

## Prefață

Îndrumătorul de laborator reprezintă materialul suport pentru punerea în aplicare a lucrărilor practice la disciplina Arhitectura Calculatoarelor, și este destinat studenților de anul 2, ciclul de licență, domeniul Calculatoare și Tehnologia Informației, din cadrul Facultății de Automatică și Calculatoare. De asemenea, îndrumătorul se adresează și celor care abordează noțiunile hardware fundamentale, la nivel de microprocesor, prin proiectarea, implementarea și testarea etapizată a arhitecturilor MIPS (Microprocessor without Interlocked Pipelined Stages) pe 32 de biți, cu ciclu unic și pipeline.

Conținutul îndrumătorului se întinde pe 12 lucrări la care se adaugă 9 anexe. Partea introductivă cuprinde 3 lucrări în care se pune accent pe descrierea sintetizabilă, în VHDL, a circuitelor logice combinaționale și secvențiale de bază, necesare ulterior. Urmează dezvoltarea graduală a procesorului MIPS cu ciclu unic, pe parcursul lucrărilor 4-8. Migrarea de la varianta cu ciclu unic la varianta pipeline este realizată 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), care sunt integrate cu procesoarele MIPS.

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 .....	3
1. Introducere în mediul de dezvoltare Vivado .....	4
2. Afișoarele pe 7-segmente și unitatea aritmetică-logică .....	12
3. Unitățile de memorare .....	17
4. Procesorul MIPS 32, ciclu unic – Introducere .....	21
5. Procesorul MIPS 32, ciclu unic – Extragerea instrucțiunilor .....	25
6. Procesorul MIPS 32, ciclu unic – Decodificare și control .....	29
7. Procesorul MIPS 32, ciclu unic – Finalizarea arhitecturii .....	33
8. Procesorul MIPS 32, ciclu unic – Testare .....	39
9. Procesorul MIPS 32, pipeline – Proiectare și implementare .....	41
10. Procesorul MIPS 32, pipeline – Rezolvarea hazardurilor .....	46
A. Anexa 1 – Ghid de utilizare Vivado .....	53
B. Anexa 2 – Simularea funcțională în mediul Vivado .....	59
C. Anexa 3 – Circuit de deplasare pe 8 biți .....	64
D. Anexa 4 – Implementarea unui Bloc de Registre 32x32 .....	65
E. Anexa 5 – Implementarea unui RAM 64x32 de tip <i>write-first</i> .....	66
F. Anexa 6 – Instrucțiuni pentru MIPS 32 .....	67
G. Anexa 7 – Programe de test pentru procesorul MIPS 32 .....	70
H. Anexa 8 – Implementări pentru automatele cu stări finite .....	72
I. Anexa 9 – Tabel cu codurile ASCII .....	76

## Obiectivele laboratorului de Arhitectura Calculatoarelor

În cadrul acestui îndrumător de laborator se urmărește implementarea de microprocesoare didactice în arhitectură MIPS, folosind limbajul de descriere hardware 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

Familiarizarea cu următoarele elemente:

- Utilitarul Vivado [1];
- Xilinx Synthesis Technology (XST) [2];
- Placa de dezvoltare Nexys A7 [3].

### 1.2. Resurse necesare

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

### 1.3. Reguli de descriere în VHDL

Pentru a eficientiza descrierea unui cod corect și sintetizabil se vor respecta următoarele reguli:

- Nu se va crea o nouă entitate pentru orice circuit. Nu creați entități separate pentru circuite simple (de bază) precum: porți logice, bistabile, registre, multiplexoare, numărătoare, decodificatoare, etc.;
- Nu se va abuza de descrierea structurală până la nivel de granularitate de porți logice;
- Bazați-vă în principal pe descrierea comportamentului ținând cont de tipul de circuit sintetizat pentru descrierea făcută;
- Se vor crea entități noi doar pentru părțile semnificativ complexe ale proiectului (acest lucru se menționează explicit pe parcurs);
- Procesele vor fi suficient de simple încât să se poată deduce tipul de circuit care se va sintetiza. Mențineți o descriere axată pe circuitele uzuale.

Semnalele se vor declara între **architecture** și **begin**, folosind strict tipurile **std\_logic** sau **std\_logic\_vector**:

- Declararea unui semnal de 1 bit:  
**signal sig\_name : std\_logic := '0';**
- Declararea unui semnal de N biți:  
**signal sig\_name : std\_logic\_vector(N-1 downto 0) := "00....0";**
- Exemple de inițializare:
  - binară, pentru semnal pe 16 biți: **"0000000000000000"**
  - hexazecimală, pentru semnal pe 32 biți: **X"00000000"**
  - binară, pentru semnal generic pe N biți: **(others => '0')**

## 1.4. Circuite de bază

### 1.4.1. Porțile logice fundamentale

Porțile logice fundamentale (Figura 1-1) sunt cele mai simple circuite combinaționale. Fiecare tip de poartă efectuează operația logică asociată conform regulilor algebrei booleene (Tabelul 1-1).

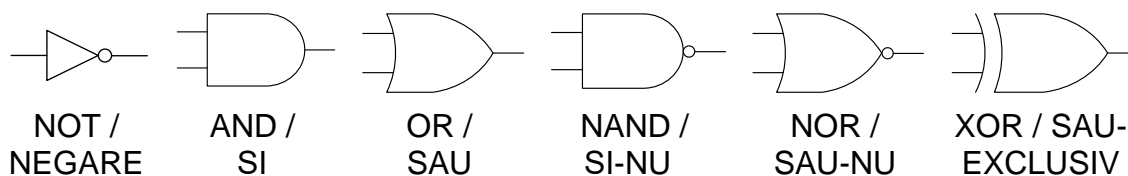


Figura 1-1: Simbolurile porților logice fundamenale

A	NOT	A	B	AND	OR	NAND	NOR	XOR
0	1	0	0	0	0	1	1	0
1	0	0	1	0	1	1	0	1
		1	0	0	1	1	0	1
		1	1	1	1	0	0	0

Tabelul 1.1: Tabelele de adevăr pentru porțile logice fundamentale

Descrierea în VHDL a unei porți logice se face prin atribuire concurentă, în afara proceselor, fără entități suplimentare. Exemple:

```
O <= not A;
O <= A and B;
O <= A or B;
O <= A nand B;
O <= A nor B;
O <= A xor B;
```

### 1.4.2. Multiplexorul

Multiplexorul (MUX) este un circuit combinațional care permite selecția pe ieșirea acestuia a uneia din mai multe intrări. Un MUX cu  $2^N$  intrări are  $N$  biți de selecție, care codifică indexul intrării transmise spre ieșire. Pentru  $N=2$  avem un MUX 4:1 (Figura 1-2).

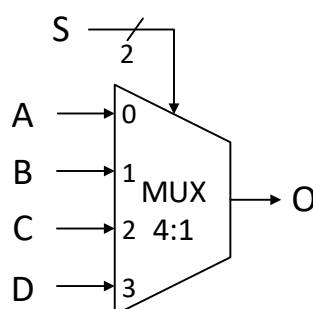


Figura 1-2: Simbolul MUX 4:1

Pini	Descriere
A, B, C, D	Intrările de date
S	Selecția
O	Ieșirea de date

Tabelul 1.2: Rolul pinilor unui MUX 4:1

Unealta de sinteză XST (Xilinx Synthesis Technology) permite mai multe moduri de descriere a unui multiplexor:

- în proces cu **if-then-else** sau cu **case**;
- concurențial cu **when-else**.

Exemple:

- Mux 2:1

```
O <= A when S = '0' else B;
sau
process(S, A, B)
begin
  if S = '0' then
    O <= A;
  else
    O <= B;
  end if;
end process;
```

- Mux 4:1

```
process(S, A, B, C, D)
begin
  case S is
    when "00" => O <= A;
    when "01" => O <= B;
    when "10" => O <= C;
    when others => O <= D;
  end case;
end process;
```

### 1.4.3. Decodificatorul

Decodificatorul (DCD) este un circuit combinațional care are  $N$  intrări și  $M \leq 2^N$  ieșiri de un bit. Varianta uzuală este decodificatorul  $N$ -la- $2^N$  în care cei  $N$  biți de intrare au rol de selecție și indică indexul (codul binar al) ieșirii care este activată, restul rămânând inactive. Pentru  $N=3$  avem un DCD 3:8 (Figura 1-3).

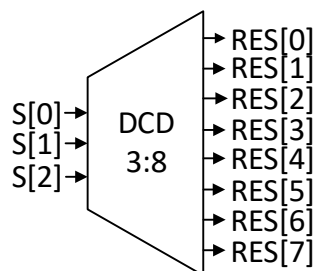


Figura 1-3: Simbolul DCD 3:8

Pini	Descriere
S	Selecția
RES	Ieșirea de date

Tabelul 1.3: Rolul pinilor unui DCD 3:8

Un mod uzual de a descrie un decodificator este folosind instrucțiunea **case** în cadrul unui proces astfel:

```

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

```

#### 1.4.4. Bistabilul D Flip-Flop

Bistabilul D Flip-Flop este un circuit secvențial care se poate afla în una dintre cele două stări stabile, iar tranziția se poate face asincron sau sincron pe frontul semnalului de ceas. Bistabilul D poate memora un bit de date. Unealta de sinteză XST recunoaște bistabilele D cu următoarele posibilități de control: Set/Reset asincron, Set/Reset sincron, cu sau fără activator (enable) pentru semnalul de ceas (clock). **Notă:** Circuitul de tip registru funcționează la fel ca bistabilul D, dar este pe mai mulți biți.

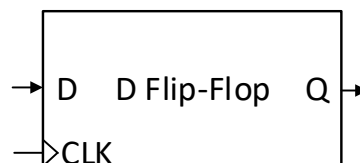


Figura 1-4: Simbolul bistabilului D Flip-Flop

Pini	Descriere
D	Semnalul de intrare (date)
CLK	Semnalul de ceas
Q	Semnalul de ieșire (date)

Tabelul 1.4: Rolul pinilor unui bistabil D Flip-Flop

Descrierea unui bistabil D Flip-Flop se face în cadrul unui proces. Pentru a identifica frontul crescător al semnalului de ceas se pot folosi variantele de mai jos:

```

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;
```

#### 1.4.5. Numărătorul

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

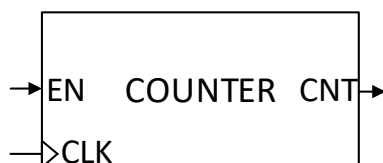


Figura 1-5: Simbolul unui numărător generic ( $N$  biți) pe frontul ascendent cu semnal EN de activare (enable) a numărării

Pini	Descriere
CLK	Semnalul de ceas
EN	Semnalul de activare (enable) a numărării
CNT	Ieșirea de date

Tabelul 1.5: Rolul pinilor unui numărător cu activarea numărării

Exemplu de descriere 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 inițierea 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.

### 1.5. Activități practice

**Notă:** Dacă este necesar, consultați ghidul online pentru VHDL [5].



1.5.1. **Implementați proiectul *test\_env* în Vivado** prin parcurgerea atentă și completă a tutorialului descris în Anexa 1.

1.5.2. **Adăugați un numărător binar reversibil pe 16 biți** în proiectul *test\_env*, prin descrierea comportamentului numărătorului în arhitectura entității. Numărarea va fi controlată de la un buton. Pașii de urmat sunt:

1. Declarați un semnal de 16 biți (de tip **std\_logic\_vector**) în arhitectură, 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 pe 1 sau decrementarea pe 0).
4. Conectați cei 16 biți ai numărătorului pe LED-uri (atribuire concurentă). **Comentați legarea comutatoarelor la LED-uri. O ieșire nu poate fi conectată direct la 2 resurse diferite!**

Dacă descrierea este corectă se poate vizualiza schema circuitului rezultat: panoul **Flow Navigator > RTL Analysis** → fereastra **Schematic** → **Reload** (**Reload** este situat în partea superioară a ferestrei).

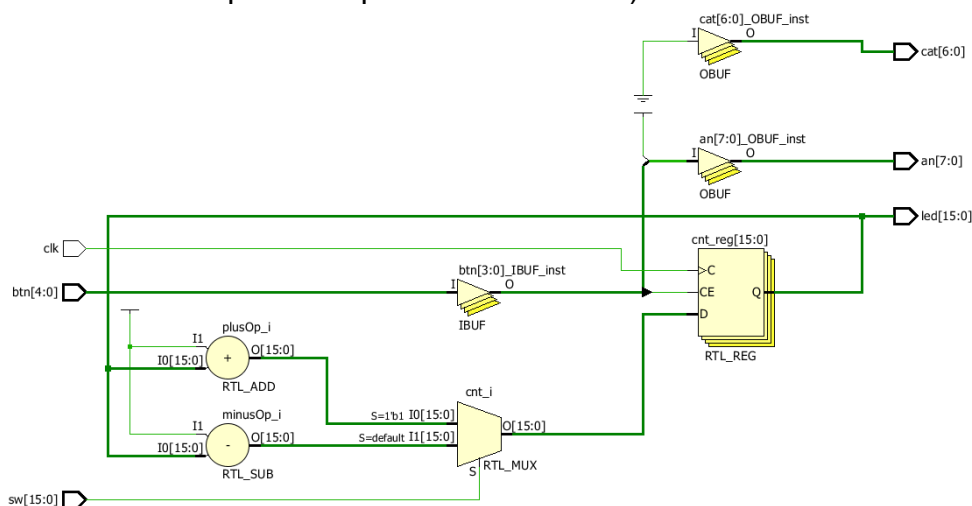


Figura 1-6: Schema circuitului cu numărător

Obțineți noul fișier \*.bit (panoul **Flow Navigator > Program and Debug > Generate Bitstream**). Încărcați proiectul pe placă. Controlați numărarea de la buton și direcția de numărare de la comutator (switch). **Observați ce probleme apar.**

### 1.5.3. Generator de Monoimpuls Sincron (MonoPulse Generator – MPG)

**Notă:** Veți continua lucrul în aceeași entitate de la punctul anterior (1.5.2).

În proiectele viitoare veți avea nevoie să controlați pas cu pas circuitele implementate cu scopul de a urmări cronologic fluxul de date și de control. Pentru aceasta va fi necesar un semnal ENABLE de activare/validare a frontului de ceas.

Arhitectura circuitului MPG care generează un singur impuls la o apăsare a butonului este prezentată în figura următoare:

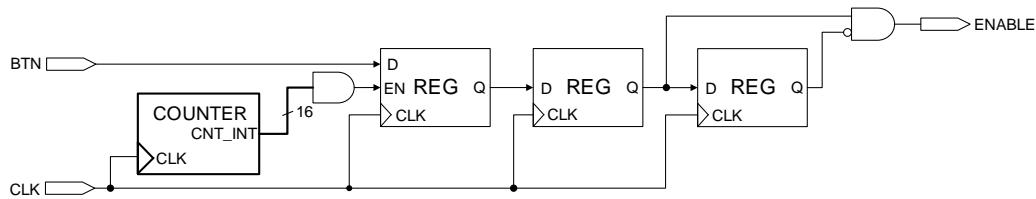


Figura 1-7: Arhitectura generatorului de monoimpuls sincron (MPG)

Rolul număratorului și a registrului 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 (16+).

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

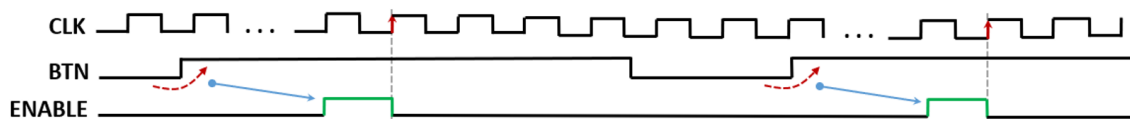


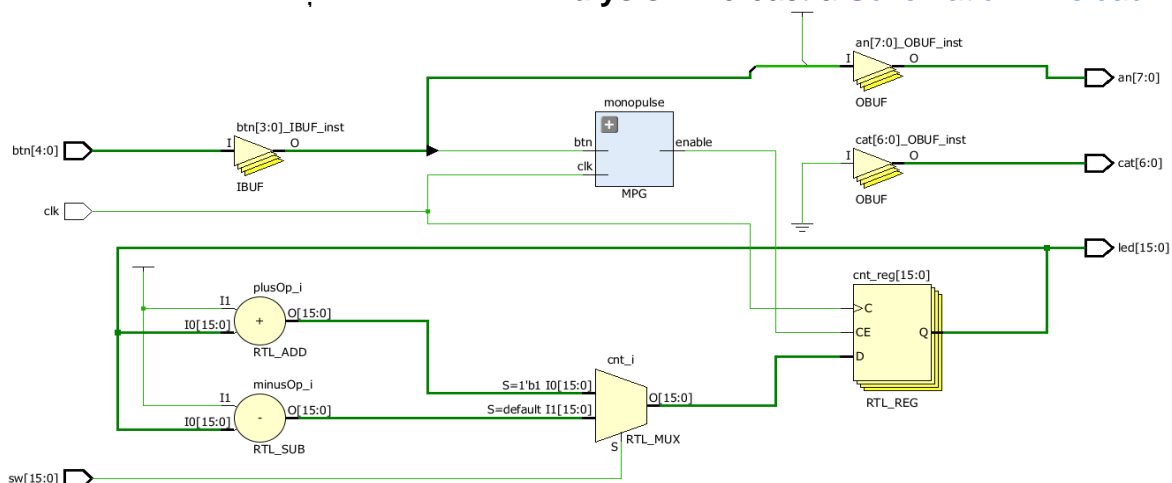
Figura 1-8: Efectul unui MPG pe ieșire

MPG-ul va fi implementat într-o altă entitate (fișier sursă), care trebuie creat 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** în secțiunea de declarare a semnalelor și instanțierea cu **port map** după **begin**.

