

# Organizarea Calculatoarelor

## LABORATOR 7 – MIPS complet

### SCOPUL LUCRĂRII

Se va termina proiectarea procesorului MIPS redus, conform Hennessy și Patterson.

### Chestiuni teoretice

Schema completă a procesorului MIPS preluată din Hennessy și Patterson este reluată în figura 1.

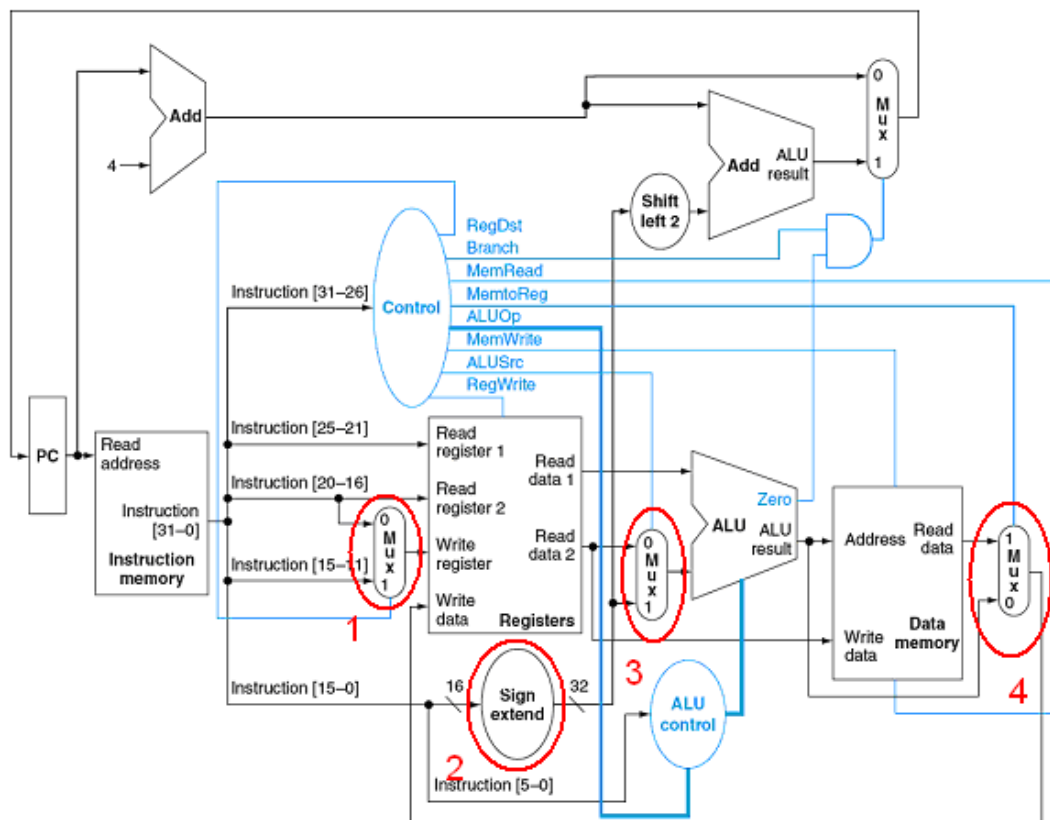


figura 1

Schema se va termina de implementat în acest laborator. Se va continua proiectul **mips** început în laboratorul precedent.

### Atenție:

După amplasarea unui nou modul este foarte posibil să fi ocupat toată lungimea planșei. Pentru a continua desenarea conform indicațiilor ce vor urma este nevoie de o planșă mai mare. Pentru a modifica dimensiunile planșei de desenare faceți **clic dreapta** pe planșă acolo unde nu există nici un obiect. Din meniul contextual care a apărut selectați **Object properties** și în următoarea fereastră schimbați **dimensiunea planșei la D**.

### Pasul 8: Blocul registrelor generale.

Una din structurile centrale din calea de date este fișierul de registre. Un fișier de registre constă dintr-un set de registre care pot fi citite și scrise prin furnizarea numărului registrului de accesat. În capitolele 5 și 6 din Hennessy și Patterson (H&P) a fost utilizat un fișier de registre care are două porturi de citire și un port de scriere. Acest număr de porturi este necesar deoarece o instrucțiunile în

format R, cum ar fi `ADD rd, rs, rt` necesită **simultan** citirea a două registre (rs și rt) plus scrierea unui registru (rd). Fișierul de registre cu două porturi de citire și un port de scriere se desenează ca în figura 2a. Această figură este reprodusă din H&P.

