Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

FPGA tervezői laboratórium

- -
- -

Budapest, 2018. április

Tartalomjegyzék

1. fejezet

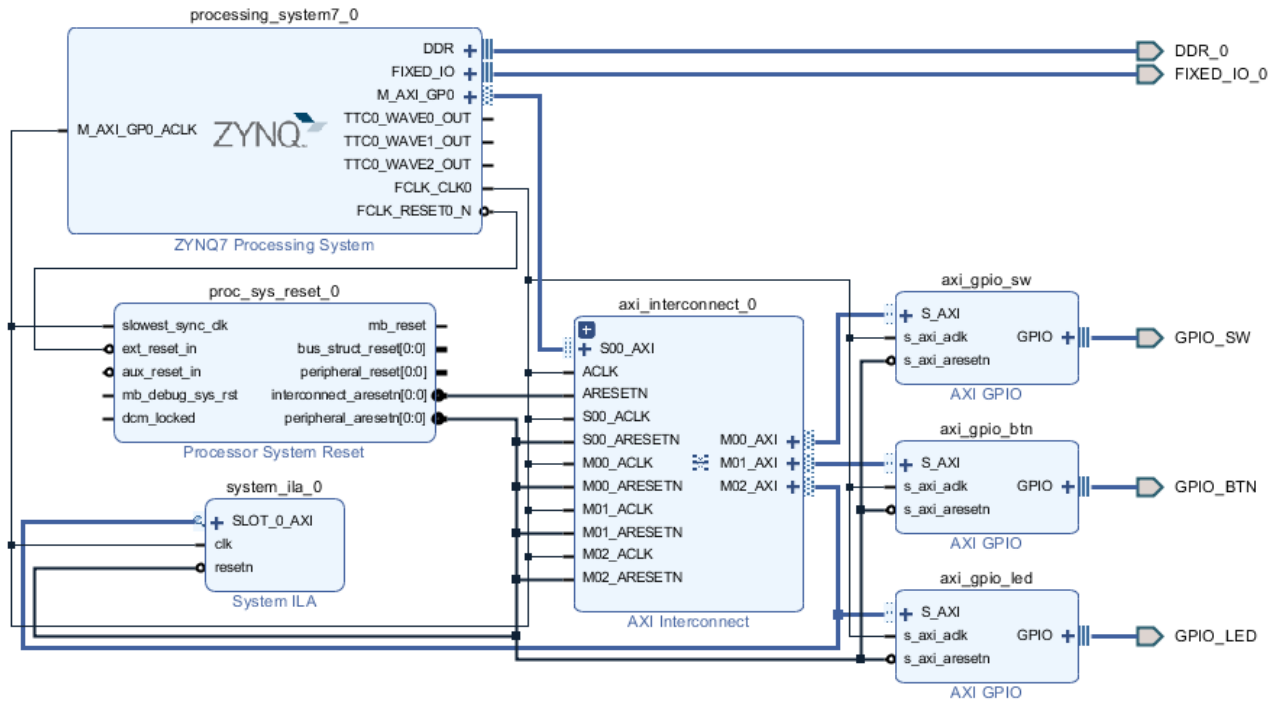
Egyszerű processzoros rendszer létrehozása

Az első labor alkalmával a Xilinx Vivado 17.4. fejlesztői környezettel ismerkedtünk meg, amellyel egy viszonylag egyszerű, demonstrációs célokat szolgáló processzoros rendszert hoztunk létre, a rendelkezésünkre bocsátott Zedboard FPGA-n. Ez a processzoros rendszer roppant egyszerűen nézett ki és az alábbi hardver részegységekből épült fel:

- arm Cortex A9-es processzoros rendszer (Zynq Processing System, PS)
- általános célú bemeneteket és kimeneteket (GPIO) modellező egységek
- összeköttetés hálózat (Interconnect)
- reset szinkronizáló
- ChipScope logikai analizátor (System ILA)

1.1. A rendszer blokkvázlata

Az előbb felsoroltakat a fejlesztői környezet Block Design funkciójával tudtuk az FPGA-tervhez hozzáadni és a közöttük lévő kapcsolatot megadni, amelynek eredményét az 1.1. ábra mutatja.



1.1. Ábra – *A Block Design-ban összeállított rendszer blokkdiagramja*

A processzoros rendszer egyik lassabb AXI interfészét (M_AXI_GP_0) használtuk arra, hogy kommunikáljunk az FPGA konfigurálható logikájában létrehozott egységekkel. Ezek a fejlesztői kártyán megtalálható nyomógombok, kapcsolók és LED-ek kezeléséért voltak felelősek. Egyszerűségüknél fogva általános célú bemenetek és kimenetek formájában kezeltük őket és egy-egy AXI GPIO modult rendeltünk hozzájuk.

Az AXI protokoll egy pont-pont kapcsolatot ír le és így egy Master-hez egy Slave interfész köthető közvetlenül. Ugyanígy, a processzoros rendszer a Master AXI interfészén csak egy Slave interfésszel tud kommunikálni, de hárommal kellene, hiszen három perifériát definiáltunk a rendszerhez. A probléma megoldásához egy AXI Interconnect buszmátrixot kellett beépíteni, amely képes volt a Master interfésztől érkező kéréseket a megfelelő Slave interfészek felé továbbítani.

A FPGA konfigurálható részében megvalósított áramkör reset-je aszinkronnak tekinthető az ott lévő egyetlen órajeltartományhoz képest, így ennek a jelnek az elengedését mindenképp szinkronizálni kell az órajelhez. Ehhez egy programozható reset szinkronizálót helyeztünk el FPGA-ba.

A ChipScope logikai analizáltort a LED GPIO modul AXI Slave interfészének megfigyelésére tettük bele a design-ba. Ezen keresztül tudtunk egy AXI írási és egy olvasási transzfert megvizsgálni. A logikai analizátorból elegendő volt egyet lerakni, hiszen az FPGA erőforrásai egyetlen órajelről jártak.

Az így létrejött rendszer perifériáihoz még hozzá kellett rendelni néhány memóriatartományt, hiszen a kommunikáció a processzoros rendszer és a perifériák között memória írás/olvasás formájában zajlott. Ezt a Block Design-ban az Address Editor fülön tudtuk megtenni, amellyel automatikusan lehetett memóriatartományokat létrehozni és hozzárendelni. Ezt mutatja az 1.2. ábra.

Address Editor | cpu_subsys_wrapper.v | Diagram

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_btn	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_led	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_sw	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF

1.2. Ábra – *Az összeállított rendszer memóriatérképe*

1.2. C kód

Ezután, hogy kipróbáljuk az összerakott rendszert, egy egyszerű C nyelvű programot írtunk, amely egy másodperc számlálót jelenített meg a LED-eken. Ehhez, az így kapott FPGA tervet átvittük az SDK környezetbe, ahol kényelmesen le tudtuk kódolni a programot, a generált rendszerleíró fejléc fájl és a perifériák kezeléséért felelős függvénykönyvtár használatával. Az általunk megírt programot az 1.1. kódrészlet mutatja be.

1.1. Kód – *A rendszeren futtatott C program*

```
1  #include "xparameters.h"
2  #include "xgpio_l.h"
3
4  void    inic_btn(){     Xil_Out32(XPAR_AXI_GPIO_BTN_BASEADDR + XGPIO_TRI_OFFSET,0xFFFFFFFF);    }
5
6  void    inic_sw(){      Xil_Out32(XPAR_AXI_GPIO_SW_BASEADDR + XGPIO_TRI_OFFSET,0xFFFFFFFF);     }
7
8  void    inic_led(){     Xil_Out32(XPAR_AXI_GPIO_LED_BASEADDR + XGPIO_TRI_OFFSET,0x0);   }
9
10 int main(){
11
12         inic_btn();
13         inic_sw();
14         inic_led();
15
16         int cntr = 0;
17         while(1){
18                 volatile int i= 0;
19
20                 while(i< XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ/20){ i++;}
21                 cntr++;
22
23                 Xil_Out32(XPAR_AXI_GPIO_LED_BASEADDR + XGPIO_DATA_OFFSET, cntr & 0xFF);
24                 Xil_In32(XPAR_AXI_GPIO_LED_BASEADDR + XGPIO_DATA_OFFSET);
25         }
26
27         return 0;
28 }
```

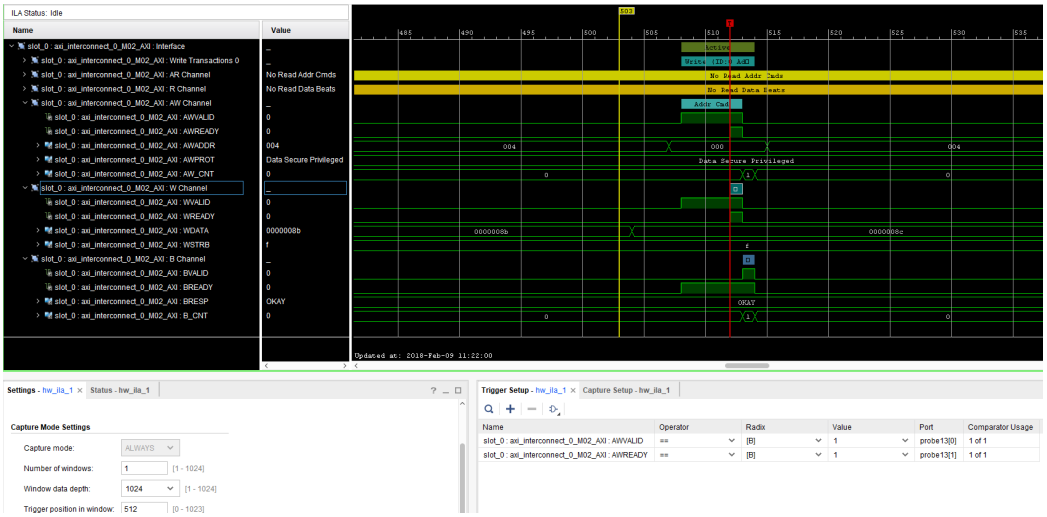
A kód először inicializálja a három GPIO modult: a GPIO modul tristate vezetékeit vezérlő regiszterbe a megfelelő értékét írja be. Ezzel állítja be, hogy a 32 bit széles busz egyes bitjei bemenetek vagy kimenetek legyenek. Logikai 1 esetén az adott vezeték bemenetként funkcionál, míg 0 esetén kimenetként. A 4 byte-os memória íráshoz a Xil_Out32 makrót használtuk. Ezeket a 4., 6. és 8. sorok írják le.

A számlálóhoz egyszerűen egy általunk deklarált változó értékét léptettük egy végtelen ciklusban, kb. másodpercenként. Azt, hogy a processzoros rendszeren mikor telik el kb. egy másodperc, egy másik számláló léptetésével értük el. Ezt a számlálót a processzoros rendszer órajelének frekvenciájához igazítottuk először (kb. 660 MHz). Nyilván ez csak akkor lenne egy értelmes megoldás, ha a számlálást végrehajtó kódrészlet mindig egy órajel periódus alatt befejeződne. De esetünkben, egy while ciklus és a ciklus mag végrehajtása több órajelet vesz igénybe, plusz még az esetleges Cache memória is beleszólhat a végrehajtási időbe. Ezért a számlálót sokkal kisebb ideig léptettük: XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ/20, ezt a 20. sor írja le.

1.3. ChipScope hullámformák

1.3.1. AXI4 Lite írási buszciklus

A kód futtatása során kipróbáltuk a GPIO_LED-re illesztett logikai analizátort, amellyel megvizsgáltunk egy AXI írás és egy olvasás transzfert. Az írás hullámformája az 1.3. ábrán látható.

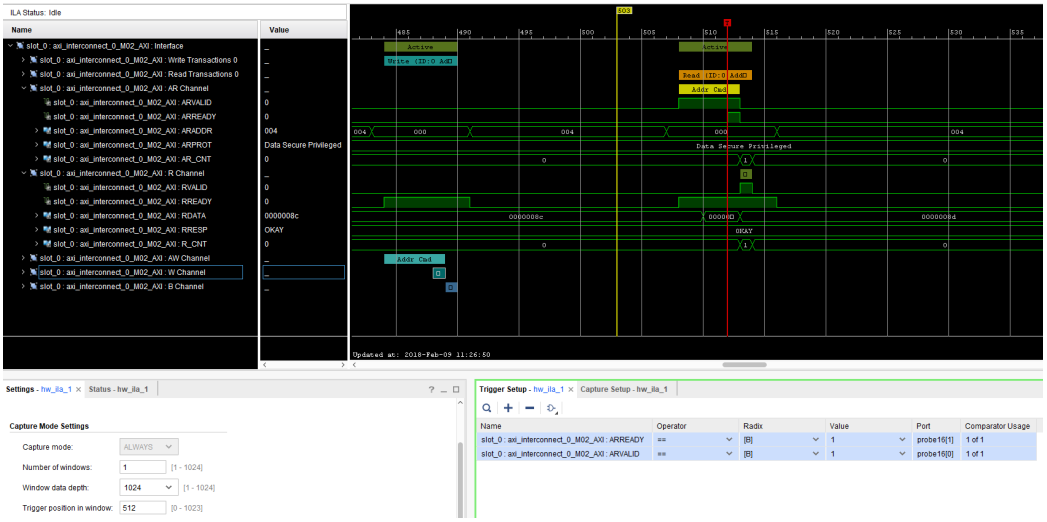


1.3. Ábra – *Egy AXI írási ciklus hullámformája a logikai analizátorban*

Ehhez a logikai analizátort AXI interfész monitorozására kellett beállítani a Block Design-ban. Az íráshoz 3 csatorna áll rendelkezésre az AXI-ban: AW (write address), W (write data) és B (write response). Mindegyik csatornán a Master és a Slave handshake-kel jelezve cserél információt (xVALID és xREADY). Tehát pl. az AW csatornán a Master interfész kirakja az írási címet és jelez a Slave interfésznek (AWVALID-ot 1-be húzza), hogy a cím buszon érvényes cím van. A Slave ezt látja, és nyugtázza ezt az AWREADY 1-be húzásával és amikor e két jel logikai 1 értékű, akkor formálisan megtörténik az adatátvitel, azaz jelen esetben a Slave letárolja az írási címet.

Az előbb említett eseményhez a handshake jeleket kellett megfigyelnünk a logikai analizátorban és a trigger esemény a két jel együttes logikai 1 értéke lett. A triggerpozíció alapból a mintaablak közepére lett beállítva, így az előtte megkezdett eseményeket is láttuk, valamint az utána zajló írást és írás választ is.

1.3.2. AXI4 Lite olvasási buszciklus



1.4. Ábra – *Egy AXI olvasási ciklus hullámformája a logikai analizátorban*

Az olvasási ciklust hasonlóképpen tudtuk megfigyelni, de alapból nem volt szükség a GPIO_LED értékének visszaolva-

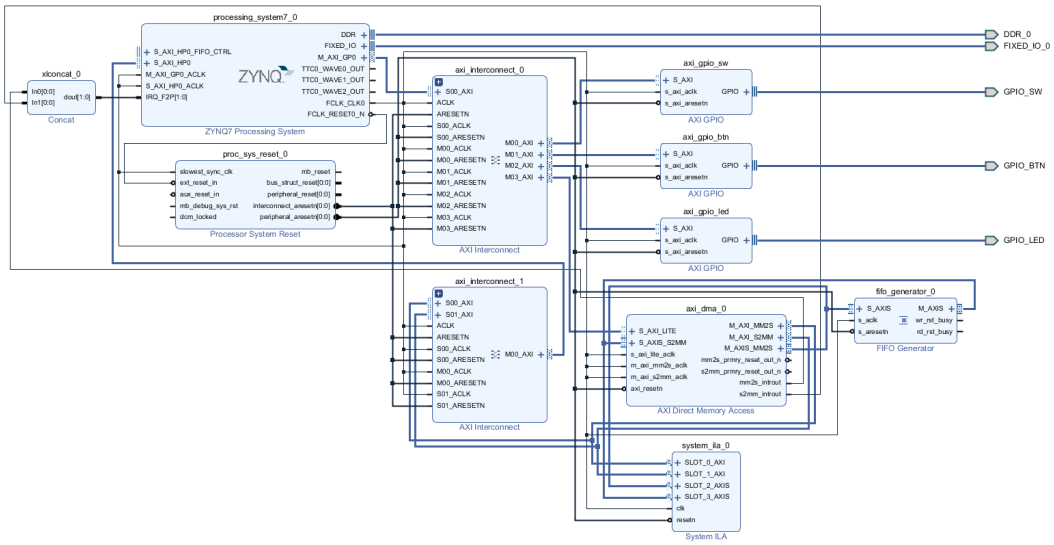
sására, ezért a kódba egy dummy olvasást írtunk bele (lásd 24. sor), hogy a logikai analizátor el tudjon kapni egy olvasást. A kapott hullámformát az 1.4. ábra mutatja.
Ez esetben is hasonló trigger feltételt fogalmaztunk meg, de most az AR csatorna handshake jeleire. Egyébként, egy olvasást az SDK XSCL Console alkalmazásával is generálni tudtunk volna.

2. fejezet

DMA vezérlő megvalósítása és tesztelése

A második labor alkalmával az első alkalommal létrehozott rendszert bővítettük ki, méghozzá egy DMA vezérlővel és egy, vele kommunikáló FIFO-val. A DMA vezérlő illesztéséhez kibővítettük a meglevő Interconnect-ünket egy újabb Master IF-szel, amelyre a DMA konfigurációs regiszter IF-szét kötöttük. Továbbá, a PS-en bekapcsoltunk egy Slave IF-et, hogy ezen keresztül a DMA belelásson az arm-os alrendszer rendszermemóriájába. Mivel egyetlen Slave IF-et engedélyeztünk a PS-en, így a DMA-t csak egy újabb, két Slave-es egy Master-es Interconnect segítségével tudtuk rákötni a PS-re.

2.1. A rendszer blokkvázlata



2.1. Ábra – *A Block Design-ban összeállított rendszer blokkdiagramja*

Diagram | Address Editor

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
axi_gpio_btn	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_led	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_sw	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

2.2. Ábra – *Az összeállított rendszer memóriatérképe*

2.2. A DMA vezérlő regiszterei

Az alábbi táblázatban összegyűjtöttük a DMA vezérlő „Direct Register Mode”-ban használható regiszterek térképét.

Offset	Név	Leírás
0x00	MM2S_DMACR	MM2S csatornát vezérlő regiszter
0x04	MM2S_DMASR	MM2S csatorna állapotát leíró regiszter
0x18	MM2S_SA	A forrás 32 bites címe: megadja, hogy a DMA melyik memória rekeszt olvassa ki és írja bele a perifériába
0x1C	MM2S_SA_MSB	64 bites forrás cím esetén a felső 32 bit
0x28	MM2S_LENGTH	A DMA-ban lévő buffer méretét tárolja byte-okban, amely a rendszermemóriából beolvasott adatok átmeneti tárolására szolgál
0x30	S2MM_DMACR	S2MM csatornát vezérlő regiszter
0x34	S2MM_DMASR	S2MM csatorna állapotát leíró regiszter
0x48	S2MM_DA	A nyelő 32 bites címe: kijelöli, hogy a DMA melyik memóriacímre írja be az adatot a rendszermemóriában, amit átvett a perifériától
0x4C	S2MM_DA_MSB	64 bites címzés esetén a nyelő címének felső 32 bitjét tároló regiszter
0x58	S2MM_LENGTH	A DMA-ban lévő buffer mérete byte-okban, amely a perifériától átvett adatok átmeneti tárolására szolgál

2.1. Táblázat – *A DMA vezérlő regisztertérképe „Direct Register Mode” esetén*

A két csatornát vezérlő regiszterek segítségével lehet például globálisan engedélyezni a DMA modult (*_DMACR[0]]), esetleg reset-elni (*_DMACR[2]]) és különböző megszakítással kapcsolatos funkciót engedélyezni, vagy tiltani.

A státusz regiszterek jobbára csak olvasható biteket tartalmaznak, melyek a DMA modul és a csatorna állapotát adják vissza olvasás esetén (*_DMASR[1:0]]). Továbbá, tartalmaznak még olyan biteket is, amelyek a hibás és kivételes állapotok bekövetkezését tükrözik, olvasás esetén.

A *_LENGTH regiszterek a DMA-ban megvalósított bufferek méreteit tartalmazzák, byte-os egységekben. A regiszter 32 bites, de kizárólag az alsó 23 bit írható ebből, azaz legfeljebb 8 MB-os lehet egy buffer.

