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

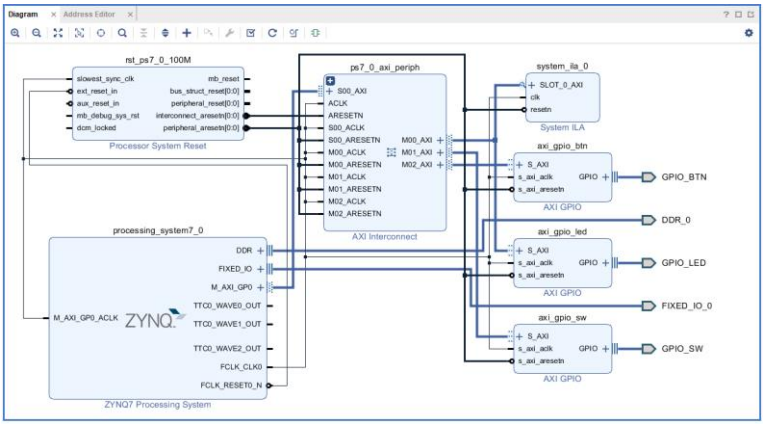
FPGA TERVEZŐI LABORATÓRIUM 2018.

Hallagtó 1.: Cseh Péter, DM5HMB

Hallgató 2.: Hajnal Bence, EWHV0Q

1 Egyszerű processzoros rendszer létrehozása

1.1 A rendszer blokkvázlata:



Címkiosztás:

Diagram | Address Editor

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

1.2 C kód

A LED-eken ~másodperc periódusidejű bináris számlálót megvalósító C kód:

```
#include "xparameters.h"
#include "xil_io.h"

int main()
{

    // init gpio: set LEDs as output, switches and buttons as input
    Xil_Out32(XPAR_AXI_GPIO_LED_BASEADDR + 4, 0x00);
    Xil_Out32(XPAR_AXI_GPIO_SW_BASEADDR + 4, 0xFF);
    Xil_Out32(XPAR_AXI_GPIO_BTN_BASEADDR + 4, 0xFF);
    // prevent optimization
    volatile int delay;
    int cntr = 0;

    while(1)
    {
        //1mp wait ~about 20 instruction/loop increment
        for(delay = 0; delay < (XPAR_CPU_CORTEXA9_0_CPU_CLK_FREQ_HZ / 20);
delay++) ;

        //write to led
        Xil_Out32(XPAR_AXI_GPIO_LED_BASEADDR, cntr);

        //increment counter
        cntr++;
    }
}
```

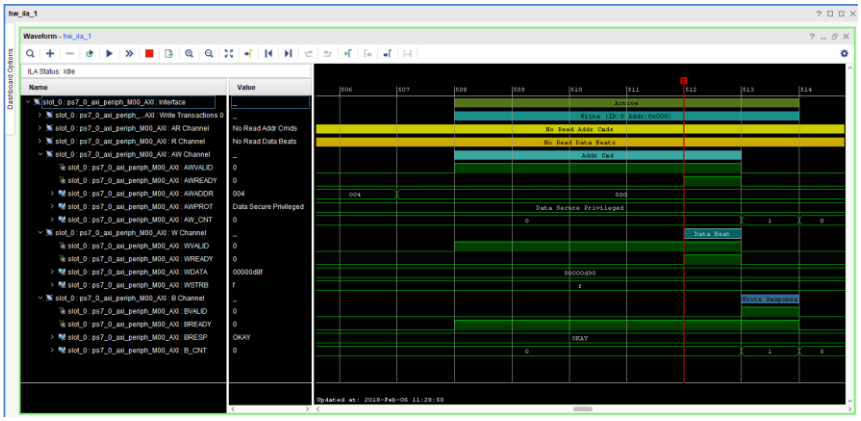
1.3 ChipScope hullámformák

1.3.1 AXI4 Lite írási buszciklus:

ChipScope trigger beállítások

A trigger feltétel az írási cím busz handshake jeleinek aktív állapota, tehát akkor van trigger ha az AWVALID magas logikai szinten van ÉS az AWREADY is magas logikai jelszinten van. A mintaregisztrátumba a trigger esemény előtt és után 512-512 minta kerül.

ChipScope hullámforma



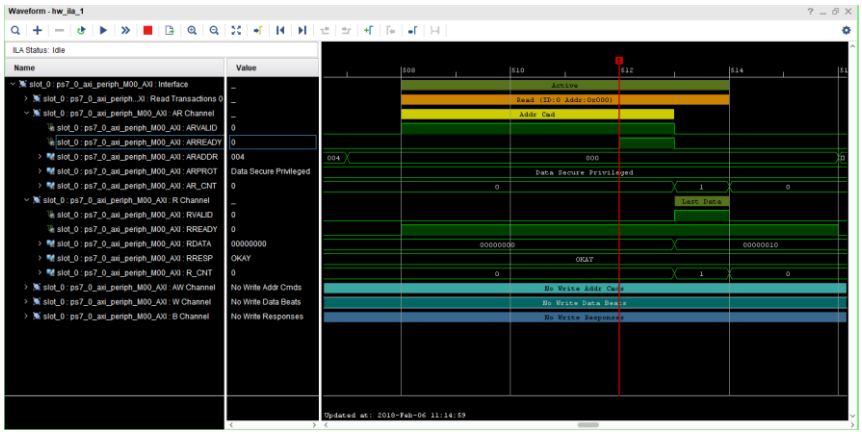
Az ábrán az AXI4 Lite írás művelet 6 órajelig tart. Az átvitel kezdetekor a master eszköz magas jelszintre állítja az AWVALID, WVALID, BREADY jeleket, ezzel jelzi, hogy az általa a vonalra kiadott cím, adat érvényes, valamint készen áll fogadni a slave eszköz reakcióját. Az átvitel ott történik meg ahol a csatornákon a handshake jelek magas logikai szinten vannak. Az írás művelet a piros függőleges vonallal jelzett órajelben történik meg, ekkor aktívak a handshake jelek az írási cím és írási adat vonalon. A slave eszköz válasza ez után egy órajellel érkezik vissza a masterhez a response csatornán (ez a BVALID jel).

1.3.2 AXI4 Lite olvasási buszciklus:

ChipScope trigger beállítások ÁBRA

Képet nem készítettünk. A trigger feltétel az olvasási cím busz handshake jeleinek aktív állapota, tehát akkor van trigger ha az ARVALID magas logikai szinten van ÉS az ARREADY is magas logikai jelszinten van. A mintaregisztrátumba a trigger esemény előtt és után 512-512 minta kerül.

ChipScope hullámforma ÁBRA + részeletes magyarázat

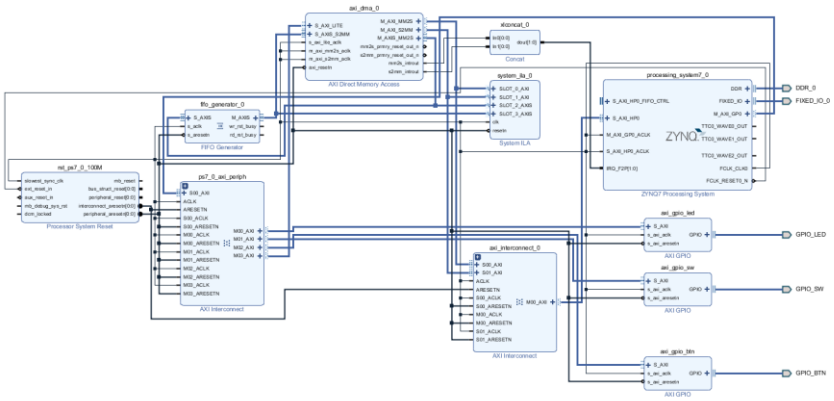


A hullámformán a busz két olvasási csatornája látható, az olvasás cím és olvasás adat csatornák. Az olvasás művelet három órajelig tart. A kezdetet jelzi az ARVALID és RREADY jelek felfutó éle, amellyel azt jelzi a master eszköz, hogy az általa kiadott olvasási cím érvényes, és kész fogadni a slave eszköz válaszát. Az olvasási cím kiadása akkor történik meg, amikor mindkét handshake jel magas állapotban van, ez a piros függőleges vonallal jelzett időpontban van. Ez után egy órajellel a slave válaszol az adattal és a tranzakció befejeződik.

