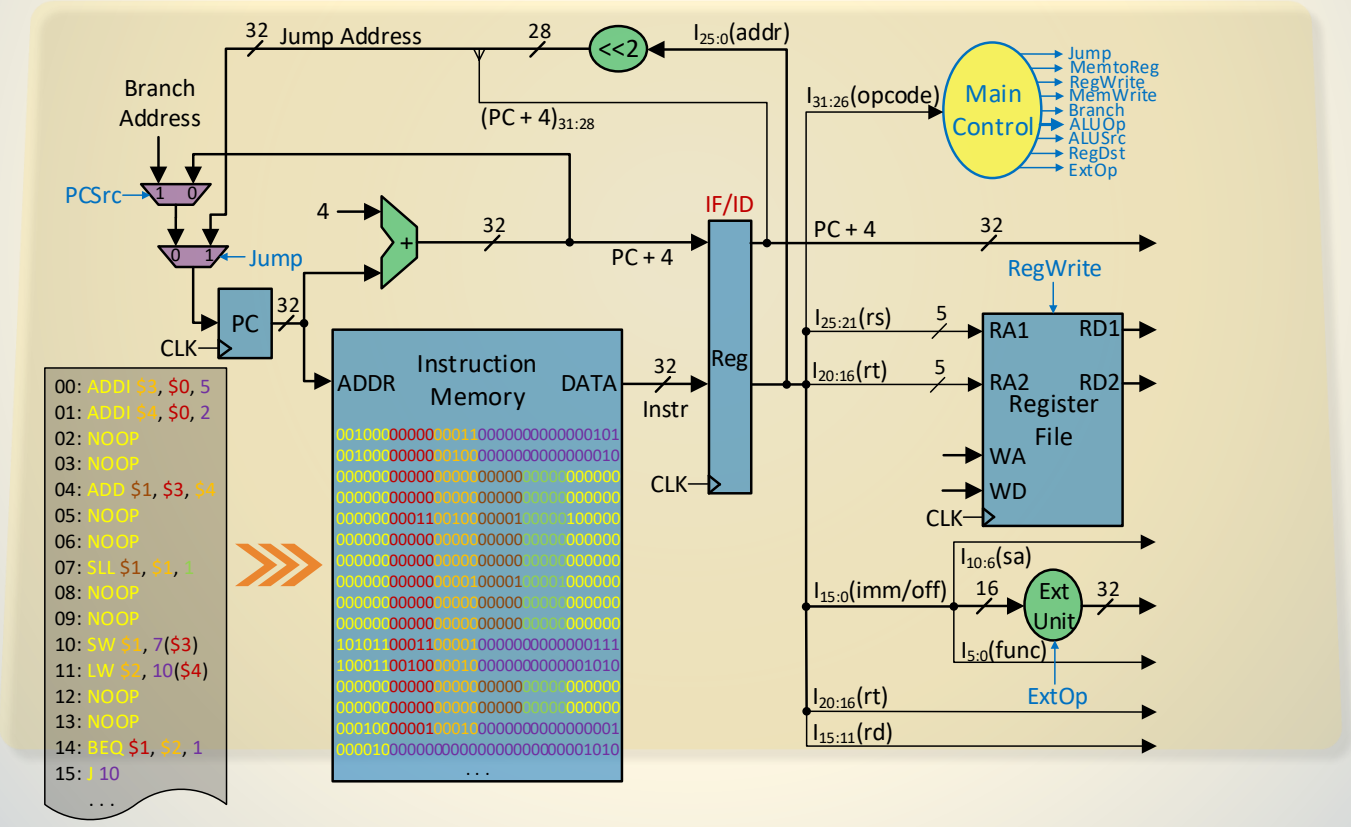
Cristian-Cosmin VANCEA Florin Ioan ONIGA

ARHITECTURA CALCULATOARELOR

Îndrumător de laborator



U.T.PRESS
Cluj-Napoca, 2026
ISBN 978-606-737- 834-4

Cristian-Cosmin VANCEA **Florin Ioan ONIGA**

ARHITECTURA CALCULATOARELOR

Îndrumător de laborator

U.T.PRESS
Cluj-Napoca, 2026
ISBN 978-606-737-834-4

Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.:0264-401.999
e-mail: utpress@biblio.utcluj.ro
https://biblioteca.utcluj.ro/editura

Recenzia: Prof.dr.ing. Radu Gabriel Dănescu
Conf.dr.ing. Tiberiu Marița

Pregătire format electronic on-line: Gabriela Groza

ISBN 978-606-737-834-4

Prefață

Îndrumătorul de laborator reprezintă materialul suport pentru punerea în aplicare a lucrărilor practice la disciplina Arhitectura Calculatoarelor, și este destinat studenților care urmează anul 2 al ciclului de licență, în domeniul Calculatoare și Tehnologia Informației. De asemenea, îndrumătorul se adresează și celor care abordează tehnicile hardware fundamentale, adoptate la nivel de microprocesor, prin proiectarea, implementarea și testarea etapizată a două variante de arhitectură MIPS (Microprocessor without Interlocked Pipelined Stages) pe 32 de biți.

Conținutul îndrumătorului se întinde pe 12 lucrări la care se adaugă 9 anexe. Partea introductivă cuprinde 3 lucrări în care se pune accent pe o descriere sintetizabilă, în VHDL, a circuitelor logice combinaționale și secvențiale de bază. Pe parcursul lucrărilor 4-8, urmează dezvoltarea procesorului MIPS cu ciclu unic. Migrarea la varianta pipeline este descrisă în lucrările 9-10. În lucrările 11-12 se studiază și se implementează arhitecturi de comunicație bazate pe protocolul de transmisie serială UART (Universal Asynchronous Receiver - Transmitter).

În cadrul lucrărilor se are în vedere o succesiune constructivă a informațiilor prezentate. Fiecare lucrare descrie la început obiectivele urmărite. Se continuă apoi cu prezentarea conceptelor la nivel de detaliu structural, punând astfel bazele implementării concrete a acestora. Finalul aparține activităților practice, concretizate cu implementări complete și testări riguroase, bazate pe o analiză a detaliilor la un nivel de granularitate ridicat. Pentru o mai bună familiarizare cu temele studiate, se recomandă studierea materialului de față, înainte de participarea la activitățile practice de laborator.

Autorii vă urează lectură plăcută!

Cuprins

Obiectivele laboratorului de Arhitectura Calculatoarelor

În cadrul acestui îndrumător de laborator se urmărește implementarea de microprocesoare didactice în arhitectură MIPS, folosind limbajul VHDL integrat în mediul de dezvoltare Vivado, și testarea acestora pe placa de dezvoltare Nexys A7. Pentru înțelegerea conceptelor prezentate este obligatorie parcurgerea activităților practice în întregime.

Obiectivele principale sunt:

- Descrierea de componente hardware sintetizabile în VHDL;
- Implementarea cu unealta Vivado și testarea pe plăcile Nexys A7;
- Deprinderea principiilor de funcționare ale arhitecturilor de tip ciclu unic și pipeline;
- Proiectarea și testarea arhitecturilor MIPS ciclu unic și MIPS pipeline;
- Implementarea protocolului de comunicare serială și integrarea cu procesorul MIPS.

Lucrarea 1

1. Introducere în mediul de dezvoltare Vivado

1.1. Obiective

Sunt descrise elementele de bază necesare realizării lucrărilor practice:

- Aplicația Vivado [1];
- Descrierea circuitelor în conformitate cu unealta de sinteză [2];
- Componentele plăcii de testare Nexys A7 [3].

1.2. Cerințele laboratorului

1. Placa Nexys A7 dotată cu FPGA din familia Artix-7 [3].
2. Aplicația Vivado – HLx Editions [4].
3. Cunoștințe de VHDL [5] studiate la disciplinele de specialitate anterioare.

1.3. Principii de bază ale descrierii circuitelor în VHDL

Pentru a facilita descrierea unui cod corect și sintetizabil se vor respecta următoarele criterii:

- Evitarea creării de entități pentru orice tip de circuit. În general, nu se creează entități pentru circuite simple, precum: porți logice, multiplexoare, decodificatoare, bistabile, registre, numărătoare.
- Evitarea descrierii structurale, la nivel de porți logice, pentru circuitele complexe.
- Utilizarea unei descrieri comportamentale sintetizabile.
- Crearea de entități doar pentru circuitele complexe.
- Menținerea proceselor la un grad de complexitate care să poată permite deducerea circuitului sintetizat.

1.4. Declararea semnalelor în VHDL

Într-o arhitectură, semnalele se declară între cuvintele cheie architecture și begin. Tipul unui semnal este std_logic, pentru 1 bit, respectiv std_logic_vector, pentru mai mulți biți. Exemple de declarații, după cum urmează:

- semnal pe 1 bit:
  ```
  signal sig_name : std_logic := '0';
  ```
- semnal pe *N* biți:
  ```
  signal sig_name : std_logic_vector(N-1 downto 0) := "00….0";
  ```

Exemple de inițializare a semnalelor:

- binară, pe 16 biți: "0000000000000000"
- hexazecimală, pe 32 biți: X"00000000"
- binară, pe un număr generic de biți: (others => '0')

1.5. Circuite de complexitate mică și medie

1.5.1. Porțile logice fundamentale

Porțile logice fundamentale (Figura 1. 1) sunt circuite combinaționale, care efectuează operații logice din algebra booleană, conform tabelului de adevăr (Tabelul 1. 1).

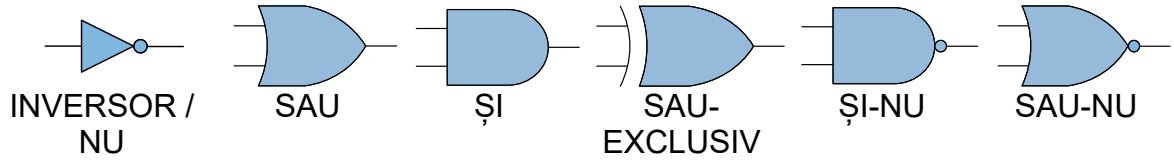


Figura 1. 1 Simbolurile porților logice fundamentale

A	INV
0	1
1	0

A	B	SAU	ȘI	SAU-EXCLUSIV	ȘI-NU	SAU-NU
0	0	0	0	0	1	1
0	1	1	0	1	1	0
1	0	1	0	1	1	0
1	1	1	1	0	0	0

Tabelul 1. 1 Tabelele de adevăr ale porților logice fundamentale

Porțile logice sunt sintetizate prin atribuire concurentă. Se descriu în afara proceselor, fără a utiliza alte entități, astfel:

```
R <= not A;          -- INVERSOR / NU
R <= A or B;         -- SAU
R <= A and B;        -- ȘI
R <= A xor B;        -- SAU-EXCLUSIV
R <= A nand B;       -- ȘI-NU
R <= A nor B;        -- SAU-NU
```

1.5.2. Multiplexorul

Multiplexorul (MUX) este un circuit combinațional, care expune la ieșire una din mai multe intrări, pe baza unui cod de selecție. Un multiplexor are 2^N intrări și N biți de selecție. Biții de selecție codifică (în binar) indexul intrării transmise spre ieșire. Pentru N=2 avem un MUX 4:1 (Figura 1. 2).

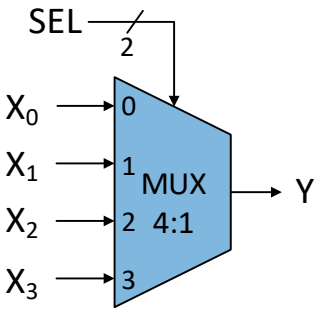


Figura 1. 2 Simbolul MUX 4:1

Pini	Descriere
X_0, X_1, X_2, X_3	Intrările de date
SEL	Intrarea de selecție
Y	Ieșirea de date

Tabelul 1. 2 Rolul pinilor unui MUX 4:1

Există mai multe posibilități de descriere a unui multiplexor sintetizabil:
a) concurențial, cu when-else;
b) în proces, cu if-then-else sau cu case.
Exemple:

- Mux 2:1

```
Y <= X0 when SEL = '0' else X1;
sau
process(SEL, X0, X1)
begin
   if SEL = '0' then
      Y <= X0;
   else
      Y <= X1;
   end if;
end process;
```

- Mux 4:1

```
process(SEL, X0, X1, X2, X3)
begin
   case SEL is
      when "00"    => Y <= X0;
      when "01"    => Y <= X1;
      when "10"    => Y <= X2;
      when others => Y <= X3;
   end case;
end process;
```

1.5.3. Decodificatorul

Decodificatorul (DCD) este un circuit combinațional care are *N* intrări și $M \leq 2^N$ ieșiri. Cei *N* biți de intrare indică (în binar) indexul ieșirii care este activată, restul rămânând inactive. Pentru *N*=3 avem un DCD 3:8 (Figura 1. 3).

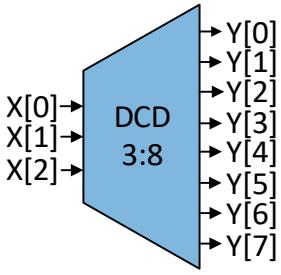


Figura 1. 3 Simbolul DCD 3:8

Pini	Descriere
X	Intrarea de selecție
Y	Ieșirea de date

Tabelul 1. 3 Rolul pinilor unui DCD 3:8

Un decodificator sintetizabil se descrie, cel mai frecvent, folosind instrucțiunea case în cadrul unui proces, astfel:

```
process(X)
begin
   case X is
      when "000"    => Y <= "00000001";
      when "001"    => Y <= "00000010";
      when "010"    => Y <= "00000100";
      when "011"    => Y <= "00001000";
      when "100"    => Y <= "00010000";
      when "101"    => Y <= "00100000";
      when "110"    => Y <= "01000000";
      when others   => Y <= "10000000";
   end case;
end process;
```

1.5.4. Bistabilul D Flip-Flop

Bistabilul D Flip-Flop (DFF) este un circuit secvențial care poate memora 1 bit de date, iar scrierea se poate face asincron sau sincron pe frontul semnalului de ceas. Se pot defini mai multe posibilități de control: Set/Reset asincron sau sincron, cu sau fără activator (enable) pentru semnalul de ceas (clock). **Notă**: Un registru funcționează ca un bistabil D pe mai mulți biți.

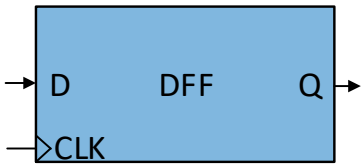


Figura 1. 4 Simbolul bistabilului D Flip-Flop

Pini	Descriere
D	Intrarea de date
CLK	Intrarea de ceas
Q	Ieșirea de date (bitul memorat)

Tabelul 1. 4 Rolul pinilor unui bistabil D Flip-Flop

Descrierea unui bistabil D sintetizabil, se face într-un proces în care se testează frontul ascendent, în felul următor:

```
if (CLK'event and CLK='1') then sau if rising_edge(CLK) then
```

Exemplu de bistabil D Flip-Flop:

```
process(CLK)
begin
```

```
    if rising_edge(CLK) then
        Q <= D;
    end if;
end process;
```

1.5.5. Numărătorul

Numărătorul este un circuit secvențial care contorizează impulsurile primite pe semnalul de ceas (clock). Există numărătoare cu semnale de control pentru: Set/Reset asincron sau sincron, încărcare (load) asincronă/sincronă, activare (enable) a numărării și definirea direcției de numărare (crescătoare, descrescătoare).

Figura 1. 5 Simbolul unui numărător sincron pe frontul ascendent cu semnal EN de activare (enable) a numărării

Pini	Descriere
CLK	Intrarea de ceas
EN	Intrarea de activare (enable) a numărării
CNT	Ieșirea de date (valoarea numărătorului)

Tabelul 1. 5 Rolul pinilor unui numărător cu activarea numărării

Exemplu de numărător sintetizabil în VHDL:

```
process(CLK)
begin
    if rising_edge(CLK) then
        if EN = '1' then
            CNT <= CNT + 1;
        end if;
    end if;
end process;
```

Notă: Pentru simularea în Vivado este necesară inițializarea semnalului CNT cu o valoare (ex. :=(others => '0')), la declararea acestuia. În lipsa unei inițializări, va avea valoarea "X…X" (necunoscută) și nu se va schimba ulterior deoarece "X…X"+1 = "X…X". În schimb, pe placă va funcționa și fără inițializare, fiindcă toate circuitele secvențiale neinițializate explicit primesc valori nule.

1.6. Activități practice

1.6.1. Implementați proiectul *test_env* în Vivado parcurgând ghidul din Anexa 1.

1.6.2. Adăugați un numărător binar reversibil pe 16 biți în arhitectura **test_env**. Descrieți comportamentul acestuia astfel încât numărarea să fie controlată de la un buton. Pașii de urmat sunt:

1. Declarați un semnal std_logic_vector de 16 biți înainte de begin. Se poate folosi Language Templates (Anexa 1) pentru a accesa descrierea comportamentală a numărătorului.
2. Folosiți un buton din porturile entității ca semnal de activare a ceasului (se va include un if suplimentar în corpul if-ului care testează frontul ascendent, pentru a verifica dacă semnalul butonului este 1).
3. Folosiți un switch pentru a controla direcția de numărare (alt if suplimentar care decide incrementarea, dacă switch='1' sau decrementarea, dacă switch='0').
4. Conectați ieșirile numărătorului la cele 16 LED-uri (atribuire concurentă). Comentați legarea comutatoarelor la LED-uri. O ieșire nu poate fi conectată la 2 resurse diferite!

Pentru verificarea corectitudinii, vizualizați schema circuitului rezultat, astfel: panoul **Flow Navigator > RTL Analysis** → fereastra Schematic → Reload (Reload este situat în partea superioară a ferestrei).

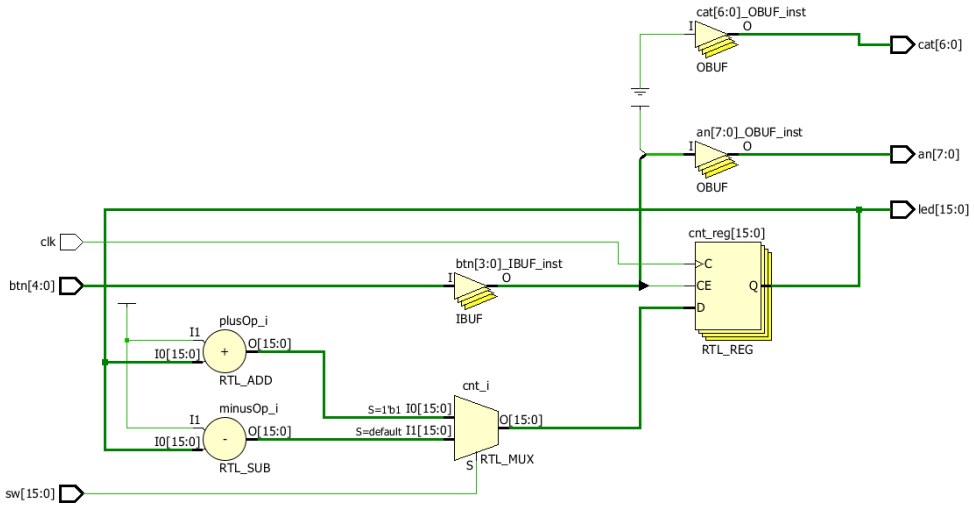


Figura 1. 6 Schema circuitului cu numărător

Generați fișierul .bit (panoul **Flow Navigator > Program and Debug > Generate Bitstream**) și programați placa. Realizați testarea controlând numărarea de la buton și direcția de numărare de la comutator (switch). Observați ce probleme apar.

1.6.3. Generator de Monoimpuls (MonoPulse Generator – MPG)

Pentru testarea arhitecturilor viitoare veți vizualiza fluxul de date și de control, pas cu pas, la fiecare apăsare a unui buton de comandă. În consecință, va fi necesar un semnal de activare controlată a frontului de ceas (denumit ENABLE).

Arhitectura circuitului MPG, care generează un astfel de semnal, sub forma unui singur impuls sincron produs la apăsarea butonului, este prezentată în figura următoare:

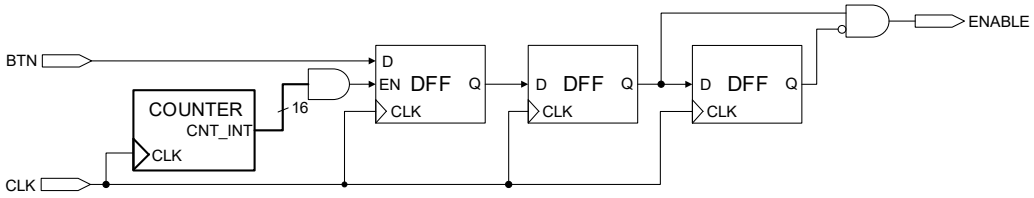


Figura 1. 7 Arhitectura generatorului de monoimpuls (MPG)

Rolul numărătorului și a bistabilului din stânga este de eșantionare regulată a semnalului BTN pentru a asigura robustețe la zgomotele electrice datorate uzării mecanice a butoanelor (pot apărea oscilații multiple la o singură apăsare). În funcție de uzură este posibil să fie nevoie de eșantionări la intervale mai mari, deci de mai mulți biți pentru numărător (ex: 16+).

Diagrama de timp care evidențiază efectul generat de MPG pe ieșirea ENABLE este prezentată în figura următoare:

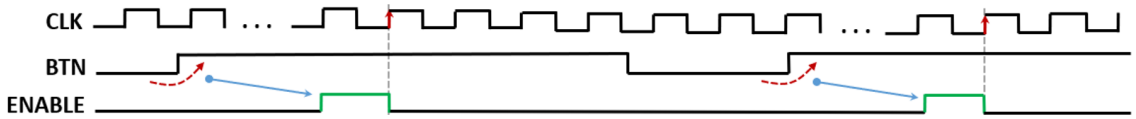
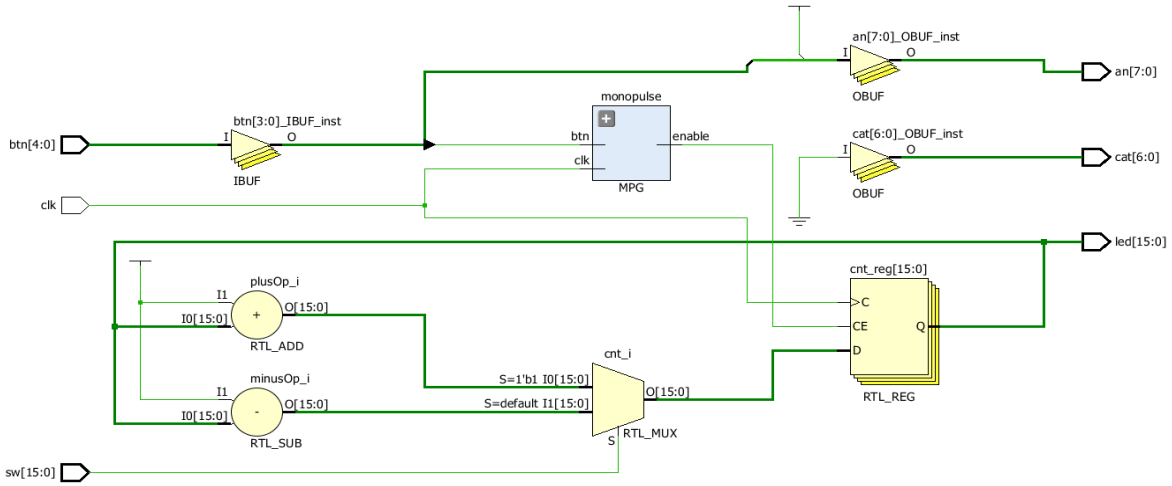


Figura 1. 8 Efectul unui MPG pe ieșire

MPG va fi o entitate separată (alt fișier), care trebuie creată de la meniu: **File → Add Sources → Add or create design sources → Create file**. Nu uitați să adăugați biblioteca IEEE.STD_LOGIC_UNSIGNED.ALL în fișierul de descriere al MPG. Utilizarea în arhitectura entității **test_env** necesită declararea cu component înainte de begin și instanțierea cu port map după begin.

Componentele din diagrama MPG (bistabile, numărător, porțile AND) se vor descrie comportamental prin declarare de semnale, definire de procese și atribuiri concurente corespunzătoare, astfel:

1. Scrieți codul VHDL pentru MPG.
2. Includeți MPG în arhitectura **test_env**.
3. Utilizați ieșirea ENABLE pentru activarea numărătorului adăugat la Activitatea 1.6.2: în descrierea numărătorului, în loc de testarea butonului se va testa dacă ENABLE este 1.
4. Vizualizați schema: **RTL Analysis** → fereastra Schematic → Reload.



5. Generați fișierul .bit, încărcați pe placă și testați.

1.6.4. Creați un nou fișier VHDL denumit *test_new*, folosind aceleași porturi ca până acum. Pașii de urmat:

1. Parcurgeți Anexa 1, începând cu **Pașii pentru crearea unui fișier VHDL** și implementați circuitul de mai jos în arhitectură. Setați ca Top Module prin click-dreapta în ierarhie pe **test_new → Set as Top**.

   Evitați pasul de adăugare a fișierului de constrângeri, deoarece fișierul de constrângeri curent definește aceleași porturi.

   MPG se va declara în arhitectura entității **test_new** cu component și se va instanția cu port map, iar restul componentelor se vor descrie în arhitectură, fără alte entități. Descrieți circuitul din Figura 1. 9, adăugând numărătorul pe 3 biți și decodificatorul DCD 3:8. Folosiți semnale, procese și atribuiri corespunzătoare.

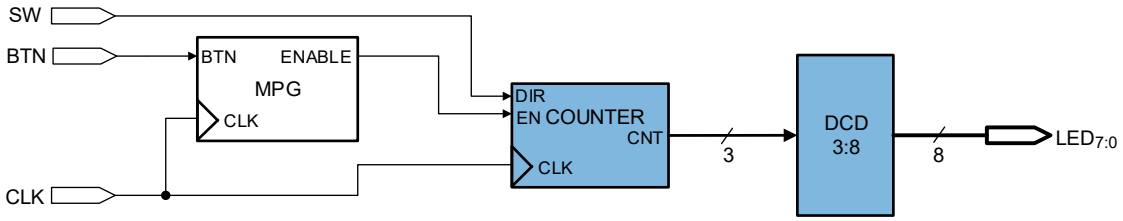


Figura 1. 9 Schema circuitului ***test_new***

2. Verificați schema: **RTL Analysis →** fereastra Schematic **→** Reload.
3. Generați fișierul .bit, încărcați pe placă și testați.
4. Realizați o simulare parcurgând cu atenție ghidul de la Anexa 2.

1.7. Bibliografie

[1] Vivado Design Suite Overview. Disponibil online: https://docs.amd.com/r/en-US/ug910-vivado-getting-started/Vivado-Design-Suite-Overview

[2] Vivado Design Suite User Guide – Synthesis (UG901). Disponibil online: https://docs.amd.com/r/en-US/ug901-vivado-synthesis

[3] Nexys A7 Reference Manual. Disponibil online: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

[4] Vivado Design Suite – HLx Editions. Disponibil online: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html

[5] VHDL Language Reference Guide. Disponibil online: https://peterfab.com/ref/vhdl/vhdl_renerta

Lucrarea 2

2. Afișoarele pe 7-segmente și unitatea aritmetică-logică

2.1. Obiective

Sunt prezentate posibilitățile de utilizare a afișoarelor pe 7 segmente și tehnica de implementare pentru o Unitate Aritmetică-Logică (Arithmetic Logic Unit – ALU). Adiacent, sunt descrise modurile de funcționare a resurselor de afișare pe placa de dezvoltare Nexys A7 [1], integrarea acestora în arhitecturi dezvoltate cu utilitarul Vivado [2] și descrierea în VHDL [3] a componentelor care efectuează operații aritmetice și logice de bază.

2.2. Afișoarele pe 7 segmente

Placa de dezvoltare este echipată cu afișoare pe 7 segmente (Seven Segment Display – SSD). Pentru afișarea unei cifre se folosesc 7 led-uri activate de 7 semnale denumite *catozi*. Fiecare afișor (cifră) este activat de un semnal denumit *anod*. *Catozii* și *anozii* sunt activi pe zero (în general, logica negativă conferă robustețe mai ridicată la eventuale zgomote din mediul de funcționare: perturbații electrice sau electromagnetice, șocuri mecanice, etc.). Din motive legate de economie a semnalelor alocate, *catozii* sunt comuni tuturor afișoarelor active, așadar ele vor afișa aceeași cifră. Pentru a afișa cifre diferite pe afișoare, acestea vor fi activate alternativ, în mod ciclic, pentru perioade scurte de timp, de 0.16ms, încât ochiul uman să nu perceapă efectul de licărire. Pentru fiecare perioadă, *catozii* vor fi configurați în conformitate cu cifra care trebuie afișată pe afișorul activ. În consecință, pentru numere cu 8 cifre, se va implementa protocolul descris de diagrama de timp din Figura 2. 1 [1].

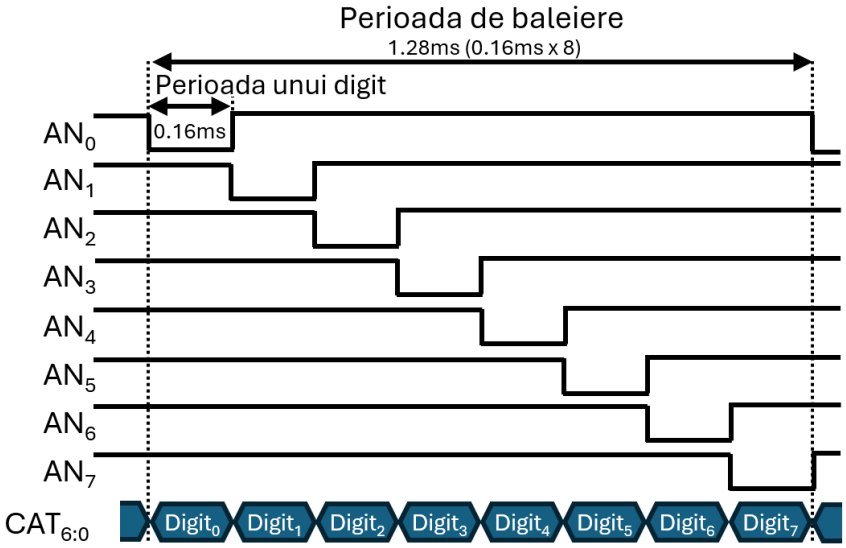


Figura 2. 1 Diagrama de timp pentru SSD

Schema logică a circuitului care implementează protocolul din Figura 2. 1 este prezentată în Figura 2. 2. Intrările sunt semnalul de ceas CLK (100MHz) și 8 semnale $Digit_{0-7}$ a câte 4 biți, care codifică cifrele hexazecimale de afișat. Ieșirile sunt *catozii* CAT și *anozii* AN. Cu ajutorul multiplexoarelor MUX 8:1, în funcție de valoarea înscrisă pe biții 16:14 ai numărătorului, afișoarele sunt activate alternativ (prin *anozi*), în paralel cu plasarea valorilor $Digit_i$ corespunzătoare pe *catozi*.