A forráscímet a MM2s_SA (a DMA melyik rendszer memóriabeli címről olvasson ki adatot), valamint a nyelő címét a S2MM_DA (a DMA melyik memóriacímre írja be a perifériától átvett adatot) regiszterek tartalmazzák. 32 bites címzés esetén ezek elegendőek, míg 64 bitesre konfigurálva a DMA-t, használhatók még a _SA_MSB és a _DA_MSB regiszterek.

2.3. C kód

A DMA teszteléséhez az xaxidma_example_simple_poll.c példaprogramot használtuk, amelyet a a 2.1. kódrészlet mutat be. A DMA két adatbuffere BRAM-ban került megvalósításra az FPGA-ban, báziscímeit a 27-es és 28-as sorok írják le. Ezek lettek leképezve a rendszer memóriatartományába.

2.1. Kód – *A rendszeren futtatott C program a DMA vezérlő tesztelésére*

```
1   #include "xaxidma.h"
2   #include "xparameters.h"
3   #include "xdebug.h"
4
5
6   #define DMA_DEV_ID              XPAR_AXIDMA_0_DEVICE_ID // A DMA azonositoja
7
8   #ifdef XPAR_AXI_7SDDR_0_S_AXI_BASEADDR
9   #define DDR_BASE_ADDR           XPAR_AXI_7SDDR_0_S_AXI_BASEADDR
10  #elif XPAR_MIG7SERIES_0_BASEADDR
11  #define DDR_BASE_ADDR   XPAR_MIG7SERIES_0_BASEADDR
12  #elif XPAR_MIG_0_BASEADDR
13  #define DDR_BASE_ADDR   XPAR_MIG_0_BASEADDR
14  #elif XPAR_PSU_DDR_0_S_AXI_BASEADDR
15  #define DDR_BASE_ADDR   XPAR_PSU_DDR_0_S_AXI_BASEADDR
16  #endif
17
18  #ifndef DDR_BASE_ADDR   // A DDR memoriavezerlo memoriacime
19  #warning CHECK FOR THE VALID DDR ADDRESS IN XPARAMETERS.H, \
20                   DEFAULT SET TO 0x01000000
21  #define MEM_BASE_ADDR           0x01000000
22  #else
23  #define MEM_BASE_ADDR           (DDR_BASE_ADDR + 0x1000000) // A rendszermemoria baziscime
24  #endif
25
26  // A DMA bufferjeinek memoriacime/tartomanya
27  #define TX_BUFFER_BASE          (MEM_BASE_ADDR + 0x00100000)
28  #define RX_BUFFER_BASE          (MEM_BASE_ADDR + 0x00300000)
29  #define RX_BUFFER_HIGH          (MEM_BASE_ADDR + 0x004FFFFF)
30
31
32  #define MAX_PKT_LEN             0x100 // 1 DMA atvitel nagysaga bajtokban
33
34  #define TEST_START_VALUE        0xC // A DMA TX bufferjenek tartalma a teszt elejen
35
36  #define NUMBER_OF_TRANSFERS     10 // Ennyiszer DMA-zunk, 1 DMA-zas = mindket iranyba
37
38
39  // Fuggveny deklaraciok
40  int XAxiDma_SimplePollExample(u16 DeviceId);
41  static int CheckData(void);
42
43  // A DMA-t leiro valtozo
44  XAxiDma AxiDma;
45
46  int main()
47  {
48          int Status;
49          Status = XAxiDma_SimplePollExample(DMA_DEV_ID); // Polling-ban beszelgetunk a DMA-val
50
51          if (Status != XST_SUCCESS) {return XST_FAILURE; }
52          return XST_SUCCESS;
53  }
54
55  int XAxiDma_SimplePollExample(u16 DeviceId)
56  {
57          XAxiDma_Config *CfgPtr; // A DMA konfigjara mutato cim
58          int Status;
59          int Tries = NUMBER_OF_TRANSFERS; // DMA-zasok szama
60          int Index;
61          u8 *TxBufferPtr; // Tx buffer a DMA-ban, a Device=FIFO fele meno adatok bufferelesere
62          u8 *RxBufferPtr; // Rx buffer a DMA-ban, a Device=FIFO-bol jovo adatok bufferelesere
63          u8 Value;
64
65          TxBufferPtr = (u8 *)TX_BUFFER_BASE ;
66          RxBufferPtr = (u8 *)RX_BUFFER_BASE;
67
68
69          CfgPtr = XAxiDma_LookupConfig(DeviceId);
70          if (!CfgPtr) {
71                  return XST_FAILURE; // Nem talalt konfigot az eszkoz azonosito alapjan
72          }
73
74          Status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
75          if (Status != XST_SUCCESS) {
76                  return XST_FAILURE; // Nem tudta incializalni a DMA-t a konfig alapjan
77          }
78
79          if(XAxiDma_HasSg(&AxiDma)){
80                  return XST_FAILURE; // Scatter-gatter-re lett konfigolva a DMA
81          }
82
83          // Az interrupt-okat a ket csatornara/iranyra kikapcsoljuk, mert polling-ban hasznaljuk a DMA-t
84          XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DEVICE_TO_DMA);
85          XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DMA_TO_DEVICE);
86
87          Value = TEST_START_VALUE;
88
89          // TxBuffer bajtjait feltoltjuk a kezdeti ertekkel
90          for(Index = 0; Index < MAX_PKT_LEN; Index ++) {
91                          TxBufferPtr[Index] = Value;
92                          Value = (Value + 1) & 0xFF;
93          }
94
95          /* Flush the SrcBuffer before the DMA transfer, in case the Data Cache
96           * is enabled
97           */
98          Xil_DCacheFlushRange((UINTPTR)TxBufferPtr, MAX_PKT_LEN);
99  #ifdef __aarch64__
100         Xil_DCacheFlushRange((UINTPTR)RxBufferPtr, MAX_PKT_LEN);
101 #endif
102
103         // DMA-zunk 'Tries'-szor
104         for(Index = 0; Index < Tries; Index ++) {
105             // S2MM irany, RxBuffer-be beolvasunk a FIFO-bol
```

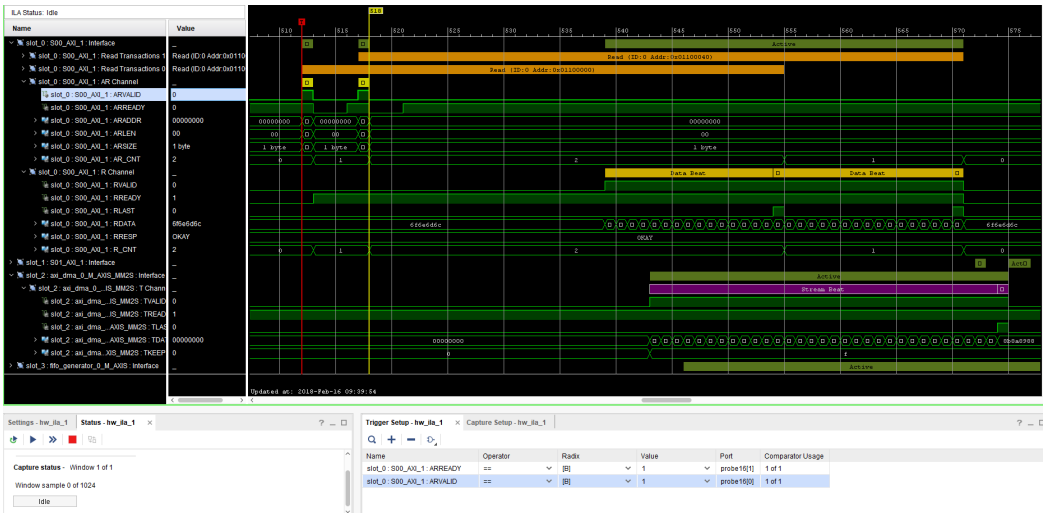
```
106                 Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) RxBufferPtr, MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
107
108                 if (Status != XST_SUCCESS) {
109                         return XST_FAILURE;
110                 }
111         // MM2S irany, TxBuffer tartalmat kikuldjuk a FIFO-nak
112                 Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) TxBufferPtr,MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
113
114                 if (Status != XST_SUCCESS) {
115                         return XST_FAILURE;
116                 }
117
118                 // Polling-olunk
119                 while ((XAxiDma_Busy(&AxiDma,XAXIDMA_DEVICE_TO_DMA)) || (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE))) {}
120
121 /*              Status = CheckData();
122                 if (Status != XST_SUCCESS) {
123                         return XST_FAILURE;
124                 }
125 */
126         }
127         return XST_SUCCESS;
128 }
129
130 static int CheckData(void) // A DMA-ba beolvasott adatot ellenorizzuk a kiindulasi adatokkal
131 {
132         u8 *RxPacket;
133         int Index = 0;
134         u8 Value;
135
136         RxPacket = (u8 *) RX_BUFFER_BASE;
137         Value = TEST_START_VALUE;
138
139         /* Invalidate the DestBuffer before receiving the data, in case the
140          * Data Cache is enabled
141          */
142 #ifndef __aarch64__
143         Xil_DCacheInvalidateRange((UINTPTR)RxPacket, MAX_PKT_LEN);
144 #endif
145
146         for(Index = 0; Index < MAX_PKT_LEN; Index++) {
147                 if (RxPacket[Index] != Value) {
148                         return XST_FAILURE;
149                 }
150                 Value = (Value + 1) & 0xFF;
151         }
152
153         return XST_SUCCESS;
154 }
```

A DMA transzferek során mindig először a S2MM irány, azaz FIFO → DMA irány kerül felprogramozásra. Mert így az adat terjedése szempontjából később elhelyezkedő blokkok kerülnek először felprogramozásra, így készen állnak adat fogadására az előző blokkból.

2.4. Átvitel vizsgálata ChipScope-pal

2.4.1. MM2S irány vizsgálata egyetlen 128 byte-os átvitel esetén

Az MM2S irányt a 2.3. ábra mutatja be. Látható, hogy először a DMA kiolvassa a memóriából az adatot (AR és R csatornák), majd továbbítja azokat a FIFO-nak (AXIS_MM2S). A teljes buszciklus kb. 59 órajelet vesz igénybe, ebből a hasznos órajelek száma 32, így a busz kihasználtsága csupán 54 %.

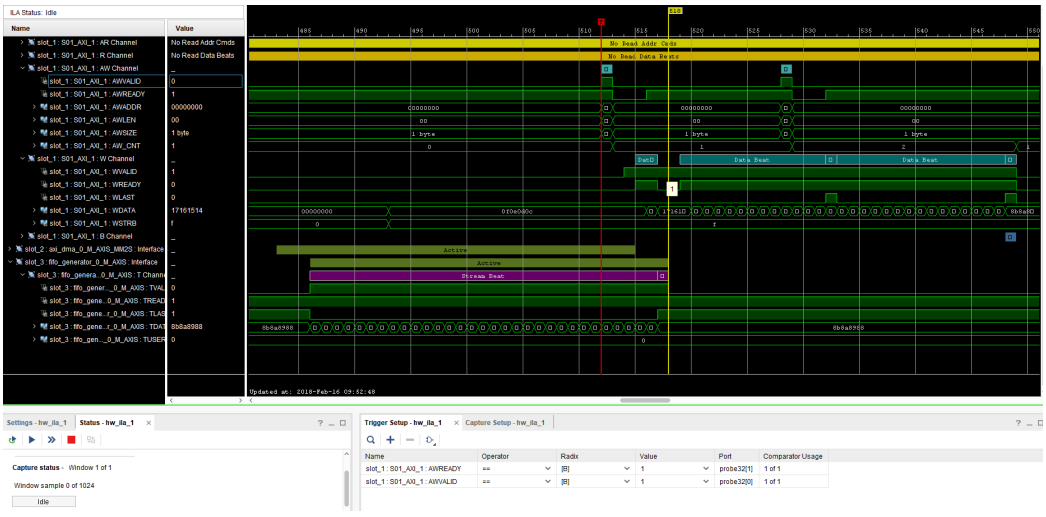


2.3. Ábra – *DMA adatmozgatás memóriából FIFO-ba*

A 128 byte átvitele két bust-ben történik, ez a két RLAST pulzusból látható, valamint a DMA beállításánál 16 hosszú burst-öt írtunk elő. A címzés és a tényleges adatátvitel közötti késleltetés órajelben számolva nagyjából 27, mert a tranzakciónak egy Interconnect-en is át kell esnie. Ha a DMA-t közvetlenül AXI HP-n a PS-be kötnénk, akkor csökkenthető lenne a késleltetés. Az AXI4 és az AXI4 Stream buszok közötti késleltetés 4 órajelperiódusnyi, amelyet a két buszrendszer (AXI és Stream) közötti konverzió okozza.

2.4.2. S2MM irány vizsgálata egyetlen 128 byte-os átvitel esetén

Az S2MM irányt a 2.4. ábra mutatja be. Nem tapasztaltunk lényeges eltérést az MM2S irányhoz képest. A DMA először beolvassa a FIFO tartalmát, majd AXI írási ciklusokat indít a PS felé, az AW és W csatornák segítségével.



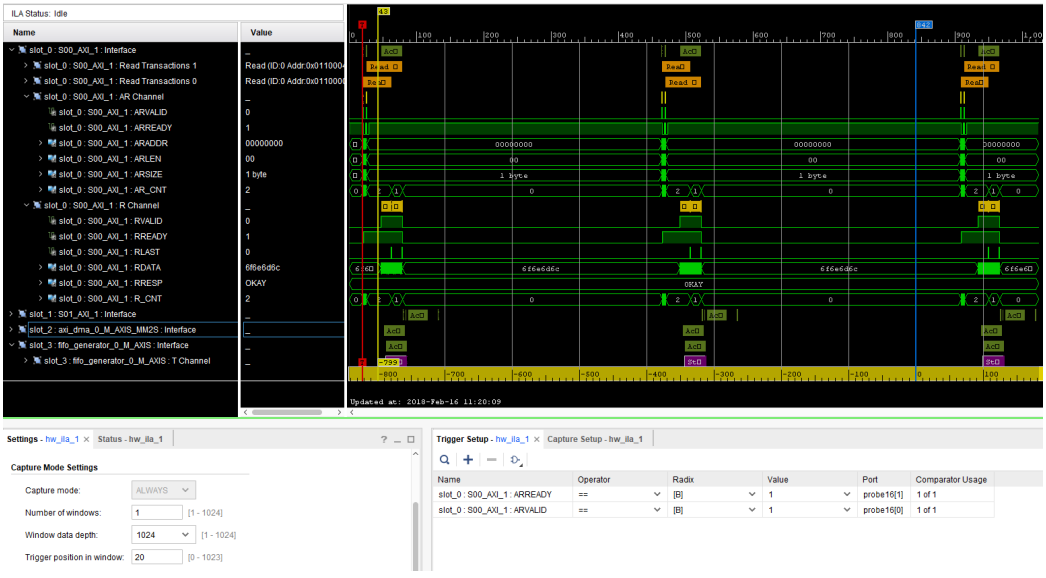
2.4. Ábra – *DMA adatmozgatás FIFO-ból memóriába*

2.4.3. MM2S irány vizsgálata több 128 byte-os átvitel esetén

Több 128 byte-os átvitelt mutat az a 2.5. ábra. A ChipScope-ban a triggerpozíciót a mintatár elejére kellett állítanunk, hogy beleférjen két FIFO írás a logikai analizátor mintatárába.

2.5. Ábra – *Két egymást követő adatmozgatás memóriából FIFO-ba*

Nagyjából 788 órajel telik el egy DMA befejezése és a következő DMA átvitel megkezdése között. Ezt úgy tudtuk csökkenti, hogy kivettük a data_check() ellenőrzést a kódból, valamint a fordítónak megmondtuk, hogy Release módban fordítson. Ennek az eredményét a 2.6. ábra mutatja.

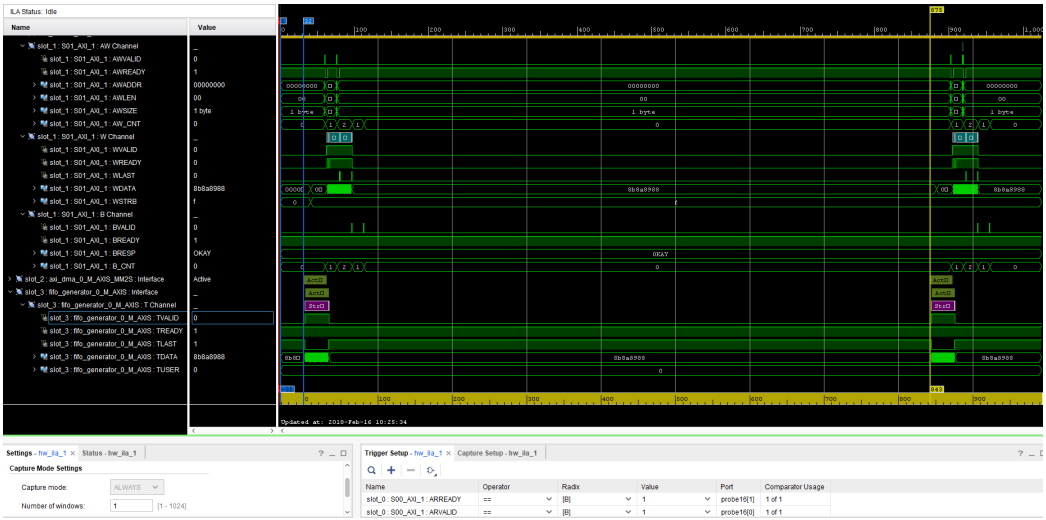


2.6. Ábra – *Egymást követő adatmozgatások memóriából FIFO-ba, csökkentve az „üres” órajel periódusok számát*

További csökkentési lehetőség lett volna még a Cache memória kikapcsolása az A9-es alrendszerben.

2.4.4. S2MM irány vizsgálata több 128 byte-os átvitel esetén

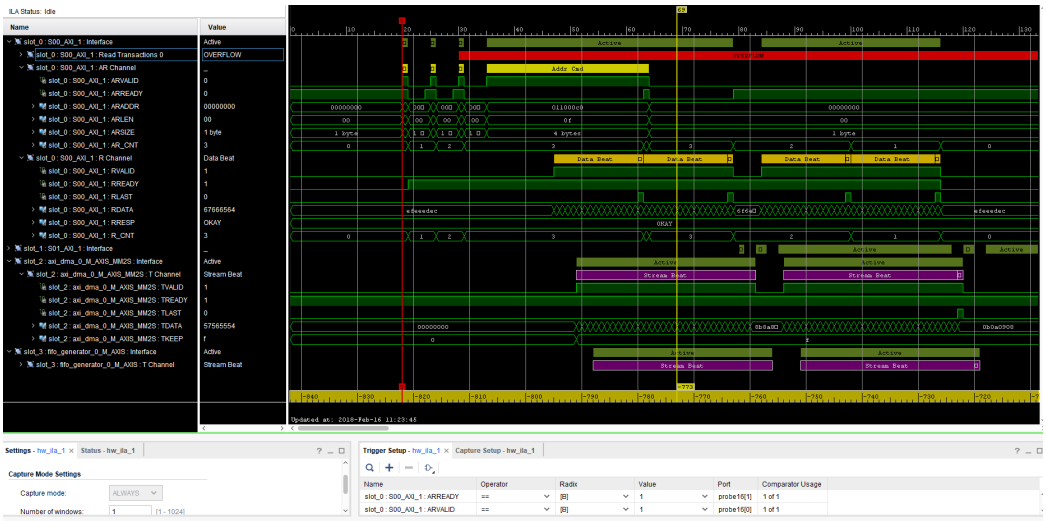
Több 128 byte-os átvitelt mutat az a 2.7. ábra. Nem tapasztaltunk lényeges különbséget az MM2S irányhoz képest.



2.7. Ábra – *Két egymást követő adatmozgatás FIFO-ból memóriába*

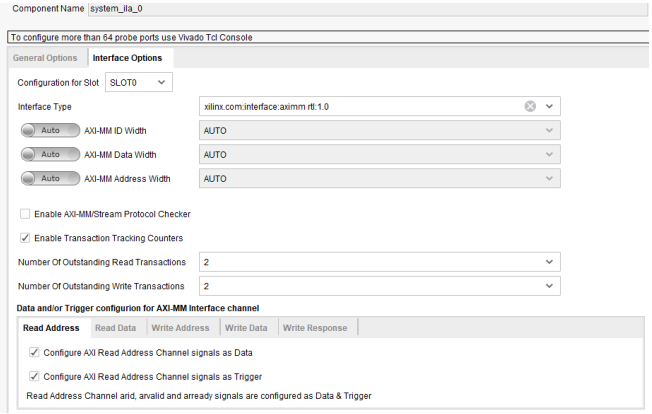
2.4.5. MM2S irány egyetlen 1024 byte-os blokk („packet”) átvitele során

Utolsó feladatként egy 1024 byte-os blokk átvitelét vizsgáltuk meg. A kódban a 32-es sort kellett átírni 0x100-ra, ezzel mondtuk meg a DMA-nak, hogy egy 1024 byte-os blokkot szeretnénk átvinni.