2 DMA vezérlő megvalósítása és tesztelése

2.1 A rendszer blokkvázlata:

ÁBRA



Címkiosztás:

ÁBRA

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	0x4122_0000	64K	0x4122_FFFF
axi_gpio_led	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_gpio_sw	S_AXI	Reg	0x4121_0000	64K	0x4121_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 A DMA vezérlő regiszterei:

Offszet Cím	Név	A regiszter funkciója (bitek ismertetése, illetve a funkcionalitás szöveges leírása)
0x0	MM2S_DMACR	Memory Map to Stream Control Register. - 0. bit, RS: start/stop bit - 2. bit, Reset: DMA core soft reset-je - 3. bit, Keyhole: olvasás kezdeményező bit - 4. bit, Cyclic BD Enable: engedélyezi a cyclic buffer descriptor módot - 12. bit, IOC_IrqEn: Interrupt On Complete Interrupt Enable - 13. bit, Dly_IrqEn: Delay Timer Interrupt Enable - 14. bit, Err_IrqEn: Interrupt on Error Interrupt Enable - 16-23. bit, IRQThreshold: megszakítás küszöb érték. Ha az Interrupt On Control esemény bekövetkezik, akkor egy számláló elkezd lefelé számolni az itt beállított értékről, és ha eléri a nullát, akkor a DMA megszakítást generál - 24-31. bit, IRQDelay: Interrupt Delay Time Out.
0x4	MM2S_DMASR	Memory Map to Stream Status Register. - 0. bit, Halted: Jelzi, hogy a DMA csatorna fut/áll - 1. bit, Idle: Jelzi, hogy a DMA csatorna IDLE állapotban van (pl egy transfer befejeződött). - 3. bit, SGIncId: Jelzi, hogy a Scatter Gather mód be van kapcsolva vagy sem - 4. bit, DMAIntErr: DMA Internal Error, értéke 1, ha a fetched descriptorban a buffer hossza 0. - 5. bit, DMASlvErr: DMA Slave Error, akkor következik be, ha slave olvasás közben a Memory Map interfészen slave error hiba történik - 6. bit, DMADecErr: DMA Decode Error - 8. bit, SGIntErr: Scatter Gather Internal Error

		- 9. bit, SGSlvErr: Scatter Gather Slave Error - 10. bit, SGDecErr: Scatter Gather Decode Error - 12. bit, IOC_Irq: Interrupt On Complete - 13. bit, Dly_Irq: Interrupt on Delay - 14. bit, Err_Irq: Interrupt on Error - 16-23. bit, IRQThresholdSts: Interrupt Threshold Status - 24-31. bit, IRQDelaySts: Interrupt Delay Status
0x18	MM2S_SA	Source Address Register. DMA olvasási címe
0x1C	MM2S_SA_MSB	Source Address Register. Ha a címtartomány 32 biten nem fér el, akkor ez a regiszter használható a címtartomány bővítésére. Az olvasási cím felső 32 bitje van benne.
0x28	MM2S_LENGTH	Megadja, hogy a DMA olvasás során hány bájtot olvas a DMA. Az alsó 23 bit használható.
0x30	S2MM_DMACR	Stream to Memory Map Control Register. A bitek leírása megegyezik a MM2S_DMACR regiszter bitjeivel
0x34	S2MM_DMASR	Stream to Memory Map Status Register. A bitek leírása megegyezik a MM2S_DMASR regiszter bitjeivel
0x48	S2MM_DA	Destination Address Register. A DMA átvitel írási címe.
0x4C	S2MM_DA_MSB	Destination Address Register. Ha a címtartomány 32 biten nem fér el, akkor ez a regiszter használható a címtartomány bővítésére. Az olvasási cím felső 32 bitje van benne.
0x58	S2MM_LENGTH	Megadja, hogy a DMA írás során hány bájtot ír a DMA. Az alsó 23 bit használható.

2.3 C kód

Egyetlen 128 byte-os átvitelt (olvasás és írás) megvalósító C kód (olvashatóan tördelve, syntax highlight-tal, SAJÁT kommentekkel):

KÓD

```
// DMA átvitel mérete bájtokban (128)
#define MAX_PKT_LEN          0x80

// átvitel során a test adat első értéke
#define TEST_START_VALUE     0x05

//hányszor küldi el
#define NUMBER_OF_TRANSFERS  1

int XAxiDma_SimplePollExample(u16 DeviceId)
{
          XAxiDma_Config *CfgPtr;
          int Status;
          int Tries = NUMBER_OF_TRANSFERS;
          int Index;
          u8 *TxBufferPtr;
          u8 *RxBufferPtr;
          u8 Value;

          //Bufferek célcímének megadása
          TxBufferPtr = (u8 *)TX_BUFFER_BASE ;
          RxBufferPtr = (u8 *)RX_BUFFER_BASE;

          /* DMA configurációs mem,óriaterületének lekérése */
          CfgPtr = XAxiDma_LookupConfig(DeviceId);
          if (!CfgPtr) {
                    xil_printf("No config found for %d\r\n", DeviceId);
                    return XST_FAILURE;
          }
           /* A DMA inicializálása */
          Status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
          if (Status != XST_SUCCESS) {
                    xil_printf("Initialization failed %d\r\n", Status);
                    return XST_FAILURE;
          }
           /* Scatter-Gather mód kikapcsolt állapotának ellenőrzése */
          if(XAxiDma_HasSg(&AxiDma)){
                    xil_printf("Device configured as SG mode \r\n");
                    return XST_FAILURE;
          }

          /* Megszakitások kikapcsolása: lekérdezéses módban használjuk */
          XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
                                                    XAXIDMA_DEVICE_TO_DMA);
          XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
                                                    XAXIDMA_DMA_TO_DEVICE);

          Value = TEST_START_VALUE;

          // Transfer buffer feltöltése a teszt adattal + ciklusváltozóval (8 biten)
          for(Index = 0; Index < MAX_PKT_LEN; Index ++) {
                              TxBufferPtr[Index] = Value;

                              Value = (Value + 1) & 0xFF;
          }
          /* DMA bufferek korábbi adatoktól való megtisztítása */
          Xil_DCacheFlushRange((UINTPTR)TxBufferPtr, MAX_PKT_LEN);
#ifdef __aarch64__
          Xil_DCacheFlushRange((UINTPTR)RxBufferPtr, MAX_PKT_LEN);
#endif

          //DMA átvitel
          for(Index = 0; Index < Tries; Index ++) {
```

```
                //RX buffer-be beolvasás
                Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) RxBufferPtr,
                                        MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);

                if (Status != XST_SUCCESS) {
                        return XST_FAILURE;
                }

                //küldés TX bufferből
                Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) TxBufferPtr,
                                        MAX_PKT_LEN, XAXIDMA_DMA_TO_DEVICE);

                if (Status != XST_SUCCESS) {
                        return XST_FAILURE;
                }
                // Várakozás az adatátvitel befejezésére
                while ((XAxiDma_Busy(&AxiDma,XAXIDMA_DEVICE_TO_DMA)) ||
                        (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE))) {
                                /* Wait */
                }
                // Adatok helyességének ellenőrzése
                Status = CheckData();
                if (Status != XST_SUCCESS) {
                        return XST_FAILURE;
                }

        }

        /* A teszt sikeresen lefutott */
        return XST_SUCCESS;
}
```

Kérdések:

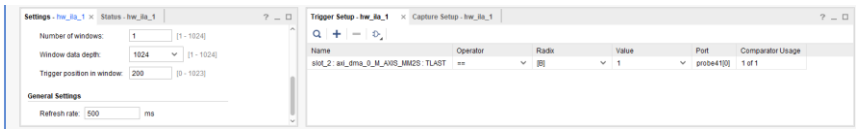
- Hol helyezkedik el a DMA két buffere (transmit és receive)?
  Block RAM-ban
- Az S2MM vagy MM2S irány felprogramozása történik először?
  MM2S
- Általában miért preferált megoldás, hogy az adatútban szereplő blokkok felprogramozása a céltól a forrás felé haladó sorrendben történik?

Mert ellenkező esetben a cél felé haladva az egyes blokkok nem állnak még készen az adatok fogadására. Ez rossz működést eredményezhet, illetve az adatút bufferei megtelhetnek, mielőtt ténylegesen használatra (kiolvasásra) kerülnének.

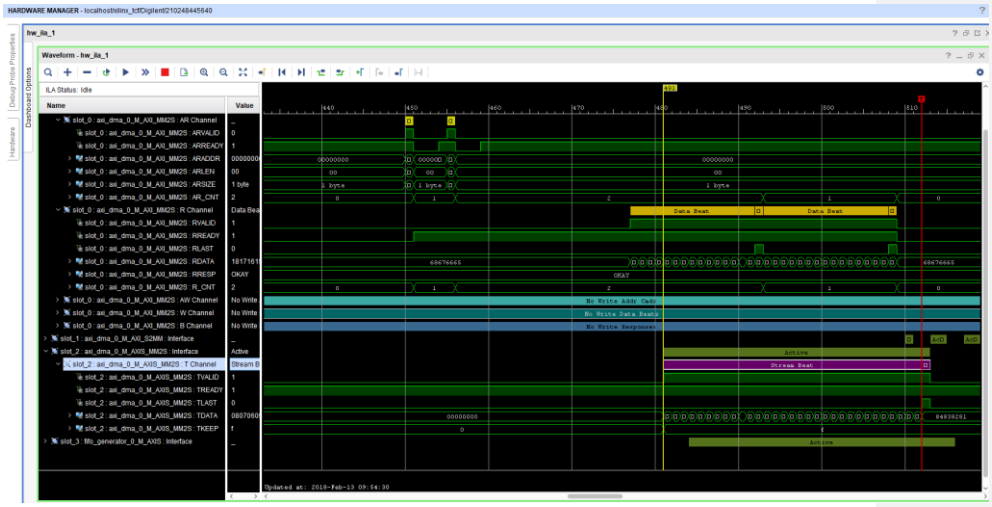
2.4 Átvitel vizsgálata ChipScope-pal

2.4.1 MM2S irány (Processzoros alrendszer → FIFO) vizsgálata egyetlen 128 byte-os átvitel esetén

Chipscope trigger beállítások ÁBRA



ChipScope hullámforma ÁBRA, a 2 lényeges interface összesen 3 channel-jével



Kérdések:

- Mennyi a teljes buszciklus ideje órajelben számolva?

