Laboratorul 2

Protocolul de comunicare AXI Stream

2.1 Introducere

Un System-on-Chip (SoC) este un sistem de calcul complex, realizat sub forma unui circuit integrat unic, care reunește pe aceeași pastilă o varietate de module specializate. Aceste module pot fi procesoare de performanțe diferite (de exemplu, un CPU de înaltă performanță și unul de joasă performanță), procesoare specializate (precum Digital Signal Processor pentru prelucrarea semnalelor), acceleratoare hardware (precum FPGA sau GPU), memorie partajată pentru schimbul de date între componente, controlere (de memorie DRAM, video etc.) și o gamă largă de periferice integrate (UART, USB, Ethernet, ADC/DAC).

In Figura 2.1 se ilustrează tocmai această diversitate de blocuri funcționale reunite într-un singur cip, care colaborează prin interconexiuni interne și memorie partajată. Astfel, un SoC poate fi privit ca un "mic computer" complet, integrat într-un singur circuit, capabil să îndeplinească sarcini complexe într-un mod eficient ca performanță, consum și dimensiune.

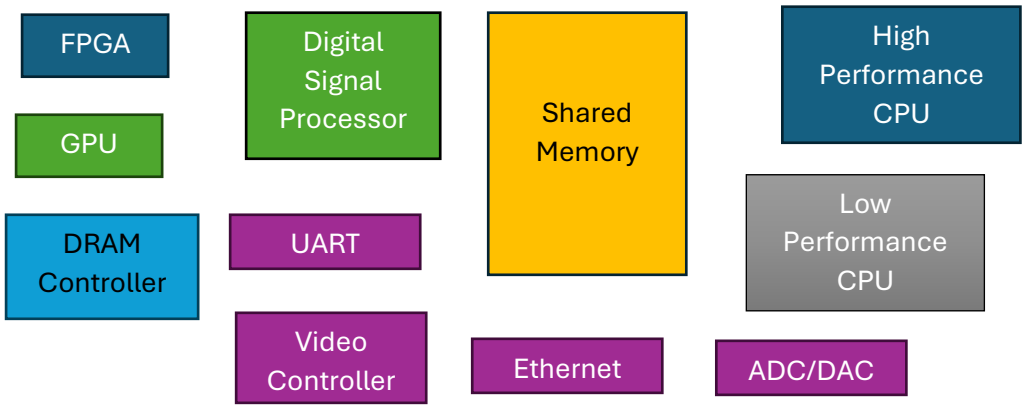

Figura 2.1. Elemente care se pot gasi pe SoC-urile moderne

În arhitecturile moderne ale sistemelor embedded și SoC, transferul eficient de date între module devine o cerință esențială. Asadar o serie de firme au dezvoltat astfel de busuri prin care se comunica eficient intre module. Printre acestea se numara IBM Core-Connect, WishBone (folosit de OpenCore IPs), AXI etc.

Protocolul AXI-Stream (Advanced eXtensible Interface - Streaming) face parte din familia de interfețe AMBA (Advanced Microcontroller Bus Architecture) și a fost proiectat de firma ARM special pentru a facilita transmisia de fluxuri continue de date cu o latență minimă și un debit ridicat. Spre deosebire de interfețele tradiționale, care gestionează adrese de memorie și control complex, AXI-Stream adoptă o abordare minimalistă, în care datele sunt livrate într-un flux direct, fără a necesita informații de adresare sau control pentru fiecare transfer individual.

Utilitatea protocolului AXI-Stream este evidentă în aplicații care necesită transmisii de date secvențiale și rapide, cum ar fi procesarea video și audio, comunicații de mare viteză (Ethernet, PCIe), interfețe de senzori, transmisii RF sau accelerare hardware pentru procesare paralelă (cum ar fi rețele neurale pe FPGA-uri). În astfel de scenarii, datele pot fi transmise în mod eficient de la un modul sursă (Transmitter) către un modul destinație (Receiver), folosind o interfață punct-la-punct care minimizează complexitatea logicii de arbitraj.

Este important de subliniat că AXI-Stream nu este „același lucru” cu AXI4 sau AXI4-Lite, chiar dacă toate fac parte din aceeași familie AMBA. AXI4 este folosit în principal pentru acces la

memorie (de exemplu, atunci când un procesor sau un modul hardware citeşte şi scrie blocuri mari de date în RAM), iar AXI4-Lite este o versiune simplificată, utilizată mai ales pentru configurarea şi controlul registrelor interne ale unui modul hardware (de exemplu, setarea parametrilor, încărcarea coeficienţilor sau pornirea/oprirea unui accelerator). În schimb, AXI-Stream reprezintă un alt tip de interfaţă AXI, gândită special pentru transmiterea continuă de date, ca un „flux" care curge de la un emiţător la un receptor, fără a mai fi nevoie de adrese de memorie. Mecanismul de control al fluxului prin perechea *TVALID–TREADY* introduce conceptul de *backpressure*, permiţând receptorului să încetinească debitul transmis fără pierderi, ceea ce asigură stabilitatea în sisteme complexe. În plus, AXI-Stream este ideal pentru a construi arhitecturi scalabile de tip pipeline, în care datele trec prin mai multe etape de procesare hardware, fiecare modul fiind conectat prin aceeaşi interfaţă standard. Fiind parte din standardul AMBA, interfaţa oferă portabilitate şi compatibilitate ridicată, facilitând reutilizarea modulelor pe diverse platforme. Printre aplicaţiile practice se numără streaming video (HDMI, DisplayPort), comunicaţii de mare viteză, achiziţie de date de la senzori LIDAR sau RADAR, precum şi acceleratoare hardware pentru reţele neurale, unde fluxurile de date continue trebuie manipulate cu latenţă minimă.

2.2 Semnalele Interfeţei de Comunicare

Protocolul defineşte un set restrâns, de semnale. Cele mai relevante dintre acestea sunt prezentate in Tabelul II.1 mai jos.

Tabelul II.1. Semnalele esenţiale din interfaţa AXI

Semnalul	Sursa	Latimea (nr de biti)	Descrierea
ACLK	Clock	1	Un semnal de ceas global. Toate semnalele sunt eşantionate pe frontul crescător(atunci când ceasul trece din 0 în 1) al lui ACLK.
ARESETn	Reset	1	Un semnal global de reset
TVALID	Transmiţătorul	1	Indică faptul că Transmitter-ul oferă date valide
TREADY	Receptorul	1	Semnalizat de Receiver pentru a arăta că este pregătit să accepte datele
TDATA	Transmiţătorul	TDATA_WIDTH	Reprezintă câmpul principal de date dintr-o interfaţă AXI4-Stream. Prin acest semnal sunt transmise efectiv informaţiile utile între Transmitter şi Receiver, spre deosebire de alte semnale de control (precum TVALID sau TREADY) care doar coordonează transferul. Lăţimea semnalului, denumită TDATA_WIDTH, trebuie să fie un multiplu întreg de octeţi (8 biţi) şi, conform recomandărilor, valorile uzuale sunt: 8, 16, 32, 64, 128, 256, 512 sau 1024 biţi. Această flexibilitate permite adaptarea interfeţei la necesarul de lăţime de bandă al aplicaţiei.
TLAST	Transmiţătorul	1	Semnalează sfârşitul unui pachet (util în cazul transmisiilor pe cadre sau secvenţe).