2.8. Ábra – *Egy nagyobb, 1024 byte-os blokk mozgatása memóriából FIFO-ba*

Mivel a DMA 16 hosszú burst-re lett konfigurálva, így csak 4 burst alatt tudta az 1024 byte-ot átvinni (1 burst az 16 szó, amely 64 byte, valamint az 1024 byte az 256 szónak felelt meg). A 2.8. ábrán észrevehető, hogy 4 Outstanding olvasási tranzakciót küld ki és erre a ChipScope AXI protokoll figyelője hibát jelez, mert az legfeljebb csak 2 Outstanding tranzakcióra lett alapból beállítva, ahogy az a 2.9. ábrán látszik.



2.9. Ábra – *A 2.8. ábrán látható OVERFLOW error magyarázata: SLOT0 a System ILA-n 2-es Outstanding Transactions-re van beálltva*

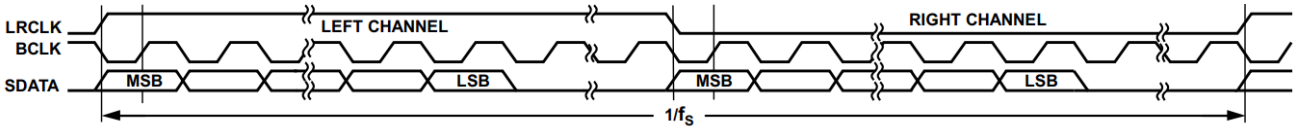
3. fejezet

Audio interfész (I2S) megvalósítása

A harmadik alkalommal egy Slave I2S interfészt kellett megvalósítanunk, amellyel az Audio CODEC által mintavételezett adatokat tudtuk párhuzamosítani. Ehhez egy egyszerű léptetőregisztert használtunk, néhány éldetektort és metastabilszűrőket. A metastabilszűrők (sorba kötött D flip-flop-ok) azért kellettek, mert az FPGA logikája és a CODEC logikája két eltérő órajeltartományban helyezkedtek el, amelyek aszinkronnak tekinthetők, hiszen sem fázisban sem frekvenciában nem volt kapcsolat közöttük. Így az onnét érkező LRCLK és BCLK jeleket először az FPGA órajeltartományára kellett szinkronizálni, máskülönben egy aszinkron módon érkező jelváltozás (az apertúra időben) tönkretehette volna a fogadó oldali flop-okat.

3.1. Right justified I2S átvitel hullámformája

A 3.1. ábrán látható az I2S időzítési diagramja. Látható, hogy az Audio CODEC master üzemmódban használva az adatot a BCLK lefutóélére adja ki és az LRCLK egyetlen periódusa alatt mindkét csatorna adata átvitelre kerül.



3.1. Ábra – *I2S időzítési diagramja Right justified üzemmód esetén*

Az LRCLK egyenlő a mintavételi frekvenciával. Azért Right justified módban használtuk az Audio CODEC-et, mivel így elegendő volt egyetlen 24 bites léptető regisztert definiálni az adatok soros párhuzamos átalakítására, hiszen egy csatorna 24 bitje szépen előáll a LRCLK élváltások után. Persze, az interfész mindig 32 bitet küld ki mindegyik csatornára, de az először küldött 8 dummy bit kiléptetésre kerül a léptetőregiszterből. Nekünk gyakorlatilag csak az LRCLK-n előforduló jelváltásokat kellett figyelnünk, valamint a BCLK felfutóélét, hiszen ekkor már biztosan stabilizálódott a CODEC által küldött bit.

3.2. A megvalósított modul HDL kódja

```
library ieee                    ;
use     ieee.std_logic_1164.all ;
use     ieee.numeric_std.all    ;

--------------------------------------------------------------------------------
entity i2s is
    port(
        clk             :   in  std_logic;
        rst             :   in  std_logic;                      -- Active LOW

        en              :   in  std_logic;                      -- Enable from PS

        adc_data        :   out std_logic_vector(23 downto 0);  --
        adc_valid_l     :   out std_logic;                      -- Signals to Block Design
        adc_valid_r     :   out std_logic;                      --

        lrclk           :   in  std_logic;                      --
        bclk            :   in  std_logic;                      -- Signals from Audio CODEC
        sdi             :   in  std_logic                       --
    );
end entity i2s;
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
architecture rtl of i2s is

    signal  shr         :   std_logic_vector(23 downto 0);  -- Shift register

    signal  re_on_bclk  :   std_logic;                  -- Rising edge

    signal  re_on_lrclk :   std_logic;                  -- Rising edge
    signal  fe_on_lrclk :   std_logic;                  -- Falling edge

    signal  sync_bclk   :   std_logic_vector(2 downto 0);
    signal  sync_lrclk  :   std_logic_vector(2 downto 0);

begin

    ----------------------------------------------------
    -- Shift register
    process(clk)    is
    begin

        if(rising_edge(clk))    then


            -- If there was a rising edge and it is enabled, because data is changing at falling-edge on 'sdi'
            if(re_on_bclk='1')  then
                shr <= std_logic_vector(shr(22 downto 0) & sdi);
            end if;

        end if;

    end process;
    ----------------------------------------------------
    ----------------------------------------------------
    adc_data    <= shr;                                 -- Driving output directly from shiftregister

    -- R channel is valid when it is enabled,
    -- When there was a rising edge on LRCLK and the double flop on the LRCLK has settled to '1'
    adc_valid_r <= '1'  when (sync_lrclk(1) = '1') and (re_on_lrclk = '1') and (en = '1')   else '0';


    -- Same here,
    -- but L channel is valid when there was a respective falling-edge and the synchronizer chain is settled to '0'
    adc_valid_l <= '1'  when (sync_lrclk(1) = '0') and (fe_on_lrclk = '1') and (en = '1')   else '0';
    ----------------------------------------------------
    ----------------------------------------------------
    -- BCLK
    process(clk)    is
    begin
        if(rst = '0')  then
            sync_bclk <= B"000";

        elsif(rising_edge(clk)) then
            sync_bclk(0)    <= bclk;                    --
            sync_bclk(1)    <= sync_bclk(0);            -- Double flopping
            sync_bclk(2)    <= sync_bclk(1);

        end if;
    end process;

    re_on_bclk  <= not(sync_bclk(2)) and sync_bclk(1);  -- Rising-edge detector
    ----------------------------------------------------
    ----------------------------------------------------
    -- LRCLk
    process(clk)    is
    begin
        if(rst = '0')  then
            sync_lrclk <= B"000";

        elsif(rising_edge(clk)) then
            sync_lrclk(0)    <= lrclk;                  --
            sync_lrclk(1)    <= sync_lrclk(0);          -- Double flopping
            sync_lrclk(2)    <= sync_lrclk(1);

        end if;
    end process;

    fe_on_lrclk  <= sync_lrclk(2) and not(sync_lrclk(1));   -- Rising-edge detector
    re_on_lrclk  <= not(sync_lrclk(2)) and sync_lrclk(1);   -- Rising-edge detector
    ----------------------------------------------------
end architecture rtl;
```

3.3. A testbench HDL kódja

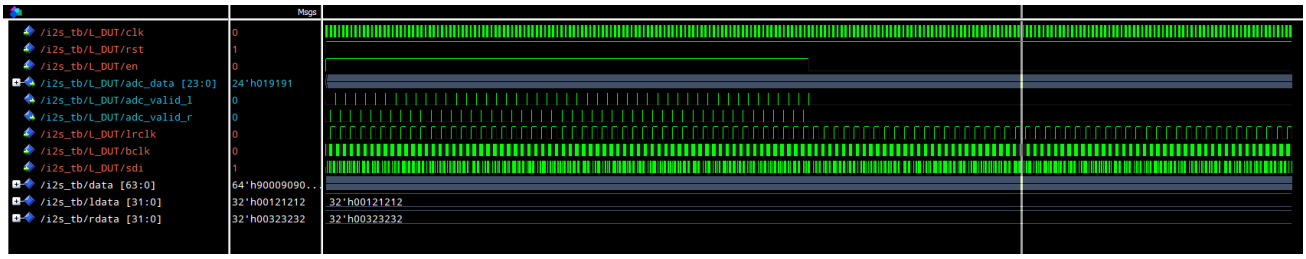
```
library ieee                            ;
library std                 ;
library work                ;
-----------------------------
use std.env.all             ;
use ieee.numeric_std.all        ;
use ieee.std_logic_1164.all     ;
-----------------------------
-------------------------------------------------------------------------------------------
entity i2s_tb is
end entity;
-------------------------------------------------------------------------------------------

architecture bhv of i2s_tb is

    constant    clk_per_c:          time        := 10 ns            ;   -- System clock: 100 MHz
    constant    lrclk_per_c:        time        := 1e6 ns / 96      ;   -- LRCLK = fsamp = for simulation purposes set to 96 kHz
    constant    bclk_per_c:         time        := lrclk_per_c / 64 ;   -- On one LRCLK the two channels data will be sent out: (1) 32 L-bits (0) 32 R-bits

    signal      tb_clk:             std_logic   := '0';
    signal      tb_rstn:            std_logic   := '1';

    signal      tb_en:              std_logic   :=  '0';
    signal      tb_adc_data:        std_logic_vector(23 downto 0);
    signal      tb_adc_valid_l:     std_logic;
    signal      tb_adc_valid_r:     std_logic;
    signal      tb_lrclk:           std_logic   :=  '0';
    signal      tb_bclk:            std_logic   :=  '0';
    signal      tb_sdi:             std_logic   :=  '0';

    signal      data:               std_logic_vector(63 downto 0);

    signal      ldata:              std_logic_vector(31 downto 0):= X"00121212";
    signal      rdata:              std_logic_vector(31 downto 0):= X"00323232";

begin

    clk_gen:        process is begin    wait for clk_per_c/2;   tb_clk      <= not tb_clk;      end process;
    lrclk_gen:      process is begin    wait for lrclk_per_c/2; tb_lrclk    <= not tb_lrclk;    end process;

    ---------------------------------------------------------------------------------------------
    bclk_gen:       process is
        variable    i   :   integer := 0;
    begin

        wait on tb_lrclk;   -- Generating the bclk clock synchronized to the lrclk

        ------------------------------------------------------------
        i := 0;
        loop

            if i < 64   then
                wait for bclk_per_c/2;  tb_bclk    <= not tb_bclk;

                i := i + 1;
            else
                exit;

            end if;
        end loop;
        ------------------------------------------------------------
    end process;
    ------------------------------------------------------
    rstn_gen:       process is begin
        ----------------------------------
        wait for 1 ns;                      --
        tb_rstn     <= '0';                 --
                                            -- Reset
                                            --
        wait for 1 ns;                      --
        tb_rstn     <= '1';                 --
        ----------------------------------
        ----------------------------------
        wait until rising_edge(tb_clk);  --
                                         --
        tb_en <= '1';                    -- Enabling 'en'
                                         --
        wait for 500 us;                 --
        ----------------------------------
        ----------------------------------
        wait until rising_edge(tb_clk); --
                                        --
        tb_en <= '0';                   -- Disabling 'en'
                                        --
        wait for 500 us;                --
        ----------------------------------
        stop;
    end process;
    ------------------------------------------------------
    ------------------------------------------------------
    process(tb_bclk, tb_rstn)  is
    begin
        if(tb_rstn= '0')    then
            data    <= std_logic_vector(rdata & ldata);

        elsif(falling_edge(tb_bclk))   then                         -- 'sdi' is driven on falling edge !!!
            data    <= std_logic_vector(data(62 downto 0) & data(63));
        end if;
    end process;

    tb_sdi <= data(63);                                 -- Driving the test data pattern

    L_DUT:  entity work.i2s(rtl)
    port map(
        clk         =>  tb_clk,
        rst         =>  tb_rstn,
        en          =>  tb_en         ,
        adc_data    =>  tb_adc_data   ,
        adc_valid_l =>  tb_adc_valid_l,
        adc_valid_r =>  tb_adc_valid_r,
        lrclk       =>  tb_lrclk      ,
        bclk        =>  tb_bclk       ,
        sdi         =>  tb_sdi
    );
end architecture;
```

3.4. Szimulációs hullámforma

A HDL alapú lineáris testbench szimulációjához a Mentor Graphics cég QuestaSim 10.4e szimulátorát használtuk az alábbi TCL szkript segítségével.

```
if [file exists "work"] {vdel -all}
vlib work

# Compile RTL
vcom -93 -work work i2s.vhd
# Compile TB
vcom -2008 -work work i2s_tb.vhd

# Optimize TB
vopt i2s_tb -o i2s_tb_opt  +acc

# Simulate TB
vsim i2s_tb_opt

# Misc.
set NoQuitOnFinish 1
onbreak {resume}
log /* -r
run -all
```

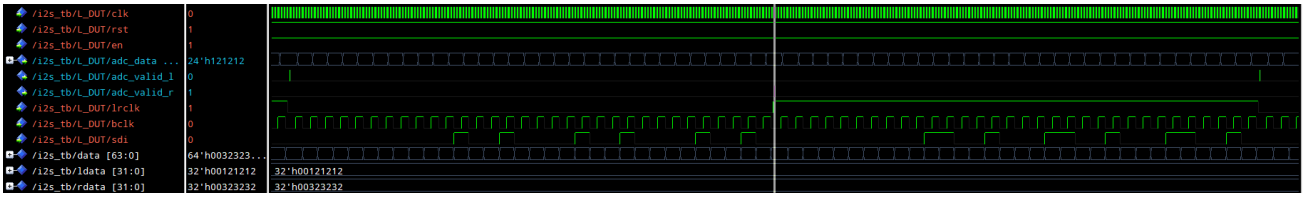
A 3.2. ábrán látható a megvalósított I2S Slave interfész viselkedésének hullámformája. A hullámformán megfigyelhető, hogy az **en** bemenet 0-ba húzása után leáll a két csatorna érvényességét jelző kimeneti jelek billegtetése.



3.2. Ábra – *Az I2S interfész hullámformája, szemléltetve az **en** bemenet hatását*

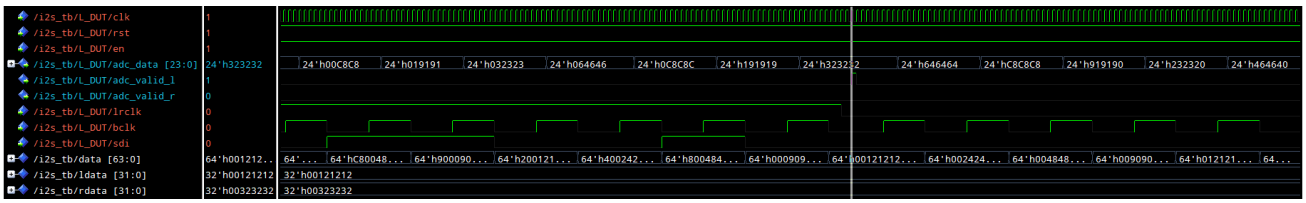
Az **ldata** és **rdata** tesztadatok, amelyek a **data** jelben kerültek egymáshoz illesztésre. E 64 bites jel sorosításával modelleztük az audio CODEC **sdin** portját.

A 3.3. ábrán pedig a Right justified üzemmód már említett előnyét ábrázoltuk. Az első kurzornál látható az **adc_valid_r** pulzus megjelenésekor, az **adc_data** már a megfelelő 24 bites értéket tartalmazta.



3.3. Ábra – *A Right justified üzemmód viselkedésének szimulációja*

A viselkedésről készítettünk egy közelebbi képet is, amely a 3.4. ábrán szerepel. Észrevehető, hogy a helyes 24 bites adat jelenik meg, az utolsó **bclk** felfutó élét követő 2 órajel periódussal, az **adc_data** kimeneten.



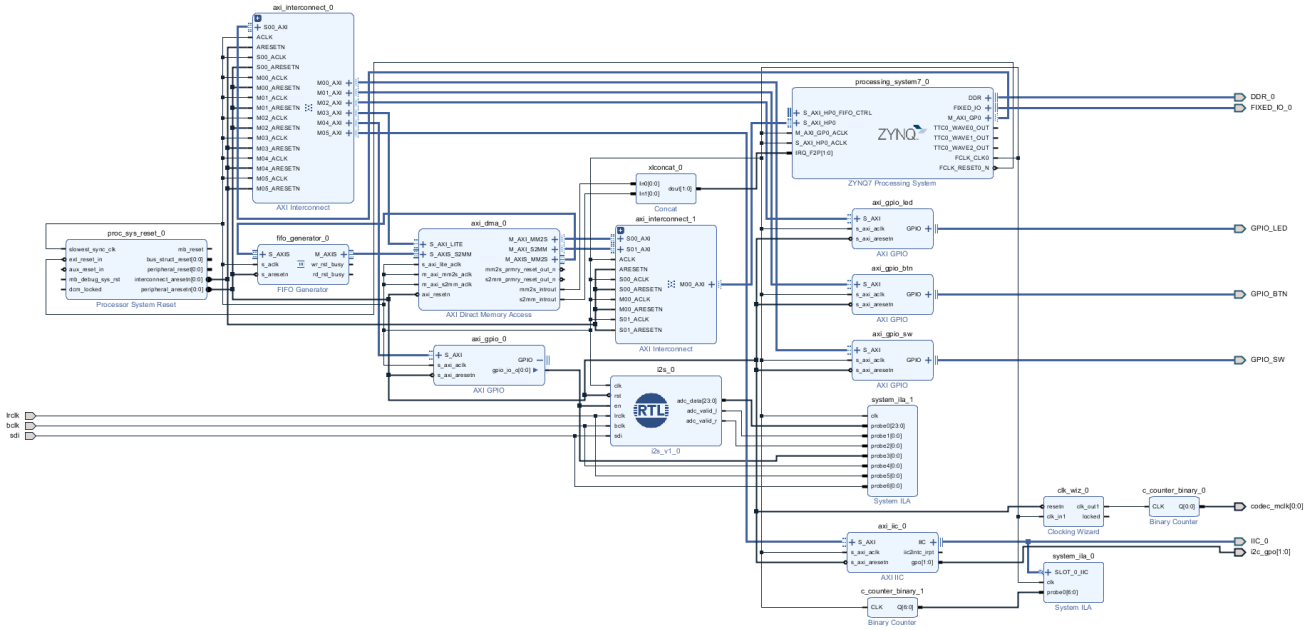
3.4. Ábra – *A 3.3. ábra közelebbről*

4. fejezet

Audió CODEC interfész

A negyedik alkalommal a blokk dizájnunk először kiegészült egy I2C konfigurációs interfésszel, amellyel a kártyán lévő Audió CODEC-et tudtuk konfigurálni, majd a 3. laboron megtervezett I2S interfészt is beépítettük a rendszerbe.

4.1. A rendszer blokkvázlata



4.1. Ábra – *A Block Design-ban összeállított rendszer blokkdiagramja*

Address Editor

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
axi_gpio_0	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
axi_gpio_btn	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_led	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_sw	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_iic_0	S_AXI	Reg	0x4160_0000	64K	0x4160_FFFF
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

4.2. Ábra – *Az összeállított rendszer memóriatérképe*

4.2. Konfigurációs interfész (I2C) megvalósítása

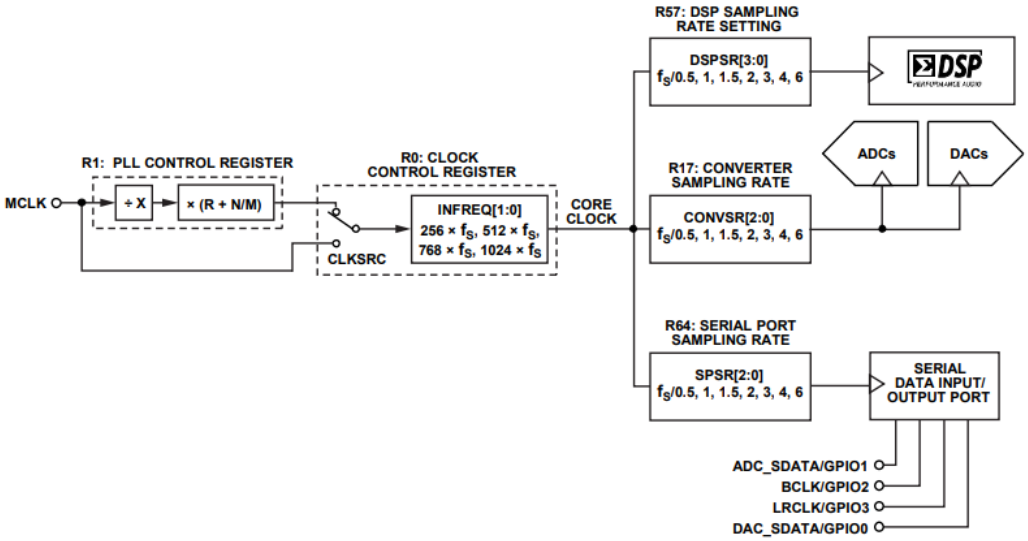
4.2.1. CODEC MCLK órajel generálása

Az Audió CODEC Master órajelét, MCLK, egy Clocking Wizard segítségével állítottuk elő az FPGA konfigurálható logikájában, a 4.1. ábrán ez **codec_mclk[0:0]** néven szerepel. Az előbb említett órajel frekvenciáját úgy kellett meghatároznunk, hogy eleget tegyen az időzítési kritériumoknak, amely a 4.3. ábrán látható, valamint illeszkednie kellett a CODEC-ben lévő órajelgeneráló hálózathoz.
A cél a 96 kHz-es mintavételi frekvencia volt, tehát a 4.4. ábrán lévő f_s-nek kellett 96 kHz-nek lennie.

