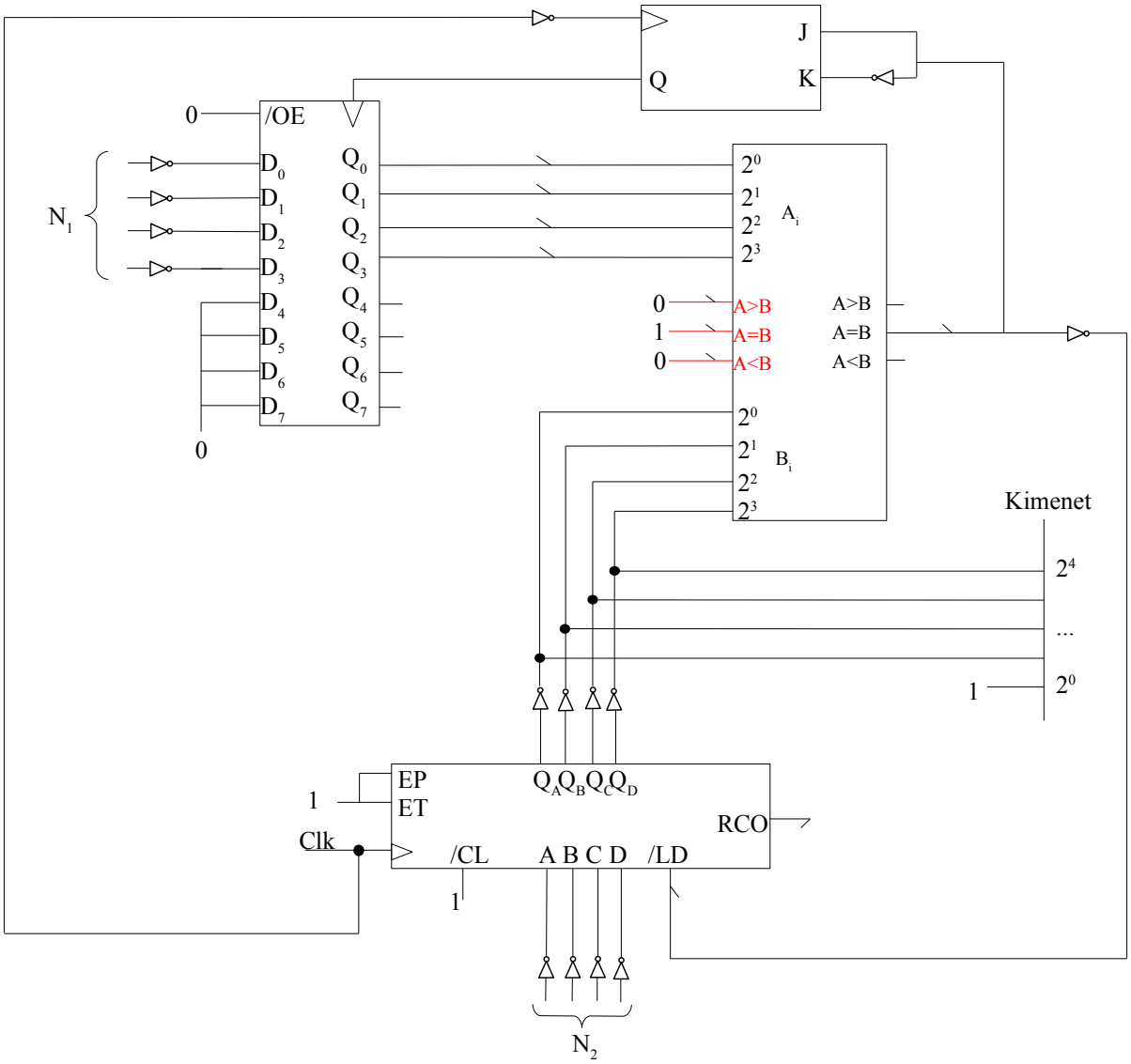
Nézzük mire kell figyelnünk:
5 biten kell számolnunk, de mivel csak páratlan számokon számolunk, ezért az utolsó bitet 1-re kell fixálnunk. Így nem lesz szükség két számlálóra.
A bevitelnél az utolsó bitet nem is kell megkérdeznünk éppen ezért a felhasználótól.
Visszafelé számlálást úgy valósíthatjuk meg, hogy /N_2-től számolunk előre /N_1-ig. A felhasználónak bitenként negált kimenetet mutassunk.
A regisztert arra használhatjuk, hogy a ciklus közben ne módosuljon az érték, csak meg kell találnunk a megfelelő órajelet hozzá. Ezt az órajelet a komparátor bemenetéről fogjuk levenni, ehhez lesz szükségünk majd a JK flip-flopra.

Készítsük el a blokksémát:



Valósítsuk meg a feladatot a konkrét építőelemekkel:

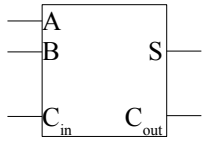
A regiszternek van egy /OE bemenete, mivel a kimeneteit magas impedanciás állapotba rakhatók.
Erre nekünk nincs szükségünk, engedélyeznünk kell, tehát 0 értéket kell rákapcsolnunk.
8 bites a regiszter, mi azonban csak 4 bittel számolunk, a másik 4 bemenetet 0-ra köthetjük.
0 értékre nincs szükségünk, így a számláló /CL bemenetét kössük 1-re.
Az engedélyező bemenetkre sincs szükségünk a számlálón, így azokat is 1-re köthetjük.
JK flip-flopot kaptunk, azonban nekünk egy D flip-flop tökéletes volna, így J=D, K=/D bekötést alkalmazzunk.



A feladatban egyetlen egy ismeretlen rész a JK flip-flop, mely a bekötés miatt D flip-flopként viselkedik, és memória szerepet tölt be.
Eredetileg felfutóél vezérelt, így az inverter miatt lefutóél vezéreltté válik. Mivel tranziensben nem tudjuk a kimenetét a szinkron számlálónak, és így a komparátornak sem, ezért előállhatna hibás órajel a regiszternek. Így viszont a flip-flop csak akkor tud mintát venni a komparátor kimenetéről, amikor az a helyes értéket mutatja. Ezzel kerüljük el azt, hogy rossz időpontban töltődjön be a felhasználó által megadott érték.

Aritmetikai műveletek:

Teljes összeadó:



A teljes összeadó rendelkezik 2 bemenettel, melyen 1-1 bites számokat fogad, egy C_{in} bemenettel, melyen azt érzékeli, hogy az előző helyiértéken képződött-e maradék (Carry). Kimenetként egy S-t találunk, amely az összeadás eredményét mondja meg, és egy C_{out}-ot, melyen a maradék jelenik meg.

1-1 bit összeadására képes.

A kimeneti függvények a következők:
$S = A \oplus B \oplus C_{in}$
$C_{out} = A*B + A*C_{in} + AC_{out}$

Több bites összeadót valósíthatunk meg, ha sorosan kapcsolunk ilyen elemeket.
Két n bites szám összege sok esetben csak n+1 biten fér el, tehát a legnagyobb helyiérték az utolsó C_{out} kimenet!
Így azonban nagyon nagy késleltetést kapunk, így nagy számok esetén ez a konstrukció nem túl működőképes.

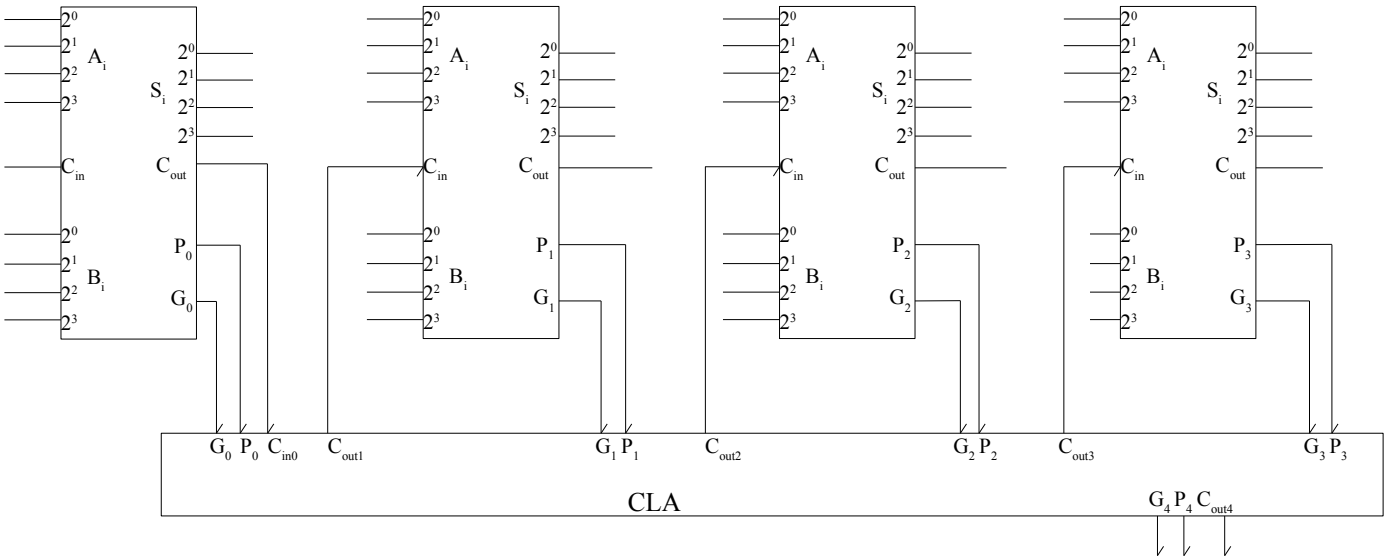
Carry-look-ahead felépítés:

Meg kell vizsgálnunk, hogy a Carry hogyan is áll elő.

- Egy-egy C_{out} a következő függvény szerint áll elő: $C_{out} = A*B + C_{in} *(A+B)$
- Jelöljük (A*B)-t G-vel és (A+B)-t P-vel: $C_{out} = G + C_{in} * P$
- Ekkor helyiértékek szerint egymásba helyettesítve a C_{in} értékeket az utolsó helyiértékre a következőt kapjuk: $C_{i+1} = G_i + P_iG_{i-1} + \ldots + P_i*P_{i-1} + \ldots P_1P_0C_0$

Tehát egy két szintű (ám bonyolult !) hálózattal előállíthatjuk az utolsó C_{out} értéket.
Az összeadók P és G értékét egy időben állítják elő, így a C_{in}-ek egyszerre előállíthatóak. A hagyományos esetben meg kéne várnunk, amíg a carry az egész hálózaton végighalad.

Valósítsunk meg egy 16 bites összeadót a fent kigondolt hálózat segítségével:
Vegyünk 4 darab 4 bites összeadót, mely előállítja a P-t és G-t is minden helyiértéknél, és vegyünk hozzá egy CLA hálózatot, mely ezek és az első C_{out} ismeretében előállítja a többi helyiértékhez a C_{in}-t, így kikerülve azt, hogy a carry nagy késleltetéssel haladjon végig a hálózatunkon.



Így egy darab összeadó és a CLA hálózat késleltetése határozza meg a működés sebességét!

Kivonás:

Összeadóval tudunk kivonni, ha ismerjük a 2-es komplemenst, mivel tudjuk, hogy A-B=A+(-B) .

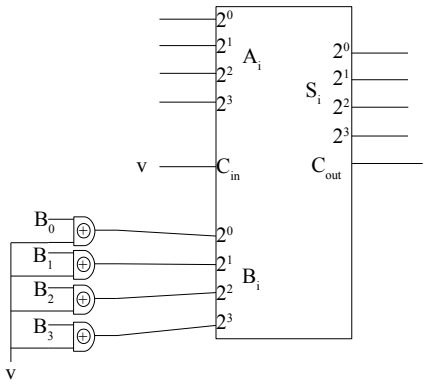
Kettes komplemens használatához a következőt kell tennünk:
Készítsük el B 1-es komplemensét. Ezt egyszerűen bitenkénti negálással végezzük el.
Egy összeadóval hozzá adhatnánk 1 bitet, de ez pazarló, így inkább az első öszeadó C_{in} bemenetére ne 0, hanem 1 értéket kössünk, és így el is készült a kivonásra használható áramkörünk.

Vezérelhető összeadó-kivonó áramkör:

Készítsünk olyan áramkört, ami vezérlőjel függvényében összead vagy kivon.

Egyszerű dolgunk lesz, ha a vezérlőjelet úgy definiáljuk, hogy v=0 esetén a hálózat adjon össze, és v=1 esetén a hálózat vonjon ki.

v=1 jelre kivonnunk kell, tehát szükségünk van egy olyan hálózatra, ami bitenként képes negálni, ha v=1-el vezéreljük.
Ez a hálózat rendkívül egyszerű, csak XOR kapukból áll.
C_{in}-re v-t köthetjük, így pont jó működést kapunk.



Ez a hálózat v=0 esetén pontosan úgy működik, mint egy összeadó. Nem változtatja B értékét.
v=1 esetén azonban a B értékeket negálja, egy bitet kap a C_{in} bemenetén, így az A-B műveletet valósítja meg.

Túlcsordulás:

A fentebbi hálózatoknál problémát jelent, hogy 2-es komplemens esetén túlcsordulás miatt helytelen értéket kapunk.
MP.: 4+6=10, de ez 2-es komplemensben a -6 érték!
MP.: -4 + -6 = -10, de ez a 6-os értéket eredményezi 2-es komplemens esetén.

- Összeadás:
  Túlcsordulás csak akkor fordulhat elő, ha azonos előjelű számokat adunk össze.
  A túlcsordulás figyelésére a következőt tehetjük:
  Figyeljük, hogy azonos előjelű számoknál milyen a végeredmény előjele (első biteket kell figyelnünk).
  Ehhez XOR kapu szükséges. Hogy az OVF ponált logika szerint működjön, XNOR kaput használjunk.
  $OVF = \overline{(X_3 \oplus Y_3)} * (X_3 \oplus S_3)$ vagy $\overline{(X_3 \oplus Y_3)} * (Y_3 \oplus S_3)$

- Kivonás:
  Túlcsordulás akkor lehet, ha különböző előjelű számokat vonunk ki egymásból.
  $OVF = (X_3 \oplus Y_3) * (X_3 \oplus S_3)$ (A kisebbítendő előjelével kell egyeznie!)

- Összeadó-kivonó hálózat:
  $OVF = /v * (X_3 \oplus Y_3) + v * (X_3 \oplus Y_3) * (X_3 \oplus S_3)$

BCD összeadó:

BCD számábrázolásnál 4 biten ábrázoltunk minden egyes decimális digitet.
4 bit esetén 16 érték áll elő, ebből azonban mi csak 10-et használunk. Ez pazarlás, azonban a nem használt 6 érték dont care érték, ami miatt az igazságtáblákban egyszerűsítések adódnak.

Valósítsunk meg 1 digites BCD összeadót:

Két 1 digites BCD szám összege 0 … 18 közt alakulhat, amennyiben kaszkádosítunk, akkor egy carry bit érkezhet az alacsonyabb helyiértékek felől, így a 19-es értéket is használnunk kell.
Azt gondolnánk, hogy ez 4 bites összeadóval nem valósítható meg, mivel a 19 BCD ábrázolás esetén 8 bit (minden digit 4 bit), azonban felhasználhatjuk, hogy az első digit csak 0 és 1 lehet, tehát két 4 bites szám összegét 5 biten kell megkapnunk. Az első bit jelöli a 10-esek számát, míg a következő 4 bit egy BCD szám 0...9 között.

Nézzük mi lehet két BCD digit összege és ez hogy viszonyul az általuk képviselt bináris számokhoz:
Amennyiben az eredmény 0 és 9 között van, nincs semmi problémánk, ugyanis két BCD szám összege megegyezik az általuk képviselt bináris számok összegével ebben az esetben.

Nézzük a BCD 10-et, és a bináris 10-et:
1 0000 → 0 1010

Az első érték a BCD 10-nek felel meg, míg a második érték a bináris 10.
Megfigyelhetjük, hogy a BCD 10 bináris értéke 16, azaz pont 6-tal több, mint az eredeti érték.
Ha tovább vizsgáljuk az értékeket, azt tapasztaljuk, hogy az általunk használt tartományban minden BCD szám 6-tal tér el az általa képviselt bináris számtól.
A bináris számokból tehát BCD számok úgy keletkeznek, hogy a bináris számok eredményéhez 6-ot adunk abban az esetben, **ha erre szükség van**!

Az 1 bites BCD összeadónk tehát úgy áll elő, hogy egy 4 bites összeadóval összeadjuk a két BCD számot mint bináris számok. Ezzel még nem oldottuk meg a 10 és nagyobb értékeket. Az előálló eredményt egy újabb 4 bites összeadóba vezetjük, és összeadjuk egy függvénnyel, ami a 6 értéket veszi fel ha szükség van erre (10 vagy nagyobb eredményt kell kapnunk), és a 0 értéket veszi fel, ha nincs szükség erre.

A második komparátor 0 vagy 6 értéket tartalmazó bemenetére tehát a 0SS0 értéket kell kötnünk, ahol a K logikai függvény a következő két módon állhat elő:

Ha a két szám összege nagyobb, mint 16, akkor az első összeadón carry bit keletkezik (mivel az 4 bites):

$K_1 = C_{out}$

Ha az eredmény 10 és 15 között van:

$K_2 = S_3S_1 + S_3S_2$

1010
1011
1100
…
1111

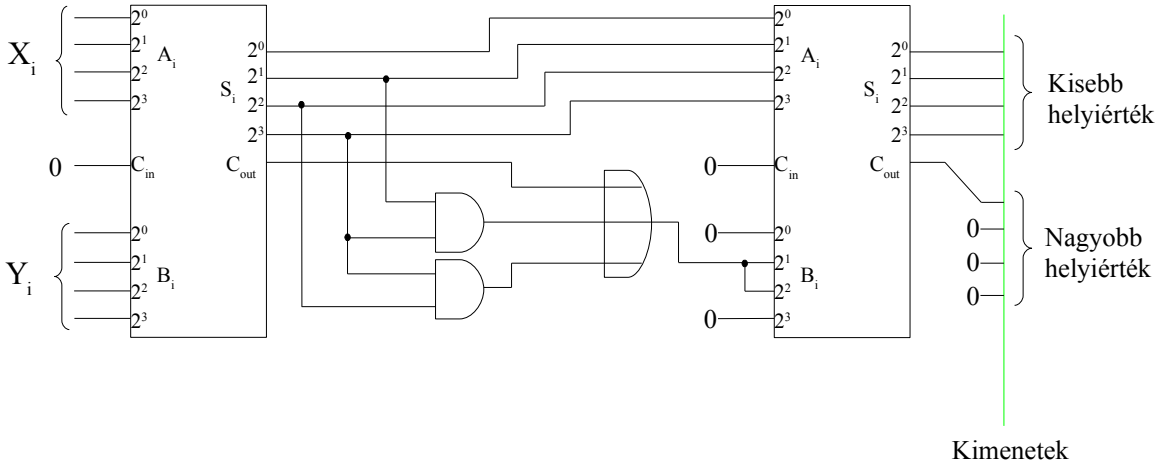
Ha az első és a harmadik, vagy az első és a második bit 1-es, akkor számunkra megfelelő szám van a kimeneten.

$\underline{K} = K_1 + K_2 = \underline{C_{out} + S_3S_1 + S_3S_2}$

Ilyen összeadókból tetszőleges méretű összeadót készíthetünk.

Az összeadót a következő oldalon rajzoljuk fel.

1 digites BCD összeadó:



Szorzás:

Kisiskolás korában úgy tanítják az embernek a szorzást, hogy 1 jegyű számokat szorozgasson össze, majd a megfelelő helyiértékekre ügyelve adja össze a számokat.
Ez a módszer binárisan is működik, sőt, sokkal egyszerűbb is, mivel míg decimálisan 10 értéket kellett tudnunk összeszorozni egymással, most csak 0 vagy 1 értékkel találkozhatunk.
A bináris szorzás műveletnek megfelelően működő áramkört ismerünk, eddig ezt AND kapunak neveztük.

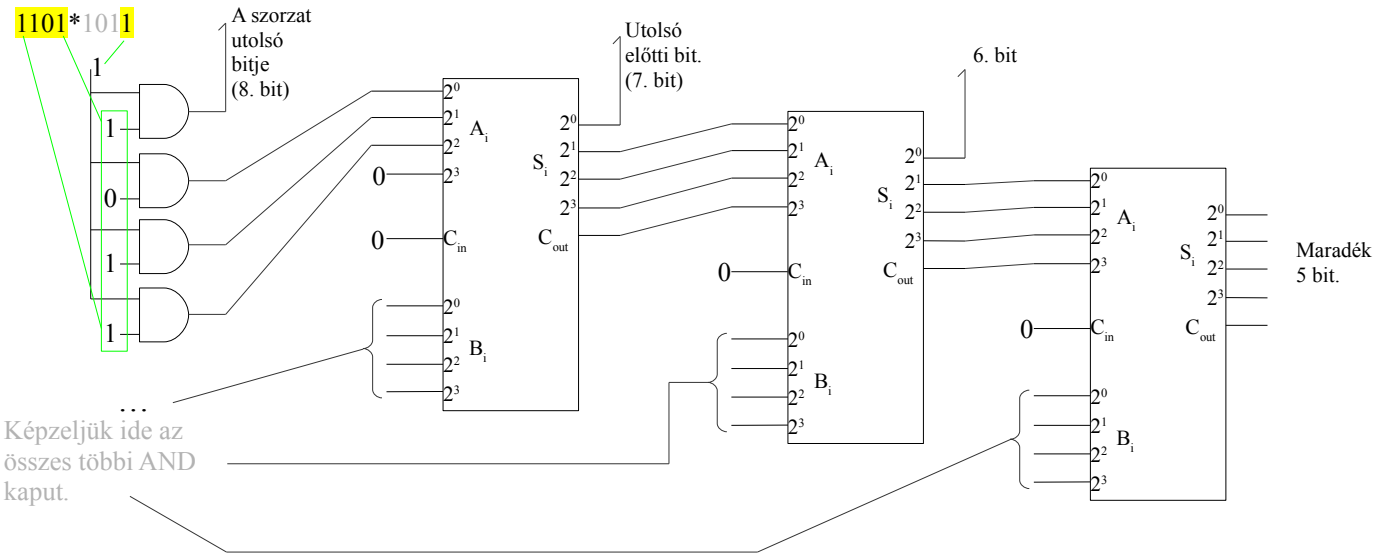
Az 1 bites szorzó áramkör tehát egy AND kapu.
Több bit esetén minden egyes szorzatot 1-1 AND kapu állít elő a megfelelő helyiértékekből.

Nézzük a szorzás eredményét egy példán keresztül:

				1	1	0	1	*	1	0	1	1
			0	1	1	0	1				
		C	+ 1	1	0	1	0				
	C	+ 0	0	0	0		0				
+	+ 1	1	0	1			0				
1	0	0	0	1	1	1	1				

Az eredmény legalacsonyabb helyiértékét minden összeadó nélkül meg tudjuk mondani, ennek az értéke $X_0 * Y_0$.

Majd összeadhatjuk egy 4 bites összeadóval a felső két sort az utolsó bittől eltekintve. Az eredmény 5 biten keletkezik.
Az így kapott eredményt összeadhatjuk a következő 4 bittel egy újabb 4 bites összeadóval (az 5. bit a C_{in}).
Ezt az eredményt ismét összeadhatjuk egy 4 bites összeadó segítségével a legalsó sorral. A legkisebb helyiértéket sose kell továbbkötnünk, hiszen ehhez már csak 0-kat adhatnánk hozzá, így ezt mindig írjuk ki.
Az utolsó bit a 3. összeadó C_{out}-ja.

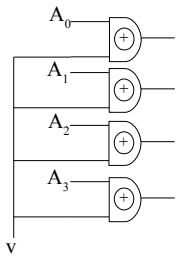


Rengeteg AND kapu (16 darab) és 3 összeadó segítségével megvalósítottunk egy 4 bites szorzó áramkört.
Figyeljünk mindig a helyiértékekre!

Egy regiszterrel sorrendi hálózatként is megvalósíthatunk szorzó áramkört.
Egy 4 bites összeadó így 3 órajelperiódus alatt képes az eredményt megmutatni, amit egy regiszterben tárolhatunk is.

Vezérelhető, bitenként negáló áramkör:

Aritmetikai műveletekhez hasznos és szükséges lesz nekünk egy vezérelhető bitenkénti negáló áramkör.
Már korábban is láttuk, hogy erre a feladatra n darab, 2 bemenetű XOR kapu



$v = 0 : A \rightarrow A$

$v = 1 : A \rightarrow \overline{A}$

MP
Készítsünk számlálót, amely az 1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3 … tartományban ciklikusan számlál.
Ehhez adott egy 74163-as számláló, mely 4 bites, bináris és szinkron /CL bemenettel rendelkezik.

Nézzük a működést:
1-ről kell indulnunk, semmi probléma nincsen egészen 5-ig, ahol muszáj beavatkoznunk.
Amikor visszafele akarunk számolni, akkor a kimeneteket negálnunk kell bitenként, amire a megfelelő áramkört már ismerjük.
A feladat megoldása során egyszerűbb dolgunk lesz azért, mert az eredmény 3 biten keletkezik, amit mi ki is használunk. Amennyiben az eredményt 4 biten kérnék (mondjuk mert BCD értéket szeretnének 7 szegmens kijelzőn kiírni), akkor a legnagyobb helyiérték helyére fix 0-t köthetnénk.

Nézzük mit kell megmutatnunk visszafele számláláskor, és mi a hozzájuk tartozó érték a negálás előtt:
Elszámoltunk 5-ig, most a 4-es érték jön:

Negálás előtt és után:
-100 → -011

A ' - ' a dont care értéket jelöli.

Majd a 3-as érték, 2-es érték:
-011 → -100
-010 → -101

Láthatjuk, hogy a számlálónak belül a 3-as vagy a 11-es értéktől kell számlálnia felfelé, és nekünk ezt kell negálnunk a kimeneten bitenként.
Használju ki, hogy nem használjuk az utolsó bitet, és fixáljuk a dont care értékeket 1-re.
Így a 11-es értéket tölthetjük be a számlálóba, ami utána 13-ig számlál, de mi a kimenet legmagasabb helyiértékét nem használjuk fel, így a felhaszáló negálás után a 4, 3, 2 értéket érzékeli a kimeneten.
Ekkor újra be kell avatkoznunk, az 1 értéket kell betöltenünk a számlálóba és a negálást el kell távolítanunk a kimenetről.

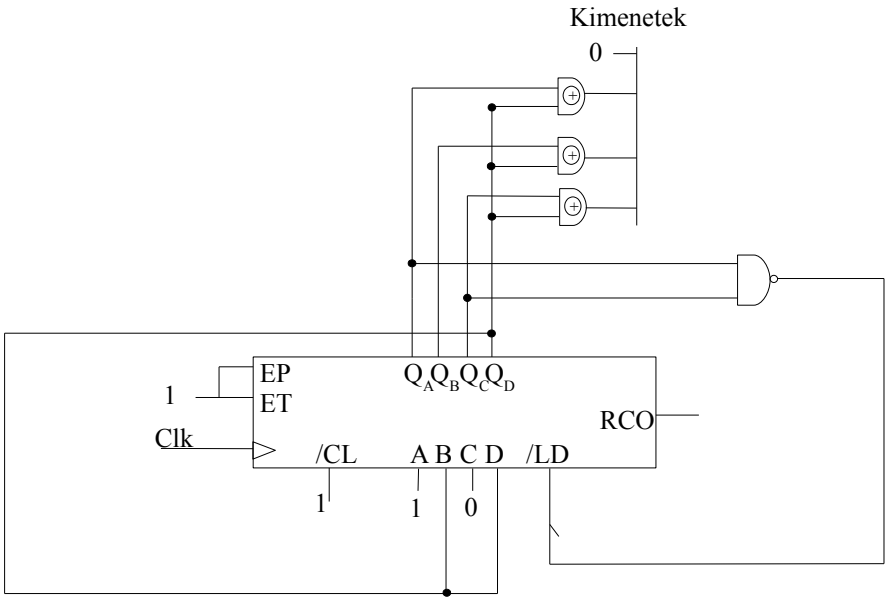
A komparátor jelen esetben egy NAND kapu mely a 2. és 4. helyiértéket figyeli, ugyanis mindkét esetben itt kell beavatkoznunk (5: 0101 és 13: 1101). A NAND kapu kimenete vezérli a /LD áramkört.