TSTRB	Transmiţătorul	TDATA_WIDTH/8	Semnalul TSTRB (strobe) are câte un bit pentru fiecare octet din TDATA. Fiecare bit din TSTRB arată dacă octetul asociat conţine date valide (şi trebuie procesat), caz in care acel bit va avea valoarea 1, sau doar ocupă o poziţie (padding, nu trebuie procesat), caz in care are valoarea 0. În acest fel, interfaţa poate transmite pachete de date de lungimi variabile fără a irosi lăţime de bandă.
TKEEP	Transmiţătorul	TDATA_WIDTH/8	Câţi octeţi din pachet sunt valizi. Spune dacă octetul respectiv face parte din fluxul de date util (de exemplu, la sfârşitul unui pachet unde nu se folosesc toţi octeţii, mai sunt si octeţi de padding)

Mai multe detalii legate despre semnalele AXI Stream si semnificatia si modul acestora de funcţionare poate fi găsit in capitolul 2 al referinţei [1].

Conectarea modulelor AXI-Stream se face, în general, în mod direct, punct-la-punct. Cu toate acestea, pot exista interconectări mai complexe, folosind multiplexoare, demultiplexoare, comutatoare sau arbitri pentru a permite scalabilitate şi flexibilitate. De asemenea, pentru a rezolva problema diferenţelor de viteză între emiţător şi receptor, sau pentru a decupla temporar sincronizarea între componente ce operează în ceasuri diferite (clock domains), se utilizează adesea FIFO-uri compatibile AXI-Stream. Acestea permit stocarea temporară a datelor şi asigură continuitatea fluxului fără pierderi sau blocaje.

Necesitatea utilizării unui FIFO apare şi în contexte de pipelining profund sau în acceleratoare hardware unde se prelucrează volume mari de date, iar o variaţie în disponibilitatea resurselor (de exemplu, procesorul este ocupat sau interfaţa de ieşire este blocată) nu trebuie să întrerupă fluxul de date. FIFO-ul asigură o zonă tampon ce permite debitul susţinut chiar şi în prezenţa unor întârzieri temporare.

Protocolul AXI4-Stream este un protocol de tip **Transmitter–Receiver** (sau **Source–Sink**). Transmitter-ul (sursa) produce şi transmite date, iar Receiver-ul (destinaţia) le primeşte şi consumă. O tranzacţie în AXI4-Stream se referă la transferul de date pe un canal unidirecţional, de la un modul hardware (sursă/Transmitter) către un alt modul hardware (destinaţie/Receiver). Transferul este controlat prin mecanismul de handshake: sursa indică faptul că datele sunt valide prin semnalul **TVALID**, iar destinaţia confirmă disponibilitatea de a le primi prin semnalul **TREADY**. Datele sunt transferate numai atunci când ambele semnale sunt active simultan.

În acest protocol nu există operaţii de tip read sau write şi nici adresare, ca în AXI4 clasic; există doar fluxul de date în sensul sursă - destinaţie.

2.3 Mecanisme de Handshacking

Handshaking-ul este un mecanism prin care două componente hardware se pun de acord dacă şi când pot transfera date. Transferul se face doar atunci când amândouă componentele implicate in comunicare sunt de acord ca sa se realizeze comunicarea, lucru care face ca atât Transmitatorul cat si receptorul sa controleze rata cu care se realizeaza comunicarea. Spre deosebire de interfeţe precum SPI sau I^2C, unde masterul decide rata de transfer, în AXI4-Stream atât sursa cât şi destinaţia pot controla viteza prin handshaking (TVALID şi TREADY).

Regula generala a transferului de date este aceea ca datele sunt transferate de la transmiţător la receptor pe primul front crescator al ceasului ACLK, in care TVALID a fost setat pe HIGH de catre transmitator si TREADY a fost setat pe HIGH de catre receptor. Aceste semnale pot fi setate

pe „1” logic fie deodată, sau pe rând, fiecare semnal putând sa fie setat primul. Exista, totuşi, o serie de reguli care trebuie respectate de modulele care comunica pe AXI-Stream, si anume:

- Odată ce Transmitter-ul activează TVALID, el trebuie menţinut activ până când handshake-ul are loc (adică până când TREADY devine HIGH şi apare frontul crescător al lui ACLK).
- Transmitter-ul nu are voie să aştepte TREADY pentru a activa TVALID. El trebuie să semnalizeze datele valide chiar dacă Receiver-ul nu este pregătit.
- Receiver-ul are voie să aştepte ca TVALID să fie activ înainte de a activa TREADY. Cu alte cuvinte, poate decide să nu-şi declare disponibilitatea până nu există date valide.
- Dacă Receiver-ul activează TREADY înainte ca TVALID să fie activ, el poate ulterior să dezactiveze TREADY şi să aştepte până când Transmitter-ul ridică TVALID.

In exemplele de mai jos, preluate din documentaţia oficiala [1] a protocolului de transmisie se pot observa cronogramele corespunzătoare pentru cele trei scenarii in care se poate realiza handshakingul.

2.3.1 Handshake cu TVALID activat înainte de TREADY

In acest scenariu Transmitter-ul (sursa) pune pe magistrală informaţiile de date şi control, şi ridică semnalul TVALID la „1” logic (HIGH). Odată activat acest semnal trebuie menţinut activ si informaţia transmisa nu se modifica pana când handshake-ul nu are loc. Receiverul poate accepta datele doar după ce activează semnalul TREADY (îl face HIGH). Transferul efectiv al datelor se realizează pe primul front crescător a lui ACLK in care ambele semnale TVALID si TREADY sunt HIGH. Acest lucru se poate observa in Figura 2.2 (preluata din [1]) in etapa T3.

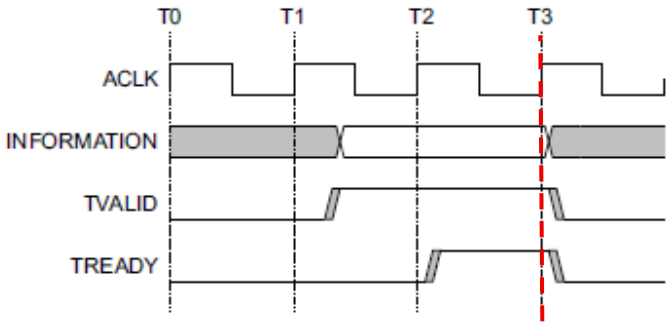

Figura 2.2 Procesul de handshaking cu semnalul TVALID activat înainte de TREADY [1]

2.3.2 Handshake cu TREADY activat înainte de TVALID

În acest scenariu Receiver-ul (destinaţia) activează (pune pe HIGH) TREADY înainte ca Transmitter-ul (sursa) să declare datele ca fiind valide. Acest lucru arată că Receiver-ul este deja pregătit să accepte date şi informaţii de control, chiar dacă ele nu sunt încă valide.