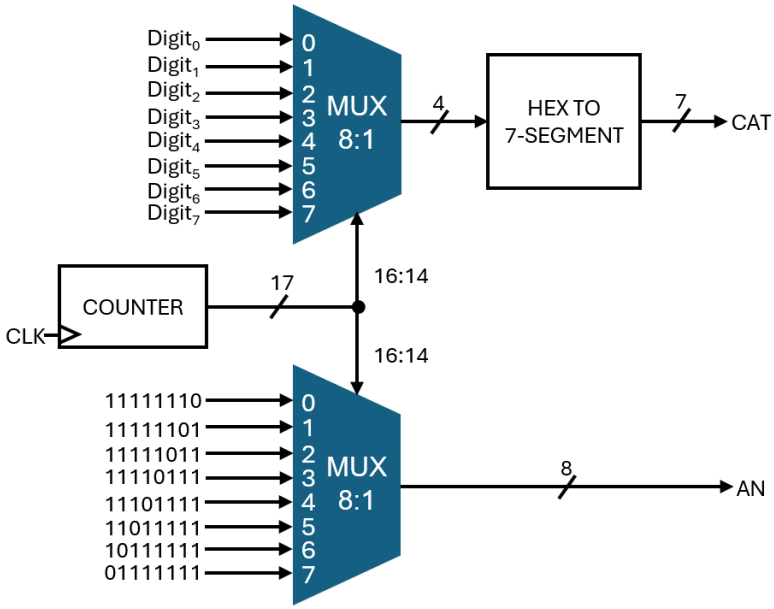


Figura 2. 2 Schema logică a circuitului SSD de afișare pe 8 afișoare

2.3. Structura de ansamblu a unui proiect

Pentru o dezvoltare sistematizată, se propune o structură generală a proiectelor implementate pe placa de dezvoltare (Figura 2. 3). Descrierea specifică în VHDL a circuitelor din fiecare lucrare se va face în blocul rezervat pentru User architecture.

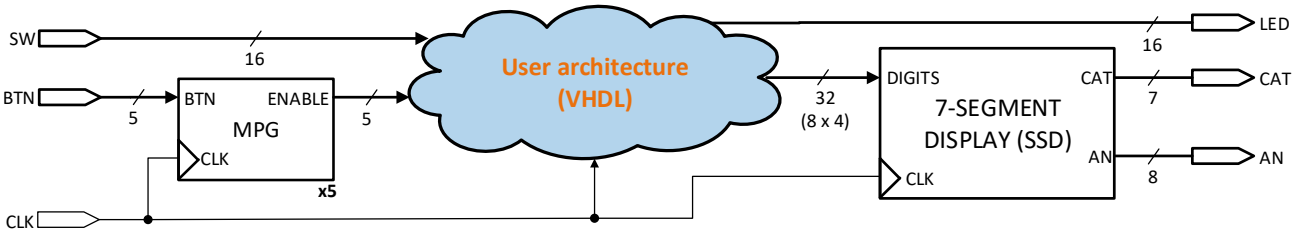


Figura 2. 3 Schema unui proiect la nivelul de abstractizare cel mai înalt

2.4. Circuite pentru operații aritmetice și logice

2.4.1. Sumatorul

Sumatorul (adder) efectuează operația de adunare. Unitatea cea mai simplă este sumatorul complet pe 1 bit, care are la bază următoarele expresii:

$S = X \oplus Y \oplus CI$, CI – Carry In

$CO = X \cdot Y + X \cdot CI + Y \cdot CI$, CO – Carry Out

Sumatoarele pe mai mulți biți sunt realizate prin cascadarea mai multor sumatoare pe 1 bit. Sumatorul pe 8 biți are simbolul din Figura 2. 4.

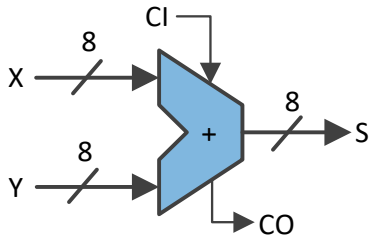


Figura 2. 4 Simbolul sumatorului complet pe 8 biți

VHDL: S <= X + Y; (se consideră CI=0, CO se ignoră)

2.4.2. Scăzătorul

Scăzătorul (subtractor) efectuează operația de scădere. În complement față de 2, scăderea se poate realiza prin adunarea cu complementul: $D = X + \overline{Y} + 1$,

VHDL: D <= X – Y;

2.4.3. Circuitele combinaționale de deplasare

Circuitele de deplasare (shifters) efectuează deplasarea biților, spre stânga sau spre dreapta, cu un număr de poziții. Prin deplasare, numărul de biți nu se schimbă! Deplasarea poate fi *logică* sau *aritmetică* (Figura 2. 5):

- *Deplasarea logică* – pozițiile rămase libere se completează cu 0;
- *Deplasarea aritmetică* – la deplasare spre stânga completează cu 0, iar la deplasare spre dreapta se completează cu bitul de semn.

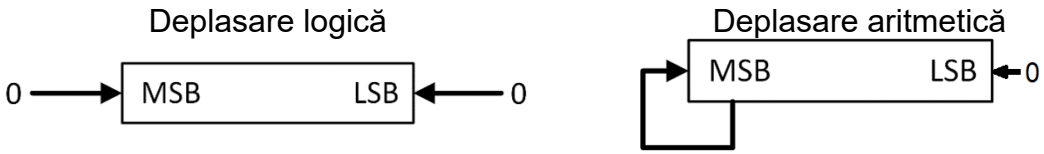


Figura 2. 5 Reprezentarea deplasărilor pe biți (MSB=Most Significant Bit, LSB=Least Significant Bit)

Definirea unui circuit de deplasare se realizează explicit cu operatorul de concatenare (&) sau cu alte funcții specializate definite în diverse librării.

VHDL: Y(31 downto 0) <= X(26 downto 0) & "00000";
(deplasare la stânga cu 5 poziții, pe 32 biți)
VHDL: Y(15 downto 0) <= "000" & X(15 downto 3);
(deplasare logică la dreapta cu 3 poziții, pe 16 biți)
VHDL: Y(7 downto 0) <= X(7) & X(7) & X(7 downto 2);
(deplasare aritmetică la dreapta cu 2 poziții, pe 8 biți)

Opțional, în Anexa 3 se prezintă un circuit combinațional de deplasare, care poate efectua deplasare între 0 și 3 poziții, în ambele direcții.

2.4.4. Circuitul de extindere cu semn sau cu zero

Un circuit de extindere cu semn (sign extender) sau cu zero (zero extender) realizează adaptarea unui semnal de dimensiune redusă la un număr mai ridicat de biți, fără a modifica valoarea stocată. Definirea unui astfel de circuit se realizează cu operatorul de concatenare (&). La extinderea cu 0, folosită la reprezentarea fără semn, se completează cu biți de 0, iar la extinderea cu semn se completează cu bitul de semn.

VHDL: Y(31 downto 0) <= x"0000" & X(15 downto 0);
(extindere cu 0, de la 16 la 32 biți)
VHDL: Y(9 downto 0) <= "000" & X(6 downto 0);
(extindere cu 0, de la 7 la 10 biți)
VHDL: Y(7 downto 0) <= X(5) & X(5) & X(5 downto 0);
(extindere cu semn, de la 6 la 8 biți)

2.4.5. Detectorul de zero

Detectorul de zero (zero detector) testează dacă un semnal pe *n* biți are valoarea 0 și se poate realiza cu o poartă NOR cu *n* intrări și răspunsul pe 1 bit.

VHDL: zero <= '1' when X=0 else '0'; -- *definirea cu when/else*

2.4.6. Comparatorul

Comparatorul (comparator) testează egalitatea a două semnale și generează un răspuns pe 1 bit. Se poate realiza cu porți logice fundamentale sau testând dacă diferența este nulă (scăzător + detector de zero).

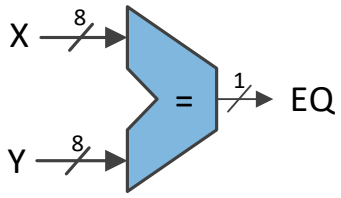


Figura 2. 6 Simbolul comparatorului pe 8-biți

VHDL: EQ <= '1' when X=Y else '0'; -- *definirea cu when/else*

2.5. Activități practice

2.5.1. Implementarea și testarea circuitului de afișare SSD

În proiectul *test_env* din lucrarea anterioară adăugați o entitate nouă denumită **SSD**, care să implementeze, în VHDL, circuitul de afișare din Figura 2. 2. Folosiți doar semnale și procese pentru a descrie elementele din schemă. Extrageți din Language Templates (VHDL > Synthesis Constructs > Coding Examples > Misc > 7-Segment Display Hex Conversion) unitatea “HEX TO 7-

SEGMENT", care convertește o valoare binară pe 4 biți la cele 7 segmente ale unui afișor.

Setați entitatea **test_env** să fie Top Module. Lucrați în arhitectura entității **test_env** următoarele:

1. Declarați entitatea SSD cu component în arhitectura entității **test_env** și instanțiați-o cu port map.
2. Modificați dimensiunea semnalului pentru numărătorul existent de la 16 la 32 de biți și conectați ieșirea acestuia la componenta SSD. Comentați (sau ștergeți) conexiunea ieșirii numărătorului la led-uri.

Numărătorul este controlat de la un buton prin componenta MPG. Parcurgeți pașii de programare pe placă și testați circuitul.

2.5.2. Implementarea pentru o unitate aritmetică-logică (ALU)

Fără a adăuga noi entități, transformați arhitectura **test_env** pentru a implementa o ALU cu 4 operații: adunare, scădere, deplasare la stânga cu 2 poziții, deplasare logică la dreapta cu 2 poziții, conform schemei din Figura 2. 7. Descrierea ALU se va face folosind doar semnale interne, procese și atribuiri concurente. Nu uitați să reduceți dimensiunea semnalului pentru numărător la 2 biți și numărarea să fie strict crescătoare. Eliminați conexiunea numărătorului la intrarea DIGITS a SSD. În schimb, conectați ieșirea MUX 4:1 la SSD.

Programați pe placă și testați circuitul.

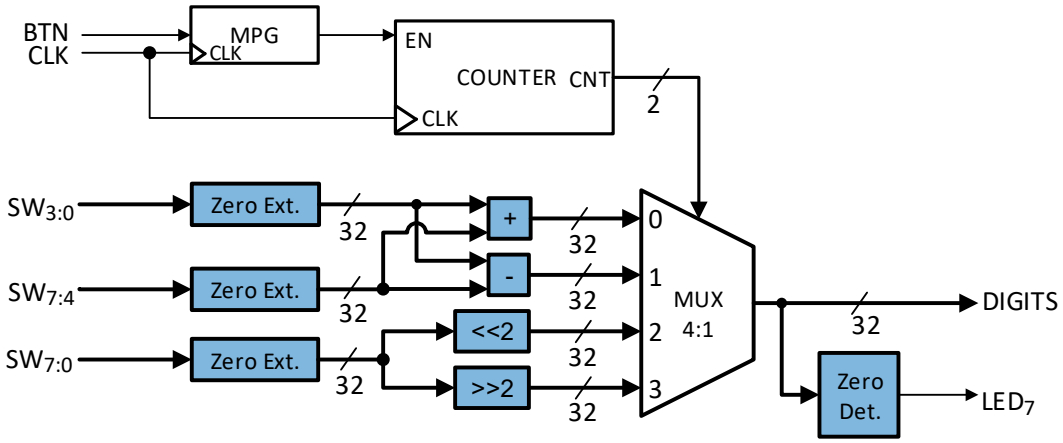


Figura 2. 7 Schema unității aritmetice-logice de test

2.6. Bibliografie

[1] Nexys A7 Reference Manual. Disponibil online: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

[2] Vivado Design Suite Overview. Disponibil online: https://docs.amd.com/r/en-US/ug910-vivado-getting-started/Vivado-Design-Suite-Overview

[3] VHDL Language Reference Guide. Disponibil online: https://peterfab.com/ref/vhdl/vhdl_renerta

Lucrarea 3

3. Unitățile de memorare

3.1. Obiective

Sunt prezentate diverse modalități de implementare a unităților de memorare sub formă de Bloc de Registre, memorii ROM (Read Only Memory) sau memorii RAM (Random Access Memory). De asemenea, sunt descrise particularitățile de organizare a resurselor de stocare pe placa de dezvoltare Nexys A7 [1], integrarea acestora în arhitecturi dezvoltate cu utilitarul Vivado [2] și descrierea în VHDL [3] a unităților de memorare cu diverse caracteristici.

3.2. Specificații de implementare

3.2.1. Implementarea Blocului de Registre

Blocul de Registre (Register File – RF) reprezintă spațiul de stocare folosit cel mai frecvent într-un procesor. Majoritatea operațiilor implică utilizarea sau modificarea datelor din Blocul de Registre, așadar accesul trebuie să atingă viteze mari și, în consecință, numărul de registre este limitat pentru a reduce timpii de propagare. Implementarea uzuală în FGPA este cu bistabile, permițând astfel operații simultane pe registre, la viteza de lucru a procesorului.

Arhitectura Blocului de Registre (Figura 3. 1) are două adrese de citire (denumite Read Address 1 și Read Address 2) și una de scriere (denumită Write Address). Valorile registrelor de la adresele de citire sunt livrate pe porturile Read Data 1, respectiv Read Data 2, citirea fiind asincronă. Datele de pe portul Write Data sunt scrise în registrul de la adresa de scriere, atunci când comanda RegWrite este activă. Scrierea este sincronă pe frontul de ceas. Anexa 4 prezintă descrierea în VHDL a unui bloc de 32 registre pe 32 de biți.

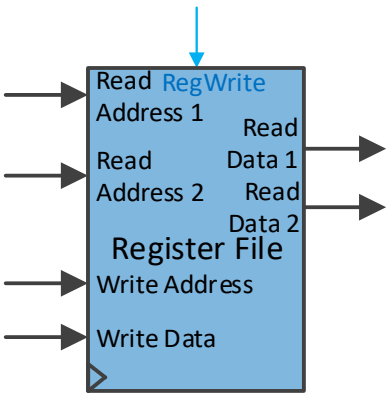


Figura 3. 1 Simbolul Blocului de Registre

3.2.2. Implementarea memoriilor ROM și RAM

Memoriile ROM permit operații de citire, iar memoriile RAM permit citirea și scrierea la adresele indicate.

Dispozitivele FPGA sunt echipate cu un anumit număr de blocuri de memorie BRAM (Block RAM) [1]. O memorie ROM poate fi implementată cu blocuri BRAM sau distribuit, cu tabele de asociere (Lookup-Up Tables – LUT). Similar, o memorie RAM poate fi implementată cu blocuri BRAM sau distribuit, cu registre. Atât la ROM cât și la RAM, implementarea distribuită este mai rapidă doar pentru dimensiuni mici, însă viteza scade odată cu creșterea capacității memoriei generate, deoarece crește numărul de componente LUT, respectiv registre implicate. Implementarea cu blocuri BRAM este puțin mai lentă, dar este afectată în mai mică măsură la generarea unor memorii mari, deoarece consumul de astfel de blocuri este mult redus, fiindcă vin dotate cu dimensiuni considerabil superioare. Așadar, la memorii de dimensiuni mici este avantajoasă implementarea distribuită, iar la memorii de dimensiuni mari este recomandată implementarea cu BRAM.

La memorii ROM, unealta de sinteză decide automat implementarea cea mai avantajoasă în funcție de context [4].

La memorii RAM, dacă citirea este asincronă se implementează distribuit, iar dacă este sincronă se utilizează blocuri BRAM disponibile [5]. Printre altele, unealta de sinteză acceptă următoarele caracteristici pentru RAM:

- cu sau fără comandă de activare (enable);
- scriere sincronă;
- citirea poate să fie sincronă sau asincronă;
- una sau două porturi de citire.

Memoria RAM cu citire sincronă acceptă trei variante de implementare [5]: *write-first*, *read-first* și *no-change*. Acestea se deosebesc prin informația care se va regăsi pe portul de ieșire după o operație de scriere, astfel:

- *write-first* – *pe ieșire apare informația care se scrie;*
- *read-first* – pe ieșire apare informația de la adresa de scriere, care era stocată înainte de scriere;
- *no-change* – ieșirea nu se modifică la operații de scriere, ci doar la operații de citire.

Anexa 5 prezintă descrierea în VHDL a unei memorii RAM 64x32 (64 locații de 32 biți) cu citire sincronă și comportament *write-first*. Citirea este sincronă fiindcă este descrisă în procesul sensibil pe clock, condiționată de frontul de ceas. Comparativ, la Anexa 4 citirea este asincronă deoarece apare în afara procesului. În consecință, Blocul de Registre este o memorie RAM distribuită.

Căutați în Language Templates la secțiunea VHDL > Synthesis Constructs > Coding Examples > RAM > Block RAM > Single Port și comparați descrierea în VHDL a celor 3 tipuri de memorie RAM cu citire sincronă (BRAM). Pentru analiză, studiați doar procesul în care se realizează scrierea și citirea din memorie, ignorând restul codului.

Notă: Dacă în VHDL accesul la memorie se face cu funcția conv_integer() atunci includeți librăria **IEEE.STD_LOGIC_UNSIGNED.ALL**, iar dacă se face cu funcția to_integer(unsigned()) atunci includeți librăria **IEEE.NUMERIC_STD.ALL**.

3.2.3. Declararea și inițializarea unei memorii

Declararea unei memorii în VHDL presupune definirea, în prealabil, a tipului acesteia ca un șir cu *N* locații (0 to *N*-1) de *M* biți (*M*-1 downto 0):

```
type <mem_type> is array (0 to N-1) of std_logic_vector(M-1 downto 0);
signal mem_name : <mem_type>;
```

Notă: Se pot declara oricâte memorii de același tip. În lipsa unei inițializări, pe placă, memoriile primesc implicit 0, dar în simulatorul Vivado primesc "U…U" (Unknown). De aceea este important să le inițializați explicit, la declarare, cu valorile dorite sau cel puțin cu valori nule, astfel:

```
signal mem_name : <mem_type> := (
        "01…0",             -- M biți în reprezentare binară
         X"E…1",            --    sau hexazecimală
        others => "00…0" -- inițializează restul locațiilor cu aceeași valoare
);
```

3.3. Activități practice

Continuați lucrul în proiectul *test_env*. Curățați arhitectura entității **test_env**, încât să conțină numai o unitate MPG și un SSD ca în Figura 2. 3.

3.3.1. Implementarea memoriei ROM

Declarați o memorie ROM 32x32 în arhitectura **test_env** (nu creați o entitate nouă!) inițializată cu câteva valori alese la alegere (vezi Secțiunea 3.2.3). Un numărător pe 5 biți va genera adresa, putând fi controlat de la un buton prin MPG. Pe SSD se va afișa valoarea din memorie de la adresa curentă. Schema circuitului este în Figura 3. 2. Comportamentul memoriei ROM se descrie ca o simplă citire asincronă. Încărcați circuitul pe placă și testați.

Figura 3. 2 Schema de test a unei memorii ROM 32x32

3.3.2. Implementarea Blocului de Registre

Adăugați în proiectul *test_env* o nouă entitate în care veți implementa un Bloc de Registre (RF) de capacitate 32x32 (Anexa 4), inițializat cu câteva valori consecutive. Ulterior, declarați componenta în arhitectura entității **test_env**. Comentați citirea din memoria ROM pentru a elimina implementarea ei și descrieți în arhitectura entității schema din Figura 3. 3. Folosiți un numărător pe 5 biți cu resetare asincronă, la comun, pentru toate adresele de citire și de scriere. Numărarea (pinul EN) și scrierea în memorie (pinul RegWr) vor fi controlate prin două butoane, cu ajutorul a două unități MPG. Valorile pe cele două porturi de ieșire de date vor fi adunate cu un sumator, iar rezultatul va fi conectat pe DIGITS la SSD și pe portul de scriere WD. Astfel, circuitul va afișa pe SSD dublul valorii de la adresa curentă, iar la fiecare scriere conținutul de la adresa curentă se va dubla. Conectați comanda de resetare asincronă a numărătorului (RST) la un al 3-lea buton. Acesta nu necesită MPG deoarece comenzile asincrone nu sunt dependente de frontul de ceas. Astfel, după parcurgerea primelor registre veți putea reveni la adresa 0, pentru a verifica dacă în urma primei parcurgeri s-a stocat valoarea dublată la locațiile în care s-a activat scrierea. Încărcați circuitul pe placă și testați.

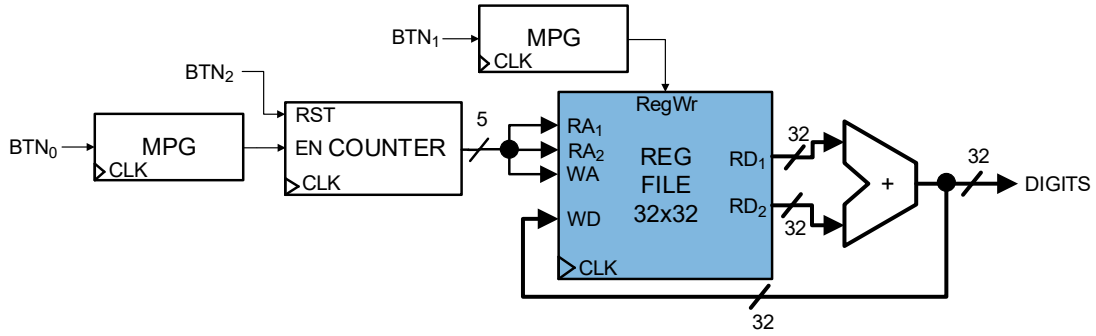


Figura 3. 3 Schema de test a unui Bloc de Registre 32x32

3.3.3. Implementarea unei memorii RAM cu citire sincronă (BRAM)

Adăugați în proiectul *test_env* o nouă entitate în care veți implementa o memorie RAM 64x32 cu citire sincronă, de tip *write-first* (Anexa 5), inițializată cu câteva valori consecutive. Ulterior, declarați componenta în arhitectura entității **test_env** și comentați instanțierea Blocului de Registre pentru a-l elimina. Înlocuiți sumatorul cu un circuit de deplasare la stânga cu 2 poziții (folosiți operatorul & de concatenare). Folosiți un numărător pe 6 biți pentru a genera adresa memoriei RAM. Ieșirea de date a memoriei se va afișa pe SSD, după deplasare, și se va conecta la portul de scriere de date. Testați funcționarea pe placă.

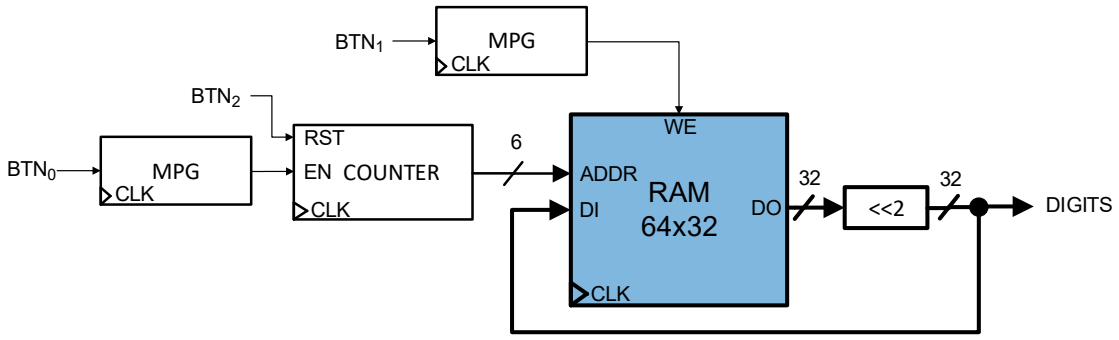


Figura 3. 4 Schema de test a unui RAM 64x32 cu citire sincronă

3.4. Bibliografie

[1] Nexys A7 Reference Manual. Disponibil online: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual
[2] Vivado Design Suite Overview. Disponibil online: https://docs.amd.com/r/en-US/ug910-vivado-getting-started/Vivado-Design-Suite-Overview
[3] VHDL Language Reference Guide. Disponibil online: https://peterfab.com/ref/vhdl/vhdl_renerta
[4] Vivado Design Suite User Guide – Synthesis (UG901): ROM HDL Coding Guidelines. Disponibil online: https://docs.amd.com/r/en-US/ug901-vivado-synthesis/ROM-HDL-Coding-Techniques
[5] Vivado Design Suite User Guide – Synthesis (UG901): RAM HDL Coding Guidelines. Disponibil online: https://docs.amd.com/r/en-US/ug901-vivado-synthesis/RAM-HDL-Coding-Guidelines

Lucrarea 4

4. Procesorul MIPS 32, ciclu unic – Introducere

Definirea setului de instrucțiuni și conceperea programului de test

4.1. Obiective

Descrierea, implementarea și testarea pentru:

- Procesorul MIPS pe 32 de biți, cu ciclu unic (single-cycle).

Familiarizarea cu:

- Setul de instrucțiuni;
- Conceperea unui program de test în asamblare și cod-mașină.

4.2. Specificațiile arhitecturii MIPS cu ciclu unic pe 32 de biți

Notă: În comparație cu varianta standard [1], se va implementa o arhitectură cu un set redus de instrucțiuni și memorii de dimensiune mai mică, dar principiile de proiectare se păstrează.

Instrucțiunile au un format pe 32 de biți și sunt clasificate în 3 categorii:

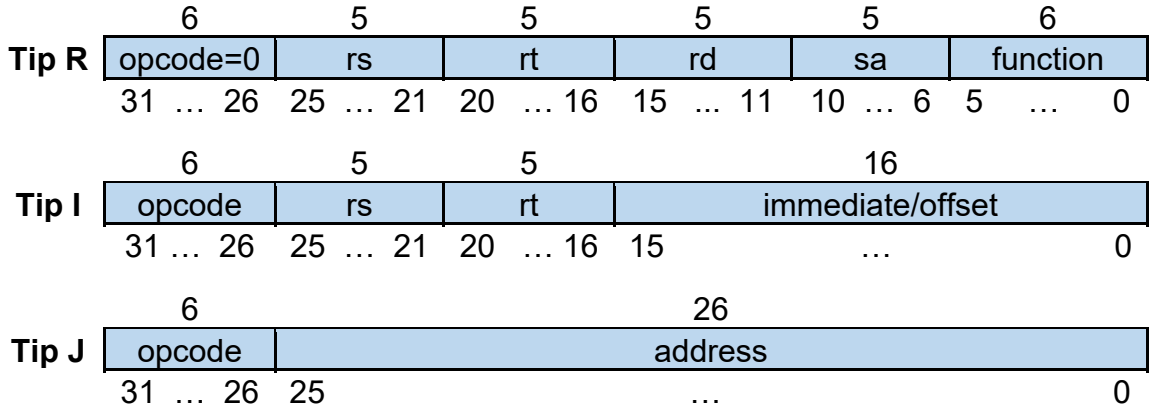


Figura 4. 1 Formatul instrucțiunilor MIPS 32

Conform standardului MIPS 32 ISA [2][3], câmpul **opcode** pe 6 biți are valoarea 0 pentru instrucțiuni de tip R (Register) și codifică unic instrucțiunile din celelalte 2 categorii I (Immediate), respectiv J (Jump). Instrucțiunile de tip R sunt codificate unic în câmpul **function**, pe 6 biți. În total, se pot codifica 64 (2^6) instrucțiuni de tip R și 63 (2^6-1) instrucțiuni de tip I și J.

Tabelul 4. 1 conține setul minimal de instrucțiuni, din fiecare tip, care se vor implementa pe procesorul MIPS 32. Pe pozițiile rămase libere veți defini alte instrucțiuni, astfel că procesorul va putea executa cel puțin 15 instrucțiuni, din care o parte le veți folosi pentru a concepe programul de test.

Tip R	Adunare	add
	Scădere	sub
	Deplasare logică la stânga (shift left logical)	sll
	Deplasare logică la dreapta (shift right logical)	srl
	Operația logică ȘI	and
	Operația logică SAU	or
	…. de definit!	
	…. de definit!	
Tip I	Adunare cu imediat (add immediate)	addi
	Încărcare cuvânt din memorie (load word)	lw
	Stocare cuvânt în memorie (store word)	sw
	Salt în caz de egalitate (branch on equal)	beq
	…. de definit!	
	…. de definit!	
Tip J	Salt necondiționat (jump)	j

Tabelul 4. 1 Setul de instrucțiuni pentru MIPS 32

Principalele elemente din calea de date a procesorului MIPS 32 sunt:

- Registrul PC (contorul de program) pe 32 biți cu încărcare sincronă pe frontul ascendent al semnalului de ceas.
- Memoria de instrucțiuni cu citire asincronă (ROM) conține:
  - o intrare pentru adresa instrucțiunii;
  - o ieșire cu conținutul instrucțiunii (pe 32 biți) de la adresa curentă.
- Blocul de registre RF conține:
  - 3 intrări de adresă pe 5 biți, din care 2 de citire (Read Address 1, Read Address 2) și una de scriere (Write Address);
  - acces multiplu pe 2 ieșiri asincrone (Read Data 1, Read Data 2) la conținutul registrelor (pe 32 biți) de la adresele de citire;
  - intrarea de date Write Data pe 32 biți pentru scriere sincronă (pe frontul ascendent) în registrul indicat de adresa de scriere;
  - semnalul de control RegWrite pentru scrierea în memorie.
- Memoria de date (RAM) conține:
  - intrarea de adresă Address;
  - ieșirea de date Read Data pe 32 biți pentru citirea asincronă a conținutului de la adresa curentă.
  - intrarea de date Write Data pe 32 biți pentru scriere sincronă la adresa curentă;
  - semnalul de control MemWrite pentru scrierea în memorie.
- Unitatea de extindere de la 16 la 32 biți, cu următoarele funcții:
  - dacă semnalul de control ExtOp = 1 se realizează extindere cu semn;
  - dacă semnalul de control ExtOp = 0 se realizează extindere cu zero.
- Unități de deplasare la stânga cu 2 biți pentru alinierea adresei de salt (jump sau branch) la multiplu de 4 octeți.
- Unitatea aritmetică-logică ALU cu operanzi și rezultat pe 32 biți.
  - semnalul de control ALUCtrl (generat din codul de pe semnalul ALUOp și câmpul de instrucțiune *function*) codifică operația de efectuat. Dimensiunea acestuia se stabilește în funcție de operațiile necesare.