A load bemenetekre mit is kell kötnünk:
Amikor az 5-höz érünk az 1011 értéket kell betöltenünk, amikor a 13-at értjük el, akkor pedig a 0001 értéket.
A második bit mindig 0, az utolsó bit mindig 1.

Melyik az az érték az áramkörön belül, aminek 5 esetén 1, 13 esetén 0 az értéke.
Célszerű volna a legmagasabb kimenetet felhasználnunk. Azonban az pont fordított értékeket vesz fel, így azt negálnunk kell.

Tehát a Q_D (ne felejtsük, hogy a D a legmagasabb helyiérték) negáltja megfelel az első és a 3. helyiértékre.
Az inverterhez vezérlőjelnek Q_D értékét választhatjuk.

A rajzot a következő oldalon találjuk.



Megj.: digitális technika 1-ből még a vezetékek helyett haszálhattunk jelöléseket. Ha például volt egy X kimenetünk, akkor bárhova írhattuk, hogy X, /X.
De digitális technika 2-ből már a konkrét megvalósítás is számít, így, ha nem áll rendelkezésünkre /X, akkor le kell rajzolnunk egy invertert, vagy egy NAND kaput, vagy bármi mást, és annak kimenetére ráírni, hogy /X. Ekkor már használhatjuk ezt a jelölést is.

MP:
Tervezzünk hálózatot, mely egy 4 bites pozitív számot megszoroz 5-tel, és az eredményt 8 bites kettes komplemensben állítja elő.

Nézzük először a kettes komplemenst:
Mivel 4 bites pozitív számról van szó, ezért az eredmény is biztosan pozitív és biztosan elfér 7 biten, így tehát az első bitet 0-ra fixálhatjuk a 8-ból, és a kettes komplemenssel nem is kell többet törődnünk.

Hogy szorozzunk egy számot 5-tel:
Kettő hatványaival nagyon egyszerűen tudunk szorozni egy számot. Ha például 4-gyel szeretnénk szorozni véletlenül, akkor annyi a dolgunk, hogy a bináris szám mögé írunk kettő darab 0-t.
Pl.:1111 * 4 = 111100.
Ha 8-cal szeretnénk szorozni, 3 nullát kell mögé írnunk, és így tovább.

Néggyel tehát tudunk szorozni, de mi 5-tel szeretnénk szorozni. Könnyen rá lehet jönni a következőre:
$X * 5 = X * (4+1) = 4X + X$

Írjuk fel tehát a 4 bites számunk 5-tel szorzását mint egy 4-gyel való szorzás (2 bittel shiftelés), és egy összeadás (4 bites összeadóink vannak, azokat gond nélkül használhatjuk.)

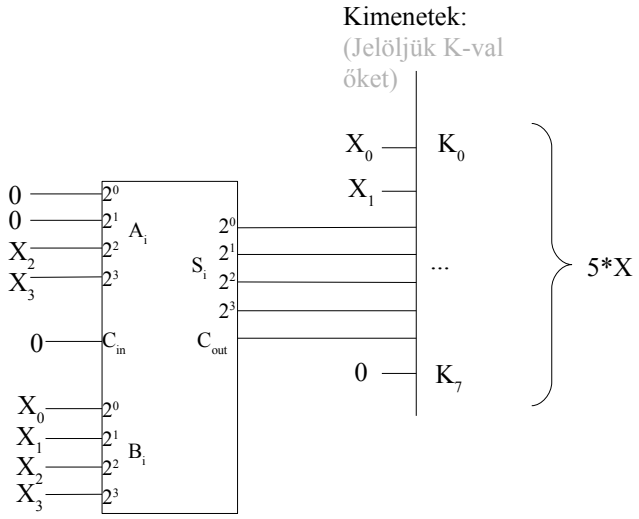
Favágó módszerrel Azt mondhatnánk, hogy Kaszkádosítsunk két darab 4 bites összeadót, és adjuk össze 4 bites és a shiftelés után előálló 6 bites számot, mint 8 bites számok (úgy, hogy eléjük megfelelő számú 0-t írunk).

De inkább gondolkodjunk, és nézzük meg az összeadást:

	0	0	0	0	X_3	X_2	X_1	X_0
+	0	0	X_3	X_2	X_1	X_0	0	0
	0	C_{out}	S_3	S_2	S_1	S_0	X_1	X_0

Láthatjuk, hogy az utolsó két helyiérték összeadó nélkül is előáll. Ne pazaroljunk.
A következő 4 bithez vegyünk egy összeadót. Két darab 4 bites szám összege 5 biten állhat elő, így a carry kimenetét is használnunk kell az összeadónak. De ez nem jelent problémát.

A megrajzolt hálózat:

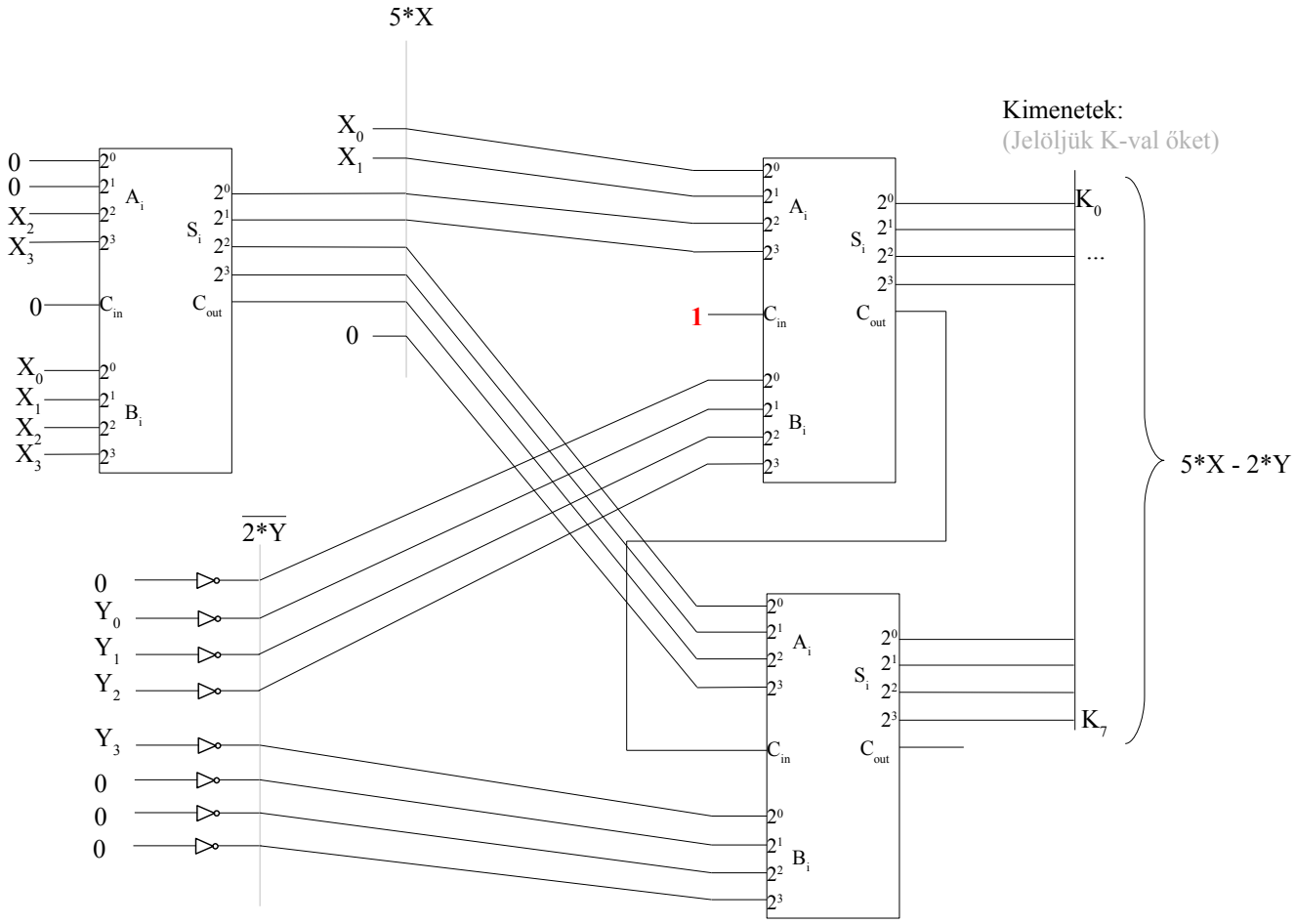


MP:
Tervezzünk hálózatot, ami bemenetként két darab 4 bites számot, X-et és Y-t kapja, és kimenetén 8 biten kettes komplemensben megjelenik az 5X – 2Y érték.

Az 5X-et az előző feladatban már előállítottuk.
2Y-t úgy állíthatjuk elő, hogy 1 bittel shifteljük.

Használjuk ki, hogy 5X – 2Y = 5X + (-2Y).
A -2Y értéket úgy állíthatjuk elő, hogy a 2Y-t bitenként negáljuk, majd hozzáadunk 1 bitet.
Ekkor kapunk kettes komplemenst, amely ábrázolásnál működnek a matematikai műveletek.
A +1 bitet az összeadó C_{in} bemenetére kössük

Megkaptuk tehát a 2 darab legfeljebb 8 bites számunkat, melyet 2 darab, kaszkádosított 4 bites összeadóval összeadhatunk, így megkaphatjuk az eredményt.



MP:
Valósítsunk meg olyan hálózatot, amely két darab **6 bites**, bináris számot kap, és amelynek kimenete a következő, 8 biten, kettes komplemensben:

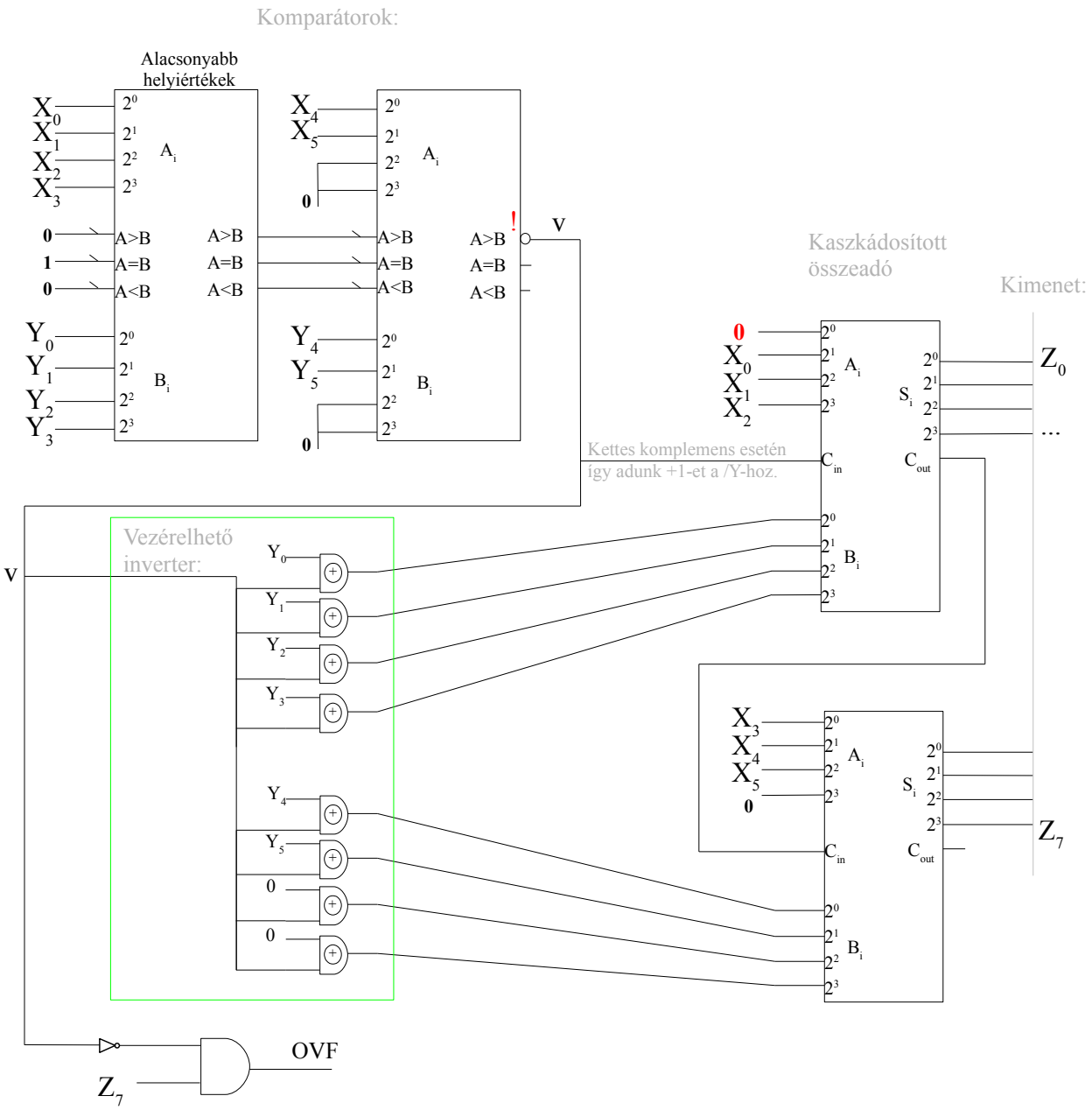
Z= 2X + Y ha X>Y
Z=2Y+X ha X=Y
Z=2X-Y ha X<Y

Először is észre kell vennünk, hogy a második egyenlet teljesen felesleges. Ha X=Y fennáll, akkor a 2X+Y = 2Y+X, így elég csak a 2X+Y-t megvalósítani.

A megfelelő egyenlet kiválasztásához egészen biztos, hogy össze kell hasonlítanunk a két számot. Mivel 6 bites számokról van szó, ezért két darab kaszkádosított, 4 bites komparátort használunk.
A két legnagyobb, nem használt helyiértékre páronként ugyan olyan értéket kell kötnünk, hogy az ne befolyásolja a komparátor működését. Ekkor a vezérlőjelünk az A<B kimenet lesz, ami **a valóságban negált logikájú!**

A 2X-et 1 bit shifteléssel állítjuk elő, míg az Y helyiértékeket (mind a 8-at!) egy vezérelhető inverteren keresztül kötjük a kaszkádosított összeadókra.

Állítsuk elő az overflow függvényt is.
Túlcsordulás ott lehet, ahol Y>X, de a kimenet előjele mégis + (pozitív irányba nem csordulhatunk túl, azzal nem kell törődnünk). Ezt pedig a következő függvény írja le: OVF = /v * z_7

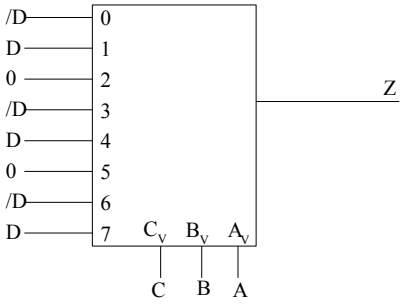


MP:
Valósítsunk meg olyan hálózatot multiplexer segítségével, amelynek kimenete 1, ha a bemenetén lévő 4 bites szám osztható 3-mal maradék nélkül.

Egy bemeneti kombináció pont akkor osztható 3-mal, ha a hozzá tartozó minterm index osztható 3-mal.
Határozzuk meg tehát a Z függvényhez tartozó minterm indexeket: 0, 3, 6, 9, 12, 15 és 4 biten nincs több.

Ez egy nagyon rosszul egyszerűsíthető függvény.
Az eddigi módszerrel történő megvalósításhoz 5 darab IC-re volna szükségünk legalább, ami sok helyet foglal.

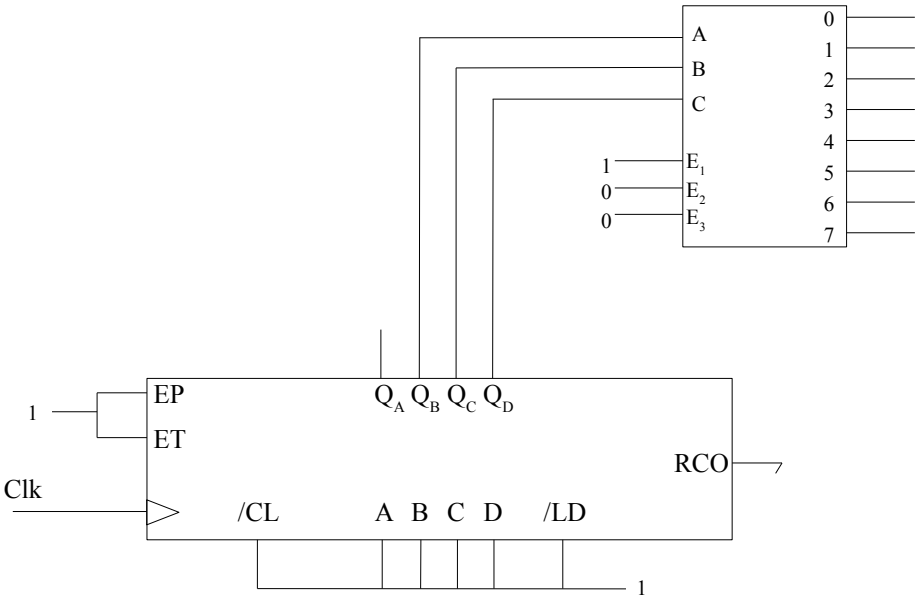
Egyetlen multiplexerrel is megoldhatjuk azonban a dolgot. Ennek menetét már korábban megtárgyaltuk.
Működik intuitívan is.
A 3 kisebb helyiértékű bittel vezéreljük mondjuk a multiplexert és nézzük végig egyesével az összes lehetséges értéküket:
Pl.: 000 vezérlőjelhez tartozó bemenethez az 1000 vagy a 0000 érték tartozhat. Ebből nekünk csak a 0000 jó, mivel az 1000 nem osztható 3-mal. Így ha a 3 vezérlő bit a C, B, A, akkor az első bemenetre nekünk /D-t kell kötnünk.
Pl.: 001 vezérlőjelhez a 0001 és az 1001 érték tartozhat. Előbbi nem, de utóbbi osztható 3-mal, így a második bemenetre a D kerül.



Az ilyen megoldás garantáltan nem hazárdmentes!

MP:
4 bites ciklikusan számláló számlálóval és egy dekoder/demultiplexerrel vezéreljünk egy áramkört (például LED-eket) úgy, hogy minden második számlálásra változzon a kimenet.

A megoldás rendkívül egyszerű. A páratlan vagy páros számokon például pont így számoltunk. A legkisebb helyiértékű kimenetet nem kell a dekóder/demultplexerre kötni, és pont a kívánt működést érjük el.



Memória áramkörök:

Regiszter tömböket memóriaelemként használhatunk. A továbbiakban ezek felhasználását tárgyaljuk részletesen.

Vegyünk a példa kedvéért egy 1kbit-es memóriamodult.
1kbit 1024 bitnyi adatot jelent számunkra. Ha minden egyes bitet egyesével szeretnénk elérni, akkor 1024-féle címet kell tudnunk előállítani, amihez 10 darab **címvezeték** szükséges.

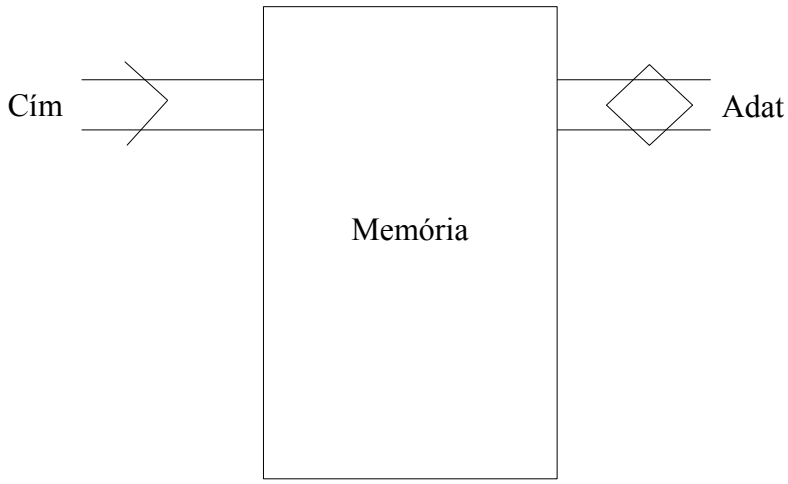
A memória modulunk **bemenetén** egy **címet kap**, melynek ismeretében a **kimenetén** az adott **címen lévő adat jelenik meg.**

Sok esetben nincs szükség arra (sőt, nem is jó feltétlenül), hogy minden bitet egyesével címezhessünk, ezért általában a memórimodulban blokkoat alakítunk ki. Egy ilyen blokkhoz 1 darab cím tartozik, azonban a blokkban nem 1 bitnyi, hanem a blokk méretétől függő adat tárolódik.

MP: Tekintsünk egy 16kbit-es memóriát, melyben 8 bites blokkokba van szervezve az adat.

16kbit esetén 2^{14} bit áll rendelkezésünkre. Ha ezeket egyesével szeretnénk megcímezni, akkor erre 14 darab címvezetékre volna szükségünk. Azonban 8 bitenként érhetjük el az adatot, így viszont $2^{14} / 8 = 2^{11}$ cím és ennek megfelelően 11 címvezeték szükséges mindössze.

A címvezetékek számozása 0-val indul!
11 címvezeték esetén tehát a legmagasabb indexű vezeték a 10-es.



Memóriák típus szerinti osztályozása:

ROM (read only memory):

- Csak olvasható memória.

MPROM (mask programable ROM):

- A memória tartalma a gyártás előtt eldől, a gyártás során „beég” a memóriába, onnantól kezdve állandó.
- Hasznos, ha nagy példányszámban van szükségünk azonos adatot tartalmazó memóriamodulokra.

PROM (programable ROM):

- Egyetlen alkalommal programozható memóriatípus.

EPROM (erasable programable ROM):

- **UVEPROM:**
  - A chipen egy apró ablakon keresztül, UV fénnyel törölhető EPROM.
  - Napon felejtve, az adatokat elveszítheti például.
- **OTPEPROM** (one time programabble EPROM):
  - Egyszerű UVEPROM, melyre nem került a törlés célját szolgáló ablak.
  - PROM megfelelője.
  - Ablak kialakítása a chipen nem olcsó.
- **EEPROM** (E^2PROM, electrically erasable PROM):
  - Elektromos feszültséggel törölhető EPROM.
  - Számítógépek alaplapján, BIOS chipként megjelentek, azonban a vírusok is hozzájuk fértek, így a BIOS tartalma törölhető lett, ami biztonsági kockázat volt.

EPROM-ok hozzáférési ideje jóval kisebb, mint az írási idejük, tehát az olvasás jóval gyorsabb művelet, mint az írás.

RAM (random acces memory):

- Írható és olvasható memóriatípus.
- Bekapcsoláskor tartalma nem definit, kikapcsoláskor pedig az adat elveszik.

SRAM (static RAM):

- Tárolásra flip-flopokat használ.
- Valódi regisztertömb.
- A flip-flop sok alkatrészből állnak, így növelik a memóriamodul méretét és drágítják az előállítást.

DRAM (dynamic RAM):

- A flip-flopok helyett kondenzátorok tárolják az adatokat, ennek megfelelően kisebb, és olcsóbban előállíthatóak az ilyen modulok.
- Problémát jelenthet, hogy a szivárgási áram lassan kisüti a kondenzátort, így elveszíthetné az adatokat. Ennek megfelelően periodikusan frissíteni kell a memória tartalmát, mely egy olvasási ciklussal valósítható meg, mely egyben újra feltölti a kondenzátorokat.

Memóriák kezelése:

Egy memóriamodul kezeléséhez szükségünk lesz a következő be és kimenetekre, vezetékekre:

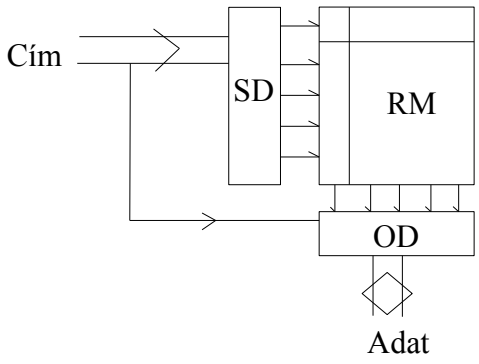
- Címvezetékhalmaz segítségével mondhatjuk meg a memóriának, hogy mely címmel szeretnénk műveletet végezni
- Egy vezérlő bemenet segítségével határozhatjuk meg a végrehajtandó műveletet.
  Ez a vezérlés határozza meg, hogy a memóriát írni vagy olvasni fogjuk.
- Egy engedélyező bemenet is található a memórián, mely kaszkádosításnál játszik szerepet.

A memória olvasása a következő módon nézhetne ki:

Cím → Dekóder → Regisztertömb → Kimeneten megjelenik az adat

Ilyen elrendezés esetén viszont minden egyes blokkot külön kell elérnünk, így n címvezetékhez 2^n vezeték volna szükséges a memóriamodulon belül!
Ezt a működést újra kell gondolnunk, hogy kevesebb vezetékkel is megvalósíthassuk a modulunkat.

A gyakorlatban a következő elrendezést használjuk:



SD: sordekóder
OD: oszlopdekóder
RM: regisztermátrix

Ebben az elrendezésben a cím két információt kódol.
A cím egy része egy egész sornyi regisztert engedélyez egyszerre, míg a cím másik részének függvényében az oszlopdekóder, ami valójában egy multiplexer, kiválasztja azt az oszlopot, amire nekünk szükségünk van.

Így n címvezeték esetén 2n vezeték szükséges a memóriamodulunkon belül.

Vezérlő bemenetek:

/CE (chip enable): a dekóderek engedélyezésére szolgáló bemenet. A memóriamodul működését aktiválja / passziválja.

/OE (output enable): a memóriamodulok általában three state kimenetekkel rendelkeznek. Az /OE bemenet ezek engedélyezésére szolgál.

R / /W (read or /write): a végrehajtandó műveletet vezérli. Az alapértelmezett művelet az olvasás.

Amennyiben a második, sor és oszlopdekódert tartalmazó szervezést használjuk, a memóriamodulunk gyorsítható, ha feltételezzük, hogy nincs szükségünk egyszerre a sor és oszlopdekóderekre.
A valóságban a programok struktúrája miatt azt is feltételezhetjük, hogy a következő művelet is ugyan abban a sorban, csak másik oszlopban lesz, és ezt kihasználva tovább gyorsíthatjuk a memória működését.

A cím a következőkből adódik össze:
RAS (row address strobe)**: sor címzése**
CAS (column address strobe)**: oszlop címzése**

Memória olvasási ciklusa:

A memória olvasásánál a következő sorrendben történnek a műveletek:

- Cím vezetékeken megjelenik az aktuális cím.
- /CE bemeneten megjelenik az engedély a chip működéséhez.
- /OE bemeneten megjelenik az engedély a kimenetek működéséhez.
- Leolvashatjuk a kimenetet.

A következő időzítésekkel kell foglalkoznunk írás esetén:

$\mathbf{t_{CE}}$: Meghatározza, hogy a chip engedélyezése után mennyi idő múlva áll rendelkezésre az adat a kimeneten.
$\mathbf{t_{OE}}$: Meghatározza, hogy a kimenet engedélyezése után mennyi idő múlva áll rendelkezésre az adat a kimeneten
$\mathbf{t_{ACC}}$ (acces time): Meghatározza, hogy a cím megérkezése után mennyi idő múlva áll rendelkezésre az adat a kimeneten. A memória sebességét ez utóbbi paraméter határozza meg.