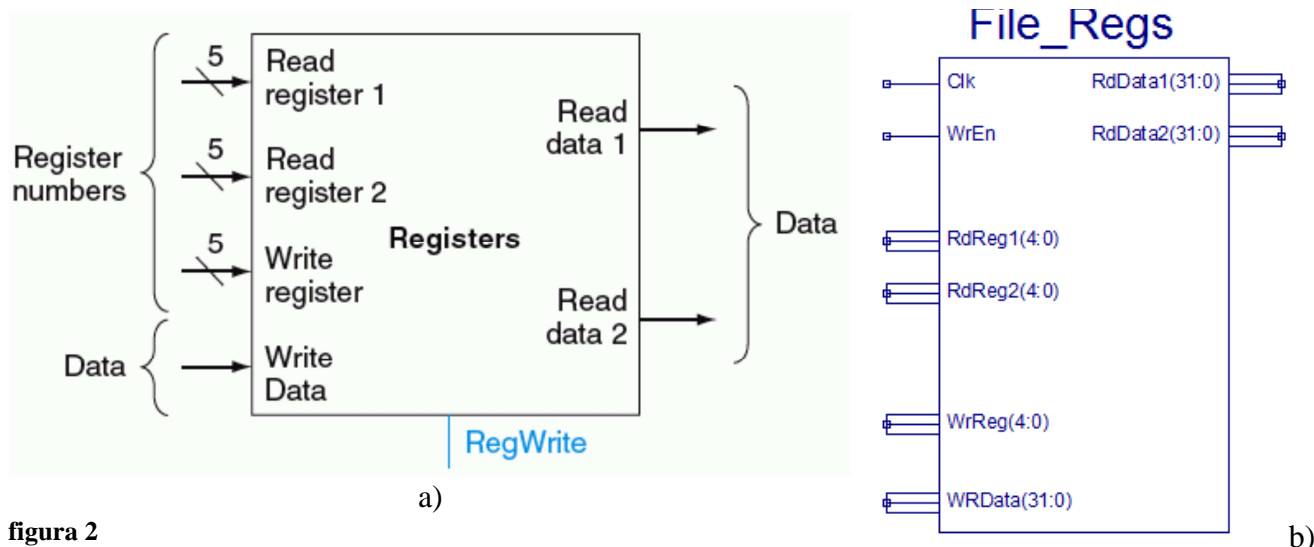


figura 2

În implementarea din acest laborator numele semnalelor din H&P se vor redenumi pentru a le face mai scurte. Redenumirea se face după cum urmează:

Semnificație	Nume în H&P	Nume după redenumire	Dimensiune
Numărul registru de citit numărul 1	Read register 1	RdReg1	5
Conținutul registrului 1	Read data 1	RdData1	32
Numărul registru de citit numărul 2	Read register 2	RdReg2	5
Conținutul registrului 2	Read data 2	RdData2	32
Numărul registrului de scris	Write register	WrReg	5
Data de scris	Write Data	WrData	32
Validare scriere	RegWrite	WrEn	1

După această redenumire fișierul de registre pe care îl vom folosi în continuare va arăta ca în figura 2b.

Este important de reamintit ca bistabilii își schimbă starea numai pe frontul ceasului. Din acest motiv în figura 2b apare și semnalul de ceas Clk.

Pentru descrierea blocului registrelor procedați după cum urmează:

1. Adăugați la proiect un nou fișier sursă de tip VHDL. Acesta se va numi **File\_Regs**.
2. Interfața modulului a fost prezentată în figura 2b. Pentru a genera automat semnalele de la nivelul entității, urmați indicațiile din laboratorul 5, pasul 2, figura 2.
3. Un fișier registre este alcătuit din mai multe registre. În cazul procesorului MIPS acest număr este 32. Fiecare registru are 32 de biți. Dacă se folosește tipul **std\_logic\_vector**, ar fi nevoie de o declarație de semnal pentru fiecare registru. Astfel pentru 32 de registre ar fi necesare 32 de declarații. Pentru a obține un cod mai scurt vom privi cele 32 de registre de 32 de biți fiecare ca un vector de vectori, la fel ca în cazul memoriei ROM.

Declarați tipul de date **tRegs**, la fel ca tipul tROM de la pasul 5, numai că numele tipului de date va fi diferit:

```
type tRegs is array (0 to 31) of std_logic_vector(31 downto 0);
```

4. Declarați semnalul **Regs** de tipul tRegs. Declarația este asemănătoare cu declararea constantei ROM. Spre deosebire de ROM, Regs este semnal și nu constantă. Nefiind constantă, inițializarea

lui Regs este opțională. Vom renunța la inițializare deoarece la majoritatea procesoarelor conținutul registrelor imediat după inițializare este aleatoriu.

```
signal s32Regs32: tRegs;
```

Declarația de tip și declararea semnalului se scriu între **architecture** și **begin**. În continuare vom trata operația de citire. Deoarece citirea unui registru nu modifică nici o stare, trebuie doar să furnizăm la intrare numărul registrului care trebuie citit iar singura ieșire va fi data conținută în acel registru. Porturile de citire pot fi implementate printr-o pereche de multiplexoare, fiecare de lărgimea registrelor fișierului. În figura 3 se prezintă implementarea H&P a celor două porturi de citire pentru un fișier de  $n$  registre de lărgime 32 biți.

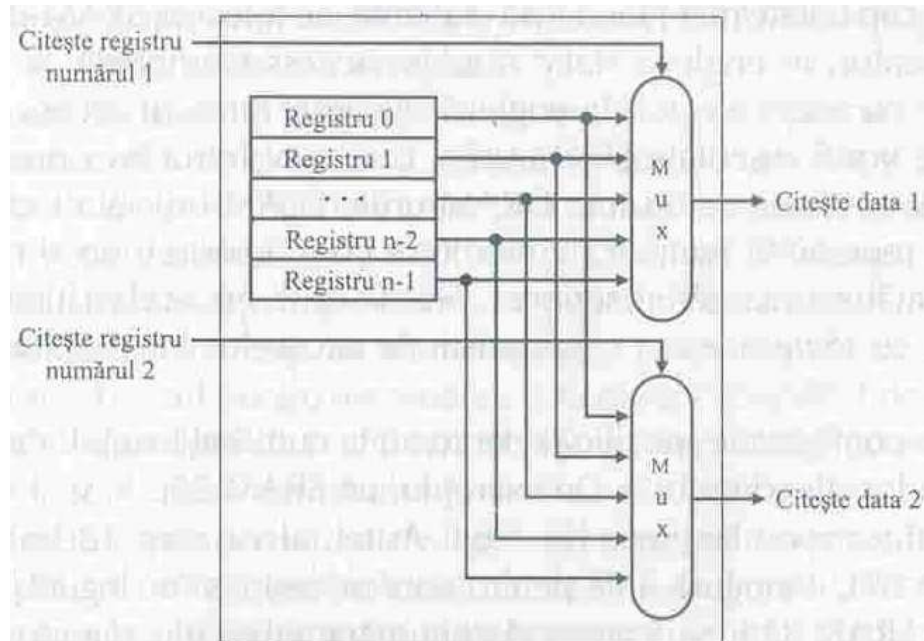


figura 3

Deși descrierea unui multiplexor se poate face în cel puțin 5 moduri, în cazul în care obiectul din care se face selecția este un vector cea mai simplă descriere este o simplă indexare:

```
ieșire_MUX <= vectorul_intrărilor( conv_integer(vector_de_selecție) );
```

Acest mod de descrierea unui multiplexor a fost folosit pentru memoria ROM de la pasul 5 (în laboratorul anterior. Vezi marcajul 3).