Componentele din diagrama MPG (bistabile, numărător, porțile AND) se vor descrie comportamental prin declararea semnalelor necesare, respectiv a proceselor și a atribuirilor concurente în arhitectură.

Pașii de urmat:

1. Scrieți și verificați codul VHDL pentru MPG.
2. Includeți MPG în entitatea **test\_env** (vezi indicațiile de mai sus).
3. Folosiți ieșirea ENABLE ca semnal de activare a număratorului adăugat la activitatea 1.5.2: se schimbă condiția ca semnalul legat la ENABLE să fie 1, în locul butonului, acolo unde se testează apariția frontului crescător de ceas în numărător.
4. Vizualizați schema: **RTL Analysis → fereastra Schematic → Reload**.



5. Încărcați pe placă și testați.

1.5.4. **Creați un nou fișier VHDL denumit *test\_new*, folosind aceleași porturi** ca pentru primul fișier. Pașii de urmat:

1. Parcurgeți Anexa 1, începând cu **Crearea unui fișier VHDL** și implementați circuitul de mai jos în arhitectură. **Setați ca entitatea de top** 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 entități adiționale. Adăugați un numărător pe 3 biți și un decodificator DCD 3:8 biți, folosind semnale declarate în arhitectura **test\_new**, procese și atribuiri corespunzătoare.

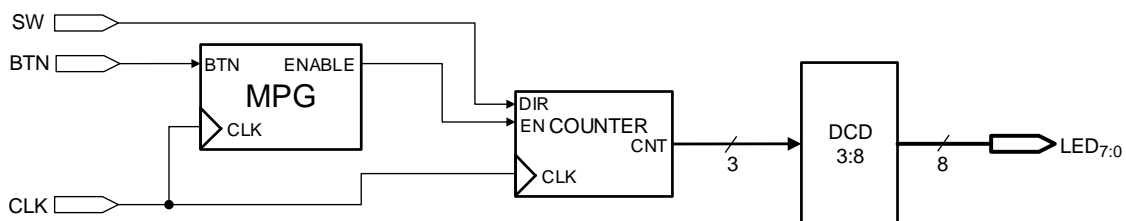


Figura 1-9: Schema circuitului pentru activitatea curentă

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

## 1.6. Referințe

- [1] Vivado Design Suite Overview. Disponibil online: <https://docs.xilinx.com/r/en-US/ug910-vivado-getting-started/Vivado-Design-Suite-Overview>
- [2] Vivado Design Suite User Guide – Synthesis (UG901). Disponibil online: <https://docs.xilinx.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](https://peterfab.com/ref/vhdl/vhdl_renerta)

## Lucrarea 2

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

#### 2.1. Obiective

Descrierea, implementarea și testarea pentru:

- Afișorul pe 7 segmente;
- O unitate Aritmetică-Logică simplă (Arithmetic Logic Unit – ALU).

Aprofundarea cunoștințelor legate de:

- Utilitarul Vivado [1];
- Placa de dezvoltare Nexys A7 [2];
- Limbajul VHDL [3].

#### 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 [2].

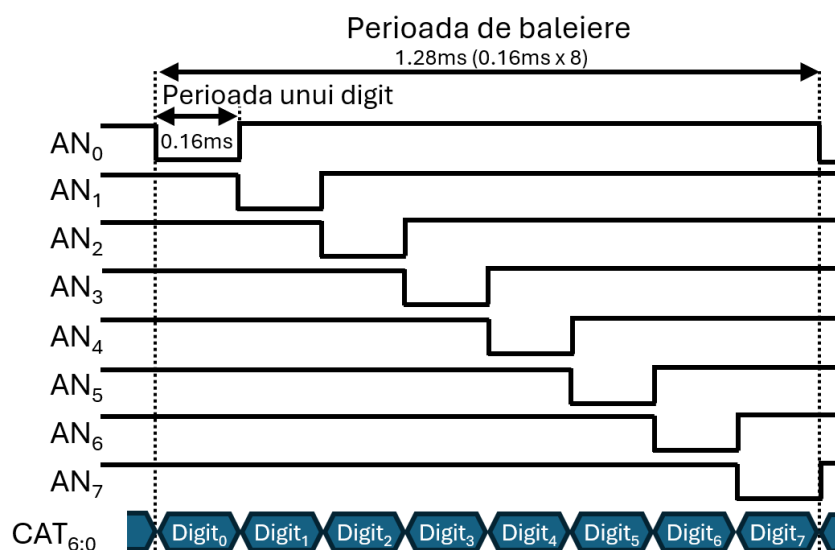


Figura 2-1: Diagrama de timp pentru SSD

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<sub>0-7</sub> 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 număratorului, afișoarele sunt activate alternativ (prin *anozi*), în paralel cu plasarea valorilor Digit<sub>i</sub> corespunzătoare pe *catozi*.

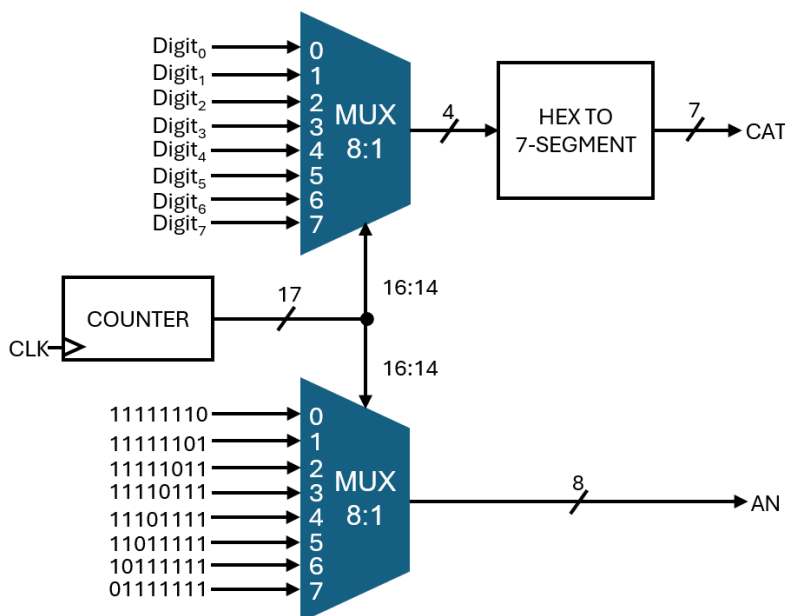


Figura 2-2: Schema logică a circuitului SSD de afișare pe 8 afișoare

## 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**.

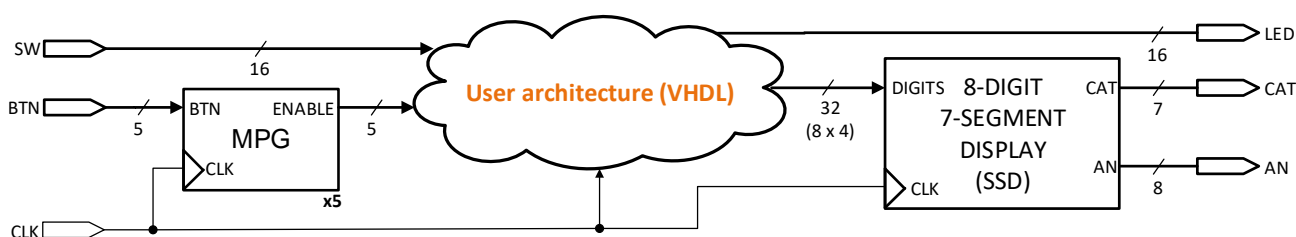


Figura 2-3: Schema unui proiect la nivelul de abstractizare cel mai înalt

## 2.4. Operații aritmetice

### 2.4.1. Sumatoare (Adders)

Sumatorul efectuează adunarea între numere. Ecuațiile booleene pentru un sumator complet pe 1 bit sunt:

$$SUM = A \oplus B \oplus CI,$$

$$CO = A \cdot B + A \cdot CI + B \cdot CI,$$

CI – Carry In

CO – Carry Out

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

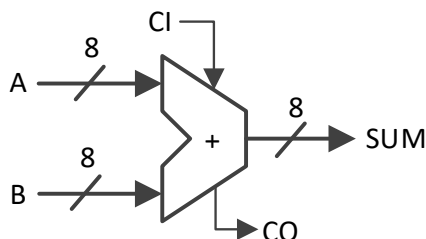


Figura 2-4: Simbolul sumatorului complet pe 8 biți

VHDL: **SUM** <= A + B; (în acest caz CI=0 și CO se ignoră)

#### 2.4.2. Scăzătoare (Subtractors)

Scăzătorul efectuează operația de scădere. Pentru numere în complement față de 2, scăderea se realizează prin adunarea cu complementul astfel:

$$A - B = A + \overline{B} + 1$$

VHDL: **DIF** <= A - B;

#### 2.4.3. Circuite combinaționale de deplasare (Shifters)

Circuitul combinațional de deplasare translatează un cuvânt de biți cu un număr specificat de poziții, spre stânga sau spre dreapta. **Prin deplasare, numărul de biți nu se schimbă!** Există două tipuri de circuite de deplasare (Figura 2-5):

- *Deplasare logică* – pe pozițiile rămase goale se completează întotdeauna cu 0, indiferent de direcția de deplasare;
- *Deplasare aritmetică* – la deplasare spre stânga completează cu 0, iar la deplasare spre dreapta se completează cu bitul de semn.

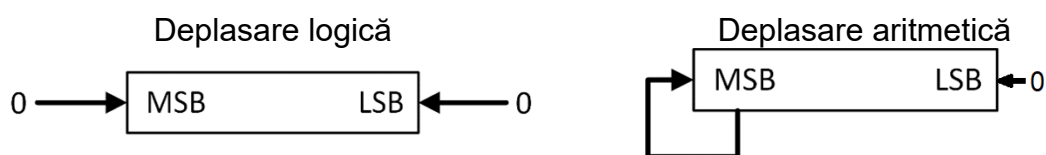


Figura 2-5: Tipuri de deplasare. MSB=Most Significant Bit, LSB=Least Significant Bit

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

VHDL: **iesire(31 downto 0) <= intrare(26 downto 0) & "00000";**  
(deplasare la stânga cu 5 poziții, pe 32 biți)

VHDL: **iesire(15 downto 0) <= "000" & intrare(15 downto 3);**  
(deplasare logică la dreapta cu 3 poziții, pe 16 biți)

VHDL: **iesire(7 downto 0) <= intrare(7) & intrare(7) & intrare(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 variabilă, între 0 și 7 poziții.

#### 2.4.4. Extindere cu semn sau cu zero (Sign Extender / Zero Extender)

Circuitul de extindere cu semn sau cu zero realizează adaptarea unui semnal de dimensiune redusă la un număr mai ridicat de biți necesari, 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: `iesire(31 downto 0) <= X"0000" & intrare(15 downto 0);`  
(extindere cu 0, de la 16 la 32 biți)

VHDL: `iesire(9 downto 0) <= "000" & intrare(6 downto 0);`  
(extindere cu 0, de la 7 la 10 biți)

VHDL: `iesire(7 downto 0) <= intrare(5) & intrare(5) & intrare(5 downto 0);`  
(extindere cu semn, de la 6 la 8 biți)

#### 2.4.5. Detector de zero (Zero Detector)

Detectorul de zero pe  $n$  biți se poate realiza cu o poartă NOR cu  $n$  intrări și răspunsul pe 1 bit. Detectorul este util pentru a verifica nulitatea unei valori binare (dacă toți biții sunt 0).

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

#### 2.4.6. Comparatoare

Comparatorul testează egalitatea a două numere binare și generează un răspuns pe 1 bit. Se poate realiza cu porți logice fundamentale sau calculând diferența și testând dacă rezultatul este 0 (un scăzător combinat cu detector de 0).

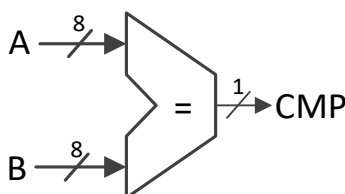


Figura 2-6: Simbolul comparatorului pe 8-biți

VHDL: `CMP <= '1' when A=B 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 separată numită **SSD**, care să implementeze, în VHDL, circuitul de afișare din Figura 2-2.

Folosiți doar semnale și procese pentru a descrie numărătorul și cele două multiplexoare (cu case). Extrageți din **Language Templates** (VHDL > Synthesis Constructs > Coding Examples > Misc > 7-Segment Display Hex Conversion) unitatea "HEX TO 7-SEGMENT", care transformă cifra pe 4 biți în combinația de 7 led-uri.

**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) conectarea ieșirii numărătorului la led-uri.

Numărătorul este controlat de la un buton prin componenta MPG. Structura proiectului devine acum în conformitate cu arhitectura din Figura 2-3. 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: ADD, SUB, SHIFT LEFT 2, LOGIC SHIFT RIGHT 2, conform diagramei 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. Pe intrarea Digits a SSD înlocuiți ieșirea numărătorului cu ieșirea de la MUX 4:1.

Programați pe placă și testați circuitul.

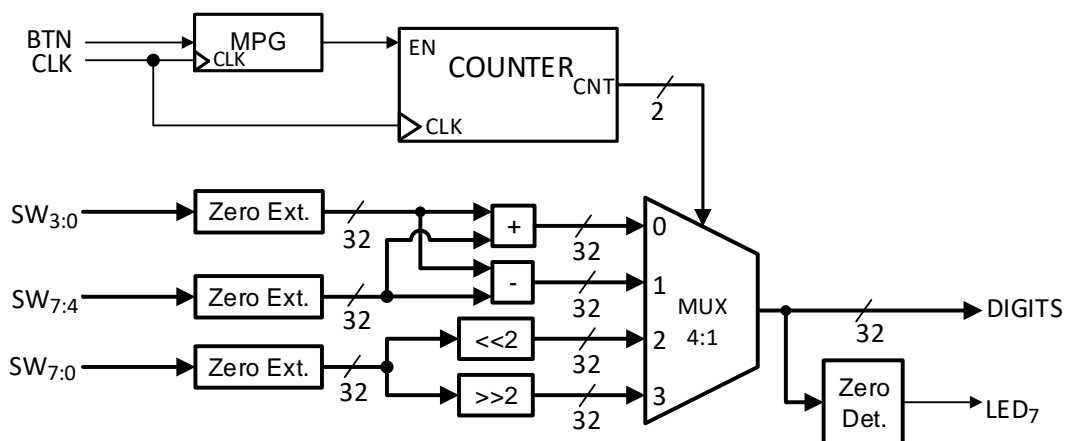


Figura 2-7: Schema unei ALU simple

## 2.6. Referințe

- [1] Vivado Design Suite Overview. Disponibil online: <https://docs.xilinx.com/r/en-US/ug910-vivado-getting-started/Vivado-Design-Suite-Overview>
- [2] Nexys A7 Reference Manual. Disponibil online: <https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual>
- [3] VHDL Language Reference Guide. Disponibil online: [https://peterfab.com/ref/vhdl/vhdl\\_renerta](https://peterfab.com/ref/vhdl/vhdl_renerta)



## Lucrarea 3

### 3. Unitățile de memorare

#### 3.1. Obiective

Descrierea, implementarea și testarea pentru:

- Blocul de registre (Register File);
- Memorii ROM (Read Only Memory);
- Memorii RAM (Random Access Memory).

Aprofundarea cunoștințelor legate de:

- Utilitarul Vivado [1];
- Placa de dezvoltare Nexys A7 [2];
- Limbajul VHDL [3].

#### 3.2. Fundamente teoretice

##### 3.2.1. Blocul de Registre (Register File – RF)

Blocul de Registre reprezintă spațiul de stocare cel mai frecvent folosit într-un procesor. Majoritatea operațiilor implică folosirea sau modificarea datelor din Blocul de Registre, așadar trebuie să fie foarte rapid și în consecință este limitat ca dimensiune pentru a reduce timpii de propagare. Implementarea uzuală în FPGA este cu bistabile, permițând astfel o viteză ridicată și acces multiplu simultan.