4.3. Activități practice

4.3.1. Proiectarea instrucțiunilor și definirea căii de date

Completați Tabelul 4.1 cu două instrucțiuni de tip R și două de tip I din Anexa 6. Veți avea 15 instrucțiuni. Scrieți punctual (pe caiet) următoarele detalii, pentru fiecare instrucțiune, având ca referință Anexa 6:

1. Sintaxa în asamblare (ASM).
2. Descrierea RTL abstract.
3. Formatul pe biți în cod mașină cu valoarea aleasă în câmpul **opcode** și **function** (unde este cazul).
4. Dați un exemplu concret în ASM și codificați-l pe biți în cod mașină. Ex. add $3, $1, $2 => B"000000_00001_00010_00011_00000_… (restul biților până la 32)". Pentru lizibilitate crescută, folosiți caracterul '_' între câmpuri și prefixul 'B'. **Notă**: VHDL acceptă separarea biților cu caracterul '_' dacă șirul de biți este precedat de caracterul 'B' pentru valori binare sau de caracterul 'X' pentru valori hexazecimale.
5. Diagrama de procesare (după modelul prezentat în cursul 3) doar pentru instrucțiunile add, lw, beq și j, iar pentru celelalte se finalizează ca temă pentru acasă.

4.3.2. Programul de testare

Concepeți un program în ASM (de 7-8 rânduri) folosind doar instrucțiunile din tabelul completat (nu neapărat toate). Rescrieți programul în cod-mașină, pe 32 de biți, cu separatorul '_' între câmpuri. Programul trebuie să conțină cel puțin:

1. o scriere într-un registru, urmată de citiri ale registrului respectiv;
2. o scriere în memorie, urmată de citiri de la aceeași adresă.

Reveniți la Activitatea 3.3.1 din Lucrarea 3 și inițializați memoria ROM cu programul de test în cod-mașină. Pentru a ușura procesul de testare pe placă, introduceți în dreptul fiecărei instrucțiuni un comentariu care să conțină codul-mașină în hexazecimal, poziția instrucțiunii (începe cu 0 și se incrementează din 1 în 1), instrucțiunea în ASM și o scurtă descriere a efectului instrucțiunii. Încărcați pe placă și verificați afișarea corectă în hexazecimal a programului în cod-mașină.

Temă: Alegeți o problemă de la Anexa 7 și scrieți programul încât să conțină cel puțin o buclă cu minim 2 iterații, realizată folosind instrucțiuni de salt (jump/branch).

4.3.3. Arhitectura procesorului personalizat

Temă: Având ca model noțiunile din Cursul 4 și Figura 5. 1 din Lucrarea 5, desenați arhitectura procesorului cu toate componentele necesare pentru cele 15 instrucțiuni. Identificați semnalele de control necesare și completați un tabel cu valorile acestora pentru fiecare instrucțiune în parte.

4.4. Bibliografie

[1] D. A. Patterson and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
[2] Imagination Technologies LTD. MIPS Architecture for Programmers, Volume I-A: Introduction to the MIPS32 Architecture, Revision 6.01, August 2014.
[3] Imagination Technologies LTD. MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual, Revision 6.06, December 2016.

Lucrarea 5

5. Procesorul MIPS 32, ciclu unic – Extragerea instrucțiunilor

Unitatea de extragere a instrucțiunilor

5.1. Obiective

Descrierea, implementarea și testarea pentru:

- Unitatea de extragere a instrucțiunilor (Instruction Fetch - IFetch).

5.2. Specificații suplimentare pentru MIPS cu ciclu unic

Execuția unei instrucțiuni are următoarele 5 etape (detaliate în cursul 4!):

1) IF – Instruction Fetch (Extragerea instrucțiunii);
2) ID/OF – Instruction Decode / Operand Fetch (Decodificarea instrucțiunii / extragerea operanzilor);
3) EX – Execute (Execuție);
4) MEM – Memory (Memorie);
5) WB – Write-Back (Scrierea rezultatului).

Structura procesorului MIPS, care implementează etapele de execuție este prezentată în figura următoare [1]. Pentru simplificare, lipsesc conexiunile complete ale semnalelor de control.

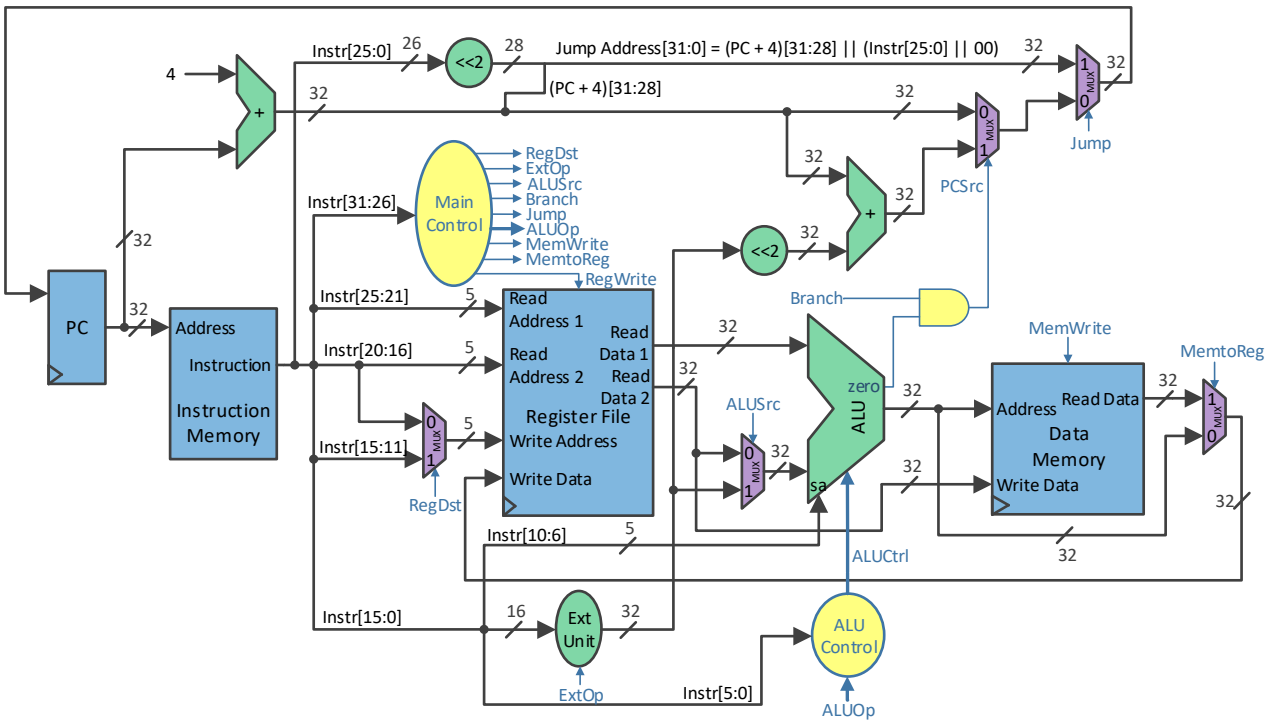


Figura 5. 1 Arhitectura de ansamblu pentru MIPS 32, ciclu-unic

Formatul instrucțiunilor diferă în funcție de categorie, astfel [2][3]:

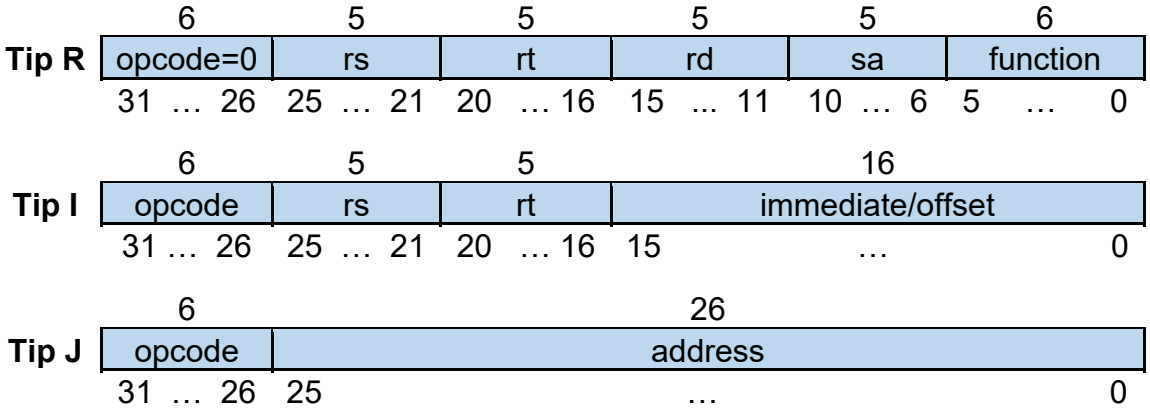


Figura 5. 2 Formatul instrucțiunilor MIPS 32

Etapa de extragere a instrucțiunii se va implementa într-o entitate separată, care se va declara și se va instanția în arhitectura **test_env**. **Notă**: Implementarea modularizată e arhitecturii cu ciclu unic nu îmbunătățește performanța, dar va ușura tranziția la varianta pipeline, care va fi studiată în lucrările viitoare.

Unitatea de extragere a instrucțiunilor IFetch (Figura 5. 3) conține următoarele elemente:

- Program Counter (PC) – registru cu adresa instrucțiunii curente;
- Instruction Memory – memoria de instrucțiuni (ROM);
- Sumator – calculează PC+4, adresa imediat următoare în ROM;
- Multiplexoare MUX 2:1 – selectează adresa viitoarei instrucțiuni, între PC+4 și adresele de salt.

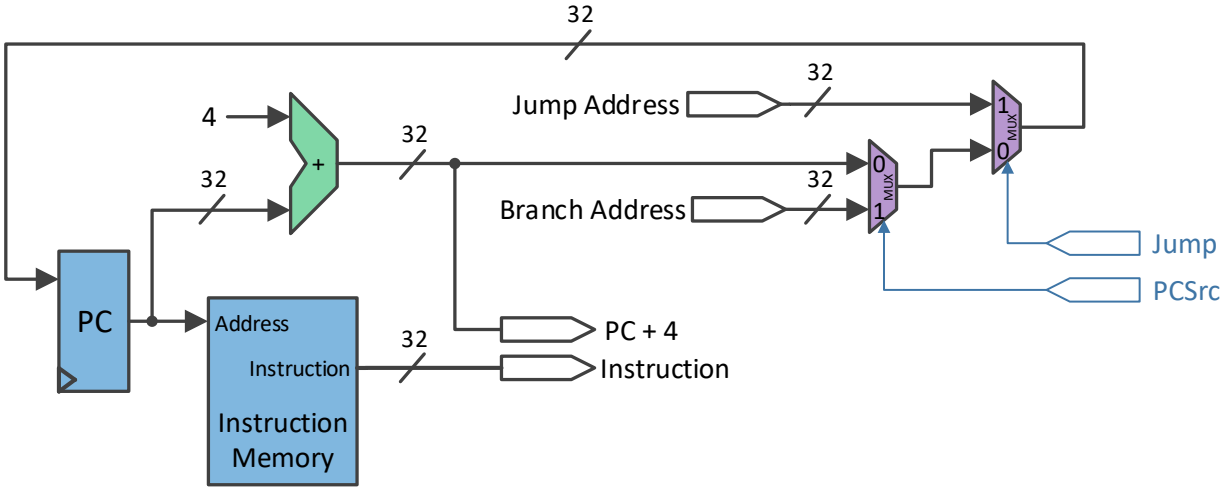


Figura 5. 3 Unitatea de extragere a instrucțiunilor IFetch

Unitatea IFetch primește pe intrările de date adresele de salt și pune la dispoziție, pe ieșiri, adresa imediat următoare (PC+4), respectiv conținutul instrucțiunii curente. Adresele pot să fie de salt condiționat (branch) sau necondiționat (jump). Cu ajutorul intrărilor de control Jump și PCSrc se decide care va fi valoarea viitoare a registrului PC, în felul următor:

- Dacă Jump = 1, atunci PC ← Jump Address;
- Dacă Jump = 0, atunci:
  - Dacă PCSrc = 1, atunci PC ← Branch Address;
  - Dacă PCSrc = 0, atunci PC ← PC+4.

Notă: Dacă se va implementa instrucțiunea JR, atunci va fi nevoie de o nouă intrare de adresă (JR Address) și un MUX 2:1 suplimentar, controlat de semnalul JmpR asociat instrucțiunii JR. Relațiile se schimbă astfel:

- Dacă JmpR = 1, atunci PC ← JR Address;
- Dacă JmpR = 0, atunci:
  - Dacă Jump = 1, atunci PC ← Jump Address;
  - Dacă Jump = 0, atunci:
    - Dacă PCSrc = 1, atunci PC ← Branch Address;
    - Dacă PCSrc = 0, atunci PC ← PC+4.

5.3. Activități practice

Resurse necesare înainte de începerea activităților practice:

- Cele 15 instrucțiuni alese, cu RTL și formatul binar (Lucrarea 4, Activitatea 4.3.1).
- Entitatea **test_env** cu memoria ROM de instrucțiuni inițializată cu programul în cod-mașină (Lucrarea 4, Activitatea 4.3.2).
- Arhitectura procesorului personalizat, având ca punct de pornire Figura 5. 1 (Lucrarea 4, Activitatea 4.3.3).

5.3.1. Elaborarea unității IFetch

Descrieți o nouă entitate **IFetch**, după arhitectura prezentată în Figura 5. 3, la care adăugați modificările necesare, în funcție de caz. Descrieți toate unitățile din schemă în cadrul arhitecturii, fără a folosi entități suplimentare:

- Definiți registrul PC pe 32 de biți cu încărcare pe frontul ascendent folosind un proces. Deoarece testarea se va realiza secvențial, controlând execuția instrucțiunilor de la un buton, registrul va avea un semnal de activare (enable) conectat la buton prin MPG. Adăugați și un semnal de reset asincron cu ajutorul căruia se va putea relua execuția (PC=0 înseamnă revenirea la prima instrucțiune), apăsând un alt buton. Comanda fiind asincronă butonul de reset nu necesită MPG.
- Definiți multiplexoarele folosind procese (atenție la semnalele din lista de sensibilitate!) sau concurențial cu when-else.
- Descrierea memoriei ROM de instrucțiuni poate fi copiată din lucrarea anterioară. Memoria are 32 de cuvinte (suficient pentru dimensiunea programului de testare), așadar adresa este pe 5 biți. Deoarece memoria accesează 4 octeți odată (32 de biți) și PC adresează la nivel de octet, valoarea PC va trebui împărțită la 4 prin deplasare la dreapta cu 2 biți. Pentru aceasta se vor ignora cei 2 biți mai puțin semnificativi $PC_{1:0}$, astfel că pe adresa ROM se vor conecta cei 5 biți $PC_{6:2}$ din totalul de 32.
- Sumatorul se poate descrie cu o singură linie de cod în VHDL.

5.3.2. Specificații de testare

Declarați și instanțiați entitatea **IFetch** în arhitectura entității **test_env**. Veți mai avea nevoie de MPG și SSD. Comentați/ștergeți elementele inutile rămase în arhitectură de la lucrarea anterioară.

Conectați componenta **IFetch**, prin MPG, la butonul de control cu care veți simula execuția secvențială, prin apăsare repetată. Conectați intrarea de reset asincron la un alt buton, fără MPG.

Ieșirile de date (instrucțiunea curentă și PC+4) vor fi afișate pe SSD prin multiplexare cu MUX 2:1 controlat de la comutatorul SW_7, astfel:

- se afișează instrucțiunea dacă $SW_7 = 0$;
- se afișează PC+4 dacă $SW_7 = 1$.

Pentru simularea salturilor, semnalele de control Jump și PCSrc vor fi conectate la SW_0, respectiv SW_1. Dacă e nevoie de JmpR, conectați-l la SW_2. Adresele de salt, indisponibile deocamdată, le veți inițializa cu valori constante, astfel: Jump Address = X"00000000", Branch Address = X"00000010", iar dacă este cazul, JR Address = X"00000000". În consecință, la o comandă de jump programul va reîncepe (după apăsarea butonului de control) cu prima instrucțiune, iar la o comandă de branch, saltul se va face la instrucțiunea de la poziția 4 (valoarea X"00000010" = 16_{10} se împarte la 4, pentru a adresa cuvinte de 32 biți).
Notă: Pentru testare, puteți alege și alte valori constante, multipli de 4.

5.4. Bibliografie

[1] D. A. Patterson and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
[2] Imagination Technologies LTD. MIPS Architecture for Programmers, Volume I-A: Introduction to the MIPS32 Architecture, Revision 6.01, August 2014.
[3] Imagination Technologies LTD. MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual, Revision 6.06, December 2016.

Lucrarea 6

6. Procesorul MIPS 32, ciclu unic – Decodificare și control

Unitatea de decodificare și unitatea de control

6.1. Obiective

Descrierea, implementarea și testarea pentru:

- Unitatea de decodificare a instrucțiunilor (Instruction Decode – ID);
- Unitatea de control (UC).

6.2. Specificații suplimentare pentru MIPS cu ciclu unic

Pentru încadrarea exactă a obiectivelor urmărite în acest laborator sunt reluate noțiunile generale de bază prezentate în lucrarea anterioară. Execuția unei instrucțiuni are următoarele 5 etape: 1) IF (Instruction Fetch); 2) ID/OF (Instruction Decode / Operand Fetch); 3) EX (Execute); 4) MEM (Memory); 5) WB (Write-Back).

Structura procesorului MIPS este prezentată în continuare [1]:

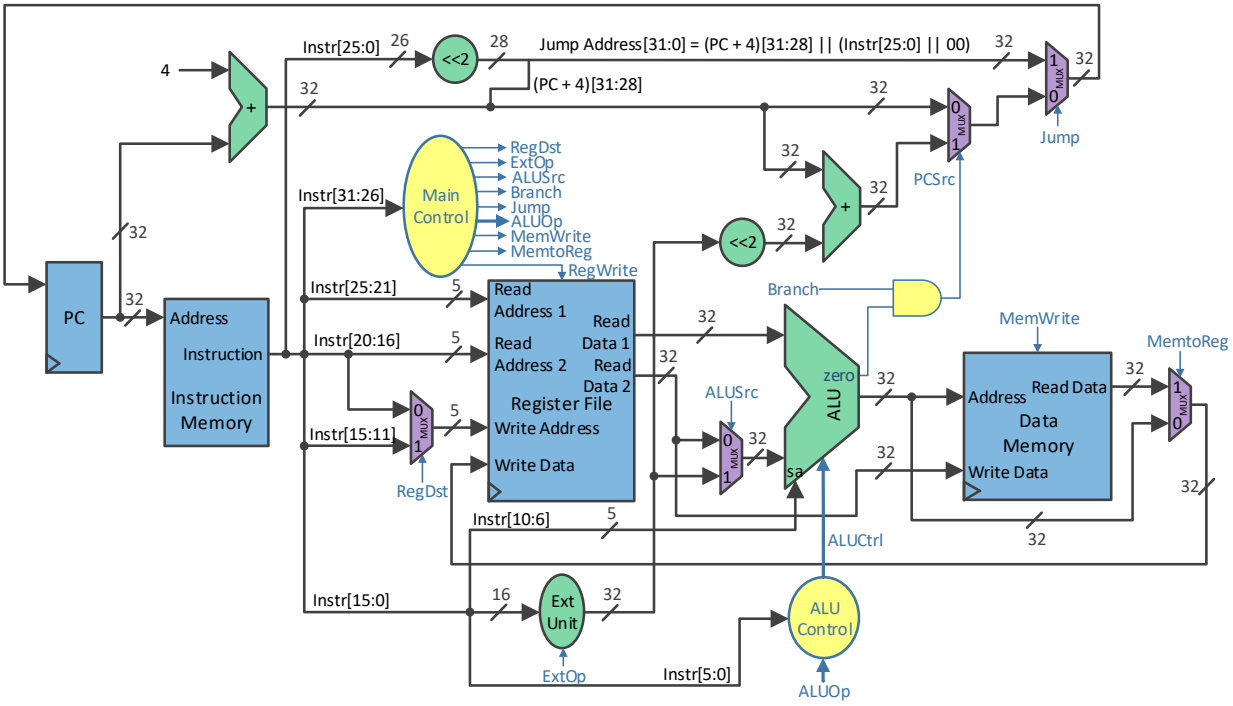


Figura 6. 1 Arhitectura MIPS 32, ciclu-unic

Formatul instrucțiunilor în funcție de categorie:

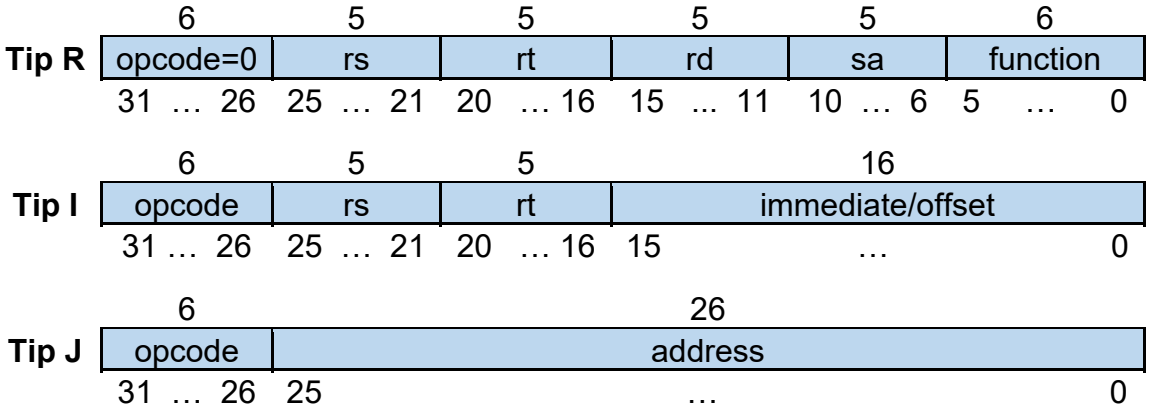


Figura 6. 2 Formatul instrucțiunilor MIPS 32

Etapa de decodificare a instrucțiunii și extragere a operanzilor se va implementa cu unitatea Instruction Decode (ID) și Unitatea de Control (UC). Separarea este motivată, în principal, de delimitarea dintre calea date și calea de control, în cadrul arhitecturii. Ambele entități se vor declara și instanția în arhitectura **test_env**.

Unitatea de decodificare a instrucțiunilor ID (Figura 6. 3) realizează extragerea operanzilor și conține următoarele elemente:

- Register File (RF) – bloc de 32 registre pe 32 de biți (Lucrarea 3, 3.2.1);
- Multiplexor MUX 2:1 – stabilește adresa de scriere în RF;
- Unitate de extindere (Ext Unit) – extinde valoarea câmpului *immediate* la 32 de biți (imediatul extins).

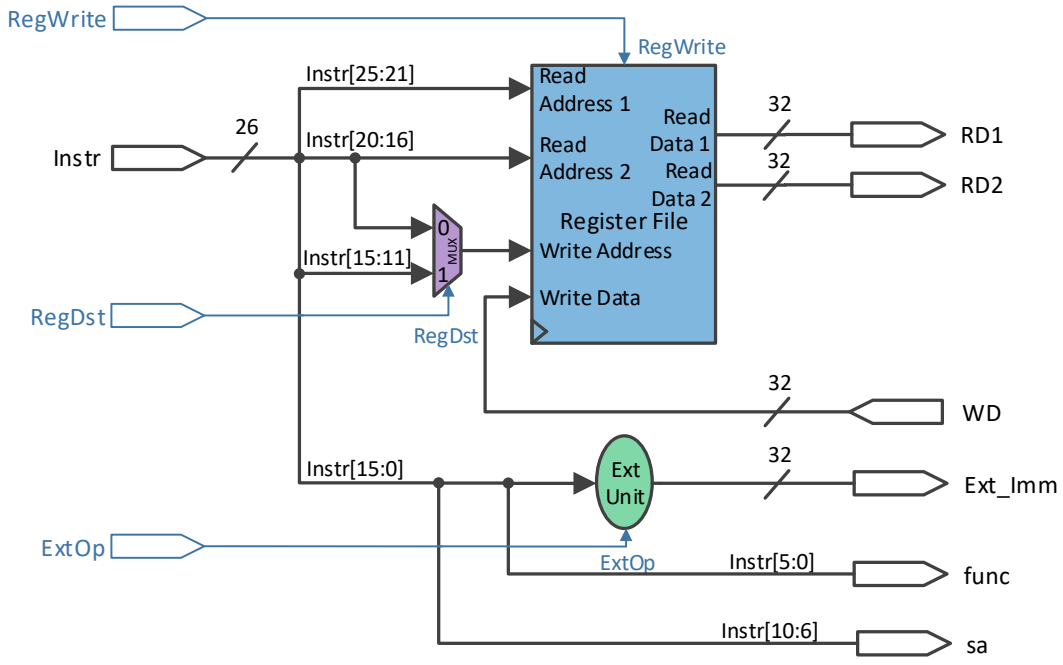


Figura 6. 3 Unitatea de decodificare a instrucțiunilor ID

Unitatea ID primește pe intrările de date instrucțiunea curentă și valoarea WD, care se scrie în RF, ambele pe 32 de biți. ID pune la dispoziție pe ieșiri,

operanzii RD1, RD2 și imediatul extins Ext_Imm, tot pe 32 de biți. Suplimentar, pe ieșire mai apar câmpurile *function* (6 biți) și *sa* (5 biți) din instrucțiune. Semnalul de control RegDst selectează registrul (adresa) în care se scrie valoarea WD atunci când semnalul de control RegWrite este activ. Scrierea este sincronă pe frontul ascendent și selecția are loc între câmpurile *rd* și *rt* din instrucțiune:

- Dacă RegDst = 0, atunci se scrie în registrul indicat de *rt*;
- Dacă RegDst = 1, atunci se scrie în registrul indicat de *rd*.

Extinderea imediatului de la 16 biți la 32 de biți se realizează în funcție de semnalul de control ExtOp:

- Dacă ExtOp = 0, atunci extinderea este cu zero (necesară la operații logice pe biți, cu valori constante);
- Dacă ExtOp = 1, atunci extinderea este cu semn.

Unitatea de Control UC (Figura 6. 4), generează semnalele care determină funcționalitatea unităților din calea de date (detalii în cursul 4 sau [1]!).

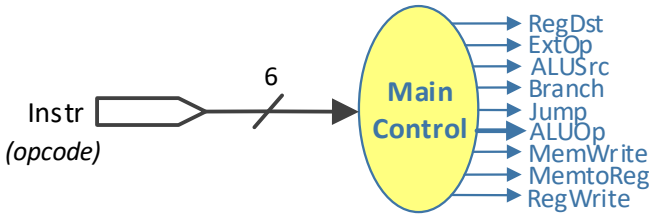


Figura 6. 4 Unitatea de control UC

Intrarea în UC este câmpul *opcode* pe 6 biți al instrucțiunii, iar ieșirea constă din semnalele de control pentru calea de date, exceptând semnalul ALUOp (pe 2+ biți), care codifică operația aritmetică-logică de efectuat pentru instrucțiunea curentă. Dacă alegeți să implementați instrucțiunea JR, atunci pe ieșire va apărea și semnalul JmpR.

6.3. Activități practice

Resurse necesare:

- Proiectul *test_env* cu unitatea IFetch de la lucrarea anterioară.

6.3.1. Elaborarea unității ID

Descrieți o nouă entitate **ID**, după arhitectura prezentată în Figura 6. 3, fără a folosi entități suplimentare:

- Definiți blocul de registre RF de capacitate 32x32 (Anexa 4) inițializat cu valori nule: := (others => X"00000000"). Scrierea este pe frontul ascendent și trebuie controlată de același buton ca în cazul registrului PC din unitatea IFetch. Așadar, definiți un semnal suplimentar de validare a scrierii. Validarea se descrie în VHDL ca o condiție suplimentară asociată testării lui RegWrite (se adaugă un and la if).
- Definiți multiplexorul cu un proces sau concurențial cu when-else.

- Pentru unitatea de extindere revizuiți tehnicile din Lucrarea 2, capitolul 2.4.4 și definiți cele două operații de extindere, în cadrul unui proces bazat pe if sau concurențial cu when-else. Exemplu:

```
Ext_Imm(15 downto 0) <= Instr(15 downto 0);
Ext_Imm(31 downto 16) <= (others => Instr(15)) when ExtOp = '1' else
                         (others => '0');
```

6.3.2. Elaborarea unității UC

Descrieți o nouă entitate **UC** (Figura 6. 4) folosind tabelul completat pentru Activitatea 4.3.3 din Lucrarea 4. În lipsa acestuia, completați-l pentru 4-6 instrucțiuni și începeți implementarea, urmând să-l finalizați ca temă. Arhitectura **UC** implementează un decodificator care se poate descrie într-un proces VHDL, cu case pe intrarea Instr (deci trebuie să apară și în lista de sensibilitate). Pe fiecare ramură when semnalele de control se pot inițializa conform cu valorile din tabel. Știind că într-un proces doar ultima atribuire are efect asupra unui semnal, o variantă simplificată ar fi să se atribuie valori nule pentru toate ieșirile, înainte de case, și pe ramurile when să se actualizeze doar ieșirile care nu trebuie să fie nule.