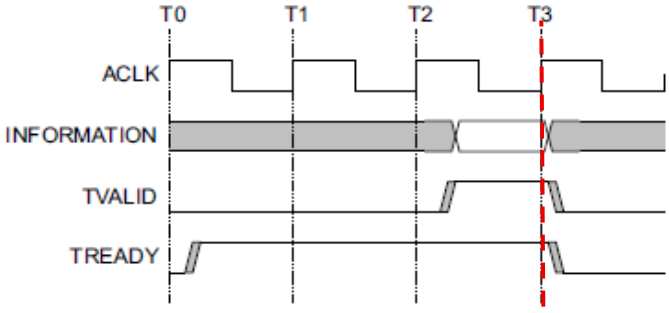

Figura 2.3 Procesul de handshaking cu semnalul TREADY activat înainte de TVALID [1]

Transferul efectiv are loc imediat ce Transmitter-ul setează TVALID pe „1” logic, deoarece în acel moment ambele semnale sunt active. Si in acest caz transferul se realizează pe primul front ascendent de ceas in care TVALID si TREADY sunt „1”. Acest scenariu poate fi observat in Figura 2.3 (preluata din [1]) in etapa T3.

2.3.3 Handshake cu TVALID și TREADY activate simultan

În acest scenariu Transmitter-ul (sursa) ridică TVALID in „1” logic în același ciclu de ceas în care Receiver-ul (destinația) activează TREADY (îl pune pe HIGH). Transferul datelor se realizează in același ciclu de ceas, imediat, întrucât ambele semnale sunt active simultan. In Figura 2.4. preluata din [1] transferul are loc in T2.

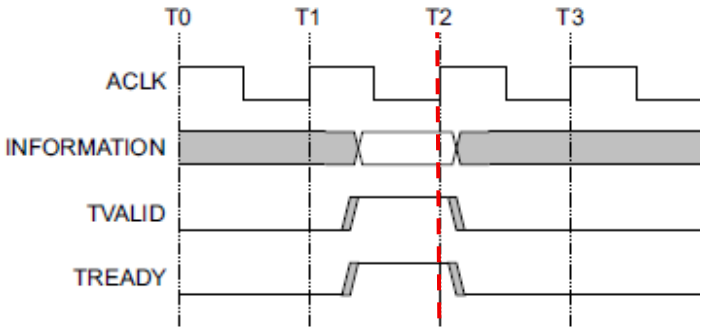

Figura 2.4 Procesul de handshaking cu semnalul TREADY si TVALID activate simultan [1]

2.4 Crearea unei aplicații in Vivado care utilizează AXI-Stream

Pentru a înțelege modul de funcționare al protocolului AXI4-Stream, vom realiza o aplicație demonstrativă în care fluxul de date este controlat pas cu pas de utilizator prin intermediul unor butoane. Transmitterul citește valori dintr-o memorie ROM internă și, la fiecare apăsare a butonului START, pune exact un singur element pe magistrala AXI4-Stream. Acest element intră într-un modul FIFO, care acționează ca tampon între transmitter și receiver și asigură decuplarea dintre viteza de scriere (transmitere) și viteza de citire (recepție). Receiverul extrage datele din FIFO numai atunci când utilizatorul apasă butonul STEP, deoarece semnalul TREADY al slave-ului este comandat direct de acest buton. La fiecare element scos din FIFO, receiverul incrementează valoarea transmisă cu unu (modulo 16) și afișează rezultatul pe LED-urile plăcii. Astfel, prin apăsarea repetată a butonului START se pot încărca mai multe valori în FIFO, iar prin apăsarea butonului STEP ele sunt scoase, unul câte unul, și vizualizate pe LED-uri.

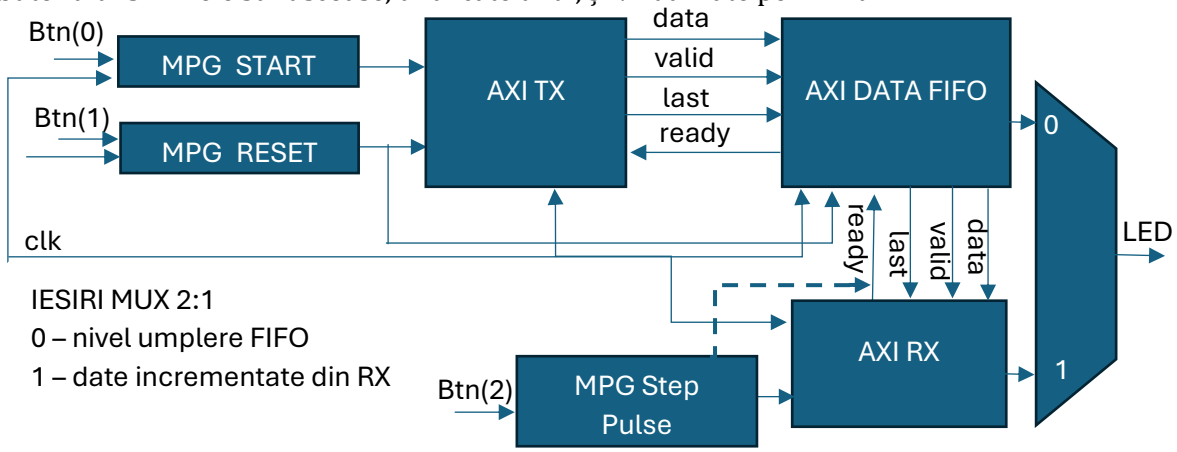

Figura 2.5 Diagrama proiectului descris in aceasta secțiune

In Figura 2.5 este ilustrata diagrama hardware a proiectului pe care îl vom implementa pentru a ilustra modul de funcționare al interfeței AXI-Stream.

2.4.1 Transmiterea AXI

Atat crearea modulelor de generare al pulsului unic MPG cat si conceptele teoretice din spatele acestui concept au fost acoperite in [2] si nu vor mai fi acoperite si in aceasta lucrare. Creati un proiect nou pentru placa Zybo asa cum a fost prezentat in laboratorul precedent si creati o sursa noua **axis_transmitter.vhd**. Codul pentru modulul de transmitere, este prezentat in caseta de text de mai jos. Ca si terminologie e important de mentionat ca un beat este un singur transfer de date realizat într-un ciclu de ceas atunci când semnalele de handshake (VALID şi READY) sunt active, iar un burst este o secvență de mai multe beats consecutive care formează o tranzacţie completă.

