Indicaţii de implementare

Şedinţa de proiect 1

1. Se va crea proiectul ISE cu numele PIC24.
2. Se va adăuga la proiect un fişier de tip schematic cu numele PIC24. Paşii 1 şi 2 sunt asemănători cu paşii Pasul 1 din laboratorul 6 SOC.
3. Copiaţi în folderul proiectului PIC24 fişierele ProgCnt.vhd, ProgCnt.sym, PC_Update.vhd şi PC_Update.sym din proiectul mips.
4. Adăugaţi la proiectul PIC24 fişierele ProgCnt.vhd şi PC_Update.vhd. Automat se vor adăuga la proiect şi simbolurile corespunzătoare, astfel încât acestea nu vor mai trebui editate.
5. Modificaţi cele două fişiere de la pasul 4 pentru PIC24. Lăţimea PC va fi conform cerinţei din Proiectul cadru. **Atenţie**: PC nu se mai incrementează cu 4! Verificaţi sintaxa iar apoi regeneraţi simbolurile pentru PC şi PC_update.
6. Plasaţi aceste două simboluri pe planşa de desenare PIC24.sch şi conectaţi-le. Inspiraţi-vă din Pasul 4 din laboratorul 6 SOC.
7. Copiaţi în folderul proiectului PIC24 fişierele ROM32x32.vhd şi ROM32x32.sym din proiectul mips. Deoarece lăţimea instrucţiunii la PIC24 este 24 de biţi, schimbaţi numele acestor fişiere la ROM32x24.vhd şi ROM32x42.sym.
8. Editaţi fişierul ROM32x32.vhd şi schimbaţi şirul de caractere ROM32x32 în ROM32x24 ori de câte ori apare .
9. Regeneraţi simbolul în ROM32x24. Plasaţi pe planşa de desenare modulul ROM şi conectaţi-l cu PC. Inspiraţi-vă din Pasul 5 din laboratorul 6 SOC.
10. Simulaţi! Inspiraţi-vă din Pasul 7 din laboratorul 6 SOC.