  63

- Mennyi ebből a tényleges adatátvitel ideje? Hány százalék a busz hatékonysága?

  32 órajel. 32/63 = 50,8 %

- Hány burst-ben történik a 128 byte átvitele? Miből látszik ez (2 megoldás)?

  1 burst. Hullámformán látszik az AXI STREAM csatornán a VALID jel felfutó élétől a TLAST magas állapotáig tart egy burst.

- Mekkora a címzés és a tényleges adatátvitel közötti késleltetés órajelben? Mi az oka ennek? Hogyan csökkenthetnénk?

  31 órajel. ???

- Mennyi az AXI4 és az AXI4 Stream buszok közötti késleltetés? Mi lehet ennek az oka?

  4. órajel. ???

[P1] megjegyzést írt: Ezt nem tudom

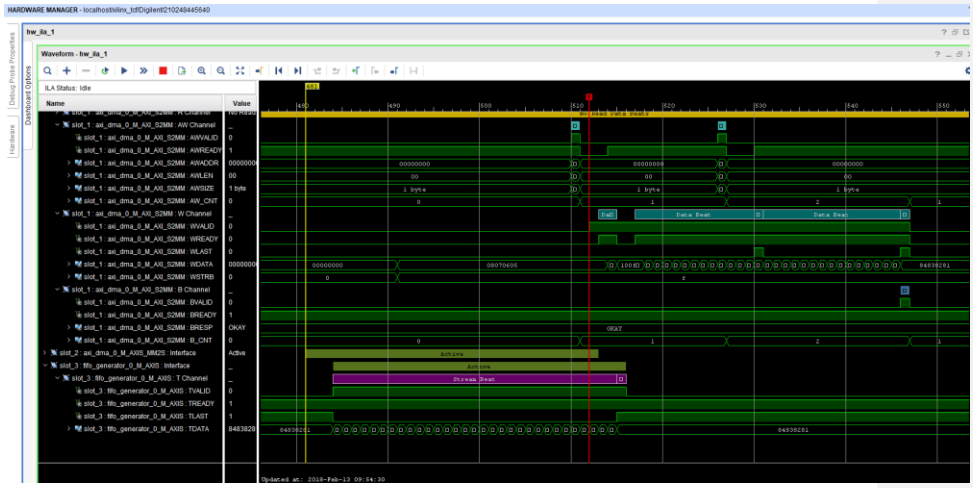
[P2] megjegyzést írt: Ezt se

2.4.2 S2MM irány vizsgálata egyetlen 128 byte-os átvitel esetén

Chipscope trigger beállítások ÁBRA

Ábra nincs. A triggerfeltétel a WVALID jel magas állapota

ChipScope hullámforma ÁBRA, a 2 lényeges interfész összesen 3 channel-jével



- Van-e lényeges különbség az MM2S irányhoz képest?

Nincs.

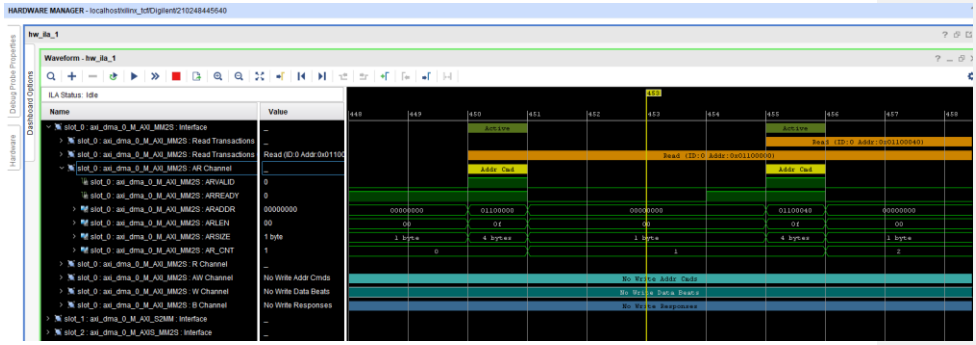
[P3] megjegyzést írt: Vagy mágis?

2.4.3 MM2S irány vizsgálata több 128 byte-os átvitel esetén

[P4] megjegyzést írt: Ehhez nincs elég forrás

Chipscope trigger beállítások ÁBRA

ChipScope hullámforma ÁBRA, a 2 lényeges interface összesen 3 channel-jével



Kérdések:

- Hova célszerű állítani a trigger pozíciót, ha minél több egymás utáni átvitelt szeretnénk látni?
- Kb. hány órajel telik el egy DMA befejezése és a következő DMA átvitel megkezdése között? Hogyan lehet ezt a legkisebbre csökkenteni?
- A csökkentés megvalósítása utáni ChipScope hullámforma ÁBRA

2.4.4 S2MM irány vizsgálata több 128 byte-os átvitel esetén

[P5] megjegyzést írt: Ehhez nincs elég forrás

- Chipscope trigger beállítások ÁBRA
- ChipScope hullámforma ÁBRA, a 2 lényeges interface összesen 3 channel-jével
- Van-e lényeges különbség az MM2S irányhoz képest?

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

[P6] megjegyzést írt: Ehhez nincs elég forrás

- Chipscope trigger beállítások ÁBRA
- ChipScope hullámforma ÁBRA, a 2 lényeges interface összesen 3 channel-jével
- Van-e lényeges különbség az első (egyetlen 128 byte-os átvitelt vizsgáló) esethez képest?

3 Audio interfész (I2S) megvalósítása

3.1 Right justified I2S átvitel hullámformája:

ÁBRA (CODEC dokumentáció)

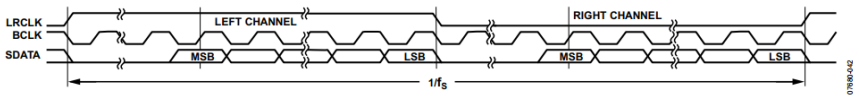


Figure 60. Right-Justified Mode—16 Bits to 24 Bits per Channel

Kérdések:

- Az interfész realizálása szempontjából miért egyszerűbb ez a mód a többinél?

  Azért, mert az adat párhuzamosításához elég egyetlen shift regiszter, amit folyamatosan működtetünk. Az adat érvényes vezérlő jelet is egyszerű generálni, ez az LRCLK élváltásakor logikai 1 értékű, egyébként 0.

- 96 kHz-es mintavételi frekvencia esetén mekkora a CODEC irányából érkező BCLK és LRCLK jelek frekvenciája? Detektálható-e ezen jelek éle megbízhatóan a 100 MHz-es rendszer órajellel?

  A CODEC adatlapja szerint egy mintavételi periódus (1/fs) alatt a bal és a jobb csatornát is fel kell dolgozni, ez az LRCLK teljes periódusa. Az egyik oldali csatorna adatainak feldolgozása 32 BCLK-t tartalmaz, tehát a teljes LRCLK periódus (mindkét csatorna) 2*32=64 BCLK-t tartalmaz. A BCLK detektálásához az órajelnek legalább a BCLK kétszeresének kell lennie, mert érzékelni kell, hogy a jelszint logikai alacsony értékből magas értékbe változik, tehát mindkét állapotból kell egy mintát venni. A rendszer órajelnek tehát legalább 96kHz*64*2 = 12,288 MHz-nek kell lennie. A 100MHz-es rendszer órajel osztásával ez teljesíthető.