5. Declarați semnalele **RdData1\_lcl** și **RdData2\_lcl** de tip `std_logic_vector(31 downto 0)`. Acestea constituie ieșirile „Citește data” a celor două MUX-uri din figura 3. **lcl** înseamnă local.
6. Descrieți cele două MUX-uri conform modelului de mai sus și exemplului din modulul ROM de la pasul 5. Atenție: ieșirile MUX-urilor NU SUNT ieșirile **RdData1** și **RdData2** ci semnale locale (**lcl**). De ce este așa va deveni evident în scurt timp.

În continuare vom trata problema scrierii. Pentru scrierea unui registru sunt necesare trei intrări: numărul registrului, data care trebuie scrisă și semnalul care controlează scrierea în registru. Implementarea portului de scriere este puțin mai complexă, deoarece trebuie modificat numai conținutul registrului specificat. Acest lucru poate fi efectuat prin utilizarea unui decodificator care va genera un semnal care va face scrierea în registru specificat. În figura 4 se prezintă implementarea H&P a portului de scriere al unui fișier de registre:

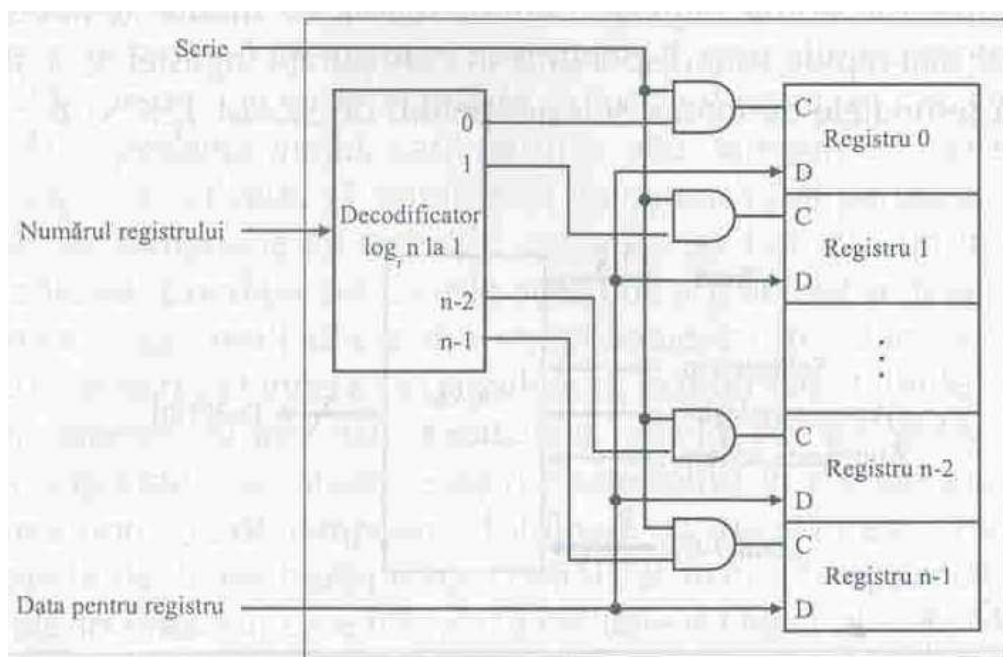


figura 4

Ansamblul DEC–porți AND din figură formează un demultiplexor (DEMUX). Terminalul C al registrelor din figura de mai sus are rolul de „Clock enable” – CE, nu de ceas. În figură, la fel ca în toate figurile din H&P semnalul de ceas nu mai este desenat, pentru a nu supraîncărca schema.

În VHDL demultiplexorul, la fel ca și multiplexorul, se poate implementa în multe moduri. În cazul în care ieșirea DEMUX-ului este un vector, cea mai simplă descriere este o simplă indexare, la fel ca la MUX. Deosebirea constă în faptul că obiectul indexat se află acum în partea stângă a asignării:

```
vector_ieșiri_DEMUX( conv_integer(vector_de_selecție) ) <= intrare_de_demultiplexat;
```

Scrierea registrului și demultiplexorul se pot combina într-o singură declarație VHDL: registrul selectat de vectorul de intrare WrReg se va înscrie cu data de scris WrData pe frontul crescător al semnalului de ceas numai dacă intrarea de validare a scrierii WrEn este '1'. Folosiți „selected signal assignment”:

```
s32Regs32( conv_integer(WrReg) ) <= WRData when rising_edge(Clk) and WrEn = '1' ;
```

**Observație:** acest mod de descriere este nestandard, motiv pentru care pot exista programe de sinteză care să nu îl accepte. Modul standard pentru scrierea unui registru cu Chip enable, acceptat de toate programele de sinteză, se găsește în exemplele din modelele **Language Templates** și se bazează pe process.