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

entity axis_transmitter is
 generic( ROM_DEPTH : natural := 16 );
 port (
  clk         : in  std_logic;
  rst_n       : in  std_logic;       -- reset activ LOW
  start       : in  std_logic;       -- impuls 1 tact (de la MPG)

  m_axis_tdata  : out std_logic_vector(7 downto 0);
  m_axis_tvalid : out std_logic;
  m_axis_tlast  : out std_logic;
  m_axis_tready : in  std_logic
 );
end entity;

architecture rtl of axis_transmitter is
 -- ROM cu valori pe 4 biti (trimitem pe 8 biti, nibble sus = 0)
 type rom_t is array(0 to 15) of std_logic_vector(3 downto 0);
 constant MEM_ROM : rom_t := (
  0=>x"0",1=>x"1",2=>x"3",3=>x"7",
  4=>x"F",5=>x"2",6=>x"4",7=>x"8",
  8=>x"5",9=>x"A",10=>x"0",11=>x"6",
  12=>x"9",13=>x"C",14=>x"D",15=>x"E"
 );
 -- index în ROM (incrementăm DUPA handshake)
 signal idx    : integer range 0 to ROM_DEPTH-1 := 0;
 -- registre AXI
 signal tdata_r  : std_logic_vector(7 downto 0) := (others=>'0');
 signal tvalid_r : std_logic := '0';
 signal tlast_r  : std_logic := '0';
begin
 m_axis_tdata  <= tdata_r;
 m_axis_tvalid <= tvalid_r;
 m_axis_tlast  <= tlast_r;
```