$\mathbf{t_{OH}}$: Meghatározza, hogy az engedélyező jelek eltűnése után mennyi idő alatt tűnik el az adat a kimenetről.
Nem ritkán 0 értéket adnak meg a memóriák specifikációjában.
$\mathbf{t_{DF}}$: Meghatározza, hogy az engedélyező jelek eltűnése után mennyi idő alatt áll be valóban magas impedanciás állapotba a kimenet.

Memória írási ciklusa:

Két esetet kell megkülönböztetnünk. Ha rendelkezünk /OE bemenettel és ha nem rendelkezünk /OE bemenettel.

/OE bemenettel:
$\mathbf{t_{WC}}$: Meghatározott a cím jelének minimális időtartama.
$\mathbf{t_{WCE}}$: Meghatározott, hogy a chip enable jelnek milyen széles impulzusnak kell lennie.
$\mathbf{t_{R/\overline{W}}}$: Meghatározott, hogy a R/ /W impulzusnak milyen szélesnek kell lennie.
$\mathbf{t_{DS}}$: Meghatározott, hogy mennyi ideig kell az adatot a memóriába írni.
$\mathbf{t_{DH}}$: Meghatározott, hogy a write jel eltűnése után mennyi ideig kell még az adatot tartani a bemeneteken.
t_{DH} értéke gyakran 0 a katalógusban.

/OE bemenet nélkül:

/OE bemenet nélkül a modul a R/ /W megjelenéséig azt hiszi, hogy olvasási ciklus kezdődik. A R/ /W jel megjelenése után azonban bizonyos idő szükséges az írás megkezdése előtt, hogy a chip felkészülhessen erre.
$\mathbf{t_{ODW}}$: Meghatározott, hogy a R/ /W jel megjelenése után mennyi idővel kezdődhet meg a memória írása.

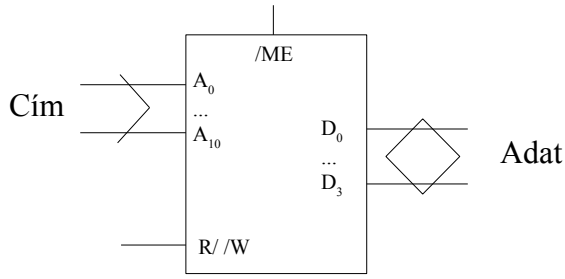
Memória mérete:

Meghatározhatjuk a memória méretét a címvezetékeinek és az adatvezetékeinek száma alapján.
Az n darab címvezeték összesen 2^n blokk megcímezhetőségét teszi lehetővé.
Az m darab adatvezeték esetén egy címen m bithez férünk hozzá.
Tehát n címvezeték és m adatvezeték esetén a memória mérete $m*2^n$ bit.

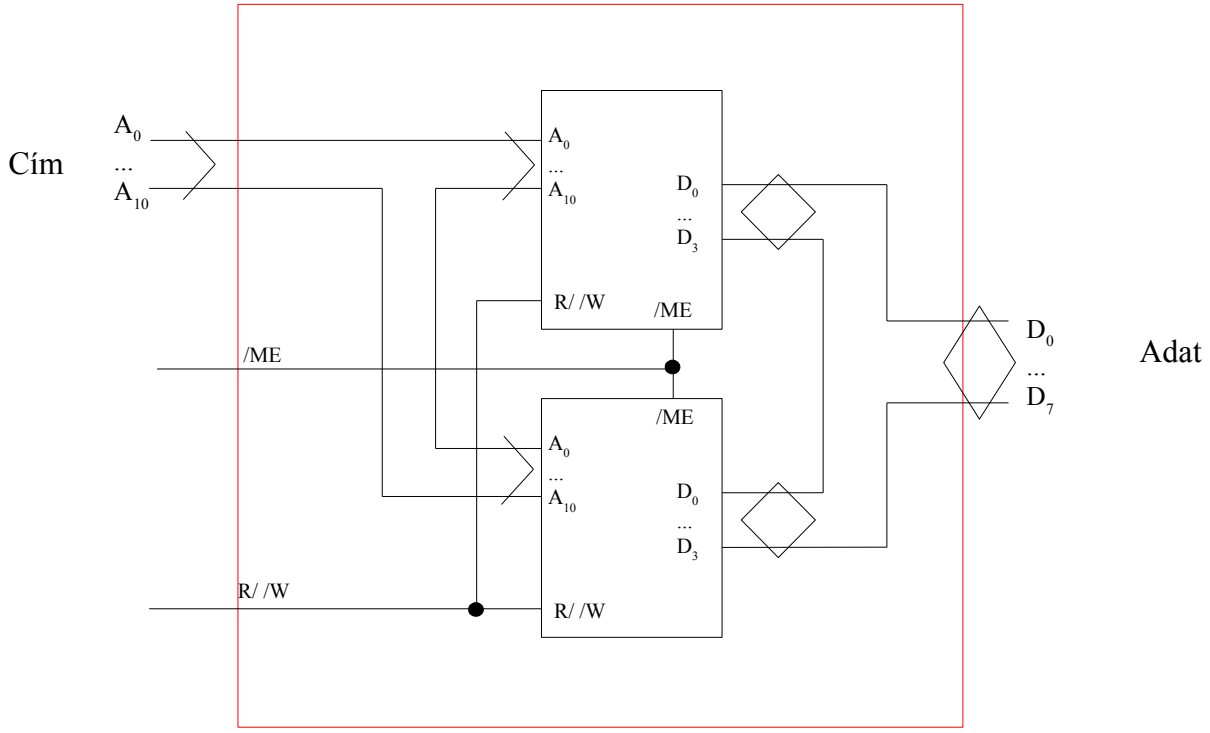
Memóriamodulok kaszkádosítása:

Bitszélesítés:
A feladat, hogy a 2k * 4bit-es memóriamoduljainkból 2k * 8bit-es memóriát állítsunk elő.

Hogy is néz ki 1 modulunk:
2k*4bit esetén 11 címvezeték szükséges (2k = 2048 = 2^{11}). Ezeket A_0 … A_{10}-zel jelöljük.
4 (4 bit) adatvezetékünk van, ezeket D_0 … D_3-mal jelöljük.
Ezen kívül egy R/ /W és /ME (/ME: memori enable) engedélyező bemenetek állnak rendelkezésünkre.



Ilyen modulokból szeretnénk olyan memóriát gyártani, aminek 8 adatvezetéke van, mivel 8 bithez szeretnénk egyszerre hozzáférni, de ugyan úgy 11 címvezetékkel címezhető, mivel a 2k-s felépítést nem akarjuk módosítani.
A kaszkádosítás magától értetődően adódik. Mindkét memóriaelem ugyan azt a címet kapja, ugyan azt az engedélyezést kapja, és a kimeneteiket egymás mellé téve, egy címen 2*4=8 bithez férünk hozzá egyszerre.



Vertikális kaszkádosítás:
A feladat, hogy a 2k*8-bites memóriamoduljainkból 8k*8bit-es memóriamodulokat kapjunk.
4 darab memóriamodulra lesz szükségünk (mivel a kapacitást meg akarjuk négyszerezni).
8k*8bit-es memóriamodulhoz azonban 13 címvezetékre van szükség, itt már közel sem triviális a kaszkádosítás.

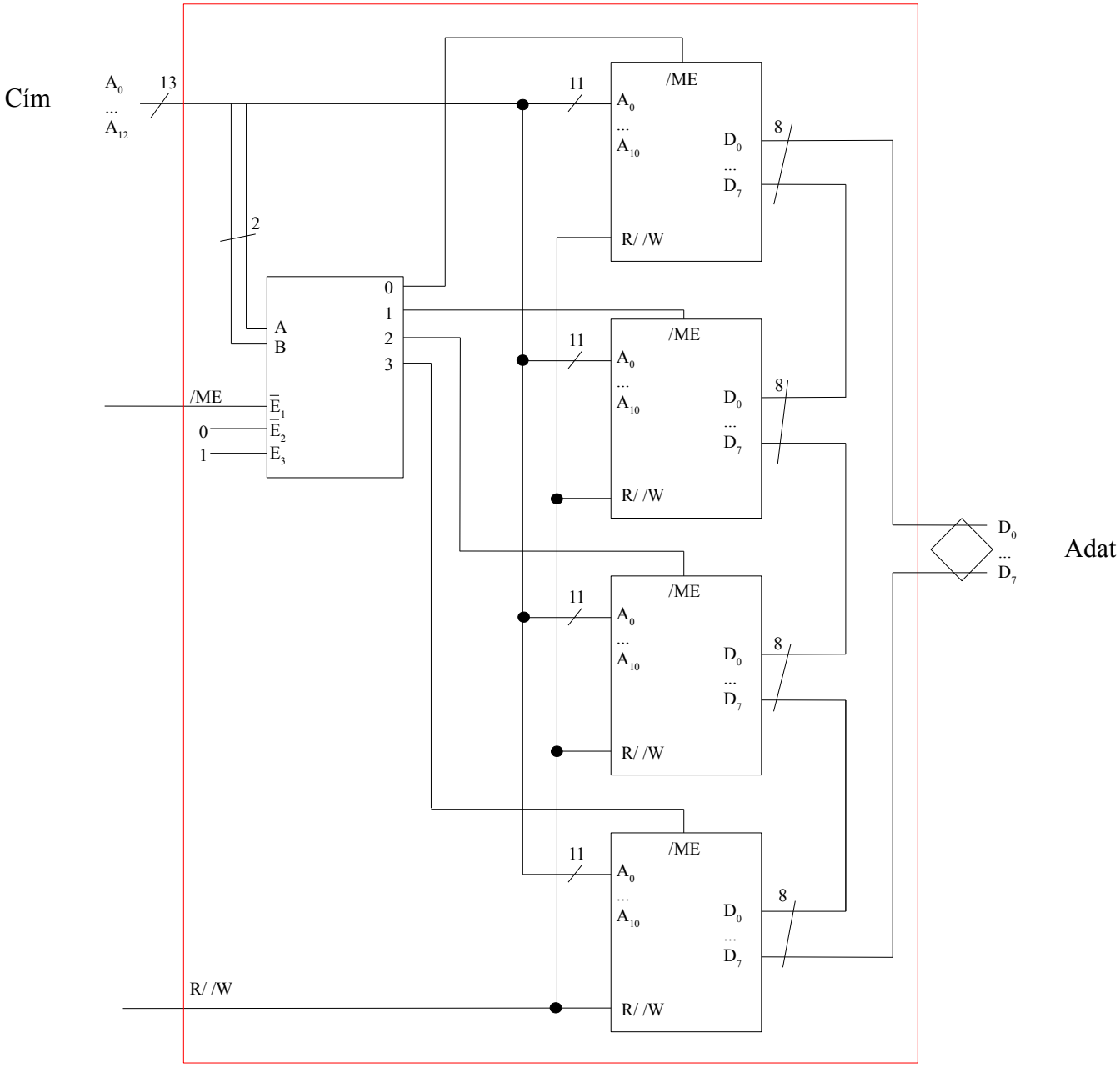
Engedélyező jelünk van, tehát feltételezhető, hogy a memóriák three state kimenettel rendelkeznek. Ekkor viszont összeköthetjük a kimeneteiket, csak a megfelelő engedélyező jelet kell előállítanunk.

A R/ /W jelet mindegyik modul kapja együtt, mivel ezen nem kell módosítanunk.

A memória belsejének szervezése több módon is lehetséges.
RAM esetén ez a belső felépítés lényegtelen, mivel csak beírás után van értelme a memória tartalmának.
ROM esetén azonban szükséges, hogy tudjuk, milyen belső felépítésű memóriával van dolgunk, mivel az adott chip már tartalmazza az adatokat, és az meghatározott, hogy azt mi milyen címen érhetjük el!

A kaszkádosítás megvalósítására egy dekóder-demultiplexerre van szükségünk, ami 4/1 kódot állít elő, így választva ki a 4 felhasznált modul közül a megfelelőt. A 4/1 kódot két címvezeték alapján állítjuk elő.
A maradék 11 címvezetéket minden memória együtt kapja meg.

A gyakorlatban a dekóder/demultiplexer áramkör negált logikájú, így egy az egyben köthető a /ME bemenetre.
A felhasználó számára látható /ME bemenetet a dekóder-demultiplexer engedélyező bemenetére kell kötnünk.



Összefoglalva tehát:

Ha a bitszélességet szeretnénk növelni, párhuzamosan kötjük a memóriákat.
Ha a címezhetősget akarjuk bővíteni, akkor dekóder-demultiplexerre van szükségünk, ami két memória esetén természetesen egy inverter is lehet, nem szükséges felesleges alkatrészeket felhasználni.

Megj.: vezetéket áthúzva, és felé számot írva jelöljük hogy adott vezeték egy „vezetékköteget” jelöl, amely kötegnek a felé írt számnak megfelelő vezeték felel meg.

Számítógép felépítése a Neumann-model szerint:

A műveletek elvégzésére definiálnunk kell egy aritmetikai logikai egységet (**ALU**).
A műveleteknek operandusaik vannak, amely operandusokat nekünk tárolnunk kell, erre szolgál a **memória**.
A műveleteket vezérelni kell, amely célra egy **vezérlő hálózatot** kell definiálnunk, amely képes megmondani az aritmetikai logikai egységnek, hogy milyen opreandusokkal milyen műveletet végezzen.
A memóriához a vezérlőegységnek és az ALU-nak is hozzá kell férni, hiszen a vezérlőegység számára is tárolnunk kell az utasításokat. Ha a memória tartalmát a vezérlőegység értelmezi, akkor az egy **utasítás**, ha ALU, akkor pedig **adat**.

A számítógépünk már el tud lenni magában, azonban szeretnénk, ha a külvilággal is tudna kommunikálni, ugyanis sok értelme nincs a számításnak, ha annak eredménye sose kerül ki a számítógépből, és ha nem mi adjuk meg, hogy mit számoljon, akkor valószínűleg nem azt az eredményt fogjuk kapni, amit szeretnénk.
Szükségünk van tehát olyan eszközökre (úgy nevezett **perifériákra**), amelyek az eredményt képesek közölni velünk, és olyanok is kellenek, amelyek segítségével adatokat juttathatunk be a számítógépünkbe.

Készen van a számítógépünk, azonban a Neumann modellt kiegészíthetjük és csiszolgathatjuk.
Kezeljük az ALU-t és a vezérlőegységet egy egységként, és a memóriából egy piciny szeletet tegyünk mellé, majd az így kapott elemet nevezzük el central processor unit-nak, röviden **CPU**-nak.

Definiáltuk tehát a számunkra szükséges eszközöket. Ezeknek viszont kommunikálniuk kell egymással, hiszen a memóriában lévő adatnak például el kell jutnia az ALU-hoz, vagy a billentyűzetünkön leütött karaktereknek el kell jutni a memóriához.
Az egységek közti kommunikációt kiszolgáló vezetékhalmazt **sínnek**, **vagy** más néven **busznak** nevezzük.
A sínnek tartalmaznia kell azt az információt, hogy éppen melyik eszközzel akarunk dolgozni, és a sín műveleteit vezérelnünk is kell.

Vezérlés szempontjából az eszközöket két nagy csoportra osztjuk:
MASTER: a master eszköz szolgáltatja a sínen a címeket, és a műveletek lebonyolításáért is ez felel.
SLAVE: a slave eszközök csak az adatokkal képesek dolgozni.

Egy időben egy buszon egyetlen master lehet, ami viszont „akárhány” slave eszközt kiszolgálhat.
Léteznek több masteres rendszerek, ezekről is fogunk beszélni, azonban ezek esetében is teljesül, hogy egy időben egyetlen master irányítja a sínt.

Az eszközeinket az adatokkal végzett műveletek szempontjából is osztályozhatjuk:
A **forráseszköz** adatot szolgáltat a sínen a vezérlésnek megfelelően.
A **céleszköz** a sínen lévő adatot feldolgozza vagy eltárolja a vezérlésnek megfelelően.

Egy időben egy buszon egyetlen forrás, és „akárhány” cél előfordulhat.

Az esetek többségében azt állíthatjuk, hogy a master vagy forrása az adatoknak vagy célja, de elképzelhető olyan eset is, hogy a master nem vesz részt az adatátviteleben.
Ilyen a direct memory acces (**DMA**), amikor egy beviteli eszközről közvetlenül a memóriába helyezzük az adatot.

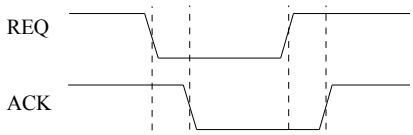
Egyetlen mastert tartalmazó rendszerek esetében haszálhatjuk az úgy nevezett totem pole kimeneteket a címek és a vezérlés biztosítására, azonban több masteres rendszerek esetében ezek a kimenetek alkalmazhatatlanok.
A valóságban általában **three-state kimeneteket** találunk a modulokon.

Az adatátvitel ütemezése:

Tervezhetünk órajellel szinkronizált **szinkron sínt**.
Ekkor az átvitel üteme definit, és amennyiben szükség van rá, az órajel egész számú többszörösével késleltethetjük az eseményeket.

Megvalósíthatunk **aszinkron sínt** is.
Az adatátivtelt ekkor úgynevezett **hand-shake** jelpárokkal valósítjuk meg.
Hand-shake rendszer esetén a következő módon történik a sín kezelése:
A forrás egy request jel (REQ) felfutó élét szolgáltatja egy kimenetén, jelezve ezzel, hogy adatot szeretne továbbítani.
A céleszköz ekkor egy acknowledge (ACK) jel felfutó élével jelzi, hogy képes fogadni az adatokat.
A forrás eszköz erre a REQ jel megszüntetésével (lefutó él) jelzi, hogy megértette ezt, majd végső stádiumban a céleszköz is megszünteti az ACK jelét (lefutó él), jelezve ezzel, hogy a kommunikáció megtörtént.

Természetesen ezen vezérlőjelek negált logikával is működhetnek, sőt, a gyakorlatban, mivel az IC-k belső felépíte általában olyan, hogy a levegőben lógó vezetékeket 1-nek érzékelik, ezért a **vezérlőjelek leggyakrabban negált logikával működnek**.



Késletetések:
Definiálnunk kell úgynevezett **SETUP és HOLD időket**.
A SETUP idő az általunk vizsgált rendszerekben néhányszor 10ns körüli érték. Ez egy adatelőkészítési időt határoz meg, azaz megszabja, hogy az adatnak a vezérlő jel előtt mennyi idővel kell stabilnak lennie.
HOLD idő a gyakorlatban sokszor elhanyagolhatóan alacsony, azaz a vezérlőjel elvétele után a vezetéken már nincs szükség az adatra.

Utasítások felépítése:
A következő pár információra van szükségünk egy művelethez:

- Műveleti kód (op code): mit szeretnénk csinálni?
- Operandusok: mivel szeretnénk csinálni?
- Eredmény címe: hova rakjuk az eredményt?
- Következő utasítás címe: hol a következő utasítás?

Ez a legegyszerűbb esetünk, egy 4 című gép, amelynek 4 címre van szüksége.
Azonban további megfontolásokkal ezt még redukálhatjuk, és redukáljuk is.

A Neumann-model alapján a programok sorban helyezkednek el a memóriában, tehát a következő utasítás a következő memóriacímen helyezkedik el, amivel máris egy címet spórolhatunk, ha definiálunk egy programszámláló egységet, amely a műveleti címet mindig autómatikusan növeli a műveletek után.

Az operandusoknak helye van a memóriában, és az ő címűket ismerjük. A művelet után rájuk már általában nincs szükség, tehát megszabhatjuk, hogy az eredmény az egyik operandu helyén keletkezzen, így el is jutottunk az úgynevezett 2 című géphez.

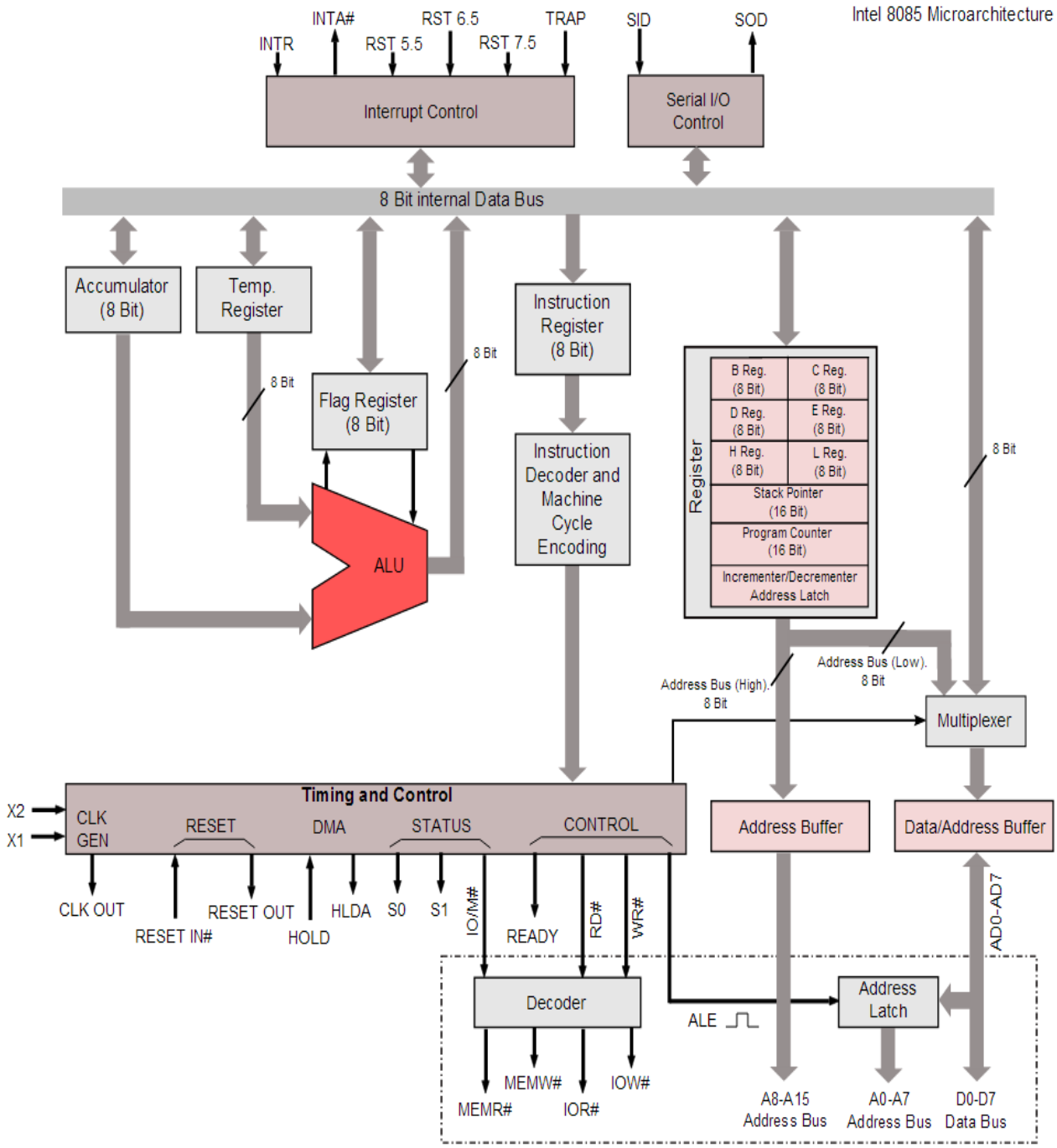
Tovább spórolhatunk, ha az egyik operandus címét kitüntetjük, és egy regiszterben tároljuk. Ezt a regisztert **akkumulátor**nak nevezzük, és ezzel el is jutottunk jelenlegi végállomásunkhoz, az 1 című géphez.

Többféle módon szolgáltathatunk adatokat:

Közvetlen adatok esetén az utasítás az operandust is tartalmazza. MP.: „Az accumulátort szorozd meg 3-mal!”
Indirekt címzés esetén az utasítás az opernadus címét tartalmazza.
Regiszterindirekt címzés esetén az utasítás azt tartalmazza, hogy a precesszor mely címregisztere tartalmazza az operandust.

Többszörös indirekciók és eltolások is elképzelhetők és léteznek is, melyek mind magasabb szintű programozási nyelvek támogatására jöttek létre, például tömbindexeléshez.
Ezeknek számunkra jelenleg nem sok haszna van, így ezeket részletesebben nem is fogjuk tárgyalni.

A 8085-ös processzor és a rá épülő rendszerek:



A 8085-ös mikroprocesszor egy 8 bites aritmetikai logikai egységgel rendelkezik.
Az ALU bemeneteire egy akkumulátor és egy átmeneti regiszter (ATM) csatlakozik. Ez utóbbi szolgál az operandusok ideiglenes tárolására.
A belső adatsínre csatlakozik az AKKU (akkumulátor), az ATM és a FLAG.
A flagek az aritmetikai műveletek eredményéről nyújtanak információt.

A következő flag-eket különböztetjük meg:
Z (zero flag): Jelzi, hogy a művelet eredménye 0 lett-e.
S (sign flag): A műveletek eredményének előjelét jelzi. Mivel 8 bites kettes komplemensben történik az eredmény ábrázolása, ezért ez a flag az eredmény első bitjével egyezik.
CY (carry flag): Jelzi, hogy a művelet eredményeként keletkezett-e átvitel.
Fontos, hogy ez nem overflow funkciót lát el, ez a processzor ilyet még nem tud!
P (parity flag): Az eredményben előforduló egyesek paritását (páros/páratlan darab) jelzi.
AC (segédátvitel flag): BCD műveleteknél jelzi, hogy a 4. és 5. bit közt keletkezett-e átvitel.

Utasításregiszter és **utasításdekóder**:
Egy vezérlőhálózat vezérli az utasításdekódert, ami az utasításregiszterért felel.

Regiszterek: a memóriának azon része, amely a processzorba integrálva található meg.
A 8085-ös 6 darab 8 bites regiszterrel rendelkezik, melyek: B, C, D, E, H, L.
Ezek adatresziterek, bár bizonyos regiszterek címzésnél is használhatóak.

A processzoron belül van egy autoinkremens (saját értékét megfelelő időben növelni képes) **programszámláló (PC)**, ami a következő művelet címét tárolja.

A processzor támogatja a stack adatszerkezetet (LIFO = last in first out), amelyhez van egy különálló **stackmutatója**, **(SP – stack pointer)** amely autómatikusan változtatja az értékét, és mindig az utoljára betett adatra mutat.

Időmultiplexelt cím és adatvezetékek:
Összesen 16 biten valósítottak meg a processzoron belül 16 bites címzést, és 8 bites adatkezelést, melyet úgy nevezett időmultiplexeléssel oldottak meg. Ez azt jelenti, hogy a címvezetékek fele az idő egy részében címvezetékként, míg az idő más részében adatvezetékként működik.

A processzor tartalmaz egy **belső megszakítást vezérlő egységet (IT – interrupt control)** .

Soros I/O portként (Serial I/O) 1 bitet használhatunk fel.

A vezérlőn egy $\mathbf{Reset_{IN}}$ és egy $\mathbf{Reset_{OUT}}$ található. Előbbi aszinkron működik, míg utóbbi az órajellel szinkron módon másolja ki a reset jelet a kimenetére, melyet a külső eszközök használhatnak fel alaphelyzetbe állásra.

A vezérlő a memória áramkörök számára **szeparált /RD és /WR jelet** ad.

A vezérlőn egy **IO/ $\overline{\mathbf{M}}$ kimenet** (továbbiakban gyakran IO/ /M) található, melynek értéke 1, ha IO eszköz címe, és 0, ha memória címe található a címvezetékeken.