Arhitectura Blocului de Registre specific procesorului MIPS (Figura 3-1) are două adrese de citire (Read Address 1 și 2) și una de scriere (Write Address). Conținutul registrelor de la adresele de citire este livrat pe porturile Read Data 1 și 2, citirea fiind asincronă. Datele de pe portul Write Data sunt scrise în registrul de la adresa de scriere, dacă semnalul de control **RegWrite** este activ. Scrierea este sincronă pe frontul de ceas. Anexa 4 prezintă descrierea în VHDL a unui bloc de 32 registre pe 32 de biți.

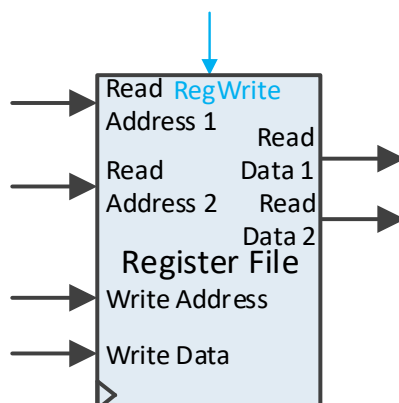


Figura 3-1: Bloc de registre cu două porturi de citire și unul de scriere

### 3.2.2. Memorii ROM și RAM

Memoriile ROM permit doar operații de citire, iar memoriile RAM permit atât citirea, cât și scrierea. Ambele tipuri sunt esențiale pentru orice procesor.

Dispozitivele FPGA sunt echipate cu un anumit număr de blocuri de memorie BRAM (Block RAM) [2]. 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 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 de generarea unor memorii mari, și consumul de resurse FPGA este mult redus. Așadar, la dimensiuni mici este avantajoasă implementarea distribuită, iar la dimensiuni mari este recomandată implementarea cu BRAM.

La memorii ROM unealta de sinteză XST 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 XST acceptă următoarele caracteristici pentru RAM:

- cu sau fără activare (enable);
- scriere sincronă;
- citire sincronă sau asincronă;
- 1 sau 2 porturi de citire.

Există trei moduri posibile de implementare a unei memorii RAM cu citire sincronă [5]: *write-first*, *read-first* și *no-change*. Acestea definesc ce informație 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.**

Accesați **Language Templates**: 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 în VHDL

Declararea unei memorii 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, memoriile primesc pe placă 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 arbitrare (vezi Secțiunea 3.2.3). Folosiți un numărator pe 5 biți pentru a genera adresa. Acesta va fi controlat de la un buton prin MPG. Conținutul memoriei ROM de la adresa curentă se va afișa pe SSD. Schema circuitului este în Figura 3-2. **Comportamentul memoriei ROM se descrie printr-o linie de cod, ca o 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 numărator 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 2 butoane, cu ajutorul a 2 unități MPG. Valorile pe cele 2 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 dublează. Conectați comanda de resetare asincronă a număratorului (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.

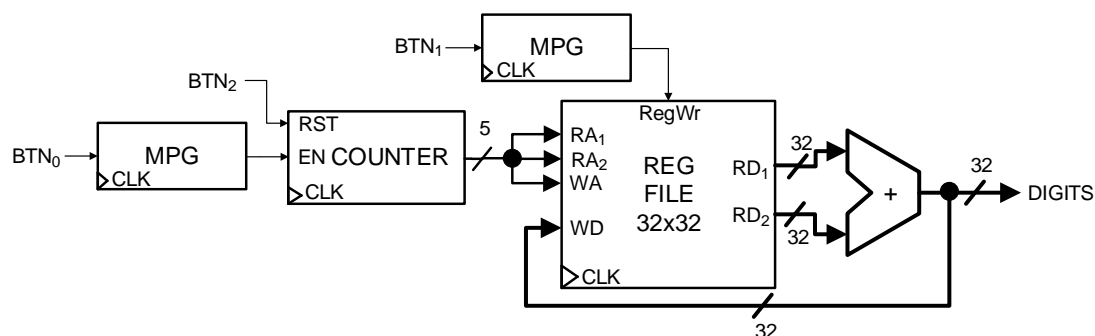


Figura 3-3: Schema de test a unui Bloc de Registre 32x32

### 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.

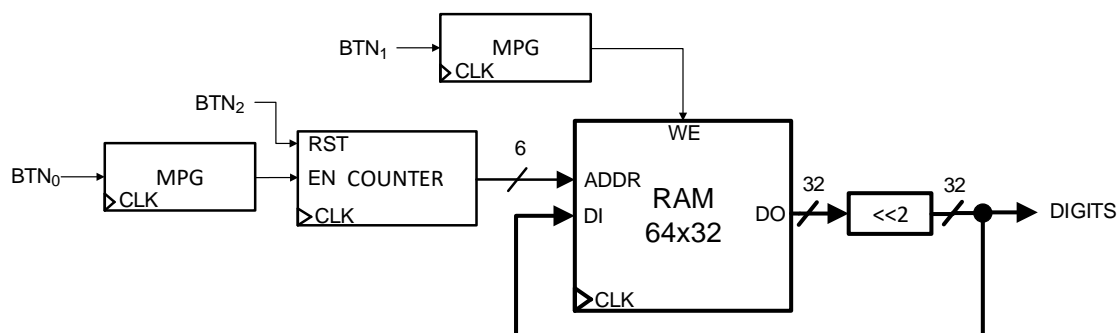


Figura 3-4: Schema de test a unui RAM 64x32 cu citire sincronă

## 3.4. Referințe

- [1] Vivado Design Suite Overview. Disponibil online: <https://docs.xilinx.com/r/en-US/ug910-vivado-getting-started/Vivado-Design-Suite-Overview>
- [2] Nexys A7 Reference Manual. Disponibil online: <https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual>
- [3] VHDL Language Reference Guide. Disponibil online: [https://peterfab.com/ref/vhdl/vhdl\\_renerta](https://peterfab.com/ref/vhdl/vhdl_renerta)
- [4] Vivado Design Suite User Guide – Synthesis (UG901): ROM HDL Coding Guidelines. Disponibil online: <https://docs.xilinx.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.xilinx.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. Descrierea procesorului MIPS pe 32 biți

**Notă:** Noțiunile din cadrul acestei lucrări sunt detaliate în cursurile 3 și 4 (care trebuie citite în prealabil) și în cartea lui Patterson și Hennessy [1]. Față de varianta standard, 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:

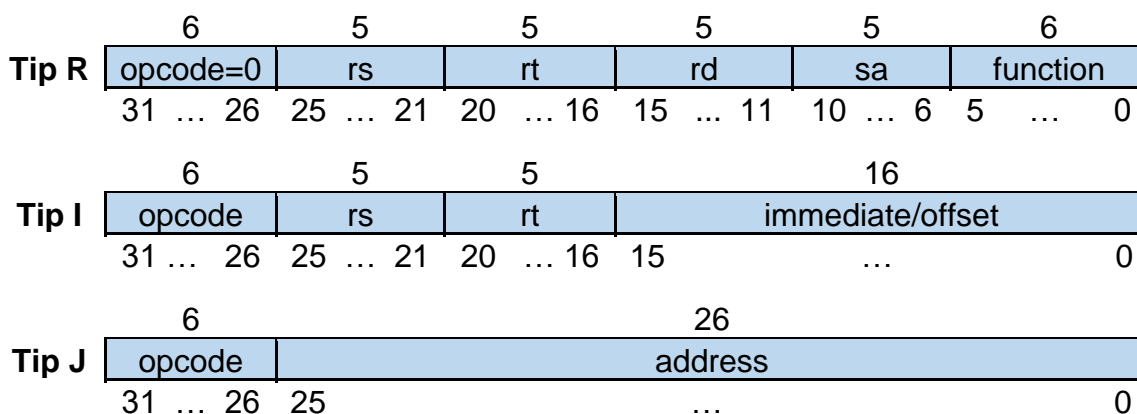


Figura 4-1: Formatul instrucțiunilor MIPS 32

Conform standardului MIPS 32 ISA, 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	Addition	add
	Subtraction	sub
	Shift Left Logical (with shift amount – sa)	sll
	Shift Right Logical (with shift amount – sa)	srl
	Logical AND	and
	Logical OR	or
	.... de definit!	
	.... de definit!	
Tip I	Add Immediate	addi
	Load Word	lw
	Store Word	sw
	Branch on Equal	beq
	.... de definit!	
	.... de definit!	
Tip J	Jump	j

Tabel 4.1: 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 clock.
- 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;
  - o intrare 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 validarea scrierii în memorie.
- Memoria de date (RAM) conține:
  - o intrare de adresă (Address);
  - o ieșire de date (Read Data) pe 32 biți pentru citirea asincronă a conținutului de la adresa curentă.
  - o intrare de date (Write Data) pe 32 biți pentru scriere sincronă la adresa curentă;
  - semnalul de control **MemWrite** pentru validarea scrierii în memorie.
- Unitatea de extindere de la 16 la 32 biți, cu următoarele funcții:
  - dacă semnalul de control **ExtOp** = 1 → extindere cu semn;
  - dacă semnalul de control **ExtOp** = 0 → extindere cu zero.
- Unități de deplasare la stânga cu 2 biți pentru alinierea adresei de salt (jump/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

**Notă:** Înainte de a începe, citiți fiecare activitate în întregime.

#### 4.3.1. Proiectarea instrucțiunilor și definirea căii de date

Adăugați la alegere, din Anexa 6, încă 2 instrucțiuni de tip R și 2 de tip I, în Tabelul 4.1. 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 $2, $4, $3 => "000000_00100_00011_00010_00000_..."` (restul biților până la 32)". Folosiți caracterul '\_' între câmpuri pentru a crește lizibilitatea;
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.

**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. Acest lucru crește lizibilitatea instrucțiunilor definite în cod-mașină. De exemplu, șirul `B"001000_01000_01100_10000_00000_000111"` este echivalent cu `"00100001000011001000000000000111"`.

#### 4.3.2. Programul de testare

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

1. o instrucțiune de scriere într-un registru, urmată de instrucțiuni care folosesc registrul respectiv ca registru sursă;
2. o instrucțiune de scriere într-o locație de memorie, urmată de instrucțiuni care vor citi acea locație de memorie și vor folosi valoarea în calcule.

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:** Extindeți programul spre ceva mai complex (12-15 instrucțiuni) încât să conțină cel puțin o buclă cu minim 2 iterații, realizată folosind instrucțiuni de salt (`jump/branch`). Opțional, puteți să vă inspirați din exemplele de la Anexa 7.



### 4.3.3. Arhitectura procesorului personalizat

**Temă:** Având ca model noțiunile din cursul 4 și Figura 5-1 din lucrarea 5, desenați arhitectura procesorului, asigurându-vă că includeți toate componentele necesare astfel încât cele 15 instrucțiuni să se execute corect. Identificați semnalele de control necesare și completați un tabel cu valorile acestora pentru fiecare instrucțiune în parte.

### 4.4. Referințe

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



## Lucrarea 5

### 5. Procesorul MIPS 32, ciclul unic – Extragerea instrucțiunilor

*Unitatea de extragere a instrucțiunilor (Instruction Fetch – IFetch)*

#### 5.1. Obiective

Descrierea, implementarea și testarea pentru:

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

#### 5.2. Descrierea procesorului MIPS pe 32 biți

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

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

Structura procesorului MIPS care implementează etapele de execuție este prezentată în figura următoare [1]. Pentru simplificare, semnalele de control nu au fost conectate explicit la destinații, dar se pot identifica ușor după nume.

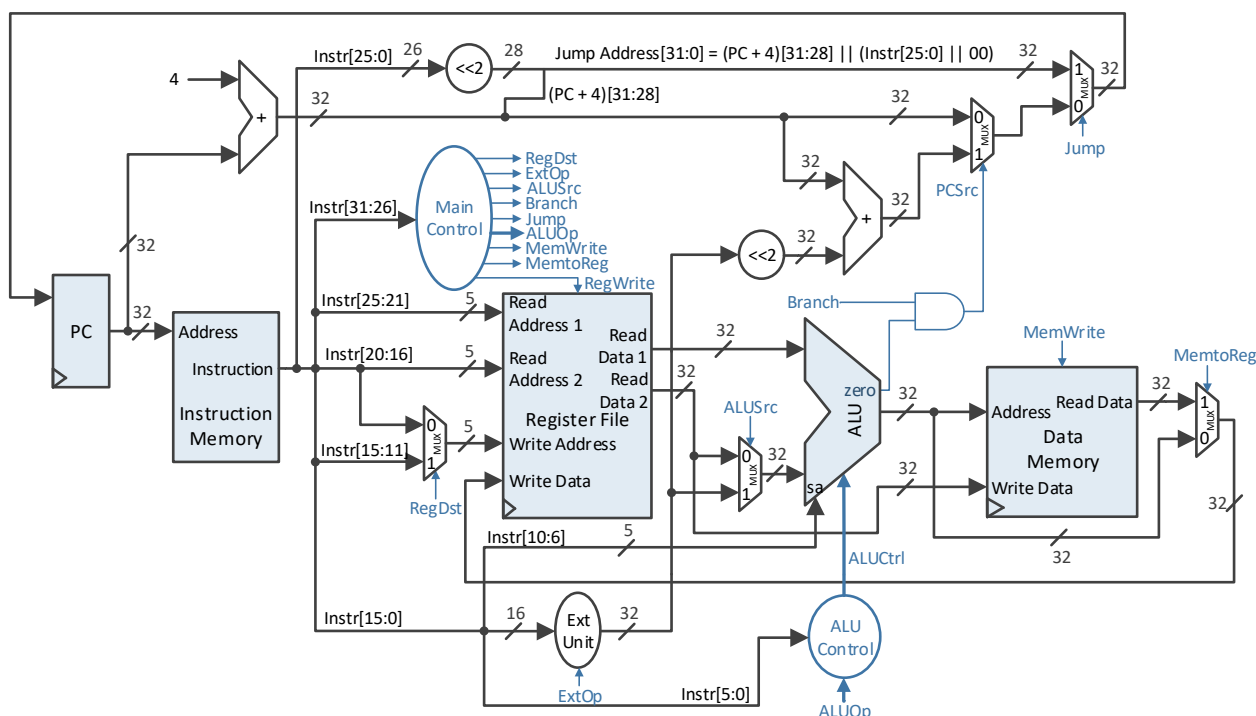


Figura 5-1: Arhitectura de ansamblu pentru MIPS 32 cu ciclul unic de ceas

Formatul instrucțiunilor diferă în funcție de categorie, astfel:

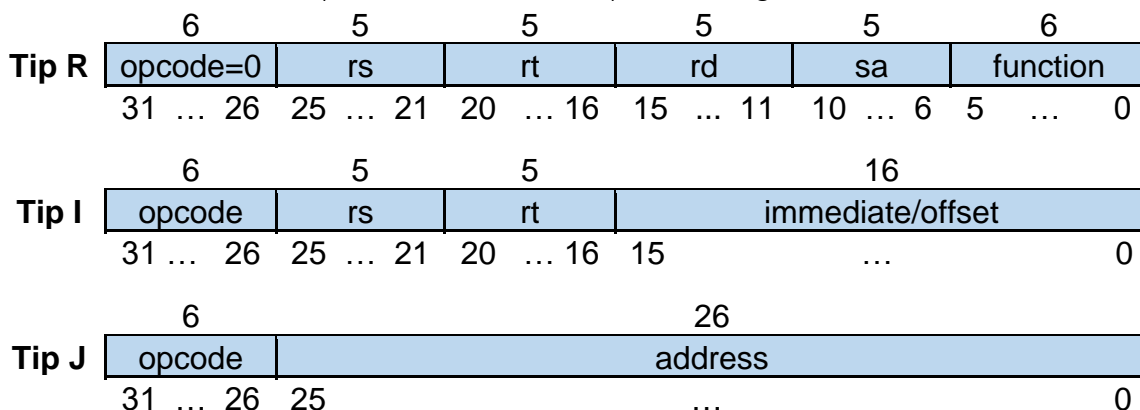


Figura 5-2: Formatul instrucțiunilor MIPS 32

Etape 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 a 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.