#### 7. Descrieți scrierea registrului selectat conform indicațiilor anterioare.

Ultima prelucrare care trebuie efectuată se referă la registrul zero: la MIPS registrul zero este întotdeauna zero. Această funcționalitate se poate implementa în două moduri:

- inițializăm toate registrele la zero și interzicem scrierea registrului zero
- nu inițializăm registrele, permitem scrierea registrului zero dar forțăm la zero dată citită din acest registru.

Deși metoda a) se implementează cu mai puține componente, vom alege metoda 2 deoarece este mai ușor de descris în VHDL de către începători. Implementați forțarea la zero a ieșirii în cazul citirii registrului zero. Ieșirea RdData<sub>i</sub> este identică cu data citită din registru, adică RdData<sub>i</sub>\_lcl,

dacă numărul registrului de citit este diferit zero și zero în caz contrar. Folosiți două declarații selected signal assignment pentru a implementa această funcționalitate.

Verificați sintaxă iar apoi copiați simbolul **File\_Regs.sym** în **directorul proiectului ISE**. Acest simbol se găsește în arhiva zip a lucrării de laborator și vă este furnizat pentru a nu mai pierde timpul cu editarea simbolului. Adăugați pe schema mips fișierul de registre și conectați-l ca în figura următoare:

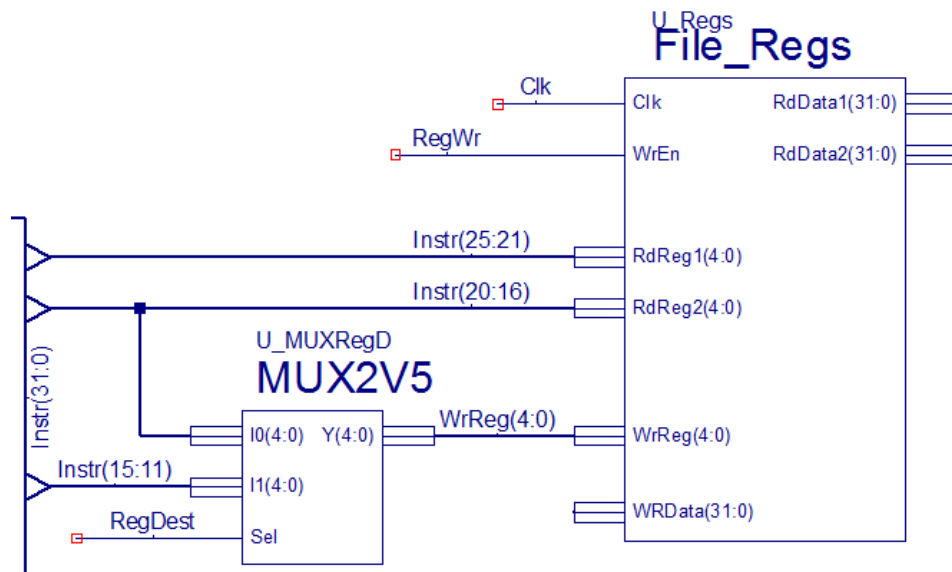


figura 5

Nu uitați să modificați referința fișierului de registre și să o faceți vizibilă.

## Pasul 9: Descrierea ALU plus blocurilor 2 și 3 din figura 1.

Pentru a reduce complexitatea schemei blocul de extensie a semnului și multiplexorul operandului 2 se vor îngloba în blocul ALU.

Adăugați la proiect un nou fișier vhd cu numele **ALU.vhd**. Entitatea ALU plus semnale interne și semnalele interne sunt definite după cum urmează:

```
entity ALU is
  Port (
    RdData1 : in  std_logic_vector (31 downto 0);
    RdData2 : in  std_logic_vector (31 downto 0);
    FAddr   : in  std_logic_vector (15 downto 0);
    ALUSrc  : in  std_logic;
    ALUOP   : in  std_logic_vector (1 downto 0);
    Y       : out std_logic_vector (31 downto 0)
  );
end ALU;

architecture Behavioral of ALU is
  signal SEAddr : std_logic_vector (31 downto 0);
  signal OP1    : std_logic_vector (31 downto 0);
  signal OP2    : std_logic_vector (31 downto 0);

begin
```

Semnificația semnalelor definite la nivelul entității este următoarea:

- **RdData1** reprezintă conținutul registrului **rs** și este furnizat de fișierul de registre.
- **RdData2** reprezintă conținutul registrului **rt** și este furnizat de fișierul de registre.

- **FAddr** (field Address) este câmpul **imm16** din tabelul 1, chestiuni teoretice
- **ALUSrc** este folosit pentru a selecta al doilea operand al ALU. Acest semnal constituie intrarea de selecția multiplexorului 3 din figura 1.
- **ALUOP** este generat de blocul de control (ce va fi proiectat la pasul 12) și selectează tipul de operație ALU după cum urmează:
  - „00” – adunare
  - „01” – scădere
  - „10” – AND
  - „11” – OR
- **Y** este ieșirea ALU

La nivelul arhitecturii se va descrie mai întâi **blocul de extindere a semnului**. În acest sens este necesară cunoașterea unei construcții VHDL numită „**slice**”. O „**slice**” este o felie dintr-un vector, fiind alcătuită din elemente ale vectorului cu indici succesivi. Pentru exemplificare, vom considera următoarea declarație:

```
signal vect      : std_logic_vector(7 downto 0);
```

vect(2 downto 1) alcătuit din vect(2) și vect(1),  
vect(5 downto 3) alcătuit din vect(5), vect(4) și vect(3),  
vect(7 downto 4) alcătuit din vect(7), vect(6), vect(5) și vect(4)

sunt slice-uri ale lui **vect**.