Az IO címek is 16 biten állnak elő, de ez feleslegesen sok eszközt jelentene. Összesen 256 különböző cím adható, melyek mindegyike úgy áll elő, hogy az első 8 bit másolata jelenik meg a második 8 biten is. Jegyezzük ezt meg vizsgára!

A processzoron található egy /**RDY (ready) bemenet**, melynek működéséről bővebben fogunk beszélni.

Az **ALE (adress latch enable)** kimenet jelzi, hogy az adat/cím vezetékeken címbitek találhatók.
Ez a jelzés a külső eszközök számára jelentős, ez alapján tudják kezelni az időmultiplexelést.

A $\mathbf{CLK_{out}}$ kimenet szintén a külső eszközök számára van fenntartva.
A processzor 6,144MHz-es órajellel működik, azonban a külvilág számára CLK_{out} kimenetén csak 3,072MHz-es órajel jelenik meg. Egy fázis így 325ns ideig tart közelítőleg.

8085-ös interface:

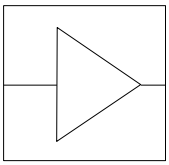
Nézzük mire van szükségünk a sín működtetéséhez:

Szükséges a műveletek azonosításához egy cím, mely meghatározza az adat helyét, amivel dolgozunk.
A sínre jellemző adat, hogy hány címet különböztethetünk meg, amely attól függ, hogy hány címvezetékünk van.
A 8085-ös mikroprocesszor esetében ez 16 darab, ahogy azt már megtárgyaltuk, így összesen 64k különböző memóriacím adható ki.
A címvezetékek száma határozza meg a memória méretét, ha tudjuk, hogy egyszerre hány bitből álló adatokkal dolgozunk.

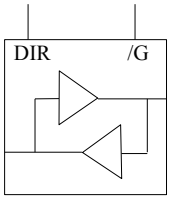
Már megbeszéltük, hogy az adat és a címbusz egy része időmultiplexelve van a 8085-ös CPU esetén, tehát 8 címvezeték az adatvezetékekkel közös, és van 8 címvezeték, ami mindig az aktuális címet hordozza.

Az időmultiplexelést nem a külső eszközöknek kell kezelni, ez felesleges pazarlás volna.
A már említett **ALE** (adress latch enable) kimenet pontosan erre szolgál, jelzi a külső eszközök számára, hogy az aktuális buszciklusban cím van a vezetékeken.
Regiszter helyett azért használunk latchet, mivel a cím már az ALE jel megérkezése előtt elérhető, így a latch a működéséből adódóan jóval gyorsabban dolgozhat.
Ez a latch szolgáltatja a címet a kimenetén.

Egy kimenetet nem terhelhet tetszőleges számú bemenet.
A sínen tehát az adat és a címvezetékekre szükséges egy úgy nevezett buffer áramkör, mely erősítőként működik.
Azokon a vezetékeken, ahol csak címek közlekednek egyszerű, **egy irányú buffer** is megfelel. Azonban az időmultiplexelt vezetékeken **két irányú bufferre** van szükség.
A két irányú buffer rendelkezik egy **DIR** vezérlő bemenettel, mely az irányt határozza meg, és egy /**G** vezérlő bemenettel, mely engedélyezi az erősítőt.



Egy és kétirányú buffer.



3 vezérlőjel áll rendelkezésünkre annak meghatározására, hogy hol hajtsuk végre a műveletet:

IO/ /M: Meghatározza, hogy IO (1) vagy memória (0) a címzett eszköz.
64K IO eszköz helyett IO esetén a 16 bitből az első és a második 8 bit azonos, így csak 256 különböző eszözt címezhetünk IO esetén.
/RD: Olvasás műveletet kell végrehajtanunk.
/WR: Írás műveletet kell végrehajtanunk.

Ahhoz, hogy a sínre több eszköz csatlakozhasson, ezen vezérlőjelekhez is buffert kell alkalmaznunk.

A tökéletes működéshez minden eszköznek elég gyorsnak kell lennie, hogy egy sínciklus alatt a feladatát teljesíteni tudja. Ez azonban nem teljesül minden esetben, ezért a processzornak van egy félig aszinkron **RDY (ready) bemenete**, amely segítségével a külső eszkötök jelezhetik, hogy nem sikerült befejezni a feladatot.

A ready bemenet különböző módokon működhetne, ezeket ready filozófiáknak szokás nevezni:

1. Alapértelmezett 1-es:
   Ebben az esetben a RDY nem aktív. Teljes szinkron működést kapunk, mely a lehető leggyorsabb, azonban minden eszköznek megfelelően gyorsnak kell lennie.

2. Alapértelmezett 1-es, wait kérés lehetőségével:
   A lassabb eszközök egy wait kéréssel jelezhetik, hogy nincsenek készre.
   A wait bemenetre csatlakoztatni kell az összes lassú eszközt. Open collector kimeneteket kell alkalmaznunk, melyek 0-ba húzhatják a RDY bemenetet, amely negált logikával működik természetesen.
   Az első gyorsaságát ötvözi a wait kérés lehetőségével.

   Az első két esetben nem vehetjük észre, hogy olyan eszközt címeztünk, amely nincs a rendszerben!

3. Alapértelmezett 0-s:
   Minden eszköznek jeleznie kell, hogy készen van. Minden eszköznek saját kiegészítő hálózatra van szüksége, amely megvalósítja ezt a jelzést.

   Ha a rendszerben nem létező eszközt címzünk, akkor a rendszer várakozik, mivel senki nem jelez.
   Erre az esetre egy **watch-dog**-nak nevezett hálózat figyel, amely a definiált időtúllépés után jelez.
   8085-ös rendszerben ez összesen 6 wait fázis után történik meg.

Nézzük hogy néz ki a sínciklus:

Az első fázisban mind a 16 vezetéken megjelenik a cím.
Ezzel párhuzamosan, de kevés időelcsúszással az ALE kimeneten jelzi a processzor a latch-nek, hogy a cím stabil és tárolható.
Megjelennek a státuszelőre jelző jelek, az S_0, S_1 és az IO/ /M, melyek előre jelzik, hogy milyen művelet következik.

A második fázis elején jelennek meg a vezérlőjelek, melyek meghatározzák, hogy írás vagy olvasás következik.
A második fázis elején, pontos időzítéssel megtörténik a RDY mintavételezése.
A RDY jel mintavételezése a vezérlőjelek megjelenése előtt/közben történik, tehát ezeket a vezérlőjeleket nem használhatjuk a RDY jel előállítására! A státuszelőrejelző biteket kell használnunk 0 WAIT-es ready esetén!
Az adat a második fázistól elérhető.

Ha egy eszköz sem jelzi, hogy készen van, akkor a processzor autómatikusan wait fázist iktat be, és megismétlődik a második fázis.

A harmadik fázisban lezáródik a művelet, a processzor visszaveszi a vezérlőjeleket, és újra kezdődik az 1. fázis.

A sínciklusról a segédlet 29-30. oldalán részletes idődiagrammokat is találunk.

Az utasítások végrehajtása gépi ciklusokból áll:

Megérkezik az utasítás, melynek végrehajtása 1-5 gépiciklus alatt történik meg.

Gépi ciklusok típusa:
FETCH: A beolvasott adatot a processzor utasításként értelmezi.
MEMORY READ: A gépi ciklusban a processzor memória címről olvas.
MEMORY WRITE: A gépi ciklusban a processzor memória címre ír.
IO READ: A gépi ciklusban a processzor IO eszközről olvas.
IO WRITE: A gépi ciklusban a processzor IO eszközrre ír.
INTA (interrupt acknowledge → IO FETCH):
Speciális gépi ciklus, később részletesen tágyaljuk, külső megszakításra alkalmas
IDLE: A gépi ciklusban a processzor nem használja az adatsínt, de dolgozik.
- 16 bites összeadás (double add = **DAD**) esetén 2 gépi ciklus alatt ad össze a 8 bites ALU.
- **RST / TRAP** : nem tartozik hozzá művelet.

HALT: A processzor áll, és innentől kezdve nem csinál semmit.

Gépi ciklusok megkülönböztetése:

Gépi ciklus: / Vezérlő jel:	IO/ /M	S_1	S_0	/RD	/WR	/INTA
FETCH	0	1	1	0	1	1
MEMORY READ	0	1	0	0	1	1
MEMORY WRITE	0	0	1	1	0	0
IO READ	1	1	0	0	1	1
IO WRITE	1	0	1	1	0	1
INTA	1	1	1	1	1	0
IDLE DAD	0	1	0	1	1	1
IDLE RST/TRAP	1	1	1	1	1	1
HALT	-	0	0	-	-	1

IDLE RST/TRAP esetén nincs vezérlőjel, más eszköz használhatja a sínt.

Minden gépi ciklus minimum 3, de legfeljebb 6 fázis alatt végrehajótdik.

A processzor a következő állapotokkal rendelkezik:

RUN állapot: A vezérlésszámláló minden egyes vezérlésre (műveletre) autómatiksuan lép.
Amíg RDY jelünk van, RUN állapotban maradunk.

WAIT állapot: Ha a RDY bemenet nem kerül 0-ba, wait állapotba kerül a processzor addig, amíg RDY jel nem jön.

HALT állapot: Halt állapotba vezethetjük a processzort HALT jellel, ebből RUN állapotba RESET vagy IT (külső megszakítás) jellel kerülhet.

Ez a három állapot szükséges akkor, ha a sínen egy időpillanatban 1 master lehet, azonban a több masteres rendszerekre is alkalmas a processzor:

HOLD állapot: Egy hold jel vezeti ide a processzort, ami ebben az állapotban a sín vezérlését más masternek engedi át.
HOLD állapotba HALT vagy RUN állapotból kerülhetünk.

Egy sín ciklus a következő módon néz ki:

T_1 állapottal kezdődik a ciklus. Amennyiben RUN állapotban van a processzor, autómatikusan T_2 állapotba kerül a rendszer.
Ha IDLE állapotba kerül a processzor, vagy nem kap RDY-t, T_{WAIT} állapotba kerül a rendszer, ahonnan a RDY megérkezése után T_2 állapotba kerül.

Ezután a CPU megvizsgálja, hogy más master nem kéri-e a sín vezérlését.
Ha más master jelez, akkor egy úgynevezett HLDAFF flip-flopot 1-es állapotba vezérel a processzor, hogy a sínciklus végén majd átadhassa a vezérlést.
Ezután mindenképpen T_3 állapot következik.
Fetch típusú gépi ciklus esetén a rendszer T_4 állapotba kerül, ahol megtörténik az udasítás dekódolása, és ha lehetséges, az utasítás végre is hajtódik. Csak 3, 4 vagy 6 köztes állapot lehetséges!

Ezután ha 6 darab köztes állapotra van szükség, akkor a CPU újra megvizsgálja, hogy más master nem kéri-e a sín vezérlését, és ha kéri, akkor megtörténik a HLDAFF flip-flop 1-es értékre állítása.

A köztes állapotok végén ellenőrzésre kerül a HLDAFF flip-flop értéke, és amennyiben 1-re lett állítva, a processzor HOLD állapotba kerül, és itt is marad, amíg a hold felkérés fennáll egy másik mastertől.
Amint a hold kérés eltűnik, a HLDAFF flip-flop 0 állapotba állítódik.

Amennyiben nem kellett HOLD állapotba kerülnie a processzornak, akkor megvizsgálja, hogy az utasítás utolsó ciklusa történt-e (ne feledjük, hogy egy utasítás több ciklust is igényelhet, pl.: DAD) .
Ha a vizsgálat eredménye nemleges, akkor T_1 állapotból kezdődik elölről a sínciklus.
Ha a vizsgálat eredménye az, hogy az utolsó ciklus megtörtént az adott műveletnél, akkor az érvényes külső megszakítási kérelem vizsgálata történik.
Érvényes megszakítás esetén az INTAFF flip-flop 1-es értékre állítódik, és az INTEFF flip-flop, ami a megszakítás engedélyezéséért felel, 0 értékre állítódik. Egyszerre tehát egyetlen megszakítás lehetséges.
Amennyiben érvényes megszakítás nem érkezett, újból T_1 állapot következik.

A rendszer T_1 állapotból HALT utasításra T_{HALT} állapotba kerülhet, és itt is marad.
Hold kérelemre a HLDAFF flip-flop 1-re állítódik, és HOLD álapotba kerül a processzor.
Amennyiben HALT állapotból került HOLD állapotba, a hold kérelem megszűnése után HALT állapotba kerül vissza.
Amennyiben nem HALT állapotból került HOLD állapoba, a hold kérelem megszűnése után T_1 állapotból folytatódik a sínciklus.
T_{HALT} állatpoból IT jelre (külső megszakítás) T_1 állapotba kerül a rendszer, és megtörténik a HLDAF 0-ra és a HLDEFF 1-re állítása is.

T_4, T_5, T_6 állapotokban nincs szüksége a processzornak a sínre, így ilyenkor képes azt átadni.
Ez a sínkihasználás optimalizálása.

IT tehát csak két sínciklus közt hajtódhat végre, de buszról lemondhatunk a ciklus közepén is.
8085-összpecifikáció szerint az IT-t visszautasíthatja, azonban hold kérést köteles teljesíteni.

A segédlet 19. oldalán erről részletes ábrát találunk.

RESET működése:

Reset jelre az utasításszámláló (PC) és az INTEFF flip-flop nullázódik.
Reset eltűnése után T_1 állapot következik és FETCH típusú gépi ciklus indul.
$S_0D=0$
RST MASZK=1
RST 7.5 flip-flop (amely az interrupt=IT figyeléséért felel) törlődik.
Ezekről később részletesebben is lesz szó.

A regiszterek és flagek értéke reset után nem definit!

Utasítás felépítése:

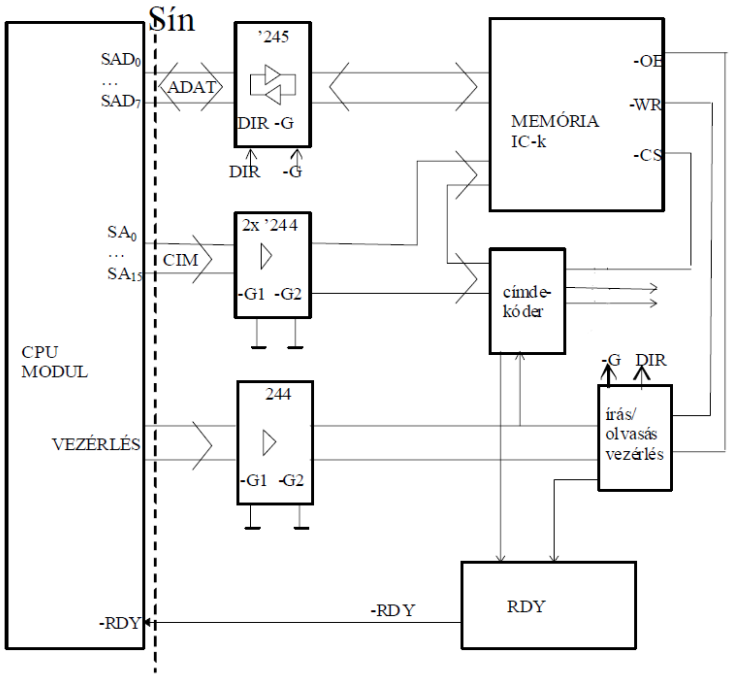
Egy utasítás legalább 8 és legfeljebb 24 bit hosszú.

Az első byte minden esetben egy **műveleti kód (opcode)**.

A második és harmadik byte a művelet operandusait tartalmazza:

- **Közvetlen adat**: Maga az operandus az adat (8 vagy 16 bit). Ekkor az operandus az utasítás része.
- **Direkt adatcímzés**: Az utasítás paramétere tartalmazza az operandus címét a memóriában (mindig 16 bit).
- **Indirekt adatcímzés**: Valamely regiszterpár címe, mely tartalmazza az operandus címét (2 darab 8 bites regiszter tartalmazza az opreandus 16 bites címét).

Memória illesztése mikroprocesszorokhoz:



Nézzük milyen ki és bemenetek állnak rendelkezésünkre:

Az adatok számára **SD_0 – SD_7** vezetékek érhetőek el.
A címezés az **SA_0 – SA_{15}** címvezetéken lehetséges.

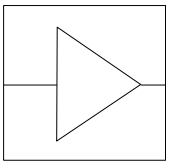
SIO/ /M: Meghatározza, hogy memóriát, vagy IO eszközt használunk.
/SMR : Memória olvasását jelzi.
/SMW: Memória írását jelzi.
/SIOR: IO eszköz olvasását jelzi.
/SIOW: IO eszköz írását jelzi.
SCLK: Órajel a szinkronizáláshoz.

S jelöli mindenhol, hogy a sín vezetékeiről van szó.

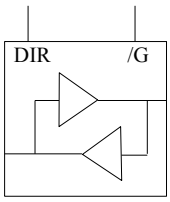
/SReady: Ready bemenet, melyen a külső eszköz jelezheti, ha készen van.

A sín tehát az adat a cím és a vezérlőjelek továbbítására szolgáló vezetékekből áll.

Korábban már megtárgyaltuk, hogy a kimeneteket a bemenetek terhelik, ezért a kimeneteket erősíteni kell. Erre szolgálnak a bufferek.



Egy és kétirányú buffer.



Az adat vezetékek esetén mindenképpen két irányú buffert kell alkalmaznunk.
A bufferek általában three-state kimenettel rendelkeznek, így tilthatóak.
A kétirányú buffere egy DIR és egy /G jellel vezérelhető. (a vezérlőjelek mindig negált logikával működnek, amit nem tüntetünk mindig fel, de az adott specifikációból mindig kiderül).

Ne feledjük, hogy a buffer két oldalán nem azonos jelek szerepelnek, így a buffer utáni jeleket mindig nevezzük el, hogy a rajzokon hivatkozhassunk rájuk!

MP:
Illesszük 8k-s memóriákat a mikroprocesszorhoz.

A 8k-s memóriánk 13 címvezetékkel rendelkezik, tehát az A_{13}, A_{14} és A_{15} címvezetékeket nem használjuk fel közvetlenül a memória címzésére.
Ezen fel nem használt címvezetékek egy **dekóder** alapján meg fogják határozni, hogy a 8 darab memóriamodul közül (16 címvezetékre 8 darab 8k-s memóriamodul illeszthető), mely modult címeztük. A dekóder az A_{13}, A_{14} és A_{15} címvezetékek alapján azt a memóriachipet fogja engedélyezni, amelyiket címezzük adott esetben.

A memóriamodulon található egy /OE (output enable) és egy /CS (chip select) vezérlő bemenet.
Előbbi a three-state kimenetek miatt lényeges, mindig legfeljebb egy chip kimenete lehet aktív.
A /CS bemenettel valósíthatjuk majd meg a vezérlést.

Kössük össze az /OE bemeneteket, és vezéreljük őket közösen.
Amíg a /CS nem aktív, addig hiába vannak engedélyezve a kimenetek, a chip nem csinál semmit.
Tehát már csak a /CS vezérlést kell megterveznünk.
A /OE és a /CS bemeneteken is 0 értéknek kell szerepelnie, hogy az adott memóriamodul működjön.

Szükségünk lesz egy **RDY logikára**, amely a /RDY jelet fogja előállítani.
Ennek működtetésére egy open collector kimenetű buffer szükséges, ami adott esetben a /RDY bemenetet 0-ba tudja húzni, így aktíválja azt.

A DIR és /G bemeneteket egy **írás-olvasás vezérlő** fogja kezelni.

A feladatunk, hogy megtervezzük a dekódert, a R/W vezérlőt és a RDY logikát.

Ha gyors memóriát illesztünk (0 wait fázist szeretnénk), akkor szigorúan tilos a R/W jelet használni a RDY logika megvalósításához.
Ready mintavétel a második fázis felfutó ideje előtti 110ns-os idősávban történik. Ez alapján a RDY jel előállítására körülbelül 50ns ideje maradna a logikának úgy, hogy ebbe még a bufferek miatti késleltetést is bele kell számolni.
Ha ez rossz ütemben történik, akkor mindenhol megjelenik egy wait ciklus, ami a rendszert nagy mértékben lassítja.

Használjunk 2764-es eprom memóriamodulokat, melyek a következő be és kimenetekkel rendelkeznek:
- A_0 … A_{12} címvezeték.
- D_0 … D_7 adatvezeték.
- /CE: chip enable bemenet.
- /OE: output enable bemenet.

A processzor 64k memóriát tud címezni, tehát a 8k-s modulokból 8 darabot használhatnánk fel. Illesszünk egyetlen egységet a 0000 – 1FFF címtartományba.
Memóriacímeknél előszeretettel használunk hexadecimális sámrendszert. A 0000 – 1FFF címtartomány a teljes tartomány első 8k-ját jelenti.

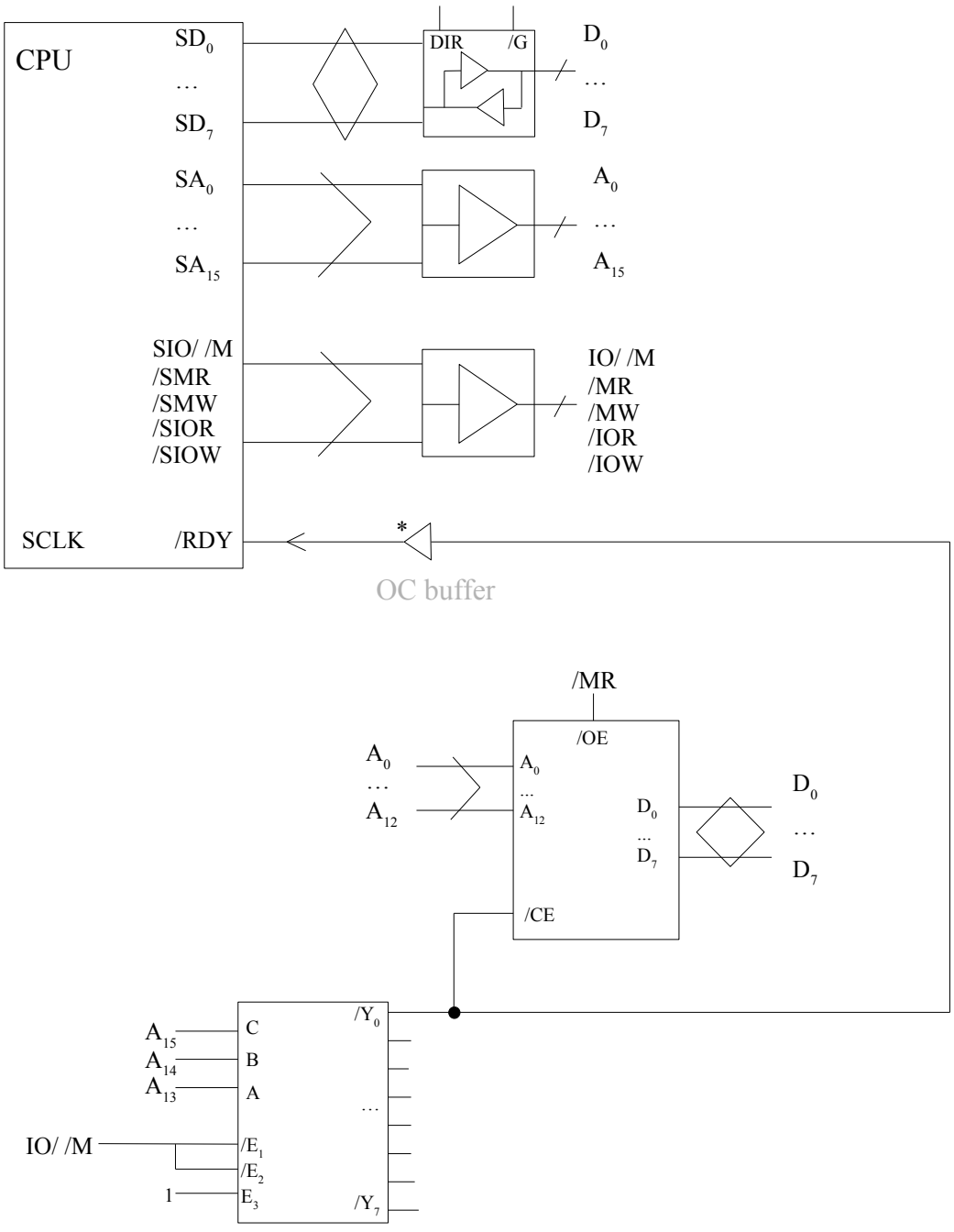
Hogy néznek ki a címeink:
0000 → 1FFF → Első 8k.
2000 → 3FFF → Második 8k.
4000 → 5FFF → Harmadik 8k.
…
DFFF → FFFF → Nyolcadik 8k.

Az első 13 címvezetéket kössük a memóriamodulunkra. Ezek fogják a memórián belül a címet biztosítani.

Az adott cím alapján a memóriamodulunkat az A_{13} … A_{15} címvezetékek és egy 74138-as dekóder segítségével fogjuk kiválasztani. A megfelelő helyiértékű címvezetéket a megfelelő helyiértékű bemenetre köthetjük. Az engedélyező bemeneteket az IO/ /M jelnek kell vezérelnie, melyet több módon i beköthetünk úgy, hogy ugyan azt a működést kapjuk. Mivel az A_{13} … A_{15} címvezetékek végig 0 értéket vesznek fel a számunkra lényeges címtartományban, ezért a dekóder $/Y_0$ kimenetén jelenik meg a 0 érték, amikor szükségünk van rá. Ez vezérli a chip enable-t. A dekóder negált kimenetekkel rendelkezik, ezért inverter nélkül köthetjük $/Y_0$-t /CE-re.

A 0 wait-et úgy valósíthatjuk meg, hogy a $/Y_0$ kimenetet kötjük a /RDY bemenetre egy opencollector bufferrel.

A memória kimenetét vezérlő /OE jelet a memória olvasásáért felelős jel, a /MR vezérli.



MP:
Illesszük a következő memóriákat a következő címtartományokra:

- Eprom(1) : 0000 – 1FFF (8k)
- Eprom(2): 2000 – 37FF (6k)
- RAM: 3800 - 3FFF (2k)

A második epromból 2k memórát nem használunk ki ebben az esetben.

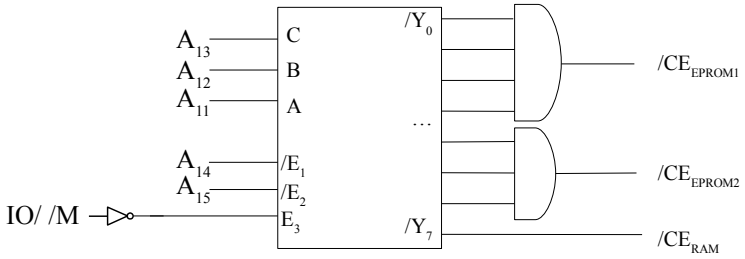
A memóriatartományt osszuk 2k-s szeletekre, mert jelenleg azzal a legkényelmesebb számolnunk:

Címtartomány:	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	
3800 - 3FFF	0	0	1	1	1	RAM
3000 - 37FF	0	0	1	1	0	Eprom(2)
2800 - 2FFF	0	0	1	0	1	
2000 - 27FF	0	0	1	0	0	
1800 - 1FFF	0	0	0	1	1	Eprom(1)
1000 - 17FF	0	0	0	1	0	
0800 - 0FFF	0	0	0	0	1	
0000 - 07FF	0	0	0	0	0	

Tervezzük meg a dekódert az illesztéshez:
A_{14} és A_{15} nem használható a megfelelő memória kiválasztásához, hiszen értékük végig 0.
Használjuk A_{13} … A_{11} címvezetékeket, majd határozzuk meg, hogy a dekóder mely kimenetei mely memóriához tartoznak, és kössük őket össze egy-egy AND kapuval, mivel a dekóder kimenetei negált logikával működnek.

Az A_{14} és A_{15} vezetékek azonban nem a teljes címtartományban nullák, így segítségükkel vezérelhetjük a dekóderünkek.
Ne feledjük, hogy IO/ /M-nek is szerepelnie kell a vezérlésben.