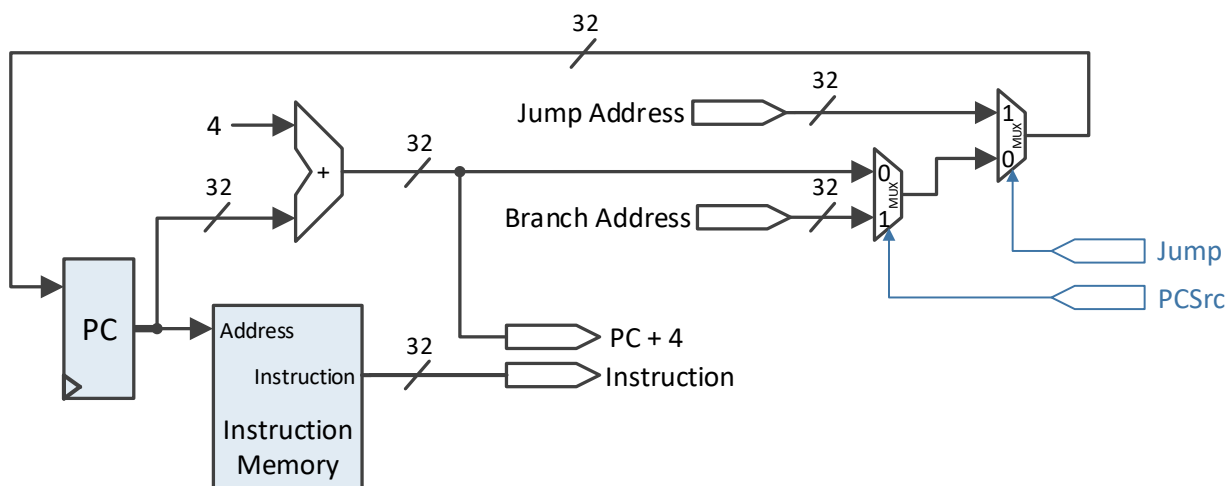


Figura 5-3: Unitatea de extragere a instrucțiunilor IFetch

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  $PC \leftarrow \text{Jump Address}$ ;
- Dacă **Jump** = 0, atunci:
  - Dacă **PCSrc** = 1, atunci  $PC \leftarrow \text{Branch Address}$ ;
  - Dacă **PCSrc** = 0, atunci  $PC \leftarrow PC+4$ .

**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 **JumpR** asociat instrucțiunii JR. Relațiile se schimbă astfel:

- Dacă **JumpR** = 1, atunci  $PC \leftarrow \text{JR Address}$ ;
- Dacă **JumpR** = 0, atunci:
  - Dacă **Jump** = 1, atunci  $PC \leftarrow \text{Jump Address}$ ;
  - Dacă **Jump** = 0, atunci:
    - Dacă **PCSrc** = 1, atunci  $PC \leftarrow \text{Branch Address}$ ;
    - Dacă **PCSrc** = 0, atunci  $PC \leftarrow PC+4$ .

### 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. Implementarea 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. Testarea unității IFetch

Declarați și instalați entitatea **IFetch** în arhitectura entității **test\_env**. Veți mai avea nevoie de MPG și SSD. Comentați/stergeți 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 printr-un mecanism de multiplexare cu MUX 2:1 controlat de la comutatorul SW<sub>7</sub>, astfel:

- dacă SW<sub>7</sub> = 0, se afișează instrucțiunea;
- dacă SW<sub>7</sub> = 1, se afișează valoarea PC+4.

Pentru simularea salturilor, semnalele de control **Jump** și **PCSrc** vor fi conectate la SW<sub>0</sub>, respectiv SW<sub>1</sub>. Dacă e nevoie de **JmpR**, conectați-l la SW<sub>2</sub>. 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<sub>10</sub> 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. Referințe

- [1] D. A. Patterson, J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
- [2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Revision 6.01, August, 2014.
- [3] 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 a instrucțiunilor (Instruction Decode – ID) și Unitatea de Control (UC)*

## 6.1. Objective

Descrierea, implementarea și testarea pentru:

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

## 6.2. Descrierea procesorului MIPS pe 32 de biți – continuare

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

Structura procesorului MIPS [1] este prezentată în continuare:

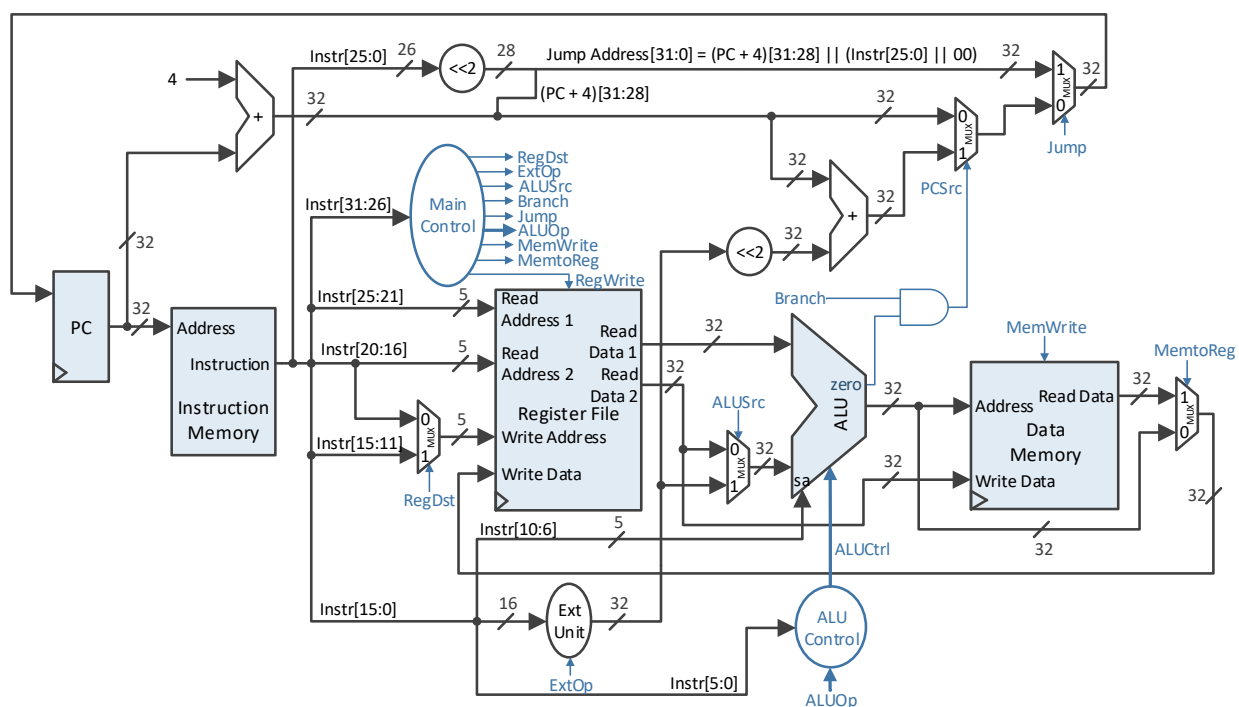


Figura 6-1: Arhitectura MIPS 32, ciclu-unic

Formatul instrucțiunilor în funcție de categorie:

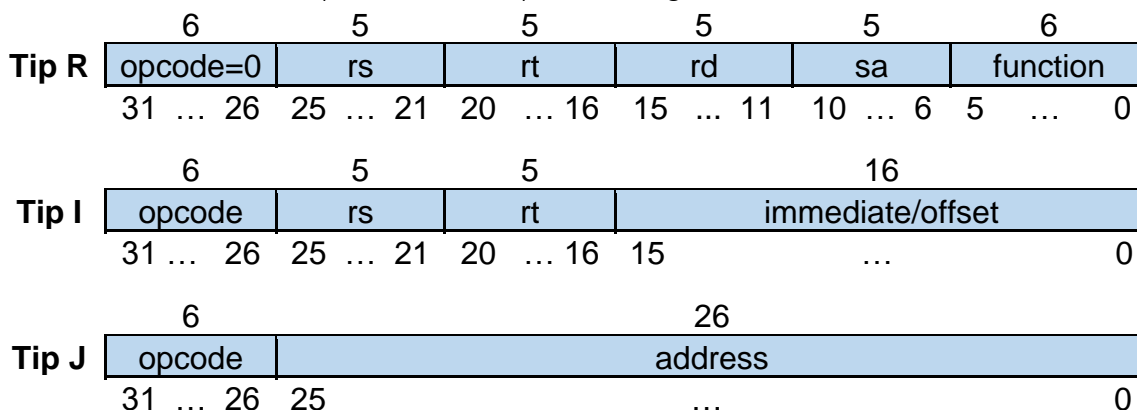


Figura 6-2: Formatul instrucțiunilor MIPS 32

Etapă 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 (immediatul extins).

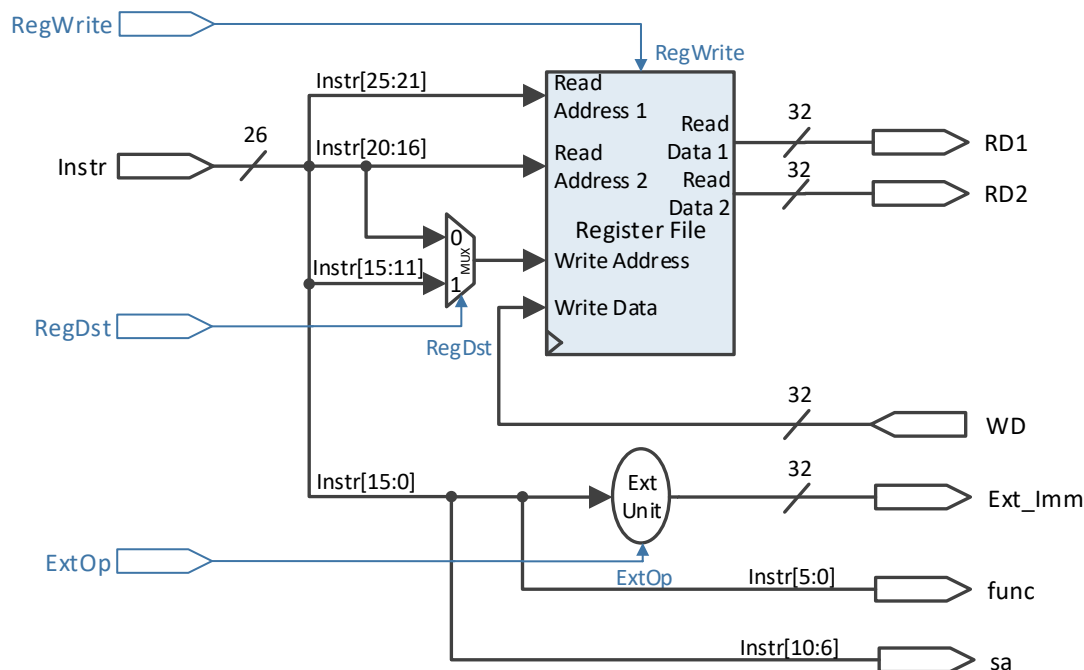


Figura 6-3: Unitatea de decodificare a instrucțiunilor ID

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 (**consultați cursul 4 sau [1] pentru detalii!**).

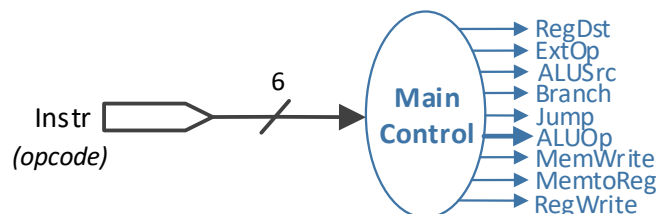


Figura 6-4: Unitatea de control UC

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. Implementarea unității ID

Describeți 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. Implementarea 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 atribue 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. Testarea unităților ID și UC

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<sub>7:5</sub>, 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. Referințe

- [1] D. A. Patterson, 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 a instrucțiunilor (Instruction Execute – EX),  
Unitatea de Memorie (MEM) și  
Unitatea de scriere a rezultatului (Write-Back – WB)*

#### 7.1. Obiective

Descrierea, implementarea și testarea pentru:

- Unitatea de execuție a instrucțiunilor (Instruction Execute – EX);
- Unitatea de Memorie (MEM);
- Unitatea de scriere a rezultatului (Write-Back – WB);
- Alte conexiuni necesare.

#### 7.2. Descrierea procesorului MIPS pe 32 de biți – continuare

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](#).

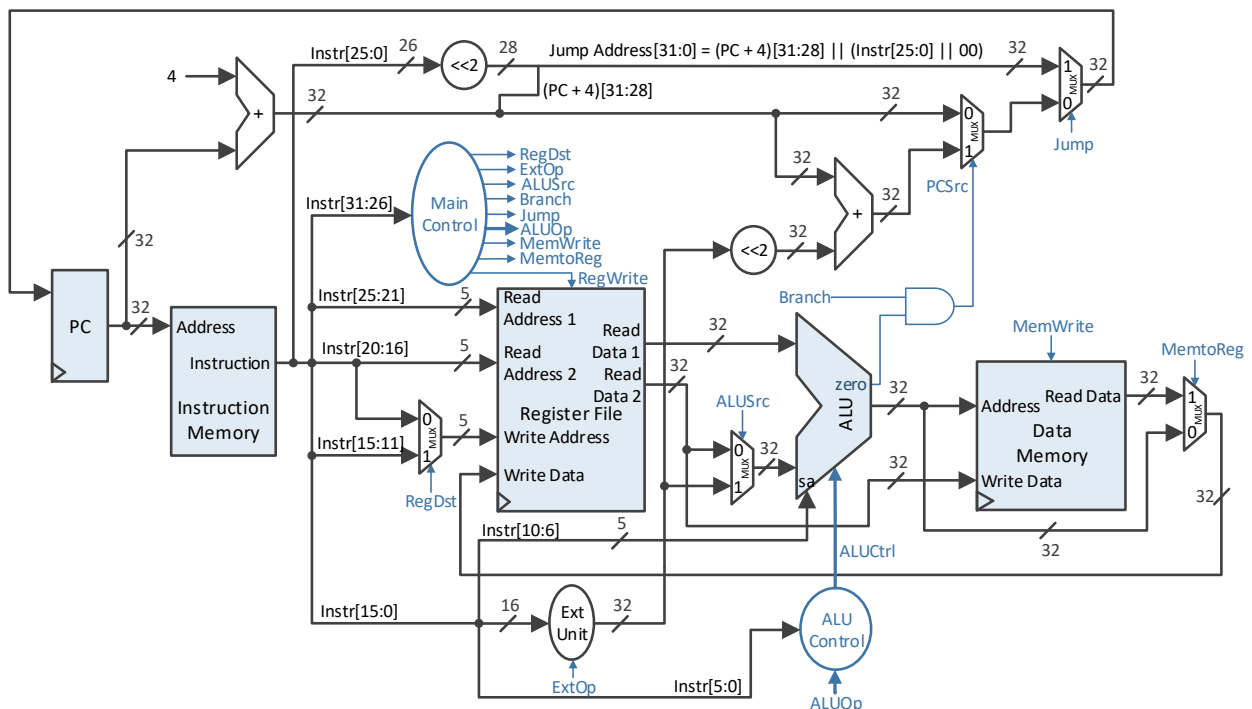


Figura 7-1: Arhitectura MIPS 32, ciclu-unic

Cele 5 etape de execuție a unei instrucțiuni sunt:

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

Formatul instrucțiunilor în funcție de categorie:

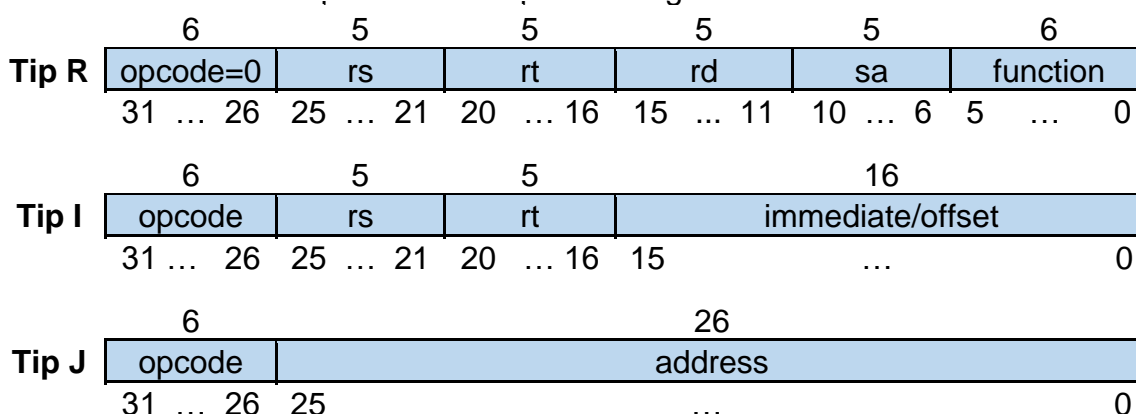


Figura 7-2: Formatul instrucțiunilor MIPS 32

Etapile 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:

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

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

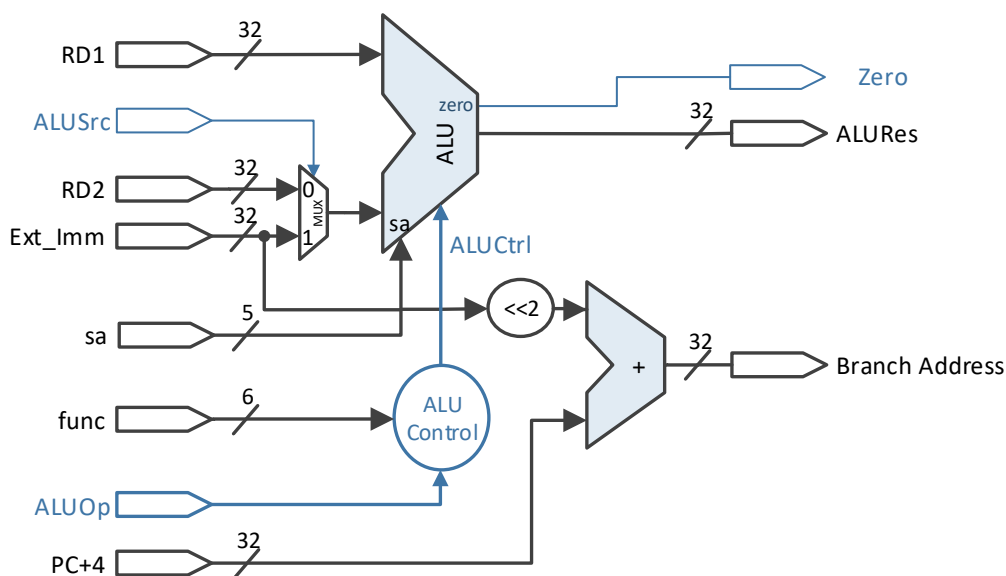


Figura 7-3: Unitatea de execuție EX

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:  $ALURes_{Out} \leftarrow ALURes_{In}$ .