**Restricție:** Gama (range in english) slice-ului este de același tip ca gama vectorul inițial. De exemplu utilizarea slice-ului `vect(2 to 4)` va genera o eroare.

**Regula 1:** Slice-urile nu se declară! Declararea unui vector declară automat toate slice-urile posibile ale acestuia.

**Regula 2:** Oriunde se poate utiliza un vector, se poate utiliza și un slice al acestuia.

De exemplu, dacă  $v_1$ ,  $v_2$  și  $r$  sunt vectori declarați ca:

```
signal v1, v2, r : std_logic_vector(7 downto 0);
```

atunci se poate scrie:

```
r(7 downto 4)      <= v1(3 downto 0) and v2(7 downto 4);
r(3 downto 0)      <= v1(7 downto 4) or  v2(3 downto 0);
```

În expresii logice și aritmetice slice-urile, vectorii și scalarii se pot mixa.

În cazul slice-uri se aplică aceleași reguli ca în cazul expresiilor vectoriale. În cazul în care o expresie logică se folosesc slice-uri, calculul se face **de la stânga la dreapta**, element cu element, indiferent de tipul gamei și de valorile indecșilor stânga și dreapta. Evident toți vectorii și slice-urile trebuie să aibă aceeași dimensiune. Expresiile vectoriale au fost tratate pe scurt în laboratorul 5.

În continuare se prezintă un exemplu de extensia a semnului. Dacă FAddr (Field Addr) are valoarea:

```

1. FAddr:                                     b"0111_0101_0110_1111"  atunci
   SEAddr: b"0000_0000_0000_0000_0111_0101_0110_1111"

2. FAddr:                                     b"1111_0101_0110_1111"  atunci
   SEAddr: b"1111 1111 1111 1111 1111 0101 0110 1111"

```

Se observă că întotdeauna `SEAddr(15 downto 0) = FAddr`. Partea superioară a lui `SEAddr`, și anume `SEAddr(31 downto 16)`, primește valoarea `b"0000_0000_0000_0000"` dacă `FAddr(15)` este ,0' sau valoarea `b"1111_1111_1111_1111"` dacă `FAddr(15)` este ,1'.

În loc de:

`b"0000_0000_0000_0000"` se poate scrie comprimat `x"0000"` iar în loc de

`b"1111_1111_1111_1111"` se poate scrie `x"FFFF"`.

**Descrieți extinderea semnelui conform indicațiilor de mai sus.**

În continuare descrieți MUX-ul 2 marcat cu 3 în figura 1. MUX-ul oferă la ieșire pe OP2, al doilea operand al ALU. **Descrieți acest MUX!**

Ieșirea ALU este semnalul Y. În funcție de ALUOP se face fie adunare, fie scădere, fie OR, fie AND. În C s-ar scrie:

```
switch ALUOP{
    case 00: Y=RdData1 + OP2; break;
    case 01: Y=RdData1 - OP2; break;
    case 10: Y=RdData1 & OP2; break;
    case 11: Y=RdData1 | OP2; break;
}
```

Echivalentul VHDL al lui switch este **selected signal assignment**. Deși s-ar putea folosi **conditional signal assignment**, cea mai eficientă descriere se obține cu selected signal assignment.

Spre deosebire de C, pe toate ramurile din partea stângă a asignărilor se află același obiect. Această declarație este descrisă în continuare:

Sintaxa:

```
with vector_signal select
target_signal    <= expression_1 when val_1,
                  expression_2 when val_2,
                  ...
                  Expression_N when val_N;
```

Sunt necesare următoarele precizări:

- `target_signal` poate fi scalar sau vector.
- `target_signal` primește valoarea `expression_i` când `vector_signal` are valoarea `val_i`.
- O valoare `val_i` nu poate apărea decât o singură dată.
- Toate valorile posibile ale lui `vector_signal` trebuie acoperite. De exemplu, dacă `vector_signal` are 2 elemente, am fi tentați să credem că există numai patru valori posibile: „00”, „01”, „10” și „11”. Deoarece tipul **std\_logic** este un tip enumerat cu 9 valori, în realitate pentru 2 elemente există 81 de posibilități.
- Valorile care nu sunt acoperite explicit pot fi acoperite de valoarea **others**. Această valoare are rolul lui **default** în C și trebuie să apară în ultima poziție, în locul lui `val_N`.

Folosiți **selected signal assignemet** pentru a descrie ALU propriu-zis. Semnalul cu care se face selecția este **ALUOP**. Operandul 1 al ALU, și anume OP1, este identic cu **RdData1** iar OP2 se află la ieșirea MUX-ul 2 marcat cu 3 în figura 1, descris anterior.