Az alábbi hálózat adódik:



MP:
Adottak a következő chipek, illesszük őket a megadott memóriatartományokba.
2 darab 2764 EPROM
1 darab 5565 statikus RAM

Címtartományok:
- EPROM(1): C000 – D7FF (6k)
- SRAM: D800 – E7FF (4k)
- EPROM(2): E800 – FFFF (6k)

A feladatokhoz meg szokás adni a sínen használható jeleket:
Címvezetékek: SA_0 … SA_{15}
Adatvezetékek: SD_0 … SD_7
IO vagy memória: SIO/ /M
Memória olvasás: /SMRD
Memória írás: /SMWR
Ready jelzéshez: /Sready

Könnyedén kiszámolható, hogy a felső 16kbit-re szeretnénk a memóriáinkat illeszteni.
A legnagyobb közös osztójuk a 2k, tehát osszuk a memóriatartományunkat 2k-s blokkokra, és vizsgáljuk meg, hogy ezekben a tartományokban hogy alakulnak a címvezetékek értékei, és hova szeretnénk illeszteni a memóriáinkat.

Honnan tudjuk, hány címvezetéket kell vizsgálnunk?
Ha csak az A_{15}-ös vezetéket nézzük, azzal a teljes 64k-s tartományt két részre oszthatjuk.
Ha hozzáveszük a következő helyiértéket, az A_{14}-et, akkor már 4 részre, azaz 16k-s szeletekre osztottuk a memóriatartományt.
Ezt a logikát követve, a 2k-s szelekethez 5 darab címvezeték figyelembevételével jutunk, tehát A_{15}-től A_{11}-ig kell figyelembe vennünk a címevezetékeket.

Címtartomány:	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	
F800 – FFFF	1	1	1	1	1	EPROM(2)
F000 – F7FF	1	1	1	1	0	
E800 – EFFF	1	1	1	0	1	
E000 – E7FF	1	1	1	0	0	SRAM
D800 – DFFF	1	1	0	1	1	
D000 – D7FF	1	1	0	1	0	EPROM(1)
C800 – CFFF	1	1	0	0	1	
C000 – C7FF	1	1	0	0	0	

Nézzük meg, hogy a 8k-s epromok esetén mely tartományokat veszítjük el, ha A_{12}-ig bekötjük a vezetékeket (8k-s epromhoz 13 vezeték szükséges):

EPROM(1)	
1800 – 1FFF	A_{12}=1 A_{11}=1
1000 – 17FF	A_{12}=1 A_{11}=0
0800 – 0FFF	A_{12}=0 A_{11}=1
0000 – 07FF	A_{12}=0 A_{11}=0

EPROM(2)	
1800 – 1FFF	A_{12}=1 A_{11}=1
1000 – 17FF	A_{12}=1 A_{11}=0
0800 – 0FFF	A_{12}=0 A_{11}=1
0000 – 07FF	A_{12}=0 A_{11}=0

Az EPROM(1) esetén a legfelső blokk pazarolódik el, hiszen abban a címtartományban, ahol őt el kell majd érnünk, az A_{12} és A_{11} vezetékeken nem fog megjelenni az 11 érték.

Az EPROM(2) esetén a legalsó blokk pazarolódik el, hiszen abban a címtartományban, ahol őt el kell majd érnünk, az A_{12} és A_{11} vezetékeken nem fog megjelenni a 00 érték.

Nézzük, hogy a 8k-s SRAM esetén mit címezhetünk és mi pazarolódik el:

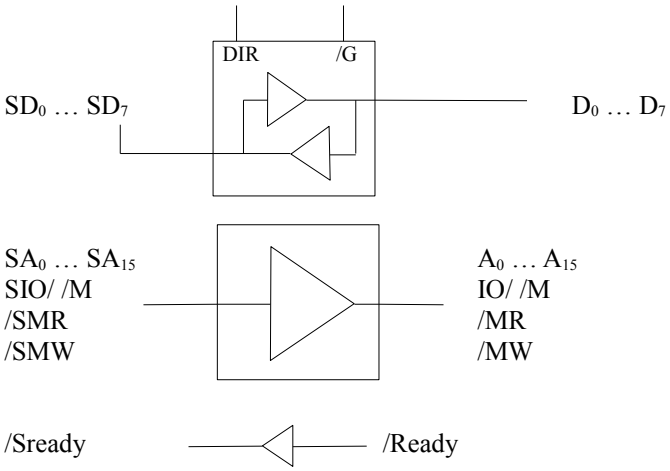
SRAM	
1800 – 1FFF	A_{12}=1 A_{11}=1
1000 – 17FF	A_{12}=1 A_{11}=0
0800 – 0FFF	A_{12}=0 A_{11}=1
0000 – 07FF	A_{12}=0 A_{11}=0

Az SRAM esetén a középső blokk pazarolódik el, hiszen abban a címtartományban, ahol őt el kell majd érnünk, az A_{12} és A_{11} vezetékeken nem fog megjelenni sem az 10 érték, sem pedig 01 érték.

Egy kis trükkel azonban elérhetnénk, hogy az SRAM-ból is folyamatos területet címezzünk. Statikus ramnál ennek még nincs jelentősége, azonban dinamikus ramnál fontos lenne, mivel a sor és oszlopcímzésnél gondokat okozna.

Azt szeretnénk, ha 00 ás 01 értékeket vennék fel a legnagyobb helyiértékek az SRAM címvezetékei közül.
A_{11}-el nincs problémánk, azonban A_{12} nem úgy működik, ahogy azt elvárnánk. Keressünk egy olyan címvezetéket, ami az SRAM tartományában a 0 értéket produkálja mindkét esetben. Az A_{13} pont egy ilyen vezeték, tehát csak annyi a dolgunk, hogy az A_{12} helyett, az A_{13}-at kötjük be az SRAM-ba.

Mielőtt elkezdenénk tervezni, válasszuk le az összes sínjelet bufferekkel. Ugyan már hivatkoztunk A_{12}-re és hasonlókra, de egyelőre csak sínjelek állnak rendelkezésünkre, így az A_{12} elnevezést sem használhatnánk például.



Tervezzük meg a dekódert:

A dekóder dekódoló bemenetein nincs értelme az A_{14} és A_{15} vezetékek használatának, mivel abban a tartományban, ahova illeszteni szeretnénk, ezek értéke végig 1-es.
Azonban használnunk kell őket, mert ahol ezek értéke nem 1-es, ott nem a mi memóriáink vannak, tehát ott nem engedélyeyzhetjük a memóriáinkat.
Használjuk fel őket a dekóder engedélyező bemeneteinél. 1 darab ponált engedélyező bemenet van, és nekünk két pontált engedélyező jelünk (A_{14}, A_{15}). Lehetséges megvalósítás, hogy egy AND kapun keresztül a ponált bemenetre kötjük. Alternatív megvalósítás lehet, hogy az egyiket invertálva egy negált engedélyező bemenetre kötjük, vagy akár mindkettőt negáljuk és a két negált bemenetre kötjük.

IO címek is lehetnek a vezetéken, az IO/ /M vezetéket is fel kell használnunk, ez lesz a 3. engedélyező bemenet. Célszerűen ezt egy negált logikájú engedélyező bemenetre érdemes kötni.

A megfelelő memória kiválasztásához az A_{13}, A_{12}, A_{11} vezetékeket kell a dekóder CBA bemeneteire kötnünk, majd meg kell határoznunk, hogy a kimenetek közül melyik tartozik melyik memóriachipünkhöz, és egy-egy AND kapuval ezeket a /CE bemenetekre kell kötnünk. Ekkor a kiválasztott memóriachip engedélyezve lesz.

A feladat általában megmondja, hogy szükséges-e várakoztatás. Ez esetben készítsünk 0 ready-s rendszert.
Az A14 és az A15 egyszerre egyedül az általunk megcímzett memóriák esetbében 1-es, ezeket tehát felhasználhatjuk egy AND kapu segítségével.
Azonban lehetséges, hogy IO cím jön, ekkor viszont nem szabad, hogy ready-t jelezzünk, hiszen az adott IO eszköz még lehet lassú. A ready jelbe tehát be kell vezetnünk az IO/ /M jelet is, ráadásul egy inverter közbeiktatásával, mivel a memória engedélyezés pont 0 esetén történik meg, nekünk viszont egy AND kapunk van.

Egy 3 bemenetű AND kapuba vezethetjük tehát A_{14}, A_{15} és $\overline{IO/\ /M}$ jeleket, és a kapu kimenete egy open collector inverteren keresztül lesz a /Ready jelünk.

Maradt még pár bekötetlen bemenet, amikre ki kell találnunk, hogy mit kössünk.

- A két irányú buffer DIR bemenetére /MR-t kell kötnünk, ekkor fogja átengedni a címeket a memóriák felé.

- Többet kell gondolkoznunk azon, hogy a G bemenetre mit kössünk, ami ugyebár azt jelzi, hogy cím vagy adat halad éppen át a két irányú bufferünkön.
  Szükségünk lesz az összes /CS jelre. Mivel negált jelek, ezért egy NAND kapuval gyűjtsük össze őket.
  Szükségünk lesz a /MR és /MW jelekre, ezeket szintén egy NAND kapuval gyűjtsük össze.
  A NAND kapuk kimenetét egy újabb NAND kapuba kössük.

  A NAND kapuk kimenete akkor 1, ha bármelyik bemenetükre 0 érték kerül.
  Pontosan akkor szeretnénk 1-es értéket, ha a memóriákból jövő adattal dolgozunk, ez pedig pontosan akkor lesz, ha memória műveletet jelző negált jeleink közül bármelyik aktív, és/vagy engedélyeztük a memóriachipjeink valamelyikét.

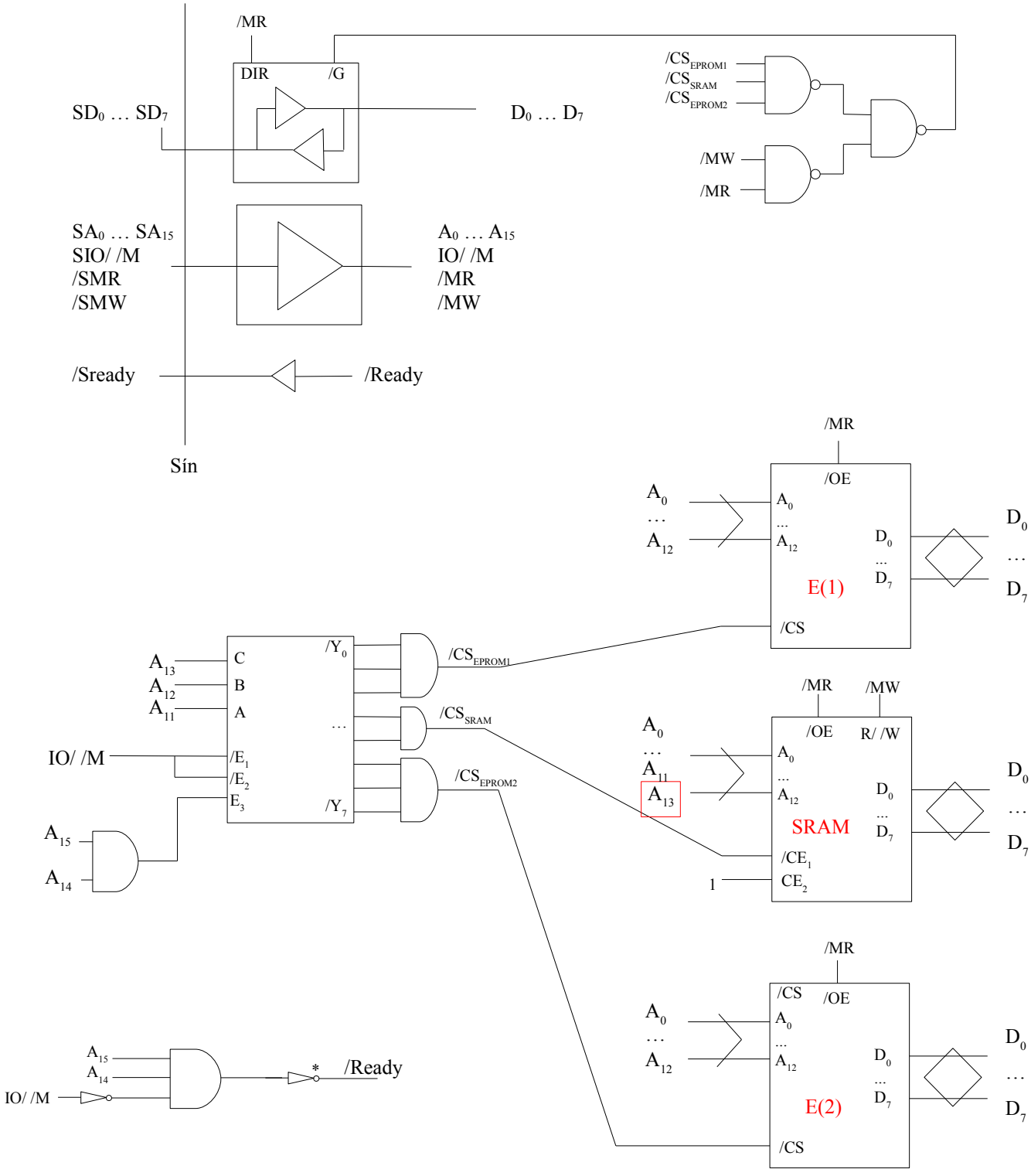
- Az epromok /OE bemenetére, mely a kimeneteiket engedélyezi a /MR jel tökéletesen megfelel.
  Ezzel nem lesz probléma, hiszen amíg nincs /CS jel, addig a chip-ek nem aktívak, így hiába engedélyeztük egyszerre az összes kimenetét.

- Az SRAM /OE bemenetére szintén köthetjük a /MR jelet.

- Az SRAM R/ /W bemenetére /MW tökéletes lesz.

- Az SRAM két darab chip select bemenettel rendelkezik, egy $/CE_1$ és CE_2 bemenet áll rendelkezésünkre a chip engedélyezéséhez.
  Nekünk azonban egyetlen negált logikájú engedélyező jel áll rendelkezésünkre.
  Vagy egy inverter segítségvel CE_2-re is az engedélyező jelet kötjük, vagy CE_2-t fixen 1 értékre kötjük.

Rajzoljuk meg:



MP:
Adottak a következő chipek, illesszük őket a megadott memóriatartományokba.
1 darab 27128 EPROM
1 darab 5565 statikus RAM
Tudjuk, hogy a memóriák 0 waittel működhetnek.

Címtartomány:
EPROM: 0000 – 3FFF (16k)
RAM: E000 – FFFF (8k)

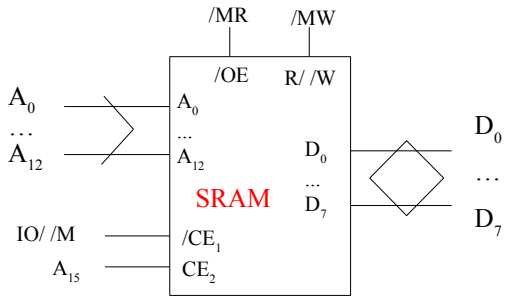
A sínen rendelkezésre álló jelek (buffereket elhagyhatjuk az egyszerűségre való tekintettel):
/MR
/MW
IO/ /M

Valósítsunk meg nem teljes memóriaillesztést, mivel tudjuk, hogy a rendszerben utólagos memóriabővítésre nem kerül sor. Figyeljünk arra, hogy IO eszközök viszont használhatják a buszt.
A célunk a legegyszerűbb címdekóder megtervezése, mely könnyen kitalálható, hogy nem egy teljes dekóder felhasználásával fog megvalósulni.

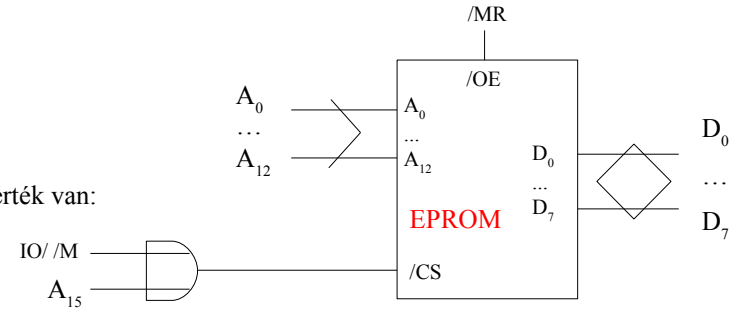
Nézzük mit tudunk használni dekóder helyett:
Az SRAM a memória címtartományának alsó nyolcadában helyezkedik el, míg az EPROM a felső negyedben. Könnyedén észrevehetjük, hogy már a legmagasabb helyiértékű címbit is egyértelműen megkülönbözteti őket, tehát ezt használhatjuk nyugodtan vezérlésre a dekóder helyett (mivel tudjuk, hogy más memória nem is fog soha bekerülni a rendszerbe, ezért nem kell azzal törődnünk, hogy több cím esetén is aktív lesz a memória).

A többi vezérlőjelet nem is tárgyaljuk, csak a rajzon tüntetjük fel:

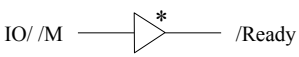
Ha s RAM memóriát keressük, és az A_{15} vezetéken 1 érték van:



Ha az EPROM-ot keressük, és az A_{15} vezetéken 0 érték van:



A readyt egy open collector bufferrel és az IO/ /M jellel könnyedén megvalósíthatjuk.



MP:
Hogyan készíthetünk 1 wait-et.

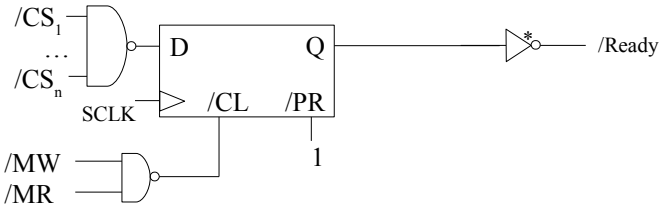
Vegyünk egy D flip-flopot.
A /PR bemenetet kössük 1-re, mivel nincs rá szükségünk.
A /CL bemenetre azokat a vezérlőjeleket kössük NAND kapun keresztül, amik esetén wait-et szeretnénk kérni.
Ha írás és olvasás esetén is szeretnénk wait-et, akkor /MR NAND /MW-t kell /CL-re kötnünk.
Ha csak írás, vagy csak olvasás esetén szeretnénk vaitet, akkor /MW vagy /MR jeleket kell a /CL bemenetre kötnünk egy inverteren keresztül.
Ezzel azt érjük el, hogy amíg nincs /MW és/vagy /MR jelünk, addig az inverter miatt a /CL bemeneten folyamatosan 0 érték jelenik meg, így a D flip-flop passzív. Amint megjönnek a vezérlő 0-k, a D flip-flop aktiválódik, mivel megszűnik a folyamatos clear állapot.

A flip-flopunk bemenetére egy NAND kapu segítségével össze kell gyűjtenünk az összes olyan eszközt, aminek 1 wait-re van szüksége.

A D flip-flop kimenete egy open collector inverteren keresztül megvalósítja a /Ready jelet, amely 1 wait fázist kér a processzortól.

Nézzük hogy néz ki ez a procedúra:
A read és write jelek inaktivitása miatt a flip-flop folyamatos törlésben van a T_1 fázisban.
A T_2 fázisban megjelenik a read vagy a write jel, tehát a flip-flop innentől kezdve működhet.
A felfutó él előtt vesz jelet a readyből a processzor, a flip-flop kimenete azonban ekkor még nem jelenik meg, tehát a processzor közbeiktat egy TW fázist.
Ekkor a felfutóél hatására a flip-flop kimenete átbillen, az open collector kimenetén megjelenik a /Ready jel, amit a processzor a következő felfutó él előtt érzékel is, így a T_3 fázisba lép.

Rajz:



Shiftregiszterhez hasonló felépítéssel 2, 3 … 6 wait fázist kérhetünk a processzortól.

A különböző ready logikák működéséről a tanszék jegyzetében találunk részletes információt.

A 8085-ös szoftvermodellje:

Milyen műveleteket milyen adatokon.
Milyen utasítások és ehhez milyen belső regiszterek

6 általános célú 8 bites regiszter (BCDEHL)
Aritmetikai logikai műveleteire kitüntetett regiszter akkumlátor (A). Eredmény is itt keletkezik.

A regiszterek párokat alkotnak bizonyos műveletekben, 16 bites regiszterként hivatkozhatunk rájuk.
Ekkor amagasabb helyiértékű regiszterre kell hivatkoznunk.
BC → B
DE → D
HL → H

Az akkumulátort egy program státusz szóval (PSW)) egészül ki.
5 bit.
Ez az 5 bit a következő flagekből áll, már korábban megtárgyaltuk őket, csak ismétlés:
Z: 0-e az eredmény.
CY: carry a következő helyiérték felé
S: sign előjel
P: parity egyesek páros v páratlan
AC: auxiliary carry segédátvitel BCD 4-5 bit közt.

Későbbi processzorokban például túlcsordulás flag …

2 darab 16 bites regiszterünk is van:
PC: program counter programszámláló, a következő utasításra mutat
SP: stack mutató, ami a stackre utolsóként berakott értékre mutat

Adatmozgatás:
regiszeterből regiszterbe
regiszter ↔ memória közt

Aritmetikai műveletek:
összeadás és kivonás

Logikai műveletek:
AND, OR, NOT (bitenkénti és / vagy / negálás)
XOR (kizáró vagy)

Vezérlésátadó:
Elágazás (jump) művelet
Szubrutin hívás, alprogram hívás (olyan programrészlet aktiválása, amely után vissza szeretnénk térni) (call és return)
Ennek meg kell jegyezni hova kell visszatérni, erre jó a stack.

Stack:
Berakás a stackbe (push)
Kiszedés a stackről (pop)

Egyéb utasítások:
IO és IT kezeléséhez szükséges utasítások.

Címzési módok:
ADAT: közvetlen adat, utasítás tartalmazza az adatot
regisztercímzés, melyik regiszter tartalmazza az adatot
közvetlen adatcímzés: az utasítás tartalmazza az adat címét
regiszter indirekt adatcímzés (regiszterpár tartalmazza az adat címét)
Általában a HL regiszterpár, néha B és D párok is.

UTASÍTÁS: közvetlen utasításcímzés, az utasítás a következő utasítás címét tartalmazza
regiszter indirekt utasításcímzés: egy regiszterpár tartalmazza a következő utasítás címét. (Ez a HL pár lesz)

A program lényegében adat.

Gépi kódú programozás:
Assembler: az a program, mely a gép utasításait szimbólumokkal helyettesíti.
PL:
B → A => 78H kód
Assemby nyelven: B → A => MOV A, B

FORRÁS → ASEEMBLER → GÉPI KÓD / LISTA / KERESZT REFERENCIA

[CÍMKE:] op kód (mnemonic). Általában a művelet [operandus] [; megjegyzés pontosvessző után]

Az opreandus lehet regiszter melyekre betüjekükkel hivatkozunk. (A, B, C, D, E, H, L)
Lehet regiszterpár (B, D, H, SP = stack pointer)
Lehet közvetlen adat vagy közvetlen cím, ezek számértékeke.
- Megadható binárisan: 01100101B
- oktálisan: 144Q
- decimális: 12345D (ha nem írunk jelölő betűt, akkor alapértelmezés a decimális)
- hexadecimális: 64H (AFH → 0AFH, hogy szám legyen!)
- ASCII konstans: 'A', 'B'
- Címke: START
- Definiált konstans pl.: DARAB (ha definiáltuk a DARAB-ot konstansként)
- Kifejezés pl.: DARAB+1

KONSTANS definiálása:
- DARAB **EQU** 5 → nemlehet felülbírálni

- SOK **SET** 5 → felül lehet bírálni
- SOK SET 3

A fordítás sorrendben történik.

ADAT definiálása:
Byte: DB 10, 0AH, 'A', 'Bab'
SZÓ: DW 1234H, 12H, 'A', 'BC'

A 8085-ös processzor úgynevezett little endian ábrázolási móddal dolgozik, ami azt jelenti, hogy az összetartozó bájtokból alacsonyabb címen az alacsonyabb helyiérték érhető el.
PL.: 1234H-nál előbb a 34H és utána a 12H lesz eltárolva.

Helyfoglalás:
ADAT1: DS 10 a megadott konstansnak megfelelő területet biztosít.
ADAT2: DS 10

Feltételes fordítás:
IF
ELSE
ENDIF

Assembler vége:
END

MEMÓRIA felhasználása:
Kód → ROM-ban kell lennie
Konstans → ROM memóriában célszerű tárolni
ADAT → menet közben változik, tehát RAM
STACK → ideiglenes adatok tárolására, RAM

CSEG → a leírt utasítássorozat kódszegmens
DSEG → a leírt utasítássorozat adatszegmens

CSEG [PAGE] → opcionális paraméterként megadható kulcsszó, amelynek a debugolásban volt jelentősége.
A page kulcsszó azt jelenti, hogy a kódnak 256-tal osztható címen kell kezdődnie.
Debugolásnál sokkal könynebb volt hibát megtalálni, de lyukak a memóriában.

ORG 16 bites cím direktíva: az adat az itt megadott címen kezdődjön.
ASEG: abszolút szegmensbe helyezhetjük az abszolút címen elhelyezkedő előre meghatározott adatokat.

PUBLIC DARAB: publikussá tehető, mindenki eléri
EXTRN DARAB: külső fileban szereplő adat, ez az előző PUBLIC ADAT-ra hivatkozik.

Milyen konkrét utasításai vannak a 8085-ös processzornak:

Tipikus utasítás: típus cél, forrás

MOV R1, R2: mozgat R1 regiszterbe R2 regiszter tartalmát; flageket nem állítja

1 FETCH típusú ciklus, 4 fázis

MOV M, r : r regiszter tartalmát a HL regiszterpár által meghatározott memóriarekeszbe mozgatja.

1 FETCH típusú gépi ciklus, 4 fázis + 1 adatátvitel, 3 fázis

MVI r, adat: 8 bites adatot mozgatja az r regiszterbe (I betű általában a közvetlen adatra utal)

1 FETCH típusú gépi ciklus, 4 fázis + 1 adatátvitel, 3 fázis

LXI rpár, adat16: 16 bite adaot a regiszterpárba mozgat (X általában 16 bites regiszterre utal, I közvetlen adat)

MP:
LXI H, 1000H
Memóriában hogy helyezkedik el:
LXI → 21H
1000H: 00 majd 10

1 FETCH + 2 adatátvitel : 10 fázis

LDA cím16: a cím 16 megy akkumulátorba, 13 fázis 4 gépi ciklusban.
STA cím 16:: a 16 bites címre mozgatja az Akku tartalmát.

LDAX rp: regiszterpár által mutatott memória tartalmát mozgatja az akkuba
STAX rp: az akkumulátort mozgatja a regiszterpár által mutatott memóriába

regiszterpárra csak B és D

2 gépi ciklus 7 fázis

XCHG DE és HL regiszterpárt cseréli.
1 gépi ciklus és 4 fázisátvitel

LHLD cím16: a HL regiszterpárba mozgatja a megadott cím tartalmát
SHLD cím16: megadott címre írja a HL regiszterpár tartalmát

1 FETCH, 2 adat, 2 adat = 5 gépi ciklus 16 fázisban

Arimetikai utasítások:

ADD r : A ← A+r állatja az összes flag-et. 1 gépi ciklus, 4 fázis.
ADD M: A ← A + [HL] flagek állítása, 2 gépiciklus, 7 fázis
ADI 8 bites adat: A ← A + 8 bites adat flagek, 2 gépi ciklus, 7 fázis.
ADC r: A ← A + r + CY (carry) 1 gépi ciklus 4 fázis
ADC M: A ← A + [HL] 2/7
ACI 8 bites adat: A ← A + adat + CY 2/7

SUB r : A ← A-r állatja az összes flag-et. 1 gépi ciklus, 4 fázis.
SUB M: A ← A - [HL] flagek állítása, 2 gépiciklus, 7 fázis
SUI 8 bites adat: A ← A - 8 bites adat flagek, 2 gépi ciklus, 7 fázis.
SBB r: A ← A + r - CY (carry) 1 gépi ciklus 4 fázis
SBB M: A ← A - [HL] 2/7
SBI 8 bites adat: A ← A - adat - CY 2/7

INR r: r++ ¼ carryt nem állít
DEC: r-- ¼ carryt nem állít
INC M: memóriarekesz értékének növelése 3/10 flagek igen, carry nem
DEC M: memóriarekesz értékének csökkentése 3/10

INX rpár regiszterpár értékét növeli 1/6 flagek nem
DEX rpár regiszterpár értékét csökkenti 1/6

DAD rpár HL ← HL + rpár 3/10 csak carryt állít.