Table 7. Digital Timing

Parameter	Limit		Unit	Description
	t_{MIN}	t_{MAX}		
MASTER CLOCK				
t_{MP}	74	488	ns	MCLK period, 256 × f_S mode.
t_{MP}	37	244	ns	MCLK period, 512 × f_S mode.
t_{MP}	24.7	162.7	ns	MCLK period, 768 × f_S mode.
t_{MP}	18.5	122	ns	MCLK period, 1024 × f_S mode.

4.3. Ábra – *Az MCLK-ra vonatkozó időzítési kritériumok*



4.4. Ábra – *A CODEC-ben lévő órajel és mintavételi frekvencia generáló hálózat blokkdiagramja*

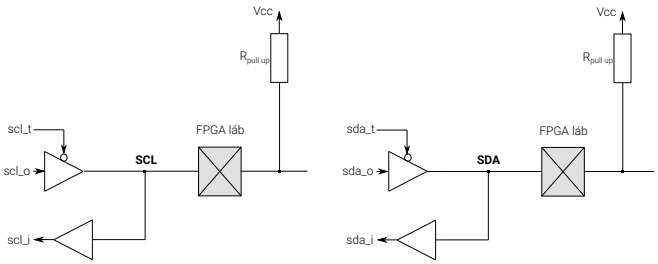
Ehhez a Clocking Wizard-ban $1024 \cdot f_s$ nagyságú órajelet generáltunk, amely így **98,304 MHz**-en pörgött. Erről járattunk egy számlálót, amelynek a 0. bitjét használtuk fel a **codec_mclk[0:0]** jel előállításához, és így egy $512 \cdot f_s$ frekvenciájú (**49,152 MHz**) négyszögjelet kaptunk. Ez a jel megfelelő volt, hiszen teljesítette a 18,5 ns-os időzítési követelményt. Valójában ezért kellett ezt megtennünk, hiszen az alap, Clocking Wizard-ban előállított órajel nem tudta volna teljesíteni.

Ezt az ábrán lévő MCLK bemenetre vezettük, amely a PPL-t kikerülve került rá a Clock Control Register bemenetére. Itt az $1024 \times f_s$ módot állítottuk be, így a kimenetén előálló Core Clock $1/2 \cdot f_s$-re adódott. Annak érdekében, hogy a megfelelő mintavételi frekvenciát megkapjuk ebből, ahhoz még a soron következő regiszterekben az $f_s/0.5$ módokat kellett használnunk.

4.2.2. CODEC I2C interfész

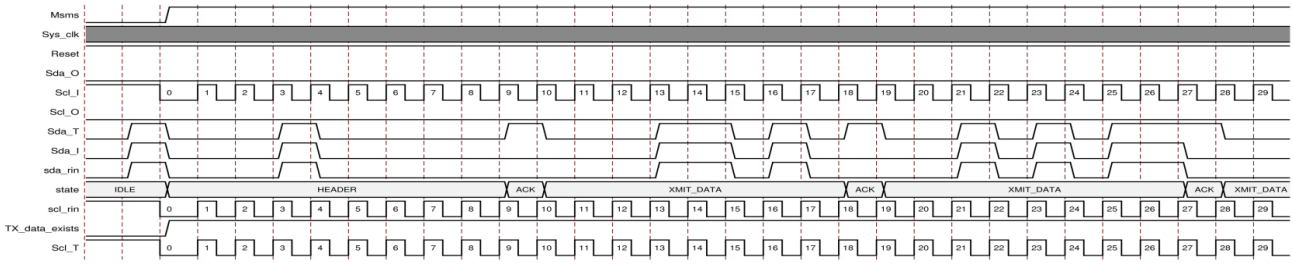
Az AXI I2C interfész írási és olvasási ciklusait mutatják a 4.6. és a 4.8. ábrák. Ezeken az MSMS jel a Master/Slave Mode Select-et jelenti, azaz amikor ezt a jelet a Control Register-ben átbillentik 0-ból 1-be, akkor az intefész Master üzemmódba kapcsol és generál egy START feltételt az I2C buszon. Ha ezt a bitet visszateszik 0-ba a Control Register-en keresztül, akkor, az előbbi gondolatmenetnek megfelelően, viszont STOP feltételt fog előidézni a modul a kimenetén.

Az **_i**, **_o** és **_t** utótagok az időzítési diagram **sda** és **scl** jelein az FPGA pad-ek előtt elhelyezett tristate bufferekre utalnak, ezt szemlélteti a 4.5. ábra.



4.5. Ábra – *Az I2C átvitelhez tartozó tristate bufferek*

Az AXI I2C modul nem biztosított tristate buffereket, amelyek a vezetékek kétirányú vezérléséhez kellettek, de az FPGA IO blokkjaiban voltak ilyenek, így azokat tudtuk használni, vagyis a Vivado szintézere tudta használni. Valamint, a lentebb lévő időzítési diagramokon jól látszódik az, hogy maga **a modul aktív alacsony** működésű, azaz, ha valamit jelezni akar a vonalakon, akkor az **sda_t** kimenetet 0-ba húzza, ezáltal képes az FPGA lábat egy ideig nullába húzni, amelyet az **sda_o** fix logikai 0-n való tartásával ér el. Amikor nincs kommunikáció az I2C csatornán, akkor a két vezetéknek logikai magasba kell kerülniük, ezt a fejlesztői kártyán lévő felhúzó ellenállások biztosítják.



4.6. Ábra – *Az I2C írási ciklusának idődiagramja*

A 4.6. ábrán az I2C írási ciklus kezdetét a Start feltétel és az MSMS 1-be billentése jelenti. Tehát, generál egy alacsony pulzust az **sda_t**-n. Ezután elindul a HEADER átvitele, amely a 7 bites Eszköz azonosító MSB bitjével kezdődik és az átvitel 8. bitje azt jelzi, hogy mit akar a Slave-vel csinálni a Master. Jelen esetben írni, így azt 0-ba húzza. A Slave-nek erre a következő **scl_o** periódusban reagálnia kell. Ilyenkor a Master elengedi az **sda**-t, hogy a Slave 0-ba tudja húzni, ezzel jelezvén, hogy észrevette magát. Majd ezt követően a Master bájtosával kezdi átküldeni az adatokat, ezt jelzi az XMIT_DATA állapot.

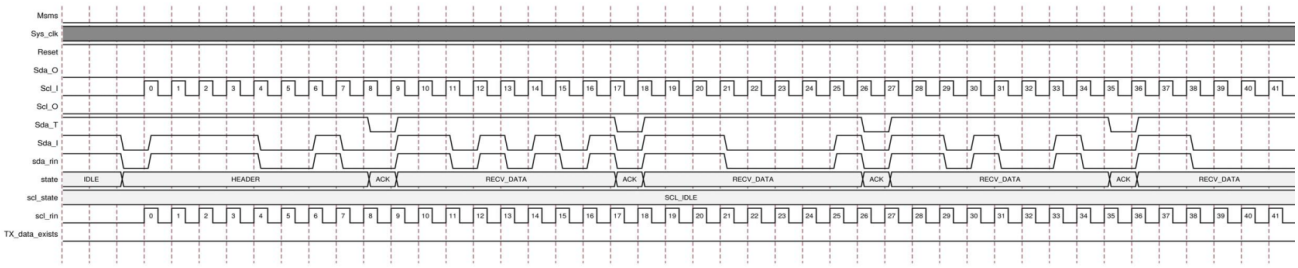
A CODEC lehetséges Eszköz azonosítóit a 4.7. ábra mutatja be. Az ábra hibás, mert valójában a Bit 0 az az MSB és így tovább. Mi a 0b0111000 címet használtuk, így az AXI I2C GPO kimenetére 0-t kellett írnunk, az inicializálás során. A CODEC regisztereit 16 bites címekkel értük el.

Table 21. ADAU1761 I²C Address and Read/$\overline{\text{Write}}$ Byte Format

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
0	1	1	1	0	ADDR1	ADDR0	R/$\overline{\text{W}}$

4.7. Ábra – *A CODEC lehetséges eszköz azonosítói*

A 4.8. ábrán az I2C olvasási hullámformát tüntettük fel. Tetszőleges címről történő olvasás esetén először el kell küldeni a Slave-nek az olvasási címet egy I2C írás formájában, majd utána lehet I2C olvasási ciklust kezdeményezni. Az olvasási ciklust ismételt Start feltétellel lehet elindítani. A RECV_DATA csomagokra a Master-nek kell reagálnia.



4.8. Ábra – *Az I2C olvasási ciklusának időidiagramja*

A hullámformán már csak az olvasási ciklust látjuk, kezdve a HEADER bájttal, ami az Eszköz azonosítót és hozzáférés jellegét tartalmazza. Majd ezt követően kezdi a Slave adogatni a bájtokat, amire a Masternek kell közben reagálnia.

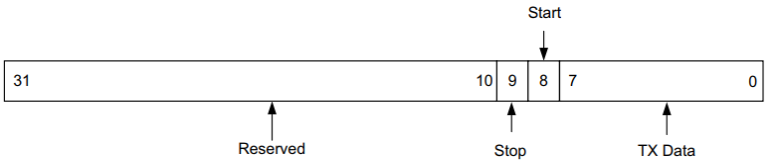
4.3. Audió CODEC konfiguráció és ellenőrzés

4.3.1. Xilinx I2C periféria programozása

Az I2C perifériát ún. Dynamic Controller Logic Flow-ban használtuk, amely a Standard Controller Logic Flow üzemmódhoz képest annyiban tér el, hogy nem kell egyesével végigmenni és végrehajtani az I2C ciklusokhoz szükséges lépéseket, hanem elég ha megadjuk az egyes transzferekhez tartozó alapvető adatokat (írás/olvasás és esetleg írási adat), ezt észreveszi egy FSM, és végigmegy a megfelelő utasításokon.

Ehhez a módhoz elegendő a Státusz regiszter, a Kontroll regiszter, a TX-FIFO regisztert és az RX-FIFO regisztert használni.

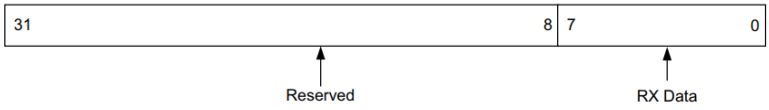
A TX-FIFO regiszter az I2C periféria báziscíméhez képest a 0x108-as címen érhető el és a bitkiosztását a a 4.9. ábra mutatja be. Látható, hogy a TX-FIFO regiszter 32 bites, de ebből csak az alsó 10 bit használható. Az alsó 8 bit azt a bájtot



4.9. Ábra – *A TX_FIFO regiszter bitkiosztása*

reprezentálja, amit a periféria Master ként az **Sda** vonalra küld, vagy eszköz azonosító + transzfer jelleg vagy írási adat. A 8-as bit a Start feltétel generálásáért felelős, míg a 9-es pedig a Stop feltételért.

Az RX-FIFO regiszter ennél egyszerűbb, mivel az csak a fogadott bájtot tárolja az alsó 8 bitjén és a 0x10C-n érhető el, ezt tükrözi a 4.10. ábra is.



4.10. Ábra – *A RX_FIFO regiszter bitkiosztása*

4.3.2. CODEC regiszterek beállítása

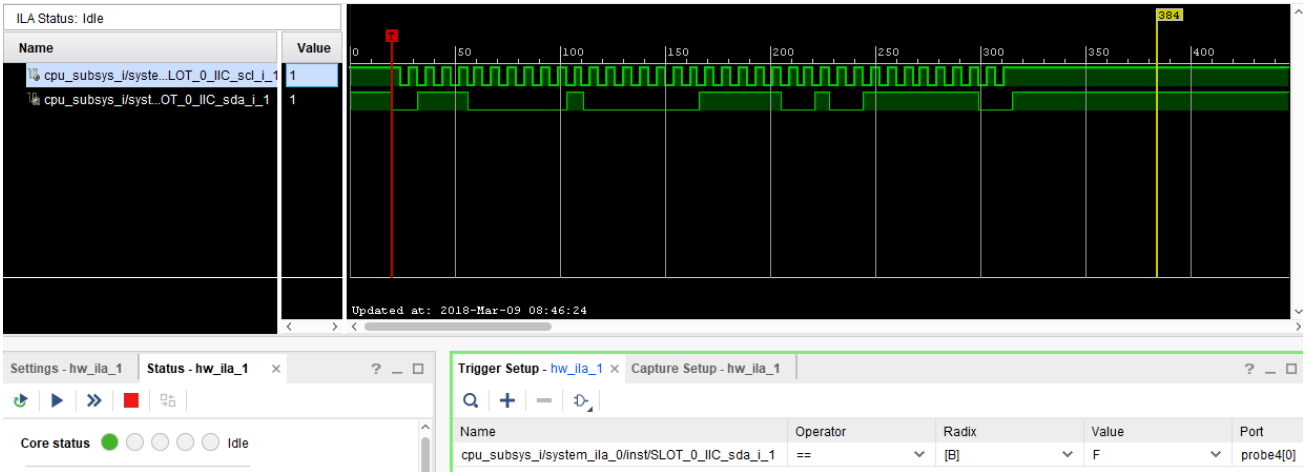
Ahhoz, hogy a CODEC a 4.2.1. szakaszban bemutatott és meghatározott órajelen működjön, valamint az I2S interfész Right-justified üzemmódban adja ki az adatokat, a 4.1. táblázatban szereplő regiszterek értékeit kellett beállítani.

Regiszter cím	Érték	A beállított érték hatása
0x4000	0b0111	**Clock Control Register, R0:** 1024× f_s és a Core clock-ot engedélyezi, az órajel forrását az MCLK lábról veszi
0x4015	0b00001001	**Serial Port Control 0, R15:** LRPOL bitet 1-be raktuk, hogy felfutó él triggerelje a bal csatorna adatának kezdetét az LRCLK jelen
	0b00001001	**Serial Port Control 0, R15:** MS bitet 1-be, hogy Masterként működjön
0x4016	0b00000010	**Serial Port Control 1, R16:** LRDEL biteket úgy állítottuk be, hogy 8 BCLK-nyi adatkésleltetés legyen az LRCLK élétől számítva
0x4017	0b00000110	**Converter Control 0, R17:** CONVSR-et 110-ra állítottuk, ezzel tudtuk megadni, hogy a használt mintavételezési frekvencia 96 kHz
0x40EB	0b000	**DSP Sampling Rate Setting, R57:** a DSPSR-t 000-ra állítottuk, hogy tudassuk a SigmaDSP maggal, hogy 96 kHz-cel mintavételezünk
0x40F8	0b110	**Serial Port Sampling Rate, R64:** az SPSR itt állítottuk be, hogy a soros portot 96 kHz-cel mintavételezze

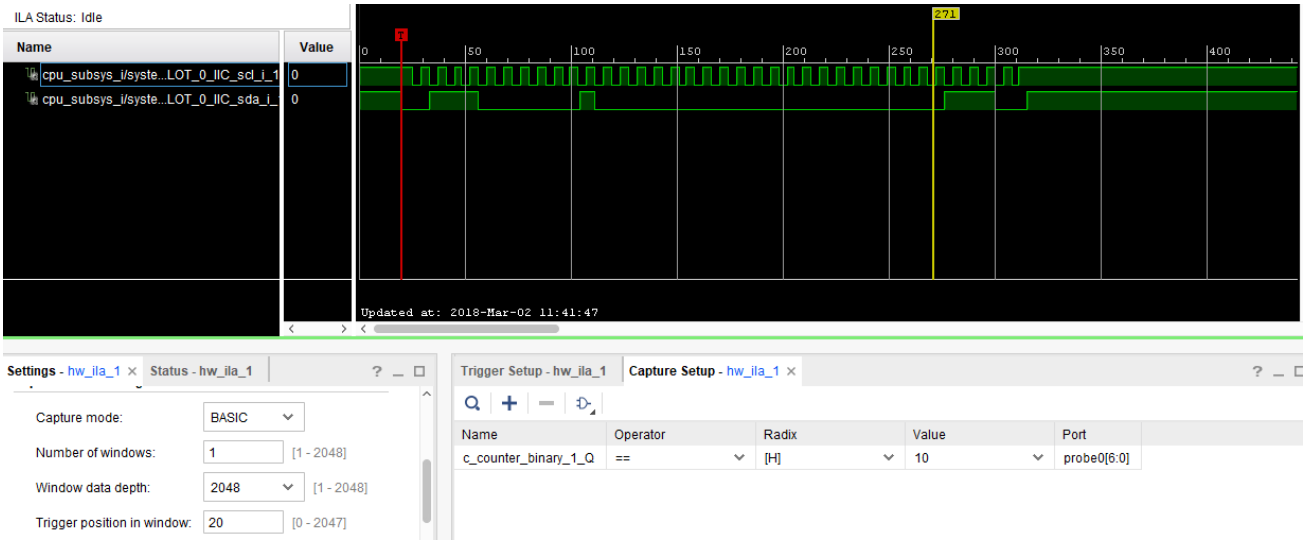
4.1. Táblázat – *A mérési útmutatóban meg nem adott regiszterek beállításai*

4.3.3. I2C átvitel ellenőrzése ChipScope-ban

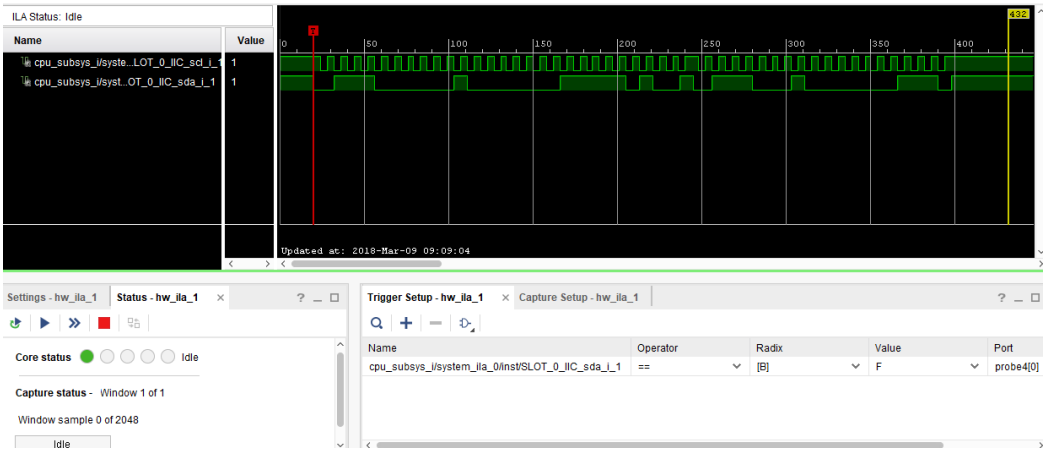
Miután felkonfiguráltuk az I2C perifériát megvizsgáltuk az I2C ciklusokat a logikai analizátorban. Egy szabadon futó számláló segítségével vezéreltük a mintavételezést, hogy ne teljen meg felesleges adatokkal a mintatár, valamint az **Sda** lefutó élére triggereltük az analizátort.