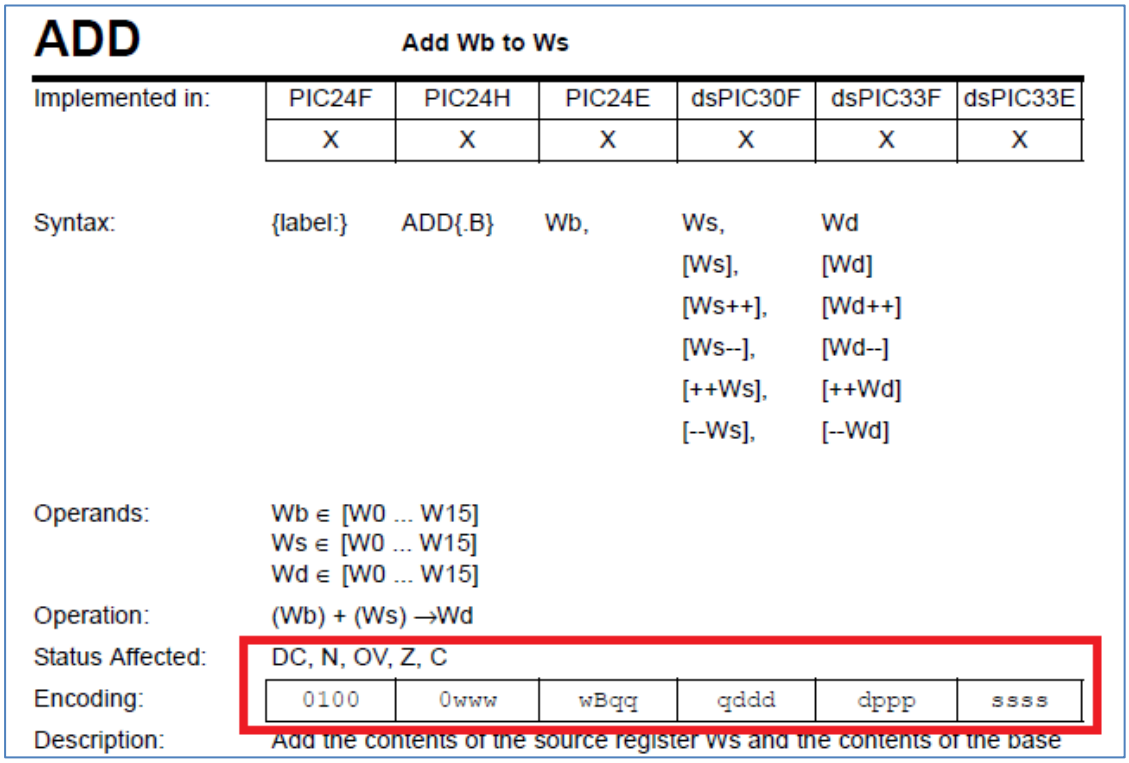
Şedinţa de proiect 2

1. Se va crea tabelul opcodurilor pentru instrucţiunile pe cate la aveți de implementat. În această tabelă se trec atât instrucţiunile comune cât şi instrucţiunile specifice:

tabelul 1

Encoding	2222 3210	1111 9876	1111 5432	11 1098	7654	3210	Flags
ADD	0100	0www	wBqq	qddd	dppp	ssss	N, OV, Z, C
SUB	0101	0www	wBqq	qddd	dppp	ssss	N, OV, Z, C
AND	0110	0www	wBqq	qddd	dppp	ssss	N, - , Z, -
IOR	…	…	…	…	…	…	N, - , Z, -
MOV f, wnd	1000	0fff	ffff	ffff	ffff	dddd	none
MOV wns, f	…	…	…	…	…	…	none
BRA expr	…	…	…	…	…	…	none
…..	…	…	…	…	…	…	…

În tabelă se copiază din ISA MIPS codificarea instrucţiunilor. De exemplu, pentru instrucţiunea ADD codificarea din ISA este prezentată în figura următoare:



ADD Add Wb to Ws

Implemented in:	PIC24F	PIC24H	PIC24E	dsPIC30F	dsPIC33F	dsPIC33E
	X	X	X	X	X	X

Syntax:	{label:}	ADD{.B}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]
				[++Ws],	[++Wd]
				[--Ws],	[--Wd]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Wb) + (Ws) →Wd

Status Affected: DC, N, OV, Z, C

Encoding:	0100	0www	wBqq	qddd	dppp	ssss

Description: Add the contents of the source register Ws and the contents of the base

Codificarea din dreptunghiul roşu şi flag-urile afectate se trec în tabelul 1 pe linia ADD. La fel se procedează pentru instrucţiunile SUB, AND şi MOV f, wnd.

2. Completaţi tabelul pentru restul de instrucţiuni comune şi specifice.
3. Analizând tabelul 1 se constată că numărul registrului destinaţie Wd se află fie pe biţii 11:7 pentru instrucţiunile ADD, SUB şi AND, fie pe biţii 3:0 pentru instrucţiunea MOV f, wnd. De aici rezultă necesitatea unui multiplexor. Acest multiplexor este asemănător cu MUX-ul rt-rd de la MIPS.

   După completarea tabelului este posibil să rezulte că multiplexorul pentru Wd are mai mult de două intrări. De asemenea este posibil să apară şi alte multiplexoare.

   Descrieți aceste multiplexoare, plasaţi-le pe planşa PIC24 şi conectaţi-le.
4. Următorul bloc care se va modifica este File_Regs. Blocul regiştrilor la PIC24 este mai simplu decât cel de la MIPS. Dacă la MIPS registrul zero este întotdeauna zero, la PIC24 nu există această condiţie. Alte deosebiri între PIC24 şi MIPS se referă la numărul regișorilor şi la lățimea acestora.