DAA
Ha A0 … A3 >9 vagy AC = 1 → A0... 3 +=6 (BCD)
Ha A4...7 > 9 vagy CY = 1 → A4...A7 += 6

Logika:
ANA r : A ← A AND r flegeket állítja (carryt törli), ¼

ANA M: A ← A AND [HL] 2/7
ANI adat 8 biten: A ← A AND adat 2/7

ORA r : A ← A OR r flegeket állítja (carryt törli), ¼
ORA M: A ← A OR [HL] 2/7
ORI adat 8 biten: A ← A OR adat 2/7

XRA r : A ← A XOR r flegeket állítja (carryt törli), ¼
XRA M: A ← A XOR [HL] 2/7
XRI adat 8 biten: A ← A XOR adat 2/7

CMP r: akkumulátorból kivonja az r regiszter értékét, és csak a flageket változtaja, az eredmény elvész. 1/4
CMP M: -"- 2/7
CMI adat8 2/7
Mire jó: Z = 1 → A=r
CY=1 → A < r

Csak 8 bit, kettes komplemensre nem jó.

RLC:
RRC:

RAL:
RAR:

CMA: A ← /A
STC: CY ← 1
CMC: CY ← /CY

Elágazások:
JMP cím16 : a programszámlálóba tölti a címet
JCC cím16 : a programszámlálóba tölti a címet ha CC igaz
CC: Z → Z=1 ugrunk
NZ → Z=0 ugrunk
C → carry = 1 ugrunk
NC
M: minus Sign flag értéke 1
P: positive sign flag értéke 0
PE: paritás = 1
PO: paritás = 0

PCHL: a programszámlálóba töthetjük a HL tartalmát. 1/6

SZUBRUTIN:
Meghívunk egy alprogramot, majd ugyan oda szeretnénk visszaugrani. Eddig más tárgyakból a függvényeknek felel meg.
Stack művelet.

CALL cím16: PC ← cím16 5 ciklus 18 fázis

Ccc cím16: PC ← cím16, ha a címke igaz.
Ha cc igaz, akkor sima CALL.
Ha cc nem igaz, akkor 2. ciklusban megszakít, ekkor 9 fázis

RET: PC ← SP majd SP ← SP + 2 3/10
Stack pointerrel mindig az alacsonyabb helyiértéket érjük el, és csak utána a magasabbat.

Rcc: feltételes vissztatérés címke szerint.
3/12 vagy 1/6 ha nem történik meg

RSTn restart utasítás (8 darab n van, 0...7)
3 gépiciklus 12 fázis.

STACK művelet:
PUSH rp: STACK ← rpár 3/12
magasabb helyiérték SP-1-re
alacsonyabb SP-2-re, SP kettővel csökken
POP rp: a STACK értékét a regiszterpárba töltjük alacsonyabb helyiértéket SP-ről, magasabbat SP+1-ről. SP értékée kettővel nő. 3/10

Rpár lehet B, , H, PSW

XTHL: stack tetjén lévő adat és HL tartalmát cseréli.
Következő képpen hajtódik végre:
FETCH gépiciklus
SP értékét behozza egy regiszterbe, SP+1 értéket másik regiszterbe, aztán SP+1-re H és SP-re L értékét.

Egyéb:

IN cím8	IO eszköz kezelése, nem működik dinamikusan, fordítási időben meg kell mondani.
OUT cím8	A felső 8 bitre (16 bitünk van) másolja az alsó 8 bitet. 3/10
HLT	A processzor HALT állapotba kerül. A HLT kódja megegyezik a MOV M, M-mel, inteligensebb fordítók szólnak.
NOP	no operation, 1/4 (Szerencsétlen, mert az üers EPROM-ban csupa 1-es van, tehát ha nem 0 lenne a NOP kódja, akkor az epromban lehetett volna helyet hagyni, ahova utólag tudunk égetni.)
RIM	IT maszkolási állapotát olvashatjuk
SIM	IT állapotát állíthatjuk be
EI	engedélyezhetjük az IT-t
DI	tilthatjuk az IT-t.

MP:
Ki szeretnénk írni egy eszközre, hogy „Hello”

Definiálnunk kell azt a karaktersorozatot, hogy hello:

```
DSEG
TEXT: DB 'HELLO', 0

…
EXTRN STROUT        (megmondjuk a fordítónak, hogy majd a linker megtalálja az STROUT -ot)
CSEG
        …
        LXI H, TEXT
        CALL STROUT
        …
```

Egy másik fileban elhelyezkedik az STROUT:

```
EXTRN CHROUT        (a külső eszközt ezen a néven érjük el)
PUBLIC STROUT       (azt szeretnénk, hogy más is lássa a filet, így a linker megtalálhatja)
CSEG: STROUT

STROUR: MOV A, M    (felhozzuk az adatot)
```

A mov utasítás nem állítja a flag-eket, logikai művelet kell, ami nem rontja el az értékeket, de beállítja a flag-eket.

```
        ORA A
        RZ
        CALL CHROUT
        INX M
        JMP STROUT


PUBLIC CHROUT
TXRDY EQU 0000001B
USARYD EQU 80H
USARTC EQU USARYD + 1

CSEG
CHROUT: PUSH PSW
TXWAIT: IN USARYD
        ANI TXRYD
        JZ TXWAIT
        POP PSW
        OUT USARXD
        RET
```

Konstansokat érdemes egy helyen definiálni.

MP:

```
ORG 1000H
LXI SP, 8000H       31
MVI A, 2            3E
STA 8000H           32
ORA A               B7
JZ TOVE             C2
CALL SZUB           CD
TOVA: HLT           76
        NOP         00
SZUB: LXI H, 8000H  21
CIKL: DCR M         35
        RZ          C8
        CMA         2F
        JMP CIKL    C3
```

Cím	Adat (utasítás kódok)	Forrás	RD/ /WR	REG	MEM	FLAG
1000	31	PC	RD			
1001	00	PC	RD			
1002	80	PC	RD	SP = 8000		
1003	3E	PC	RD			
1004	02	PC	RD	A = 2		
1005	32	PC	RD			
1006	00	PC	RD			
1007	80	PC	RD			
8000	02	cím16	WR	[8000] = 2		
1008	B7	PC	RD	CY = 0, Z=0, S=0		
1009	C2	PC	RD			
100A	0F	PC	RD			
100B	10	PC	RD			
100C	CD	PC	RD			
100D	11	PC	RD			
100E	10	PC	RD	PC=1011		
7FFF	10	SP	WR	PC eredeti értékét a STACK-re kell menteni Hogy visszatérhessünk.		
7FFE	0F	SP	WR			
1011	21	PC	RD			
1012	00	PC	RD			
1013	80	PC	RD			
1014	35	PC	RD			
8000	02	HL	RD			
8000	01	HL	WR	[8000] = 01		
1015	C8	PC	RD			
1016	2F	PC	RD	A=FD, S=1		
1017	C3	PC	RD			
1018	14	PC	RD			
1019	10	PC	RD	PC=1014		
1014	35	PC	RD			
8000	01	HL	RD			
8000	00	HL	WR	[8000] = 00, z=1, S=0		
1015	C8	PC	RD			
7FFE	0F	SP	RD			
7FFF	10	SP	RD	SP = 8000, PC=100F		
100F	76	PC	RD			
HALT						

EGY BITES IO:
SID: serial input data
SOD: serial output data

Vezérlés: RIM/ SIM

RIM esetén az akkumulátorban keletkezik 8 bites adat, amely legmagasabb helyiértéke a SID értéke.

SIM esetén az akkumulátor vár egy 8 bites adatot, amelynek legmagasabb helyiértéke a SOD értéke, utána levő helyiértéken SOD írásengedély van.

Írjunk programot, amely a SID bemenetet átmásolja a SOD kimenetre.

SID MSK EQU 80H (definiáljuk a konstanstt, ami megmondja, hogy a beolvasott adatban mi a SID aktuális értéke)

SODSET EQU 40H

RIM (read interrupt mask)
ANI SIDMASK (a legmagasabb helyiérték 1, ha SID=1 volt, minden más 0, egyébként minden 0)
(ezt szeretnénk kivinni a SOD-ra)
(Ha negáltan szeretnénk:
// XRI SIDMSK //

ORI SODSET
SIM

PROGRAMMAL ELLENŐRZÖTT KÉSZENLÉT

kell valamit csinálni?
Ha nem, akkor tesszük a dolgunk majd megkérdezzük újra
Ha igen akkor megcsináljuk, és utána megint megkérdezzük.

MEGSZAKÍTÁS (interrupt, IT)

Fut a főprogram, jön az IT, végrehajtódik az IT kezelés, majd a főprogram megfelelő helyére visszatérünk.

ÉL vezérelt → flip-flop kell a tároláshoz.

SZINT vezérelt → Folyamatos jelzés

ÉL + SZINT → egyszeri jel után folyamatos jelzés (speciális eset)

Mi kell az IT bekövetkezéséhez:

Fajtái:
- maszkolás
- prioritás
- nem maszkolható IT

Kell:
Engedélyezve legyen
ne legyen nagyobb prioritású aktuális IT
aktuális utasítás fejeződjön be.

MP:
0 wait, de kizárólag olvasásra.

A processzornak van egy státuszjel csoportja.

	IO/ /M	S1	S0
OP kód fetch	0	1	1
Memória olvasás	0	1	0
Memória írás	0	0	1

(Fetch: memóriából olvassa a processzor a műveleti kódot.)

Az a feladatunk, hogy meghatározzuk, hogy a memória szempontjából mikor olvasunk adatot.
Ez az S1= 1 és S0=- érték esetén áll elő.

A /CS (chips select) jelet invertáljuk, éseljük össze S1-el, és kössük egy OC nverteren keresztül a /RDY-re.

MP:
RAM:
0Wait
$/CS_R$ vezérlőjele van.
EPROM:
1W olvasás, /CSEPROM1
EPROM2:
2W, csak olvasás, /CSEPROM2

RAM:
Egyszerűen egy OC buffer /Rdy-re a /CSR

Waitekhez flip-flopot használtunk.
2Wait-hez minimum 2 D flip-flop kell.

Órajelek a rendszersínen lévő CLKout.ról vezérelhetjük
Preset nem haszált, logikai 1

/MR-rel vezérelhetjük a /CL-t egy inverteren keresztül.

Kössünk 1-est az első flip.flopra.

Az első ff kimenetét egy OC NAND kapuval kössük össze /CSEPROM1 invertáltjával.
A második ff kimenetét ugyan úgy /CSEPROM2 invertáltjával.

Készen vagyunk.

MP:

Adott:
2764Eprom
5565RAM

SA0 … SA15
SD0 … SD7

Illesszük:
EPROM: 0000 – 07FF
2000 – 2FFF

Vezérlőjelek:
/SR
/SW
SIO/ /M
CLKout

Címtérkép:

SA15	SA14	SA13	SA12	SA11	SA10 … SA0
0	0	1	0	1	x
0	0	1	0	0	x
0	0	0	0	1	x
0	0	0	1	0	x
0	0	0	0	0	x

Dekódolásra SA11... SA13
Engedélyezőre SA14, SA15, IO/ /M tagadottja mondjuk.

AND kapukkal készen vannak a chips sleectek.

Az EPROMba viszont nem mindegy mit kötünk.
Ha A12-ig mindent bekötnénk, akkor tévesen kétszer képeznénk ugyan oda.
Ide A13-at kell kötni!!!

/OE a /MR-nek felel meg.
/CS pedig

PROGRAMOZÁS:
Memóriából 3 órajel alatt hozhatjuk ki az adatot.

A szoftvermodell → memóriamodell:

Teljes memóriát 4 részre oszthatjuk.

Reset hatására PC 0000 címre áll.
Az itt lévő memória egy nemfelejtő memória (EPROM, ROM …). Ez a kódmemória.
A memória legmagasabb címétől lefele a STACK pointer terjeszkedik lefele. Ez mindenképpen egy írható memória kell, hogy legyen, tipikusan RAM.
A RAM azon része, amit nem használunk el STACK-re, az inicializálatlan változók tárolására szolgál.

A ki nem használt rész az inicializált változók tárolására jó. Ez RAM/ROM, mindkettő szükséges.
Ezt úgy használhatjuk, hogy a ROM tartalmát akód elején lemásoljuk.

CSEG: a kódrészhez tartozik.
DSEG: az inicializált változókhoz tartozik
ASEG, BSEG, (ESEG): inicializálatlan változókhoz tartozik
SSEG: a stack részhez tartozik.

Konvenciók, szimbólumok, bármelyiket használhatnánk, de ne tegyük.
Fordítónak szól, mi hova kerüljön.

A fordító:
Fordítói direktívák:
ORG cím16
CSEG
DSEG

Adatterület definiálása:
DB adat8 [string]
DW adat16
DS darab

Konstans definiálása:
EQU

pl.: I8255A EQU 0F0H

A címke kötelezően betűvel kezdődik, a szám kötelezően számmal.

pl.: I8255A EQU I8255A+1

MP:
Nézzük a következő kis programot:
ORG 100h

DB 0, 128
DB 0FFh
DB 'A'
DB „Hello”
DW 1234h

buff:
DS 2
DB 0Ah

Nézzük milyen címen milyen adat keletkezik:

0100	00
0101	80
0102	FF
0103	41
0104	48
0105	65
0106	6C
0107	6C

0108	6F
0109	34
010A	12
010B	xx (Nem változik az érték, lefoglalja mint buffer)
010C	xx
010D	0A

PUBLIC és EXTRN párok.

MP:
9000h → 32bájt másolás 9800h-tól.

LXI H, 9000h (HL registerpárt használjuk)
LXI D, 9800h (DE regiszterpárt használjuk)
MVI C, 32

ciklus: (így ő egy cimke)
MOV A, M (bejön az akkumulátorba az M adat)
STAX D (Az akkumulátor tartalmát viszi a D által mutatott címre)
INX H
INX D
DCR C
JNZ

Megszakítások:
Szinkron: rendszeresen megkérdezem
Aszinkron: a külvilág szól

5 ajtó van a 8085-ös processzoron interrupt (külső megszakítás) kezelésére.

RST 5.5 (szint érzékeny)
RST 6.5 (szint érzékeny)
RST 7.5 (él érzékeny = kell egy 0-1 átmenet, hogy IT bekövetkezhessen, ezt egy flip-flop tárolja, ami az IT után törlődik.)

Az ezekhez tartozó szubrutin címét úgy kell kiszámolni, hogy az interrupt sorszámát szorozzuk nyolccal.
Call utasítás azzal jár, hogy a stackbe elmentjük a PC aktuális értékét, és a call utasításnak adjuk a vezérlést.
Ekkor az interrupt engedélyező flip-flop törlődik.
Szintvezérelt interrupt esetén végtelen ciklusba kerülnénk, ha nem törlődne ez a flip-flop, mivel a call utolsó utasításánál az IT várna, mert még nem szolgálták ki.

Az interrupt flip-flop visszaállítását külön paranccsal kell engedélyezni.

Tehát egy IT rutin vége a következőképpen néz ki:
…
EI
RET

Az EI utasításnak azonban van egy érdekes tulajdonsága. Az IT engedélyezése nem egyből, hanem a RET után következik be.
Ekkor a stack.et nem terheljük akkor sem, ha több interruptot ágyazunk egybe.

Van másik ajtónk:

TRAP (4.5) (él + szint érzékeny, IT elfogadásához 0-1 átmenet kell, és az 1-es értéknek az IT bekövetkeztéig ott kell lennie, de egészen addig nem lesz újabb interrupt, amíg újabb 0-1 átmenet nem következik).

Egyszerre bekövetkező események közül milyen sorrendben szolgáljuk ki az eseményeket?
Processzorunk először a TRAP-et, majd a sorszám szerint visszafele szolgálja ki a bemeneteket.

Hogy tilthatunk IT-t.
Maskolási technikát használ a processzor.
RST interruptokra bemenetenként megmondhatom, hogy ki legyenek-e szolgálva.
A TRAP nem maszkolható.

SIM parancs 8 bitje közül az alsó 4-et használhatjuk.
ME 7.5 6.5 5.5

Maszk engedélyezéséhez 1-s értéket kell írnunk, a maszkoló bitnek viszont pont 0-nak kell lennie, hogy tiltsuk az IT-t.

Ha tiltani szereetnénk akkor pár dolgot meg kell tennünk:
Csak a 6.5-öst szeretnénk tiltani, de nem tudjuk, mi van a többivel.

Az RIM utasítás használható. Beolvassa az interrupt maskot.
Az utolsó 3 bitje érdekel.
Olyan utasítás szükséges, ami a 7.5 és 5.5 értékeket békénhagyja, de a 6.5-öt nullázza.
Éseljük össze az 101 bitmintával (=5).

Nézzük az utasítássorozatot:

RIM
ANI 5 (persze ezt 3 nap múlva nem tudjuk miért, ezért megtehetjük, hogy IT65EN EQU 5 kódot írjuk a kódsorozatunk elé, és ezt írjuk az ANI mögé, vagy kommentelhetünk).
ORI 8 (ITMASZK EQU 8 a kódsor elé, majd 8 helyére ez a kódsorozat = a maszk beírását engedélyeznünk kell!!!!)
SIM

A RIM utasítás 2. 3. és 4. bitje az IT aktuális értéke a 7.5, 6.5 és 5.5 IT-k értéke.

A RIM utasítás 5. bitje az INTE flip-flop értéke.
Ennek csak akkor van értelme, ha preIT-be léptünk be, egyébként garantáltan 0 lesz az INTE flip-flop értéke.
Éppen ezért ez a bit nem azt mutatja meg, hogy mi az aktuális érték, hanem azt mutatja meg, hogy az előző érték mi volt.

Preinterruptból visszatéréskor megkell nézni, hogy mi volt az előző értéke az INTE flip-flop-nak.

```
INTEFF EQU 8
RIM
ANI INTEFF
JNZ INTE
POP PSW
RET
INTE: POP PSW
EI
RET
```

Mi a feltétele annak, hogy egy IT beköetkezhessen.

1. INTEFF = 1   (TRAP-re nem vonatkozik)
2. ITMASZK = 0 (RST 5.5, 6.5, 7.5-re vonatkozik)
3. „Mi legyünk a legmagasabb prioritásúak.” (Az RST 6.5 bekövetkezéséhez az kell, hogy ne várakozzon se TRAP, se RST 7.5)
4. aktuális utasítás fejeződjön be

Ezen feltételek megléte mellett mi következik:
1. BUS IDLE ciklus indul
    INTEFF = 0 megtörténik
    (RST 7.5 FF = 0 (ha 7.5 kerül kiszolgálásra, mivel élvezérelt, ezért kell törölni))
2. PC mentése STACK-be.
3. PC-be kerül az aktuális interrupt sorszáma * 8 érték.
    (6.5 esetén például a 52 cím)
    Minden más a programozó feladata.

A SIM utasítás 3. és 4. bitjét még nem tudjuk mire jó.
Lehetőséget kell biztosítani arra, hogy a folyamat elindulása előtt RST 7.5 esetén azt tudjuk mondani, hogy mostantólfogadjuk a 0-1 átmeneteket.
A SIM utasítás 4. bitje ha 1, akkor a 7.5 flip-flopot töröljük.

Tipikus IT rutin:

```
ITRUT:  PUSH PSW (mentenünk kell az akumulátor értékét)
        PUSH … (ha más regisztereket is használunk, akkor azokat is mentenünk kell)
        …
        „IT kiszolgálása”
        …

        POP …  (fordított sorrendben visszatöltjük a regiszterek tartalmát)
        POP PSW
        EI
        RET
```

Az adatfeldolgozást sose IT rutinon belül végezzük (csak nagyon ritkán, ekkor azonban gondoskodnunk kell arról, hogy más IT be tudjon következni, az IT rutin legyen újrahívható). IT rutin pár utasítás hosszú legyen.

Az IT rutin megírására nem sok helyünk van.
0 címen az RST 0 utasítás helyezkedik el.
RST1 a hexa 8 címen helyezkedik el.
RST2 a hexa 10 címen van.
…
A TRAP (RST 4.5) a hexa 24 címen helyezkedik el.
Az RST5 a hexa 25 címen helyezkedik el.
Az RST 5.5 a hexa 26 címen helyezkedik el.

A TRAP utasításhoz mintegy 4 byte helyünk van.
Ide csak egy JMP TRAPIT utasítást tudunk elhelyezni.

EI
RET
utasítássorozattal védekezhetünk az ellen, hogy ne bekövetkezzen az IT. Ekkor nincs JMP, csak visszatérés.

Mi az utolsó IT?
Ugrási tábla (feles interrupt?)
Legalacsonyabb prioritás.
Nem maszkolható.
Csak a teljes IT blokkolásával blokkolhatjuk.

Ekkor egy INTA gépi ciklus indul, ami egy FETCH típusú gépi ciklus.
Körülbelül 40 fázisnyi időnk van arra, hogy a külvilág IT-jét eltudjuk kezdeni kezelni.

Trükk:
Nézzük a következő utasítássorozatot:

…
STC
VAR: JC VAR
…

Ez egy végtelenciklus.

INTA ciklusban ORA A utasításkódját adjuk fel.
Ez állít egy flag-et, és mi erre a flag állításra várunk.

Másik lehetőség, hogy hívjunk szubrutint.
Melyik szubrutinhívás kódját adjuk fel?
A CALL utasítás esetén 3 INTA típusú gépiciklus lesz. Nem mindegy, mi mikor történik.
Van nekünk azonban 1 bájtos szubrutin hívó utasításunk is, az RSTn is.
Ez az n*8-as címre adja a vezérlést.Olyan bájtot kér, ahol középen találjuk a sorszámot, előtte és utána pedig 1esek állnak.

Ehhez egy kombinációs hálózat is elég.

8 IT lehetőségünk van. Ezekből szeretnénk kiválasztani és előállítani a megfelelő kódját.
Erre tökéletes egy enkoder, melynek 8 bemenete van.
74148 prioritásenkoder:
EI engedélyező.
8 adatbemenet (0, 1 … 7)
3 kimenet (A2, A1, A0)

Működés:
Ha EI-n HI van, akkor minden kimenet HI értékű.
Ha EI engeédlyezve van (=0) de nincs aktív bemenet, akkor minden kimenet HI.

A 7-es bemenethez tartozik a 000 kimenet.
A 6-os bemenethez a 001 kimenet tartozik.
…
Ttehát negált logikával működik!
A legmagasabb prioritás a 7-es.

Egy ilyen segítségével valósítjuk meg az IT-t.
INTA jel engedélyezi.
A bementére invertált jeleket kell kötnünk.

Az IT7 az RST0 kódot jelenti.
Az IT0 az RST7-et jelenti.

Eddig 74244-et használtunk.
74240-el (invertáló erősítő) esetén pont fordul, de a prioritás nem változik.

Ez egy hardveres interrupt eldöntő készülék lett.

Létezik IT kezelő áramkör. 8259.

Ezeket kaszkádosítva (9 darab) 64 IT-t kezelhetünk.
Nem nehéz megoldani, hogy az IT-k prioritása állítható legyen.
Erre forgó prioritásbeállítási lehetőségünk van a vezérlőkben.
A 8259-es támogat élvezérelt és él+szint vezérelt módot is.

8259-es képes együttműködni mai PC-vel. Funkcionalitását tekintve megegyezik a mai IT vezérlőkkel.

Adatmozgató utasítások:

Közvetlen adat:

MVI C, 32	C regiszterbe tölti a 32 értéket.
LXI H, 9000h	H regiszterbe mozgatja a 9000h értéket.

Közvetlen adatcímzés:

LHLD 9000h	A HL regiszterpárba tölti a 9000h címtől kezdve az adatokat.

Indirekt regisztercímzés:

STAX D	
MOV A,M	Az M (=HL) regiszterpár által mutatott értéket bemásolja az akkumulátorba.

Regisztercímzés:

MOV A,C	A C regisztert bemásolja az akkumulátorba.

Aritmetikai utasítások:

DCR C	
INR B	
DCX D	Flageket nem állítja.
INX H	Flageket nem állítja.

Logikai utasítások:

ORA
ANA
XRA

Vezérlés átadó utasítások:

Feltétel nélküli:

JMP
CALL

Feltételes:

Jcc	cc szerint ugrik. Például JNZ ha a zero flag 0, akkor ugrik.

MP:
9000h memóriacímtől 32 darab szót szeretnénk másolni 9800h-ra.