6.3.3. Specificații de testare

Declarați și instanțiați entitățile **ID** și **UC** în arhitectura entității **test_env** de la lucrarea anterioară. Conectați-le cu componenta IFetch prin semnalele comune, urmărind schema de la Activitatea 4.3.3 din Lucrarea 4 sau Figura 6. 1:

- sursa IFetch: Instruction se conectează la ID și UC;
- sursa UC: Jump (și eventual JmpR) se conectează la IFetch; RegWrite, RegDst și ExtOp se conectează la ID.

Conectați semnalul de validare a scrierii în blocul de registre la ieșirea MPG. Pentru a testa operația de scriere, calculați RD1+RD2 și returnați rezultatul la WD. **Notă**: Scrierea sumei în RF se va face doar pentru instrucțiunile care implică o scriere în blocul de registre. Pe adresele de la IFetch păstrați valorile constante.

Pentru a verifica dacă semnalele de control sunt corecte (corespund cu valorile din tabel), conectați ieșirile UC la led-uri. Pentru ALUOp folosiți mai multe led-uri, în funcție de dimensiunea sa.

Afișarea pe SSD a mai multor semnale din calea de date se va realiza cu un MUX 8:1, în funcție de comutatoarele $SW_{7:5}$, după următoarea configurație:

- "000": afișează Instruction (de la IFetch);
- "001": afișează PC+4 (de la IFetch);
- "010": afișează RD1 (de la ID);
- "011": afișează RD2 (de la ID);
- "100": afișează WD (de la ID);
- … alte semnale: func sau sa (de la ID, extinse cu zero la 32 de biți).

Folosiți butoanele de control pentru a testa execuția secvențială și resetul.

6.4. Bibliografie

[1] D. A. Patterson and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.

Lucrarea 7

7. Procesorul MIPS 32, ciclu unic – Finalizarea arhitecturii

Unitatea de execuție, unitatea de memorie și unitatea de scriere a rezultatului

7.1. Obiective

Descrierea, implementarea și testarea pentru:

- Unitatea de execuție a instrucțiunilor (Instruction Execute – EX);
- Unitatea de memorie pentru date (MEM);
- Unitatea de scriere a rezultatelor (Write-Back – WB);
- Alte conexiuni necesare.

7.2. Specificații suplimentare pentru MIPS cu ciclu unic

Obiectivele urmărite în cadrul acestui laborator acoperă ultimele 3 etape de execuție ale unei instrucțiuni. Suportul acestora în structura procesorului MIPS [1] (Figura 7. 1), este reprezentat de unitățile ALU, Data Memory, respectiv multiplexorul MUX 2:1, cu selecția MemtoReg.

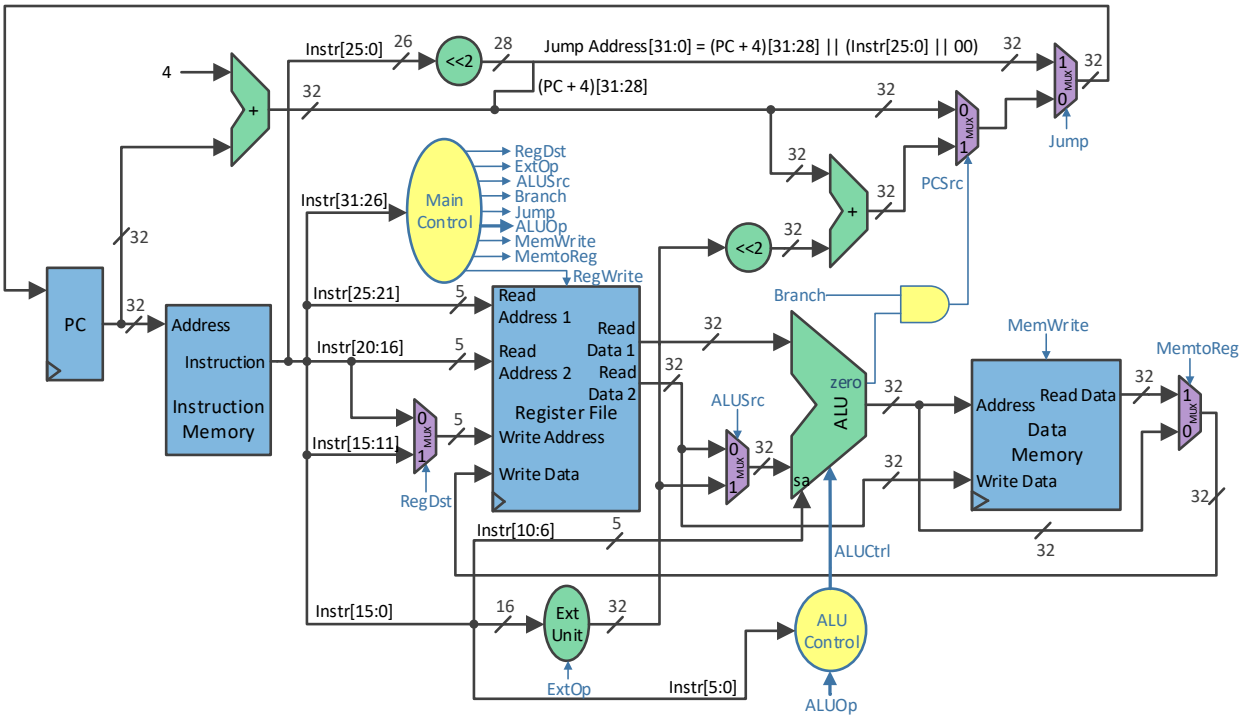


Figura 7. 1 Arhitectura MIPS 32, ciclu-unic

Cele 5 etape de execuție a unei instrucțiuni sunt [2][3]: 1) IF (Instruction Fetch); 2) ID/OF (Instruction Decode / Operand Fetch); 3) EX (Execute); 4) MEM (Memory); 5) WB (Write-Back).

Formatul instrucțiunilor în funcție de categorie:

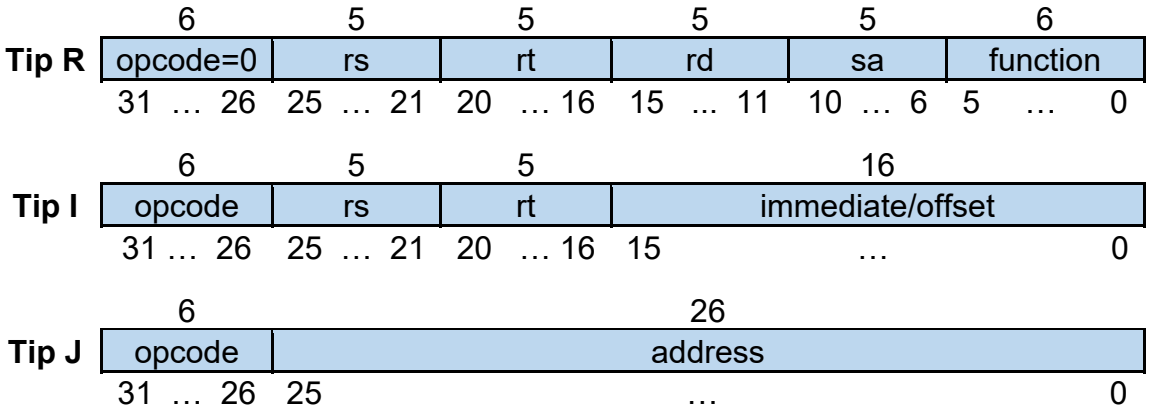


Figura 7. 2 Formatul instrucțiunilor MIPS 32

Etapele de execuție și memorie se vor implementa cu entitățile EX, respectiv MEM, care vor fi declarate și instanțiate în arhitectura **test_env**. Datorită structurii sale simple, etapa de scriere a rezultatului se va descrie în cadrul arhitecturii **test_env**, fără entități suplimentare.

Unitatea de execuție EX (Figura 7. 3) realizează operațiile aritmetice și logice necesare instrucțiunii. Are în componență următoarele elemente:

- Unitatea Aritmetică-Logică (ALU – Lucrarea 2, Activitatea 2.5.2);
- Unitatea de control pentru ALU (ALU Control) – generează codul operației pentru ALU;
- Multiplexor MUX 2:1 – stabilește sursa celui de-al 2-lea operand pentru ALU, între Read Data 2 (RD2) și imediatul extins (Ext_imm);
- Unitatea de deplasare la stânga cu 2 poziții a imediatului extins și sumatorul pentru calculul adresei de salt condiționat (branch).

Unitatea EX primește pe intrările de date registrele RD1 și RD2 de la blocul de registre, imediatul extins Ext_imm și adresa de instrucțiune imediat următoare PC+4, codificate pe 32 de biți. Suplimentar, apar câmpurile func și sa din instrucțiunea curentă, pe 6 biți, respectiv 5 biți. EX pune la dispoziție rezultatul ALU cu semnalul de validare Zero (care indică un rezultat nul) și adresa de salt condiționat Branch Address, calculată astfel:

$$\text{Branch Address} \leftarrow (\text{PC+4}) + (\text{Ext\_imm} << 2)$$

Primul operand în ALU este RD1, iar cel de-al doilea este ales între RD1 și Ext_imm cu ajutorul semnalului de control ALUSrc.

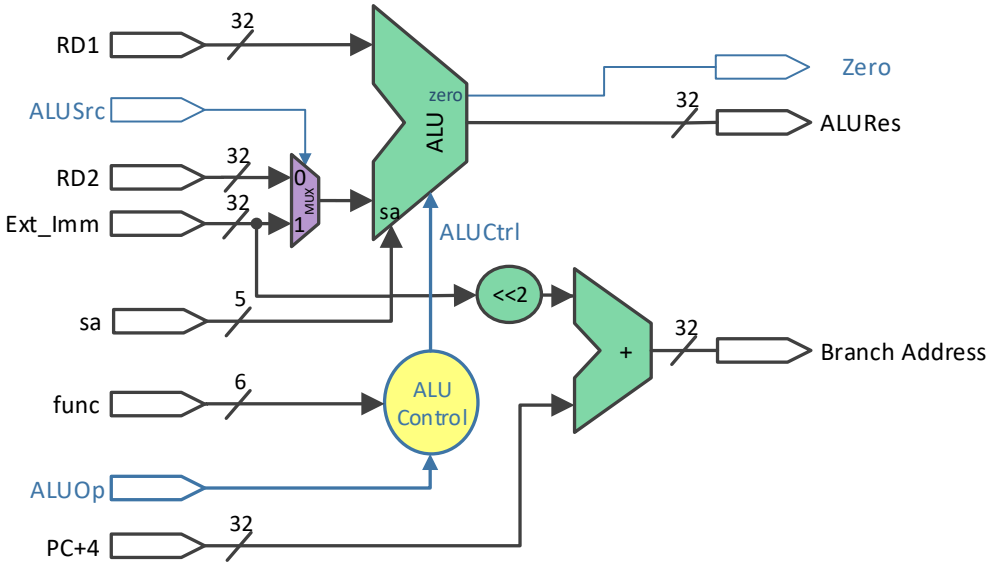


Figura 7. 3 Unitatea de execuție EX

Codul operației ALUOp generat de unitatea de control UC (Lucrarea 6, Activitatea 6.3.2) este convertit la codul ALUCtrl, care determină operația de efectuat în ALU. Această recodificare este necesară deoarece ține cont de câmpul *func*, în cazul instrucțiunilor de tip R. **Observație**: Instrucțiunile de tip I pot păstra (nu e obligatoriu!) codul ALUOp și pentru ALUCtrl, dar va trebui reprezentat pe numărul corect de biți, dacă ALUCtrl are o dimensiune diferită de ALUOp. Dimensiunea ALUCtrl se stabilește în funcție de numărul total de operații prevăzute pentru unitatea ALU.

Unitatea de memorie MEM (Figura 7. 4) are rol de stocare a datelor, pe 32 de biți. Scrierea în memorie este sincronă pe frontul de ceas ascendent și citirea este asincronă, ca la blocul de registre RF. O memorie similară este descrisă în Anexa 5, cu deosebirea că citirea este sincronă.

Memoria primește pe intrările de date adresa curentă (ALURes de la ALU, pe 32 de biți) și valoarea registrului RD2 (pe 32 de biți), care se va scrie la locația indicată, dacă semnalul de control MemWrite este activ. De asemenea, memoria pune la dispoziție, pe ieșirea MemData, cuvântul de 32 biți aflat la adresa curentă. Instrucțiunile de acces la memorie sunt SW (Store Word) și LW (Load Word).

În paralel, rezultatul ALURes de la ALU este înaintat la ieșire, pentru a putea fi scris în blocul de registre: $ALURes_{Out} \leftarrow ALURes_{In}$.

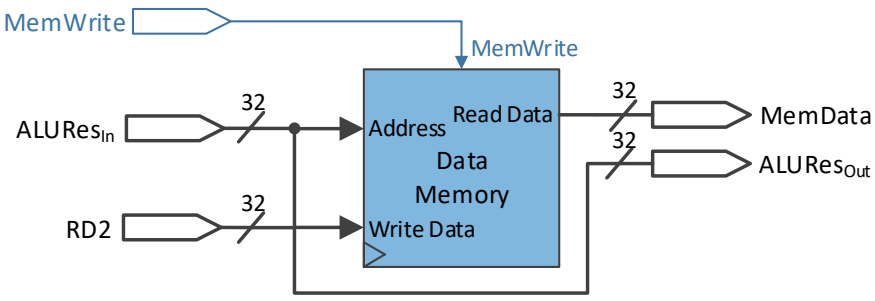


Figura 7. 4 Unitatea de memorie MEM

Unitatea de scriere a rezultatului WB (Write-Back) este reprezentată de multiplexorul cu selecția MemtoReg:

- dacă MemtoReg = 0, se scrie $ALURes_{Out}$ în blocul de registre RF;
- dacă MemtoReg = 1, se scrie MemData în blocul de registre RF.

Restul elementelor neincluse în cele 3 unități (cf. cu Figura 7. 1) sunt:

- Validarea saltului condiționat prin activarea PCSrc, de la unitatea IFetch, folosind o poartă ȘI între Branch (de la UC) și Zero (de la EX):

  PCSrc ← Branch and Zero;

- Calculul adresei de salt necondiționat Jump Address de la IFetch:

  JumpAddress ← (PC+4)[31:28] || Instruction[25:0] || "00";

- (nu apare pe schemă) Conectarea Read Data 1 la adresa de salt JR Address de la IFetch, dacă se implementează instrucțiunea JR.

7.3. Activități practice

Resurse necesare:

- Proiectul *test_env* cu unitățile IFetch, ID, UC de la lucrarea anterioară.

7.3.1. Elaborarea unității EX

Descrieți o nouă entitate **EX**, conform cu arhitectura prezentată în Figura 7. 3, fără a folosi entități suplimentare:

- Descrieți componenta ALU Control într-un proces cu case după semnalul ALUOp. Fiecare ramură when asociată unui cod de operație ALUOp va determina un cod corespunzător pentru ALUCtrl. În cazul excepțional al instrucțiunilor de tip R, ramura when va conține un alt case după câmpul func.
- Definiți multiplexorul cu selecția ALUSrc folosind un proces sau concurențial cu when-else.
- Definiți unitatea ALU într-un un proces cu case după semnalul ALUCtrl, în care ramurile when vor implementa câte o operație aritmetică sau logică pentru fiecare cod alocat la proiectare. Indicații:
  1. La operațiile de deplasare folosiți operatorii sll, srl sau sra, astfel:

     Result <= to_stdlogicvector(to_bitvector(T_1) sll conv_integer(T_2));

  2. Pentru compararea a 2 semnale cu semn, necesară la instrucțiunile SLT și SLTI, se poate folosi operatorul <, în felul următor:

     if signed(T_1) < signed(T_2) then

  Notă: Includeți doar una din librăriile **IEEE.numeric_std.ALL** sau **IEEE.std_logic_arith.ALL** pentru funcția signed(). Restul funcțiilor de conversie sunt definite în **IEEE.STD_LOGIC_UNSIGNED.ALL**.
- Generați semnalul Zero cu un proces sau concurențial cu when-else.
- Implementați calculul adresei de salt condiționat Branch Address într-o singură linie de cod:

  BranchAddress ← (PC+4) + Ext_Imm[29:0] || "00".

7.3.2. Elaborarea unității MEM

Pentru memoria de date descrieți o nouă entitate MEM, după arhitectura prezentată în Figura 7. 4, fără a folosi entități suplimentare.

- Definiți o memorie RAM de capacitate 64x32 (Anexa 5) în care citirea este asincronă, deci trebuie extrasă în afara procesului sensibil pe clock. Opțional, inițializați memoria cu valori necesare programului de test. Scrierea este pe frontul ascendent și trebuie controlată de același buton, ca în cazul registrului PC și a blocului de registre. Așadar, introduceți o validare suplimentară testării lui MemWrite (folosiți enable).
  Notă: Din motive de economie a resurselor utilizate, memoria de date este redusă la 64 de locații (în loc de 2^{32}), dar poate fi extinsă, la nevoie. Pentru 64 de locații sunt necesari 6 biți de adresă. Deoarece memoria accesează 4 octeți odată (32 de biți) și semnalul $ALURes_{In}$ adresează la nivel de octet, valoarea acestuia va trebui împărțită la 4 prin deplasare la dreapta cu 2 biți. În consecință, se vor utiliza cei 6 biți $ALURes_{In}[7:2]$ ca intrare de adresă și se ignoră biții [1:0].
  Realizați înaintarea ALURes de la intrare spre ieșire (doar o conexiune).

7.3.3. Finalizarea procesorului

Declarați și instanțiați entitățile **EX** și **MEM** în arhitectura entității **test_env** de la lucrarea anterioară, din care eliminați sumatorul pentru RD1+RD2. Conectați-le cu componentele IFetch, ID și UC urmărind schema din Figura 7. 1.

Adăugați multiplexorul pentru unitatea WB. Implementați logica de control PCSrc a saltului condiționat, și de calcul a adresei de salt necondiționat Jump Address. Înlocuiți adresele constante la IFetch cu conexiunile relevante. Finalizați orice alte legături necesare pe calea de date și control.

7.3.4. Specificații de testare

Pentru verificarea semnalelor de control, acestea sunt conectate la led-uri (încă din lucrarea anterioară, Activitatea 6.3.3). Există minim 8 semnale de 1 bit, și semnalul ALUOp pe 2-3 biți, în funcție de necesități.

Multiplexorul MUX 8:1 utilizat pentru afișarea mai multor semnale pe SSD va avea următoarele intrări utile, în funcție de valoarea pe comutatoarele $SW_{7:5}$:

- "000": afișează Instruction (de la IFetch);
- "001": afișează PC+4 (de la IFetch);
- "010": afișează RD1 (de la ID);
- "011": afișează RD2 (de la ID);
- "100": afișează Ext_Imm (de la ID);
- "101": afișează ALURes (de la EX);
- "110": afișează MemData (de la MEM);
- "111": afișează WD (de la ID).

La nevoie, oricare din intrări se poate înlocui cu alte semnale relevante din calea de date și control, precum adresele de salt sau codul ALUCtrl.

Folosiți butoanele de control pentru a testa execuția și resetul. Verificați la fiecare instrucțiune corectitudinea valorilor prezente pe SSD și led-uri. Testați

scrierea corectă în blocul de registre și în memoria de date verificând conținutul locațiilor scrise, atunci când sunt folosite ca operanzi sursă în instrucțiunile următoare (se pot vizualiza pe SSD).

Temă: Realizați trasarea execuției programului notând pentru fiecare instrucțiune curentă valorile semnalelor care ar trebui să apară pe SSD, respectiv Instruction, PC+4, RD1, RD2, Ext_Imm, ALURes, MemData, WD și adresele de salt. Dacă programul conține buclă este suficient să trasați până la finalul primei iterații. Veți verifica execuția pe placă folosind datele de trasare.

7.4. Bibliografie

[1] D. A. Patterson and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
[2] Imagination Technologies LTD. MIPS Architecture for Programmers, Volume I-A: Introduction to the MIPS32 Architecture, Revision 6.01, August, 2014.
[3] Imagination Technologies LTD. MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual, Revision 6.06, December 2016.

Lucrarea 8

8. Procesorul MIPS 32, ciclu unic – Testare

Execuția programului de test pe procesor

8.1. Obiective

Execuția pe pași a programului de test și urmărirea propagării semnalelor pe calea de date și control cu scopul înțelegerii funcționării arhitecturii MIPS și a identificării eventualelor greșeli de implementare sau de descriere în VHDL, pentru corectarea acestora.

8.2. Specificații de testare

În timpul testării procesorului cu programul propriu, verificați corectitudinea semnalelor de control prin vizualizarea acestora pe led-uri, la fiecare pas de execuție. Verificarea se poate face comparând ceea se afișează cu valorile corecte din tabelul realizat la Lucrarea 4, Activitatea 4.3.3.

Concomitent, în funcție de valoarea pe comutatoarele $SW_{7:5}$, se pot vizualiza pe SSD diverse semnale de interes din calea de date, în următoarea schemă de codificare:

- "000": afișează Instruction (de la IFetch);
- "001": afișează PC+4 (de la IFetch);
- "010": afișează RD1 (de la ID);
- "011": afișează RD2 (de la ID);
- "100": afișează Ext_Imm (de la ID);
- "101": afișează ALURes (de la EX);
- "110": afișează MemData (de la MEM);
- "111": afișează WD (de la ID).

La nevoie, semnalele afișate se pot înlocui cu altele. De exemplu, dacă nu funcționează o instrucțiune de salt puteți afișa adresa de salt corespunzătoare.

Folosiți trasarea realizată ca temă la lucrarea anterioară pentru confruntarea valorilor afișate cu cele corecte. Urmăriți rezultatele înscrise pe calea de date, pe parcursul execuției și la final de program, în paralel cu succesiunea corectă a execuției instrucțiunilor, care este evidențiată de valorile afișate pentru PC+4. În cazul în care se ivesc probleme, există o serie de abordări corespunzătoare:

- La instrucțiuni de tip R sau I cu scriere în registru se verifică pe SSD transmiterea valorilor de la instrucțiune către ieșirile blocului de registre (RD1, RD2), imediatul extins (Ex_Imm), rezultatul ALU (ALURes) și valoarea pregătită pentru a fi salvată în blocul de registre (WD).
- Pentru lucrul cu memoria apar ca semnale de interes adresa de memorie (ALURes) și valoarea înscrisă (RD2) sau citită din memorie (MemData).

- Pentru a testa scrierea în blocul de registre sau memorie, se pot vizualiza valorile salvate când sunt folosite ca operanzi sursă în instrucțiunile viitoare. Exemplu:

```
ADDI $3, $0, 7  -- în registrul $3 se încarcă 7
…
ADD $9, $1, $3 -- la acest pas, pe RD2 ar trebui să fie 7, dacă
                -- valoarea nu a fost modificată de altă instrucțiune
```

8.3. Bibliografie

[1] D. A. Patterson and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
[2] Imagination Technologies LTD. MIPS Architecture for Programmers, Volume I-A: Introduction to the MIPS32 Architecture, Revision 6.01, August 2014.
[3] Imagination Technologies LTD. MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual, Revision 6.06, December 2016.

Lucrarea 9

9. Procesorul MIPS 32, pipeline – Proiectare și implementare

Crearea versiunii pipeline din arhitectura cu ciclu unic

9.1. Obiective

Descrierea și implementarea pentru:

- Procesorul MIPS 32 pipeline.

9.2. Specificațiile arhitecturii MIPS pipeline pe 32 de biți

Această activitate pornește de la structura procesorului MIPS, ciclu unic [1] (Figura 9. 1), având ca reper cele 5 etape de execuție a unei instrucțiuni [2][3]: 1) IF (Instruction Fetch); 2) ID/OF (Instruction Decode / Operand Fetch); 3) EX (Execute); 4) MEM (Memory); 5) WB (Write-Back).

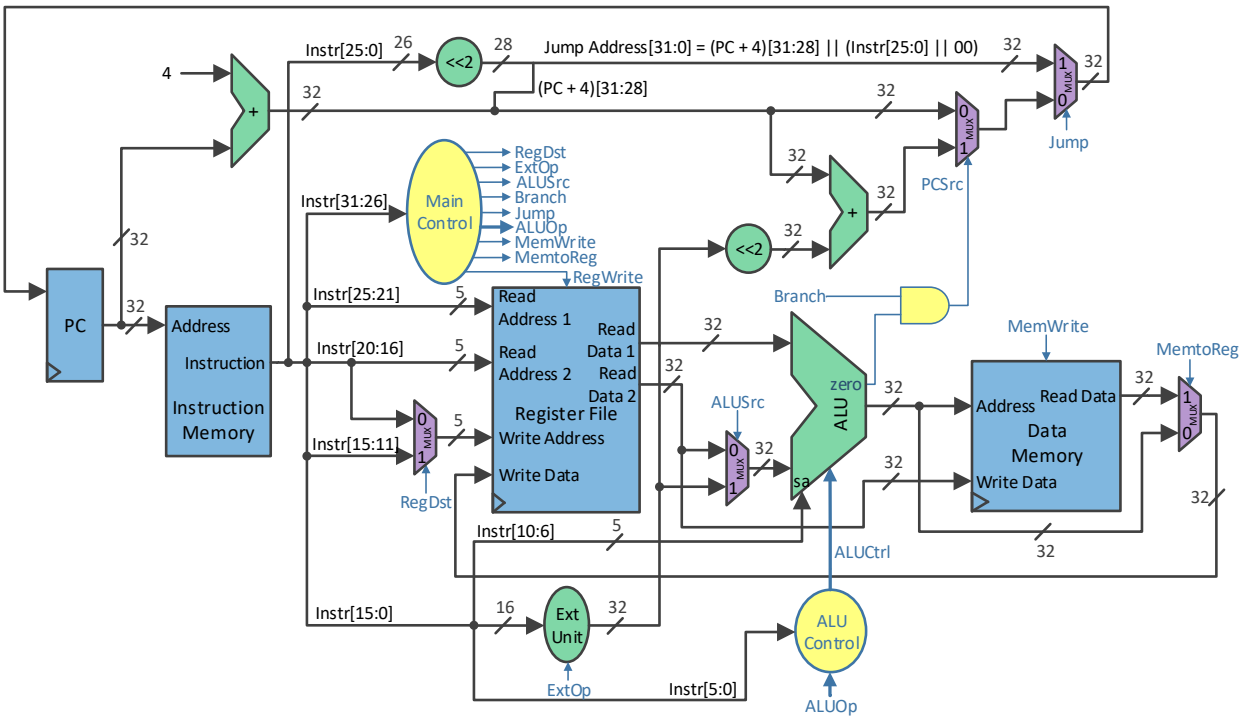


Figura 9. 1 Arhitectura MIPS 32, ciclu-unic

Formatul instrucțiunilor în funcție de categorie [2][3]:

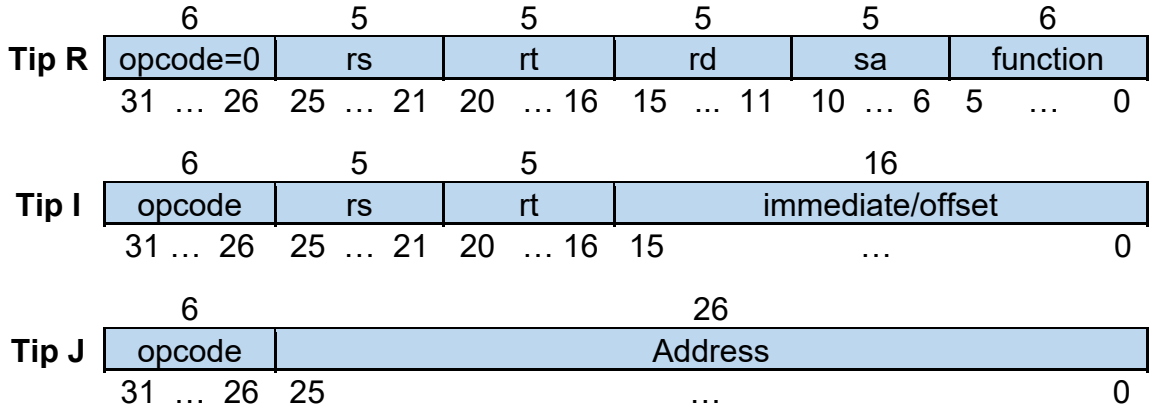


Figura 9. 2 Formatul instrucțiunilor MIPS 32

Execuția lentă la varianta cu ciclu unic este datorată perioadei de ceas, care trebuie să acopere propagarea semnalelor pe calea cea mai lungă (calea critică), întâlnită în cazul instrucțiunii load word (LW). În consecință, restul instrucțiunilor vor suferi aceeași întârziere. Durata perioadei se poate reduce prin separarea diferitelor elemente din calea de date combinațională cu ajutorul unor registre de memorare a rezultatelor intermediare, rezultând arhitectura pipeline din Figura 9. 3, în care **etajele** de lucru (etaje pipeline) corespund celor 5 etape de execuție a instrucțiunii. Registrele pipeline dintre etaje vor fi identificate conform cu etapele de execuție pe care le separă, astfel: IF/ID, ID/EX, EX/MEM, MEM/WB. Ele memorează rezultatele de la etajul anterior și le pun la dispoziție elementelor funcționale din etajul următor. În acest fel, în pipeline se pot executa concomitent până la 5 instrucțiuni consecutive, fiecare în etape diferite. După propagarea prin etajele pipeline instrucțiunile se vor finaliza, una câte una, la fiecare impuls de ceas, astfel că se obține o productivitate medie de 1 instrucțiune / ciclu, cu frecvență de lucru mărită.

Deoarece funcționarea componentelor din calea de date este dirijată de semnalele de control, ele trebuie transmise concomitent cu fluxul de date, de la un etaj la următorul, prin intermediul registrelor pipeline, până la etajul în care își îndeplinesc scopul. Pe schema din Figura 9. 3 se observă faptul că semnalele de control au fost grupate în cadrul registrelor pipeline după numele etajului până la care trebuie propagate.