3.2 A megvalósított modul HDL kódja

Az I2S interfészt megvalósító modul HDL kódja (olvashatóan tördelve, syntax highlight-tal, kommentekkel):

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

entity i2s is
    port
    (
        clk : in std_logic;
        rst : in std_logic;
        en : in std_logic;

        lrclk : in std_logic;
        bclk : in std_logic;
        sdi : in std_logic;

        adc_valid_l : out std_logic;
        adc_valid_r : out std_logic;
        adc_data : out std_logic_vector(23 downto 0)
    );
end i2s;

architecture rtl of i2s is

signal edge_ff_bclk : std_logic_vector(2 downto 0) := (others => '0');
signal bclk_rise : std_logic := '0';

signal edge_ff_lrclk : std_logic_vector(2 downto 0) := (others => '0');
signal lrclk_rise : std_logic := '0';
signal lrclk_fall : std_logic := '0';

signal adc_shr : std_logic_vector(23 downto 0) := (others => '0');
signal sdi_shift_ff : std_logic_vector(2 downto 0) := (others => '0');

begin

--Shift BCLK
proc_bclk: process(clk)
begin
    if(rising_edge(clk)) then
        if(rst = '1') then
            edge_ff_bclk <= (others => '0');
            edge_ff_lrclk <= (others => '0');
        else
            --Shift BCLK
            edge_ff_bclk <= edge_ff_bclk(1 downto 0) & bclk;

            --Shift LRCLK
            edge_ff_lrclk <= edge_ff_lrclk(1 downto 0) & lrclk;

            --Shift SDI
            sdi_shift_ff <= sdi_shift_ff(1 downto 0) & sdi;
        end if;
    end if;
end process proc_bclk;

--Detect rise & fall
bclk_rise <= '1' when (edge_ff_bclk(2) = '0' and edge_ff_bclk(1) = '1') else
'0';
lrclk_rise <= '1' when (edge_ff_lrclk(2) = '0' and edge_ff_lrclk(1) = '1')
else '0';
lrclk_fall <= '1' when (edge_ff_lrclk(2) = '1' and edge_ff_lrclk(1) = '0')
else '0';

--Shift ADC
proc_adc_shr : process(clk)
begin
    if(rising_edge(clk)) then
        if(bclk_rise = '1') then
            adc_shr <= adc_shr(22 downto 0) & sdi_shift_ff(2);
        end if;
    end if;
end process proc_adc_shr;

--ADC Valid
```

```
adc_valid_r <= lrclk_rise and en;
adc_valid_l <= lrclk_fall and en;

--ADC Data
adc_data <= adc_shr;

end rtl;
```

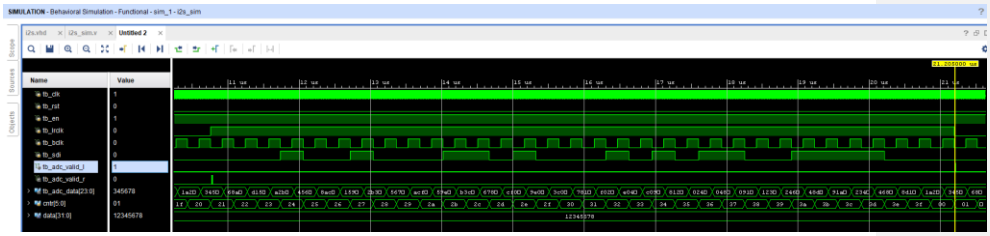
3.3 A tesztkörnyezet HDL kódja

A modul tesztelésére szolgáló egyszerű testbench kódja (olvashatóan tördelve, syntax highlight-tal, kommentekkel):

```
`timescale 1ns / 1ps

module i2s_sim ();
    reg tb_clk;
    reg tb_rst;
    reg tb_en;

    reg tb_lrclk;
    reg tb_bclk;
    reg tb_sdi;

    wire tb_adc_valid_l;
    wire tb_adc_valid_r;
    wire [23:0] tb_adc_data;

    i2s uut
    (
        .clk(tb_clk),
        .rst(tb_rst),
        .lrclk(tb_lrclk),
        .bclk(tb_bclk),
        .sdi(tb_sdi),
        .en(tb_en),
        .adc_valid_l(tb_adc_valid_l),
        .adc_valid_r(tb_adc_valid_r),
        .adc_data(tb_adc_data)
    );

    initial begin
        tb_clk <= 0;
        tb_rst <= 1;
        tb_en <= 1;
        tb_lrclk <= 0;
        tb_bclk <= 0;
        tb_sdi <= 0;
        cntr <= 0;

        #20
        tb_rst <= 0;
    end

    //GENERATE CLOCK (100MHz)
    always #5 tb_clk = ~tb_clk;

    //GENERATE BCLK (~6.144MHz)
    always #163 tb_bclk = ~tb_bclk;

    //GENERATE LRCLK (25MHz / 32), LOAD SDI
    localparam data = 32'h12345678;
    reg [5:0] cntr;

    always @ (negedge tb_bclk)
```

```
    begin
        if(tb_rst)
            cntr <= 0;
        else begin

            cntr <= cntr + 1;

            tb_lrclk <= cntr[5];

            tb_sdi <= data[31 - cntr[4:0]];