```
process(clk)
 begin
  if rising_edge(clk) then
   if rst_n = '0' then
    idx     <= 0;
    tdata_r  <= (others=>'0');
    tvalid_r <= '0';
    tlast_r  <= '0';
   else
    -- PORNIRE: la fiecare START, daca nu avem un beat în asteptare,
    -- încarcam UN singur beat si setam TVALID=1, TLAST=1
    if (start = '1') and (tvalid_r = '0') then
     tdata_r  <= (7 downto 4 => '0') & MEM_ROM(idx);
     tvalid_r <= '1';
     tlast_r  <= '1';         -- pachet de 1 beat
    end if;
    -- HANDSHAKE: când FIFO (sau RX) e gata, consumam beat-ul
    if (tvalid_r = '1') and (m_axis_tready = '1') then
     tvalid_r <= '0';
     tlast_r  <= '0';
     -- trecem la urmatoarea valoare din ROM (wrap)
     if idx = ROM_DEPTH-1 then
      idx <= 0;
     else
      idx <= idx + 1;
     end if;
    end if;
   end if;
  end if;
 end process;
end architecture;
```

2.4.2 Data FIFO

Pentru a include FIFO-ul in diagrama sistemului descris pentru a comunica prin intermediul AXI vom folosi un modul IP preexistent. Pentru a adăugat un astfel de modul navigaţi in **Flow Navigator->Project Manager** si daţi **click pe IP Catalog** ca si in Figura 2.6.

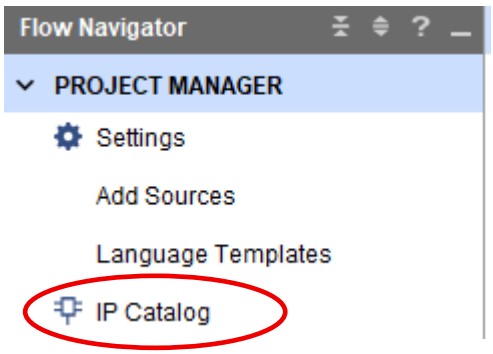

Figura 2.6. Adăugarea unui IP existent.

După ce apăsați pe butonul de IP Catalog, va apărea o fereastra ca si in Figura 2.7, iar in bara de search scrieți AXI DATA FIFO, dați dublu click stânga pe AXI4-Stream Data FIFO.

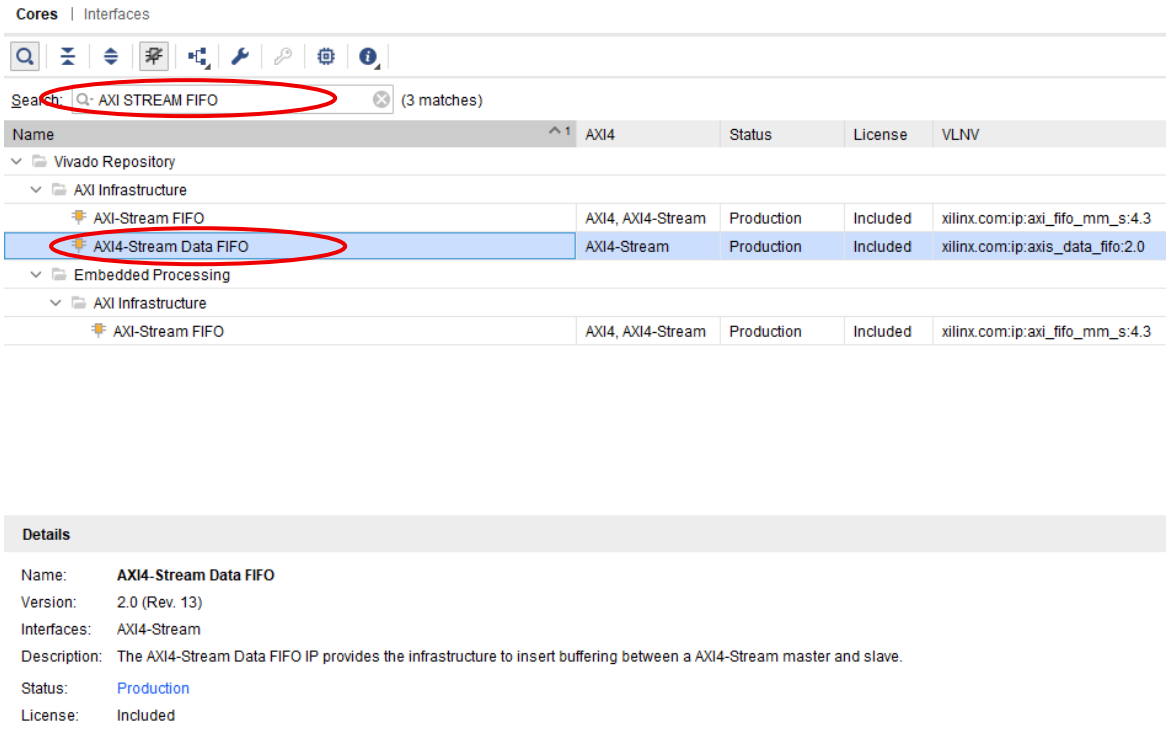

Figura 2.7. Selectarea AXI Stream FIFO

Odată ce ați dat dublu click va apărea o fereastra de configurare, in care va trebui sa completam caracteristicile AXI-FIFO-ul nostru. Setați pentru tabul general opțiunile ca si in Figura 2.8.

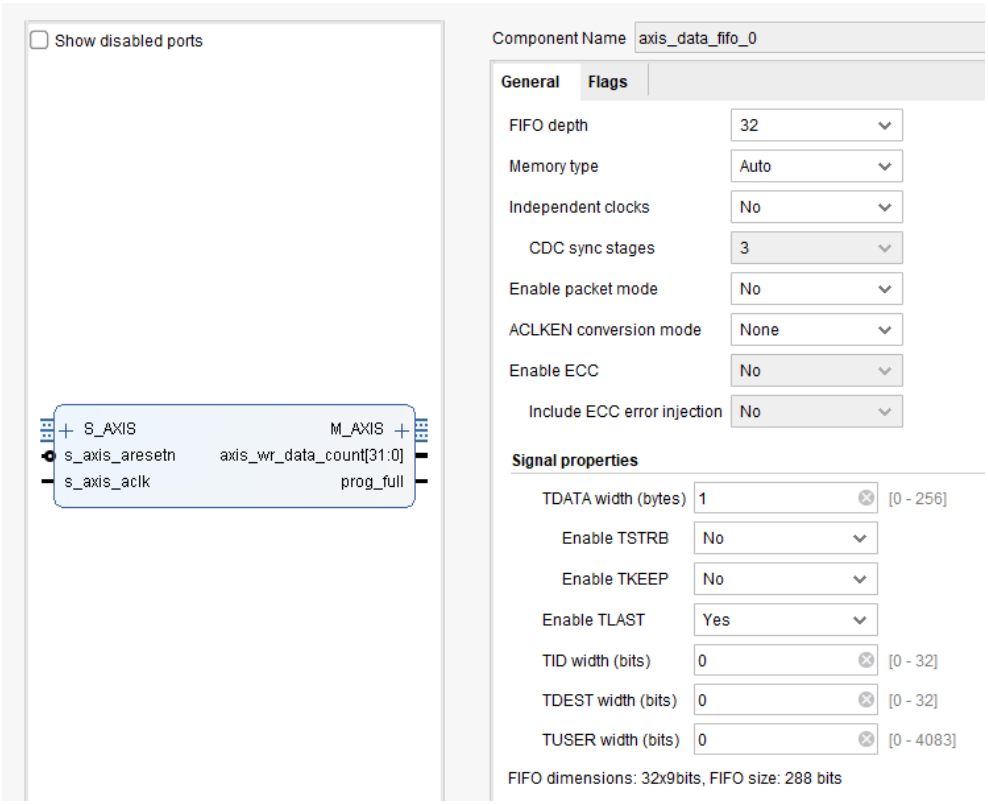

Figura 2.8. Setările din tabul general al memoriei AXI4-Stream Data FIFO.

In acest proiect didactic pe care îl construim, nu vom avea clock-uri independente pentru transmițător si receptor. De asemenea, întrucât in unul din exercițiile propuse la finalul acestei lucrări va trebui sa număram cate pachete sunt transmise intre cei doi care comunica, vom activa flagul de TLAST. Mai mult pentru a vedea nivelul de umplere al FIFO-ului in tabul Flags setați opțiunile ca si in Figura 2.9.

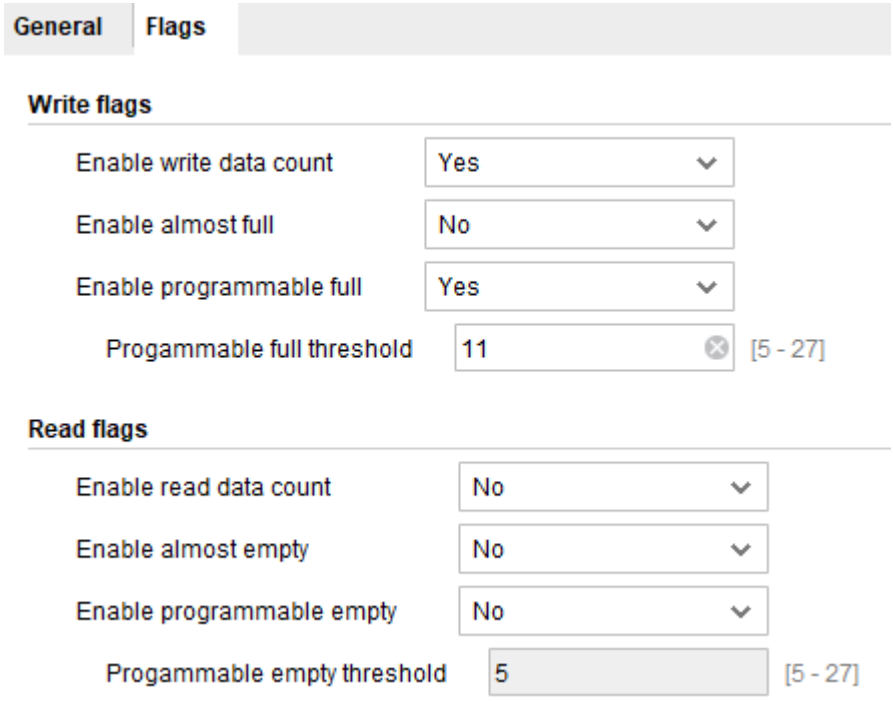

Figura 2.9. Setările memoriei FIFO din tabul Flags

Odată realizate setările ca in Figura 2.9 apăsați pe butonul **OK**, iar apoi in noua fereastra care apare, ca si cea din Figura 2.10, apăsați butonul **Generate** pentru a va genera memoria FIFO.

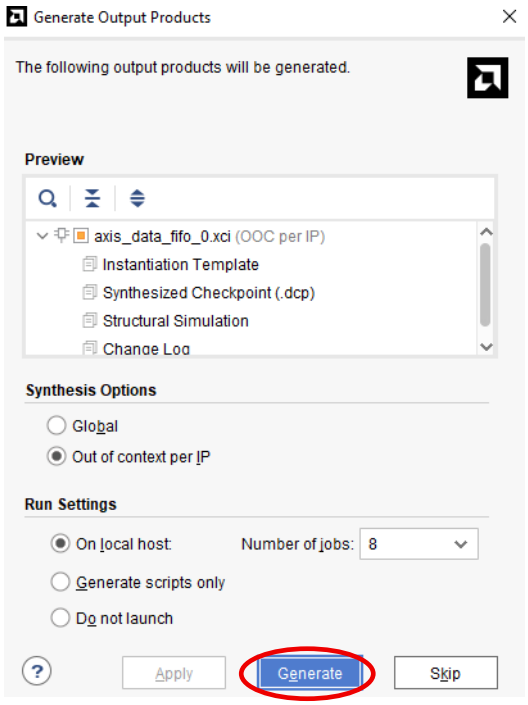

Figura 2.10. Generarea memoriei FIFO.

2.4.3 Recepția AXI

In modulul de recepție, se vor primi date de la FIFO, se va incrementa acea valoare si se va trimite spre afișare pe ledurile plăcii. Semnalul de ready de la modulul de recepție așa cum a fost explicat si la începutul acestui capitol in Figura 2.5. se va controla de la un buton, pentru a putea observa mai eficient cum se face handshakingul si pentru urmări mai ușor fluxul de date. Adăugați o sursa noua vhdl la proiectul vostru si numiți acest fișier **axis_receiver.** Codul pentru receiver este prezentat in caseta de cod de mai jos.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity axis_receiver is
port(
   clk         : in  std_logic;
   rst_n       : in  std_logic;         -- reset activ LOW
   -- control backpressure
   ready_en     : in  std_logic;  -- '1' = accepta date (TREADY=1), '0' = blocheaza (TREADY=0)
   -- AXI4-Stream Slave (Sink)
   s_axis_tdata  : in  std_logic_vector(7 downto 0);
   s_axis_tvalid : in  std_logic;
   s_axis_tlast  : in  std_logic;   --pt exercitiile propuse
   s_axis_tready : out std_logic;
   -- ieșire catre placa
   leds        : out std_logic_vector(3 downto 0)
 );
end axis_receiver;
architecture Behavioral of axis_receiver is
   signal leds_r : std_logic_vector(3 downto 0) := (others => '0');
begin
 -- backpressure simplu: TREADY urmează ready_en
 s_axis_tready <= ready_en;
  process(clk)
 begin
   if rising_edge(clk) then
    if rst_n = '0' then
      leds_r <= (others => '0');
    else
     -- Lasam transferul să aiba loc DOAR la handshake: TVALID=1 și TREADY=1
     if (s_axis_tvalid = '1' and ready_en = '1') then
      -- luam nibble-ul jos si adunam 1; aritmetica pe 4 biti face wrap (mod 16)
       leds_r <= std_logic_vector(unsigned(s_axis_tdata(3 downto 0)) + 1);
     end if;
    end if;
   end if;
 end process;
 leds <= leds_r;
end Behavioral;
```