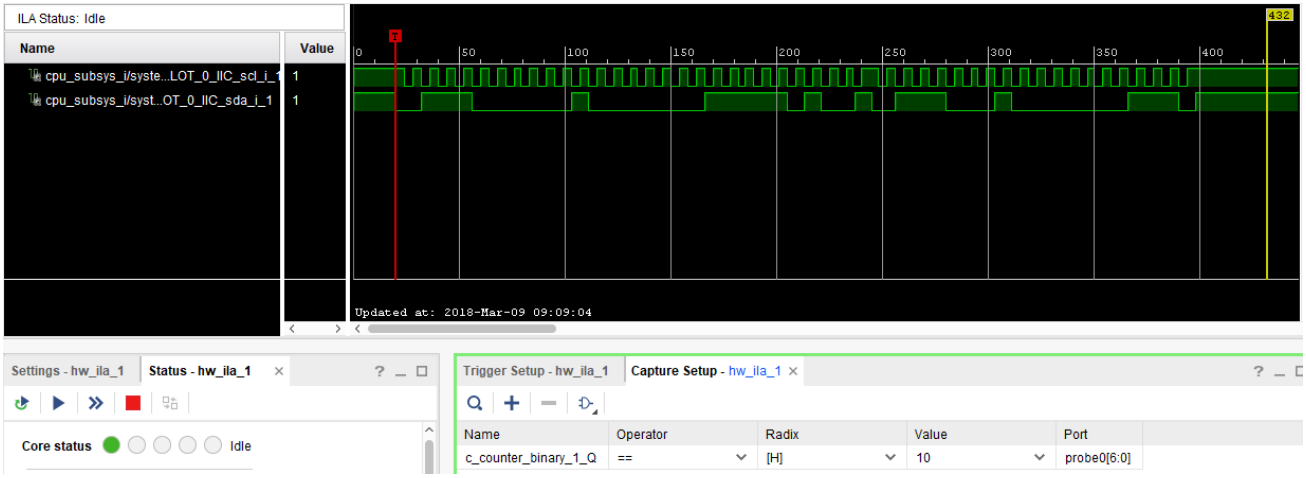
4.11. Ábra – *I2C írási hullámforma, trigger beállításokkal*



4.12. Ábra – *I2C írási hullámforma, capture beállításokkal*



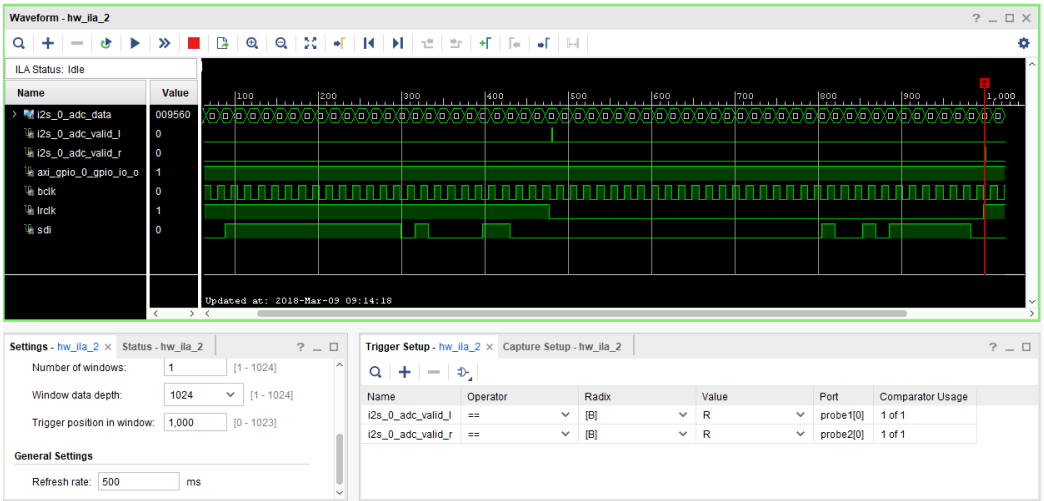
4.13. Ábra – *I2C olvasási hullámforma, trigger beállításokkal*



4.14. Ábra – *I2C olvasási hullámforma, capture beállításokkal*

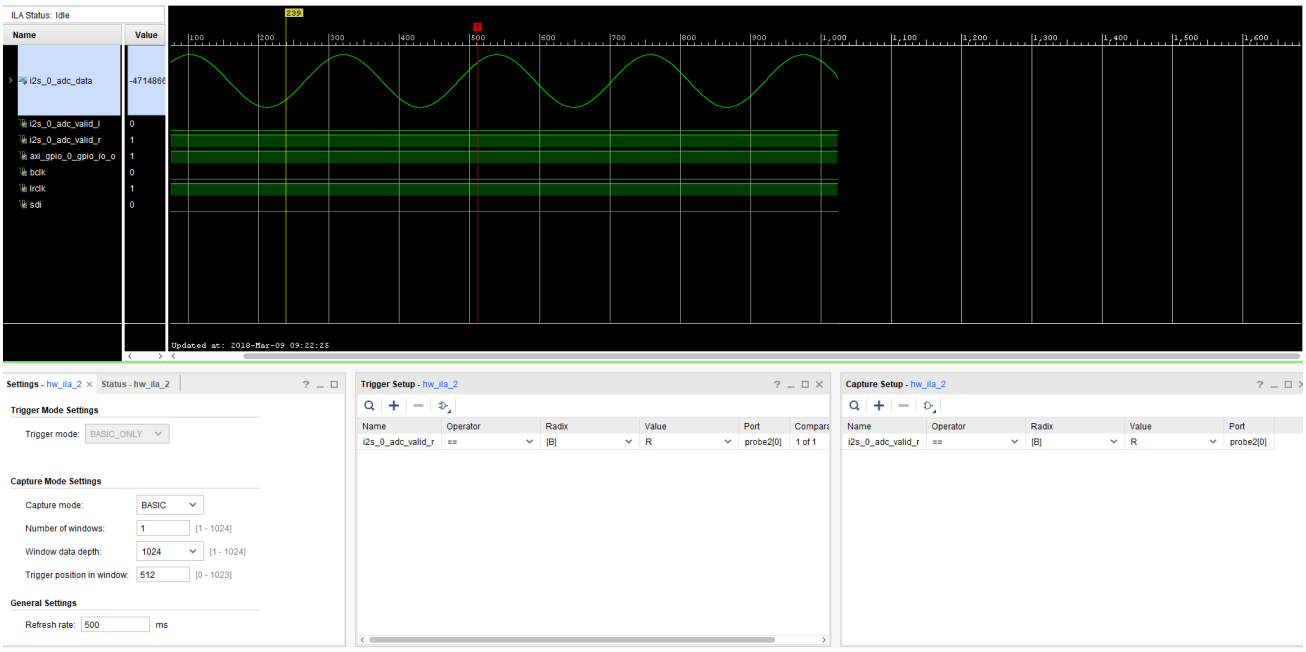
4.3.4. I2S interfész ellenőrzése

Az I2S interfész ellenőrzéséhez generáltunk egy szinuszjelet a számítógépen, amelynek 3,5-es Jack kimenetét összekötöttük a kártya LINE IN bemenetével. Egy szimpla bal és jobb csatorna adatátvitelt mutat a 4.15. ábra.



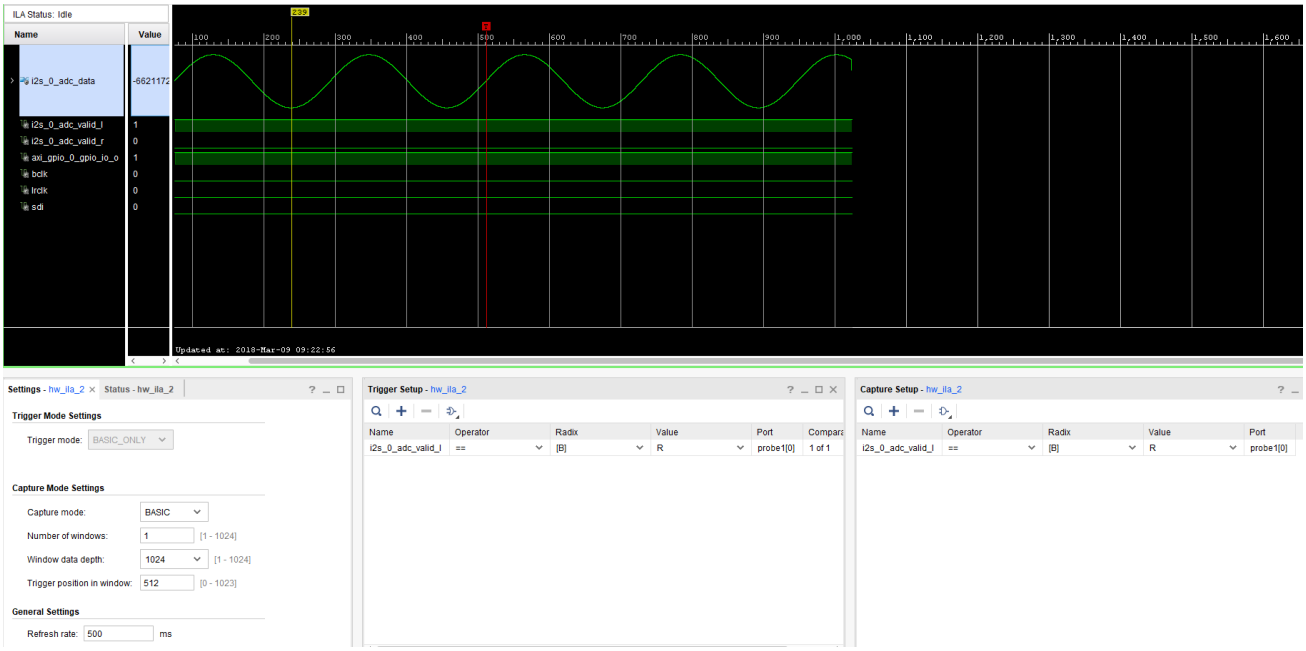
4.15. Ábra – *Az I2S interfész hullámformája, mintavételezett adatokkal*

a 4.16. ábrán a logikai analizátorban a jobb csatorna adataihoz tartozó eseményeket mintavételeztük. Csak akkor vettünk mintát, ha az **adc_valid_r**-en jött egy felfutó él.



4.16. Ábra – *A jobb csatorna adatainak hullámformája*

Ugyanezt szemléltettük a 4.17. ábrán is, csak itt a bal csatorna adatai szerepelnek.



4.17. Ábra – *A bal csatorna adatainak hullámformája*

5. fejezet

I2S → AXI Stream interfész megvalósítása Vivado HLS-ben

Az ötödik alkalommal egy I2S→ AXI Stream adapter modult kellett megterveznünk, amelyhez a Vivado HLS fejlesztői környezetet használtuk. Annyi volt modul a feladata, hogy átvegye a bemeneteire kötött FIFO-kból a jobb és bal csatorna adatait, majd azokat továbbítsa sorosan egy AXI Stream interfészen keresztül a 2. fejezetben már beépített AXI Stream FIFO felé. Ott ennek a FIFO-nak az írási oldala vissza volt kötve a DMA megfelelő bemenetére, de itt viszont már a HLS blokknak kellett meghajtani az FIFO írás oldali interfészét.

5.1. Vivado HLS

A magas szintű szintézishez a fent említett adapter modult az 5.1. kóddal modelleztük.

5.1. Kód – *Az illesztő modult megvalósító forráskód (fir_hw.cpp)*

```
#include "iostream"
#include "fir_hw_common.h"

void fir_hw(
    ap_uint<16>         tlast_dnum  ,
    din_t*              input_l     ,
    din_t*              input_r     ,
    out_stream_struct*  res
)
{
#pragma HLS INTERFACE axis register both port=res       // Set 'res' interface type to AXI Stream
#pragma HLS INTERFACE ap_hs port=input_r // Set port level handshakes
#pragma HLS INTERFACE ap_hs port=input_l // Set port level handshakes
#pragma HLS INTERFACE s_axilite port=tlast_dnum // Set 'tlast_dnum' interface type to AXI Lite
#pragma HLS INTERFACE ap_ctrl_none port=return // Remove block level handshake signals

    //-------- Internals ----------------
    static ap_uint<16>  packet_cntr     = 0;
    static ap_uint<1>   l_r_indicator   = 0;
    din_t               l_din   ;
    din_t               r_din   ;
    out_stream_struct   dout    ;
    //-----------------------------------

    #ifdef DEBUG_ON
    std::cout << "packet_cntr_=_"<< (int)(packet_cntr) << std::endl;
#endif

    // Storing inputs to registers
    l_din = (*input_l);
    r_din = (*input_r);

    // Doing the "multiplexing" between the channels
    // Even: l_din
    // Odd:  r_din
    dout.tdata = (l_r_indicator == 0) ? (l_din): (r_din);

    // Generating 'tlast' when it is needed
    if(packet_cntr == tlast_dnum){
        dout.tlast  = true;
        packet_cntr = 0;
    } else {
        dout.tlast  = false;
        packet_cntr++;          // Incrementing packet counter
    }

    #ifdef DEBUG_ON
    std::cout << "dout.tdata_=_"  << (float)(dout.tdata) << std::endl;
        #endif

    // Toggling channel indicator
    l_r_indicator++;

    // Driving outputs
    (*res) = dout;
}
```

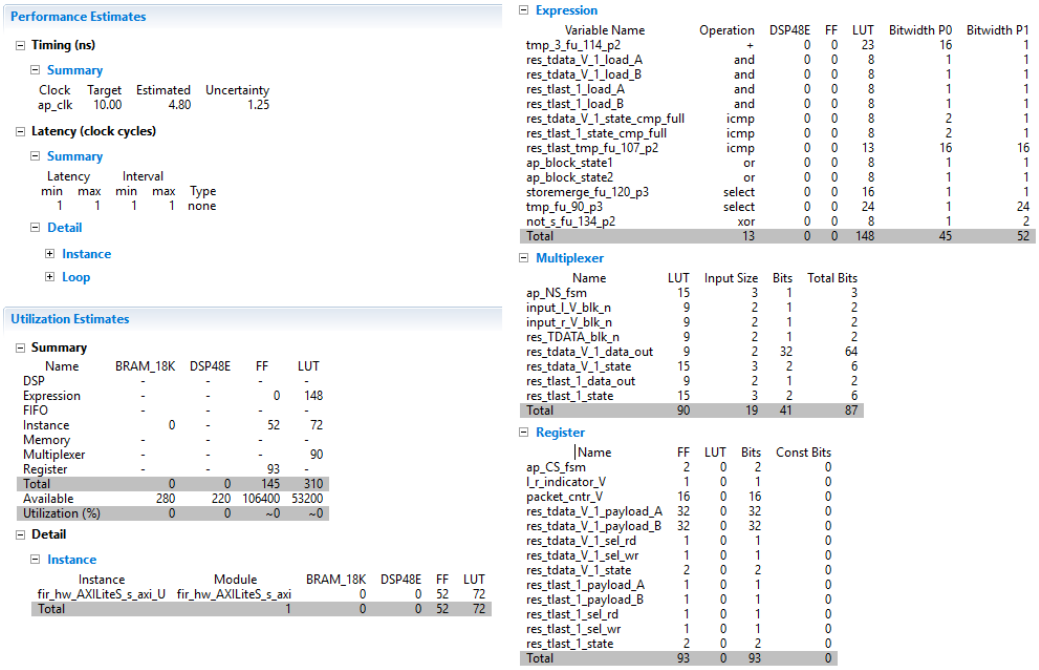
Az 5.2. kódrészlet a használt portok típusdeklarációját mutatja be.

5.2. Kód – *A forráskódhoz tartozó deklarációk (fir_hw_common.h)*

```
#include "ap_int.h"
#include "ap_fixed.h"


typedef ap_fixed<24, 1, AP_TRN_ZERO, AP_WRAP >  din_t;
typedef ap_fixed<32, 1, AP_TRN_ZERO, AP_WRAP >  dout_t;


struct out_stream_struct{
    dout_t  tdata;
    bool    tlast;
};
```

A szintetizált modell erőforrásigénye az 5.1. ábrán látható. Észrevehető a **Register** részen, hogy a **res** kimeneten, amely AXI Stream típusúnak lett beállítva, mind a két csatornára létrehozott egy-egy 32 bites regiszter, valamint még a **TLAST** jelzéssel összefüggő **packet_cntr** 16 bites változóból is regiszter következtetett ki a szintézer.

Továbbá, egy FSM-et is létrehozott, a hozzá tartozó regiszterezett vezérlőjelekkel, amely az előbb említett interfészt vezérelte.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	4.80	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	1	1	1	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	148
FIFO	-	-	-	-
Instance	0	-	52	72
Memory	-	-	-	-
Multiplexer	-	-	-	90
Register	-	-	93	-
Total	0	0	145	310
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
fir_hw_AXILiteS_s_axi_U	fir_hw_AXILiteS_s_axi	0	0	52	72
Total		1	0	0	52	72

Expression

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
tmp_3_fu_114_p2	+	0	0	23	16	1
res_tdata_V_1_load_A	and	0	0	8	1	1
res_tdata_V_1_load_B	and	0	0	8	1	1
res_tlast_1_load_A	and	0	0	8	1	1
res_tlast_1_load_B	and	0	0	8	1	1
res_tdata_V_1_state_cmp_full	icmp	0	0	8	2	1
res_tlast_1_state_cmp_full	icmp	0	0	8	2	1
res_tlast_tmp_fu_107_p2	icmp	0	0	13	16	16
ap_block_state1	or	0	0	8	1	1
ap_block_state2	or	0	0	8	1	1
storemerge_fu_120_p3	select	0	0	16	1	1
tmp_fu_90_p3	select	0	0	24	1	24
not_s_fu_134_p2	xor	0	0	8	1	2
Total		13	0	0	148	45	52

Multiplexer

Name	LUT	Input Size	Bits	Total Bits
ap_NS_fsm	15	3	1	3
input_l_V_blk_n	9	2	1	2
input_r_V_blk_n	9	2	1	2
res_TDATA_blk_n	9	2	1	2
res_tdata_V_1_data_out	9	2	32	64
res_tdata_V_1_state	15	3	2	6
res_tlast_1_data_out	9	2	1	2
res_tlast_1_state	15	3	2	6
Total	90	19	41	87

Register

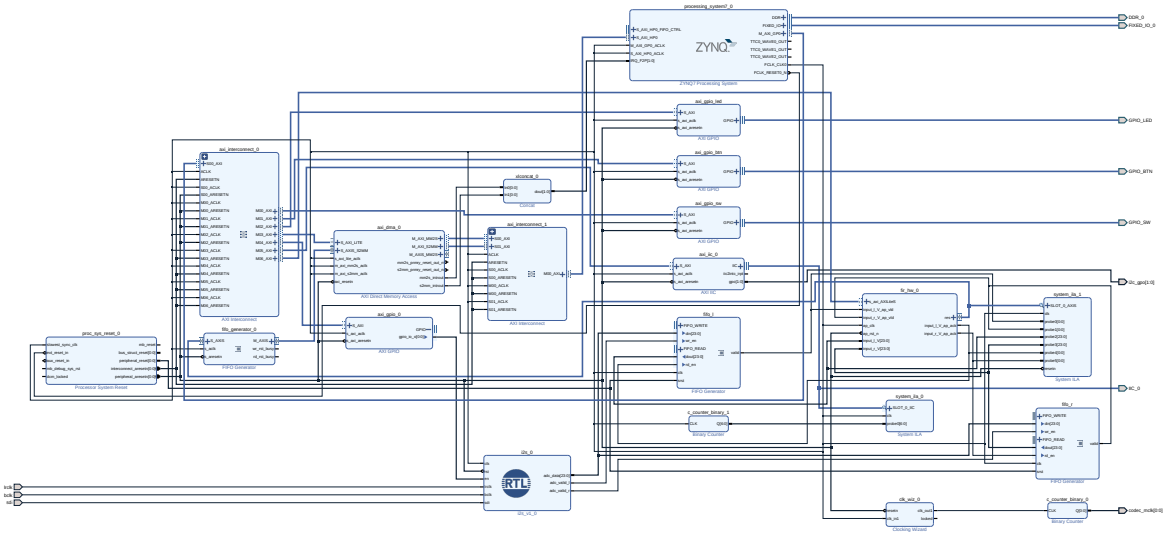
Name	FF	LUT	Bits	Const Bits
ap_CS_fsm	2	0	2	0
l_r_indicator_V	1	0	1	0
packet_cntr_V	16	0	16	0
res_tdata_V_1_payload_A	32	0	32	0
res_tdata_V_1_payload_B	32	0	32	0
res_tdata_V_1_sel_rd	1	0	1	0
res_tdata_V_1_sel_wr	1	0	1	0
res_tdata_V_1_state	2	0	2	0
res_tlast_1_payload_A	1	0	1	0
res_tlast_1_payload_B	1	0	1	0
res_tlast_1_sel_rd	1	0	1	0
res_tlast_1_sel_wr	1	0	1	0
res_tlast_1_state	2	0	2	0
Total	93	0	93	0

5.1. Ábra – *A szintetizált modell erőforrásigénye*

A maradék erőforrást a Slave AXI Lite interfész megvalósítása jelentette a **tlast_dnum** változóra, ahogy az az **Instance** fülön látható.