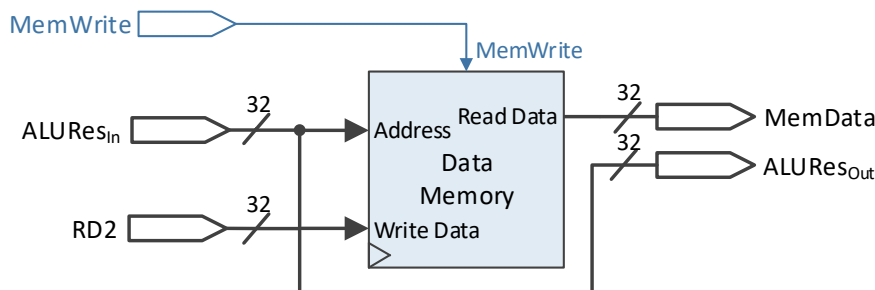


Figura 7-4: Unitatea de memorie MEM

**Unitatea de scriere a rezultatului WB (Write-Back)** constă din multiplexorul cu selecția **MemtoReg**:

- dacă **MemtoReg** = 0, se scrie **ALUResOut** în blocul de registre RF;
- dacă **MemtoReg** = 1, se scrie **MemData** în blocul de registre RF.

**Restul elementelor** neincluse în cele 3 unități (cf. Figuri 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):

$$\text{PCSrc} \leftarrow \text{Branch and Zero};$$

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

$$\text{JumpAddress} \leftarrow (\text{PC}+4)[31:28] \parallel \text{Instruction}[25:0] \parallel "00";$$

- (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. Implementarea 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 *function*.
- Definiți multiplexorul cu selecția **ALUSrc** folosind un proces sau concurențial cu **when-else**.
- Definiți unitatea ALU într-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:

$$\text{Result} \leftarrow \text{to\_stdlogicvector}(\text{to\_bitvector}(T_1) \text{ sll conv\_integer}(T_2));$$

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

$$\text{if signed}(T_1) < \text{signed}(T_2) \text{ then}$$

**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:

$$\text{BranchAddress} \leftarrow (\text{PC}+4) + \text{Ext\_Imm}[29:0] \parallel "00".$$

### 7.3.2. Implementarea 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<sub>In</sub>** 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<sub>In</sub>[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 implementării 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. Testarea procesorului

**Notă:** În funcție de timpul disponibil, testarea poate începe în cadrul acestui laborator și va fi finalizată în următorul.

Pentru verificarea semnalelor de control, acestea sunt conectate la led-uri (încă din lucrarea anterioară, activitatea 6.3.3). Există cel puțin 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<sub>7:5</sub>**:

- "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 operați sursă în instrucțiunile următoare (se pot vizualiza pe SSD).

**Temă:** Realizați trasarea execuției programului pentru a ușura testarea pe placă. Pentru fiecare instrucțiune trasată se notează valorile semnalelor care pot apărea pe SSD: RD1, RD2, ALURes, adresele de salt, etc. Dacă programul conține buclă este suficient să trasați până la finalul primei iterații.

## 7.4. Referințe

- [1] D. A. Patterson, J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
- [2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Revision 6.01, August, 2014.
- [3] 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. Testarea procesorului

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 ce afișează cu valorile corecte din tabelul realizat la Lucrarea 4, activitatea 4.3.3. Există cel puțin 8 semnale de 1 bit, și semnalul **ALUOp** pe 2-3 biți.

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:

- $SW_{7:5} = "000"$  → afișează Instruction (de la IFetch);
- $SW_{7:5} = "001"$  → afișează PC+4 (de la IFetch);
- $SW_{7:5} = "010"$  → afișează RD1 (de la ID);
- $SW_{7:5} = "011"$  → afișează RD2 (de la ID);
- $SW_{7:5} = "100"$  → afișează Ext\_Imm (de la ID);
- $SW_{7:5} = "101"$  → afișează ALURes (de la EX);
- $SW_{7:5} = "110"$  → afișează MemData (de la MEM);
- $SW_{7:5} = "111"$  → 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, prin intermediul valorilor 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 propagarea corectă a datelor plecând de la descrierea în cod mașină a instrucțiunii și apoi urmărind ieșirile din blocul 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 se procedează la fel, doar că apar ca semnale de interes adresa de memorie (ALURes) și valoarea înscrisă (RD2) sau citită din memorie (MemData).

- Pentru a testa salvarea corectă a rezultatelor în blocul de registre sau în memorie, se pot urmări valorile de la locațiile vizate când sunt folosite ca operanzi sursă în instrucțiunile viitoare. De 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. Referințe

- [1] D. A. Patterson, J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
- [2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Revision 6.01, August, 2014.
- [3] 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, versiunea pipeline.

#### 9.2. Transformarea MIPS 32 ciclu unic în arhitectură pipeline

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.

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

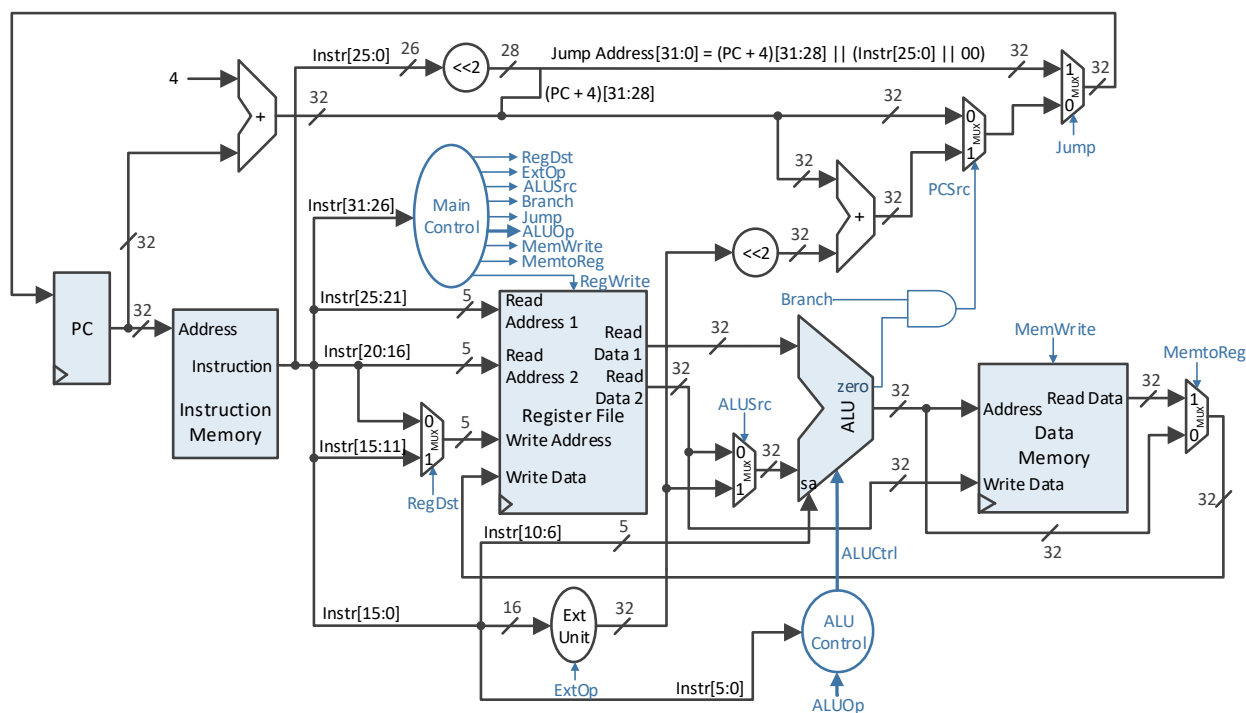


Figura 9-1: Arhitectura MIPS 32, ciclu-unic

Formatul instrucțiunilor în funcție de categorie:

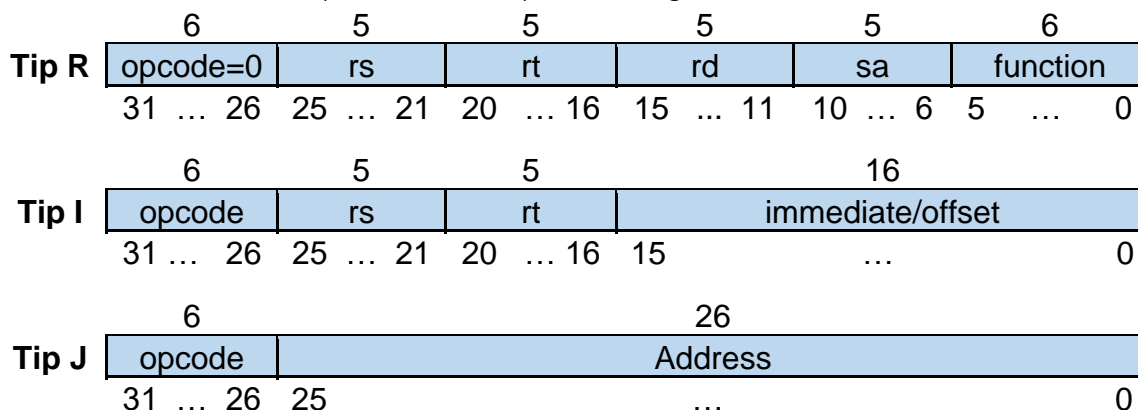


Figura 9-2: Formatul instrucțiunilor MIPS 32

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: IF, ID, EX, MEM și WB. 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 trebuiesc 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 avansate 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.

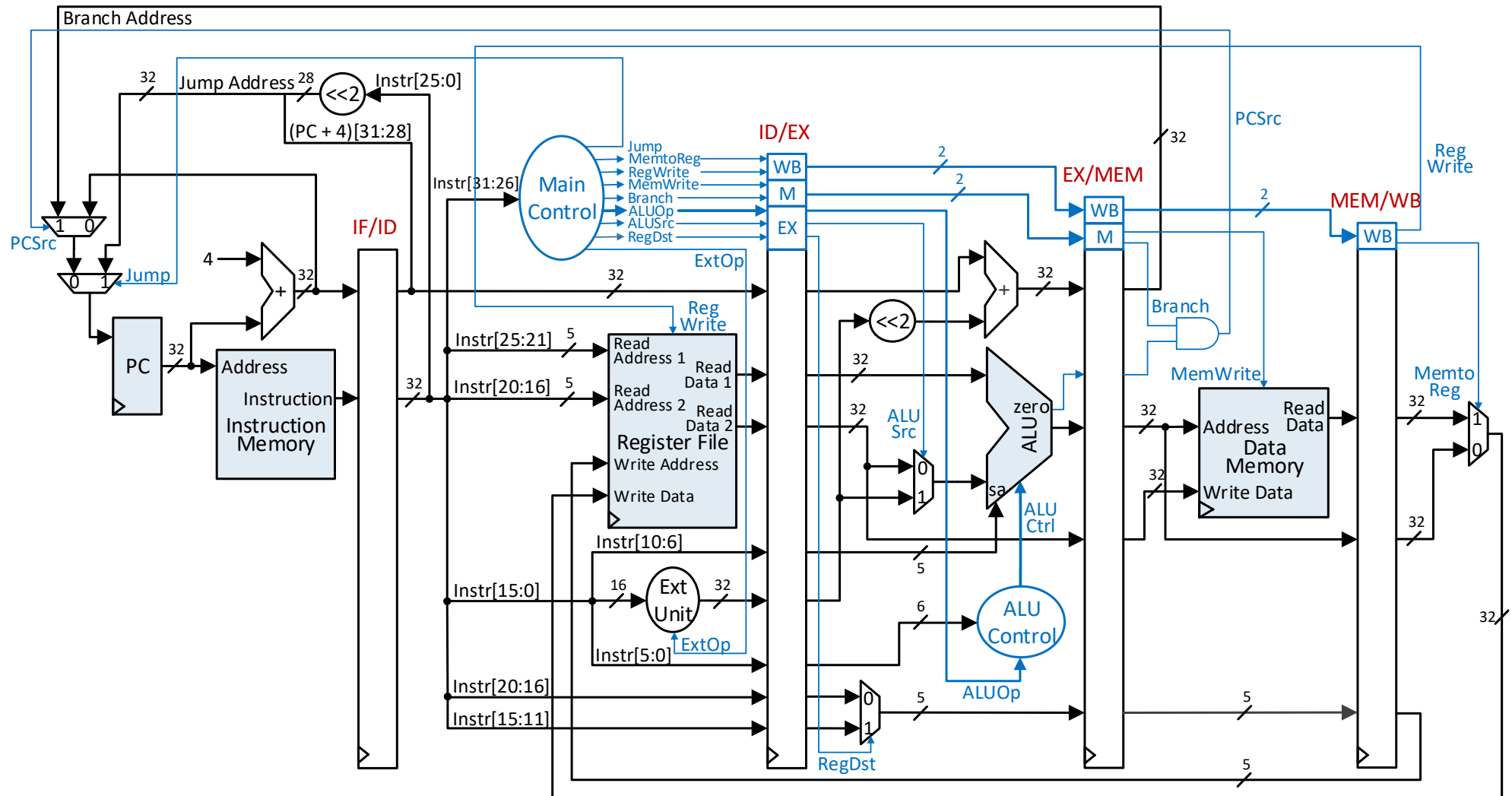


Figura 9-3: Arhitectura MIPS 32, pipeline

### 9.3. Activități practice

Resurse necesare:

- Proiectul *test\_env* care implementează MIPS 32 ciclu unic, complet.

#### 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 în laboratorul anterior se recomandă acest lucru înainte de a începe următoarele activități.