Kell egy forrás mutató és egy célmutató:

```
LXI H, 9000h    (A HL regiszterpárba másolja a 9000h címet.)
LXI D, 9800h    (A DE regiszterpárba másolja a 9800h címet.)
MVI C, 32       (Ez lesz a ciklusváltozó.)

Ciklus:
                //A szó első felének átmásolása
MOV A, M
STAX D
INX H
INX D
                //A szó második felének átmásolása
MOV A, M
STAX D
INX H
INX D

DCR C           (A ciklusváltozó csökkentése.)
JNZ ciklus      (Ha a ciklusváltozó elérte a 0-t, akkor nem ugrunk vissza.
```

MP:
9000h.tük 512 bájt másolása 9800h-ra:

```
LXI H, 9000h
LXI D, 9800h
LXI B, 512      (Itt a különbség, a BC regiszterpárra hivatkozunk, így 16 bit is elfér.)

Ciklus:
MOV A, M
STAX D
INX H
INX D

DCX B           (A BC regiszterpár értékének csökkentése.)
                // NEM állítja a flageket, kell egy ötlet.
                // Hozzuk OR kapcsolatba a B és a C regiszterpárt. DE fel kell használnunk az akkumulátort.

MOV A, C        //A C regiszter értékét az akkumulátorba visszük.
ORA B           //Az akumulátor értékét, ami megegyezik a C értékével összehasonlítjuk B-vel. Ez a flegkeet
                // is beállítja, így már ugorhatunk a zero flag szerint.

JNZ ciklus      (Ha a ciklusváltozó elérte a 0-t, akkor nem ugrunk vissza.)
```

MP:
Másoljunk bizonytalan hosszú sztringet a 9000h-ról a 9800h-ra.

```
LXI H, 9000h
LXI D, 9800h

Ciklus:
MOV A,M
STAX D
INX H
INX D
ORA A          (Hozzuk az akkumulátort AND vagy OR kapcsolatba magával. Ha 0, akkor ez beállítja a zero flaget.)

JNZ ciklus
```

MP:
Írjunk szubrutint, amely keres a megadott címtőla sztringben egy a betüt.
A carry flag legyen 1, ha talált, és 0, ha nem talált a betűt.

```
Keres:                  //Kell egy azonosító:
LXI H, 9000h

Ciklus:
MOV A, M
ORA A
JZ nem
CPI 'A'                 //Az akkumulátor értékéből kivonja az 'A' konstanst, és beállítja a flageket. A zero flag beáll
                        // 0-ra, ha az összehasonlítás eredménye az, hogy egyeznek. Carry-t is beállítja.
JZ talalat
INX H
JMP ciklus

nem:
STC             //STC carry-t beállítja 1-esre.
talalat:
RET
```

MP:
9000h-tól van egy 16 bites szavunk.
9100h-tól van egy másik 16 bites szavunk.
Adjukössze ezeket, és az eredmény legyen a 9000h címtől.
Intelnél alacsonyabb helyiérték az alacsonyabb címen van.

```
LXI H, 9000h
LDA 9100h

//A carryt ki kell nulláznunk, mert még nem volt összeadás.
//Van két ADD utasításunk azonban, ADD nem használja a carryt, ADC pedig használja. Így nem kell a carryt
nuláznunk külön.

ADD M
```

```
MOV M, A        //Békénhagyja a flageket. A carry jó értéken áll.
INX H
LDA 9101h
ADC M
MOV M, A
```
Alternatív megoldás:
```
LHLD 9000h
XCHG
LHLD
DAD D
SHLD 9000h
```

MP:
9000h-tól, 32 byte-ra modulo 256 ellenőrzőösszeg.

```
LXI H, 9000h    //A cím, amivel dolgozunk.
MVI C, 32       //Ciklusváltozó
XRA A

Ciklus:
ADD M
INX H
DCR C
JNZ Ciklus

                //A ciklus lefutása után az akkumulátort tartalmazza az ellenőrzőöszeget.

MOV M,A         //Elrakjuk a 32 byte utánra az ellenörzőösszeget
```

MP:
Számoljuk meg, hogy hány darab 1-es van 32 byteon.

```
LXI H, 9000h
MVI C, 32
LXI D, 0

Ciklus1:
MVI B, 8
MOV A, M

    Ciklus2:
    RLC             (Helyben rotálja a 8 bites számunkat, beállítja a carryt.)
    JNC Ciklus3
    INX D

    Ciklus3:
    DCR B
    JNZ Ciklus2
    DCR C
    JNZ Ciklus 1
```

MP:
Írjunk ki ASCI karaktereket:

```
bin2hexa:
PUSH PSW      //Akkumulátort menti a stackbe.
ANI 0Fh       //Bitenként összeéselünk 00001111 értékkel, így eltűnik a felső 4 bit.
ADI 30h       // Adjunk hozzá 30 hexát, hogy jó értéket kapjunk.
CPI 3Ah       //Ha 9-nél kisebb számunk van, akkor negatív értéket kapunk.
JC Ciklus1    // Ha az összehasonlítás után a carry 1, akkor elugrunk.
ADI 7         // Ha A betüt kell már kiírnunk, akkor 7 értéket hozzá kell adnunk.

Ciklus1:
MOV C, A
POP PSW
RLC
RLC
RLC
RLC
ANI 0Fh
ADI 30h
CPI 3Ah
JC Ciklus2
ADI 7
Ciklus2:
MOV B, A
RET
```

interruptok folytatás:

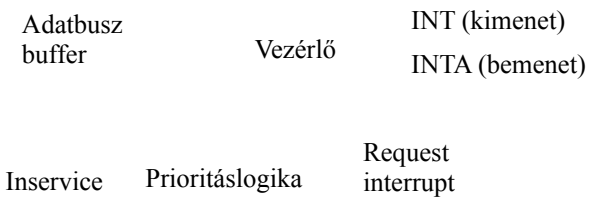
8259-es:

Első körben 8 iIT fogadására alkalmas.
Kaszkádosítható, 9 darabbal 64 IT kezelhető.
Prioritás állítható.

Megadhatóak szubrutin címek. De mennyire rugalmasak ezek?
Hogyan tehetjük meg?

Ha mind a 8 külön meghatározható lenne, 16 byteot kéne tárolnunk, és mindet címeznünk is kéne.
Ehelyett ugrási táblát használ. Megadható a kezdőcím, és utána minden címmegadott távolságra van.

Blokkvázlat:



mask

Kaszkád logika

R/ W
Logika

Hogy néz ki egy intterrupt kezelés:

1. ITn 0 → 1
   IRn → 1

2. Prioritás + Maszk → INT= 1
3. Ha a CPU-n fennállnak a feltételek, /INTA = 0

4. CALL kódja (0CDH) kerül a sínre.
   ISn = 1, IRn=1

5. A következő 2 INTA ciklusban a szubrutin cím alsó és felső byte-ja kerül kiadásra.

6. ISn = 0

   -Automatikusan
   -Programból

Kaszkádosítás:

1 master, több (max 8) slave.

A master dönti el, hogy melyik slave szólaljon meg.

A probléma a kaszkádosítással akkor van, ha nem egy darab közös adaterősítő van.
Erre jó az SP/ /EN jel. Ez vezérli az adaterősítő G bemenetét.

Ha nincs 8 darab slave, akkor az IT0 vezetékre TILOS slavet kapcsolni.
Amikor a master által elfogadott interrupt kerül kiszolgálásra, akkor a kaszkád vonalak 0 állapotban vannak, ilyenkor az itt álló slave nem tudná, hogy neki, vagy a masternek kell dolgoznia éppen.

Programból:
Soros IT rutin.

Elkezdődik egy IT, befejezés előtt közvetlenül engedélyeződik csak az interrupt (EI parancs), visszatér, és jöhet a következő IT.

Egymásba ágyazott IT:
Elkezdődik az IT, majd valahol azt mondja, hogy ha van nála magasabb prioritású IT, akkor azt engedi érvényrejutni.
Ekkor azonban a RET parancs előtt EOI (end of intterrupt) parancsot kell kiadnunk, hogy tudjuk, hogy mikor van vége az IT kiszolgálásának.

Ebben az esetben fix prioritású rendszerben „kiéheztetés” történhet. Alacsony prioritású IT nem juthat érvényre, ha IT-vel terhelt a rendszer.
Erre megoldás a forgó prioritás, melyet a prioritáskezelő áramkörünk ki is tud szolgálni.

Autómatikus EOI esetén, szintérzékeny IT esetén saját magát is megszakíthatja a szubrutin.

Nézzünk egy olyan esetet, ahol a master 3. lábára van kötve egy slave.
Amikor elindul az IT kiszolgálás, a masteren a 2 és nagyobb indexű lábak blokkolódnak, és ha az IT a slvae 3-as lábán jön, akkor a 3 és feletti lábak blokkolódnak. Ekkor viszont a slave a 0-s lábán hiába kapna IT-t például, a masterig eljutna, de a masternek ezen lába tiltva van. Borul ilyenkor a sorrend.
Erre egy úgynevezett rögzített prioritási mód a megoldás, amit nekünk szftverből kell állítanunk.

A0 = 0 ICW1 vagy OCW2, 3 parancsunk van.
A0 = 1 ICW 2, 3, 4 vagy OCW 1 parancsunk van.

Inicializálás:
-Szubrutintábla címe
-Él vagy szintérzékeny bemenet
-Master-slave konfiguráció?
-EOI autómatikus vagy nem?
-Buffereljünk vagy sem? (1 vagy több adaterősítőnk van-e)
- Milyen mikoprocesszoros rendszerben vagyunk?

ICW1 A0 = 0

A7	A6	A5	1	LTIM	ADI	SINGLE	IC4

IC4: ICW4 is lesz
SINGLE: Master/lave vagy sem, ms esetén ICW3 is lesz
ADI: Ugrótábla gap, 4 vagy 8 byteos az ugrótábla eltolása
LTIM: szint vagy élérzékeny

A7...A5: az ugrótábla alsó byte-ja.

ICW2 A0=1

A15							A8

Megadhatjuk az ugrótábla felső byteját.

ICW3 A0=1

MASTER: hol vannak a slaveek (bitminta)
SLAVE: Melyik master bemenetre csatlakoznak.

ICW4 A0=1

			S	MS	BAF	AEOI	MP

MP: milyen mikroprocesszoros rendszerünk van (85/86)

AEOI: automatikus end of interrupt
BAF: Bufferelt üzemmód
MS: Master/slave
S: speciális prioritás

OCW1 A0=1

Maszk állítása

Azokra a helyekre kel 1 értéket írnunk, amiket nem szeretnénk, hogy érvényre jussanak.
ICW1 után a maszkregisztert újra kell írnunk.

OCW2 A0=0

Prioritás állítható ezzel a paranccsal.

			0	0			

A 00-ból ismeri fel, hogy OCW2-es parancsunk van.

Parancsok:
EOI A legmagasabb prioritású IT-t befejeztük. Hogy ez melyik volt, azt csak a 8259 tudja
Specifikus EOI Sorszámban megmondhatjuk, hogy melyiket fejeztük be.
EOI és forgatás Befejeztük a legmagasabb prioritású IT-t és szeretnénk forgatni (a mostani lesza legalacsonyabb prior)
AEOI forgatásengedély
AEOI forgatástiltás
specifikus EOI és forgatás Amit befejezünk, az lesz a legalacsonyabb rpiroitás.
Prioritásbeállítás.

OCW3:
IR
IS regiszterolvasás

POLL
speciális maszkolás
Prioritást tudjuk felülírni, azt mondhatjuk, hogy jöhet a következő IT, aminek mindegy a prioritása.

MP:
Illesszünk kapcsolót I/O eszközként.

A kapcsoló bal oldala legyen föld, a jobb oldala legyen felhúzva tápfeszültségre.

Illesszük a 0000h címre.

Kell egy 9 bemenetű OR kapu, SA0 … SA7 és egy IO/ /M, de ez utóbbit negálnunk kell egy inverterrel.

Az /IOR jelnek is hasznát kell vennünk.

Ezt az előző OR kapu kimenetével kössük egy OR kapura.

A kapcsoló jelét egy three state kimenetű bufferen keresztül kell vezérelnünk az előzőleg előállított jellel.

A /RDY a nagy OR kapu kimenete.

MP:
Programmal ellenőrzött készenlét:

var1:
IN 00h // Beolvassuk a gomb állapotát.
ANI 80h //Maszkoljuk az akkumulátort
JZ var1 //Az ANI beállítja a zero flag-et. Ha az eredmény 0, akkor mégmaradunk a ciklusban.
RET

Gondoljunk a késleltetésre is. A szubrutinunk ~10us körül fut le. A nyomógombot 40ms-nél gyorsabban nem nyomhatjuk meg, így ezzel nem lesz baj.

MP:
Legyen két gombunk, a két legnagyobb helyiértéken. Csak akkor ugrunk, ha mindkettő le van nyomva:

var2:
IN 00h
ANI 0C0h
JNZ var2
RET

MP:
Két gomb közül kell választanunk, hogy melyiket nyomták meg.
A két gomb a legalacsonyabb helyiértéken van.

Var3:
IN 00h //Beolvasunk
ANI 03h //Alsó helyiértékeken vannak az adatok, többit maszkoljuk.
JZ var3 //Ha 0, akkor még nem csinálunk semmit.
RRC //Akkor érünk ide, ha nem 0. Jobbra forgatjuk a biteket, ekkor az alsó bekerül a carry-be is.
JC also // Carryt vizsgáljuk , az előbb jól beállt.
ANI 01h // Most az 1-es értkkel maszkolunk, mert csak az utolsó bit érdekel.
JZ var3 // most már használhatjuk a zero flaget.

felso:
…

also:
...

MP:

Definiáljunk egy ugrótáblát:

Jtab: DW cim1, cim2, cim3

Fontos, hogy ezek hogy tárolódnak, mert 16 bitesek, azonban a memóriában 8 bites blokkok vannak.
Az alsó 8 bit az alacsonyabb memóriacímen van.

Var4:
IN 00h
ANI 03h

```
JZ var4
DCR A
ADD A
LXI H, Jtab
MVI D, 0
MOV E, A
DAD D
MOV E,M
INX H
MOV D,M
XCHG
PCHL
```

```
MP:
LXI H, 1122h      //HL regisszterpárba rakjuk az 1122h értéket.
PUSH H            //Stack-be mentjük az értéket.
CALL Rutin
POP H

Rutin:
LXI H, 2h
DAD SP
MOV E, M
INX
MOV D, M
…
RET
```

Ha azt szeretnénk, hogy visszatérés után jók legyenek a regiszterek értékei:

```
Rutin:
PUSH PSW
PUSH B
PUSH D
PUSH H
LXI H, 0Ah
DAD SP
MOV E, M
INX
MOV D, M
…

POP H
POP D
POP B
POP PSW
RET
```

MP:
Konstans paraméterek átadása:

```
ORG 20h
CALL Rutin
DB 23h, 24h
HLT
ORG 100h
XTHL
```

```
MOV E,M
INX H
MOV D,M
INX H
XTHL
…
RET
```

Visszatérési értékek előállítása:
Legegyszerűbb: processzor flag-jét állítsuk, és azzal jelezzünk.

Rutin:
```
PUSH D
PUSH PSW

…

C0:
POP PSW
STC             //Carry 1-re áll.
CMC             //Carry megfordul.
POP D
RET

C1:
POP PSW
STC
POPD
RET

MP:
Z1:     //Akku nem ronthatjuk, falgeket elronthajuk, z legyen 1.
POP PSW         //Akkut és flageket visszahozzuk.
MOV E, A        //E-be rakjuk az akkumulátort
XRA A           //ANI 2 byte, XRA csak 1
MOV A,E         //Visszaállítjuk az akkut
POP D           //Visszaállítjuk a DE-t.
RET

Z0:
POP PSW
MOV E, A
ORI 01h
MOV A, E
POP D
RET

MP:
Z00:            //Akkut nem ronthatjuk, flageket nem ronthatjuk, csak zero-t állítsuk 0-ra.
POP D           //A PSW-t a DE-be hozzuk fel. Ekkor az E regiszterben vannak a flag-ek.
MVI A, 0BFh     //A-ba visszük a maszkot.
ANA E           //Kimaszkoljuk a zero flag-et, ami a 6. a flagek közül.
MOV E, A
PUSH D          //Dt visszarakjuk.
POP PSW         //Felhozzuk mint PSW, így llnak be a flagek.
POP D
RET
```

```
Z11:
POP D          //A PSW-t a DE-be hozzuk fel. Ekkor az E regiszterben vannak a flag-ek.
MVI A, 40h     //A-ba visszük a maszkot.
ORA E          //Kimaszkoljuk a zero flag-et, ami a 6. a flagek közül.
MOV E, A
PUSH D         //Dt visszarakjuk.
POP PSW        //Felhozzuk mint PSW, így állnak be a flagek.
POP D
RET
```

DMA (Direct memory access)

Képzeljünk el egy egyszerű sínt, amire csatlakozik egy CPU, egy memória és egy periféria.

Első körben ne törődjünk a perifériával.
A master a CPU lesz, míg a slave szerepet a memória tölti be.
Forrás és cél is lehet a CPU és a memória is.

Ha a perifériából szeretnénk behozni adatot a memóriába, akkor felvetődik egy kérdés, hogy mi legyen a master.

Nézzünk egy mintapéldát, hogyan hozhatnánk be adatot a perifériából?

```
Ciklus:
IN STAT          //Beolvasunk valami státuszjelzőt.
ANI VANADAT      //Valami megfelelően megválasztott konstans.
JZ Ciklus        //Ha a státusz nincs készet jelez, akkor ugrunk.
IN ADAT          //Adat behozása
MOV M, A         //Az akkuból a memóriáab tesszük az adatot.
INX H            //HL regiszterpár értékét növeljük
DCR C
JNZ Ciklus
```

Mennyi ideig tart 1 bájt átvitele? Nagyjából 75 fázis, azaz 25mikrosecundum.

Nézzük hogy néz ki egy másoló ciklus IT-vel:

```
JMP ITRUTIN
…

ITRUTIN:
PUSH PSW
PUSH H
LHLD BUFFCIM
POP H
POP PSW
EI                    //Flip-flpok tiltódtak, engedélyeznünk kell visszatérés előtt.
RET
```

Nézzük mire is jó a DMA:

A sínen megjelenik egy új master a DMA vezérlő, amely arra kényszerítheti a processzort, hogy az engedje el a sínt, és ekkor a DMA vezérlő fogja generálni a címet, és ő fogja meghatározni az adatmozgatásokat is.

A forrás ilyenkor a periféria lesz, míg a cél szerepét a memória tölti be.

A vezérlés trükkös lesz, a memória írás jelet kap, a periféria pedig olvasás jelet kell kapjon.
A címsinen mindig memóriacím lesz, a DMA a perifériát külön engedélyezi.

Ehhez szeparált vezérlőjelekre lesz szükségünk.
DRQ – DMA kérelem jelzés
DACK – DMA elfogadása

Az adatátvitel időzítését az /IOR és /IOW jelek fogják meghatározni.

Az általunk használt DMA vezérlő a 8257-es lesz.

8257:

Van neki egy adatbuffere, D0 … D7 ki-bemenetekkel

Van egy R/W logikája, amelyen ki-.bemenet az /IOR és /IOW
Órajellel működik, tehát van egy CLK bemeneb
Van egy reset bemenete és 4 címvezeték A0 … A3
Van egy /CS chip-select vezeték

A vezérlőn A4 … A7 címvezeték található, melyek kimenetek
Van egy Ready bemenet.
HRQ kimenet.
HLDA bemenet.
/MR és /MW
AEN kimenet.
ADSTB kimenet
MARK és ITC kimenet.

4 csatornát képes kezelni. Minden csatornának külön van DRQ bemenete és /DACK kimenete.

Van prioritáskezelő.

Szoftvermodell:
4 darab 16 bites címregiszter, ami azt tartalmazza, hogy a memóriába hova szeretnénk rakni az első adatot.
Van 4 darab 16 bites számláló (terminal count) regiszter, amely 14 biten számlálja a byteokat, és a két legmagasabb helyiértéken lévő bitje 3 darab üzemmódot határoz meg.

TC15 TC14
00 vizsgálati ciklus
01 memóriába ír a perifériából
10 perifériára ír a memóriából
11 tiltott

1 darab MODE SET regiszter, mely 8 bites és vezérlést határoz meg:

7	6	5	4	3	2	1	0
AL ENG	TC STOP	KI ENG	FP ENG	CH3 ENG	CH2 ENG	CH1 ENG	CH0 ENG

Fix prioritás: CH0 a legmagasabb prioritás.
Forgó prioritás (FP): körbeforgat
Kiterjesztett írás (KI)
TC stop
AUTO LOAD (AL): kizárlóag a kettes csatornára

A MODE SET regiszter csak írható.
Ezen a címen olvasás paranccsal elérhető a STATUS byte.

7	6	5	4	3	2	1	0
0	0	0	Update flag	CH3 TC bit	CH2 TC bit	CH1 TC bit	CH0 TC bit

Update flag: autoload funkcióhoz való.

Belső regiszterek címzése az A0 … A3 vezetékeken:

A3 A2 A1 A0 Flip-flop
0000 $^{0}/_{1}$ CH0 címregiszter
0001 CH0 TC
0010 CH1 cím
0011 CH1 TC
0100 CH2 cím
0101 CH2 TC
0110 CH3 cím
0111 CH3 TC
1000 0 W: MODE SET
1000 - R: STATUS

A szeparált vezérlőjeleket egy dekóderrel tudjuk megvalósítani.
A processzor által kiadott jeleket kell egy egnedélyezett dekóderre kötnünk.

DE: a kimenetre kell még egy three-stat meghajtó, amelyet az AEN jel vezérelhet.

Gyak:

RIM:

SID	RST 7.5	RST 6.5	RST 5.5	INTE f-f	RST 7.5	RST 6.5	RST 5.5

Maszk bitek

Bemeneti értékek.

INTE f-f értékét kaphatjuk. Ez akkor fontos például, ha TRAP megszakítást használunk, ekkor ugyanis ugyan úgy kell visszaállítanunk az INTE ff- értéket, ahogy az a megszakítás előtt állt.
A CPU ebbe a bit-be menti az INTE f-f értékét.

SIM:

SOD	ESOD	-	CRST 7.5	ME	RST 7.5	RST 6.5	RST 5.5

SOD állítás engedélyezése
Maszk bitek beállítása (0 engedélyez)
Maszkolás engedélyezése
RST 7.5 flip-flop törlése
Bemeneti értékek.

MP:
Használjuk a SID bemenetet és a SOD kimenetet arra, hogy a SID bemenetre illesztett pergésmentesített nyomógomb bekapcsolja a Hifi készülékünket.
Azt szeretnénk, hogy a nyomógomb megnyomásakor a hifi kapcsoljon be, és amikor újra megnyomjuk a gombot, akkor

pedig az elengedés pillanatában kapcsoljon ki.

A működés egy sorrendi hálózattal írható le.
T1 állapotban vagyunk amíg a SID bemeneten 0 van. SID=1-re S2 állapotba ugrunk és a SOD kimenetet 1-re állítjuk.
SID=0-ra továbbugrunk S3-ba és addig maradunk, amíg a SID=0.
SID=1-re továbbugrunk T4 állapotba, és addig maradunk, amíg SID=1.
A SID=0 érték megérkeztekor T1 állapotba ugrunk.

Hogy néz ki a szoftver:

```
ORG 0000h           //0-s címtől indulunk

C0:
MVI A, 40h          //SOD-ot 0-ra állítjuk, ehhez az accuba kell vinnunk a 40h értéket.
SIM

C1:
RIM
ANI 80h
JZ C1
MVI A, 0C0h
SIM

C2:
RIM
ANI 80h
JNZ C2

C3:
RIM
ANI 80h
JZ C3

C4:
RIM
ANI 80h
JNZ C4
JMP C0
```

MP:
Tervezzünk lopásgátlót egy autóba.

Az RST 5.5 bemenetre illesszünk egy pergésmentesített nyomógombot a SID mellé.

A működés legyen a következő:
Ha az RST 5.5-ön 1-es érték van, akkor számoljuk a felfutó éleket a SID bemeneten. Ha pontosan 256-szor nyomták meg a SID bemenet nyomógombját, akkor a SOD kimenet 1-re kapcsoljon, és az autó indítható legyen.

Ehhez már regisztereket is kell használnunk.

B: előző SID
D: aktuális SID
E: aktuális RST
L: számláló

Programkód:

```
ORG 8000h
```

```
INIT:
RIM             //Beolvassuk az értékeket.
ANI 80h         //Akkuba megy a kimaszkolt érték
MOV B, A        // Mentjük a B-be.
MVI A, 4Fh
SIM                     //Lemaszkoltuk az összes utasítást, és a SOD-ot 0-ba állítjuk
XRA A           //
MOV L, A        //Kinulláztuk a számlálót

START:
RIM
PUSH PSW        //Akkumulátor értékének mentése a stack-be.
ANI 10h         //Kimaszkoljuk az RST 5.5-öt, minden más törlődik
MOV E, A        //E-be mentjük az RST 5.5-öt
POP PSW
ANI 80h         //Maszkoljuk a SID-et.
MOV D, A        //SID-et mentjük D-be.
XRA B           // Megnézzük, hogy változott-e
JZ START        //
XRA A
ORA B
JZ UP

DOWN:
MOV B, D
JMP START

UP:
XRA A
ORA E
JZ DOWN
DCRL L
JNZ DEL         //Ha
MVI A, 0c0h
SIM
JMP DOWN

DEL:
MVI A, 40h
SIM
JMP DOWN

END
```

Hogy csinálhatnánk ezt meg megszakításokkal?

Megszakításhoz mindig van valamilyen megszakítási szubrutin.

```
ORG 0000h
JMP INIT
ORG 2Ch                 //RST 5.5 belépési pontja, fontos!

RUT:
DCR L
JNZ DEL
MVI A, 0C0h
SIM
RET

DEL:
MVI A, 40h
```

```
SIM
RET


INIT:
LXI SP, 0          //Stack pointer értékét beállítjuk a memória legvégére feltételezve, hogy ott a RAM
RIM
ANI 80h
MOV B, A
MVI A, 4Eh
SIM
XRA A
MOV L, A

START:
RIM
ANI 80h
MOV D, A
XRA B
JZ START
XRA A
ORA B
JNZ DOWN

UP:
EI
NOP
DI

DOWN:
MOV B, D
JMP START


END
```

Perifériák:
Információátvitel: 1bit
Párhuzamos átvitel: 8bit

Nagy távolságokon általában soros átvitelt valósíthatunk meg, mert a párhuzamos átvitel esetén a zaj miatt elveszik a jel.

Adatátvitel vezérlése:
1. nincs vezérlés (INPUT esetén a pillanatnyi érték, pl kapcsoló, OUTPUT esetén következő értékig megmarad, de a külvilág itt csak a változást érzékelheti, ekkor a kimenetre egy latch szerű eszközt kell helyezni)
2. Handshake jelek állnak rendelkezésre az adatokon kívül. Ekkor valaki adó áramkör és valaki vevő.
    Adó: data ready jel
    Vevő: acknowledge jel

    Az adatok és más jelek időzítését mindig az adott protokol határozza meg.

8255-ös néven fejlesztettek a 8085-ös processzorhoz PIO áramkört, azaz párhuzamos ki és bemenetet biztosító áramkört.
A 8255-ös lábai:
DATA BUS BUFFER: 8 adatvezetéken kommunikál a processzorral.

R/W logika:

/RD és /WR bemenetek, melyekkel a processzorral tud
A0, A1 címvezetékek, tehát 4 címen látszik (256 címünk van perifériáknak)
/CS: a chip kiválasztására szolgál
/RESET: alaphelyzetbe állítás.

PA és PB portok:
Együtt programozhatóak, megmondható, hogy az A és B port egyenként be vagy kimenet legyen.

PC port:
4-4 bitenként állítható, hogy kimenet vagy bemenet legyen.

A portok vezérléséhez egy A és B vezérlő áramkör áll rendelkezésre, melyek a C portot félig-félig kezelik.
Ezeket a vezérlőket az R/W logika vezérli.

Alapállapotban az eszköz bemenetként áll be, így nem bántja a külvilágot.

3 üzemmódja van az eszköznek:
MODE 0: egyszerű I/O, a kimeneti portok megtartják a rájuk írt adatot latch szerűen, a bemeneti portok a pillanatnyi értékeket vizsgálják.

MODE 1: stróbolt I/O
INPUT: adó a periféria DATAREADY jel (STB), vevő a 8255 ACK jel (IBF – input buffer full)
A kezelés két módon lehetséges, programmal ellenőrzött készenléttel vagy interrupt-tal.
IT-t kérni csak akkor szabad, ha az STB jel lefut. Ekkor már a jó adatot olvastuk ki, és ha nagyon gyors a kiszolgálás, akkor sem jelent problémát a szintérzékenység, mert nem kér újabb IT-t.

Az STB jel rendszerint negált logikájú. Az IBF ennél az áramkörnél ponált, azonban a perifériák általában negált logikával kérik, így majd invertert kell használnunk sok esetben.

OUTPUT: adó a 8255DATAREADY jel (OBF – output buffer full), a vevő a periféria, aki ACK jellel (ACK) válaszol.
Mikor is kell IT-t kérni? A processzor a forrás, de a külvilágnak kéne IT-t kérnie. Ezt a külvilágnak akkor kell megtennie, amikor kész adatot fogadni.
Ekkor azonban alapállapotban interrupt van. De nekünk nem mindig van mit kiküldeni a perifériára.
Az IT-ttehát tudnunk kell maszkolni. Amikor ki akarunk vinni adatot, akkor elvesszük a maszkot, hogy az IT érvényre juthasson.

Az OBF és ACK jelek negált logikájúak ebben a rendszerben.

MODE 2: 2 irányú stróbolt I/O
Egy vezetéken tud oda-vissza kommunikálni a periféria és a 8255 hardveres hand-shake jelekel.

Jegyezzük meg:
0-s MODE-ban minden port használható.
1-es MODE-ban A és B port használható, de a C port egyes bitjei ekor nem használhatók. (2-es MODE-hoz 3 Cport bit kell portonként).

2-es MODE-ban csak az A port használható. A B portot még használhatjuk 1-es MODE-ban, ekkor nincs több C port bitünk.

Nézzük mi történik ha kimenetként vagy bemenetként programozzuk az áramkört:

PC	INPUT	OUTPUT	MODE2
0	INTR(B)	INTR(B)	-
1	IBF(B)	OBF(B)	-
2	STB(B)	ACK(B)	-
3	INTR(A)	INTR(A)	INTR(A)

4	IBF(A)	-	STB
5	STB(A)	-	IBF
6	-	ACK(A)	ACK
7	-	OBF	OBF

A vezérlőszó:

Nem visszaolvasható, tehát felprogramozás után nem tudjuk visszaolvasni az értéket.