2.4.4. Interconectarea modulelor implementate in fişierul TOP

Atât fişierele create anterior cat si generatoarele de monopuls vor fi interconectate in fişierul top.vhd. Creaţi un nou fişier de tip vhdl cu numele top si introduceţi codul din caseta de cod de mai jos pentru a lega toate componentele.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity top is
 port(
  clk : in  std_logic;
  led : out std_logic_vector(3 downto 0);
  sw  : in  std_logic_vector(3 downto 0);
  btn : in  std_logic_vector(3 downto 0)
 );
end top;
architecture Behavioral of top is
 component MPG is
 generic(
  CLK_HZ     : integer := 125_000_000; -- ZYBO Z7: 125 MHz
  DEBOUNCE_MS : integer := 10
 );
 port(
  enable : out std_logic;  -- impuls de 1 tact la fiecare apasare
  btn    : in  std_logic;  -- buton (active-HIGH pe ZYBO/Nexys/Basys)
  clk    : in  std_logic   -- clock
 );
 end component;
component axis_data_fifo_0 is
 port (
  s_axis_aresetn     : in  std_logic;
  s_axis_aclk        : in  std_logic;

  s_axis_tvalid      : in  std_logic;
  s_axis_tready      : out std_logic;
  s_axis_tdata       : in  std_logic_vector(7 downto 0);
  s_axis_tlast       : in  std_logic;

  m_axis_tvalid      : out std_logic;
  m_axis_tready      : in  std_logic;
  m_axis_tdata       : out std_logic_vector(7 downto 0);
  m_axis_tlast       : out std_logic;

  axis_wr_data_count  : out std_logic_vector(31 downto 0);
  prog_full          : out std_logic
 );
end component;
```

```
  ------------------------------------------------------------------------------------------------------------------
  -- Semnale AXI: TX -> FIFO (S side)
  ------------------------------------------------------------------------------------------------------------------
  signal s_tdata  : std_logic_vector(7 downto 0);
  signal s_tvalid : std_logic;
  signal s_tready : std_logic;
  signal s_tlast  : std_logic;
  ------------------------------------------------------------------------------------------------------------------
  -- Semnale AXI: FIFO -> RX (M side)
  ------------------------------------------------------------------------------------------------------------------
  signal m_tdata  : std_logic_vector(7 downto 0);
  signal m_tvalid : std_logic;
  signal m_tready : std_logic;
  signal m_tlast  : std_logic;
  ------------------------------------------------------------------------------------------------------------------
  -- Status FIFO
  ------------------------------------------------------------------------------------------------------------------
  signal fifo_wr_count : std_logic_vector(31 downto 0);
  signal fifo_prog_full: std_logic;
  ------------------------------------------------------------------------------------------------------------------
  -- LED-uri
  ------------------------------------------------------------------------------------------------------------------
  signal leds_data  : std_logic_vector(3 downto 0);--elementul de la indicele curent din FIFO
  signal leds_level : std_logic_vector(3 downto 0); --cate elemente mai avem in FIFO
  ------------------------------------------------------------------------------------------------------------------
  -- Control butoane
  ------------------------------------------------------------------------------------------------------------------
  signal start_pulse : std_logic;  -- un beat de pachet la transmitere (btn0)
  signal rst_pulse   : std_logic;  -- puls MPG pe reset
  signal rst_n       : std_logic;  -- reset activ LOW
  signal step_pulse  : std_logic;  -- TREADY un ciclu
begin
------------------------------------------------------------------------------------------------------------------
  -- MPG pentru START (btn0)
  ------------------------------------------------------------------------------------------------------------------
  mpg_start: MPG
    generic map (
    CLK_HZ     => 125_000_000,  -- ZYBO Z7
    DEBOUNCE_MS => 10
  )
    port map(
      enable    => start_pulse,
      btn => btn(0),
      clk => clk
    );
```

```
-----------------------------------------------------------------------------------------------------------------
-- MPG pentru RESET (btn1)
-----------------------------------------------------------------------------------------------------------------
 mpg_reset: MPG
 generic map (
  CLK_HZ     => 125_000_000,  -- ZYBO Z7
  DEBOUNCE_MS => 10
 )
  port map(
   enable    => rst_pulse,
   btn => btn(1),
   clk => clk
  );
 rst_n <= not rst_pulse;
-----------------------------------------------------------------------------------------------------------------
 -- MPG pentru STEP (btn2) -> TREADY un singur ciclu la fiecare apasare
-----------------------------------------------------------------------------------------------------------------
 mpg_step: MPG
  generic map (
  CLK_HZ     => 125_000_000,  -- ZYBO Z7
  DEBOUNCE_MS => 10
 )
  port map(
   enable    => step_pulse,
   btn => btn(2),
   clk => clk
  );
-----------------------------------------------------------------------------------------------------------------
 -- Transmitter (AXI-Stream Master)
-----------------------------------------------------------------------------------------------------------------
 u_tx: entity work.axis_transmitter
  generic map( ROM_DEPTH => 16 )
  port map(
   clk          => clk,
   rst_n        => rst_n,
   start        => start_pulse,
   m_axis_tdata  => s_tdata,
   m_axis_tvalid  => s_tvalid,
   m_axis_tlast   => s_tlast,
   m_axis_tready  => s_tready
  );
-----------------------------------------------------------------------------------------------------------------
 -- Receiver (AXI-Stream Slave)
 -- Backpressure "pas cu pas": m_tready = step_pulse (un beat/apasare btn2)
-----------------------------------------------------------------------------------------------------------------
 m_tready <= step_pulse;
```

```
u_rx: entity work.axis_receiver
  port map(
    clk         => clk,
    rst_n       => rst_n,
    ready_en    => m_tready,
    s_axis_tdata  => m_tdata,
    s_axis_tvalid => m_tvalid,
    s_axis_tlast  => m_tlast,
    s_axis_tready => open,
    leds        => leds_data
  );
-------------------------------------------------------------------------------------------------------------------
 -- FIFO (single clock)
 -------------------------------------------------------------------------------------------------------------------
u_fifo: axis_data_fifo_0
 port map(
  s_axis_aclk       => clk,
  s_axis_aresetn    => rst_n,

  s_axis_tvalid     => s_tvalid,
  s_axis_tready     => s_tready,
  s_axis_tdata      => s_tdata,
  s_axis_tlast      => s_tlast,

  m_axis_tvalid     => m_tvalid,
  m_axis_tready     => m_tready,
  m_axis_tdata      => m_tdata,
  m_axis_tlast      => m_tlast,

  axis_wr_data_count  => fifo_wr_count,
  prog_full         => fifo_prog_full
 );
-------------------------------------------------------------------------------------------------------------------
 -- Bargraph pentru nivelul FIFO
 -------------------------------------------------------------------------------------------------------------------
 process(fifo_wr_count)
  variable cnt : integer;
 begin
  cnt := to_integer(unsigned(fifo_wr_count(5 downto 0)));
  if cnt = 0 then
    leds_level <= "0000";
  elsif cnt < 8 then
    leds_level <= "0001";
  elsif cnt < 16 then
    leds_level <= "0011";
  elsif cnt < 24 then
    leds_level <= "0111";
```