5.2. IP beillesztése a rendszerbe

A szintetizált HLS blokkot ezután hozzárendeltük a Vivado projektünk IP Repository-jához, majd beillesztettük az eddigi rendszerünkbe, emellett még két FIFO-t használtunk az I2S modulunkból jövő adatok bufferelséséhez. Az egyik logikai analizátorunkat lecseréltük egy AXI Stream típusúra, amelynek segítségével a HLS blokk kimenetén lejátszódó AXI Stream protokollt tudtuk ellenőrizni. Az így kapott dizájnt az 5.2. és az 5.3. ábrák mutatják be.



5.2. Ábra – *A Block Design-ban összeállított rendszer blokkdiagramja*

Diagram | Address Editor

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
axi_gpio_0	S_AXI	Reg	0x4123_0000	64K	0x4123_FFFF
axi_gpio_btn	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_led	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_gpio_sw	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
axi_iic_0	S_AXI	Reg	0x4160_0000	64K	0x4160_FFFF
fir_hw_0	s_axi_AXILiteS	Reg	0x43C0_0000	64K	0x43C0_FFFF
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

5.3. Ábra – *Az összeállított rendszer memóriatérképe*

5.3. Tesztelés

A HLS blokk teszteléséhez egy egyszerű Cpp testbench-et terveztünk, amellyel az alapvető működést ellenőrizni tudtuk, ezt az 5.3. kód szemlélteti.

5.3. Kód – *A testbench forráskódja (fir_hw_tb.cpp)*

```
#include "iostream"


#include <cstdlib>
#include <ctime>

#include "fir_hw_common.h"

#define DEBUG_ON

extern void fir_hw(
    ap_uint<16>         tlast_dnum  ,
    din_t*              input_l     ,
    din_t*              input_r     ,
    out_stream_struct*  res
);

int main(){


    ap_uint<16>         tb_tlast_dnum;
    din_t               tb_ldata = 0;
    din_t               tb_rdata = 0;
    out_stream_struct   tb_res  ;


    // 'tlast' indicator
    tb_tlast_dnum = 32;

    // Use current time as seed for random generator
    std::srand(std::time(NULL));

    for(int i = 0; i < tb_tlast_dnum; i++){

        std::cout << "Test_[" << i <<"]" << std::endl;
        std::cout << "============="<< std::endl;



        // Generate random inputs
        if(i%2 == 0){
            tb_ldata    = (float)(std::rand()) / (float)(RAND_MAX+1);
        } else {
            tb_rdata    = (float)(std::rand()) / (float)(RAND_MAX+1);
        }

        std::cout << "tb_ldata_=_"  << (float)(tb_ldata) << std::endl;
        std::cout << "tb_rdata_=_"  << (float)(tb_rdata) << std::endl;


        // Drive the random inputs to the HLS block
        fir_hw(
            tb_tlast_dnum   ,
            &tb_ldata       ,
            &tb_rdata       ,
            &tb_res
        );

        std::cout << std::endl;
        std::cout << "End_test" << std::endl;
        std::cout << std::endl;
    }

    return 0;
}
```

Az 5.4. ábrán a testbench szimulációjának egy részeredménye látható. Észrevehető, hogy egyszer a jobb aztán a bal csatorna adatát rakja a kimenetre.

5.4. Kód – *A Cpp testbench szimulációjának egy részlete*

```
Test [0]
=============
tb_ldata = 0.426544
tb_rdata = 0
packet_cntr = 0
dout.tdata = 0.426544

End test

Test [1]
=============
tb_ldata = 0.426544
tb_rdata = 0.632446
packet_cntr = 1
dout.tdata = 0.632446

End test

Test [2]
=============
tb_ldata = 0.973053
tb_rdata = 0.632446
packet_cntr = 2
dout.tdata = 0.973053

End test
```

A HLS blokk viselkedését valós áramköri környezetben is megvizsgáltuk, a 2.3. szakaszban használt DMA vezérlő kód felhasználásával és módosításával.

5.5. Kód – *A testbench forráskódja (fir_hw_tb.cpp)*

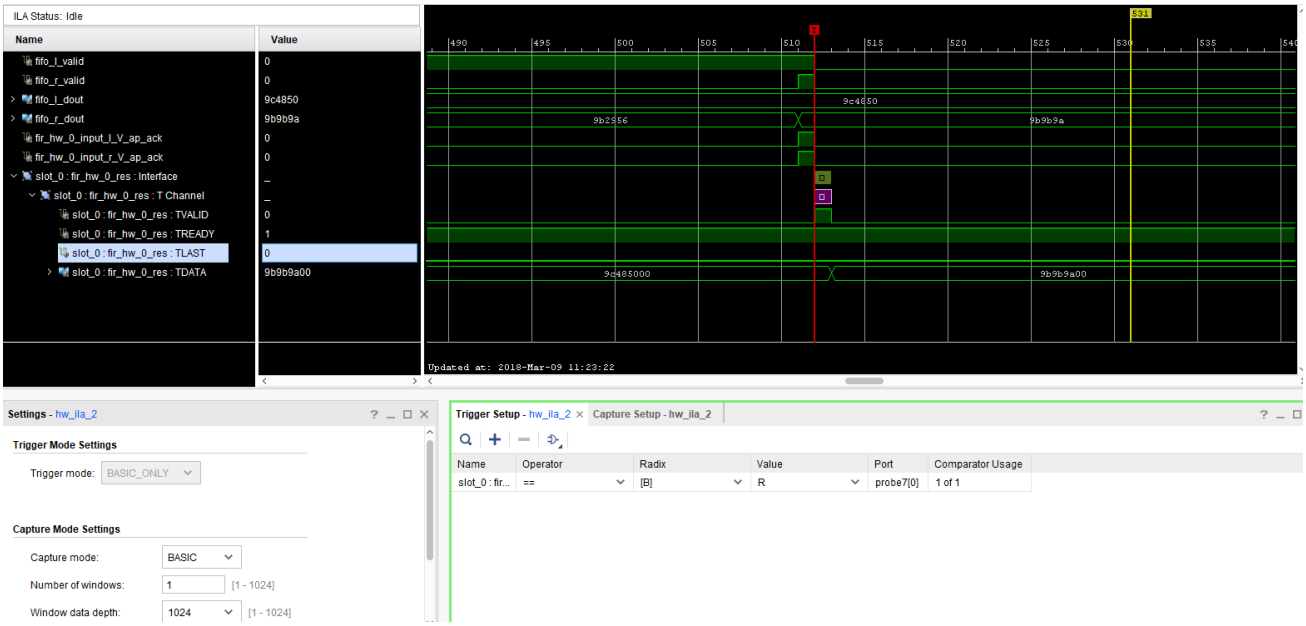
```
#include "xaxidma.h"
#include "xparameters.h"
#include "xdebug.h"
#include "xgpio_l.h"
#include "xfir_hw.h"
#include "xfir_hw_hw.h"
#include "stdint.h"

#define SR 0x104
#define TX_FIFO 0x108
#define RX_FIFO 0x10C
#define GPO 0x124
#define CR 0x100
#define RX_FIFO_PIRQ 0x120

#define TX_FIFO_EMPTY 7
#define RX_FIFO_EMPTY 6
#define BB 2
#define RSTA 5
#define START 8

void i2c_init(){
    uint8_t temp;
    // GPO = 0
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + GPO, 0);
    // RX_FIFO_PIRQ = 0x0F
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + RX_FIFO_PIRQ, 0x0F);
    // Reset TF_FIFO
    temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + CR);
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + CR, (temp | 0x02));
    // Enable
    temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + CR);
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + CR, (temp | 0x01));
    // Remove reset
    temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + CR);
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + CR, (temp & ~0x02));
}

void i2c_write(int address, uint8_t data) {
    // Wait for empty/idle
    int temp;
    temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);
    while( (temp & ((1 << TX_FIFO_EMPTY)|(1 << RX_FIFO_EMPTY)|(1 << BB))) != ((1 << TX_FIFO_EMPTY) | (1 << RX_FIFO_EMPTY)) ){
        temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);
    }

    int wr_data = 0x100 + 0x70;
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + TX_FIFO, wr_data);
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + TX_FIFO, ((address >> 8) & 0x00FF));
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + TX_FIFO, address & 0x00FF);
    wr_data = 0x200 + data;
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + TX_FIFO, wr_data);

    temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);
    while((temp & (1 << TX_FIFO_EMPTY)) != (1 << TX_FIFO_EMPTY))
        temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);

    return;
}

int i2c_read(int address) {
    // Wait for empty/idle
    int temp;
    temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);
    while((temp & ((1 << TX_FIFO_EMPTY)|(1 << RX_FIFO_EMPTY)|(1 << BB))) != ((1 << TX_FIFO_EMPTY) | (1 << RX_FIFO_EMPTY))){
        temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);
    }

    int rd_data = 0x100 + 0x70;
    int wr_data = 0x100 + 0x71;
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + TX_FIFO, rd_data);
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + TX_FIFO, (address & 0xFF00) >> 8);
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + TX_FIFO, address & 0x00FF);
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + TX_FIFO, wr_data);
    Xil_Out32(XPAR_AXI_IIC_0_BASEADDR + TX_FIFO, 0x201);

    temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);
    while((temp & (1 << TX_FIFO_EMPTY)) != (1 << TX_FIFO_EMPTY))
        temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);

    temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);
    while((temp & (1 << RX_FIFO_EMPTY)) == (1 << RX_FIFO_EMPTY))
            temp = Xil_In32(XPAR_AXI_IIC_0_BASEADDR + SR);

    return Xil_In32(RX_FIFO & 0xFF);
}

#define DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID

#ifdef XPAR_AXI_7SDDR_0_S_AXI_BASEADDR
```

```
#define DDR_BASE_ADDR       XPAR_AXI_7SDDR_0_S_AXI_BASEADDR
#elif XPAR_MIG7SERIES_0_BASEADDR
#define DDR_BASE_ADDR   XPAR_MIG7SERIES_0_BASEADDR
#elif XPAR_MIG_0_BASEADDR
#define DDR_BASE_ADDR   XPAR_MIG_0_BASEADDR
#elif XPAR_PSU_DDR_0_S_AXI_BASEADDR
#define DDR_BASE_ADDR   XPAR_PSU_DDR_0_S_AXI_BASEADDR
#endif

#ifndef DDR_BASE_ADDR
#warning CHECK FOR THE VALID DDR ADDRESS IN XPARAMETERS.H, \
        DEFAULT SET TO 0x01000000
#define MEM_BASE_ADDR       0x01000000
#else
#define MEM_BASE_ADDR       (DDR_BASE_ADDR + 0x1000000)
#endif

#define TX_BUFFER_BASE      (MEM_BASE_ADDR + 0x00100000)
#define RX_BUFFER_BASE      (MEM_BASE_ADDR + 0x00300000)
#define RX_BUFFER_HIGH      (MEM_BASE_ADDR + 0x004FFFFF)

#define MAX_PKT_LEN     0x100

#define TEST_START_VALUE    0xC

#define NUMBER_OF_TRANSFERS 10

#if (!defined(DEBUG))
extern void xil_printf(const char *format, ...);
#endif

int XAxiDma_SimplePollExample(u16 DeviceId);
static int CheckData(void);

XAxiDma AxiDma;

int main()
{
    int temp;
    // CODEC init
    i2c_init();
    i2c_write(0x4000, 0b0111   );   // Clock control register 4'b0111
    i2c_write(0x40F9, 0x7F  );
    i2c_write(0x40FA, 0x3   );
    i2c_write(0x4015, 0b00001001);  //LRPOL = 1, Master
    i2c_write(0x4016, 0b00000010);  //LRDEL = 8 (right justified)
    i2c_write(0x4017, 0b00000110);  //fs/0.5 -> 96kHz
    i2c_write(0x40EB, 0b000);
    i2c_write(0x40F8, 0b110);
    i2c_write(0x400A, 0x1   );
    i2c_write(0x400B, 0x5   );
    i2c_write(0x400C, 0x1   );
    i2c_write(0x400D, 0x5   );
    i2c_write(0x401C, 0x21  );
    i2c_write(0x401E, 0x41  );
    i2c_write(0x4023, 0xE7  );
    i2c_write(0x4024, 0xE7  );
    i2c_write(0x4025, 0xE7  );
    i2c_write(0x4026, 0xE7  );
    i2c_write(0x4019, 0x3   );
    i2c_write(0x4029, 0x3   );
    i2c_write(0x402A, 0x3   );
    i2c_write(0x40F2, 0x1   );
    i2c_write(0x40F3, 0x1   );
    i2c_write(0x40FA, 0x3   );

    // Enable I2S
    Xil_Out32(XPAR_AXI_GPIO_0_BASEADDR + 0x04, 0);
    Xil_Out32(XPAR_AXI_GPIO_0_BASEADDR, 1);
    // Set tlast_dnum in fir_hw HLS
    Xil_Out32(XPAR_FIR_HW_0_S_AXI_AXILITES_BASEADDR + XFIR_HW_AXILITES_ADDR_TLAST_DNUM_V_DATA, 31);

    int Status;

    xil_printf("\r\n---␣Entering␣main()␣---␣\r\n");

    /* Run the poll example for simple transfer */
    Status = XAxiDma_SimplePollExample(DMA_DEV_ID);

    if (Status != XST_SUCCESS) {
        xil_printf("XAxiDma␣Simple␣Poll␣Example␣Failed\r\n");
        return XST_FAILURE;
    }

    xil_printf("Successfully␣ran␣XAxiDma␣Simple␣Poll␣Example\r\n");

    xil_printf("---␣Exiting␣main()␣---␣\r\n");

    return XST_SUCCESS;

}

int XAxiDma_SimplePollExample(u16 DeviceId)
{
    XAxiDma_Config *CfgPtr;
    int Status;
    int Tries = NUMBER_OF_TRANSFERS;
```

```
189                 int Index;
190                 u8 *TxBufferPtr;
191                 u8 *RxBufferPtr;
192                 u8 Value;
193
194                 TxBufferPtr = (u8 *)TX_BUFFER_BASE ;
195                 RxBufferPtr = (u8 *)RX_BUFFER_BASE;
196
197                 /* Initialize the XAxiDma device.
198                  */
199                 CfgPtr = XAxiDma_LookupConfig(DeviceId);
200                 if (!CfgPtr) {
201                     xil_printf("No config found for %d\r\n", DeviceId);
202                     return XST_FAILURE;
203                 }
204
205                 Status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
206                 if (Status != XST_SUCCESS) {
207                     xil_printf("Initialization failed %d\r\n", Status);
208                     return XST_FAILURE;
209                 }
210
211                 if(XAxiDma_HasSg(&AxiDma)){
212                     xil_printf("Device configured as SG mode \r\n");
213                     return XST_FAILURE;
214                 }
215
216                 /* Disable interrupts, we use polling mode
217                  */
218                 XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
219                         XAXIDMA_DEVICE_TO_DMA);
220                 XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
221                         XAXIDMA_DMA_TO_DEVICE);
222
223                 Value = TEST_START_VALUE;
224
225                 for(Index = 0; Index < MAX_PKT_LEN; Index ++) {
226                     TxBufferPtr[Index] = Value;
227
228                     Value = (Value + 1) & 0xFF;
229                 }
230                 /* Flush the SrcBuffer before the DMA transfer, in case the Data Cache
231                  * is enabled
232                  */
233                 Xil_DCacheFlushRange((UINTPTR)TxBufferPtr, MAX_PKT_LEN);
234             #ifdef __aarch64__
235                 Xil_DCacheFlushRange((UINTPTR)RxBufferPtr, MAX_PKT_LEN);
236             #endif
237
238                 while(1) {
239                     Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) RxBufferPtr,
240                             MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
241
242                     if (Status != XST_SUCCESS) {
243                         return XST_FAILURE;
244                     }
245
246                     Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) TxBufferPtr,
247                             MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);
248
249                     if (Status != XST_SUCCESS) {
250                         return XST_FAILURE;
251                     }
252
253                     while ((XAxiDma_Busy(&AxiDma,XAXIDMA_DEVICE_TO_DMA)) ||
254                             (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE))) {
255
256                     }
257                 }
258
259                 return XST_SUCCESS;
260             }
```

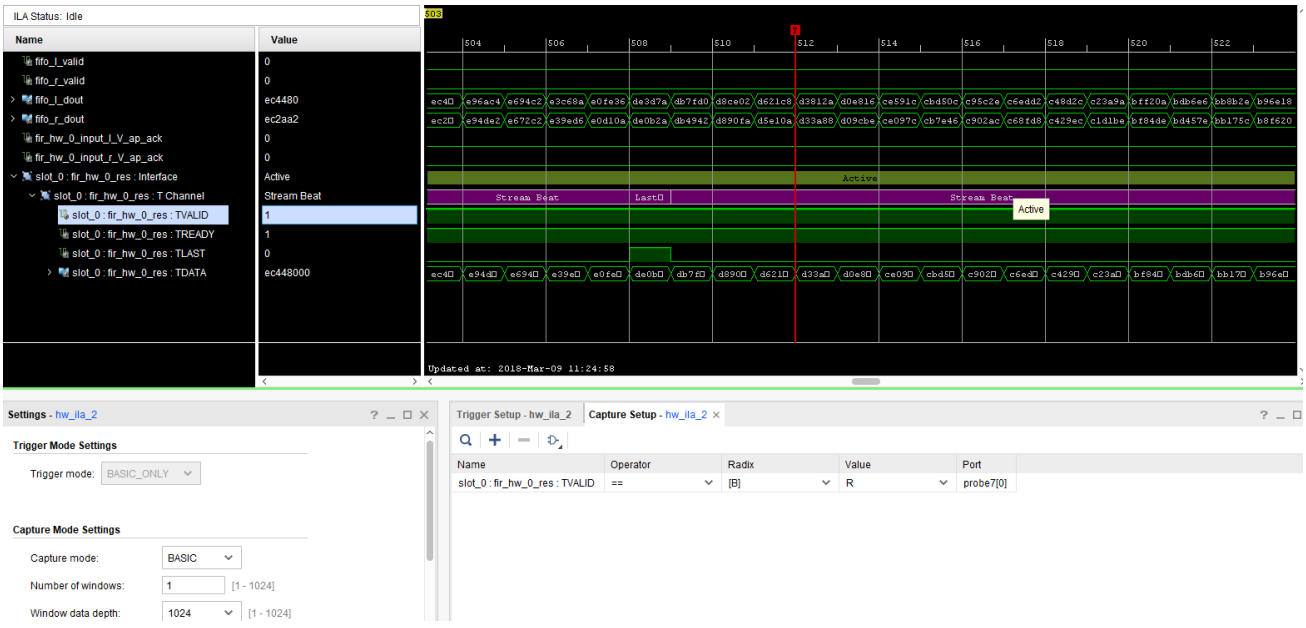
Tehát a 2.3. szakaszban használt kódhoz hozzáadtuk a 4. fejezetben használt kódot, ez látszik a 130-as sortól kezdődően. Majd a 162-es sorban beírtuk a HLS blokkunk AXI Lite interfészén a **tlast_dnum** értékét. Az ezt követő részek a DMA vezérlésével voltak kapcsolatosak. A 238-as sorban lecseréltük a for ciklus-t, amely előre megadott számú DMA írási és olvasási ciklust hajtott végre, egy végtelen ciklusra.

Ahhoz, hogy nagyjából tudjuk, hogy a DMA jó adatokat ír be a rendszermemóriába, ahhoz először még meg kellett győződnünk arról, hogy a HLS blokk alapvetően jól működik-e. Azaz ebben az esetben generálja-e **TVALID** jelet az AXI Stream FIFO számára, hiszen ilyenkor ő a Master. Ezt mutatja az 5.4. ábra, ahol is egyszerűen a **TVALID** felfutóélére triggereltük a logikai analizátort.