#### 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. Descrierea registrelor pipeline în VHDL

Registrele pipeline vor fi implementate în arhitectura entității **test\_env**, unde sunt instanțiate 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 potrivită (cf. cu activitatea 9.3.2), iar transferul datelor în acesta se va face sincron cu frontul de ceas și trebuie controlat prin MPG de același buton ca în cazul registrului PC. Butonul de control va facilita testarea pe pași a execuției în pipeline. Veți introduce astfel în cadrul procesului o validare suplimentară testării frontului de ceas. 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 downto 32) <= Instruction;**

În acest caz, semnalele PC+4 și Instruction provin de la entitatea **IFetch**, iar REG\_IF\_ID<sub>31:0</sub> și REG\_IF\_ID<sub>63:32</sub> 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 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 în numele semnalelor etajele adiacente). În acest caz, descrierea în cadrul procesului va fi următoarea:

*dacă front de ceas și enable atunci*

$PC\_IF\_ID \leq PC+4;$

$Instruction\_IF\_ID \leq Instruction;$

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* nu este neapărat cea situată la dreapta, ci următoarea în fluxul de date și control. 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. Referințe

- [1] D. A. Patterson, J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 5th edition. Morgan–Kaufmann, October 2013.
- [2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Revision 6.01, August, 2014.
- [3] 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 tipurilor 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 procesorul MIPS 32, 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), conform celor 5 faze de execuție a unei instrucțiuni:

- 1) IF – Extragerea instrucțiunii (Instruction Fetch);
- 2) ID/OF – Decodificarea instrucțiunii / extragerea operanzilor (Instruction Decode / Operand Fetch);
- 3) EX – Execuție (Execute);
- 4) MEM – Memorie (Memory);
- 5) WB – Scriere rezultat (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 efectele 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:

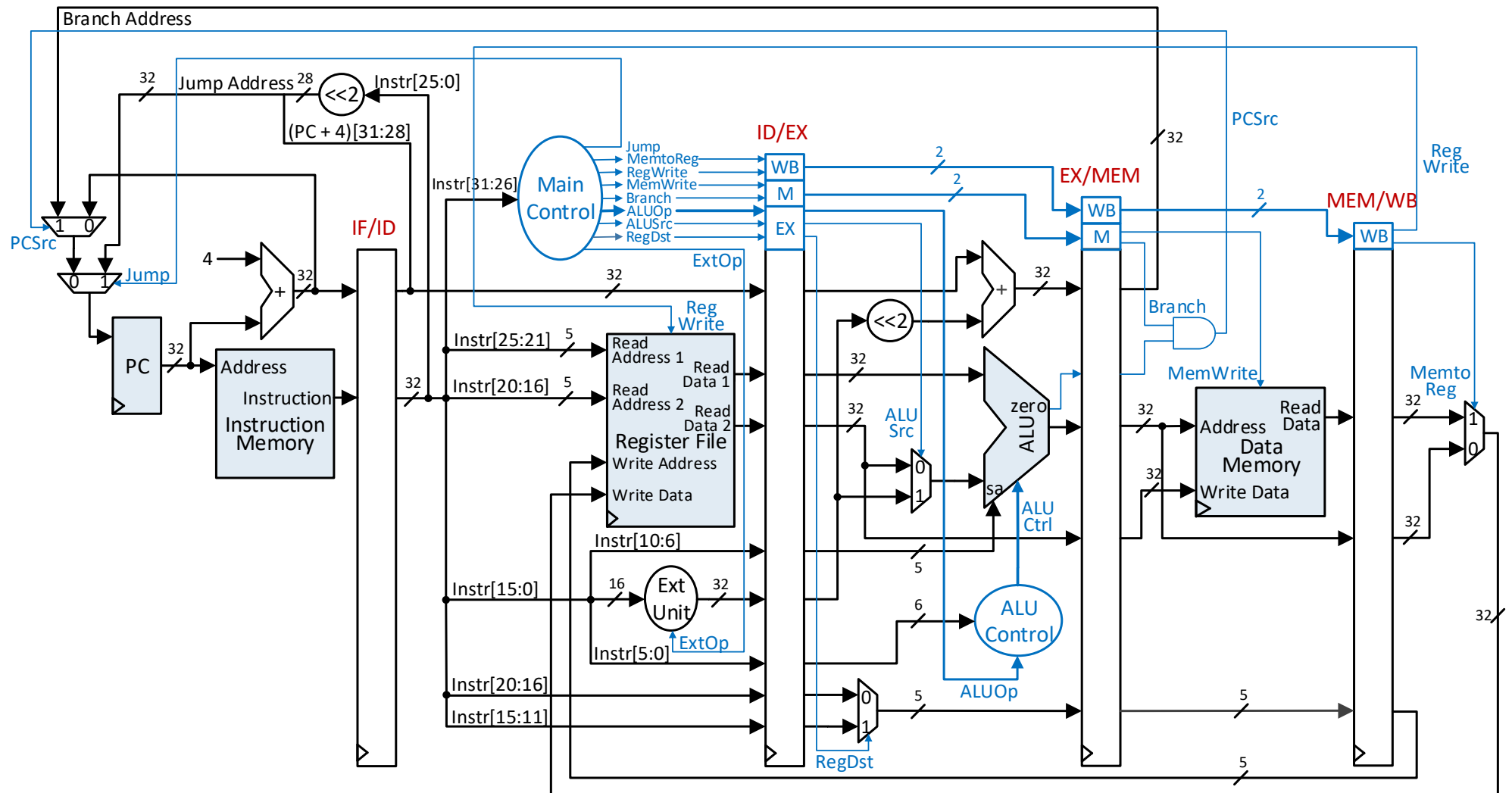


Figura 10-1: Arhitectura MIPS 32, pipeline

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.
3. *Hazard de control* – apare la instrucțiuni de branch sau jump, fiindcă decizia și adresa de salt sunt cunoscute cu întâziere, în etaje superioare etajului IF. În consecință, până la momentul saltului, în pipeline se încarcă și se execută 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), iar registrul PC se incrementează normal. Ea poate fi implementată în mai multe forme, astfel: SLL \$0, \$0, 0 sau ADD \$0, \$0, \$0 sau 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 prima instrucțiune salvează rezultatul într-un registru folosit ulterior ca operand sursă de a 3-a instrucțiune. 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<sub>5</sub>, 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<sub>5</sub>, atunci instrucțiunea SUB va citi în C<sub>5</sub> valoarea veche a registrului (dinainte de actualizare) și o va utiliza mai departe în calculele din ciclul C<sub>6</sub>. Din punct de vedere al execuției programului, această nesincronizare va genera rezultate incorecte.

Instrucțiune\Ciclu	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	C <sub>5</sub>	C <sub>6</sub>	C <sub>7</sub>	C <sub>8</sub>
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 poată fi citită asincron în partea a 2-a a perioadei de ceas, pentru a fi propagată spre etajele superioare.



2. Se introduce o instrucțiune NoOp suplimentară între instrucțiunile cu hazard, pe oricare din poziții (distanța de 3 va extinsă la 4):

```

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

```

**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 folosesc operanzi (registre) a căror valoare este în curs de prelucrare în etaje pipeline superioare și nu a fost finalizată la momentul citirii în etajul ID. 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:** Din acest punct de vedere hazardul structural definit în capitolul anterior este un hazard de date, însă datorită utilizării simultane 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:

```

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

```

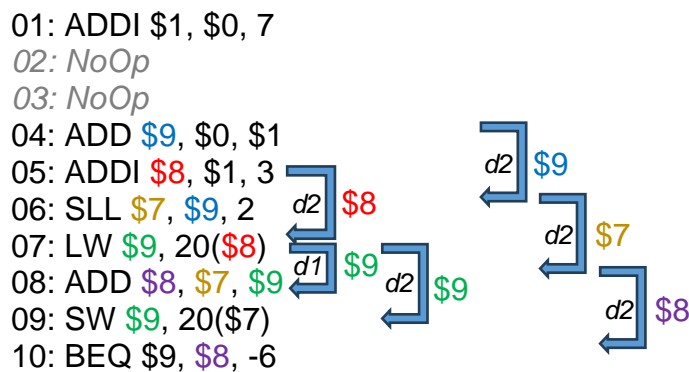
În diagrama de execuție pipeline redată mai jos, hazardurile sunt identificate analizând instrucțiunile în ordinea de apariție în cadrul programului. La fiecare hazard, registrele implicate sunt marcate cu culori distincte, evidențiind astfel scrierea târzie în etajul WB, ulterior citirii în etajul ID.

Instrucțiune\Ciclu	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	C <sub>5</sub>	C <sub>6</sub>	C <sub>7</sub>	C <sub>8</sub>	C <sub>9</sub>	C <sub>10</sub>	C <sub>11</sub>	C <sub>12</sub>
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, 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 renumerează 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 situații, 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:



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 cea situată înaintea sa, care oricum trebuie să se execute. Interschimbarea este posibilă doar dacă instrucțiunea anterioară 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 la care se face 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 corepunzătoare evidențiază efectul instrucțiunilor NoOp inserate după BEQ.

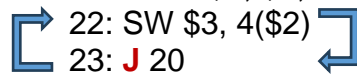
Instrucțiune\Ciclu	...	C <sub>14</sub>	C <sub>15</sub>	C <sub>16</sub>	C <sub>17</sub>	C <sub>18</sub>	C <sub>19</sub>	C <sub>20</sub>	C <sub>21</sub>	C <sub>22</sub>
14: BEQ \$9, \$8, -10		IF	ID	EX	MEM	WB				
15: NoOp			IF	ID	EX	MEM	WB			
16: NoOp				IF	ID	EX	MEM	WB		
17: NoOp					IF	ID	EX	MEM	WB	
?: 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 20              → adresa de salt neafectată 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 eliminarea lor în programul de test

Identificați hazardurile din program și eliminați-le inserând instrucțiuni NoOp, unde este cazul. Modificați adresele de salt și offset-urile, în conformitate cu re poziționarea instrucțiunilor. Desenați diagrama de execuție pipeline pentru întreg programul, înainte și după inserarea de instrucțiuni NoOp.

### 10.4.2. Testarea programului pe procesor

Converțiți 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. Referințe

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

## A. Anexa 1 – Ghid de utilizare Vivado

**Notă:** Acest ghid este adaptat pentru versiunea 2016.4, iar în versiunile mai noi pot fi mici diferențe.

### Lansarea utilitarului Vivado

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

### Crearea unui proiect nou

În cadrul aplicației, creați un nou 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...**. Va apărea o nouă fereastră de dialog. Apăsați **Next**.
2. Introduceți numele proiectului **test\_env** în câmpul Project name.
3. Introduceți o locație în câmpul Project location. Opțiunea Create project subdirectory să fie bifată pentru a crea directorul **test\_env** la locația precizată. Apăsați **Next**.
4. La tipul proiectului selectați **RTL Project**. Bifați Do not specify sources at this time. Apăsați **Next**.
5. Se aleg proprietățile plăcii conform listei de mai jos:
  - Product Category: **All**
  - Family: **Artix-7**
  - Package: **csg324**
  - Speed grade: **-1**

În tabel alegeți modelul **xc7a100tcsg324-1**. Apăsați **Next** și **Finish**.

Product category:

All

Speed grade:

-1

Family:

Artix-7

Temp grade:

All Remaining

Package:

csg324

Reset All Filters

Search:

Q






Part	I/O Pin Count	Block RAMs	DSPs	FlipFlops	GTPE2 Transce
 xc7a15tcsg324-1	324	25	45	20800	0
 xc7a35tcsg324-1	324	50	90	41600	0
 xc7a50tcsg324-1	324	75	120	65200	0
 xc7a75tcsg324-1	324	105	180	94400	0
 xc7a100tcsg324-1	324	135	240	126800	0

Figura A-1: Alegerea proprietăților plăcii

### Crearea unui fișier VHDL

1. Selectați la meniu **File → Add Sources** sau în panoul **Flow Navigator > Project Manager > Add Sources**.

2. Alegeți **Add or create design sources**, click **Next**.
3. Apăsați **Create file** și selectați la File type: **VHDL**, apoi introduceți numele fișierului **test\_env**. Numele fișierului va fi și numele entității create. Apăsați **OK**.

Figura A-2: Opțiunile pentru crearea unui fișier sursă VHDL

4. Apăsați **Finish**. Se va deschide o fereastră de definire a porturilor entității.
5. Declarați porturile (**neapărat cu litere mici!**) ca în figura următoare. **Notă:** Aceste porturi sunt suficiente pentru lucrările studiate. Apăsați **OK** pentru a finaliza crearea fișierului. Acesta va conține codul VHDL de declarare a porturilor.




Port Name	Direction	Bus	MSB	LSB
clk	in	<input type="checkbox"/>	0	0
btn	in	<input checked="" type="checkbox"/>	4	0
sw	in	<input checked="" type="checkbox"/>	15	0
led	out	<input checked="" type="checkbox"/>	15	0
an	out	<input checked="" type="checkbox"/>	7	0
cat	out	<input checked="" type="checkbox"/>	6	0

Figura A-3: Definirea porturilor

**Observație:** Se poate sări peste pasul 5 anterior (de definire a porturilor în interfață) deoarece porturile pot fi introduse explicit la secțiunea de declarare a entității, în noul fișier creat.

Fișierul care conține declararea entității **test\_env** și arhitectura ei este afișat automat pentru editare în mediul Vivado (Figura A-4). Dacă nu apare, apăsați dublu click pe **test\_env** în ierarhie la panoul **Sources** (tab-ul **Hierarchy**). Faptul că numele **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**.

**Verificare:** Accesați **Project Summary**  în bara superioară și verificați dacă la secțiunile **Synthesis** și **Implementation** câmpul Part are numele xc7a100tcs324-1. Dacă nu coincide, înseamnă că ați sărit peste pasul 5 mai sus, la **Crearea unui proiect nou**. Pentru corectare, accesați la meniu **Tools** → **Project Settings** , iar la secțiunea **General**, în dreptul Project Device apăsați  și reintroduceți proprietățile corecte ale plăcii.

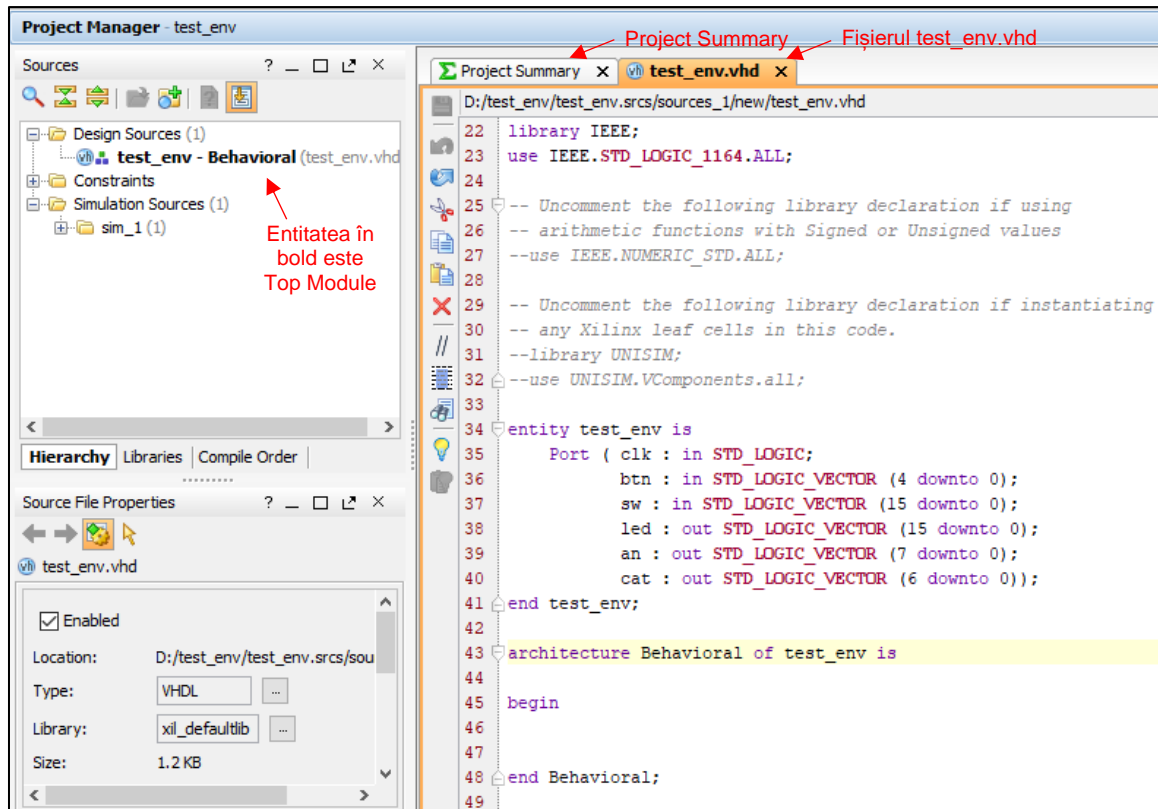



Figura A-4: Conținutul proiectului *test\_env* după creare

În fereastra de editare a entității asigurați-vă că este inclusă librăria **std\_logic\_unsigned**, iar dacă nu, adăugați-o cu:

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**  
(valabil pentru orice fișier sursă creat pe viitor!)

### Folosirea Language Templates (opțional)

Următorul pas îl reprezintă adăugarea descrierii comportamentale în cadrul arhitecturii. O modalitate facilă (recomandată la început pentru acomodarea cu VHDL) este folosirea șabloanelor de cod:

1. Plasați cursorul în fișier la locația la care doriți să adăugați codul.
2. Deschideți fereastra **Language Templates** de la meniul: **Tools → Language Templates**. Navigați în ierarhie prin simbolul  către categoria dorită de cod șablon: **VHDL > Synthesis Constructs > Coding Examples > ...**
3. Selectați componenta dorită, copiați codul și inserați-l în destinație.
4. Închideți fereastra **Language Templates**.
5. Înlocuiți denumirea implicită a semnalelor din codul inserat cu denumirea semnalelor din entitatea pe care o descrieți.

### 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 instanțierile de componente cu *port map*, descrierea comportamentală cu procese și atribuirile concurente.



2. Pentru acest prim proiect adăugați următoarele atribuiri concurente după **begin** (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 cu **File → Save File** sau **Ctrl+S**.
4. Selectați entitatea **test\_env** în ierarhia din panoul **Sources**.
5. Vizualizați circuitul rezultat sub formă schematică pentru a verifica dacă ați legat corect elementele acestuia: panoul **Flow Navigator > RTL Analysis > Open Elaborated Design** → (eventual) click **OK**, dacă apar alte ferestre de dialog. (corectați eventuale probleme raportate în panoul **Messages**, situat în partea de jos)  
Se va deschide o schemă bloc a circuitului principal. 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 încadrate ca **Errors** sau **Critical Warnings** în panoul **Messages**. 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.