        end
    end
endmodule
```

3.4 Szimulációs hullámforma, magyarázattal:



A szimulációban generáltam a BCLK és LRCLK jeleket. A BCLK frekvenciája kb. 6,144 MHz, az LRCLK frekvenciája ennek a 64-ede. Az SDI jel generálásához egy konstans adatot használok (0x12345678), aminek minden BCLK lefutó élkor egy bitjét az SDI vonalra kiteszem, az MSB felől. Az enable jelet konstans 1-el gerjesztem.

A modul kimenetén azt kell látni, hogy minden LRCLK élváltást követően megjelenik az egyik csatorna valid jele (ha az enable jel értéke 1). Az LRCLK felfutó élekor a jobb csatorna valid, lefutó élekor a bal csatorna valid. A valid jel kiadásakor az adat kimeneten a konstans adat felső 24 bitjének megfelelő értéket kell látnom.

4 Audió CODEC interfész

4.1 Konfigurációs interfész (I2C) megvalósítása

4.1.1 CODEC MCLK órajel generálása

Kérdések:

- A CODEC adatlapja alapján (időzítési kritériumok (11. oldal) és órajel alrendszer (26. oldal)) 96 kHz-es mintavételi frekvencia eléréséhez mekkora MCLK órajel szükséges?

  Az adatlap 11. oldala alapján a CODEC MCLK órajel periódus ideje minimum 18,5 ns (1024 x fs módban), azaz a maximális frekvenciája az MCLK-nak 1/18,5ns = 54,054 MHz.

  Az MCLK meghatározásához az ADC/DAC mintavételi frekvenciájától indulunk el, az adatlap 30. ábrája alapján (26. oldal). Ahhoz, hogy a mintavételi frekvencia 96kHz legyen, ahhoz fs/0.5 sampling rate-et kell használni (27. oldal, 13. táblázat). Ekkor a „Core Clock" nevű csomópontban 48kHz frekvenciának kell lennie. A Control Clock Register-t ha 1024xfs módban használjuk és a CODEC PLL-jét nem használjuk, akkor a bemeneti MCLK frekvenciája 49,152MHz.

- Ebben az esetben az MCLK a mintavételi frekvenciának hányszorosa (256x, 512x vagy 1024x)?

  1024

- Az FPGA-ban generált 98,304 MHz frekvenciájú órajelnek ez hányad része? A rendszerbe illesztett bináris számlálónak melyik bitjét kell ehhez MCLK jelként kivezetni az FPGA-ból?

  A generált 98,304MHz frekvenciájú órajelnek ez a fele, tehát a számlálónak a 0. bitját fogjuk kivezetni MCLK jelént.

4.1.2 CODEC I2C interfész

ÁBRA: CODEC I2C írás időzítési diagram (CODEC adatlap 39. oldal)

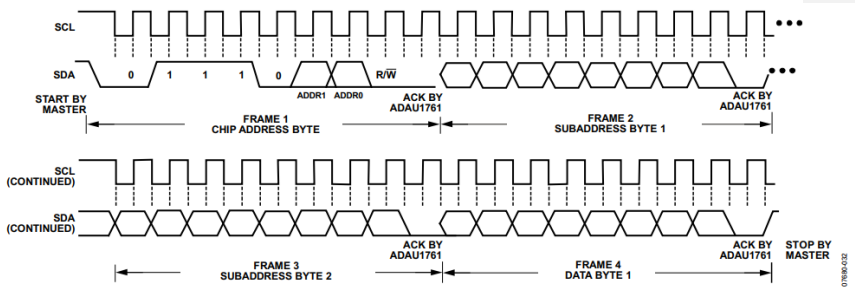


Figure 49. I²C Write to ADAU1761 Clocking

Kérdések:

- Mi azonosítja az I2C buszciklus kezdetét és végét?

  A master START és STOP jele

- Mi az egyes byte-ok szerepe az időzítési diagramon?

  Az egyszavas (egy bájtos) íráshoz szükség van 3 cím bájt küldésére (eszköz cím és 2 regiszter cím) és 1 bájt adat küldésére a master eszköz részéről.

- Mik az eszköz azonosító lehetséges értékei (chip address)? Milyen 2 bites értéket kell a 0x38 címhez beállítani a Xilinx I2C periféria GPO kimenetén?

  Lehetséges címek: 0x38 – 0x3B

  A GPO kimenetén 0b00 értéket kell beállítani

- Hány bites a CODEC regisztereinek címe?

  16

ÁBRA: CODEC I2C olvasási diagram (CODEC adatlap 39. oldal)

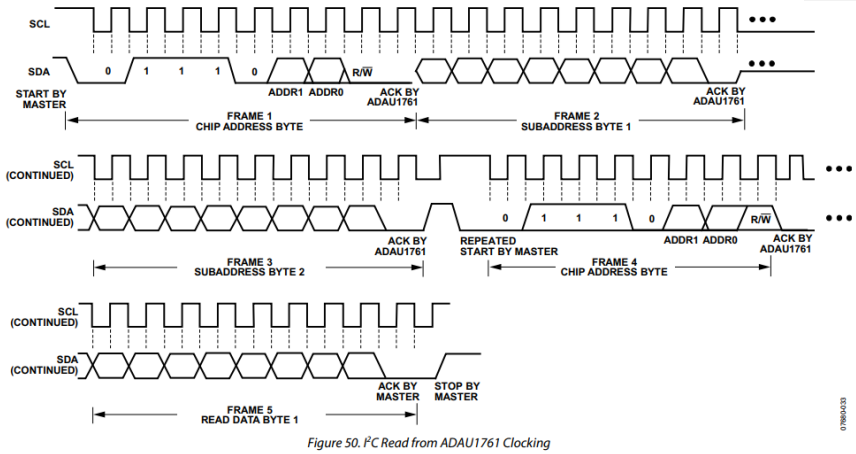


Figure 50. I²C Read from ADAU1761 Clocking

Kérdések:

- I2C busz esetén hogyan történik a tetszőleges címről történő olvasás? Azaz mi ez egyes byte-ok szerepe?
  Külön ki kell választani a regiszter címet és külön olvasni kell, ez két tranzakció.
  Eszköz címzése, írás művelet, regiszter megcímzése.
  Új start jel, eszköz cím, olvasás művelet.

4.2 Audió CODEC konfiguráció és ellenőrzés

4.2.1 Xilinx I2C periféria programozása

Kérdések:

- Vázlatosan ismertesse, hogy mi a különbség a Xilinx I2C periféria „Standard Controller Logic Flow” és a használt „Dynamic Controller Logic Flow” között?
  A Standard Controller Logic Flow esetén ha a TX FIFO-ba adatot írunk, akkor azt egyből elküldi. Dynamic Controller Logic Flow esetén először feltöltjük a TX FIFO-t és aztán adunk ki egy küldés parancsot, így kevesebb ideig kell foglalni az I2C buszt, mint Standard esetben.
- Mely regiszter címeket kell használni „Dynamic Controller Logic Flow” alkalmazása során (lásd útmutató, illetve Xilinx IIC dokumentáció 36. oldal)?
  TX_FIFO, Control Register

- Hány bites a transmit, illetve receive FIFO? Ebből hány bit a vezérlés (mik ezek a bitek), és hány bit az I2C SDA vonalra kerülő érték?

  TX_FIFO: 10 bites(2 vezérlő bit, 8 adat bit):

  9.bit: Stop

  8.bit: Start

  7-0.bit: TX Data

  RX_FIFO: 8 bites, csak RX Data

4.2.2 CODEC regiszterek beállítása

A mérési útmutatóban nem megadott regiszter értékek meghatározása. A táblázatokat bővítse szükség szerint.

Kérdések:

- A 4.1 pontban meghatározott órajelek használatához milyen értékekkel kell felprogramozni a CODEC megfelelő regisztereit (a mérési útmutatóban nem megadott regiszterek)?

Regiszter cím	Érték	A beállított érték hatása
0x4000	0x07	A clock control regiszterben engedélyezi az órajelet a DSP, DAC/ADC, Serial Port számára (0. bit). Valamint az input clock frekvenciához az 1024xfs osztást használja.

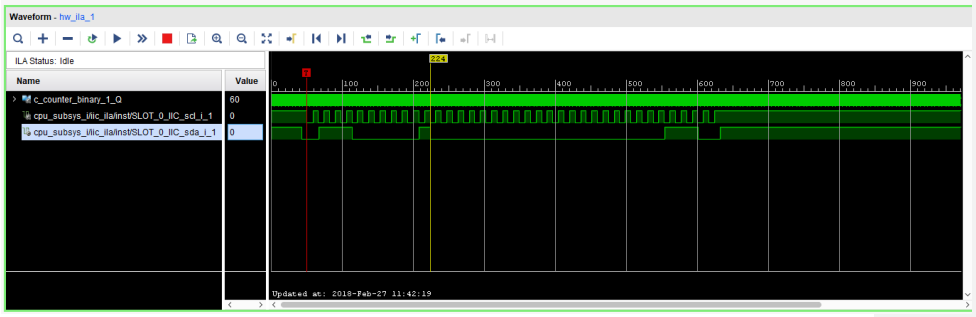
- Hogyan kell felprogramozni a CODEC megfelelő regisztereit ahhoz, hogy az I2S interfész „right justified” módban működjön?

Regiszter cím	Érték	A beállított érték hatása
0x4015	0x01	CODEC master módban, sztereó csatorna, LRCLK lefutó élekor kezdődik a bal csatorna, BCLK lefutó élekor történik az adatváltás, LRCLK 50%os kitöltési tényező.
0x4016	0x02	8 BCLK órajelet késik az SDI-n az adat az LRCLK élváltást követően (Right Justified Mode), MSB jelenik meg először az SDI-n, egy teljes LRCLK

		periódusban 64 BCLK órajel van

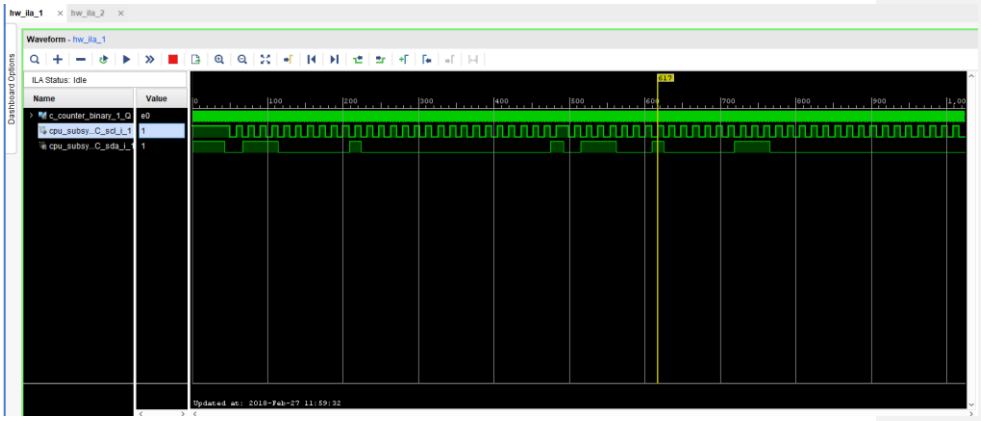
4.2.3 I2C átvitel ellenőrzése ChipScope-ban

ÁBRA: Egy I2C írás hullámformája. Megfelel a hullámforma az elvártnak?



Igen, a trigger feltétel a start jel. Ez után látszik az eszköz címe(0b0111000) és a R/W bit = 0, tehát ez egy írás lesz. Ez után a regiszter cím (2 bájt) és az adat

ÁBRA: Egy I2C olvasás hullámformája. Megfelel a hullámforma az elvártnak?



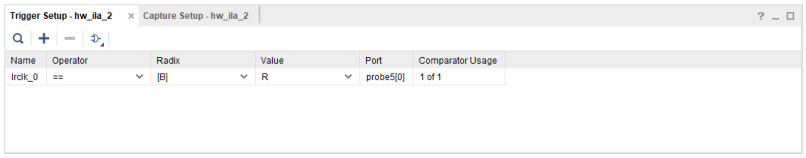
Igen, a trigger feltétel szintén a start jel. Ez után egy írás során megadjuk a regiszter címét. Utána repeated start és az olvasás.

4.2.4 I2S interfész ellenőrzése

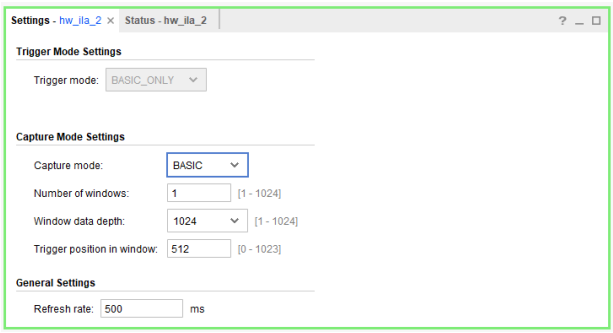
[P7] megjegyzést írt: Ehhez nincs elég forrás

Kösse össze a PC line-out kimenetét a ZedBoard bemenetével, majd generáljon PC-n szinusz jelet (pl. Audacity). Megfelelő ChipScope beállítások használatával ellenőrizze, hogy az I2S interfész kimenetének hullámformája megfelel-e a generált jelnek.

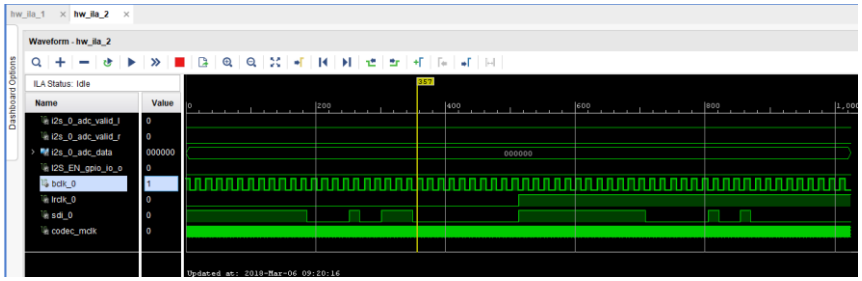
ÁBRA: ChipScope trigger beállítás



ÁBRA: ChipScope Storage beállítás



ÁBRA: ChipScope hullámforma



(Megjegyzés: A CODEC-et rosszul konfiguráltuk, ezért ezen az ábrán NEM Right Justified módban működik. Ezt később korrigáltuk)

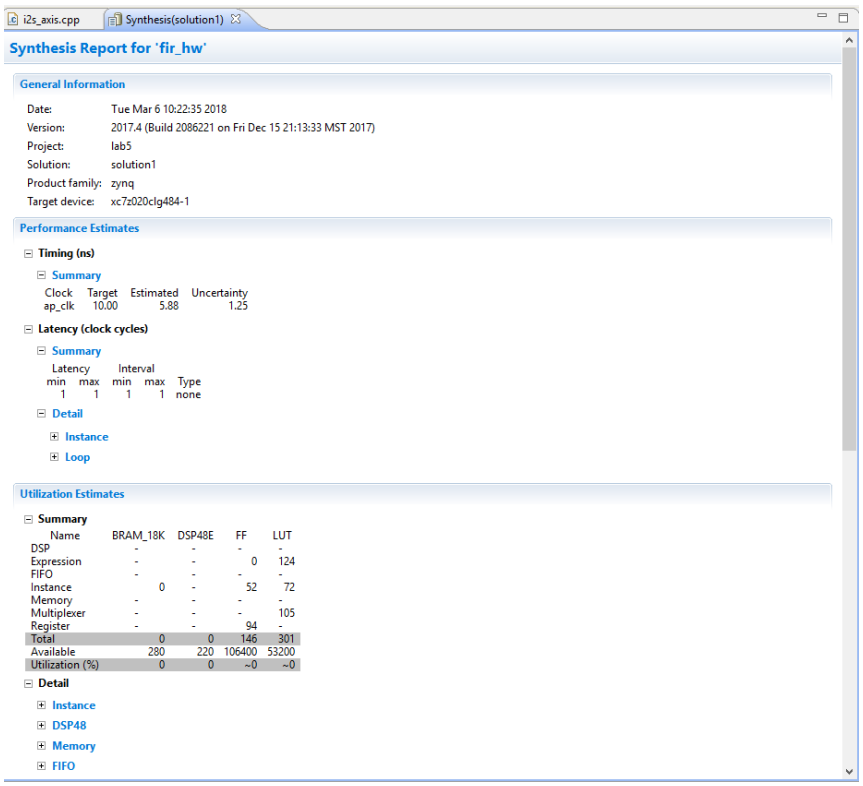
5 I2S → AXI Stream interfész megvalósítása Vivado HLS-ben

5.1 Vivado HLS

Vivado HLS kód (+syntax highlight, komment)

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

#include "i2s_axis.h"

void fir_hw(ap_uint<16> tlast_dnum, din_t *input_l, din_t *input_r,
          out_stream_struct *res)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS DATA_PACK variable=res
#pragma HLS INTERFACE axis register both port=res
#pragma HLS INTERFACE ap_hs port=input_r
#pragma HLS INTERFACE ap_hs port=input_l
#pragma HLS INTERFACE s_axilite port=tlast_dnum

          //számláló TLAST jelhez, és left/right
          static ap_uint<16> cntr = 0;
          static int lr = 0;

          //kimeneti változó
          out_stream_struct result;

          //TLAST generálása az utolsó adat esetén
          if(++cntr == tlast_dnum)
          {
                    result.tlast = true;
                    cntr = 0;
          }
          else
          {
                    result.tlast = false;
          }

          //Jobb és bal csatorna kiadása
          if(lr == 0)
          {
                    result.tdata = *input_l;
                    lr = 1;
          }
          else
          {
                    result.tdata = *input_r;
                    lr = 0;
          }

          //kimenet
          *res = result;
}
```