**Terminați descrierea blocului ALU, verificați sintaxa și creați simbolul ALU.**



**Continuați** desenarea MIPS conform figurii următoare. Notați toate conexiunile și referințele ca în figura următoare:

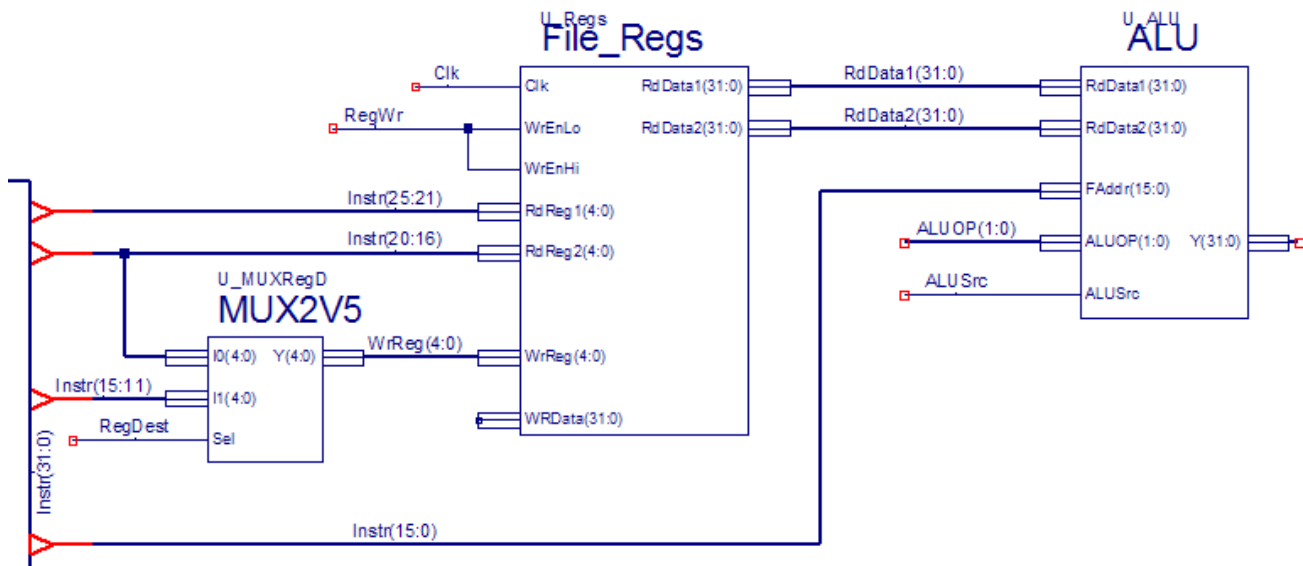


figura 6

### Pasul 10: Crearea multiplexorului data (marcaj 4 în figura 1).

Creați un MUX2 pentru vectori de 32 elemente. Fișierul vhd care conține descrierea multiplexorului se va numi **MUX2V32.vhd** iar entitatea asociată este următoarea:

```
entity MUX2V32 is
    port (
        IO    : in  std_logic_vector(31 downto 0);
        I1    : in  std_logic_vector(31 downto 0);
        Sel    : in  std_logic;
        Y      : out std_logic_vector(31 downto 0)
    );
end MUX2V32;
```

**Nu uitați** să scrieți codul vhd la nivelul arhitecturii. Verificați sintaxa și creați simbolul pentru acest multiplexor.

### Pasul 11: Adăugarea memoriei de date și a multiplexorului data

Blocul memoriei de date conține o memorie RAM alcătuită din 16 locații de 32 de biți plus 3 locații speciale:

- Cele 16 locații de 32 de biți ocupă spațiul de adrese **x"0000\_0000" - x"0000\_003f"**
- Locația de la adresa **x"0000\_0040"** este de tipul „**Read only**”. **Întotdeauna** data citită din această locație are valoarea markerului I/O de intrare **INW0**. Această locație este folosită pentru introducerea de date. Scrierea acestei locații nu are efect: data scrisă se va pierde.
- Locația de la adresa **x"0000\_0044"** este de tipul „**Read only**”. **Întotdeauna** data citită din această locație are valoarea markerului I/O de intrare **INW1**. Această locație este folosită pentru introducerea de date. Scrierea acestei locații nu are efect: data scrisă se va pierde.
- Locația de la adresa **x"0000\_0048"** este de tipul „**Write only**”. Data scrisă în această locație va apare în exteriorul sistemului prin intermediul markerului I/O de ieșire **OUTW0**. Această locație este folosită pentru extragerea de date. **Întotdeauna** data citită din această locație este nedefinită. Data citită din această locație **NU** coincide cu data scrisă anterior.



În continuare execuțați următoarele operații:

1. Fișierele **DataMem.vhd** și **DataMem.sym** se copiază în directorul proiectului ISE. Apoi **adăugați** la proiect memoria de date cu **Add Source** în fereastra **Sources**.
2. **Continuați** desenarea MIPS conform figurii următoare. Dacă este necesar **editați simbolul** multiplexorului data pentru ca acesta să arate **exact** ca în figura 7. **Atenție la ordinea intrărilor** de date. De sus în jos ordinea este I1 și apoi I0.

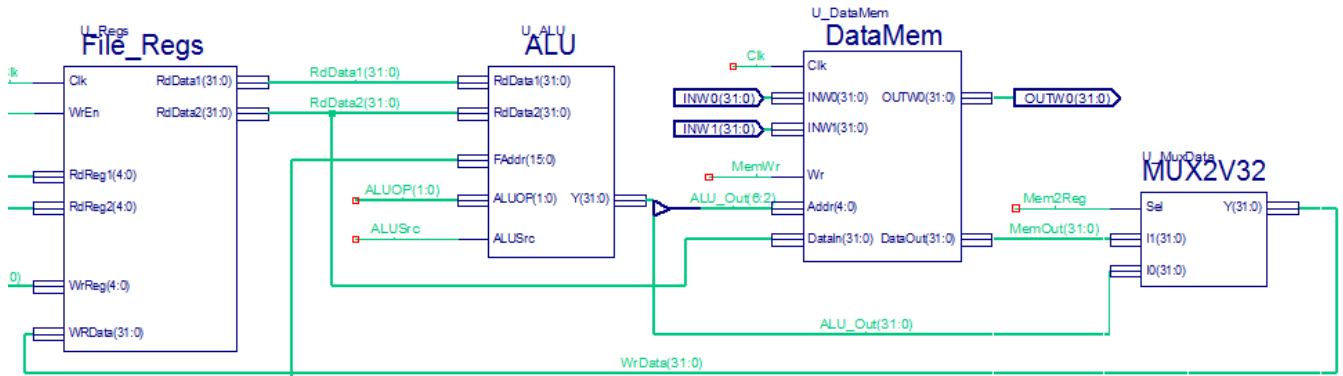


figura 7

3. Notați toate conexiunile și referințele ca în figura 7.

## Pasul 12: Adăugarea blocului de control

Blocul de control este figurat cu albastru în figura 1. Fișierul **vhd** care conține descrierea blocului de control se va numi **ctrl.vhd** iar entitatea asociată este următoarea:

```
entity ctrl is
    port (
        OP      : in  std_logic_vector(5 downto 0);
        Funct   : in  std_logic_vector(5 downto 0);
        ALUSrc  : out std_logic;
        ALUOP   : out std_logic_vector(1 downto 0);
        MemWr   : out std_logic;
        Mem2Reg : out std_logic;
        RegWr   : out std_logic;
        RegDest : out std_logic
    );
end ctrl;
```