### Stabilirea resurselor utilizate în fișierul de constrângeri

Specificați resursele de pe placă atribuite porturilor din entitatea top level, **test\_env**:

1. Pentru simplificare, resursele necesare au fost definite în fișierul [NexysA7\\_test\\_env.xdc](#) [1]. Descărcați fișierul și asigurați-vă că are extensia corectă .xdc .
2. Adăugați fișierul la proiect accesând **File → Add Sources** sau în panoul **Flow Navigator > Project Manager > Add Sources**.
3. Alegeți **Add or create constraints**, click **Next**.
4. Apăsați **Add files** → selectați fișierul de constrângeri pe disc (opțiunea **Copy constraints files into project** să fie bifată) → click **OK** → click **Finish**.

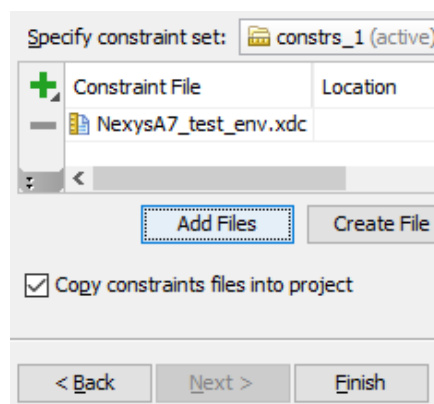


Figura A-5: Adăugarea fișierului de constrângeri la proiect

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



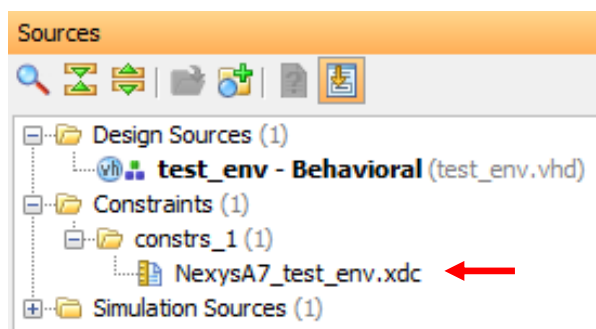


Figura A-6: Prezența fișierului de contrângeri în ierarhie




Apăsați dublu-click pe el pentru vizualizarea conținutului și editare.  
**Observație:** Cu excepția semnalului clk celelalte resurse sunt definite printr-un singur rând. Locația resurselor apare în fișier după cuvântul rezervat PACKAGE\_PIN, iar pe placă, între paranteze, în dreptul resursei (Figura A-7).



Figura A-7: Locația resurselor se află imprimată pe placă între paranteze. De exemplu butonul de jos 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, și click **Cancel**, la final, dacă apare fereastra **Synthesis Completed**.
2. Lansați etapa de implementare: **Flow Navigator > Implementation > Run Implementation**  → (eventual) click **OK**, dacă apar alte ferestre de dialog, și click **Cancel**, la final, dacă apare fereastra **Implementation Completed**. Ignorați acele avertismente care fac referire la semnale neconectate, care 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, și click **Cancel**, la final, dacă apare fereastra **Bitstream Generation Completed**. Se va crea un fișier cu extensia .bit care se va încărca pe placă la pasul următor.

**Recomandări:** Puteți lansa direct 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 să fie conectată placa la portul USB și alimentați-o punând comutatorul POWER pe ON. Se va activa led-ul de alimentare.
2. Accesați panoul **Flow Navigator > Program and Debug > Open Hardware Manager > Open Target → Auto Connect**.

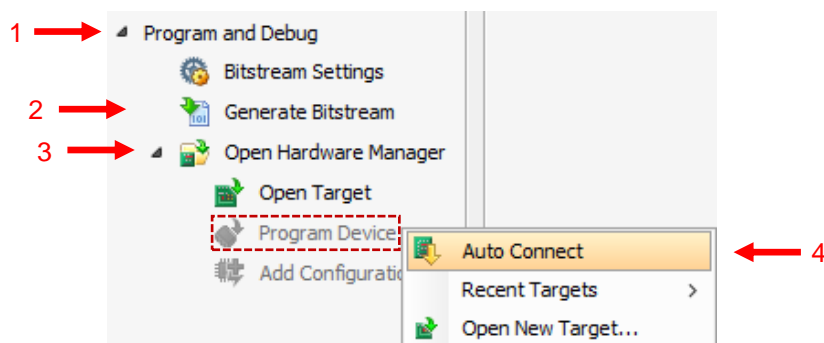


Figura A-8: Pașii de conectare la placa de dezvoltare

Dacă placa a fost recunoscută, ar trebui ca opțiunea **Program Device** (marcată cu un dreptunghi în Figura A-8) să fie activă, altfel accesați **Flow Navigator > Program and Debug > Open Hardware Manager > Open Target → Open New Target...** și urmați pașii cu click pe **Next**.

3. Accesați **Program Device** și apoi click în meniul afișat.
4. În dialogul deschis selectați calea spre fișierul .bit dacă nu este completată automat (acest fișier se găsește în directorul proiectului `test_env.runs/impl_1/`). Apăsați **Program**.

După încărcarea pe placă realizați testarea practică a circuitului folosind comutatoarele și butoanele.

**Important:** Conexiunea rămâne activă cât timp placa este alimentată. Nu opriți alimentarea și astfel pentru reprogramare va fi suficient să reluați doar ultimii doi pași.

**Notă:** Dacă la pasul al 2-lea placa nu este recunoscută încercați în ordine:

- 1) Schimbați cablul;
- 2) Schimbați portul USB;
- 3) Reporniți Vivado;
- 4) Reporniți stația;
- 5) Schimbați placa.

## Referințe

- [1] Fișierul de constrângeri NexysA7\_test\_env.xdc. Disponibil online:  
<https://drive.google.com/file/d/1I8D1splyVvYC9mD0I3jAC07qjRgar19Qr/view?usp=sharing>

## B. Anexa 2 – Simularea funcțională în mediul Vivado

**Notă:** Acest ghid este adaptat pentru versiunea 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 1.5.4. **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).

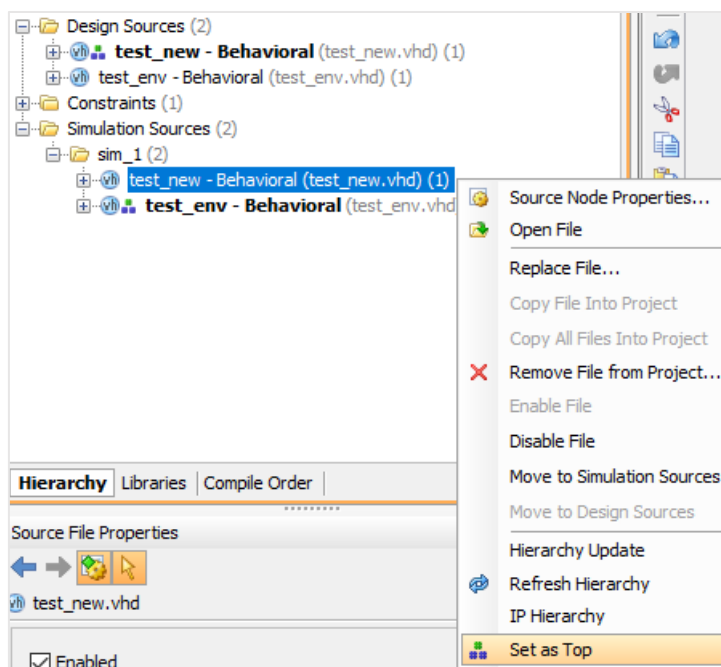



Figura B-1: Arhitectura simulată trebuie să fie Top Module

### Inițializarea simulatorului și elementele constitutive

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**.

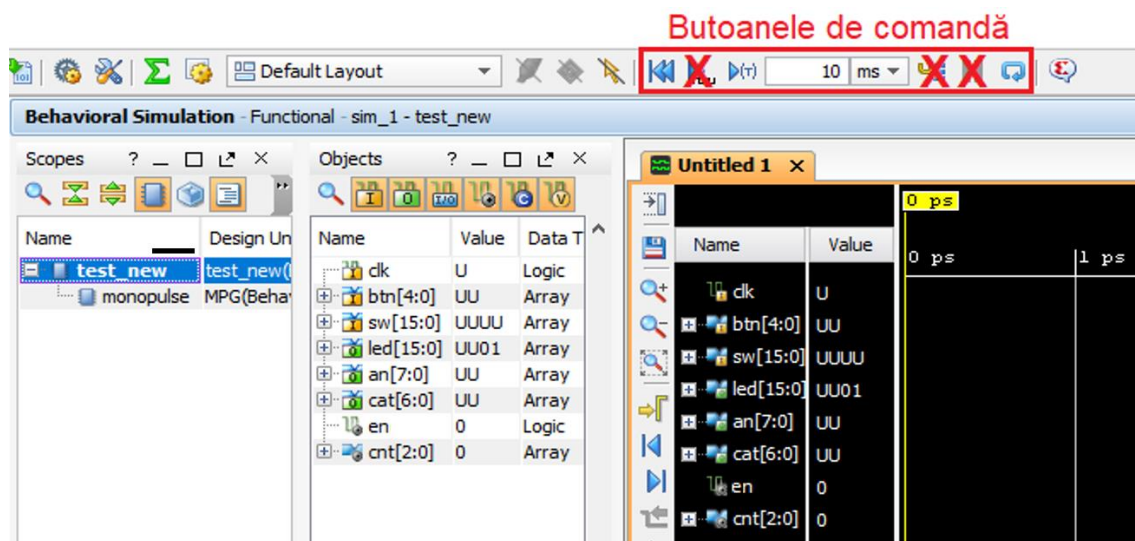


Figura B-2: Elementele mediului de simulare Vivado

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.

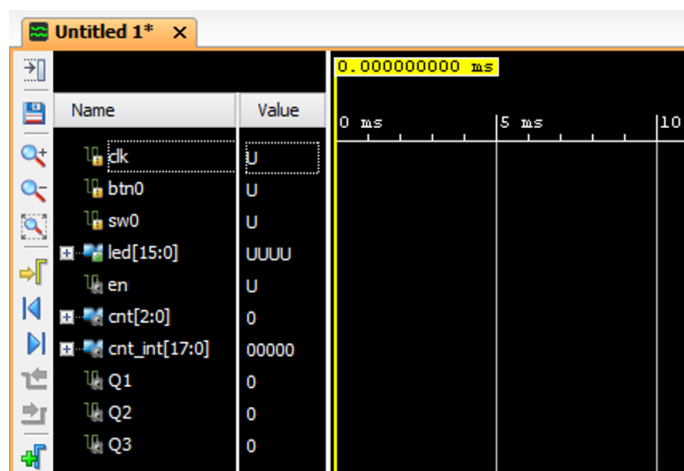


Figura B-3: Lista semnalelor de urmărit în timpul simulării

### 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 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 aceiași 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.

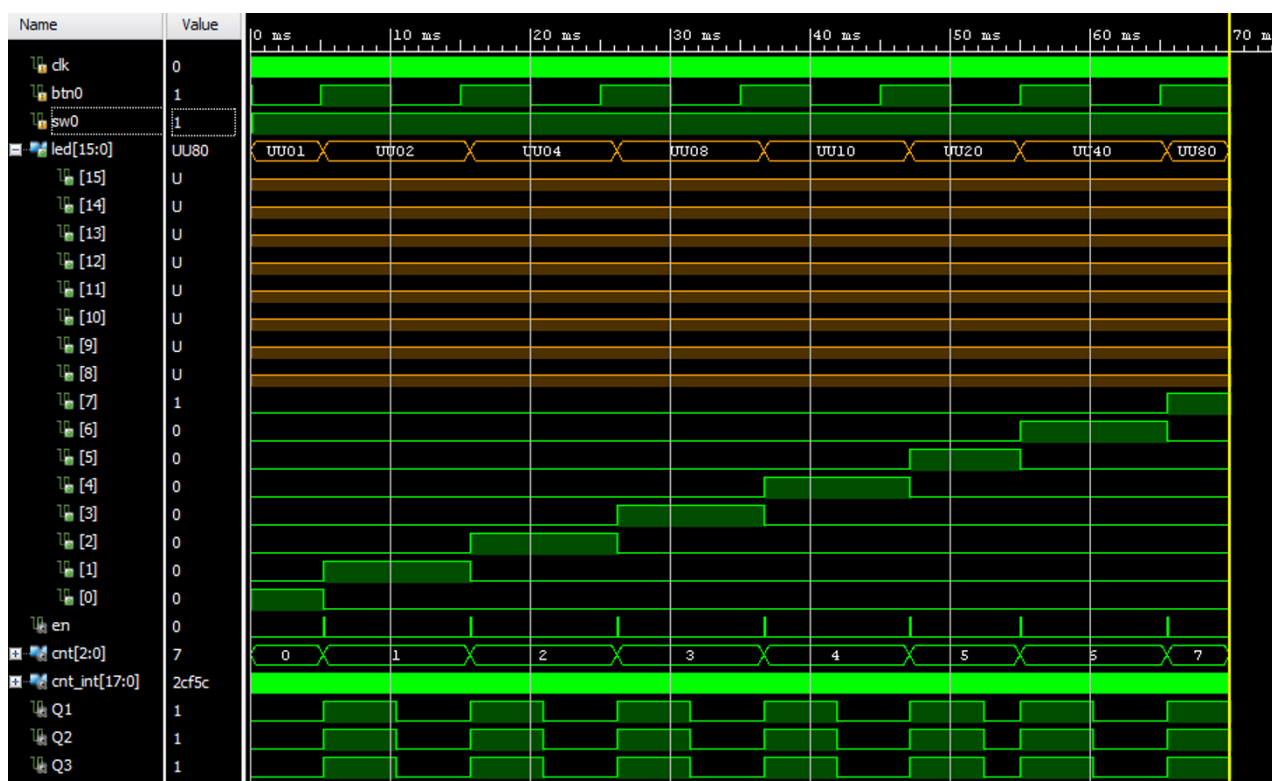




Figura B-5: Formele de undă generate în timpul simulării


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 pe 8 biți

Un circuit de deplasare cu număr variabil de poziții se poate implementa ca o secvență multi-nivel de multiplexoare MUX 2:1. Ieșirile de pe fiecare nivel se conectează deplasat la intrările următorului nivel, în funcție de distanța de deplasare dorită. În figura următoare se prezintă un circuit de deplasare la dreapta pe 8 biți, realizat cu 3 niveluri de multiplexoare.