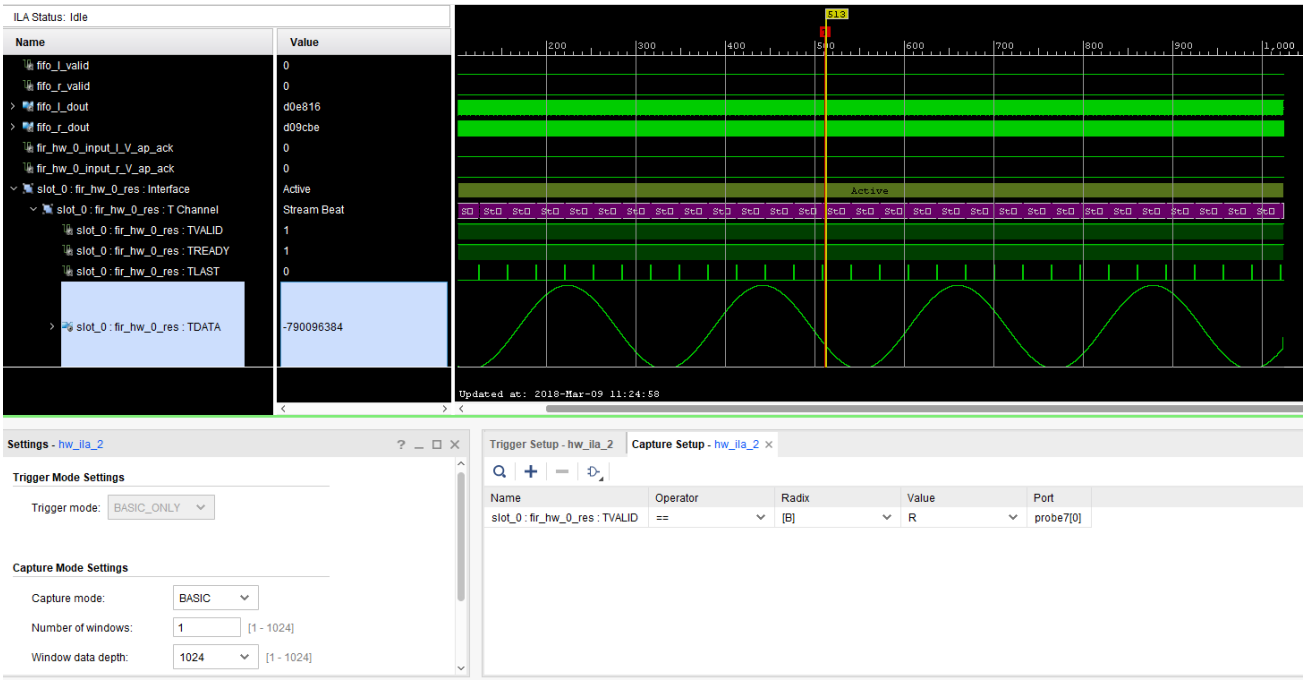
5.4. Ábra – *A HLS blokk TVALID jelének ellenőrzése*

Ezt követően megnéztük több csomag átvitelét is úgy, hogy a Capture Setup-ban beállítottuk, hogy a **TVALID** felfutó élére vegyen csak mintát a logikai analizátor, ezt mutatja az 5.5. ábra.



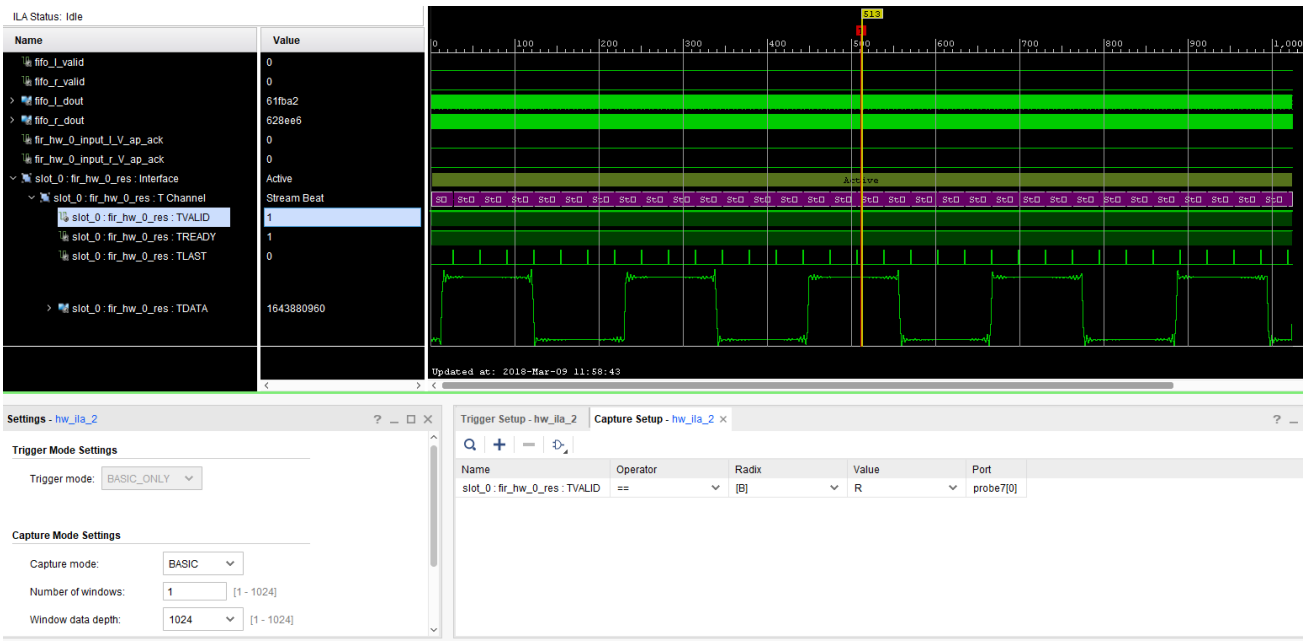
5.5. Ábra – *Több AXI Stream csomag ellenőrzése*

Továbbá, úgy is ellenőriztük a CODEC által küldött értékeket, hogy a ChipScope-ban a **TDATA** megjelenítését átraktuk analógra, ezt mutatja az 5.6. ábra.



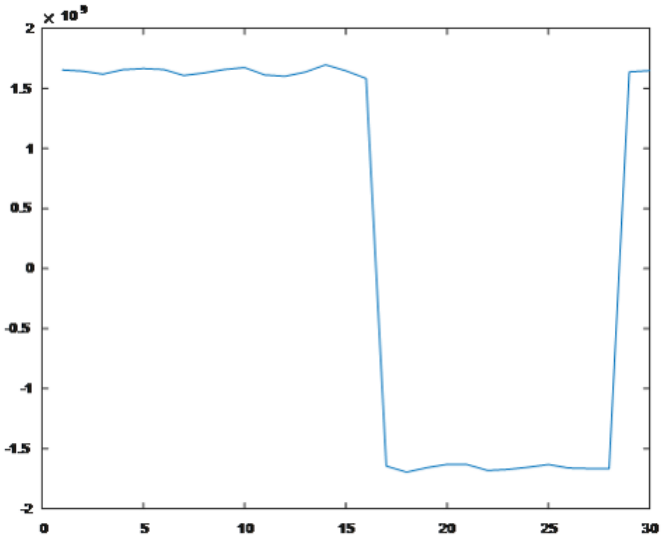
5.6. Ábra – *A mintavételezett szinusz jel analóg megjelenítése*

Majd legvégül arról is szerettünk volna meggyőződni, hogy a DMA által beírt adatok jellegre megegyeztek-e a bemeneti jellel. Ezért egy négyszögjelet adtunk a CODEC LINE IN bemenetére, hogy a DMA RX bufferének báziscímétől kiolvasott adatott könnyen értelmezni tudjuk. A mintavételezett hullámformát az 5.7. ábra mutatja be.



5.7. Ábra – *A négyszögjel gerjesztés hullámformája*

Miután beírt pár adatot a DMA, azután megállítottuk a futó szoftvert és XSCT-ben egyszerű memória olvasással kiolvastunk néhány adatot a DMA RX bufferének báziscíméről kezdődően, amelyeket MATLAB-ban ábrázoltunk, ez látható az 5.8. ábrán.



5.8. Ábra – *A DMA RX bufferéből kiolvasott adatok*

Persze, nem kellett volna a MATLAB, egyszerűen eldönthető lett volna 32 bites hexadecimális értékek legfelső számjegyei alapján, hogy nagyjából jók-e a bevitt adatok.

6. fejezet

FIR szűrő megvalósítása Vivado HLS-ben

A következő alkalommal az 5. fejezetben megvalósított adaptermodult lecseréltük egy FIR szűrőre, amelynek a mintavételi frekvenciája 96 kHz volt. A modul két porton keresztül fogadta a jobb és bal csatorna adatait, amelyeken a beállításától függően 2-szeres vagy 4-szeres decimálást is végre tudott hajtani. Ezekben az esetekben 256 és 512 számú szűrőegyütthatót használt, míg egyszeres decimálás esetén 128-at, de a szűrő együtthatószáma szabadon programozható volt egészen 512-ig.

A bemenő adatokat 1.23, míg az együtthatókat 1.31 alakú fixpontos számonként várta, valamint a szűrés végeredményét 1.31 alakban szolgáltatta.

6.1. Vivado HLS

A magas szintű szintézishez a FIR szűrőt a 6.1. kóddal modelleztük.

6.1. Kód – *A FIR szűrőt megvalósító forráskód (fir_hw.cpp)*

```
#include "iostream"
#include "fir_hw_common.h"

void fir_hw(
                ap_uint<16>         tlast_dnum      ,
                ap_uint<3>          smpl_rd_num     ,
                ap_uint<9>          tap_num_m1      ,
                coeff_t             coeff_hw[N]     ,
                din_t*              input_l         ,
                din_t*              input_r         ,
                out_stream_struct*  res
){
#pragma HLS INTERFACE axis register both port=res

        // Handshake ports
#pragma HLS INTERFACE ap_hs port=input_r
#pragma HLS INTERFACE ap_hs port=input_l

        // AXI Lite ports
#pragma HLS INTERFACE s_axilite port=tlast_dnum
#pragma HLS INTERFACE s_axilite port=smpl_rd_num
#pragma HLS INTERFACE s_axilite port=tap_num_m1
#pragma HLS INTERFACE s_axilite port=coeff_hw

        // Block level handshake off
#pragma HLS INTERFACE ap_ctrl_none port=return


        //-------- Internals ----------------
        static ap_uint<16>  packet_cntr     = 0 ;
        static ap_uint<1>   l_r_indicator   = 0 ;
        static ap_uint<9>   wr_addr_l       = 0 ;
        static ap_uint<9>   wr_addr_r       = 0 ;
        static ap_uint<3>   iteration       = 0 ;
        static din_t        smpl[N][2]          ;

        ap_uint<9>              addr = 0        ;
        ap_uint<9>              mask            ;
        din_t               l_din       ;
        din_t               r_din       ;
        out_stream_struct   dout        ;
        int                 i           ;
        accu_t              accu        ;
        //------------------------------------

        // Create overflow mask
        switch(smpl_rd_num){
        case 1:     mask = 0x07F; break;    // 7 bit (128 sample)
        case 2:     mask = 0x0FF; break;    // 8 bit (256 sample)
        default:    mask = 0x1FF;           // 9 bit (512 sample)
        }

        // DEBUG
#ifdef DEBUG_ON
        std::cout << "packet_cntr_=_"<< (int)(packet_cntr) << std::endl;
#endif

        // Doing the "multiplexing" between the channels
        if(l_r_indicator){
#ifdef DEBUG_ON
```

```
61                  std::cout << "Channel = LEFT "<< std::endl;
62  #endif
63
64                  // Storing inputs to sample arrays
65                  addr = wr_addr_l;
66                  smpl[addr][l_r_indicator] = (*input_l);
67                  // Increase and mask wr_adress
68                  wr_addr_l = (wr_addr_l + 1) & mask;
69          }
70          else{
71  #ifdef DEBUG_ON
72                  std::cout << "Channel = RIGHT " << std::endl;
73  #endif
74
75                  // Storing inputs to sample arrays
76                  addr = wr_addr_r;
77                  smpl[addr][l_r_indicator] = (*input_r);
78                  // Increase and mask wr_adress
79                  wr_addr_r = (wr_addr_r + 1) & mask;
80          }
81          // Decimation (run only in every smpl_rd_num. iteration)
82          if (iteration <= 1){
83                  // Init accu
84                  accu = 0;
85                  // Loop for coefficients
86                  for_mac: for (i = 0; i <= tap_num_m1; i++)
87                  {
88          #pragma HLS LOOP_TRIPCOUNT min=127 max=511
89          #pragma HLS PIPELINE II=1
90                          // MAC for correct channel
91                          accu = accu + (coeff_hw[i] * smpl[addr][l_r_indicator]);
92
93                          // Decrease address
94                          if(addr == 0)
95                                  addr = tap_num_m1;
96                          else
97                                  addr--;
98                  }
99
100                 // Output
101                 dout.tdata = accu;
102                 // Generating 'tlast' when it is needed
103                 if(packet_cntr == tlast_dnum){
104                         dout.tlast  = true;
105                         packet_cntr = 0;
106                 } else {
107                         dout.tlast  = false;
108                         packet_cntr++;          // Incrementing packet counter
109                 }
110
111                 // Driving outputs
112                 (*res) = dout;
113
114                 // DEBUG
115                 std::cout << "dout.tdata = "  << (float)(dout.tdata) << std::endl;
116                 }
117
118         // Toggling channel indicator
119         l_r_indicator++;
120
121         // Increase iteration
122         if(iteration >= ((smpl_rd_num*2)-1))
123                 iteration = 0;
124         else
125                 iteration++;
126 }
```

A 60. sorban lévő l_r_indicator változót használtuk arra, hogy multiplexáljunk a bemenő jobb és bal csatorna adatai között, hiszen az I2S interfészelésért felelős modulunk felváltva adogatta ki magából az Audio CODEC által mintavételezett adatokat.

A 82. sorban az iteration változó értékének vizsgálatával valósítjuk meg a decimálást, olyan módon, hogy az ezt követő tényleges jelfeldolgozást végző rész az smpl_rd_num változó függvényében csak 1/2/4 bementi változónként kerül végrehajtásra. A FIR szűrőt a 86. sorban kezdődő for_mac ciklus írja le, amire PIPELINE pragmát állítottunk be 1-es inicializációs intervallummal, valamint az időzítések analíziséhez használtuk a LOOP_TRIPCOUNT pragmát is. A kód további részében a szűrés eredményének kimenetre vezetése, a TLAST jel generálása és a vezérlést végző két változó léptetése látható.

A szűrőhöz használt típusokat és makrókat a 6.2. kód szemlélteti.

6.2. Kód – *A FIR szűrőhöz használt típusok és makrók forráskódja (fir_hw _common.h)*

```
1   #include "ap_int.h"
2   #include "ap_fixed.h"
3
4   #ifndef FIR_HW_COMMON_H
5   #define FIR_HW_COMMON_H
6
7   #define N 512
8
9   #define DEBUG_ON
10
11  typedef ap_fixed<24, 1, AP_TRN_ZERO, AP_WRAP >  din_t;
12  typedef ap_fixed<32, 1, AP_TRN_ZERO, AP_SAT >  dout_t;
13  typedef ap_fixed<32, 1, AP_RND_ZERO, AP_WRAP >  coeff_t;
14  typedef ap_fixed<41, 10, AP_TRN_ZERO, AP_WRAP >  accu_t;
15
16  struct out_stream_struct{
17      dout_t  tdata;
18      bool    tlast;
19  };
20  #endif
```

6.2. Tesztelés

A HLS szintéziséhez használt kódot a 6.3. Cpp testbench-el ellenőriztük.

6.3. Kód – *A szimulációhoz használt Cpp testbench forráskódja (fir_hw_tb.cpp)*

```
#include "iostream"
#include <cstdlib>
#include <ctime>

#include "fir_hw_common.h"
#include "fir_coeffs.h"

//#define TAP_NUM_128
#define TAP_NUM_256
//#define TAP_NUM_512

void fir_hw(
    ap_uint<16>         tlast_dnum      ,
    ap_uint<3>          smpl_rd_num     ,
    ap_uint<9>          tap_num_m1      ,
    coeff_t             coeff_hw[N]     ,
    din_t*              input_l         ,
    din_t*              input_r         ,
    out_stream_struct*  res
);

int main(){
    ap_uint<16>         tb_tlast_dnum;
    ap_uint<3>          tb_smpl_rd_num;
    ap_uint<9>          tb_tap_num_m1;
    coeff_t             tb_coeff_hw[N];
    din_t               tb_ldata = 0;
    din_t               tb_rdata = 0;
    out_stream_struct   tb_res  ;

    // Setup
    tb_tlast_dnum = 31;

    #ifdef   TAP_NUM_128
        tb_tap_num_m1   = 127;
        tb_smpl_rd_num  = 1;
    #endif
    #ifdef    TAP_NUM_256
        tb_tap_num_m1   = 255;
        tb_smpl_rd_num  = 2;
    #endif
    #ifdef    TAP_NUM_512
        tb_tap_num_m1   = 511;
        tb_smpl_rd_num  = 4;
    #endif

    std::cout <<std::endl;

    std::cout << "Setup"<< std::endl;
    std::cout << "=============="<< std::endl;
    std::cout << "tap_num_m1 = " << (int)(tb_tap_num_m1) << std::endl;
    std::cout << "smpl_rd_num = " << (int)(tb_smpl_rd_num) << std::endl;
    std::cout << "=============="<< std::endl;
    std::cout <<std::endl;




    //for(int i = 0; i < tb_tlast_dnum; i++){
    for(int i = 0; i < (int)(2*tb_tap_num_m1); i++){

        std::cout << "Test [" << i <<"]" << std::endl;
        std::cout << "=============="<< std::endl;

        // Use current time as seed for random generator
        std::srand(std::time(NULL));

        // Dirac impulse
        if(i == 0)
            tb_rdata    = 0.99;
        else
                tb_rdata    = 0.0;

        if(i == 1)
            tb_ldata    = 0.99;
        else
            tb_ldata    = 0.0;

        std::cout << "tb_ldata = "  << (tb_ldata) << std::endl;
        std::cout << "tb_rdata = "  << (tb_rdata) << std::endl;

        fir_hw(
            tb_tlast_dnum   ,
            tb_smpl_rd_num  ,
            tb_tap_num_m1   ,
            //tb_coeff_hw     ,
            #ifdef      TAP_NUM_128
                filter_128,
            #endif
            #ifdef    TAP_NUM_256
                filter_256,
            #endif
            #ifdef    TAP_NUM_512
                filter_512,
            #endif
            &tb_ldata       ,
            &tb_rdata       ,
            &tb_res
        );

        std::cout << std::endl;
        std::cout << "==============================" << std::endl;
        std::cout << std::endl;
    }
    return 0;
}
```

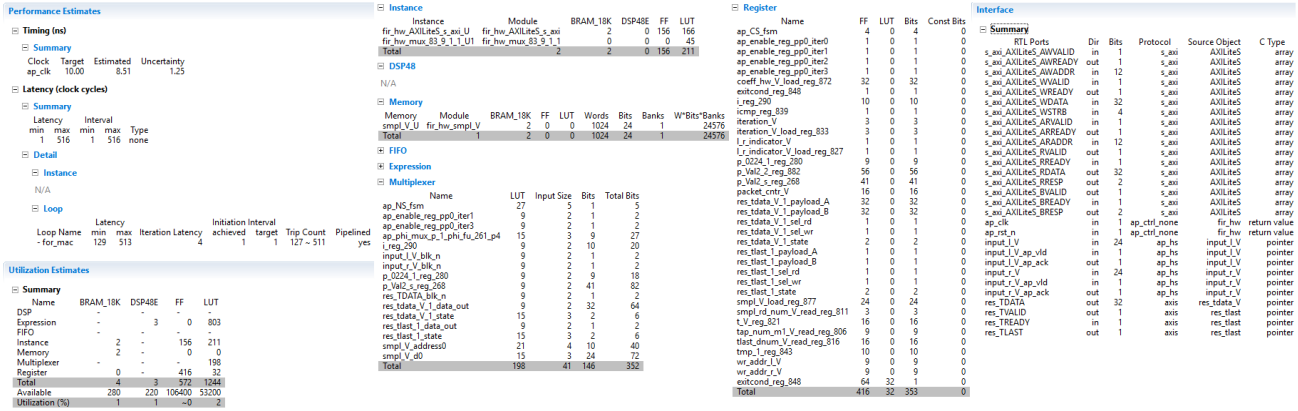
A 6.4. ábrán a testbench szimulációjának egy részeredménye látható, ahol 256-os együtthatószámot és kétszeres decimálást alkalmaztunk. Észrevehető, hogy a modul a kimeneti adat portokon csak minden második meghívás alkalmával szolgáltat új adatot.