```
else
    leds_level <= "1111";
  end if;
 end process;
 ----------------------------------------------------------------------------------------------------------
 led <= leds_level when sw(2)='1' else leds_data;
end Behavioral;
```

Pentru a adăuga MPG-ul, in **Flow Navigator**, navigaţi la **Project Manager**, iar apoi daţi click pe **Add Sources**. In fereastra care apare selectaţi opţiunea **Add or create design sources,** iar apoi apăsaţi butonul **Next**. In noua fereastra apăsaţi pe **Add Files** si navigaţi la locaţia unde aţi descărcat MPG-ul pus la dispoziţie in git-ul proiectului si daţi dublu click pe acesta pentru a-l include in proiectul vostru. Bifaţi si caseta **Copy sources into project** pentru a copia fişierul in proiectul vostru (nu doar referinţa la fişier), iar apoi apăsaţi butonul **Finish**. Aceasta opţiune e ilustrata in Figura 2.11

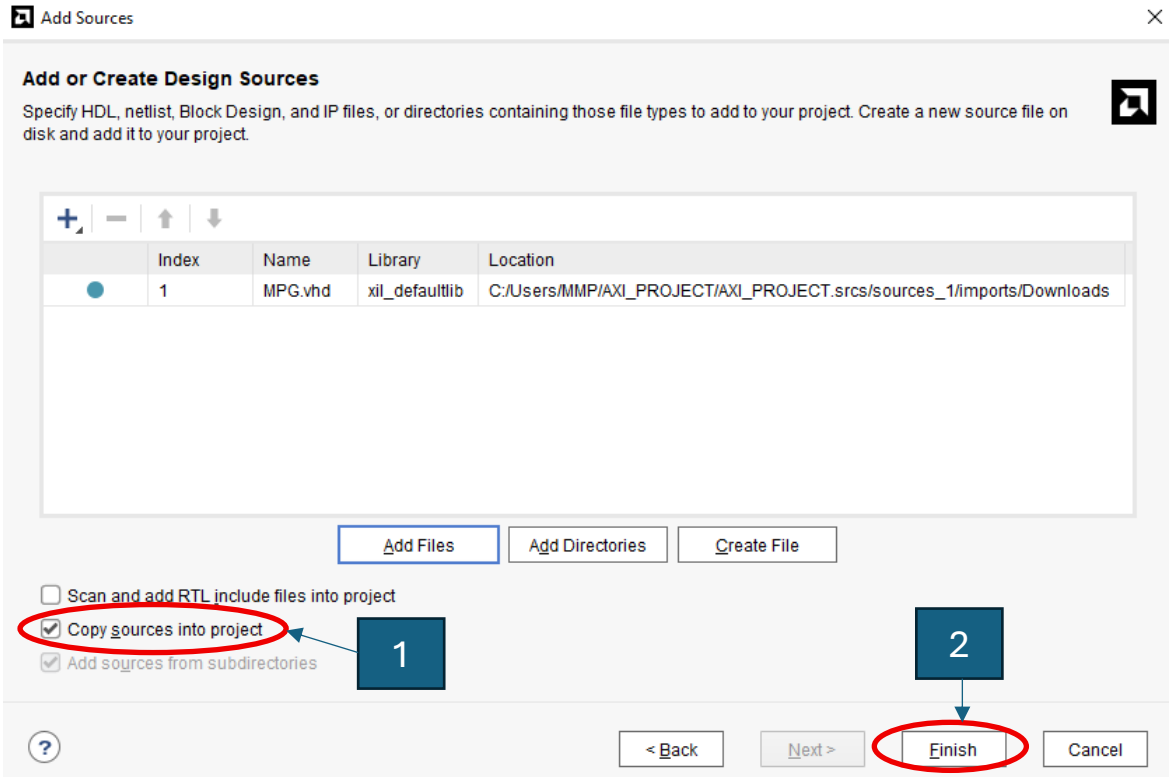

Figura 2.11. Adăugarea MPG-ului in proiect.

După adăugarea MPG-ului va trebui creat fişierul de constrângeri XDC pentru a putea face conectarea intre periferice si FPGA. Daţi click pe **Add Sources** din **Flow Navigator,** iar apoi selectaţi opţiunea **Add or create constraints.** In fereastra care apare daţi click pe **Create File** si introduceţi numele fişierului de constrângeri. In cazul nostru îl vom numi zybo.xdc fiindcă placa pe care lucram este placa zybo. In final apăsaţi pe butonul **Finish.** Dupa adaugarea fisierului de constrângeri, îl veţi vedea in secţiunea Window->Sources->Constraints->constrs_1(1). După ce aţi găsit fişierul, daţi click dreapta pe acesta si setaţi opţiunea **Set as Target Constraints File**. Daţi dublu click pe acest fişier, si introduceţi constrângerile prezentate in caseta de text de mai jos.