Vivado HLS fordítás eredményei



i2s_axis.cpp | Synthesis(solution1)

Synthesis Report for 'fir_hw'

General Information

Date: Tue Mar 6 10:22:35 2018
Version: 2017.4 (Build 2086221 on Fri Dec 15 21:13:33 MST 2017)
Project: lab5
Solution: solution1
Product family: zynq
Target device: xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.88	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	124
FIFO	-	-	-	-
Instance	0	-	52	72
Memory	-	-	-	-
Multiplexer	-	-	-	105
Register	-	-	94	-
Total	0	0	146	301
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Detail

Instance

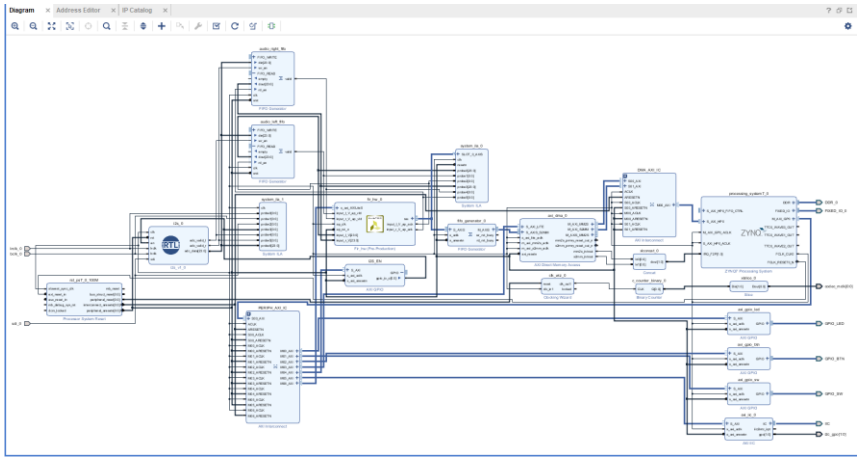
DSP48

Memory

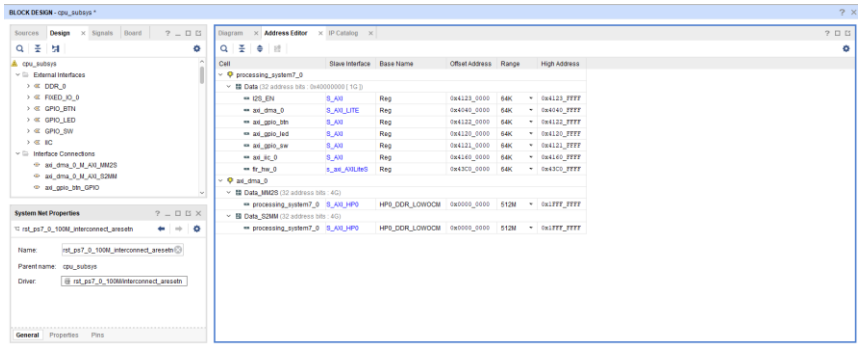
FIFO

5.2 IP beillesztése a rendszerbe

ÁBRA: Block Design blokkvázlat



ÁBRA: Block Design címkiosztás



5.3 Tesztelés

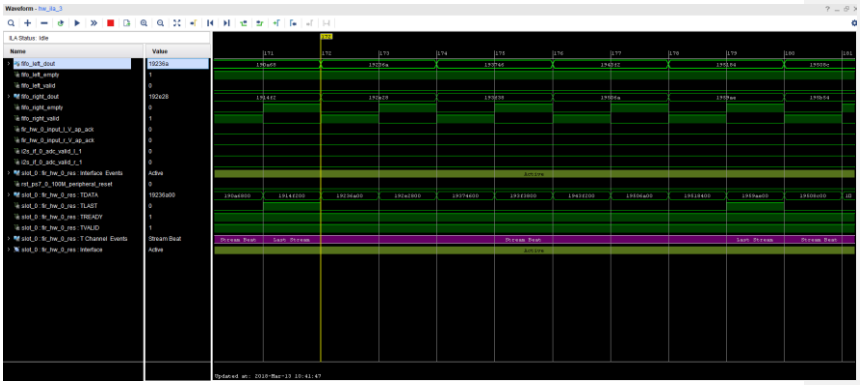
A teszteléshez használt C kód forráskódja (+syntax highlight, komment)