6.4. Kód – *A Cpp testbench szimulációjának egy részlete*

```
Setup
=============
tap_num_m1  = 255
smpl_rd_num = 2
=============

Test [0]
=============
tb_ldata = 0
tb_rdata = 0.99
packet_cntr = 0
Channel = RIGHT
dout.tdata = 2.44053e-006

================================

Test [1]
=============
tb_ldata = 0.99
tb_rdata = 0
packet_cntr = 1
Channel = LEFT
dout.tdata = 2.44053e-006

================================

Test [2]
=============
tb_ldata = 0
tb_rdata = 0
packet_cntr = 2
Channel = RIGHT

================================

Test [3]
=============
tb_ldata = 0
tb_rdata = 0
packet_cntr = 2
Channel = LEFT

================================

Test [4]
=============
tb_ldata = 0
tb_rdata = 0
packet_cntr = 2
Channel = RIGHT
dout.tdata = -3.21306e-006

================================

Test [5]
=============
tb_ldata = 0
tb_rdata = 0
packet_cntr = 3
Channel = LEFT
dout.tdata = -3.21306e-006

================================

Test [6]
=============
tb_ldata = 0
tb_rdata = 0
packet_cntr = 4
Channel = RIGHT

================================

Test [7]
=============
tb_ldata = 0
tb_rdata = 0
packet_cntr = 4
Channel = LEFT

================================
...
```

6.3. Szintézis eredmények

Az szűrő szintézise után kapott eredményeket a 6.1. ábra foglalja össze.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.51	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	516	1	516	none

Detail

Instance

N/A

Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- for_mac	129	513	4	1	1	127 ~ 511	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	3	0	803
FIFO	-	-	-	-
Instance	2	-	156	211
Memory	2	-	0	0
Multiplexer	-	-	-	198
Register	0	-	416	32
Total	4	3	572	1244
Available	280	220	106400	53200
Utilization (%)	1	1	~0	2

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
fir_hw_AXILiteS_s_axi_U	fir_hw_AXILiteS_s_axi	2	0	156	166
fir_hw_mux_83_9_1_1_U1	fir_hw_mux_83_9_1_1	0	0	0	45
Total		2	2	0	156	211

DSP48

N/A

Memory

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
smpl_V_U	fir_hw_smpl_V	2	0	0	1024	24	1	24576
Total		2	0	0	1024	24	1	24576

FIFO

Expression

Multiplexer

Name	LUT	Input Size	Bits	Total Bits
ap_NS_fsm	27	5	1	5
ap_enable_reg_pp0_iter1	9	2	1	2
ap_enable_reg_pp0_iter3	9	2	1	2
ap_phi_mux_p_1_phi_fu_261_p4	15	3	9	27
i_reg_290	9	2	10	20
input_I_V_blk_n	9	2	1	2
input_r_V_blk_n	9	2	1	2
p_0224_1_reg_280	9	2	9	18
p_Val2_s_reg_268	9	2	41	82
res_TDATA_blk_n	9	2	1	2
res_tdata_V_1_data_out	9	2	32	64
res_tdata_V_1_state	15	3	2	6
res_tlast_1_data_out	9	2	1	2
res_tlast_1_state	15	3	2	6
smpl_V_address0	21	4	10	40
smpl_V_d0	15	3	24	72
Total	198	41	146	352

Register

Name	FF	LUT	Bits	Const Bits
ap_CS_fsm	4	0	4	0
ap_enable_reg_pp0_iter0	1	0	1	0
ap_enable_reg_pp0_iter1	1	0	1	0
ap_enable_reg_pp0_iter2	1	0	1	0
ap_enable_reg_pp0_iter3	1	0	1	0
coeff_hw_V_load_reg_872	32	0	32	0
exitcond_reg_848	1	0	1	0
i_reg_290	10	0	10	0
icmp_reg_839	1	0	1	0
iteration_V	3	0	3	0
iteration_V_load_reg_833	3	0	3	0
l_r_indicator_V	1	0	1	0
l_r_indicator_V_load_reg_827	1	0	1	0
p_0224_1_reg_280	9	0	9	0
p_Val2_2_reg_882	56	0	56	0
p_Val2_s_reg_268	41	0	41	0
packet_cntr_V	16	0	16	0
res_tdata_V_1_payload_A	32	0	32	0
res_tdata_V_1_payload_B	32	0	32	0
res_tdata_V_1_sel_rd	1	0	1	0
res_tdata_V_1_sel_wr	1	0	1	0
res_tdata_V_1_state	2	0	2	0
res_tlast_1_payload_A	1	0	1	0
res_tlast_1_payload_B	1	0	1	0
res_tlast_1_sel_rd	1	0	1	0
res_tlast_1_sel_wr	1	0	1	0
res_tlast_1_state	2	0	2	0
smpl_V_load_reg_877	24	0	24	0
smpl_rd_num_V_read_reg_811	3	0	3	0
t_V_reg_821	16	0	16	0
tap_num_m1_V_read_reg_806	9	0	9	0
tlast_dnum_V_read_reg_816	16	0	16	0
tmp_1_reg_843	10	0	10	0
wr_addr_I_V	9	0	9	0
wr_addr_r_V	9	0	9	0
exitcond_reg_848	64	32	1	0
Total	416	32	353	0

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_AXILiteS_AWVALID	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_AWREADY	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_AWADDR	in	12	s_axi	AXILiteS	array
s_axi_AXILiteS_WVALID	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_WREADY	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_WDATA	in	32	s_axi	AXILiteS	array
s_axi_AXILiteS_WSTRB	in	4	s_axi	AXILiteS	array
s_axi_AXILiteS_ARVALID	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_ARREADY	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_ARADDR	in	12	s_axi	AXILiteS	array
s_axi_AXILiteS_RVALID	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_RREADY	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_RDATA	out	32	s_axi	AXILiteS	array
s_axi_AXILiteS_RRESP	out	2	s_axi	AXILiteS	array
s_axi_AXILiteS_BVALID	out	1	s_axi	AXILiteS	array
s_axi_AXILiteS_BREADY	in	1	s_axi	AXILiteS	array
s_axi_AXILiteS_BRESP	out	2	s_axi	AXILiteS	array
ap_clk	in	1	ap_ctrl_none	fir_hw	return value
ap_rst_n	in	1	ap_ctrl_none	fir_hw	return value
input_I_V	in	24	ap_hs	input_I_V	pointer
input_I_V_ap_vld	in	1	ap_hs	input_I_V	pointer
input_I_V_ap_ack	out	1	ap_hs	input_I_V	pointer
input_r_V	in	24	ap_hs	input_r_V	pointer
input_r_V_ap_vld	in	1	ap_hs	input_r_V	pointer
input_r_V_ap_ack	out	1	ap_hs	input_r_V	pointer
res_TDATA	out	32	axis	res_tdata_V	pointer
res_TVALID	out	1	axis	res_tlast	pointer
res_TREADY	in	1	axis	res_tlast	pointer
res_TLAST	out	1	axis	res_tlast	pointer

6.1. Ábra – *A szűrő szintézise után kapott eredmények*

A 18x25-ös DSP MAC-ek közül 3-at is használ a szintetizált modell, viszont elméletileg 2-vel meglehetne oldani a FIR szűrésnél használt műveleteket.

7. fejezet

A hálózati kommunikáció megvalósítása

Az utolsó gyakorlat alkalmával egy UDP alapú hálózati kommunikációt kellett megvalósítanunk, a számítógép és az FPGA-n lévő beágyazott processzoros rendszer között. Ehhez az Ethernet interfész fölé definiált LightweightIP TCP/IP stack-et használtuk. A fejlesztés során az SDK-ban elérhető lwip141 példaalkalmazást módosítottuk, hogy képes legyen lekezelni az UDP csomagokat a PC és az FPGA között.

A példa alkalmazásban két forrásfájl volt: main.c és echo.c. Ez előbbi a főprogramot míg az utóbbi az alkalmazásspecifikus részt tartalmazta.
A main.c-ben csak a MAC címet és az IP címet kellett megfelelően beállítanunk.

7.1. Kód – *A main.c-ben végrehajtott módosítások*

```
    /* the mac address of the board. this should be unique per board */
    unsigned char mac_ethernet_address[] =
    { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x08 };

...
    /* initliaze IP addresses to be used */
    IP4_ADDR(&pcip,  192, 168,   1, 176);
    IP4_ADDR(&ipaddr,  192, 168,   1, 61);
    IP4_ADDR(&netmask, 255, 255, 255,  0);
    IP4_ADDR(&gw,      192, 168,   1,  1);
```

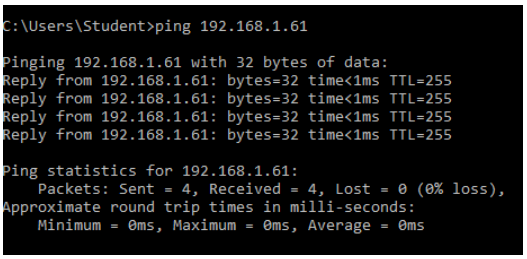
Az echo.c-t átneveztük application.c-re és az ott szereplő függvényeket átírtuk azok UDP változataira, ezt mutatja a 7.2. ábra.

7.2. Kód – *Az application.c-ben megvalósított UDP-kezelő függvények*

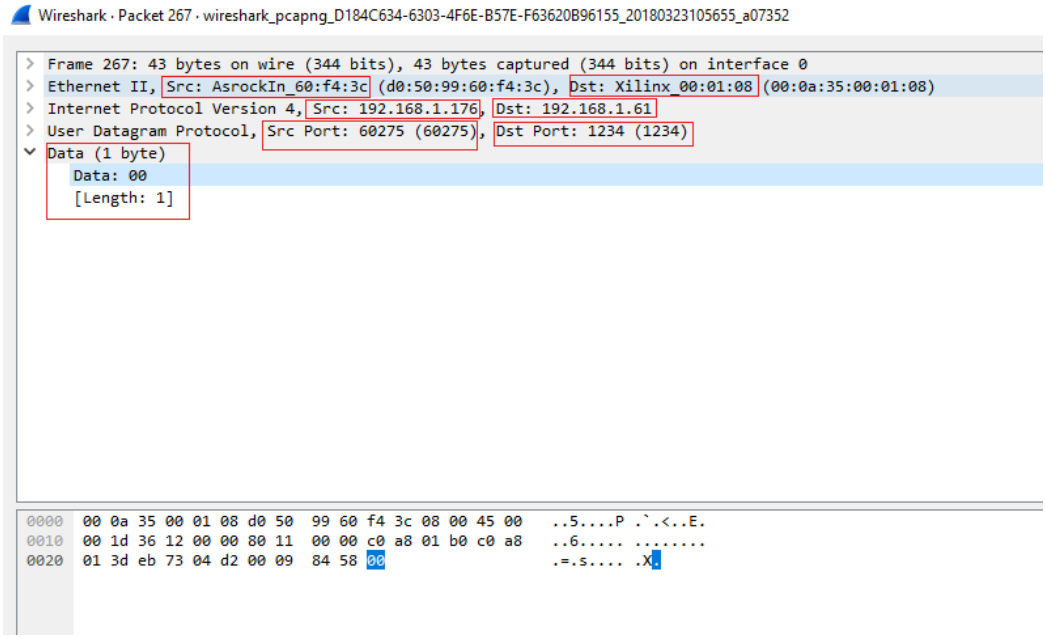
```
// Global variable to hold command of the UDP viewer application
int send_data = 0;
// Array to store 1024 byte to transmit
int data_to_pc [256];

struct udp_pcb *pc_pcb;
unsigned pc_port;
ip_addr_t pc_addr;

...

// Transmit data to the IP address
int transfer_data(ip_addr_t *ip) {
    // If the command is 0x01 from the UDP viewer
    if (send_data){

        // Create a packet buffer for 1024 bytes
        struct pbuf *p;
        p = pbuf_alloc(PBUF_TRANSPORT, 1024, PBUF_REF);
        // Point its payload to global data array
        p->payload = data_to_pc;
        // Send them
        udp_sendto(pc_pcb, p, &pc_addr, pc_port);
        // Destruct the packet buffer
        pbuf_free(p);
    }
    return 0;
}

...

// Callback function: this will get called on an UDP packet reception on the FPGA
void recv_callback(void *arg, struct udp_pcb *pcb,
                   struct pbuf *p, ip_addr_t *addr, u16_t port)
{
    pc_pcb = pcb;
    pc_port = port;
    pc_addr = *addr;

    // Do not read the packet if we are not in ESTABLISHED state
    if (!p) {
        udp_recv(pcb, NULL, NULL);
        return;
    }

    // If the received data is 0x00 then the UDP viewer
    // wants from the FPGA to send out data
    if (*((int*)(p->payload)) == 0x00){
```

```
        send_data = 1;
    }
    // If it is 0x01 then the transmit should be stopped
    else if (*((int*)(p->payload)) == 0x01){
        send_data = 0;
    }
    // Destruct the packet buffer
    pbuf_free(p);

    return;
}

...

int start_application()
{
    // Pointer to UDP protcol control block
    struct udp_pcb *pcb;
    // Used port
    unsigned port = 1234;
    // Return values for udp_* methods
    err_t err;

    // Test data to UDP viewer software application
    int i;
    for (i=0; i<128; i++){
        data_to_pc [2*i] = i;
        data_to_pc [2*i+1] = i;
    }

    // Create new PCB
    pcb = udp_new();
    if (!pcb) {
        xil_printf("Error creating PCB. Out of Memory\n\r");
        return -1;
    }

    // Assign IP and Port to the used PCB
    err = udp_bind(pcb, IP_ADDR_ANY, port);
    if (err != ERR_OK) {
        xil_printf("Unable to bind to port %d: err = %d\n\r", port, err);
        return -2;
    }

    // Register the PCB struct and the callback function
    // with UDP reception handler
    udp_recv(pcb, recv_callback, NULL);

    return 0;
}
```

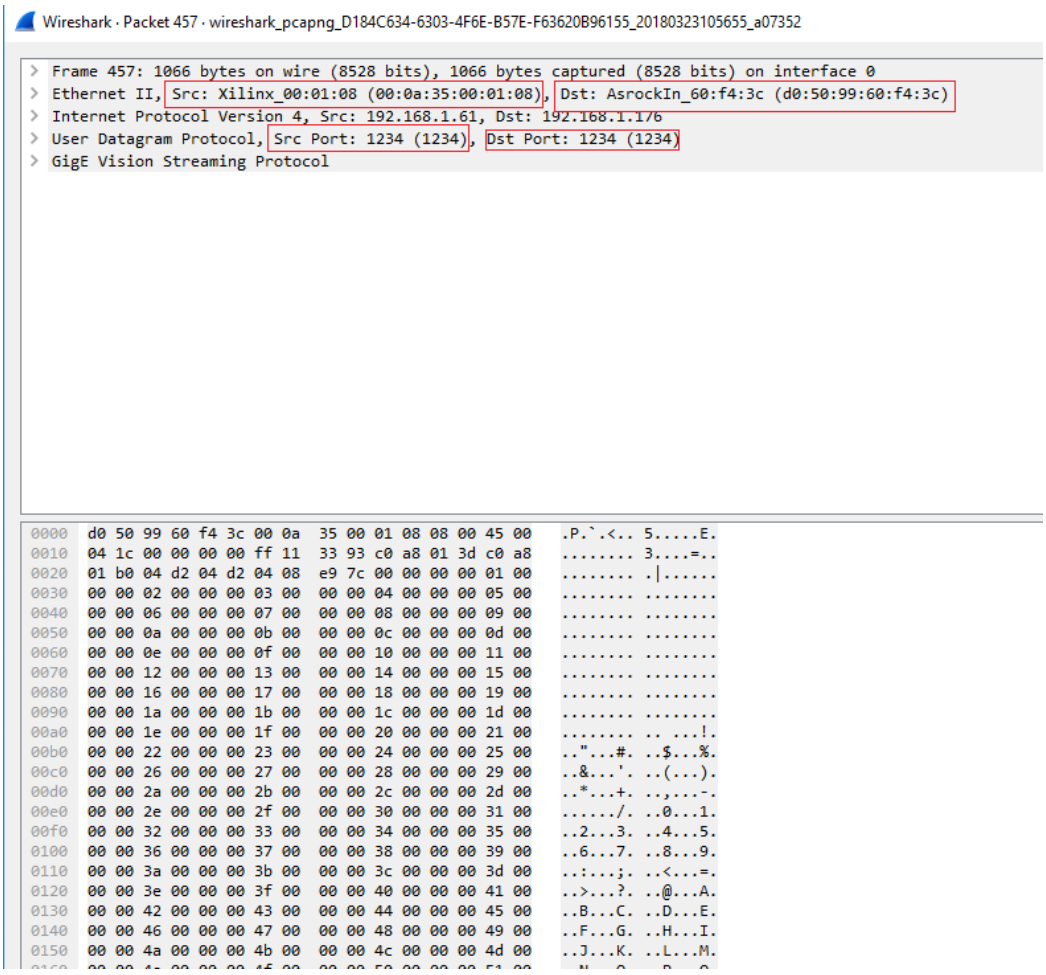
Az így kapott kapcsolatot először a Windows ping parancsával ellenőriztük le, ez szerepel a 7.1. ábrán, valamint a Wireshark programmal is megtettük ugyanezt. Ezeket a 7.2. és a 7.3. ábrákon lehet látni.



7.1. Ábra – *A hálózati kapcsolat ellenőrzése a Windows ping paranccsal*

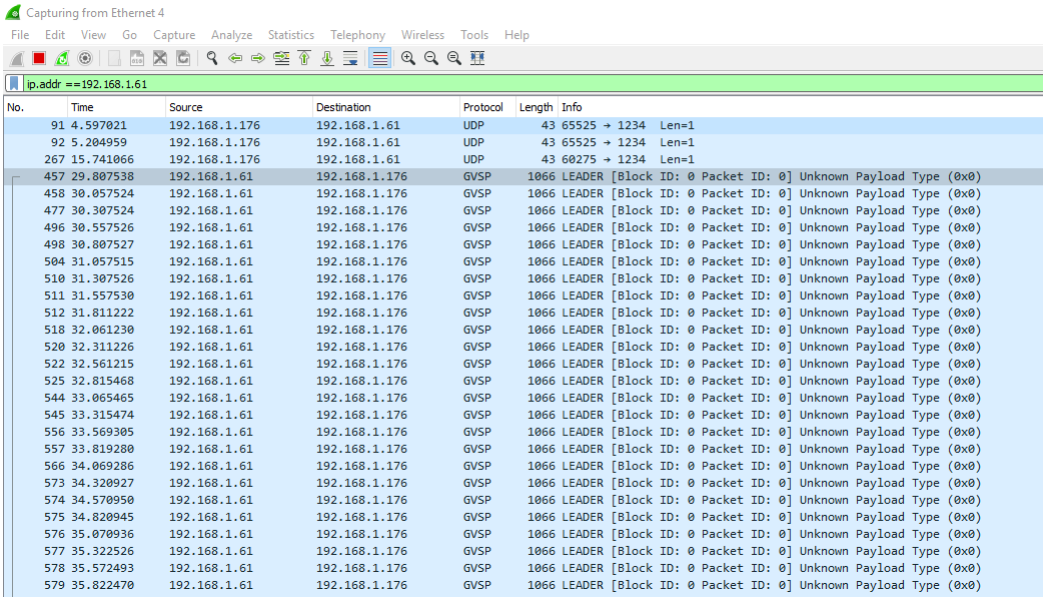


7.2. Ábra – *A hálózati kapcsolat ellenőrzése Wireshark-kal*



7.3. Ábra – *Az FPGA válasza a Wireshark-on küldött lekérdezésre*

Miután létrejött az adatkapcsolat, kipróbáltuk az adatátvitelt a 7.2. kódrészlet 74. és 76. sora közötti adatokkal. Az FPGA-ról indított UDP broadcastot a 7.4. ábra, míg a PC által fogadott adatokat a 7.5. ábra szemlélteti.



7.4. Ábra – *UDP csomagok indítása az FPGA-ról*

7.5. Ábra – *UDP csomagok fogadása a PC-n*

A 7.5. ábrán látható hullámformához mindkét csatornához 0. . .127 mintát állítottunk elő, így 256 darab tesztmintát tudtunk küldeni csomagonként. Ennek megfelelően, 1024 minta megjelenítésénél, mindkét csatornán 8 periódusnyi fűrészfog jelnek kell látszódnia.