Urmărind schema din Figura 9. 3, se remarcă faptul că, spre deosebire de varianta cu ciclu unic, la varianta pipeline, stabilirea registrului destinație are loc în etapa de execuție (în etajul EX), deoarece permite implementarea unei tehnici hardware de evitare a hazardurilor. Hazardurile vor fi studiate în lucrarea următoare. Această modificare presupune relocarea multiplexorului cu semnalul de selecție RegDst, din entitatea ID, în entitatea EX, împreună cu toate semnalele conectate pe intrările sale, respectiv câmpurile rt, rd și semnalul de control RegDst. Această relocare a semnalelor necesită salvarea lor în registrul pipeline ID/EX. Cele 2 câmpuri rt și rd vor apărea ca porturi de ieșire ale entității ID. Alături de RegDst, rt și rd vor deveni porturi de intrare ale entității EX. Ieșirea multiplexorului se va propaga până la ultimul etaj WB și va furniza indicele registrului destinație în care se salvează rezultatul.

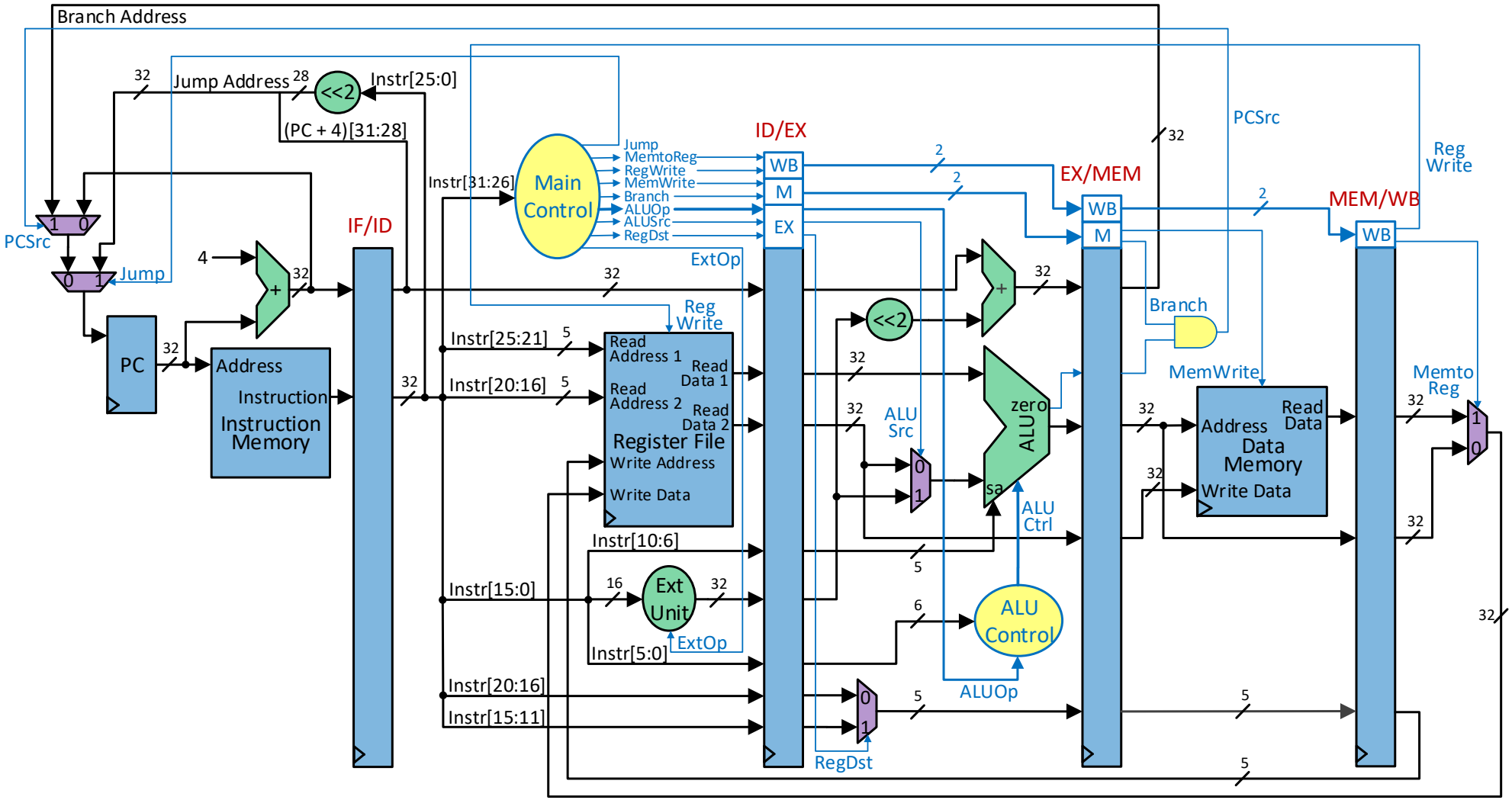


Figura 9. 3 Arhitectura MIPS 32, pipeline

9.3. Activități practice

Resurse necesare:

- Proiectul *test_env* care implementează MIPS 32 ciclu unic.

9.3.1. Verificarea MIPS 32 cu ciclu unic

Implementarea versiunii pipeline trebuie să se bazeze pe o arhitectură cu ciclu unic, care să fie corectă din punct de vedere funcțional. În cazul în care testarea ei nu a fost finalizată se recomandă acest lucru înaintea activităților următoare.

9.3.2. Proiectarea registrelor pipeline

Această activitate se va realiza într-un tabel în care se vor enumera pe coloane semnalele ce trebuie să le stocheze fiecare registru aflat între etajele pipeline. Semnalele se pot extrage din Figura 9. 3 (în funcție de instrucțiunile implementate pot să apară și altele, suplimentare). De exemplu, în registrul IF/ID se vor salva semnalele PC+4 și Instruction. În consecință, numărul de biți necesar pentru registrul IF/ID este 64. Se va proceda similar și cu celelalte registre pipeline. La fiecare registru, vizualizați entitățile pe care le separă și identificați semnalele asociate registrului, în lista de porturi de intrare și ieșire a entităților.

9.3.3. Instanțierea registrelor pipeline în VHDL

Registrele pipeline vor fi implementate în arhitectura entității **test_env**, alături de celelalte unități principale din arhitectura cu ciclu unic. Ele vor fi descrise prin proces(e) și nu instanțiate ca entități. Fiecare registru va fi declarat ca un semnal de dimensiune corespunzătoare, conform cu Activitatea 9.3.2, iar transferul datelor în acesta se va face controlat prin MPG, de același buton ca în cazul registrului PC, și sincron pe frontul de ceas. Butonul de control va facilita testarea pe pași a execuției în pipeline. De exemplu, pentru registrul IF/ID se poate declara semnalul REG_IF_ID pe 64 de biți, iar descrierea acestuia în cadrul procesului va avea următoarea formă:

dacă front de ceas și enable atunci
REG_IF_ID(31 downto 0) <= PC+4;
REG_IF_ID(63 donwto 32) <= Instruction;

În acest caz, semnalele PC+4 și Instruction provin de la entitatea **IFetch**, iar $REG\_IF\_ID_{31:0}$ și $REG\_IF\_ID_{63:32}$ vor fi conectate la porturile de intrare PC+4, respectiv Instruction, ale entității **ID**.

Alternativ, fiecare registru se poate înlocui cu semnale individuale, corespunzătoare câmpurilor stocate în registru. De exemplu, în loc REG_IF_ID pe 64 de biți se pot declara semnalele PC_IF_ID și Instruction_IF_ID, fiecare pe 32 de biți (pentru unicitate, s-au inclus etajele adiacente în denumirea semnalelor). În acest caz, descrierea în cadrul procesului va fi următoarea:

dacă front de ceas și enable atunci
PC_IF_ID <= PC+4;
Instruction_IF_ID <= Instruction;

Atât PC_IF_ID cât și Instruction_IF_ID vor fi conectate mai departe la porturile de intrare PC+4, respectiv Instruction, ale entității **ID**.

Observație: Procesul de transformare a arhitecturii cu ciclu unic presupune, în paralel cu implementarea registrelor pipeline și anumite modificări aduse entităților principale. Astfel, în conformitate cu Figura 9. 3, se poate observa faptul că la entitatea **ID** este necesar un nou port de intrare pentru adresa de scriere în blocul de registre, deoarece va fi pusă la dispoziție de registrul MEM/WB. În consecință, se remarcă faptul că, în pipeline, unitatea care *urmează* este dictată de secvența fluxului de date și control (nu e neapărat situată la dreapta). Un alt exemplu este semnalul de control RegWrite pentru scrierea în blocul de registre. Acesta trebuie propagat până în etajul WB, concomitent cu fluxul de date și apoi se întoarce la intrarea corespunzătoare în blocul de registre.

Descrieți în VHDL toate conexiunile necesare, de la entități la registre, de la registre la entități sau între registre, în funcție de caz.

9.3.4. Arhitectura procesorului pipeline personalizat

Temă: Având ca model Figura 9. 3, desenați arhitectura procesorului pipeline, asigurându-vă că includeți toate componentele necesare astfel încât cele 15 instrucțiuni să se execute corect.

9.4. Bibliografie

[1] D. A. Patterson and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
[2] Imagination Technologies LTD. MIPS Architecture for Programmers, Volume I-A: Introduction to the MIPS32 Architecture, Revision 6.01, August 2014.
[3] Imagination Technologies LTD. MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual, Revision 6.06, December 2016.

Lucrarea 10

10. Procesorul MIPS 32, pipeline – Rezolvarea hazardurilor

Detectarea hazardurilor și eliminarea lor prin modificarea programului

10.1. Obiective

Se urmărește aprofundarea următoarelor cunoștințe:

- Studiul categoriilor de hazarduri și a soluțiilor de eliminare a acestora;
- Modificarea programului din memoria de instrucțiuni într-o variantă care evită hazardurile detectate, prin inserare de instrucțiuni NoOp;
- Testarea programului rezultat pe arhitectura pipeline.

10.2. De la MIPS 32 cu ciclu unic la arhitectura pipeline

Deoarece la arhitectura cu ciclu unic execuția lentă a instrucțiunii LW este cea care stabilește perioada de ceas pentru celelalte instrucțiuni, se recurge la secționarea căii de date, prin intercalarea de registre pipeline (Figura 10. 1) [1], conform celor 5 faze de execuție a unei instrucțiuni [2][3]: 1) IF (Instruction Fetch); 2) ID/OF (Instruction Decode / Operand Fetch); 3) EX (Execute); 4) MEM (Memory); 5) WB (Write-Back).

Registrele pipeline preiau rezultatele de la un etaj și le pun la dispoziție către etajul următor. În consecință, calea combinațională dintre 2 elemente de stocare (memorie sau registre) se scurtează, ceea ce permite reducerea perioadei de ceas și prelucrarea datelor la o frecvență mai ridicată. De asemenea, compartimentarea pe etaje pipeline facilitează încărcarea și executarea în paralel a 5 instrucțiuni, fiecare într-o fază de execuție diferită. Astfel, ulterior încărcării arhitecturii pipeline cu primele 5 instrucțiuni, la fiecare ciclu de ceas se va finaliza o instrucțiune, în ordinea de execuție. Finalizarea unei instrucțiuni coincide cu încărcarea unei noi instrucțiuni, în timp ce alte 4 se află în diverse faze de execuție.

Notă: Chiar dacă unele instrucțiuni își încheie efectul mai devreme, în etajele primare, toate instrucțiunile parcurg integral cele 5 etaje.

10.3. Tipuri de hazarduri specifice arhitecturii MIPS pipeline

În pofida avantajelor oferite de execuția pipeline, există și situații conflictuale ce pot apărea pentru anumite secvențe de instrucțiuni, rezultatele fiind altele decât cele așteptate. Astfel de situații poartă denumirea generică de *hazarduri* și în funcție de contextul în care apar, sunt împărțite pe 3 categorii:

1. *Hazard structural* – apare când instrucțiuni diferite folosesc simultan aceeași unitate hardware în scopuri diferite.
2. *Hazard de date* – apare când o instrucțiune accesează operanzi în curs de prelucrare în alte etaje pipeline.

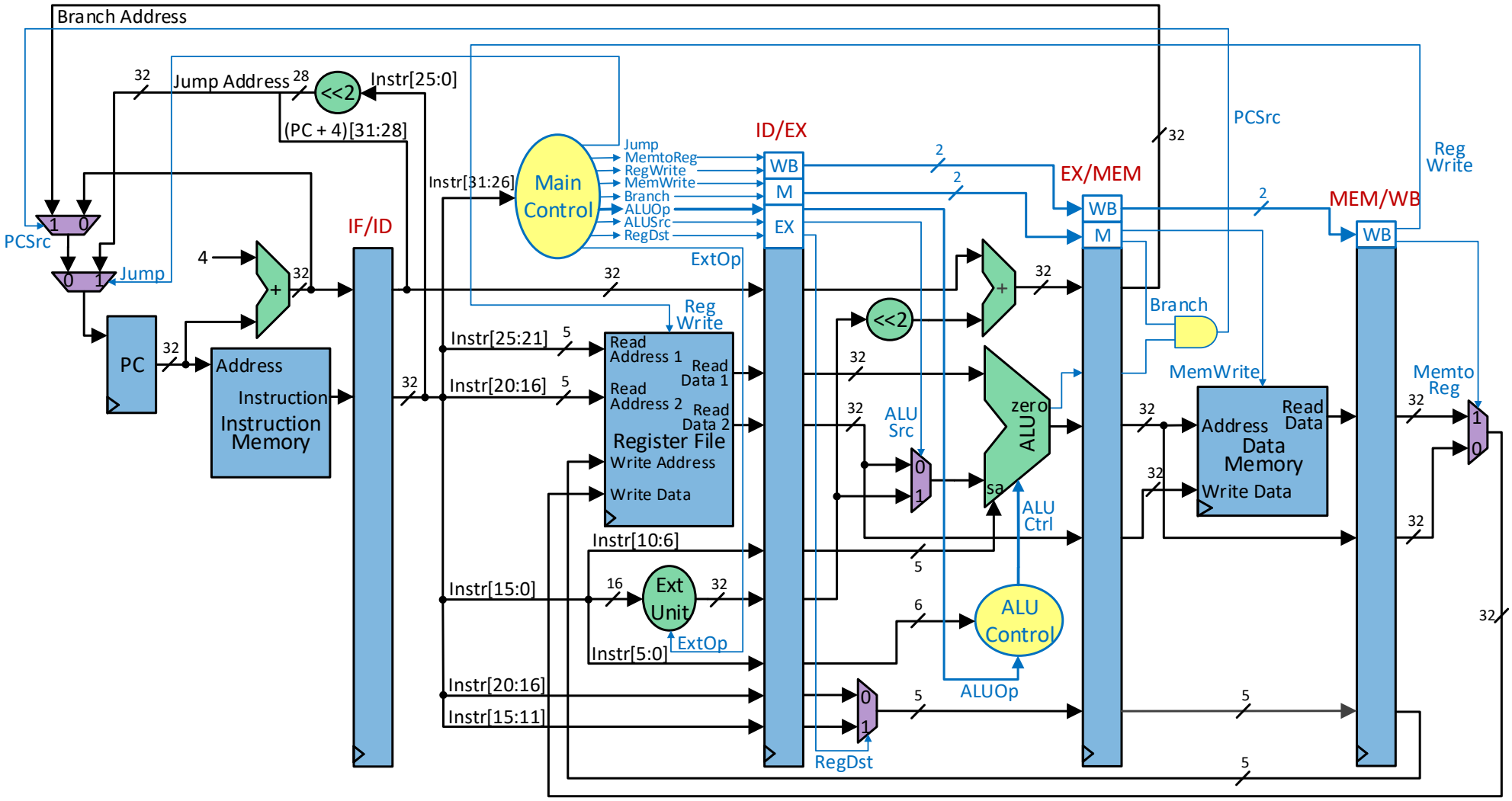


Figura 10. 1 Arhitectura MIPS 32, pipeline

3. *Hazard de control* – apare la instrucțiuni de branch sau jump, fiindcă decizia și adresa de salt sunt cunoscute cu întârziere, în etaje superioare etajului IF. În consecință, până la momentul saltului, etajele inferioare se încarcă cu instrucțiuni care pot să nu facă parte din fluxul de execuție corect.

Există mai multe soluții software și hardware pentru eliminarea hazardurilor. În cadrul lucrării este abordată soluția software prin inserare de instrucțiuni NoOp în program. NoOp (No Operation) este o pseudo-instrucțiune care nu afectează elementele de stare ale procesorului (registre și memorii), exceptând incrementarea registrului PC. Ea poate fi implementată în mai multe forme, astfel: a) SLL $0, $0, 0; b) ADD $0, $0, $0 sau c) ORI $0, $0, 0 etc.

10.3.1. Hazardul structural

Definit ca un conflict datorat accesului simultan la aceeași resursă, hazardul structural poate fi identificat între instrucțiuni aflate la distanță 3 (*d3*), una de cealaltă. Hazardul apare când o instrucțiune salvează rezultatul într-un registru folosit ulterior ca operand sursă de a 3-a instrucțiune după ea. O astfel de situație este evidențiată în secvența următoare, între instrucțiunile LW și SUB, pe registrul $1:

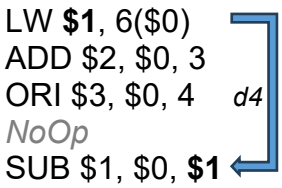
```
LW $1, 6($0)
ADD $2, $0, 3
ORI $3, $0, 4
SUB $1, $0, $1
```

Diagrama de execuție pipeline corespunzătoare secvenței (redată în continuare) facilitează identificarea hazardului prin evidențierea suprapunerii, în ciclul C_5, a scrierii registrului, cu citirea acestuia. Dacă scrierea efectuată de instrucțiunea LW, în blocul de registre, are loc pe frontul ascendent, la finalul ciclului C_5, atunci instrucțiunea SUB va citi în C_5 valoarea neactualizată a registrului și o va utiliza mai departe în calculele din ciclul C_6. Din punct de vedere al execuției programului, această nesincronizare va genera rezultate incorecte.

Instrucțiune\Ciclu	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
LW $1, 6($0)	IF	ID	EX	MEM	WB($1)			
ADD $2, $0, 3		IF	ID	EX	MEM	WB		
ORI $3, $0, 4			IF	ID	EX	MEM	WB	
SUB $1, $0, $1				IF	ID($1)	EX	MEM	WB

Pentru rezolvarea hazardului structural există 2 soluții alternative:

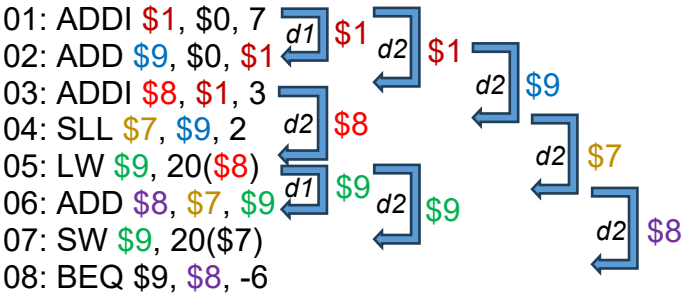
1. **Se recomandă** modificarea scrierii în blocul de registre RF pe frontul descendent (în VHDL, se testează falling_edge(clk) sau clk'event and clk='0'), astfel încât valoarea actualizată după scriere să fie citită asincron în partea a 2-a a perioadei de ceas, înainte de următorul front ascendent.
2. Se introduce o instrucțiune NoOp suplimentară, între instrucțiunile cu hazard, pe oricare din poziții (distanța de 3 va extinsă la 4):



Observație: Deoarece a 2-a soluție este mai costisitoare din punct de vedere al timpului de execuție, în continuare se va presupune implementarea scrierii în blocul de registre pe frontul descendent.

10.3.2. Hazardul de date

Hazardul de date apare la instrucțiuni care, la momentul citirii operanzilor în etajul ID, folosesc registre a căror valoare este în curs de prelucrare (în etaje pipeline superioare). Contextul în care apare acest hazard este definit de o instrucțiune de scriere într-un registru, urmată de una de citire a aceluiași registru. Dacă instrucțiunea care scrie este de load din memorie (LW), atunci i se mai spune *Load Data Hazard*, iar dacă este una aritmetică-logică, atunci se mai numește hazard *Read After Write*. **Notă**: Prin analogie, hazardul structural definit în capitolul anterior este un hazard de date, însă datorită acțiunilor simultane de scriere și citire a blocului de registre, va fi considerat hazard structural, ușor de identificat fiindcă apare la distanță 3. Atunci când distanța dintre registre este 1 (*d1*) sau 2 (*d2*), se va considera hazard de date. Pentru exemplificare, se va analiza următoarea secvență de program, care evidențiază majoritatea situațiilor de hazard de date posibile:



În diagrama de execuție pipeline redată mai jos, hazardurile sunt identificate analizând instrucțiunile în ordinea de apariție în cadrul programului. La fiecare hazard, registrele implicate sunt marcate cu culori distincte, evidențiind astfel scrierea târzie în etajul WB, ulterior citirii în etajul ID.

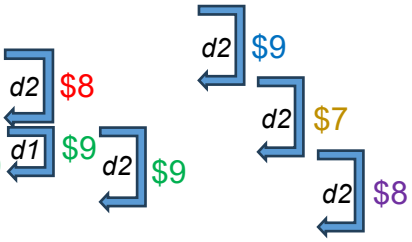
Instrucțiune\Ciclu	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}
01: ADDI $1, $0, 7	IF	ID	EX	MEM	WB($1)							
02: ADD $9, $0, $1		IF	ID($1)	EX	MEM	WB($9)						
03: ADDI $8, $1, $3			IF	ID($1)	EX	MEM	WB($8)					
04: SLL $7, $9, 2				IF	ID($9)	EX	MEM	WB($7)				
05: LW $9, 20($8)					IF	ID($8)	EX	MEM	WB($9)			
06: ADD $8, $7, $9						IF	ID($7,$9)	EX	MEM	WB($8)		
07: SW $9, 20($7)							IF	ID($9)	EX	MEM	WB	
08: BEQ $9, $8, -6								IF	ID($8)	EX	MEM	WB

Important: Nu este necesară identificarea și rezolvarea hazardurilor structurale și de date între instrucțiuni separate de jump sau branch.

Odată cu identificarea hazardurilor se pot aplica și soluții de rezolvare, prin introducerea de NoOp, încât distanțele să fie extinse la 3, adică forțarea către un hazard structural, care este rezolvat de scrierea pe frontul descendent în blocul de registre. De exemplu, hazardul de date dintre instrucțiunea 01 (ADDI) și 02 (ADD) pe registrul $1 poate fi eliminat prin inserare de 2 instrucțiuni NoOp între acestea, pentru a extinde distanța dintre ele, de la 1 la 3. După inserare, toate instrucțiunile următoare se renumerotează și se decalează în diagrama pipeline, cu 2 cicluri de ceas la dreapta. În consecință, distanța dintre instrucțiunile 01 (ADDI) și 03 (ADDI) va fi extinsă, la rândul ei, de la 2 la 4, eliminând hazardul existent între ele pe registrul $1. **Observație**: În unele cazuri, rezolvarea unui hazard cu inserare de NoOp poate anula și alte hazarduri aflate în proximitate.

Situația rezultată după inserarea a 2 instrucțiuni NoOp între 01 și 02 arată în felul următor și exemplifică eliminarea celor 2 hazarduri pe $1:

```
01: ADDI $1, $0, 7
02: NoOp
03: NoOp
04: ADD $9, $0, $1
05: ADDI $8, $1, 3
06: SLL $7, $9, 2
07: LW $9, 20($8)
08: ADD $8, $7, $9
09: SW $9, 20($7)
10: BEQ $9, $8, -6
```



Se continuă analiza cu hazardul dintre instrucțiunile 04 (ADD) și 06 (SLL) pe registrul $9. În acest caz, este necesar un singur NoOp între 04 și 05 sau între 05 și 06. Se va alege varianta secundară (mai avantajoasă), deoarece inserarea unui NoOp între 05 și 06 elimină și hazardul de date dintre 05 (ADDI) și 07 (LW) pe registrul $8. Rezolvarea hazardurilor avansează în mod similar, până la ultima instrucțiune. La final se obține următoarea secvență corespunzătoare eliminării hazardurilor de date cu un număr redus de instrucțiuni NoOp:

```
01: ADDI $1, $0, 7
02: NoOp
03: NoOp
04: ADD $9, $0, $1
05: ADDI $8, $1, 3
06: NoOp
07: SLL $7, $9, 2
08: LW $9, 20($8)
09: NoOp
10: NoOp
11: ADD $8, $7, $9
12: SW $9, 20($7)
13: NoOp
14: BEQ $9, $8, -10
```

Important: Inserarea de NoOp afectează poziția instrucțiunilor în cadrul programului, motiv pentru care trebuie analizate (și eventual actualizate) toate

adresele de jump și toate offset-urile de branch, în conformitate cu noile poziții. De exemplu, offset-ul de la BEQ $9, $8, **-6** a fost corectat la **-10** pentru ca saltul să păstreze locația corectă.

Notă: În cazul în care scrierea în blocul de registre se face pe frontul ascendent, este necesară extinderea distanțelor dintre instrucțiunile cu hazard, la minim 4, prin inserare de NoOp suplimentare.

10.3.3. Hazardul de control

Apariția acestui hazard este indusă de prezența instrucțiunilor de salt, deoarece se întârzie modificarea registrului PC, până în etajul ID, pentru salturi necondiționate (jump), respectiv până în etajul MEM, pentru salturi condiționate (branch). La momentul saltului, prima instrucțiune, respectiv primele 3 instrucțiuni, care se succed celor de salt, vor fi de asemenea încărcate în pipeline, ceea ce poate să nu corespundă cu fluxul de execuție descris în cadrul programului.

Soluția pentru rezolvarea hazardului de control constă în introducerea unui număr adecvat de NoOp, imediat după instrucțiunile de salt, încât execuția lor să asigure o întârziere suficientă, care nu afectează starea procesorului. Vor fi necesare: 1 instrucțiune NoOp după fiecare jump, respectiv 3 instrucțiuni NoOp după fiecare branch.

Observație: O optimizare în cazul unui salt necondiționat poate fi renunțarea la NoOp și interschimbarea instrucțiunii de jump cu instrucțiunea anterioară, dacă aceasta îndeplinește simultan următoarele condiții:

1. Nu este de salt;
2. Nu se face salt la ea în cadrul programului;
3. Nu va genera hazard cu instrucțiunile de la adresa de salt.

Notă: În cazul secvenței de program de la Secțiunea 10.3.2 instrucțiunea BEQ de la final introduce un hazard de control, care se va elimina cu 3 instrucțiuni NoOp inserate după aceasta, astfel:

```
. . .
14: BEQ $9, $8, -10
15: NoOp
16: NoOp
17: NoOp
```

Diagrama de execuție pipeline corespunzătoare evidențiază efectul instrucțiunilor NoOp inserate după BEQ.

Instrucțiune\Ciclu	. . .	C_{14}	C_{15}	C_{16}	C_{17}	C_{18}	C_{19}	C_{20}	C_{21}	C_{22}
14: BEQ $9, $8, -10		IF	ID	EX	MEM	WB				
15: NoOp			*IF*	*ID*	*EX*	*MEM*	*WB*			
16: NoOp				*IF*	*ID*	*EX*	*MEM*	*WB*		
17: NoOp					*IF*	*ID*	*EX*	*MEM*	*WB*	
??: Instr						IF	ID	EX	MEM	WB

Un alt exemplu, la bucla din secvența următoare se pot identifica 2 hazarduri de control generate de instrucțiunile BEQ și J:

20: **BEQ** $1, $0, 3 ≡ **hazard de control**
21: ADDI $1, $1, -1
22: SW $3, 4($2)
23: **J** 20 ≡ **hazard de control**

Rezolvarea hazardului pentru BEQ va necesita 3 instrucțiuni NoOp, iar în cazul instrucțiunii J se poate realiza schimbul avantajos cu instrucțiunea SW anterioară. Soluția fără hazarduri va avea următoarea formă:

20: BEQ $1, $0, **6** → offset-ul a fost recalculat
21: NoOp
22: NoOp
23: NoOp
24: ADDI $1, $1, -1
25: J 2 → adresa de salt nu este afectată de cele 3 NoOp
26: SW $3, 4($2)

10.4. Activități practice

Resurse necesare:

- Proiectul *test_env* cu implementarea MIPS 32, pipeline.

10.4.1. Determinarea hazardurilor și rezolvarea lor în programul de test

Pași de urmat: 1) Desenați diagrama de execuție pipeline pentru întreg programul. 2) Determinați hazardurile și eliminați-le inserând instrucțiuni NoOp, unde este cazul. 3) Desenați noua diagramă de execuție pipeline după inserarea de instrucțiuni NoOp. 4) La final, modificați adresele de salt și offset-urile, în conformitate cu repoziționarea instrucțiunilor.

10.4.2. Testarea programului pe procesor

Convertiți în cod-mașină programul corectat de la Activitatea 10.4.1 și actualizați memoria de instrucțiuni. Încărcați arhitectura pipeline pe placă și testați execuția programului. Verificați dacă rezultatele finale sunt corecte, în caz contrar realizați o trasare a execuției. Folosiți afișarea pe SSD a semnalelor de interes din calea de date. Verificați cu atenție etajele de la care provin aceste semnale, deoarece fiecare etaj execută o altă instrucțiune. Țineți cont de faptul că în pipeline se execută simultan 5 instrucțiuni consecutive, la fiecare pas. Identificați eventuale erori și corectați-le în VHDL, apoi reluați testarea pe placă.

10.5. Bibliografie

[1] D. A. Patterson and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
[2] Imagination Technologies LTD. MIPS Architecture for Programmers, Volume I-A: Introduction to the MIPS32 Architecture, Revision 6.01, August 2014.
[3] Imagination Technologies LTD. MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual, Revision 6.06, December 2016.

Lucrarea 11

11. Interfațare serială cu periferice – Transmisia

Implementarea transmisiei seriale folosind automate cu stări finite

11.1. Obiective

Descrierea și implementarea pentru:

- Unități de control folosind automate cu stări finite (Finite State Machine – FSM) [1];
- Transmisia serială către periferice [2].