Adăugați la proiect acest fișier.

Pentru început se vor implementa numai instrucțiunile din tabelul următor. Codificarea acestor instrucțiuni este extrasă din ISA (Instruction Set Architecture) MIPS și copiată în tabel:

	6	5	5	5	5	6
	<b>Opcode</b>					<b>Function</b>
<b>ADD</b>	0	Rs	Rt	Rd	0	10_0000
<b>SUB</b>	0	Rs	Rt	Rd	0	10_0010
<b>AND</b>	0	Rs	Rt	Rd	0	10_0100
<b>OR</b>	0	Rs	Rt	Rd	0	10_0101
<b>LW</b>	10_0011	Rs	Rt	Imm16		
<b>SW</b>	10_1011	Rs	Rt	Imm16		

Pe baza câmpurilor **Opcode** și **Function** se specifică semnalele de control conform tabelului următor:

	Opcode	Function	ALUSrc	ALUOP	MemWr	Mem2Reg	RegWr	RegDest
<b>ADD</b>	00_0000	10_0000	0	00	0	0	1	1
<b>SUB</b>	00_0000	10_0010	0	01	0	0	1	1
<b>AND</b>	00_0000	10_0100	0	10	0	0	1	1
<b>OR</b>	00_0000	10_0101	0	11	0	0	1	1
<b>LW</b>	10_0011	-	1	00	0	1	1	0
<b>SW</b>	10_1011	-	1	00	1	0	0	0

Rolul semnalelor de control este explicat pe larg în H&P. Pe scurt, rolul acestor semnale și valorile lor logice în funcție de tipul instrucțiunii este:

- **RegDest** – registru destinație. Acest semnal controlează MUX-ul rt-rd marcat cu 1 în figura 1 și implementat la pasul 7. Pentru instrucțiunile în format R (scrise cu roșu în prima coloană a tabelului de mai sus) registru destinație este *rd* iar pentru instrucțiunile în format I (scrise cu negru în prima coloană a tabelului de mai sus) este *rt*. Pentru a selecta *rd* pentru instrucțiunile R semnalul RegDest trebuie să fie ,1' iar pentru a selecta *rt* pentru instrucțiunile I trebuie să fie ,0'.
- **RegWr** – scrie registru. Toate instrucțiune scriu rezultatul într-un registru cu excepția SW care scrie rezultatul în memorie.
- **MemWr** – scrie memorie. Singura instrucțiune care scrie rezultatul în memorie este SW.
- **Mem2Reg** – memorie la registru. Datele care se scriu în registrul destinație provin de la ALU sau de memoria de date. Singura instrucțiune la care datele provin de la memorie este LW.
- **ALUOP** – operația executată de ALU. La pasul 9 – Descrierea ALU – s-a definit semnalul ALUOP care precizează operația executată de ALU: „00” – adunare, „01” – scădere, „10” – AND și „11” – OR. Pentru instrucțiunile ADD, SUB, AND și OR este evident ce valoare va avea ALUOP. În cazul instrucțiunilor LW și SW ALU este folosit pentru a calcula adresa locației de memorie. Această adresă se calculează prin adunarea conținutului registrului *rs* cu câmpul *imm* al instrucțiunii extins ca semn. În concluzie pentru LW și SW ALUOP trebuie să fie „00” pentru ca ALU să execute adunare.
- **ALUSrc** – Sursa ALU. Pentru instrucțiunile în format R al doilea operand ALU este conținutul registrul *rt*, adică RdData2, iar pentru instrucțiunile în format I este câmpul **imm16** al instrucțiunii, extins ca semn.

Tabelul specifică 7 funcții ce depind fie de 6, fie de 12 variabile. O funcție logică specificată prin intermediul tabeli sale de adevăr se poate implementa cu **selected signal assignment**, ca în laboratorul 5, la implementarea depășirii. De exemplu, pentru MemWr se poate scrie:

```
MemWr      <= ,1' when OPCODE = b"10_1011" else ,0';
```

Deoarece avem de implementat 7 funcții, trebuie scrise 7 **conditional signal assignment**. Scrieți aceste **asignări**.