A működés vizsgálata ChipScope-ban

ÁBRA: trigger beállítások

Trigger feltétel a bal csatorna valid jele

ÁBRA: ChipScope hullámforma



6 FIR szűrő megvalósítása Vivado HLS-ben

A FIR szűrőt megvalósító HLS-kód:

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

#include "types.h"
#define N 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)
{
#pragma HLS INTERFACE s_axilite port=tlast_dnum
#pragma HLS INTERFACE axis register both port=res
#pragma HLS INTERFACE ap_hs port=input_r
#pragma HLS INTERFACE ap_hs port=input_l
#pragma HLS DATA_PACK variable=res
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=coeff_hw
#pragma HLS INTERFACE s_axilite port=smpl_rd_num
#pragma HLS INTERFACE s_axilite port=tap_num_m1
////////////////////////////////////////////////////////////////////////////////
//VARIABLES
////////////////////////////////////////////////////////////////////////////////
      //Left and Right channel buffer and select
      static din_t buff[2][N];
      static int lr = 0;
      static ap_uint<9> start_idx[2] = {0, 0};
      //Sample counter
      static ap_uint<16> sample_cntr[2] = {0, 0};
      //Output
      out_stream_struct out_data;
#pragma HLS DATA_PACK variable=out_data
      //TLAST counter
      static ap_uint<16> tlast_cntr = 0;
      //Iterator
      int i;

      //Acc, circular idx
      accu_t acc = 0;

      ap_uint<9> circ_idx = 0;

////////////////////////////////////////////////////////////////////////////////
//READ
////////////////////////////////////////////////////////////////////////////////
      if(lr == 0) {
            buff[lr][start_idx[lr]] = *input_l;
      }
      else {
            buff[lr][start_idx[lr]] = *input_r;
      }
////////////////////////////////////////////////////////////////////////////////
//VALID
////////////////////////////////////////////////////////////////////////////////
      int valid = 0;
      if(smpl_rd_num == 1)
            valid = 1;

      if(smpl_rd_num == 2)
            if(sample_cntr[lr] % 2 == 1)
                  valid = 1;

      if(smpl_rd_num == 4)
            if(sample_cntr[lr] % 4 == 3)
                  valid = 1;
////////////////////////////////////////////////////////////////////////////////
//MAC
////////////////////////////////////////////////////////////////////////////////
      if(valid == 1) {
            acc = 0;
            circ_idx = start_idx[lr];
            for_mac: for (i = 0; i <= tap_num_m1; i++) {
                  #pragma HLS PIPELINE II=1
                  #pragma HLS LOOP_TRIPCOUNT min=128 max=512

                  acc = acc + (coeff_hw[i] * buff[lr][circ_idx]);

                  circ_idx--;
            }
      }
////////////////////////////////////////////////////////////////////////////////
//OUTPUT
////////////////////////////////////////////////////////////////////////////////

      if(valid == 1) {
            tlast_cntr++;
            out_data.tdata = acc;
      }

////////////////////////////////////////////////////////////////////////////////
//GENERATE TLAST
////////////////////////////////////////////////////////////////////////////////

      if (tlast_cntr == tlast_dnum) {
            out_data.tlast = 1;
            tlast_cntr = 0;
      } else
            out_data.tlast = 0;

////////////////////////////////////////////////////////////////////////////////
//LEFT / RIGHT CHANNELS
////////////////////////////////////////////////////////////////////////////////

      start_idx[lr]++;
      sample_cntr[lr]++;

      if(lr == 0)
            lr = 1;
      else
            lr = 0;

      *res = out_data;
}
```

A pontosság növelésének érdekében a dirac-deltát érdemes a 1-nél kisebb vagy -1 nagyságúnak megválasztani. A szintézis eredménye megmutatja, hogy a késleltetés a elfogadható határon belül mozog (~1000 órajelenként érkezik újabb minta a codec felől és csak minden negyedik után kell elindítani a feldolgozást, így ~4000 órajelnyi idő áll rendelkezésre a feldolgozásra, ha 4-es decimálást választunk, a FIR szűrés pedig maximum 517 órajelig tart mintánként; ha a decimálás kisebb, akkor bár gyakrabban jönnek feldolgozandó minták, de arányaiban ugyanennyivel gyorsabban végbemegy a FIR szűrés is): a MAC műveletet képes 1 órajel alatt végrehajtani a realizált hardver a pipeline használatának köszönhetően.

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
3	518	3	518	none

Detail

Instance

N/A

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- for_mac	130	514	4	1	1	128 ~ 512	yes

A hardverigény is megfelelően kicsi: összesen 4 BRAM blokkra van szükség (csatornánként 1 a bejövő szimbólumoknak: “Memory” és kettő egyéb célra) és 3 DSP blokkra (2 db a 24*32 bites szorzásra, mivel 25*18-as szorzáskra képes egy DSP, és 1 az összeadásra a pipeline-osítás miatt).

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	678
FIFO	-	-	-	-
Instance	2	-	156	166
Memory	2	-	0	0
Multiplexer	-	-	-	213
Register	0	-	564	32
Total	4	3	720	1089
Available	280	220	106400	53200
Utilization (%)	1	1	~0	2

Detail

A szintézis eredményei kis különbséggel megegyeztek a portálon megtalálható megvalósítással, és a teszt is megfelelően működött (a portálon fent lévő teszt kóddal), azonban az importálás során problémák adódtak (nem megfelelően frissítettük az IP

katalógust, és az 5. laborgyakorlaton elkészített hardver maradt a tervben), így végül a tanszéki portálra feltöltött projekttel dolgoztunk a továbbiakban.

7 A hálózati kommunikáció megvalósítása

A példaalkalmazás importálása és a BSP generálása után elkészítettük a megfelelő funkcionalitásokat a mérési útmutató alapján.

```c
#include <stdio.h>
#include <string.h>

#include "xil_io.h"
#include "lwip/err.h"
#include "lwip/tcp.h"
#include "lwip/udp.h"
#if defined (__arm__) || defined (__aarch64__)
#include "xil_printf.h"
#endif

struct udp_pcb *udp;

/* 1024 Byte adat */
u32_t txPayload[256];
volatile u8_t udp_enable = FALSE;

err_t recv_callback(void *arg, struct udp_pcb *pcb, struct pbuf *p,
		ip_addr_t *addr, u16_t port);

int transfer_data() {
	struct pbuf *buffer;
	err_t error = 0;
	/* A DMA buffer címének használata: forrás a dma_handler forrásfájlban */
	extern unsigned long *dmaBufferPtr;
	/* A DMA adat engedélyező jele: forrás a dma_handler forrásfájlban */
	extern volatile int dataValid;

	if (udp_enable && dataValid) {
		/* Memória foglalás */
		buffer = pbuf_alloc(PBUF_TRANSPORT, 1024, PBUF_REF);
		/* Adatmutató beállítása és cache invalidálás */
		buffer->payload = dmaBufferPtr;
		Xil_DCacheInvalidateRange(dmaBufferPtr, 1024);

		error = udp_sendto(udp, buffer, &udp->remote_ip, udp->remote_port);
		if (error != ERR_OK) {
			xil_printf("Error during usp send! Error Code: %u\n\r", error);
		}
		/* A használt terület felszabadítása */
		pbuf_free(buffer);
	}

	return error;

}

void print_app_header() {
	xil_printf("\n\r\n\r-----lwIP TCP echo server ------\n\r");
	xil_printf("TCP packets sent to port 6001 will be echoed back\n\r");
}

int start_application() {
	err_t err;
	/* Az UDP port beállítása */
	unsigned port = 1235;
	int i = 0;
	ip_addr_t remote_ip;

	for (; i < 256; i++) {
		txPayload[i] = i;
	}

	/* create new UDP PCB structure */
	udp = udp_new();
	if (!udp) {
		xil_printf("Error creating PCB. Out of Memory\n\r");
		return -1;
	}

	err = udp_bind(udp, IP_ADDR_ANY, port);
	if (err != ERR_OK) {
		xil_printf("Unable to bind to port %d: err = %d\n\r", port, err);
		return -2;
	}

	/* Regist callback */
	udp_recv(udp, (udp_recv_fn) recv_callback, &udp);
	xil_printf("Init succeeddd!%d\n\r", udp->local_port);
	xil_printf("local ip: %x\n\r", udp->local_ip.addr);

	// todo: HERE NEED TO INIT DMA & FIR
	return 0;
}

/* Vételi esemény kezelőfüggvénye: a felhasználói alkalmazás által küldött
0 és 1 értékekre engedélyezni vagy tiltani kell az UDP küldést */
err_t recv_callback(void *arg, struct udp_pcb *pcb, struct pbuf *p,
		ip_addr_t *addr, u16_t port) {
	/* Bejövő adat: mindig 1 byte */
	u8_t rxPayload;

	rxPayload = *(u8_t*) p->payload;

	if (rxPayload) {
		udp_enable = FALSE;
		xil_printf("udp disable \n\r");
	} else {
		Xil_Out32(XPAR_AXI_GPIO_LED_BASEADDR, 0x06);
		pcb->remote_ip = *addr;
		pcb->remote_port = port;
		udp_enable = TRUE;
		xil_printf("udp enable \n\r");
	}
	/* Terület felszabadítása */
	pbuf_free(p);

	return ERR_OK;