11.2. Specificații de implementare

11.2.1. Realizarea automatelor cu stări finite

Automatele cu stări finite facilitează realizarea unităților de control. Funcționarea lor poate fi descrisă cu ajutorul grafurilor orientate, în care stările sunt reprezentate de noduri și arcele reprezintă tranzițiile. Automatele au porturi de intrare și de ieșire. Tranzițiile între stări au loc la fiecare impuls de ceas și pot fi condiționate de intrări: *stare_următoare* = F_{st_urm}(*stare_curentă* [, in_1, in_2, in_3, …]). Ieșirile au valori care pot depinde de starea curentă (ieșiri pe stare) sau de tranziția ce va avea loc din starea curentă (ieșiri pe tranziție): $ieșire_i$ = F_i(*stare_curentă* [, in_1, in_2, in_3, …]). Un automat cu ieșirile pe stare se numește automat Moore. În acest caz, ieșirile sunt sincrone deoarece tranziția între stări este sincronă cu semnalul de ceas. Dacă ieșirile depind de tranziții, deci depind de starea curentă și de intrări, atunci avem un automat Mealy. Un automat Mealy are ieșirile asincrone deoarece intrările pot fi asincrone. Există automate hibride care pot avea ambele tipuri de ieșiri.

Unealta de sinteză acceptă trei tipuri de descriere a automatelor în VHDL, cu 1, 2, respectiv 3 procese, care pot fi analizate în exemplele din Anexa 8. Stările automatului sunt declarate ca un tip enumerat, iar pentru stocarea stării curente se folosește un semnal de tipul declarat. În exemplul următor sunt definite 5 stări dintre care prima este implicită (atribuirea este opțională, dar indicată pentru simulare).

```
type state_type is (st1, st2, st3, st4, st5);
signal state: state_type := st1;
```

În cadrul descrierii VHDL stările sunt referite prin numele lor. Aplicația va asocia eficient stările la coduri binare după un proces de optimizare automată.

11.2.2. Protocolul pentru transmisia serială

Unitatea de bază în transmisia serială este bitul. Cum între sursă și destinație nu există sincronizare bazată pe un semnal de ceas comun, fiecare bit

transmis va ocupa canalul de comunicație pentru un interval de timp prestabilit, calculat în funcție de rata de transfer, denumită *baud rate*. Valorile de *baud rate* uzuale pot fi de la 300 la 115200 biți pe secundă (bps) și peste. Având în vedere faptul că sistemele de calcul folosesc o reprezentare bazată pe octeți, este necesar un circuit de conversie de la octeți la șiruri de biți, capabil să implementeze protocolul de comunicare serială. Un astfel de circuit poartă denumirea de Universal Asynchronous Receiver-Transmitter (UART). Conform protocolului, când nu se transmite nimic (starea *idle*) linia de comunicare este menținută la valoarea 1. Pentru fiecare octet de date se va transmite un bit de START (valoarea 0), urmat de biții de date (cei 8 biți ai octetului, deși standardul permite între 5 și 8 biți) și o valoare de STOP (valoarea 1 pe durata a 1, 1.5 sau 2 biți), apoi se revine în *idle* sau se continuă cu transmiterea unui nou octet. Biții de date sunt transmiși, în ordine, de la cel mai puțin semnificativ (lsb) la cel mai semnificativ (msb). Opțional, se poate adăuga și 1 bit de paritate pe poziția cea mai semnificativă, pentru detecția erorilor de transmisie. În concluzie, în lipsa bitului de paritate, cu bitul de START și cu cel de STOP, pentru fiecare octet se transmite o succesiune de 10 biți (secvența marcată cu galben, ordonată cronologic de la stânga la dreapta):

$1_{(idle)}$ … $0_{(START)}$ $D0_{(lsb)}$ D_1 D_2 D_3 D_4 D_5 D_6 $D7_{(msb)}$ $1_{(STOP)}$ … $1_{(idle)}$

În transmisia serială simbolurile (caracterele) sunt înlocuite cu codurile ASCII [3] asociate. Codurile sunt pe 1 octet și se pot consulta în Anexa 9. De exemplu, la un *baud rate* specific de 9600 bps se pot transmite 960 caratere pe secundă. De exemplu, caracterul 'W' are codul hexazecimal $(57)_{16}$, deci valoarea binară $(01010111)_2$, ceea ce înseamnă că pe linia serială se va transmite, următoarea secvență: **$0_{(START)}$ $1_{(lsb)}$ 1 1 0 1 0 1 $0_{(msb)}$ $1_{(STOP)}$**.

La destinație, semnalul primit pe linia de transmisie se eșantionează la o rată mai mare decât *baud rate*, pentru a evita posibile erori (ex. pierderea sau citirea multiplă a aceluiași bit) cauzate de decalaje între semnalele de ceas de la sursă și de la destinație, chiar dacă au frecvențe apropiate. Folosind o frecvență superioară, se poate detecta mijlocul intervalului primului bit (de START), iar restul biților vor fi citiți la viteza de *baud rate*, începând cu acest moment, ceea ce asigură că următorii 9 biți vor fi citiți în jurul mijlocului intervalului alocat lor. În general, se folosește o rată de eșantionare de 16 ori mai mare decât *baud rate*. Concret, fiecare bit este citit de 16 ori, dar întotdeauna se reține ce s-a citit la mijlocul intervalului. Tehnica va fi detaliată în lucrarea următoare.

Conversia de la paralel la serial și vice-versa se realizează cu un registru de deplasare. Funcționarea acestuia se va integra în descrierea comportamentului automatului de stare.

11.3. Activități practice

11.3.1. Elemente necesare comunicației seriale

Instalați aplicația HTerm [4] necesară testării comunicației seriale între stația locală și placa Nexys A7 [2]. Conectați placa la stație și activați alimentarea. Ulterior, porniți aplicația. Înainte de conectarea aplicației la portul serial (prin apăsarea butonului **Connect**) verificați următoarele setări: Baud = 9600 (bps), Data = 8 (biți), Stop = 1 (bit), Parity = None (fără bit de paritate). Ca opțiune de

vizualizare alegeți Ascii (opțional, se pot selecta și alte variante). Apăsați butonul R (refresh) pentru actualizarea listei de porturi seriale. Dacă apar mai multe porturi, verificați lista în timp ce cablul USB și placa sunt deconectate de la stație, dar și după reconectare (realizați refresh de fiecare dată!); utilizați portul care apare după reconectare.

Adăugați în entitatea **test_env** un port de transmisie serială tx (port de ieșire) și unul de recepție serială rx (port de intrare), ambele pe 1 bit. Decomentați în fișierul de constrângeri .xdc, cele 2 rânduri corespunzătoare secțiunii #UART. Dacă nu apar, adăugați următoarele rânduri de legătură a porturilor tx și rx la pinii de comunicație serială RXD, respectiv TXD, ai dispozitivului de interfațare serială prin USB de pe placă (în cazul Nexys A7 pinii poartă denumirile D4, respectiv C4):

```
set_property -dict { PACKAGE_PIN D4    IOSTANDARD LVCMOS33 } [get_ports {tx}];
set_property -dict { PACKAGE_PIN C4    IOSTANDARD LVCMOS33 } [get_ports {rx}];
```

11.3.2. Arhitectura pentru transmisie serială

Modificați arhitectura entității **test_env** încât să nu conțină niciun element. Declarați în cadrul arhitecturii un semnal de activare TX_EN, care va fi implementat să aibă frecvența de 9600 Hz. Știind că frecvența semnalului de ceas al circuitului FPGA este 100 MHz, raportul devine (100 x 10^6) / 9600 = 10416 perioade de ceas pentru o perioadă a TX_EN. Acest semnal se poate genera cu un numărător pe 14 biți, care numără crescător de la 0 la 10415, urmând apoi reluarea buclei. Semnalul TX_EN se va activa numai când numărătorul atinge valoarea 10415, pentru o perioadă de ceas, adică o dată la 10416 perioade.

Creați o entitate separată pentru automatul de transmisie serială cu numele **TX_FSM**. Porturile entității sunt ilustrate în Figura 11. 1.

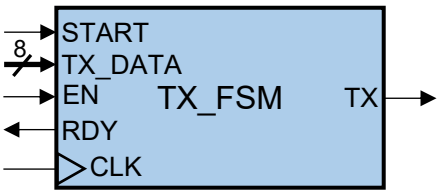


Figura 11. 1 Simbolul automatului de transmisie serială

Funcționarea automatului implementează protocolul de transmisie serială a 8 biți de date, conform diagramei din Figura 11. 2.

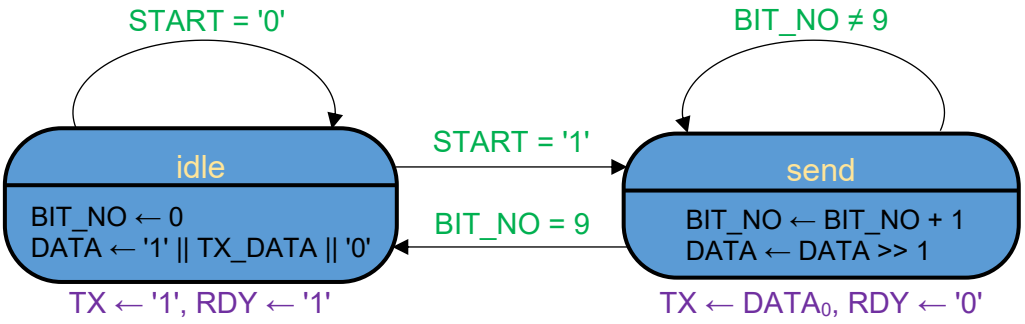


Figura 11. 2 Diagrama automatului de transmisie serială

Pentru a asigura persistența unui bit pe linia de transmisie serială TX în intervalul corespunzător ratei de 9600 bps, tranziția între stările automatului trebuie să se efectueze cu o viteză de 9600 Hz, de aceea intrarea de activare EN va fi conectată la TX_EN. Concret, în cadrul procesului de implementare a tranzițiilor se va testa întâi prezența unui front ascendent al CLK și activarea EN = '1'. Semnalul BIT_NO are comportamentul unui numărător intern al automatului, cu bucla 0-9, care este resetat în starea *idle* și se incrementează în starea *send*. Semnalul DATA conține cei 10 biți de transmis în ordine cronologică și are comportamentul unui registru de deplasare, care este încărcat cu datele în starea *idle* și realizează deplasare la dreapta în starea *send*. Astfel, biții de transmis se vor regăsi întotdeauna pe poziția 0 ($DATA_0$). START este comanda de transmisie a unui octet și RDY este un indicator de disponibilitate. Descrieți comportamentul automatului în VHDL, cu 2 sau 3 procese, conform exemplelor din Anexa 8.

Conectați intrarea TX_DATA la 8 comutatoare ale plăcii și ieșirea TX la portul cu același nume al entității. Comanda START se va activa de la un buton conectat la MPG, însă trebuie să persiste până se intră în starea *send*. Ca o soluție, se va folosi un bistabil D Flip-Flop (DFF) conectat la START. Bistabilul se va activa când ieșirea MPG = '1' și RDY = '1' (RDY indică starea *idle* în care se poate începe o transmisie), altfel se va inactiva când TX_EN = '1', iar în rest își va păstra starea curentă. Diagrama circuitului este ilustrată în Figura 11. 3.

Pentru testarea transmisiei înscrieți coduri ASCII (Anexa 9) pe cele 8 comutatoare, apăsați butonul de pe placă și verificați apariția simbolurilor corespunzătoare în aplicația HTerm, în secțiunea **Received Data**.

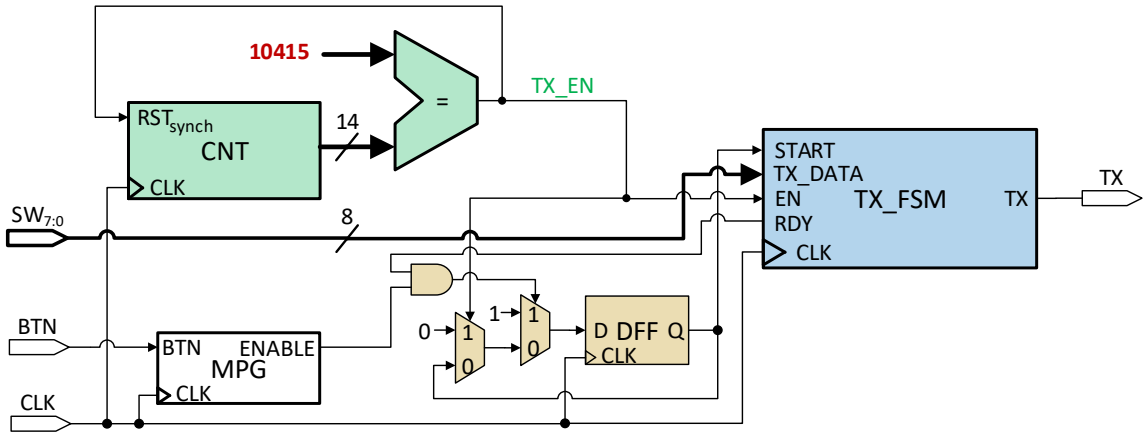


Figura 11. 3 Diagrama circuitului de test

11.4. Bibliografie

[1] Vivado Design Suite User Guide – Synthesis (UG901): FSM Components. Disponibil online: https://docs.amd.com/r/en-US/ug901-vivado-synthesis/FSM-Components

[2] Nexys A7 Reference Manual, Chapter 6: USB-UART Bridge (Serial Port). Disponibil online: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual#usb-uart_bridge_serial_port

[3] ASCII Table. Available online: https://www.asciicharstable.com

[4] HTerm Application. Available online: https://der-hammer.info/pages/terminal.html

Lucrarea 12

12. Interfațare serială cu periferice – Recepția

Implementarea recepției seriale folosind automate cu stări finite

12.1. Obiective

Descrierea și implementarea pentru:

- Recepția serială de la periferice [1].

12.2. Protocolul pentru recepția serială

Protocolul de comunicare serială presupune transmiterea a 5-8 biți de date, precedați de un bit de START (valoarea 0) și urmați de un bit de STOP (valoarea 1). Opțional, la biții de date se poate adăuga și un bit de paritate. Când nu se transmite nimic linia este menținută la valoarea 1. Timpul în care un bit persistă pe linia de comunicație este invers proporțional cu rata de transmisie (*baud rate*), care poate avea valori standard de la 300 la peste 15200 biți pe secundă (bps). Pentru 8 biți de date fără bit de paritate, se transmit 10 biți, în ordine cronologică, după următoarea succesiune: $0_{(START)}$ $D_{0(lsb)}$ D_1 D_2 D_3 D_4 D_5 D_6 $D_{7(msb)}$ $1_{(STOP)}$.

În lipsa unei sincronizări bazate pe un semnal comun de ceas între sursă și destinație, la recepție, linia de comunicație trebuie citită (eșantionată) la o frecvență superioară ratei de transmisie. Dacă s-ar folosi o rată similară cu transmisia, în prezența unor diferențe neglijabile între ele pot apărea decalaje care duc la ratarea sau citirea multiplă a unui bit. Chiar dacă astfel de situații ar apărea cu o frecvență redusă, nu pot fi ignorate (de exemplu, pierderea bitului de START poate atrage după sine evitarea întregului octet care-i urmează).

Pentru evitarea acestui aspect, la recepție se utilizează o supra-eșantionare de 16 ori mai mare decât rata de transmisie. Concret, la fiecare bit se fac 16 citiri, dar valoarea este reținută numai la jumătatea intervalului (la a 8-a citire din cele 16). Prima jumătate de interval este stabilită la a 8-a citire consecutivă a bitului de START. Acesta este și momentul primei valori reținute. Următoarele 9 valori se rețin din 16 în 16 citiri (supra-eșantionări), începând cu acest moment luat ca reper, ceea ce asigură extragerea următorilor 9 biți la mijlocul intervalului lor. Desigur, în prezența unor eventuale desincronizări nu se va respecta mijlocul intervalului pentru cei 9 biți, însă efectele negative pot fi evitate, dacă nivelul de desincronizare se situează sub un prag. De exemplu, la o diferență de *boud rate* între sursă și destinație de sub 1%, prima eroare ar apărea după 100 de biți, deci corectitudinea ar fi asigurată pentru cei 9 biți care urmează bitului de START.

12.3. Activități practice

12.3.1. Arhitectura pentru recepție serială

Continuați să lucrați în arhitectura entității **test_env** cu transmisia serială de la Activitatea 11.3.2 din Lucrarea 11. Implementați semnalul de activare RX_EN

cu frecvența de supra-eșantionare de 16 x 9600 Hz, ceea ce înseamnă că se va activa o dată la (100 x 10^6) / (16 x 9600) = 651 perioade de ceas al FPGA (cu frecvența de 100 MHz). Se va implementa un numărător pe 10 biți, care numără ciclic în bucla 0 – 650. RX_EN se va activa de fiecare dată la atingerea valorii 650, pentru o perioadă de ceas.

Creați entitatea **RX_FSM** pentru automatul de recepție, cu porturile ilustrate în Figura 12. 1.

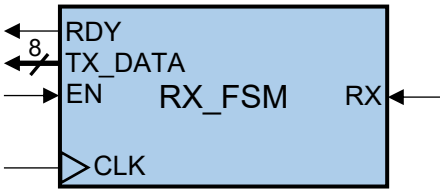


Figura 12. 1 Simbolul automatului de recepție serială

Automatul va avea funcționarea descrisă în diagrama din Figura 12. 2. Tranzițiile între stări se vor efectua la frecvența de 16 x 9600 Hz, motiv pentru care EN se va conecta la RX_EN, iar în cadrul procesului de implementare a tranzițiilor se va testa întotdeauna prezența unui front ascendent al CLK și activarea EN = '1'.

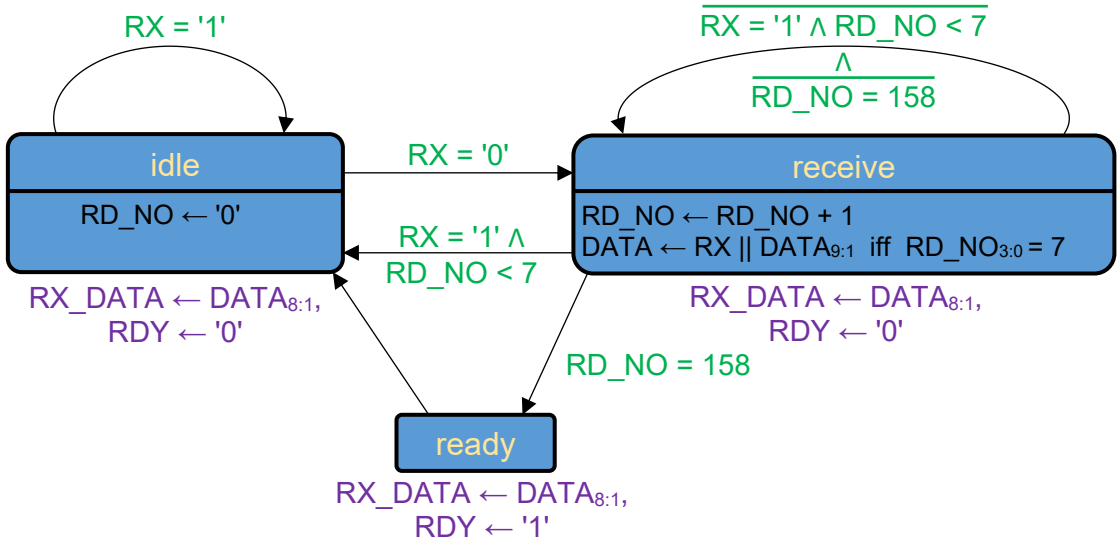


Figura 12. 2 Diagrama automatului de recepție serială

Semnalul RD_NO (8 biți) contorizează intern momentele corespunzătoare eșantionărilor prevăzute în timpul celor 10 biți recepționați la fiecare octet (total: 16 x 10 = 160 eșantionări, numerotate de la 0 la 159). Valoarea sa este resetată în starea *idle*. Incrementarea are loc în starea *receive* corespunzătoare citirii biților de pe linia serială. Starea *ready*, corespunzătoare ultimei eșantionări, are rolul de a semnaliza, prin activarea RDY, faptul că datele citite sunt prezente pe RX_DATA (ultimul bit fiind citit anterior, la jumătatea intervalului său). Incrementarea RD_NO nu se mai realizează în *ready* deoarece ar fi inutilă. Tranziția de la starea *receive* la starea *ready* are loc după eșantionarea cu numărul 158 (penultima), iar din *ready* se revine în starea *idle*, după expirarea ultimului interval de eșantionare.

Stocarea valorilor de pe linia serială RX se realizează în cadrul registrului intern DATA. Conform diagramei din Figura 12. 2, registrul este resetat în starea *idle* și este încărcat în starea *receive*. Momentul încărcării registrului DATA cu câte

un bit de pe RX are loc la jumătatea intervalului rezervat fiecărui bit transmis serial. Este vorba de ce-a de-a 8-a perioadă din cele 16 corespunzătoare eșantionărilor unui bit, adică atunci când RD_NO % 16 = 7 sau echivalent $RD\_NO_{3:0} = 7$.

Registrul DATA va avea 10 biți și încărcarea se va face serial, prin deplasare la dreapta. După 10 încărcări cu valorile de pe linia serială, acestea vor fi amplasate în registru, în ordinea cronologică de apariție, de la bitul cel mai puțin semnificativ la cel mai semnificativ, începând cu bitul de START și terminând cu bitul de STOP (**Notă**: biții cei mai puțin semnificativi fiind primii recepționați serial, vor ajunge, în final, prin deplasare în cadrul registrului DATA, pe pozițiile cele mai puțin semnificative, iar ultimii recepționați, pe pozițiile cele mai seminificative). Valorile celor 8 biți de date din cadrul octetului recepționat serial se vor afla pe biții $DATA_{8:1}$ și vor fi disponibili pe portul de ieșire RX_DATA. Deoarece atribuirea RX_DATA ← $DATA_{8:1}$ este omniprezentă în diagramă, se va realiza în afara oricărui proces de descriere a automatului.

Trecerea din starea *idle* în starea *receive* se realizează la prima întâlnire (eșantionare) a unui bit de START (valoarea 0) pe linia serială. De remacat faptul că bitul de START trebuie să persiste pe RX până la mijlocul intervalului său (până la a 8-a eșantionare) pentru a fi validat, în caz contrar, se revine în starea *idle* și se anulează secvența de citire a întregului octet. O astfel de revenire în *idle* ar corespunde situațiilor în care ar apărea zgomote pe linia serială, caracterizate de prezența unor biți de START falși, de scurtă durată.

Descrieți comportamentul automatului în VHDL, în varianta cu 2 sau 3 procese, conform exemplelor din Anexa 8.

În momentul activării indicatorului RDY al automatului de recepție, stocați valoarea de pe ieșirea RX_DATA într-un registru de 8 biți a cărui valoare se va extinde cu zero la 32 biți pentru afișare pe SSD. La început, registrul va fi inițializat cu 0. Diagrama circuitului de recepție este ilustrată în Figura 12. 3 (inferior).

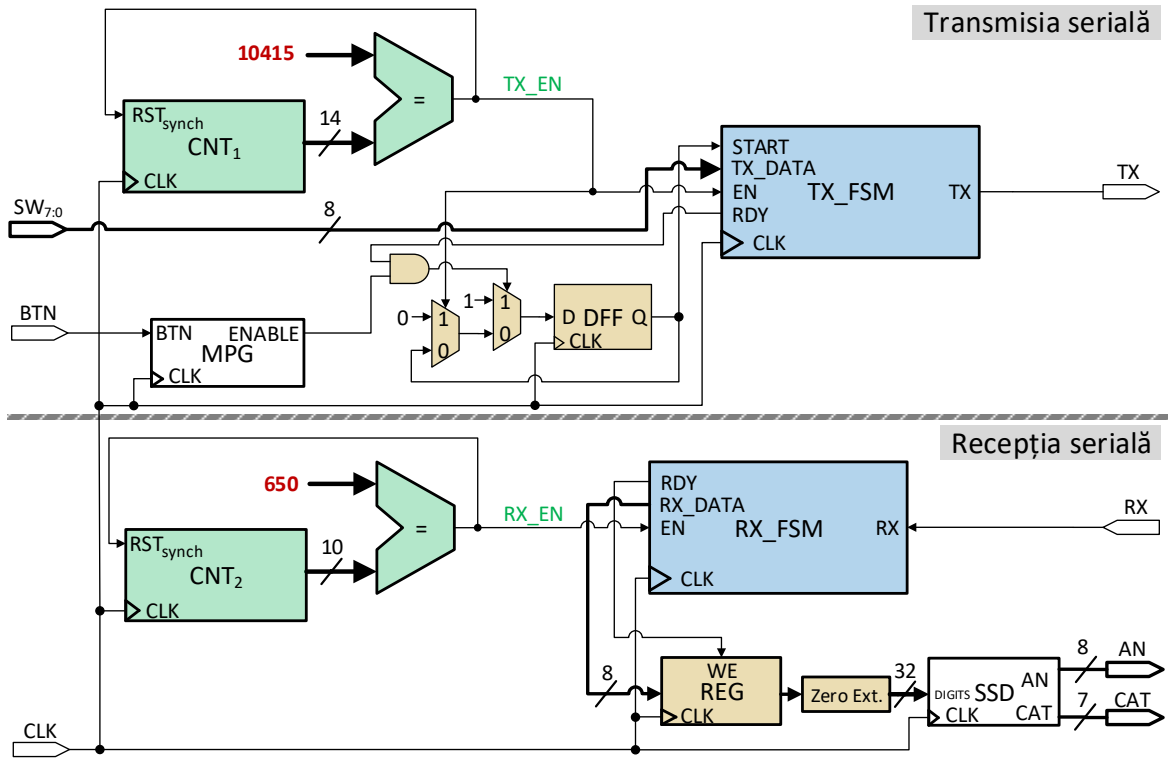


Figura 12. 3 Diagrama circuitului de testare a comunicației seriale bidirecționale

Folosiți fișierul de constrângeri .xdc de la Activitatea 11.3.1 din Lucrarea 11. Pentru testarea recepției seriale utilizați aplicația HTerm [2], configurați-o corespunzător (Baud = 9600, Data = 8, Stop = 1, Parity = None, vizualizare Ascii) și porniți conexiunea cu portul serial (**Observație**: Apăsați butonul R pentru actualizarea listei de porturi seriale. Dacă apar mai multe porturi, verificați lista în timp ce cablul USB și placa sunt deconectate de la stație, dar și după reconectare. Realizați refresh de fiecare dată și utilizați portul care apare după reconectare). Trimiteți caractere ASCII din aplicația HTerm (tastați câte un caracter în secțiunea **Input control** și apoi *Enter*) și verificați apariția corectă pe SSD a codului ASCII [3] (Anexa 9) în hexazecimal. Deoarece arhitectura cuprinde și partea de transmisie serială, testați comunicarea în ambele direcții.

12.4. Bibliografie

[1] Nexys A7 Reference Manual, Chapter 6: USB-UART Bridge (Serial Port). Disponibil online: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual#usb-uart_bridge_serial_port
[2] HTerm Application. Available online: https://der-hammer.info/pages/terminal.html
[3] ASCII Table. Available online: https://www.asciicharstable.com

A. Anexa 1 – Ghid de utilizare Vivado

Notă: Acest ghid corespunde versiunii Vivado 2016.4.

Lansarea utilitarului Vivado

Se poate folosi icoana **Vivado 2016.4** de pe Desktop sau de la meniu: **Start > Xilinx Design Tools > Vivado 2016.4**

Pașii pentru crearea unui proiect

În cadrul aplicației, creați un proiect Vivado, care va fi configurat pentru dispozitivul FPGA prezent pe placa de dezvoltare. Pașii de urmat sunt:

1. Selectați la meniu **File → New Project...** și apăsați **Next**.
2. Introduceți numele proiectului test_env în câmpul Project name, locația în câmpul Project location și asigurați-vă ca opțiunea Create project subdirectory este bifată (pentru a crea directorul test_env la locația precizată), apoi apăsați **Next**.
3. La tipul proiectului selectați RTL Project și bifați Do not specify sources at this time. Apăsați **Next**.
4. Setați proprietățile plăcii, astfel:
   - Product Category: All
   - Family: Artix-7
   - Package: csg324
   - Speed grade: -1

   Selectați modelul xc7a100tcsg324-1 din tabelul afișat, apăsați **Next** și **Finish**.

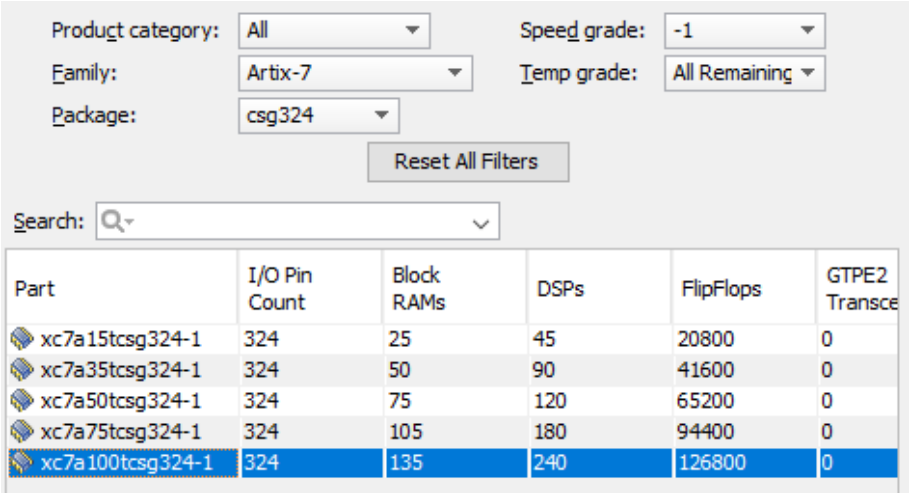


Figura A. 1 Configurarea caracteristicilor plăcii

Modelul se poate schimba ulterior, astfel: accesați la meniu **Tools → Project Settings**, iar la secțiunea **General**, în dreptul Project Device apăsați ... și accesați proprietățile.

Pașii pentru crearea unui fișier VHDL

1. Accesați la meniu **File → Add Sources** sau în panoul **Flow Navigator > Project Manager > Add Sources**.
2. Alegeți **Add or create design sources**, apăsați **Next**.
3. Click pe **Create file** și în noul dialog selectați File type: VHDL, introduceți numele fișierului test_env (va reprezenta și numele entității create), click **OK**.