**Verificați sintaxa și creați simbolul pentru blocul de control. Plasați** simbolul blocului de control sub multiplexorul rt-rd. **Continuați** desenarea MIPS conform figurii următoare. Notați toate conexiunile și referințele ca în figura următoare:

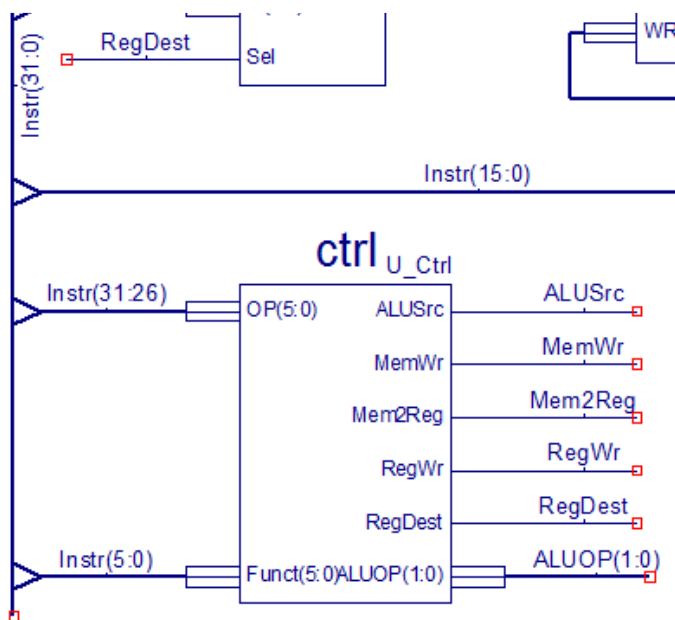


figura 8

**Conectarea semnalelor de control se face prin nume:** două fire (sau mai multe) cu același nume se consideră automat conectate. **Nu conectați prin fire continue** semnalele de control deoarece ar supraîncărca schema!

1. **Verificați** schema. Treceți mai departe dacă nu sunt warning-uri și/sau erori. În caz contrar încercați să le eliminați. Dacă nu reușiți, chemați profesorul.
2. **ATENȚIE:** Înainte de implementare trebuie schimbată o opțiune implicită a procesului de sinteză. Pentru aceasta, în fereastra **Processes** faceți clic dreapta pe **Synthesize** și în meniul contextual ce va apare selectați **Properties....** Va apare fereastra din figura următoare:

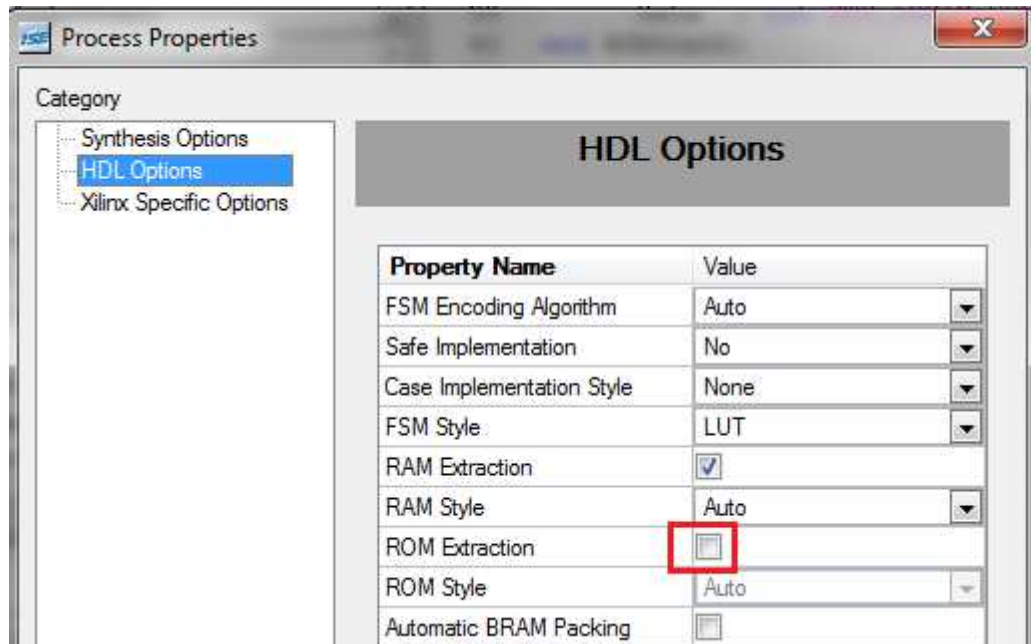


figura 9

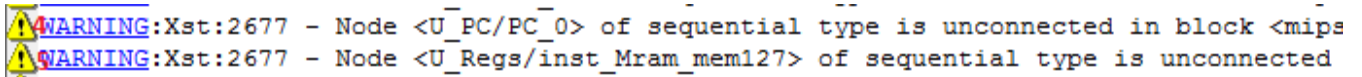
În categoria **HDL Option** debifați căsuța de validare **ROM Extraction**, ca în figura 9.

3. Faceți **sinteza** și apoi parcurgeți raportul de sinteză. Se acceptă următoarele categorii de warning-uri:

```

WARNING:HDLParasers:3607 - Unit work/File_Regs is now defined in a different file
WARNING:HDLParasers:3607 - Unit work/File_Regs/File_Regs_Arch is now defined in a
WARNING:Xst:2734 - Property "use_dsp48" is not applicable for this technology.
  
```

Warning-urile de mai sus pot apărea pentru mai multe module. Se ignoră.



WARNING:Xst:2677 - Node <U\_PC/PC\_0> of sequential type is unconnected in block <mips  
WARNING:Xst:2677 - Node <U\_Regs/inst\_Mram\_mem127> of sequential type is unconnected

Warning-ul 2677 va apare pentru mai multe valori ale PC și pentru diferite valori ale lui *inst\_Mram\_mem* sau poate să nu apară de loc. Dacă apare se ignoră.

Dacă numărul de erori este **zero** iar warning-uri sunt numai din categoriile discutate mai sus, treceți la implementare. În caz contrar încercați să eliminați erorile și warning-urile. Dacă nu reușiți, chemați profesorul.

4. Faceți **implementarea**. Trebuie să nu aveți erori dar warning-urile sunt permise. Dacă aveți erori încercați să le eliminați. Dacă nu reușiți, chemați profesorul.

**Dacă ați ajuns aici, chemați profesorul pentru validare!**