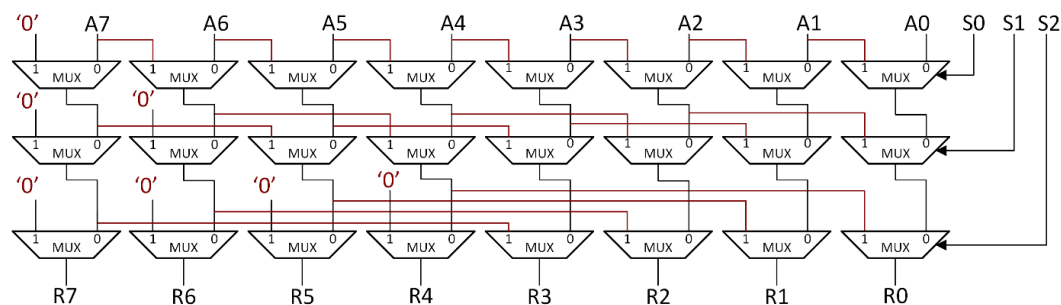


Figura C-1: Circuit de deplasare la dreapta realizat cu multiplexoare

Exemplu în VHDL, de circuit de deplasare pe 8 biți, cu număr variabil de poziții:

- SW<sub>7:0</sub> – semnalul deplasat la stânga sau la dreapta aritmetic;
- SW<sub>9:8</sub> – câte poziții se deplasează: 0, 1, 2 sau 3;
- SW<sub>10</sub> – direcția de deplasare: 0 – la stânga; 1 – la dreapta, aritmetic.

process(sw) -- *deplasare cu 1 poziție*

begin

if sw(8) = '1' then

if sw(10) = '0' then -- *deplasare la stânga*

shift1 <= sw(6 downto 0) & '0';

else

-- *deplasare aritmetică la dreapta*

shift1 <= sw(7) & sw(7 downto 1);

end if;

else

shift1 <= sw(7 downto 0);

end if;

end process;

process(sw, shift1) -- *deplasare cu 2 poziții*

begin

if sw(9) = '1' then

if sw(10) = '0' then -- *deplasare la stânga*

shift2 <= shift1(5 downto 0) & "00";

else

-- *deplasare aritmetică la dreapta*

shift2 <= shift1(7) & shift1(7) & shift1(7 downto 2);

end if;

else

shift2 <= shift1;

end if;

end process;

led <= shift2;



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

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

entity reg_file is
port ( clk      : in std_logic;
      ra1      : in std_logic_vector(4 downto 0);
      ra2      : in std_logic_vector(4 downto 0);
      wa      : in std_logic_vector(4 downto 0);
      wd      : in std_logic_vector(31 downto 0);
      regwr    : in std_logic;
      rd1      : out std_logic_vector(31 downto 0);
      rd2      : out std_logic_vector(31 downto 0));
end reg_file;

architecture Behavioral of reg_file is

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

begin

    process(clk)
    begin
        if rising_edge(clk) then
            if regwr = '1' then
                reg_file(conv_integer(wa)) <= wd;
            end if;
        end if;
    end process;

    rd1 <= reg_file(conv_integer(ra1));
    rd2 <= reg_file(conv_integer(ra2));

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;
      en      : in std_logic; -- optional
      addr    : in std_logic_vector(5 downto 0);
      di      : in std_logic_vector(31 downto 0);
      do      : out std_logic_vector(31 downto 0));
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 ram : ram_type := (
    others => X"00000000");

begin

    process(clk)
    begin
        if rising_edge(clk) then
            if en = '1' then -- optional
                if we = '1' then
                    ram(conv_integer(addr)) <= di;
                    do <= di;
                else
                    do <= ram(conv_integer(addr));
                end if;
            end if;
        end if;
    end process;

end Behavioral;
```

## F. Anexa 6 – Instrucțiuni pentru MIPS 32

Pentru fiecare instrucțiune se prezintă: descrierea, RTL abstract, sintaxa în asamblare și formatul pe biți. **Notă:** Setul complet de instrucțiuni poate fi consultat în referințele din lucrarea 4, cu subiectul MIPS® Architecture for Programmers.

### ADD – ADD

Descriere	Adună două registre și memorează rezultatul în al treilea
RTL	$\$d \leftarrow \$s + \$t; PC \leftarrow PC + 4;$
Sintaxă	add \$d, \$s, \$t
Format	000000 sssss tttt ddddd 00000 100000

### ADDI – ADD Immediate

Descriere	Adună un registru cu o valoare imediată și memorează rezultatul în alt registru
RTL	$\$t \leftarrow \$s + SE(imm); PC \leftarrow PC + 4;$
Sintaxă	addi \$t, \$s, imm
Format	001000 sssss tttt iiiiiiiiiiiii

### AND – bitwise AND

Descriere	ȘI logic între două registre cu memorarea rezultatului în al treilea
RTL	$\$d \leftarrow \$s \& \$t; PC \leftarrow PC + 4;$
Sintaxă	and \$d, \$s, \$t
Format	000000 sssss tttt ddddd 00000 100100

### ANDI – AND Immediate

Descriere	ȘI logic între un registru și o valoare imediată, cu rezultatul în alt registru
RTL	$\$t \leftarrow \$s \& ZE(imm); PC \leftarrow PC + 4;$
Sintaxă	andi \$t, \$s, imm
Format	001100 sssss tttt iiiiiiiiiiiii

### BEQ – Branch on Equal

Descriere	Salt condiționat dacă este egalitate între două registre
RTL	if $\$s = \$t$ then $PC \leftarrow (PC + 4) + (SE(offset) \ll 2)$ else $PC \leftarrow PC + 4;$
Sintaxă	beq \$s, \$t, offset
Format	000100 sssss tttt oooooooooooooooooo

### BGEZ – Branch on Greater than or Equal to Zero

Descriere	Salt condiționat dacă un registru este mai mare sau egal cu 0
RTL	if $\$s \geq 0$ then $PC \leftarrow (PC + 4) + (SE(offset) \ll 2)$ else $PC \leftarrow PC + 4;$
Sintaxă	bgez \$s, offset
Format	000001 sssss 00000 oooooooooooooooooo

### BGTZ – Branch on Greater Than Zero

Descriere	Salt condiționat dacă un registru este mai mare ca 0
RTL	If $\$s > 0$ then $PC \leftarrow (PC + 4) + (SE(offset) \ll 2)$ else $PC \leftarrow PC + 4;$
Sintaxă	bgtz \$s, offset
Format	000111 sssss 00000 oooooooooooooooooo

## BNE – Branch on Not Equal

Descriere	Salt condiționat dacă două registre sunt diferite
RTL	if $\$s \neq \$t$ then $PC \leftarrow (PC + 4) + (SE(offset) \ll 2)$ else $PC \leftarrow PC + 4$ ;
Sintaxă	bne $\$s, \$t, offset$
Format	000101 sssss tttt 0000000000000000

## J – Jump

Descriere	Salt la adresă absolută
RTL	$PC \leftarrow (PC + 4)[31:28] \parallel (addr \ll 2)$ ;
Sintaxă	j addr
Format	000010 aaaaaaaaaaaaaaaaaaaaaaaaaa

## JR – Jump Register

Descriere	Salt la adresa absolută conținută de registrul $\$s$
RTL	$PC \leftarrow \$s$ ;
Sintaxă	jr $\$s$
Format	000000 sssss 00000 00000 00000 001000

## LW – Load Word

Descriere	Un cuvânt din memorie este încărcat într-un registru
RTL	$\$t \leftarrow MEM[\$s + SE(offset)]$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	lw $\$t, offset(\$s)$
Format	100011 sssss tttt 0000000000000000

NOOP – NO OPeration, echivalentă cu SLL  $\$0, \$0, 0$  (fără efect în procesor)

Descriere	Nu se efectuează nicio operație
RTL	$\$0 \leftarrow \$0 \ll 0$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	noop
Format	000000 00000 00000 00000 00000 000000

## OR – bitwise OR

Descriere	SAU logic între două registre și memorează rezultatul în al treilea
RTL	$\$d \leftarrow \$s \mid \$t$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	or $\$d, \$s, \$t$
Format	000000 sssss tttt dddd 00000 100101

## ORI – bitwise OR Immediate

Descriere	SAU logic între un registru și o valoare imediată, memorează rezultatul în alt registru
RTL	$\$t \leftarrow \$s \mid ZE(imm)$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	ori $\$t, \$s, imm$
Format	001101 sssss tttt iiii

## SLL – Shift-Left Logical

Descriere	Deplasare logică la stânga pentru un registru, rezultatul este memorat în alt registru, se introduc zerouri
RTL	$\$d \leftarrow \$t \ll h$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	sll $\$d, \$t, h$
Format	000000 00000 tttt dddd hhhh 000000

## SLLV – Shift-Left Logical Variable

Descriere	Deplasare logică la stânga pentru un registru, cu un număr de poziții indicat de alt registru, iar rezultatul este memorat într-un al treilea registru
RTL	$\$d \leftarrow \$t \ll \$s$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	sllv \$d, \$t, \$s
Format	000000 sssss ttttt ddddd 00000 000100

## SLT – Set on Less Than (signed)

Descriere	Dacă $\$s < \$t$ , \$d este inițializat cu 1, altfel cu 0
RTL	$PC \leftarrow PC + 4$ ; if $\$s < \$t$ then $\$d \leftarrow 1$ else $\$d \leftarrow 0$ ;
Sintaxă	slt \$d, \$s, \$t
Format	000000 sssss ttttt ddddd 00000 101010

## SLTI – Set on Less Than Immediate (signed)

Descriere	Dacă \$s este mai mic decât un imediat, \$t este inițializat cu 1, altfel cu 0
RTL	$PC \leftarrow PC + 4$ ; if $\$s < SE(imm)$ then $\$t \leftarrow 1$ else $\$t \leftarrow 0$ ;
Sintaxă	slti \$t, \$s, imm
Format	001010 sssss ttttt iiiiiiiiiiiiii

## SRA – Shift-Right Arithmetic

Descriere	Deplasare aritmetică la dreapta pentru un registru, rezultatul este memorat în altul. Se repetă valoarea bitului de semn
RTL	$\$d \leftarrow \$t \gg h$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	sra \$d, \$t, h
Format	000000 00000 ttttt ddddd hhhhh 000011

## SRL – Shift-Right Logical

Descriere	Deplasare logică la dreapta pentru un registru, rezultatul este memorat în altul, se introduc zerouri
RTL	$\$d \leftarrow \$t \gg h$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	srl \$d, \$t, h
Format	000000 00000 ttttt ddddd hhhhh 000010

## SUB – SUBtract

Descriere	Scade două registre și memorează rezultatul în al treilea
RTL	$\$d \leftarrow \$s - \$t$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	sub \$d, \$s, \$t
Format	000000 sssss ttttt ddddd 00000 100010

## SW – Store Word

Descriere	Valoarea unui registru este stocată în memorie la o anumită adresă
RTL	$MEM[\$s + SE(offset)] \leftarrow \$t$ ; $PC \leftarrow PC + 4$ ;
Sintaxă	sw \$t, offset(\$s)
Format	101011 sssss ttttt oooooooooooooooooo

## XOR – bitwise eXclusive-OR

Descriere	SAU-Exclusiv logic între două registre, memorează rezultatul în alt registru
RTL	$\$d \leftarrow \$s \wedge \$t$ ; $PC \leftarrow PC + 4$ ;
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  $A$  ( $A \geq 8$ ) 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  $A$  ( $A \geq 8$ ). 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 tripul lor, dacă sunt mai mari decât  $X$ , altfel cu jumătatea lor obținută prin împărțire întreagă. Șirul se află în memorie începând cu adresa  $A$  ( $A \geq 12$ ) 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-cmmm-c-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  $A$  ( $A \geq 16$ ), 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  $\leq 2^{31}$  citit din memorie de la adresa 0. Rezultatul se va scrie în memorie la adresa  $A$  ( $A \geq 8$ ). Valoarea lui  $A$  se citește din memorie de la adresa 4.
11. Să se determine câte elemente se află în submulțimea cel mai lungă de numere ordonate crescător, dintr-un șir de  $N$  numere. Șirul începe în memorie de la adresa  $A$  ( $A \geq 12$ ).  $A$  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  $A$  ( $A \geq 12$ ).  $A$  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  $A$  ( $A \geq 12$ ) 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  $A$  ( $A \geq 12$ ).  $A$  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  $A$  ( $A \geq 8$ ). O iterație presupune parcurgerea elementelor de la primul la ultimul și fiecare element se interchimbă 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  $X$  ( $X \leq 255$ ) și  $Y$  ( $Y \leq 255$ ) 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  $A$  ( $A \geq 12$ ).  $A$  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 – Implementări pentru automatele cu stări finite

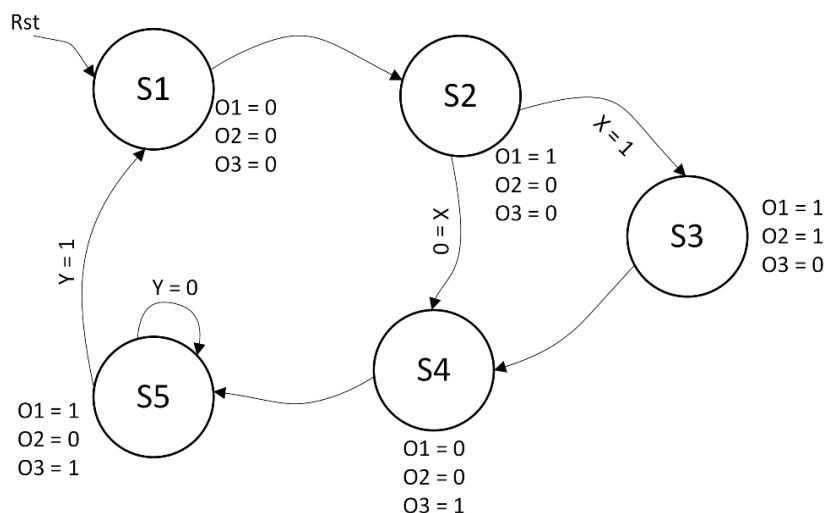


Figura H-1: Exemplu FSM

Pini intrare/ieșire	Descriere
clk	Semnal de ceas (front ascendent)
Rst	Reset asincron (activ pe 1)
X, Y	Intrări FSM
O1, O2, O3	Ieșiri FSM

Tabelul H.1: Descrierea pinilor FSM

În mod implicit, Xilinx încearcă să recunoască FSM-urile scrise în codul VHDL. Pe paginile următoare se prezintă cele 3 tipuri de descriere pentru FSM-ul prezentat în Figura G-1, care sunt recunoscute automat de Xilinx, la sintetizare: cu 1, 2 sau 3 procese.

În implementarea cu un proces, toate atribuirile (starea următoare, decodificarea semnalelor de ieșire) se fac sincron, pe frontul ascendent de ceas.

În implementarea cu 2 procese, în primul proces se calculează și atribuie sincron starea următoare. Decodificarea stării curente, pentru a stabili valoarea ieșirilor, se face combinațional în al doilea proces.

În varianta de implementare cu 3 procese, două sunt combinaționale: cel care decodifică starea curentă în valoarea ieșirilor și cel care calculează valoarea stării următoare. Al treilea proces este sincron, realizând tranziția între stări (atribuirea sincronă a stării următoare la starea curentă).



**Exemplu FSM cu 1 proces**

```
entity fsm_1 is
port ( clk, rst, x, y : in std_logic;
      o1, o2, o3 : out std_logic);
end entity;
```

```
architecture Behavioral of fsm_1 is
type state_type is (s1, s2, s3, s4, s5);
signal state: state_type := s1;
begin
```

```
    process(clk, rst)
    begin
        if rst = '1' then
            state <= s1;
            o1 <= '0'; o2 <= '0'; o3 <= '0';
        elsif rising_edge(clk) then
            case state is
                when s1 => state <= s2;
                           o1<='1'; o2<='0'; o3<='0';
                when s2 => if x = '1' then
                           state <= s3;
                           o1<='1'; o2<='1'; o3<='0';
                           else
                           state <= s4;
                           o1<='0'; o2<='0'; o3<='1';
                           end if;
                when s3 => state <= s4;
                           o1<='0'; o2<='0'; o3<='1';
                when s4 => state <= s5;
                           o1<='1'; o2<='0'; o3<='1';
                when s5 => if y = '1' then
                           state <= s1;
                           o1<='0'; o2<='0'; o3<='0';
                           else
                           state <= s5;
                           o1<='1'; o2<='0'; o3<='1';
                           end if;
            end case;
        end if;
    end process;
```

```
end Behavioral;
```

**Exemplu FSM cu 2 procese**

```

entity fsm_2 is
port (  clk, rst, x, y      : in std_logic;
        o1, o2, o3         : out std_logic);
end entity;

architecture Behavioral of fsm_2 is

type state_type is (s1, s2, s3, s4, s5);
signal state: state_type := s1;

begin

    process1: process(clk, rst)
    begin
        if rst = '1' then
            state <= s1;
        elsif rising_edge(clk) then
            case state is
                when s1 => state <= s2;
                when s2 => if x = '1' then
                            state <= s3;
                        else
                            state <= s4;
                        end if;
                when s3 => state <= s4;
                when s4 => state <= s5;
                when s5 => if y = '1' then
                            state <= s1;
                        else
                            state <= s5;
                        end if;
            end case;
        end if;
    end process;

    process2: process(state)
    begin
        case state is
            when s1 => o1 <= '0'; o2 <= '0'; o3 <= '0';
            when s2 => o1 <= '1'; o2 <= '0'; o3 <= '0';
            when s3 => o1 <= '1'; o2 <= '1'; o3 <= '0';
            when s4 => o1 <= '0'; o2 <= '0'; o3 <= '1';
            when s5 => o1 <= '1'; o2 <= '0'; o3 <= '1';
        end case;
    end process;
end Behavioral;

```

**Exemplu FSM cu 3 procese**

```

entity fsm_3 is
port (  clk, rst, x, y      : in std_logic;
        o1, o2, o3         : out std_logic);
end entity;

architecture Behavioral of fsm_3 is

type state_type is (s1, s2, s3, s4, s5);
signal state: state_type := s1;
signal next_state: state_type;

begin

    process1: process(clk, rst)
    begin
        if reset = '1' then
            state <= s1;
        elsif rising_edge(clk) then
            state <= next_state;
        end if;
    end process;

    process2: process(state, x, y)
    begin
        case state is
            when s1 =>  next_state <= s2;
            when s2 =>  if x = '1' then
                            next_state <= s3;
                        else
                            next_state <= s4;
                        end if;
            when s3 =>  next_state <= s4;
            when s4 =>  next_state <= s5;
            when s5 =>  if y = '1' then
                            next_state <= s1;
                        else
                            next_state <= s5;
                        end if;
        end case;
    end process;

    process3: process(state)
    begin
        case state is
            when s1 => o1 <= '0'; o2 <= '0'; o3 <= '0';
            when s2 => o1 <= '1'; o2 <= '0'; o3 <= '0';
            when s3 => o1 <= '1'; o2 <= '1'; o3 <= '0';
            when s4 => o1 <= '0'; o2 <= '0'; o3 <= '1';
            when s5 => o1 <= '1'; o2 <= '0'; o3 <= '1';
        end case;
    end process;

end Behavioral;

```

## I. Anexa 9 – Tabel cu 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

Figura I-1: ASCII Codes