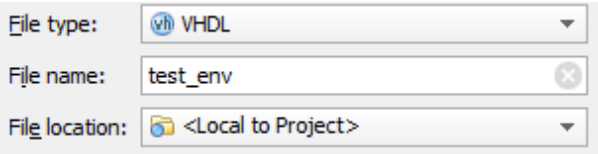


Figura A. 2 Setările pentru crearea unui fișier VHDL

4. Click **Finish**. Se va deschide o fereastră de definire a porturilor entității.
5. Declarați porturile (neapărat cu litere mici!) ca în figura următoare. Aceste porturi sunt suficiente pentru lucrările studiate. Click **OK** pentru a finaliza crearea fișierului. Acesta va conține codul VHDL de declarare a porturilor.
   Notă: Se poate sări peste acest pas deoarece porturile pot fi introduse explicit, prin editarea fișierului după crearea sa.

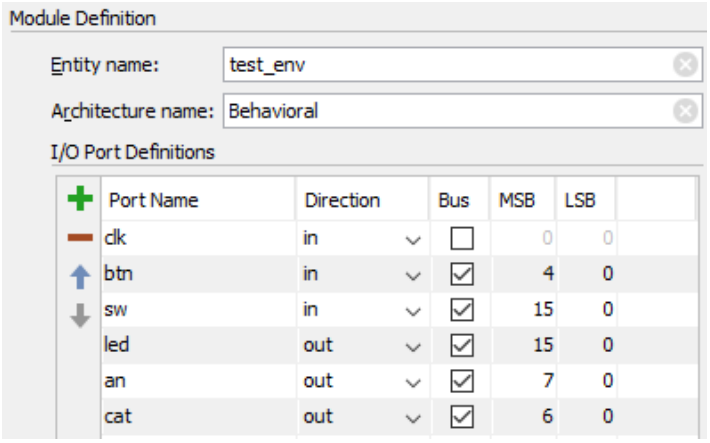


Figura A. 3 Declararea porturilor

Fișierul care conține declararea entității **test_env** și arhitectura ei este afișat automat pentru editare în mediul Vivado (Figura A. 4) sau se poate deschide apăsând dublu click pe **test_env** în ierarhie la panoul **Sources (**tab-ul **Hierarchy)**. Faptul că **test_env** apare cu bold în ierarhie înseamnă că este modulul principal al proiectului (Top Module).

Notă: Dacă sunt mai multe module într-un proiect se poate defini oricare ca Top Module, prin click-dreapta pe numele modulului și opțiunea **Set as Top** (întotdeauna modulul de lucru curent să fie definit Top Module).

În fereastra de editare a entității asigurați-vă că este inclusă librăria std_logic_unsigned, altfel adăugați-o, astfel (valabil pentru orice fișier sursă creat pe viitor!):

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

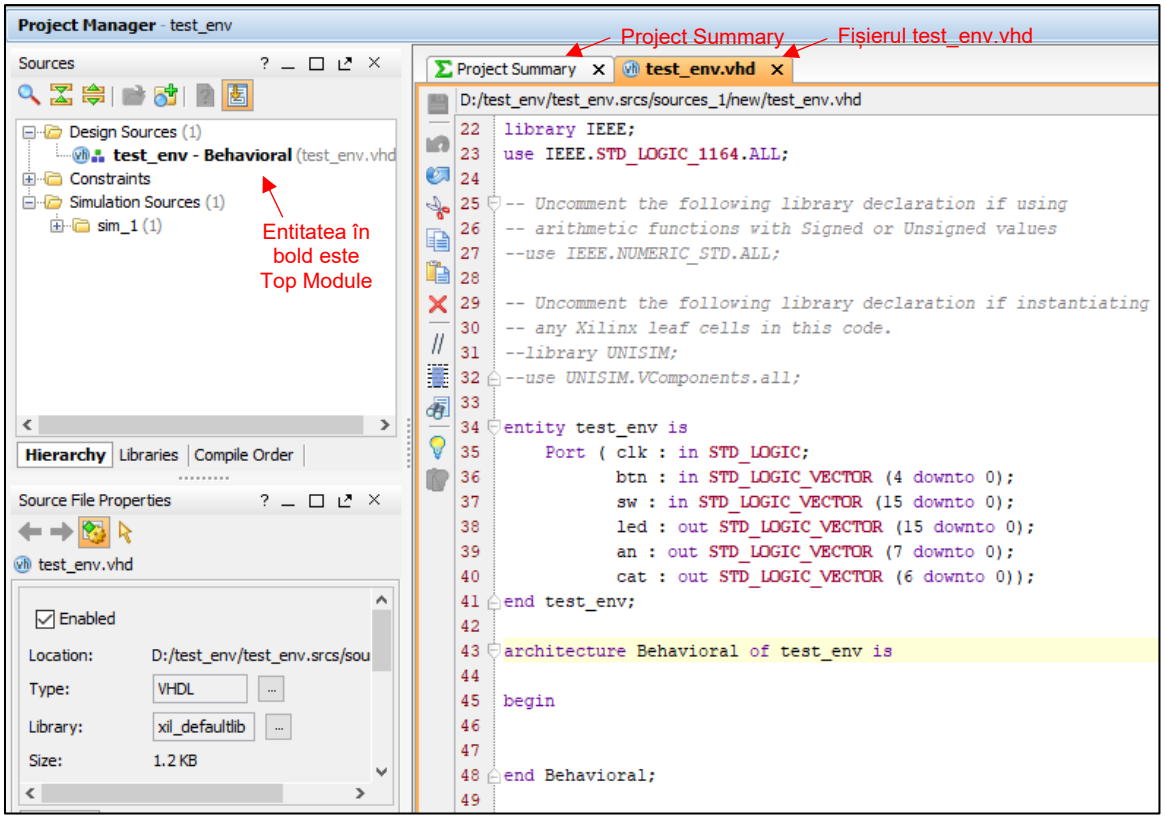


Figura A. 4 Conținutul proiectului *test_env* după creare

Accesarea Language Templates (opțional)

Pentru a facilita descrierea arhitecturii de la pasul următor, se pot folosi șabloanele de cod predefinite (pentru acomodarea cu limbajul VHDL), în felul următor:

1. Deschideți fereastra Language Templates de la meniu: **Tools → Language Templates**. Navigați în ierarhie cu ajutorul simbolurilor ⊞ către categoria dorită: **VHDL > Synthesis Constructs > Coding Examples > …**
2. Selectați componenta și copiați codul în destinație. Închideți Language Templates.
3. Corectați denumirea implicită a semnalelor din codul inserat.

Finalizarea codului și etapa de sinteză

1. Între architecture și begin sunt declarate entitățile integrate în arhitectură și semnalele interne, iar între begin și end sunt definite instanţierile de componente cu *port map*, descrierea comportamentală cu procese și atribuirile concurente.
2. Pentru început, după begin adăugați următoarele atribuiri concurente (pentru a evita erorile, nu folosiți Copy-Paste):

```
led <= sw;                          -- conectează switch-uri la led-uri
an(7 downto 4) <= "1111";           -- dezactivează anozii superiori
an(3 downto 0) <= btn(3 downto 0);  -- conectează anozii inferiori la butoane
cat <= (others=>'0');               -- activează catozii SSD
```

3. Salvați fișierul: **File → Save File** sau **Ctrl+S**.

4. Selectați entitatea **test_env** în ierarhia din panoul **Sources**.
5. Verificați corectitudinea vizualizând schema circuitului, astfel: panoul **Flow Navigator > RTL Analysis > Open Elaborated Design →** (eventual) click **OK**, dacă apar alte ferestre de dialog. Se va deschide schema bloc a circuitului. Cu dublu-click pe unele blocuri-entitate se poate vizualiza structura lor internă. Se poate reveni apăsând butonul **Previous** (situat la stânga, în bara de unelte).

Notă: Problemele de corectat sunt cele menționate ca Errors sau Critical Warnings în panoul **Messages** din partea inferioară. Întotdeauna începeți cu Critical Warnings (deoarece acestea pot elimina multe din erori) și corectați în ordine, de sus în jos. Nu uitați să salvați modificările.

Definirea resurselor utilizate în fișierul de constrângeri

Specificați resursele plăcii asociate cu porturile entității **test_env**:

1. Pentru simplificare, resursele necesare au fost definite în fișierul NexysA7_test_env.xdc [1].
2. Adăugați fișierul la proiect accesând **File → Add Sources** sau în panoul **Flow Navigator > Project Manager > Add Sources**.
3. Alegeți **Add or create constraints**, apoi click pe **Next**.
4. Click pe **Add files →** selectați fișierul de constrângeri pe disc (opțiunea Copy constraints files into project să fie bifată) **→** click **OK →** click **Finish**.

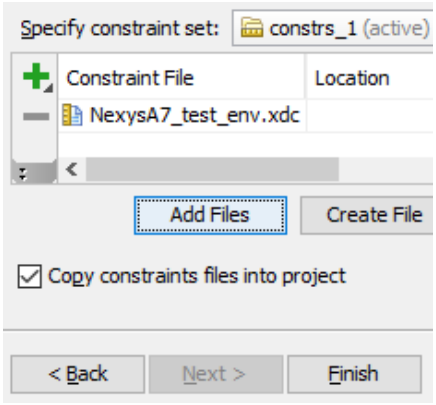


Figura A. 5 Pasul de adăugare a fișierului de constrângeri în cadrul proiectului

5. Fișierul de constrângeri adăugat devine vizibil în ierarhie la **Constraints**.

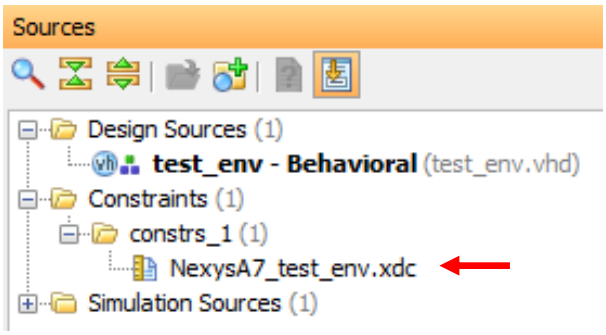


Figura A. 6 Evidențierea fișierului de constrângeri în ierarhie

Apăsați dublu-click pe fișier pentru vizualizarea și modificarea conținutului. **Observație**: Cu excepția semnalului clk celelalte resurse sunt definite printr-un singur rând. Locația resurselor apare în fișier după cuvântul rezervat PACKAGE_PIN, dar și pe placă, între paranteze, în vecinătatea resursei (Figura A. 7).

Figura A. 7 Pe placă, locația resurselor este menționată în imediata vecinătate, între paranteze. De exemplu, butonul inferior are locația P18

Etapele de sinteză, implementare și generare a fișierului de programare

Parcurgeți pe rând pașii următori de la panoul **Flow Navigator** corectând de fiecare dată eventualele probleme raportate la **Messages**:

1. Lansați etapa de sinteză: **Flow Navigator > Run Synthesize** → (eventual) click **OK**, dacă apar alte ferestre de dialog. La final, dacă apare fereastra Synthesis Completed, apăsați **Cancel**.
2. Lansați etapa de implementare: **Flow Navigator > Implementation > Run Implementation** → (eventual) click **OK**, dacă apar alte ferestre de dialog. La final, dacă apare fereastra Implementation Completed, apăsați **Cancel**. Ignorați acele avertismente care fac referire la semnale neconectate, dacă nu sunt necesare în arhitectură.
3. Generați fișierul de programare: **Flow Navigator > Program and debug > Generate Bitstream** → (eventual) click **OK**, dacă apar alte ferestre de dialog. La final, dacă apare fereastra Bitstream Generation Completed, apăsați **Cancel**. Fișierul generat va avea extensia .bit .

Recomandări: Puteți lansa doar ultimul pas și se vor parcurge automat pașii anteriori. De asemenea, pentru simplificare, apariția unor ferestre de dialog se poate evita prin bifarea opțiunii Don't show this dialog again, când este posibil.

Încărcarea fișierului de programare .bit pe placă

1. Verificați conexiunea plăcii cu portul USB și alimentați-o setând comutatorul POWER pe ON. Se va activa led-ul de alimentare.
2. Activați comunicarea aplicației cu placa accesând **Flow Navigator > Program and Debug > Open Hardware Manager > Open Target → Auto Connect**, ca în figura următoare. Odată stabilită comunicarea, se va activa opțiunea **Program Device** (marcată cu un dreptunghi în Figura A. 8).

   Notă: Dacă stabilirea unui canal de comunicare eșuează încercați, în ordine, următoarele soluții: 1) Schimbați cablul; 2) Schimbați portul USB; 3) Reporniți Vivado; 4) Reporniți stația; 5) Schimbați placa.

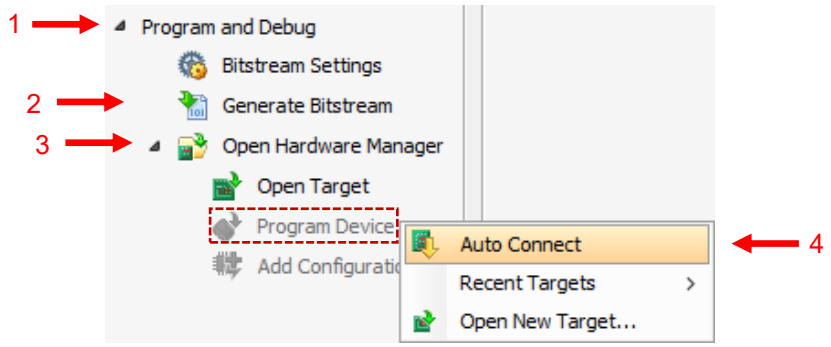


Figura A. 8 Pașii de conectare la placa de dezvoltare

3. Accesați **Program Device**, apoi click pe meniul afișat, selectați calea spre fișierul .bit, dacă nu apare automat (**Notă**: fișierul .bit se generează în directorul_proiectului/test_env.runs/impl_1/) și apăsați **Program**.

După transferul pe placă, realizați testarea fizică a circuitului folosind comutatoarele și butoanele.

Important: Conexiunea rămâne activă cât timp placa este alimentată. Nu opriți alimentarea și astfel, pentru reprogramare (cu orice fișier .bit) va fi suficient să reluați doar pasul 3.

Bibliografie

[1] Fișierul de constrângeri NexysA7_test_env.xdc. Disponibil online: https://drive.google.com/uc?export=download&id=1l8D1splyVyC9mD0l3jAC07gjRgar19Qr

B. Anexa 2 – Simularea funcțională în mediul Vivado

Notă: Acest ghid corespunde versiunii Vivado 2016.4.

Mediul Vivado oferă o modalitate facilă și flexibilă de simulare a funcționalității unei arhitecturi definite în cadrul proiectului dezvoltat. Se va prezenta în continuare simularea funcțională cu **Simulatorul Vivado** și se va aborda ca și studiu de caz simularea entității **test_new**, de la Lucrarea 1, punctul 1.6.4. Este necesar să se rezolve toate erorile de cod VHDL înainte de simulare.

Selectarea arhitecturii de simulat

Accesați **Flow Navigator > Project Manager** și în cadrul ierarhiei alegeți ramura **Simulation Sources > Sim_1**. Dacă aici entitatea **test_new** nu apare cu bold setați-o ca Top Module cu click dreapta pe numele ei și alegeți **Set as Top** (Figura B. 1).

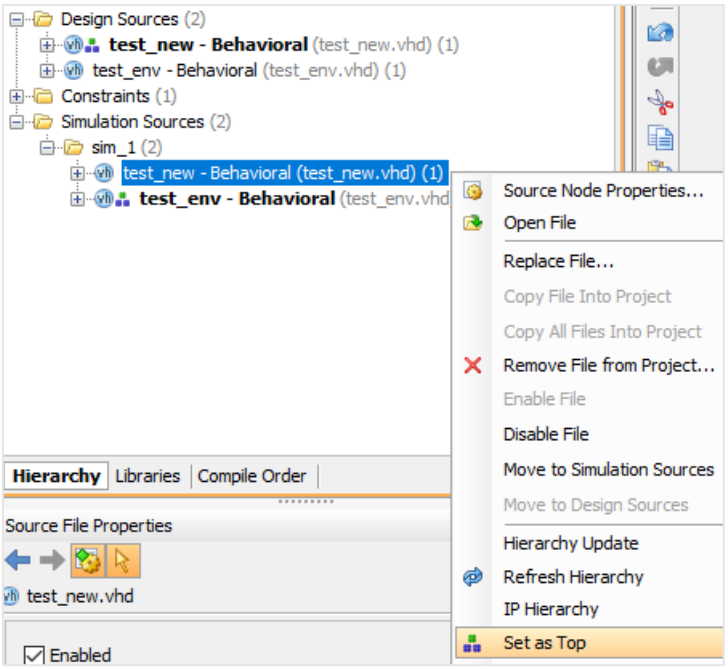


Figura B. 1 Arhitectura simulată trebuie să fie Top Module

Inițializarea simulatorului și elementele constituente

Lansați mediul de simulare astfel: panoul **Flow Navigator > Simulation > Run Simulation → Run Behavioral Simulation**. Va apărea mediul de simulare (Figura B. 2) care conține 3 ferestre de bază, în ordine de la stânga la dreapta: Scopes, Objects și fereastra cu formele de undă, care primește automat un nume (ex. Untitled 1). În ultima fereastră se pot distinge porturile și semnalele din arhitectura **test_new - behavioral**.

În partea superioară apare bara cu butoanele de comandă (Figura B. 2) în care se pot identifica, printre altele, următoarele elemente esențiale de la stânga la dreapta:

- butonul **Restart** – repornește o simulare resetând formele de undă și păstrează stimulatorii cel mai recent definiți pentru semnale (definirea stimulatorilor va fi explicată ulterior). **Notă:** Dacă se dorește repornirea simulării în urma unei modificări în descrierea VHDL, atunci se folosește comanda **Relaunch Simulation** (explicată mai jos);
- butonul **RunFor** – realizează simularea pentru perioada de timp definită la dreapta sa;
- **timpul** unui pas de simulare cu **RunFor**. Schimbați valoarea la 10ms;
- butonul **Relaunch Simulation** – reia simularea cu reanalizarea codului VHDL al fișierelor sursă; în acest caz se pierd stimulii prezenți pe semnale și va fi necesară redefinirea acestora. **Notă**: Orice modificare a codului sursă atrage după sine necesitatea relansării simulării cu **Relaunch Simulation**.

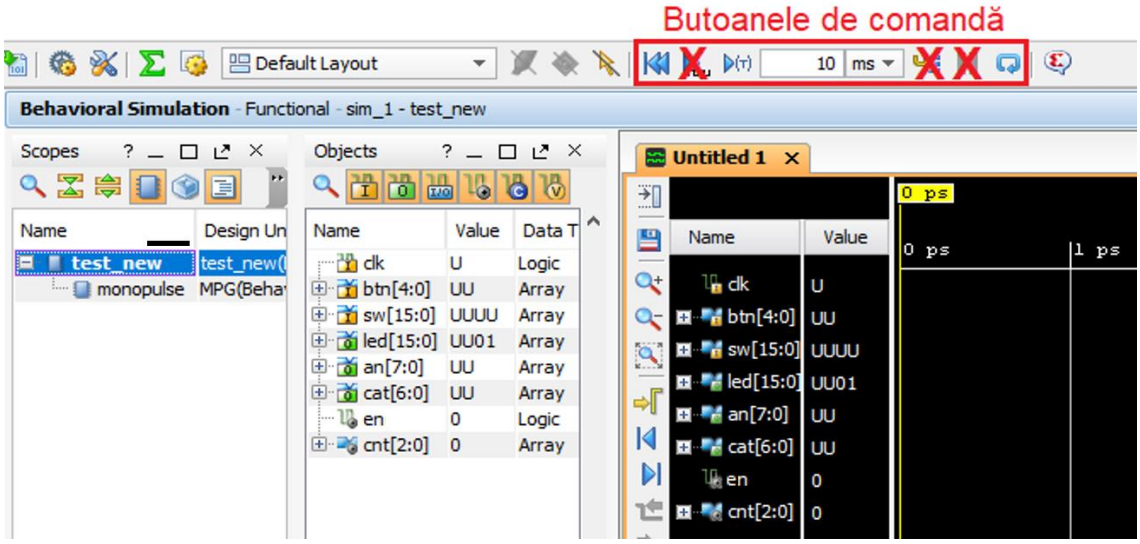


Figura B. 2 Elementele mediului de simulare Vivado

În fereastra Scopes se regăsesc entitatea de top și componentele instanțiate în cadrul arhitecturii sale. Selectând oricare din componente se va reactualiza fereastra vecină Objects cu lista de porturi și semnale pe care le deține.

În fereastra Objects apare lista de porturi și semnale ale componentei selectate în Scopes. În cazul magistralelor se apasă ⊞ în dreptul lor pentru a accesa semnalele interne. De aici se pot adăuga semnale noi în fereastra Untitled 1, cu *drag&drop* în poziția dorită.

În fereastra Untitled 1 apar semnalele care vor fi urmărite pe parcursul simulării prin forme de undă. Cu *drag&drop* se poate schimba ordinea semnalelor. Apăsând click-dreapta pe un semnal, apar opțiuni de copiere (**Copy**), ștergere (**Delete**) sau, în cazul unei magistrale, de setare a bazei de numerație la afișare (**Radix**). Interiorul unei magistrale se poate desfășura apăsând pe ⊞ în dreptul ei.

Configurarea listei de semnale simulate

Semnalele urmărite în timpul simulării sunt listate în fereastra Untitled 1. Pentru simularea curentă se pot șterge din listă magistralele btn[4:0], sw[15:0], an[3:0] și cat[6:0] (click-dreapta pe magistrale → **Delete**). Din fereastra Objects se va adăuga **btn[0]** (desfășurați **btn[4:0]** cu ⊞ → selectați **[0]** → *drag&drop*). Pentru

identificare, redenumiți semnalul adăugat la btn0 (click-dreapta pe semnal → **Rename**). Procedați similar pentru **sw[0]** și redenumiți-l la sw0. Ulterior, selectați componenta *monopulse* în Scopes și adăugați semnalele **cnt_int[17:0]**, **Q1**, **Q2** și **Q3**, din Objects, în Untitled 1 (**Notă**: În cazul de față *monopulse* este eticheta de instanțiere cu *port map* a componentei *MPG*, iar numele poate să difere). Urmăriți să obțineți ordinea din Figura B. 3.

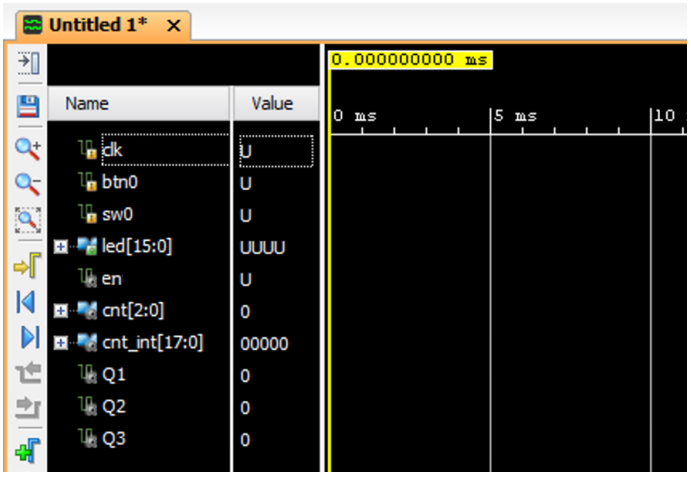


Figura B. 3 Lista semnalelor de urmărit în timpul simulării

Aplicarea stimulilor pe porturi și semnale

Porturile de intrare care necesită stimuli sunt **clk**, **btn[0]** și **sw[0]**:

1. Semnalul de clock al plăcii are frecvența de *100MHz* și perioada de *10ns*. Definiți pentru **clk** un semnal oscilatoriu cu perioada de *10ns* astfel: click-dreapta pe **clk** în Untitled 1 → **Force Clock …**. În fereastra de dialog care apare introduceți valori în următoarele câmpuri (Figura B. 4) și apăsați **OK**:
   - Leading edge value: 0;
   - Trailing edge value: 1;
   - Period: 10ns.

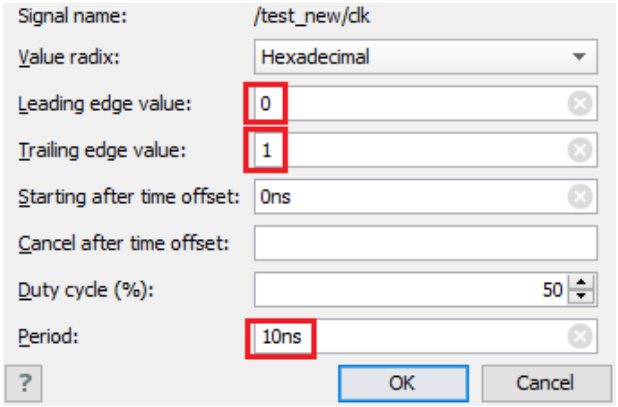


Figura B. 4 Configurarea unui semnal de tact de 10 ns

2. Apăsarea repetată a butonului **btn[0]** de către utilizator poate fi simulată prin asocierea sa cu un semnal oscilatoriu având o frecvență redusă și perioada mai ridicată decât **clk**, de exemplu *10ms*. Definiți pentru **btn0** un

astfel de stimul, repetând aceeași pași ca și pentru **clk**, cu deosebirea că valoarea pentru Period o veți seta la 10ms.

3. Comutatorul **sw[0]** acceptă valoarea 0 sau 1. Aplicarea unei valori se realizează cu click-dreapta pe **sw0 → Force Constant …**. În fereastra de dialog care apare setați opțiunea Force Value la 1 și apăsați **OK**. Comutatorul va fi activat la 1 logic ceea ce va determina numărarea crescătoare. Setarea la valoarea 0 ar determina numărarea inversă.
   Notă: În aceeași manieră, la nevoie, se pot introduce valori constante pe magistrale, exprimate în diverse baze de numerație, în funcție de setarea Value radix. Valorile se pot modifica în orice moment, înainte și după pornirea simulării.

Lansarea procesului de simulare

Porniți procesul de simulare apăsând în mod repetat butonul **RunFor** din zona de comandă (vezi Figura B. 2). La fiecare apăsare simularea va avansa cu câte 10ms. În partea stângă a panoului de simulare aveți la dispoziție butoanele **Zoom In** și **Zoom Out** (Figura B. 3). Apăsați în mod repetat **Zoom Out** pentru a avea o perspectivă mai amplă a intervalului de timp simulat (ca în Figura B. 5) și observați pe formele de undă cum la fiecare activare a lui **btn0**, semnalul **en** (semnalul **enable** generat de MPG) se activează pentru un impuls, determinând astfel incrementarea numărătorului **cnt[2:0]** și modificarea valorilor pe led-urile 0-7. Led-urile 8-15 au valoare necunoscută (U – unknown) deoarece nu sunt conectate.

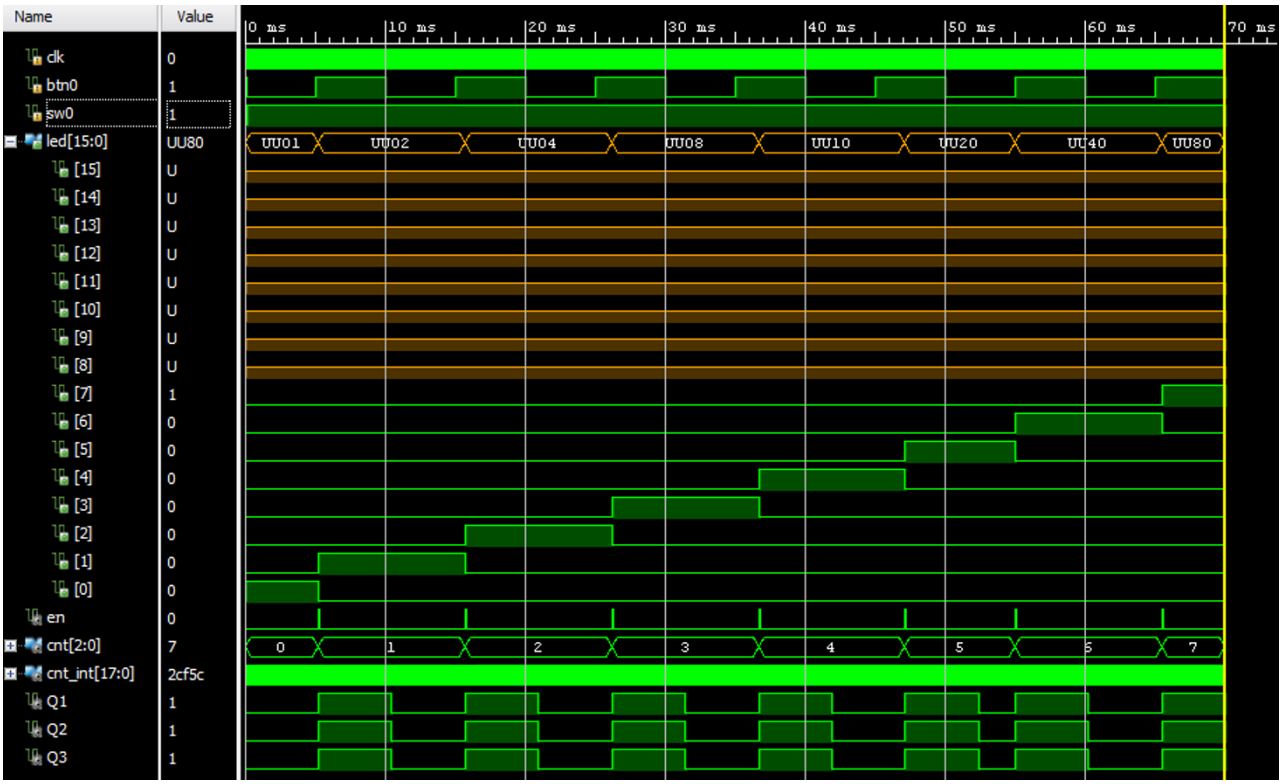


Figura B. 5 Formele de undă generate în timpul simulării

Modificați valoarea comutatorului **sw0** la 0 cu click-dreapta → **Force Constant …**. Apoi rulați câțiva pași de simulare apăsând în mod repetat pe butonul **RunFor** și observați numărarea inversă.