PA
PB
PC CONTROL

üzemmód
Bit SET/RESET

Egy regiszterhez két funkciót rendelünk:
A 8255-ös nem szekvenciában programozható, az adat értéke dönti el, hogy mit programozunk.

Ha a legmagasabb helyiérték 1-es, akkor mode-ot állítunk.

Beállíthatjuk az A, B, C portok irányát (bev vagy kimenet) és a módját.
Üzemmódok az A portra 0, 1, 2, a B port 0, 1, a C portra nem állíthatunk mode-ot.
C porton az irányt 4 bitenként mondhatjuk meg, külön az alsó 4-re és külön a felső 4-re.

Hogy mit melyik bittel érünk el, azt a segédletből olvashatjuk ki.

Ha a legmagasabb helyiérték 0-s, akkor Bit SET/RESET funkciót érjük el:

A C port bitenként címhezhető. A regiszter alsó 4 bitje állítja ezt, az utolsó a kivivendő érték, a másik 3 pedig a cím.
Erre eddig beolvasást, és bitenkénti vagyolást használtunk, de nem biztos, hogy beolvashatóak az értékek, ezért van szükség a bitenkénti címezésre és állításra.

Nézzük mit olvashatunk be:

PC	Státusz
0	INTR(B)
1	IBF(B) / OBF(B)
2	INTE(B)
3	INTR(A)
4	INTE(A) in
5	IBF(A)
6	INTE(A) set
7	OBFA

A soros átvitel 2 vezetékével szemben egy 8 bites párhuzamos átvitel 9 vezeték, ami elszomorít minket.

SOROS ÁTVITEL:

A hardverrel szemben nagyobb elvárás.
Soros-párhuzamos átalakításra shift-regiszter.

BaudRate (bit idő) meghatározza, hogy másodpercenként hány bitet viszünk át.
Kezdetben 75 bitnyi információt lehett átvinni.
A szabvány szerint a 75 kettő hatványaival vannak szorozva.
A szabványos soros átvitelnek nagyjából 115200 bit/sec körül van vége.

Nincsenek hand-shake jelek. Valamilyen módon meg kell találnunk az első bitet.
Erre két módszerünk van: aszinkron és szinkron adatátvitel.

Tudnunk kell, hogy hol van az adatcsomag eleje és vége.
Ehhez kereteznünk kell.
5 és 8 bit közötti adatot vihetünk át egyszerre.

Aszinkron átvitel keretezése:
START bit: 1 bitidőnyi időre az adatvezetéket 0 állapotba tesszük. Az adatátvitelt az első 1-0 átmenet indítja.
Innentől 5-8 adatbitnyi információt viszünk át pontosan 1 bitidőnyi ideig látszik minden bit.
A szabvány szerint a legalacsonyabb helyiértékű bitet visszük át először.
Hogy vehetjük észre, ha hibádzik egy bit?
Ellenőrző bitekkel.
Hozzáadhatunk az adathoz egy paritásbitet, mellyel 1 bit megváltozását észlelhetjük.
A paritás állítható általában.
STOP BITEK:
Garantáltan egyes értékek, beállítható a számuk, egy vagy kettő. Garantálják, hogy a következő bit 1-0 átmenettel kezdődjön (START bit).

PL:
9600, 8, N, 1
9600 baud rattel viszünk át 8 bites adatot, nincs paritás, 1 stop bit van.

1200, 7, E, 2
1200baudrate, 7 bit, Even paritás (páros → 1), 2 stop bit.

Nézzük a tényleges sebességet az első példa esetén például:
1 startbit, 8 adatbit, 0 paritásbit és 1 stopbit, 10-ből csak 8 bit használható.
1000 szó/sec kb.

Mit csinál a vevő:
észleli a lefutóélt, a bitidő felénél mintát vesz, és ha ott 0-t talál, akkor indítja az olvasást.

Ez azt jelenti, hogy ~10 bit alatt fél órajelperiódust csúszhatnak az órajelek. Ez nagyjából 5%-os eltérést jelenthet, ami kifejezetten pozitív.

Milyen hibát észlelhetünk:
Parity error: nem stimmmel a paritásbit.
Frame error: nem stimmel a strat bit pozíciója.

Break karakter:
Szándékosan kerethibás adat:
START bittől a STOP bitig 0 bitet tartalmaz (A stop pozícióján is).
Ezt „figyelemfelhívásra” használhatjuk.

Szinkron adatátvitel:

Nincs bájtonkénti keretezés, folyamatos bitsorozat érkezik az adó felől.
Keretezés azonban van, csak adatcsomagonként.
Be kell vezetnünk egy úgynevezett szinkron karaktert. Az adó, ha nincs mit adni, akkor egy szinkron karaktert ad ki magából.
A szinkronkarakter bármi lehet, különböző szabványok különböző módon definiálják.
Azonban a szinkron karaktert ki kell zárni az adatfolyamból!
Az átvitel 100%-ban hasznos adatot visz át.

Az órajellel együt kell az adatot továbbítani, és az órajelre elég érzékeny az átvitel.

Hogyan találhatjuk meg a kiindulási állapotot?
A vevőeszköznek kereső (hunt) üzemmóddel kell rendelkezni.
A keretet a szinkron karakter biztosítja.

PL: SDLC protokol:

Az adatcsomag szinkronbittel kezdődik, majd egy cím és egy parancs következik, majdvalahány bitnyi adat, majd egy 16 bites CRC ellenőrző kód és egyújabb szinkron bit.
A szinkron bit a 7Eh.
Ez a bitminta csak szinkron karakterként fordulhat elő.
Bit stuffing: ha 6 darab 1-es érték jönne, akkor az 5. 1-es után beszúr egy 0-t.
Bit unstuffing: el tudja távolítani ezt a 0-t.

Milyen átvitelek léteznek:
Szimplex:
Egy adó és egy vevő, nincs visszajelzés.

Félduplex:
Egy pillaatban csak egy adó, de mindkét oldal lehet adó. Általában Master-slave kapcsolat időzítéssel.

Fulduplex:
2 külön vezeték az adó és a vevő részére. Mindekettő beszéhet egyszerre.

Szabvány :
PL.: RS232 elég nagy tűrést megenged.

Modem (modulátor-demodulátod) valósítja meg például a soros átvitelt telefonvonalakon, melyekhez vezérlőjelek tartoznak.

Data terminál ready: jelzi, hogy kész az adatvételre.
Data Set ready: jelzi, hogy adatot továbbítana.
Ezek a kapcsolatfelvételhez szükséges jelek.

Az adatblokkok keretezéséhez:
Request to send:
Clear to send:

Úgy nevezett null modemes összeköttetés, amikor a saját CTS a saját RTS-sel van összekötve és a saját DTR a saját DSR-rel van összekötve.
Csak az adatot viszem át.
Ebben az esetben az adás feltartása szoftverből valósulhat csak meg.
Az adatfolyamban definiálon olyan bájtokat (XON és XOFF) amelyek azt mondják meg, hogy jöhet-e a következő adat.
Ebben az esetben legalább 2 karakteres buffer kell.

8251-es áramkör:

C/ /D bemenet: ha 0, akkor az adatregiszterét, ha 1, akkor command regiszterét érjük el.
/RD és /WR bemenetek.
RESET és Chip select bemenetek.
Szinkron sorrendihálózat, tehát neki CLK kell! Legalább az átviteli sebesség 30x-osa.

Van egy saját belső sínje, erre csatlakozik az előbb tárgyalt R/W logika és az adatsín buffer.
Van egy adó puffer és egy adó vezérlő áramkör.
Utóbbi a következinterface-szelbír:
Külső órajellel rendelkezik, mely az adás órajelének kell megfelelnie.

Van két kimenete melyek IT kérésre is használhatóak.
TX rdy : újabb adat kivitelére kész az eszköz.
TX E: kimeneti puffer kiürült. Az adás végét jelzi.

Vevő puffer és vevő vezérlő is van.
Vevő pufferbe érkezik az adat.
RX rdy a vezérlőn: kimenet, melyen az jelenik meg, hogy az átmeneti bemeneti puffer kiolvasásra került.
RX CLK: saját órajel az adó sebességének megfelelően.

OVERRUN ERROR: olyan hiba, mely abból keletketzik, hogy nem vizsgáljuk az RX rdy-t. A kinem olvasott adatot az új adat felülírja.

Vezérlőn SYNDET / BRKDET ki-bemenet.
Külső szinkronizáló eszközt használhatunk. Kívülről mondhatjk meg, hogy hol van a kiindulási állapot, mikor kell vennünk az adatokat.

Vanak még úgy nevezett MODEM vezérlőjelek, melyek negált logikájúak.
DSR, DTR, CTS, RTS (be-ki-be-ki)

DSR: külső eszköz jelzi, hogy küldené az adatot.
DTR: modem jelzi, hogy jöhet az adat.
RTS: jelzi, hogy kész vagyunk az adat vételére.
CTS: jelzi, hogy adnánk az információt.

Valódi hardver-handshake.
Null-modem esetén a külvilágnak nincs esélye beavatkozni.

Az R/W logika vezérli az összes egységet.

Az eszköz két címen látszik, parancs regiszteren és adatregiszteren. Ha olvassuk az utóbbit, akkor az adatbuffert olvashatjuk, ha írjuk, akkor az adatbufferbe írhatunk.
Ha aparancsot olvasswuk, akkor státuszt olvasunk.
Ha írjuk, akkor szekvenciában programozhatjuk. Szinkron-aszinkron mód, engedélyezések ...

Hogy illesszünk több IT-t egyetlen egy IT bemenethez?

A 3 IT vezetéket egy OR kapuba vezethetjük.
De így csak egyetlen szubrutint hívhatunk, a processzornak ki kell derítenie hogy ki kér IT-t.
Ezt IO címen juttathatjuk be a processzorba.

Az SRT 5.5 viszont szintérzékeny, így az OR kapura egy D flip-flopot is el kell helyeznünk.
A D flip-flop élvezérelt, ezilleszt a szintérzékeny bementre.
Törölnünk is kell, erre a legegyszerább megoldás, ha az írás jelet használjuk,és szoftverből írás parancsot adunk ki.

Ha ezt akarjuk illeszteni, akkor kell egy vezérlő is.
A vezérlő fogja törölni a flip-flopot, ez fogja a /RDY-t adni, amihez fel kell ismernie, hoyg őt címezték, és ha az adatbuszon be akarjuk olvasni a flip-flop értékét, akkor vezérelnie kell egy three-state erősítőt is.

Nézzük a függvényeket:

Buffer engedélyező: EN = Sajátcím * IO/ /M * RD
CLR = Sajátcím * IO/ /M * WR
READY = Sajátcím * IO/ /M

A Ready lényegében egy Chip select jel lényegében.

A saját cím egy konstans komparátorral kapható meg, amit például egy 8 bemenetű AND kapu lehet.

Ha ezt a jelet eőállítottuk, kakkor a /WR és /RD jeleket invertálva és NAND kapuba kötve előállíthatjuk a többit is.

Ha nem cska egy IT-t akarunk, akkor a memóriaillesztéshez hasonlóan dekóder áramkört használhatunk, persze ez csak akkor működik, ha sorban érhetőek el a címek a különböző hardvereinkhez.

Írjunk szubrutint, mely kezeli az IT kérést:

```
IT1 EQU 78h
IT2 EQU 79h
IT3 EQU 7Ah

ITJOTT EQU 01h


ORG 2Ch
JMP ITRUT

ORG 100h

ITRUT: PUSH PSW
IN IT1
ANI ITJOTT
JZ CIT2
INR B

ITVEG:
OUT IT1
POP PSW
EI
RET

CIT2:
IN IT2
ANI ITJOTT
JZ CIT3
INR C
OUT IT2
JMP ITVEG

CT3:
IN IT3
ANI ITJOTT
JZ ITVEG
INR D
OUT IT3
JMP ITVEG
```

Prioritást állítottunk fel az IT-k közt a szoftverben!!!

Ha a rutin kiszolgálás után nem térünk vissza, hanem megnézzük a többi IT-t is, akkor a prioritás eltűnik.

A maszk akkor engedélyez, ha 0-t tartalmaz az adott bit!

Hogy engedélyezhetjük csak az 5.5-öt, a többit nem bánthatjuk:

```
IT55EN EQU 6 // Ezzel fogunk 0-zni, a SET IT maszk egyéb funkcióit nem bántja
ITEN EQU 8       //Ezzel 1-re fogjuk állítani az IT engedélyező bit-et.


ITINIT:
RIM
ANI IT55EN
ORI ITEN
SIM
RET
```

Ez eddig nagyon jó, de mi van, ha a 7.5-ös IT-t használjuk. Az élérzékeny, tehát ha elindul az IT rutin, megállapítjuk, hogy IT1 még nincs, ajd elkezdjük kiszolgálni az IT2-t, de a kiszolgálás közbenmegjelenik az IT1, akkorő ott ragad, mert nem produkál több élt.

Definiáljunkegy engedélyező jelet, mely egy kimeneten látszik, ha oda írunk.
Ha ez az írás ideje alatt 0, és a huzalozott IT.k engedélyezésére használjuk, akkor íráskor, megjelenít egy 0 értéket egy rövid ideig, így 0-zza az egészet.
Ha az IT rutin visszatérése előtt ide írunk, akkor az RST-n megjelenik egy 1-0-1 átmenet.

A 8251-esnek 2 regisztere van, egy adat és egy vezérlés.
Milyen paramétereket tudunk állítani a vezérléshez:

Aszinkron átvitelnél:

7	6	5	4	3	2	1	0

A 0-1 biteken beállíthatunk egy külső frekvenciaosztást, amely 1, 16 vagy 64-es osztást végez a külső órajelen.
A 2-3-as biten az adatbitek számát határozhatjuk meg, 5, 6, 7, 8 bitet választhatunk.
A 7 bit jellemzően ASCII, a bináris adatok átviteléhez 8 bitre van szükségünk.
A 4. bit engedélyezi a paritást, az 5. bit pedig megmondja, hogy páros vagy páratlan paritást szeretnénk-e használni.
A 6. és 7. biteken pedig a stopbitek számát állíthatjuk 1, 1.5 és 2 értékekre.

Szinkron:

7	6	5	4	3	2	1	0

A 0-1 biteken alapértelmezetten a 00 érték kell megjelenjen.
A 2-3-as biten az adatbitek számát határozhatjuk meg, 5, 6, 7, 8 bitet választhatunk.
A 7 bit jellemzően ASCII, a bináris adatok átviteléhez 8 bitre van szükségünk.
A 4. bit engedélyezi a paritást, az 5. bit pedig megmondja, hogy páros vagy páratlan paritást szeretnénk-e használni.

A 6. bit ESD, a syndet irányát állíthatjuk.
A 7. biten állíthatjuk be a szinkron karakterek számát, mely 0 és 1 lehet.

Az első utasítás egy command instruction, ami az első felprogramozás adatait hordozza.
A comman instruction egy bitjén beállíthatjuk, hogy a következő utasítás mode instuction legyen, és aaz eszközön módot válthassunk.
Command:

7	6	5	4	3	2	1	0

0: adás engedélyezése
1: DTR kimenet egnedélyezése → 1 : /DTR = 0
2: vétel engedélyezése
3: break karakter küldése aszinkronüzemmódnál
4: hibák törlése
5: RTS állítás → 1: /RTS = 0
6: Belső reset (a következő programparancs mode instuction)
7: Hunt üzemmód szinkron átvitelhez

Státusz:

7	6	5	4	3	2	1	0

0: a következő karakter tölthető
1: vétel puffer teli
2: az utolsó karakter elküldve
3: Paritáshiba (csak jelzi hogy volt, kezelni a programozónak kell)
4: Túlfutási hiba (nem olvastuk ki az előző karaktert (vagy többet), de már érkezett újabb)
5: Kerethiba
6: SYNDET bemenet értéke
7: DSR bemenet értéke

Vegyük észre, hogy a CTS jel értékét nem látjuk. A CTS és RTS hardverhangshake, ezt nem kell kezelnünk és nem tudjuk befolyásolni.

Írjunk programot a 8251.es inicializálására:
Tegyük fel, hogy a 80h-n és 81h-n helyezkedik el a chipünk.

```
USARTD EQU 80h          //Adatregiszter elhelyezkedése
USARTC EQU USARTD + 1    //A parancsregiszter elhelyezkedése

//Nem tudjuk milyen állapotban van, ezért parancsot szeretnénk neki kiadni:

USARTR EQU 01000000b     //Reset parancs konstansa

//16-os órajelosztást, 8 bitet, 1 STOPbitet, paritás nélkül szeretnénk beállítani. A megfelelő értékeket a segédletből olvassuk ki a konstans megvalósításához

USARTM EQU 01001110b     //A fenti átvitel konstansa


USARTINIT EQU 00110111b  //nincs hunt üzemmód, nincs belső reset,  /RTS kimenet 0hibákat töröltük, nem
                         küldünk break karaktert, vételt és adást engedélyezzük, a /DTR kimenet 0

XRA A
OUT USARTC
OUT USARTC
OUTUSARTC
```

```
MVI A, USARTR          // Akkuba mozgatuk a konstanst
OUT USARTC             // Kivisszük a megfelelő címre az Akku tartalmát, amit előbb beállítottunk
MVI A, USARTM
OUT USARTC
MVI A, UARTINIT
OUT USARTC
```

Nézzük szinron esetben:

```
USARTS EQU 10001100b    //1 szinkronbit, a syndet kimenet, nem kell paritás, 8 bit
SYNCHAR EQU 7Eh         //A szinkron karakter kiválasztása
USARTSINIT EQU 10110111b //hunt üzemmód, nincs belső reset, /RTS kimenet 0, hibák törlése, nincs break
                         karakter, adás és vétel engedélyezve, /DTR kimenet 0
```

```
XRA A
OUT USARTC
OUT USARTC
OUTUSARTC

MVI A, USARTR
OUT USARTC
MVI A, USARTS
OUT USARTC
MVI A, SYNCHAR
OUT USARTC
MVI A, USARTSINIT
OUT USARTC
```

9600 bode rate-hez a processzor órajelét pont 320-al kell osztani.
16-tal tud osztani az áramkör, a maradék 20-as osztást nekünk kell kitalálni.

Jó volna, ha volna egy áramkörünk, amely programozhatóan képes előállítani valamilyen órajelet.
Számos időzítőszámlálási feladatunk lehet, így nem meglepő, hogy 8253 néven megvalósítottak neknk egy ilyet.

Egyébként gond nélkül tudnánk mi is ilyet csinálni, vegyünk például egy számlálót és egy komparátort, amely 1-et tölt be, ha elértük a megfelelő értéket.
Ekkor 1 … n számlálási ciklust valósítottunk meg.
Hogy lesz ebből órajel:
komparáljuk az n/2 értékkel, és használjuk a komparátor <= kimenetét.

8253:
3 darab 16 biets számláló
Bináris / BCD számlálás.
Lefutóél vezérelt.
Lefelé számlál.
Egy apró baki van a szerkezetben, az órajele maximum 2,6MHz, tehát a processzor órajlét előbb egy T flip-floppal le kell osztanunk. Ez után viszont már csak 10-se osztást kell beállítanunk!

4 címen látszik, ebből 1 vezérlő, 3 számláló.

MODE0:
IT ha a számláló lejár.

MODE1:
Adott szélességű impulzus előállítása.
Újraindítható.
A számlálást a Gate bemenetenérkező 0 megállítja, a gate bemeneten érkező felfutóél újraindítja.
Tehát egy monostabil számláló, aza újraindítható.

N-nel osztás:
A betöltött adattól számláls visszafele, és ha eléri az 1-et, akkor 0-ig kiad egy impulzust (0 értéket,mertnegált)
A kitöltési tényező N-nel arányosan csökken. Nagy N-re nagyon pici.

Négyszögjel generátor:
Az előbb felvázolt komparátoros számláló áramkör, a kitöltési tényező páros számokra 50%, páratlan számokra pedig picit nagyobb, mint 50%.

Jelzés adott idő múlva:
(Szoftver triggerelt strób impulzus)
A gate által engedélyezett esetben a számláló visszafele számlál,lejáratnál impulzust ad.
A gate engedélyezi, de nem indítja újra.

Hardware triggerelt strobe:
A gate 0 értéke alatt a számlálás folytatódik, a gate felfutóélére a számlálóújraindul.

A szoftware modell:
A1 A0 bemenetek vannak, tehát 4 cím van.
00 CNT0
01 CNT1
10 CNT2
11 Control

16 bites számlálókat 1 regiszterrel hogy kezelhetünk?
Beállítható, hogy előbb az alacsonyabb, majd a magasabb helyiérték vivődjön át.
Beállítható, hogy csak a felső vagy csak az alsó bitekhez férjünk hozzá.

Mi van az előbbi esetben?
Lehet, hogy kiolvassuk az alsó bitet, majd kiolvassuk a felső bitet, és ez a két értéknem tartozik össze, ha a számláló időközben átbillen.
Ez ellen úgy védekezhetünk, hogy a beépített átmeneti regisztert használjuk, melybe a számláló ideiglenesen kitölti az éppenaktuális értéket, és az olvasás csak erre a regiszterre történik.

Vezérlőszó:

7	6	5	4	3	2	1	0

0-1 : számláló sorszáma
2-3: olvasás 11: LSBMSB, 01: MSB, 10:LSB, 00:tárolás
4-5-6: üzemmód (000 : 0 , 001 : 1, -10 : 1 , -11 : 3, 100 : 4, 101 : 5)
7: BCD 0, bin 1

Írjuk meg a programot ami oszt 10-zel:

```
CNTR0 EQU 90h                 //Itt látszik a számláló
CNTRC EQU CNTRO+3

CNTRCW EQU 00100110b          //a controlword

MVI A, CNTRCW
OUT CNTRC
MVI A, 10
OUT CNTR0
```

Megszakítás kezelők alkalmazása:

A CPU modul nem tartalmaz megszakításkezelőt.

Sínen: INT, /INTA

8259A:

IR0 … IR7 bemenetek, itt kérhetünk megszakítást.
CAS0, CAS1, Cas2

/SP / /EN: állítható kimenet vagy bemenet.
Ha bemenet, akkor azt állíthatjuk, hogy az IC master vagy slave módban viselkedjen.

Van egy /CS chip select.
D0 … D7 adatvezetéek.

A0 bemenet, /RD és /RW bemenet.

INT kimenet egy bufferel leválasztva.
/INTA bemenet, melyen a processzor jelzi, hogy elfogadta a megszakításkérést.

Az adatvezetékeket le kell választanunk egy 79245-ös kétirányú bufferrel.

Kell egy chip kiválasztó vezérlés. Fogadja az A7 … A1 vezetékeket IO címről van szó, thát 8 bit kell, az A0-t már felhasználtuk). Kell az IO/ /M jel is.
Ad egy readyt.
Szüksége van a /RD és /INTA jelekre.
Vezérli a 79245-ös /G és DIR bemeneteit.

Nézzük a veérlést részletesen:

Chipselect akkor kell, ha az IO/ /M -en IO-t észlelünk és a címvezetékeken a saját címünk van.

Adaterősítő vezérlése:
G-t vagy a Chipselect vagy pedig az INTA vezérli. Ha bármelyik van, akkor engedélyeznünk kell.
DIR-t akkor kell kifele engedélyeznünk, ha INTA vagy Read van.

Ready előállítása:
Ready t kell adnunk, ha INTA ciklus van, vagy engedélyeztük a chip-et.

MP:
Legyen a cím a 20h. Ekkor a hipet a 20h és a 21h címen érhetjük el (az A0 vezeték miatt).

Tervezzük meg a hardvert.

Írjuk meg az IT rutint:

-regiszterek mentése (amit használni szeretnénk,a PSW-t szinte mindig mentjük)
-IT kiszolgálás (IT törlés)

-regiszterek visszaállítása
-IT engedélyezése (8259, 8085)

Autómatikus EOI

Amikor sikerült feladni az ugrási címet és az utasítás kódját, akkor egyből engedélyezi az új IT rutint.
Csak akkor használható, ha IT rutinok nem megszakíthatóak, és nincs magasabb prioritású IT-nk.

```
ITRUT0:
PUSH PSW
…                    // Itt nincs engedélyezve az újabb IT kérés

POP PSW
EI
ERT
```

Programozott EOI
-hosszú IT rutinok esetén
-nagy prioritású IT esetén

```
ITRUT1:
PUSH PSW

…
Nem megszakítható kódrész

EI

…
Megszakítható


MVI A, 20h
OUT X59
POP PSW
RET
```

Inicializáljuk a megszakításkezelőt a következő működjön:
01c0h címen legyen az ugrás, él+szint érzékeny, 4byte cím intervallumokal, AEOI

```
ICW1: 11010111
ICW2: 00000001
ICW4: 00000010              //Csak az autómatikus end of interrupt miatt


ITMASK EQU 0FFh
ICW1 EQU 0D7h
ICW2 EQU 01h
ICW4 EQU 02h
X59 EQU 20h
ITBASE EQU 01C0h


DI          //IT kikapcsolása.
MVI A, ICW1
OUT X59
```

```
MVI A, ICW2
OUT X59+1
MVI A, ICW4
OUT X59+1                  //Itt az inicializálás vége, innentől már a mask regisztert írhatjuk.
MVI A, ITMASK
OUT X59+1

LXI H, ITBASE
MVI A, 0C3h
MOV M, A
INX H
LXI D, ITRUT0
MOV M, E
INX H
MOV M, D
INX H
INX H
-------
MOV M, A
INX H
LXI D, ITRUT1
MOV M, E
INX H
MOV M, D
INX H
INX H
-------

…


EI
RET
```

Kiskérdésekhez:
Mi történik, ha kaszkádosított rendszerben a master egység IR 0 bemenetére csatlakozik slave?
Ekkor 7 darab IT-t tudunk csak kiszolgálni. A Master a kaszkádosító vonalakra a 0 kombinációt adja ki alapértelmezetten, tehát az ide kötött megszakításvezérlő úgy érzékeli, hogy őt kiszolgálták.

LCD panel illesztése

…

Az adatregiszter írható és olvasható.
Olvasásnál státuszt kapunk.
2*4 bitesként programozható, mert gyakran mikrokontrollerrel vezérlik.

A parancsok végrehajtása sajnos nem 0 idejű.

A képernyő teljes törlése kb. 4.5ms
A kurzor visszatérése a bal felső sarokba 1.5ms
A kurzor irányállítása, képernyő ki és bekapcsolása, képernyő léptetése, CGRAM és DDRAM címek beállítása, üzemmód állítása és CG/DDRAM állítása 37mikroszekundum.

Új utasítást csak akkor adhatunk ki, ha az előző befejeződött.

Bekapcsoláskor autómatikus reset:

MP:
Írjunk szubrutint, amely egy nyomógomb állapotát vizsgálja, és jelzi, ha a gombot megnyomták.
Minket az érdekel, hogy a gombot megnyomták-e, tehát egy 0-1 átmenetre várunk a gombon.
Ha valaki folyamatosan nyomja a gombot, az számunkra nem jó!

```
GOMB EQU 1
GOMBADRESS EQU 80h

NYOM:
IN GOMBADRESS
ANI GOMB
JNZ NYOM                    //Amíg a gomb meg van nyomva, addig várunk.

Ciklus1:
IN GOMBADRESS
ANI GOMB
JZCiklus1
```

//Szoftverből pergésmentesítsünk, állítsuk 10ms-ra a várakozást, a gombot 100-szor nyomhatjuk így meg, ami elég, és agomb sem pereg többet:

```
CALL WAIT
IN GOMBADRESS
ANI GOMB
JZ C1

RET
```

```
WAIT:
LXI B, W10ms            //Mekkora legyen a W10ms konstans? Nagyjából 1400-ra van szükségünk.

WAIT1:
DCX B
MOV A, B
ORA C
JNZ WAIT1
RET
```