```
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { clk }];
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { clk }];
###Switches
set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [get_ports { sw[0] }];
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { sw[1] }];
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports { sw[2] }];
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { sw[3] }];
###Buttons
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { btn[0] }];
set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports { btn[1] }];
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { btn[2] }];
set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports { btn[3] }];
##LEDs
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { led[0] }];
set_property -dict { PACKAGE_PIN M15   IOSTANDARD LVCMOS33 } [get_ports { led[1] }];
set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { led[2] }];
set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { led[3] }];
```

In secţiunea **Program and Debug** din Flow Navigator apasati pe **Generate Bitstream** pentru a genera fişierul .bit. In fereastra care apare apăsaţi **Yes** apoi **Ok.**
După generarea fişierului .bit încărcaţi-l pe placa Zybo aşa cum a fost descris in laboratorul precedent.

Pentru a testa modulul AXI, apăsăm butonul 0 pentru a introduce în FIFO o valoare din ROM-ul transmitterului. Această valoare este stocată temporar în FIFO şi va putea fi transmisă mai departe doar atunci când apăsăm butonul 2, care acţionează ca semnal TREADY şi permite transferul către receiver. Receiverul preia valoarea, o incrementează cu unu modulo 16 şi afişează rezultatul pe LED-uri. În orice moment, butonul 1 poate fi apăsat pentru a reseta sistemul, ştergând atât conţinutul FIFO, cât şi revenind transmitterul şi receiverul în starea iniţială. În plus, prin comutatorul SW(2) putem selecta ce anume se afişează pe LED-uri: fie rezultatele incrementate din receiver, fie nivelul de umplere al FIFO sub forma unui bargraph.

Un mic tabel care ilustrează ce vedem pe leduri pentru fiecare valoare trimisa de transmitter e prezentat mai jos.

Tabelul II.2. Corelaţia dintre datele transmise pe AXI4-Stream şi rezultatele afişate pe LED-uri

Nr. Element	Intrare	Iesire	Nr. Element	Intrare	Iesire
1	0000	0001	9	0101	0110
2	0001	0010	10	1010	1011
3	0011	0100	11	0000	0001
4	0111	1000	12	0110	0111
5	1111	0000	13	1001	1010
6	0010	0011	14	1100	1101
7	0100	0101	15	1101	1110
8	1000	1001	16	1110	1111

La început, indiferent daca switchul sw2 e pe ‚1' sau pe ‚0' nu se va vedea nimic pe leduri. Daca mutaţi switchul 1 si începeţi sa apăsaţi pe butonul btn0 veţi observa cum încep ledurile sa se aprindă in funcţie de nivelul de umplere al FIFO-ului. Apăsaţi pe butonul de start pana vedeţi primele 3 leduri aprinse. Mutaţi apoi switchnul sw2 pe ‚0' logic. Se va observa ca la început ledurile sunt stinse. Pe măsura ce apăsam pe butonul 2 vom observa pe leduri elementele din Tabelul II.2. In Figura 2.12 sunt ilustrate 2 imagini cu placa Zybo cate una pentru fiecare scenariu.

De menţionat este faptul ca outputul prezentat in Figura 2.12 poate fi obţinut pe orice placa care are cipul Zynq.

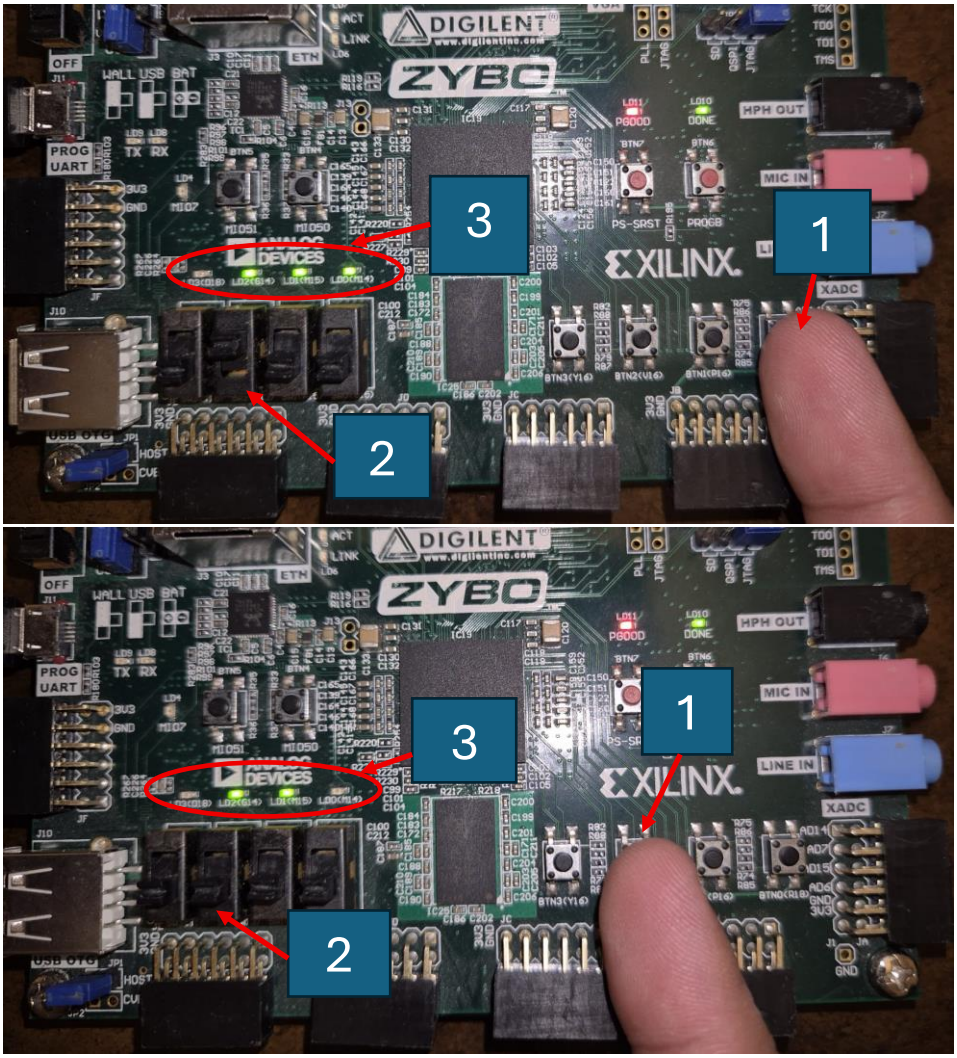

Figura 2.12. Testarea aplicației implementate

Tutorialul cu sursele complete se găseşte la linkul de git:
https://github.com/mirceamp/StructuraSistemelorDeCalcul/tree/main/Lab2

Exerciţii

1. Implementaţi si testaţi pe plăcuta exemplul din laborator si încercaţi sa înţelegeţi conceptele prezentate.
2. Pentru a vedea daca aţi înţeles conceptele din laborator încercaţi sa răspundeţi la întrebările de mai jos:
    a. Care este rolul unui FIFO între două blocuri AXI? De ce nu am conecta transmitter-ul direct la receiver?
    b. Explicați, pe baza diagramei bloc, cine produce semnalul TVALID, cine produce TREADY şi în ce condiţii are loc transferul efectiv al datelor.
3. Modificaţi aplicația astfel încât daca switchul 3 este pe ‚1' logic sa se afişeze valorile primite prin axi (fără a aduna 1 si apoi modulo 16). Daca switchul 3 este pus pe ‚0' logic se va lua in considerare logica actuala realizata pentru switchul 2.

4. Extindeți designul astfel încât să existe un contor care se incrementează la fiecare pachet transmis complet (adică atunci când se detectează TLAST='1' și handshake-ul se realizează). Afișați numărul de pachete pe LED-uri.
5. Implementați un mecanism timeout în receiver care aprinde un LED de eroare dacă nu sosește niciun pachet în ultimele N cicluri de clock.
6. Creați un test bench in vivado si analizați cronograma aplicației. Identificați momentele in care apare handshake-ul dintre transmiter si receiver.

Tema

1. Modificați transmitter-ul astfel încât, pe lângă nibble-ul de 4 biți din ROM, să calculeze și un bit de paritate (par sau impar, la alegere) pentru acei 4 biți. Transmitter-ul va transmite apoi un pachet de 8 biți format din nibble-ul original în partea de jos și bitul de paritate în partea de sus (restul de biți pot fi lăsați la zero). Receiver-ul trebuie extins astfel încât să verifice corectitudinea parității la fiecare pachet primit și să semnalizeze o eroare pe un LED separat dacă paritatea nu este corectă.
2. Exercițiile din lucrarea de laborator pe care nu le veți încheia in timpul laboratorului sa vor finaliza acasă.

Bibliografie

[1] Arm Ltd., “AMBA AXI-Stream Protocol Specification,” ARM IHI 0051B (ID040921), Cambridge, UK, 2010, rev. Apr. 2021.

[2] C. Vancea, F. Oniga, Arhitectura Calculatoarelor Îndrumător de Laborator, https://users.utcluj.ro/~vcristian/Indrumator_laborator_2024_Draft.pdf