Pentru a vizualiza mai îndeaproape (mai rafinat) formele de undă, la un anumit moment de timp parcurs deja de simulator, se apasă click pe oricare din ele în dreptul momentului dorit, determinând astfel deplasarea liniei verticale galbene în poziția respectivă. Ulterior, se poate apăsa în mod repetat pe **Zoom In**  pentru mărire. Dacă se dorește continuarea simulării se apasă **Zoom Fit**  și se amplasează (cu click) linia galbenă la capătul din dreapta a formelor de undă, după care se poate apăsa butonul **RunFor** pentru avansarea simulării.

Salvarea ferestrei de simulare pentru utilizări ulterioare (opțional)

Configurația semnalelor din fereastra Untitled 1 se poate salva pentru utilizări ulterioare, accesând la meniu **File → Save Waveform Configuration**  . Această configurație nu include nici formele de undă generate și nici stimulii asociați semnalelor. Cu **File → Open Waveform Configuration …** se poate încărca o configurație salvată anterior.

Închiderea simulatorului

Închideți simulatorul de la meniu: **File → Close Simulation**. Nu este necesară salvarea ferestrei cu formele de undă, dacă vi se cere.

Concluzii

Simulatorul Vivado reprezintă o unealtă integrată în mediul Vivado, ușor de configurat și utilizat pentru proiectele dezvoltate în cadrul acestuia. Facilitățile de simulare bazate pe stimuli de tip clock sau valori constante facilitează urmărirea cu ușurință a propagării semnalelor în cadrul unităților componente, lucru esențial pentru arhitecturi complexe dezvoltate pe mai multe niveluri ierarhice.

C. Anexa 3 – Circuit de deplasare variabilă

Un circuit de deplasare variabilă se poate implementa cu o ierarhie de multiplexoare MUX 2:1 organizate pe mai multe niveluri. Ieșirile multiplexoarelor de la un nivel se conectează la intrările multiplexoarelor de la nivelul următor, cu grade de deplasare, puteri ale lui 2, în funcție de nivel.

Exemplu în VHDL a unui circuit de deplasare variabilă pentru date pe 8 biți, organizat pe 2 niveluri:

- $X_{7:0}$ – intrarea de date;
- $N_{1:0}$ – numărul de poziții deplasate, în binar (0-3);
- DIR – direcția de deplasare: ‘0’ – la stânga; ’1’ – la dreapta aritmetic;
- Y – rezultatul după deplasare;
- tmp – semnal intern folosit pentru conexiunile dintre cele 2 niveluri.

```
process(X, N(0), DIR) -- deplasare cu 1 poziție
begin
        if N(0) = '1' then
                if DIR = '0' then       -- deplasare la stânga
                        tmp <= X(6 downto 0) & '0';
                else                    -- deplasare aritmetică la dreapta
                        tmp <= X(7) & X(7 downto 1);
                end if;
        else
                tmp <= X(7 downto 0);
        end if;
end process;

process(tmp, N(1), DIR) -- deplasare cu 2 poziții
begin
        if N(1) = '1' then
                if DIR = '0' then       -- deplasare la stânga
                        Y <= tmp(5 downto 0) & "00";
                else                    -- deplasare aritmetică la dreapta
                        Y <= tmp(7) & tmp(7) & tmp(7 downto 2);
                end if;
        else
                Y <= tmp;
        end if;
end process;
```

D. Anexa 4 – Implementarea unui Bloc de Registre 32x32

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rf is
port (  clk     : in std_logic;
        ra1     : in std_logic_vector(4 downto 0);          -- Read Address 1
        ra2     : in std_logic_vector(4 downto 0);          -- Read Address 2
        wa      : in std_logic_vector(4 downto 0);          -- Write Address
        wd      : in std_logic_vector(31 downto 0);         -- Write Data
        regwr   : in std_logic;                             -- comanda RegWrite
        rd1     : out std_logic_vector(31 downto 0);        -- Read Data 1
        rd2     : out std_logic_vector(31 downto 0));       -- Read Data 2
end rf;

architecture Behavioral of rf is

type rf_type is array(0 to 31) of std_logic_vector(31 downto 0);
signal mem : rf_type:= (
        others => X"00000000");

begin

        process(clk)
        begin
                if rising_edge(clk) then
                        if regwr = '1' then
                                mem(conv_integer(wa)) <= wd; -- scriere sincronă
                        end if;
                end if;
        end process;

        rd1 <= mem(conv_integer(ra1)); -- citire asincronă
        rd2 <= mem(conv_integer(ra2)); -- citire asincronă

end Behavioral;
```

E. Anexa 5 – Implementarea unui RAM 64x32 de tip *write-first*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ram_wr_1st is
port (  clk     : in std_logic;
        we      : in std_logic;                         -- comanda Write Enable
        en      : in std_logic; -- opțional
        addr    : in std_logic_vector(5 downto 0);      -- Read/Write Address
        di      : in std_logic_vector(31 downto 0);     -- Data In
        do      : out std_logic_vector(31 downto 0));   -- Data Out
end ram_wr_1st ;

architecture Behavioral of ram_wr_1st is

type ram_type is array (0 to 63) of std_logic_vector(31 downto 0);
signal mem : ram_type := (
        others => X"00000000");

begin

        process(clk)
        begin
                if rising_edge(clk) then
                        if en = '1' then -- opțional
                                if we = '1' then        -- scriere sincronă
                                        mem(conv_integer(addr)) <= di;
                                        do <= di;
                                else                    -- citire sincronă
                                        do <= mem(conv_integer(addr));
                                end if;
                        end if;
                end if;
        end process;

end Behavioral;
```

F. Anexa 6 – Instrucțiuni MIPS 32

Pentru fiecare instrucțiune se prezintă descrierea, RTL abstract, sintaxa în asamblare și formatul pe biți. **Notă**: Setul de instrucțiuni complet se găsește în [2] și [3] de la Lucrarea 4.

ADD – ADDition

Descriere	Adună registrele $s și $t. Stochează rezultatul în registrul $d.
RTL	$d ← $s + $t; PC ← PC + 4;
Sintaxă	add $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100000

ADDI – ADD Immediate

Descriere	Adună registrul $s cu valoarea imm. Stochează rezultatul în registrul $t.
RTL	$t ← $s + SE(imm); PC ← PC + 4;
Sintaxă	addi $t, $s, imm
Format	001000 sssss ttttt iiiiiiiiiiiiiiii

AND – bitwise AND

Descriere	ȘI între registrele $s și $t. Stochează rezultatul în registrul $d.
RTL	$d ← $s & $t; PC ← PC + 4;
Sintaxă	and $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100100

ANDI – AND Immediate

Descriere	ȘI între registrul $s și valoarea imm. Stochează rezultatul în registrul $t.
RTL	$t ← $s & ZE(imm); PC ← PC + 4;
Sintaxă	andi $t, $s, imm
Format	001100 sssss ttttt iiiiiiiiiiiiiiii

BEQ – Branch on EQual

Descriere	Realizează salt condiționat dacă registrele $s și $t sunt egale.
RTL	if $s = $t then PC ← (PC + 4) + (SE(offset) << 2) else PC ← PC + 4;
Sintaxă	beq $s, $t, offset
Format	000100 sssss ttttt oooooooooooooooo

BGEZ – Branch on Greater than or Equal to Zero

Descriere	Realizează salt condiționat dacă registrul $s este pozitiv.
RTL	if $s ≥ 0 then PC ← (PC + 4) + (SE(offset) << 2) else PC ← PC + 4;
Sintaxă	bgez $s, offset
Format	000001 sssss 00000 oooooooooooooooo

BGTZ – Branch on Greater Than Zero

Descriere	Realizează salt condiționat dacă registrul $s este strict pozitiv.
RTL	If $s > 0 then PC ← (PC + 4) + (SE(offset) << 2) else PC ← PC + 4;
Sintaxă	bgtz $s, offset
Format	000111 sssss 00000 oooooooooooooooo

BNE – Branch on Not Equal

Descriere	Realizează salt condiționat dacă registrul $s diferă de $t.
RTL	if $s ≠ $t then PC ← (PC + 4) + (SE(offset) << 2) else PC ← PC + 4;
Sintaxă	bne $s, $t, offset
Format	000101 sssss ttttt oooooooooooooooo

J – Jump

Descriere	Realizează salt necondiționat la adresa absolută.
RTL	PC ← (PC + 4)[31:28] || (addr << 2);
Sintaxă	j addr
Format	000010 aaaaaaaaaaaaaaaaaaaaaaaaaa

JR – Jump Register

Descriere	Realizează salt necondiționat la adresa indicată de registrul $s.
RTL	PC ← $s;
Sintaxă	jr $s
Format	000000 sssss 00000 00000 00000 001000

LW – Load Word

Descriere	Încarcă un cuvânt de memorie în registrul $t.
RTL	$t ← MEM[$s + SE(offset)]; PC ← PC + 4;
Sintaxă	lw $t, offset($s)
Format	100011 sssss ttttt oooooooooooooooo

NOOP – NO OPeration

Descriere	Instrucțiune fără efect asupra procesorului. Echivalent SLL $0, $0, 0.
RTL	$0 ← $0 << 0; PC ← PC + 4;
Sintaxă	noop
Format	000000 00000 00000 00000 00000 000000

OR – bitwise OR

Descriere	SAU între registrele $s și $t. Stochează rezultatul în registrul $d.
RTL	$d ← $s | $t; PC ← PC + 4;
Sintaxă	or $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100101

ORI – bitwise OR Immediate

Descriere	SAU între registrul $s și valoarea imm. Stochează rezultatul în registrul $t.
RTL	$t ← $s | ZE(imm); PC ← PC + 4;
Sintaxă	ori $t, $s, imm
Format	001101 sssss ttttt iiiiiiiiiiiiiiii

SLL – Shift-Left Logical

Descriere	Deplasare logică la stânga pentru registrul $t cu h poziții. Stochează rezultatul în registrul $d.
RTL	$d ← $t << h; PC ← PC + 4;
Sintaxă	sll $d, $t, h
Format	000000 00000 ttttt ddddd hhhhh 000000

SLLV – Shift-Left Logical Variable

Descriere	Deplasare logică la stânga pentru registrul $t cu $s poziții. Stochează rezultatul în registrul $d.
RTL	$d ← $t << $s; PC ← PC + 4;
Sintaxă	sllv $d, $t, $s
Format	000000 sssss ttttt ddddd 00000 000100

SLT – Set on Less Than (signed)

Descriere	Dacă registrul $s < $t, atunci registrul $d devine 1, altfel devine 0.
RTL	PC ← PC + 4; if $s < $t then $d ← 1 else $d ← 0;
Sintaxă	slt $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 101010

SLTI – Set on Less Than Immediate (signed)

Descriere	Dacă registrul $s < imm, atunci registrul $t devine 1, altfel devine 0.
RTL	PC ← PC + 4; if $s < SE(imm) then $t ← 1 else $t ← 0;
Sintaxă	slti $t, $s, imm
Format	001010 sssss ttttt iiiiiiiiiiiiiiii

SRA – Shift-Right Arithmetic

Descriere	Deplasare aritmetică la dreapta pentru registrul $t cu h poziții. Stochează rezultatul în registrul $d.
RTL	$d ← $t >> h; PC ← PC + 4;
Sintaxă	sra $d, $t, h
Format	000000 00000 ttttt ddddd hhhhh 000011

SRL – Shift-Right Logical

Descriere	Deplasare logică la dreapta pentru registrul $t cu h poziții. Stochează rezultatul în registrul $d.
RTL	$d ← $t >> h; PC ← PC + 4;
Sintaxă	srl $d, $t, h
Format	000000 00000 ttttt ddddd hhhhh 000010

SUB – SUBtraction

Descriere	Scade registrul $t din $s. Stochează rezultatul în registrul $d.
RTL	$d ← $s - $t; PC ← PC + 4;
Sintaxă	sub $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100010

SW – Store Word

Descriere	Stochează într-un cuvânt de memorie valoarea registrului $t.
RTL	MEM[$s + SE(offset)] ← $t; PC ← PC + 4;
Sintaxă	sw $t, offset($s)
Format	101011 sssss ttttt oooooooooooooooo

XOR – bitwise eXclusive-OR

Descriere	XOR între registrele $s și $t. Stochează rezultatul în registrul $d.
RTL	$d ← $s ⊕ $t; PC ← PC + 4;
Sintaxă	xor $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100110

G. Anexa 7 – Programe de test pentru procesorul MIPS 32

Notă: Aceste exemple pot să constituie un punct de pornire al programului de test, dar pot suporta modificări sau adaosuri pentru o complexitate adecvată.

Important: Pentru problemele propuse în continuare, adresele la memorie sunt întotdeauna exprimate în octeți, elementele din șiruri sunt numere întregi cu semn pe 32 de biți și soluțiile trebuie să fie implementate cu bucle.

1. Să se determine numărul de valori pozitive și impare dintr-un șir de *N* elemente stocat în memorie începând cu adresa 8. *N* se citește din memorie de la adresa 4. Numărul de valori determinate se va scrie în memorie la adresa 0.

2. Să se parcurgă un șir de 8 elemente aflat în memorie începând cu adresa *A* (*A*≥8) și pentru fiecare poziție se va reține valoarea corespunzătoare primilor 16 biți mai puțin semnificativi, cu scopul de a calcula suma acestora. Valoarea *A* se citește din memorie de la adresa 0. Suma se va scrie la adresa 4.

3. Să se înlocuiască fiecare element x_i, dintr-un șir de dimensiune *N*, cu valoarea $x_i - x_{N-i+1}$, unde $i=\overline{1,N}$. Șirul se află în memorie începând cu adresa 4. *N* se citește de la adresa 0. Pentru verificare, se poate adăuga o buclă de citire a elementelor șirului, la final.

4. Să se scrie primele *N* elemente ale șirului lui Fibonacci, înmulțite cu 8, la locații consecutive în memorie, pe 32 de biți, începând cu adresa *A* (*A*≥8). Valorile *A* și *N* se citesc din memorie de la adresele 0, respectiv 4. Pentru verificare, se poate adăuga o buclă de citire a elementelor șirului, la final.

5. Să se înlocuiască fiecare element x_i, dintr-un șir de dimensiune *N*, cu valoarea $x_i + x_{i+1}$, unde $i=\overline{1,N-1}$, și x_N cu $x_N + x_1$. Șirul se află în memorie începând cu adresa 4. *N* se citește de la adresa 0. Pentru verificare, se poate adăuga o buclă de citire a elementelor șirului, la final.

6. Să se înlocuiască toate elementele dintr-un șir cu triplul lor, dacă sunt mai mari decât *X*, altfel cu jumătatea lor obținută prin împărțire întreagă. Șirul se află în memorie începând cu adresa *A* (*A*≥12) și are *N* elemente. *A*, *N* și *X* se citesc din memorie de la adresele 0, 4, respectiv 8. Pentru verificare, se poate adăuga o buclă de citire a elementelor șirului, la final.

7. Să se determine cel mai mare divizor comun a 2 numere *X* și *Y* citite din memorie de la adresa 0, respectiv 4. Se va implementa algoritmul lui Euclid cu scăderi (https://www.pbinfo.ro/articole/73/cmmdc-si-cmmmc-algoritmul-lui-euclid). Se vor scrie în memorie, la locații consecutive pe 32 de biți, începând cu adresa 8, valorile *X*, *Y,* urmate de cele intermediare generate de algoritmul lui Euclid. Ultimul număr scris va fi cel mai mare divizor comun. Pentru verificare, se poate adăuga o buclă de citire a elementelor șirului, la final.

8. Să se mute un șir de *N* elemente de la adresa *A* (*A*≥16), la adresa *A*-8. Valorile *A* și *N* se citesc din memorie de la adresele 0, respectiv 4. Pentru verificare, se poate adăuga o buclă de citire a elementelor noului șir, la final.

9. Să se scrie în memorie biții unui număr, în ordine, de la cel mai puțin semnificativ la bitul de 1 cel mai semnificativ, începând cu adresa 8. Fiecare bit va ocupa o locație de 32 de biți în memorie. Numărul este pozitiv și se citește din memorie de la adresa 4, pe 32 de biți. La adresa 0 se va scrie numărul de biți de 1 detectați. Pentru verificare, se poate adăuga o buclă de citire a elementelor șirului, la final.

10. Să se determine puterea lui 2 cea mai apropiată de un număr pozitiv ≤ 2^{31} citit din memorie de la adresa 0. Rezultatul se va scrie în memorie la adresa *A* (*A*≥8). Valoarea lui *A* se citește din memorie de la adresa 4.

11. Să se determine câte elemente se află în subsetul cel mai lung de numere ordonate crescător, dintr-un șir de *N* numere. Șirul începe în memorie de la adresa *A* (*A*≥12). *A* și *N* se citesc de la adresele 0, respectiv 4. Rezultatul se va scrie în memorie la adresa 8.

12. Să se determine valoarea pară maximă dintr-un șir de *N* numere stocate în memorie începând cu adresa *A* (*A*≥12). *A* și *N* se citesc de la adresele 0, respectiv 4. Rezultatul se va scrie în memorie la adresa 8.

13. Să se parcurgă memoria de la adresa *A* (*A*≥12) până se întâlnește o valoare nulă și să se determine câte valori întâlnite sunt mai mici ca *X*. *X* și *A* se citesc din memorie de la adresele 0, respectiv 4, iar rezultatul se va scrie la adresa 8.

14. Să se determine suma elementelor cu valori în intervalul [*X*, *Y*], dintr-un șir de *N* numere stocate în memorie începând cu adresa 16. Valorile *X*, *Y* și *N* se citesc din memorie de la adresele 0, 4, respectiv 8. Rezultatul se va scrie în memorie la adresa 12.

15. Să se determine dacă valorile unui șir de *N* elemente sunt ordonate crescător. Șirul este stocat în memorie începând cu adresa *A* (*A*≥12). *A* și *N* se citesc de la adresele 0, respectiv 4. Rezultatul (1=true / 0=false) se va scrie la adresa 8.

16. Să se aplice o iterație a algoritmului de ordonare crescătoare Bubble Sort pe un șir de *N* numere stocate în memorie începând cu adresa *A* (*A*≥8). O iterație presupune parcurgerea elementelor de la primul la ultimul și fiecare element se interschimbă cu următorul, dacă este mai mare decât acesta. *A* și *N* se citesc de la adresele 0, respectiv 4. Pentru verificare, se poate adăuga o buclă de citire a elementelor șirului, la final.

17. Să se calculeze produsul a 2 numere pozitive *X* (*X*≤ 255) și *Y* (*Y*≤255) citite din memorie de la adresele 0, respectiv 4. Rezultatul se va scrie în memorie la adresa 8. Calculul va avea la bază adunări repetate, astfel: se parcurg biții lui *Y* de la Y_0 la Y_7 și dacă bitul curent este 1 se adună la rezultatul final valoarea lui *X*, deplasată la stânga cu numărul corespunzător de poziții.

18. Să se limiteze elementele unui șir între 2 valori *X* și *Y* (*X*<*Y*), astfel: dacă un element este mai mic decât *X*, atunci devine *X*, dacă este mai mare decât *Y*, atunci devine *Y*, altfel rămâne neschimbat. Șirul are *N* valori și începe în memorie de la adresa 12. *X*, *Y* și *N* se citesc din memorie de la adresele 0, 4, respectiv 8. Pentru verificare, se poate adăuga o buclă de citire a elementelor șirului, la final.

19. Să se determine dacă un șir de *N* elemente este progresie aritmetică (diferența dintre oricare 2 elemente consecutive este constantă). Șirul este stocat în memorie începând cu adresa *A* (*A*≥12). *A* și *N* se citesc de la adresele 4, respectiv 8. Rezultatul (1=true / 0=false) se va scrie în memorie la adresa 0.

20. Să se determine cea mai mică valoare pozitivă dintr-un șir de *N* elemente, care începe în memorie de la adresa 8. *N* se citește din memorie de la adresa 4. Rezultatul se va scrie în memorie la adresa 0.

H. Anexa 8 – Automate cu stări finite

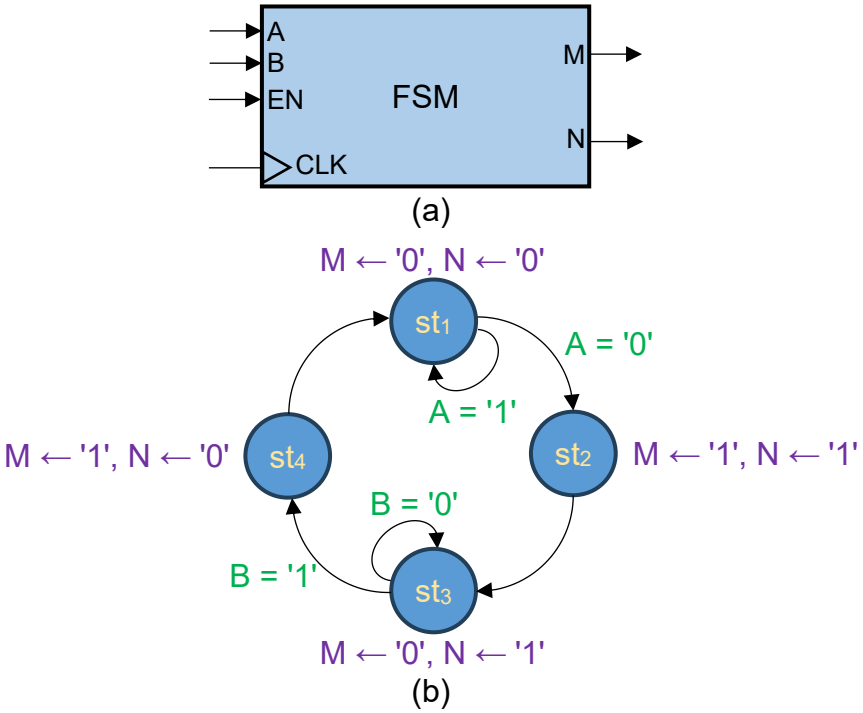


Figura H. 1 Automat Moore cu 4 stări: a) simbolul; b) diagrama de tranziții

Sunt prezentate în continuare 3 variante de descriere (cu 1, 2 și 3 procese) a automatului cu comportamentul din Figura H. 1.

Varianta cu 1 proces

În varianta cu 1 proces, comportamentul este descris integral în cadrul procesului. Tranzițiile și ieșirile sunt sincrone cu semnalul de ceas. Acest lucru este posibil fiindcă este un automat Moore (cu ieșirile pe stare).

```
entity fsm1p is
port (   clk, en, a, b     : in std_logic;
         m, n              : out std_logic);
end entity;

architecture Behavioral of fsm1p is
type state_type is (st1, st2, st3, st4);
signal state: state_type := st1;
begin

         p1: process(clk)
         begin
                  if rising_edge(clk) and en = '1' then
                           case state is
                                    when st1 =>    if a = '0' then
                                                            state <= st2;
                                                            m <= '1'; n <= '1';
                                                   else
                                                            state <= st1;
                                                            m <= '0'; n <= '0';
                                                   end if;
                                    when st2 =>    state <= st3;
                                                   m <= '0'; n <= '1';
                                    when st3 =>    if b = '1' then
                                                            state <= st4;
                                                            m <= '1'; n<= '0';
                                                   else
                                                            state <= st3;
                                                            m <= '0'; n <= '1';
                                                   end if;
                                    when st4 =>    state <= st1;
                                                   m <= '0'; n <= '0';
                           end case;
                  end if;
         end process;

end Behavioral;
```

Varianta cu 2 procese

În varianta cu 2 procese, în primul se definesc sincron tranzițiile, iar al doilea definește combinațional (asincron) valorile pe ieșiri în funcție de starea curentă.

```
entity fsm2p is
port (  clk, en, a, b   : in std_logic;
        m, n            : out std_logic);
end entity;

architecture Behavioral of fsm2p is

type state_type is (st1, st2, st3, st4);
signal state: state_type := st1;

begin

        p1: process(clk)
        begin
                if rising_edge(clk) and en = '1' then
                        case state is
                                when st1 =>   if a = '0' then
                                                new_st <= st2;
                                              else
                                                new_st <= st1;
                                              end if;
                                when st2 =>   new_st <= st3;
                                when st3 =>   if b = '1' then
                                                new_st <= st4;
                                              else
                                                new_st <= st3;
                                              end if;
                                when st4 =>   new_st <= st1;
                        end case;
                end if;
        end process;

        p2: process(state)
        begin
                case state is
                        when st1 => m <= '0'; n <= '0';
                        when st2 => m <= '1'; n <= '1';
                        when st3 => m <= '0'; n <= '1';
                        when st4 => m <= '1'; n <= '0';
                end case;
        end process;

end Behavioral;
```

Varianta cu 3 procese

În varianta cu 3 procese, primul stabilește natura sincronă a tranzițiilor între stări, iar următoarele calculează combinațional (asincron) noua stare, respectiv ieșirile corespunzătoare stării curente.

```
entity fsm3p is
port (  clk, en, a, b    : in std_logic;
        m, n             : out std_logic);
end entity;

architecture Behavioral of fsm3p is

type state_type is (st1, st2, st3, st4);
signal state: state_type := st1;
signal new_st: state_type;

begin

        p1: process(clk)
        begin
                if rising_edge(clk) and en = '1' then
                        state <= new_st;
                end if;
        end process;

        p2: process(state, a, b)
        begin
                case state is
                        when st1 =>   if a = '0' then
                                              new_st <= st2;
                                      else
                                              new_st <= st1;
                                      end if;
                        when st2 =>   new_st <= st3;
                        when st3 =>   if b = '1' then
                                              new_st <= st4;
                                      else
                                              new_st <= st3;
                                      end if;
                        when st4 =>   new_st <= st1;
                end case;
        end process;

        p3: process(state)
        begin
                case state is
                        when st1 => m <= '0'; n <= '0';
                        when st2 => m <= '1'; n <= '1';
                        when st3 => m <= '0'; n <= '1';
                        when st4 => m <= '1'; n <= '0';
                end case;
        end process;

end Behavioral;
```

I. Anexa 9 – Codurile ASCII

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040	&#032;	Space	64	40	100	&#064;	@	96	60	140	&#096;	`
1	1	001	Start of Header	33	21	041	&#033;	!	65	41	101	&#065;	A	97	61	141	&#097;	a
2	2	002	Start of Text	34	22	042	&#034;	"	66	42	102	&#066;	B	98	62	142	&#098;	b
3	3	003	End of Text	35	23	043	&#035;	#	67	43	103	&#067;	C	99	63	143	&#099;	c
4	4	004	End of Transmission	36	24	044	&#036;	$	68	44	104	&#068;	D	100	64	144	&#100;	d
5	5	005	Enquiry	37	25	045	&#037;	%	69	45	105	&#069;	E	101	65	145	&#101;	e
6	6	006	Acknowledgment	38	26	046	&#038;	&	70	46	106	&#070;	F	102	66	146	&#102;	f
7	7	007	Bell	39	27	047	&#039;	'	71	47	107	&#071;	G	103	67	147	&#103;	g
8	8	010	Backspace	40	28	050	&#040;	(	72	48	110	&#072;	H	104	68	150	&#104;	h
9	9	011	Horizontal Tab	41	29	051	&#041;	)	73	49	111	&#073;	I	105	69	151	&#105;	i
10	A	012	Line feed	42	2A	052	&#042;	*	74	4A	112	&#074;	J	106	6A	152	&#106;	j
11	B	013	Vertical Tab	43	2B	053	&#043;	+	75	4B	113	&#075;	K	107	6B	153	&#107;	k
12	C	014	Form feed	44	2C	054	&#044;	,	76	4C	114	&#076;	L	108	6C	154	&#108;	l
13	D	015	Carriage return	45	2D	055	&#045;	-	77	4D	115	&#077;	M	109	6D	155	&#109;	m
14	E	016	Shift Out	46	2E	056	&#046;	.	78	4E	116	&#078;	N	110	6E	156	&#110;	n
15	F	017	Shift In	47	2F	057	&#047;	/	79	4F	117	&#079;	O	111	6F	157	&#111;	o
16	10	020	Data Link Escape	48	30	060	&#048;	0	80	50	120	&#080;	P	112	70	160	&#112;	p
17	11	021	Device Control 1	49	31	061	&#049;	1	81	51	121	&#081;	Q	113	71	161	&#113;	q
18	12	022	Device Control 2	50	32	062	&#050;	2	82	52	122	&#082;	R	114	72	162	&#114;	r
19	13	023	Device Control 3	51	33	063	&#051;	3	83	53	123	&#083;	S	115	73	163	&#115;	s
20	14	024	Device Control 4	52	34	064	&#052;	4	84	54	124	&#084;	T	116	74	164	&#116;	t
21	15	025	Negative Ack.	53	35	065	&#053;	5	85	55	125	&#085;	U	117	75	165	&#117;	u
22	16	026	Synchronous idle	54	36	066	&#054;	6	86	56	126	&#086;	V	118	76	166	&#118;	v
23	17	027	End of Trans. Block	55	37	067	&#055;	7	87	57	127	&#087;	W	119	77	167	&#119;	w
24	18	030	Cancel	56	38	070	&#056;	8	88	58	130	&#088;	X	120	78	170	&#120;	x
25	19	031	End of Medium	57	39	071	&#057;	9	89	59	131	&#089;	Y	121	79	171	&#121;	y
26	1A	032	Substitute	58	3A	072	&#058;	:	90	5A	132	&#090;	Z	122	7A	172	&#122;	z
27	1B	033	Escape	59	3B	073	&#059;	;	91	5B	133	&#091;	[	123	7B	173	&#123;	{
28	1C	034	File Separator	60	3C	074	&#060;	<	92	5C	134	&#092;	\	124	7C	174	&#124;	|
29	1D	035	Group Separator	61	3D	075	&#061;	=	93	5D	135	&#093;	]	125	7D	175	&#125;	}
30	1E	036	Record Separator	62	3E	076	&#062;	>	94	5E	136	&#094;	^	126	7E	176	&#126;	~
31	1F	037	Unit Separator	63	3F	077	&#063;	?	95	5F	137	&#095;	_	127	7F	177	&#127;	Del

https://www.asciicharstable.com

Figura I. 1 Codurile ASCII ale caracterelor