}
```

Az application.c forrásfájl:

A megszakításos működésen alapuló alkalmazást nem sikerült elkészíteni, ezért az egyszerűbb lekérdezéses alkalmazást vettük alapul. Csak a DMA kezelő példakód módosításait/felhasznált részeit mutatjuk be:

```
#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        0x400
...
/* Globális változók a DMA adatátvitel kezeléséhez */
unsigned long dmaBufferSize;
unsigned long *dmaBufferPtr;

/* Adat rendelkezésre állás flag */
volatile int dataValid;
/* XAxiDma példányosítása */
static XAxiDma AxiDma;

int XAxiDma_SimplePollExample(void) {
     XAxiDma_Config *CfgPtr;
     int Status;
     int Tries = 1;
     int Index;
     u8 Value;

     /* Nem használunk dupla bufferelést, mert vannak elhagyott csomagok */
     dmaBufferPtr = (u8*) RX_BUFFER_BASE;
     dataValid = 1;

     /* XAxiDma inicializálás. */
     CfgPtr = XAxiDma_LookupConfig(DMA_DEV_ID);
     if (!CfgPtr) {
          xil_printf("No config found for %d\r\n", DMA_DEV_ID);
          return XST_FAILURE;
     }

     Status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
     if (Status != XST_SUCCESS) {
          xil_printf("Initialization failed %d\r\n", Status);
          return XST_FAILURE;
     }

     if (XAxiDma_HasSg(&AxiDma)) {
          xil_printf("Device configured as SG mode \r\n");
          return XST_FAILURE;
     }


     /* A megszakítások kikapcsolása, polling módot használunk */
     XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DEVICE_TO_DMA);
     XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DMA_TO_DEVICE);

     /* Data cache használata esetén Flush művelet elvégzése */
#ifdef __aarch64__
     Xil_DCacheFlushRange((UINTPTR)RxBufferPtr, MAX_PKT_LEN);
#endif

     /* DMA művelet végrehajtása (csak 1 irányt használunk) */
     Status = XAxiDma_SimpleTransfer(&AxiDma, (UINTPTR) dmaBufferPtr,
     MAX_PKT_LEN, XAXIDMA_DEVICE_TO_DMA);
     if (Status != XST_SUCCESS) {
          return XST_FAILURE;
     }

     /* Nem várunk a művelet végrehajtásának végéig, mert vannak
     elveszett csomagok */
     return XST_SUCCESS;
}
```

A felhasználást bemutató kód (main.c):

```
#include <stdio.h>
#include "xparameters.h"
#include "netif/xadapter.h"
#include "platform.h"
#include "platform_config.h"
#if defined (__arm__) || defined(__aarch64__)
#include "xil_printf.h"
#endif
#include "lwip/udp.h"
#include "lwip/tcp.h"
#include "xil_cache.h"
#include "xil_io.h"

#if LWIP_DHCP==1
#include "lwip/dhcp.h"
#endif

extern void codec_config();

extern int XAxiDma_SimplePollExample(void);

int config_fir_filter(int dec_fact, int dma_packet_size);

/* defined by each RAW mode application */
void print_app_header();
int start_application();
int transfer_data();
void tcp_fasttmr(void);
void tcp_slowtmr(void);

/* missing declaration in lwIP */
void lwip_init();

#if LWIP_DHCP==1
extern volatile int dhcp_timoutcntr;
err_t dhcp_start(struct netif *netif);
#endif

/* Az UDP struktúra */
extern struct udp_pcb *udp;

extern volatile int TcpFastTmrFlag;
extern volatile int TcpSlowTmrFlag;
static struct netif server_netif;
struct netif *echo_netif;

/* Debug célokra használt led */
int led = 0;

/* Késleltetés függvény a polling megvalósitására */
void delay() {
     for (int i = 0; i < 0x17FFFF; i++)
          ;
}

void print_ip(char *msg, struct ip_addr *ip) {
     print(msg);
     xil_printf("%d.%d.%d.%d\n\r", ip4_addr1(ip), ip4_addr2(ip), ip4_addr3(ip),
               ip4_addr4(ip));
}

void print_ip_settings(struct ip_addr *ip, struct ip_addr *mask,
          struct ip_addr *gw) {

     print_ip("Board IP: ", ip);
     print_ip("Netmask : ", mask);
     print_ip("Gateway : ", gw);
}

#if defined (__arm__) && !defined (ARMR5)
#if XPAR_GIGE_PCS_PMA_SGMII_CORE_PRESENT == 1 || XPAR_GIGE_PCS_PMA_1000BASEX_CORE_PRESENT == 1
int ProgramSi5324(void);
int ProgramSfpPhy(void);
#endif
#endif

#ifdef XPS_BOARD_ZCU102
#ifdef XPAR_XIICPS_0_DEVICE_ID
int IicPhyReset(void);
#endif
#endif

int main() {
     struct ip_addr ipaddr, netmask, gw;

     /* MAC cím átírása */
     unsigned char mac_ethernet_address[] =
               { 0x00, 0xFa, 0x35, 0x00, 0x01, 0x02 };
```

```
    echo_netif = &server_netif;
#if defined (__arm__) && !defined (ARMR5)
#if XPAR_GIGE_PCS_PMA_SGMII_CORE_PRESENT == 1 || XPAR_GIGE_PCS_PMA_1000BASEX_CORE_PRESENT == 1
    ProgramSi5324();
    ProgramSfpPhy();
#endif
#endif

    /* Define this board specific macro in order perform PHY reset on ZCU102 */
#ifdef XPS_BOARD_ZCU102
    IicPhyReset();
#endif

    init_platform();

#if LWIP_DHCP==1
    ipaddr.addr = 0;
    gw.addr = 0;
    netmask.addr = 0;
#else
    /* Az IP cím beállítása */
    IP4_ADDR(&ipaddr, 192, 168, 1, 31);
    IP4_ADDR(&netmask, 255, 255, 255, 0);
    IP4_ADDR(&gw, 192, 168, 1, 1);
#endif


    lwip_init();

    /* Add network interface to the netif_list, and set it as default */
    if (!xemac_add(echo_netif, &ipaddr, &netmask, &gw, mac_ethernet_address,
    PLATFORM_EMAC_BASEADDR)) {
          xil_printf("Error adding N/W interface\n\r");
          return -1;
    }
    netif_set_default(echo_netif);
    platform_enable_interrupts();
    netif_set_up(echo_netif);
    print_ip_settings(&ipaddr, &netmask, &gw);

    start_application();
    /* Az inicializálás belülről kifelé haladva. A DMA inicializálása
    minden adaátvitelkor megtörténik, mert nem sikerült a megszakításos
    vezérléssel használni. */
    config_fir_filter(1, 4096);
    codec_config();

    /* I2S engedélyezése */
    Xil_Out32(XPAR_AXI_GPIO_I2S_EN_BASEADDR + 4, 0x00);
    Xil_Out32(XPAR_AXI_GPIO_I2S_EN_BASEADDR, 0xFF);

    /* Adatok vétele és feldolgozása: Lesznek elhagyott minták */
    while (1) {
          if (TcpFastTmrFlag) {
              tcp_fasttmr();
              TcpFastTmrFlag = 0;
          }
          if (TcpSlowTmrFlag) {
              tcp_slowtmr();
              TcpSlowTmrFlag = 0;
          }
          xemacif_input(echo_netif);
          /* DMA adatátvitel */
          XAxiDma_SimplePollExample();
          /* Várakozás: A DMA adatátvitel után nincs várakozás az átvitel végéig*/
          delay();
          /* UDP adatátvitel */
          transfer_data();
          led_toggle();
    }

    /* Nem hívódhat meg */
    cleanup_platform();

    return 0;
}

/* Debug célt szolgáló lámpa kapcsolás */
void led_toggle() {
    led = ~led;
    Xil_Out32(XPAR_AXI_GPIO_LED_BASEADDR, led);
}
```

A teszteléshez az Audacity nevű programmal generált 440 Hz-es szinuszos jelet vezettünk a kártya audió bemeneti portjátra, és a tanszéki portálról letöltött UDP kliens segítségével ábrázoltuk a hullámformát:

A hullámformán látszik, hogy vannak elhagyott minták, mivel az UDP küldési sebességet kisebbre állítottuk, mint a DMA sebességét [delay() függvény használata miatt], amit pedig a kodek sebessége határoz meg, de az egy UDP-csomaghoz tartozó minták alapján látjuk, hogy jól működik a hardver, megjelenik a szinuszos jelalak.