   Adaptaţi File_Regs, plasaţi-l pe planşa de desenare PIC24.sch şi conectaţi-l. Inspiraţi-vă din Pasul 8 din laboratorul 7 SOC.

Ședința de proiect 3

Se vor implementa blocurile ALU, DataMem şi control. Se vor conecta toate blocurile implementate până acum.

Mai întâi se vor trece în revistă, pe scurt, trei modurile de adresare ale memoriei de date. Modul de adresare influențează proiectarea ALU şi conexiunile ALU – DataMem.

1. Modul fundamental de adresare a memoriei este modul **indirect** via registru. În cazul acestui mod de adresare adresa locaţiei de memorie se află într-unul din registrele generale. ALU nu este implicat în calcularea adresei.

2. Un alt mod de adresare este modul de adresare **direct**. În cazul acestui mod de adresare adresa locaţiei de memorie se află în interiorul instrucţiunii. Nici în acest caz ALU nu este implicat în calcularea adresei.

3. Ultimul mod discutat este **adresarea bazată**. Adresarea bazată este o combinaţie între adresarea indirectă şi cea directă. Acesta este modul de adresare folosit de MIPS. În cazul MIPS adresa se calculează prin **adunarea** unui registru general cu ofsetul specificat în instrucţiune, extins ca semn.

   Deoarece calculul adresei necesită operaţia de adunare, implementarea hardware necesită un sumator. Dar nu are sens să adăugam un sumator pentru calculul adresei iar sumatorul din ALU să nu fie utilizat. Este mai rentabil să folosim sumatorul din ALU plus un multiplexor pentru unul din operanzi. Aceasta este soluţia adoptată la MIPS.

În cazul implementării PIC24 din proiect instrucţiunile MOV cu care se accesează memoria folosesc numai modul de adresare direct. Aşa cum s-a arătat mai sus, pentru acest mod **ALU nu este folosit la calcularea adresei**.

Observaţia anterioară simplifică proiectarea ALU şi modifică conectare memoriei. În ALU de la PIC24 ne se face extinderea semnului, nu mai există multiplexorul pentru operandul 2 şi nici selecţia acestuia (ALUSrc).

Adaptaţi blocul ALU conform celor precizate mai sus, plasaţi-l pe planșa de desenare PIC24.sch şi conectaţi-l.

Spre deosebire de MIPS, procesorul PIC24 este prevăzut cu indicatori de condiţie. Indicatorii de condiţie se implementează în ALU. Un indicator de condiţie este un bit care oferă o informaţie de tip Da-Nu despre o anumită proprietate a rezultatului operaţiei efectuate de ALU. De exemplu indicatorul N spune dacă rezultatul operaţiei, considerat număr cu semn, este negativ sau nu. Indicatorul Z specifică dacă rezultatul este zero sau nu, C specifică transportul iar OV depăşirea formatului de reprezentare (în cazul în care operanzii sunt numere cu semn).

În tabelul 2 s-a centralizat modul în care instrucţiunile ce trebuie implementate afectează indicatorii de condiţie:

tabelul 2

Encoding	CE_NF (CE Flag N)	Flags
ADD	1	N, OV, Z, C
SUB	1	N, OV, Z, C
AND	1	N, - , Z, -
IOR	1	N, - , Z, -
MOV f, wnd	0	none
MOV wns, f	0	none
BRA expr	0	none
…	…	…

Analizând tabelul se observă că anumite instrucţiuni nu afectează nici un indicator de condiţie iar alte instrucţiuni afectează numai anumiți indicatori. De exemplu, instrucţiunile logice nu afectează transportul (pentru că nu ar avea sens) iar MOV-urile şi salturile nu afectează nici un indicator. De aici rezultă că un indicator trebuie să-şi păstreze valoarea dacă instrucţiunea curentă nu modifică respectivul indicator. Dacă ceva trebuie să-şi păstreze valoarea, acel ceva se implementează cu un bistabil.

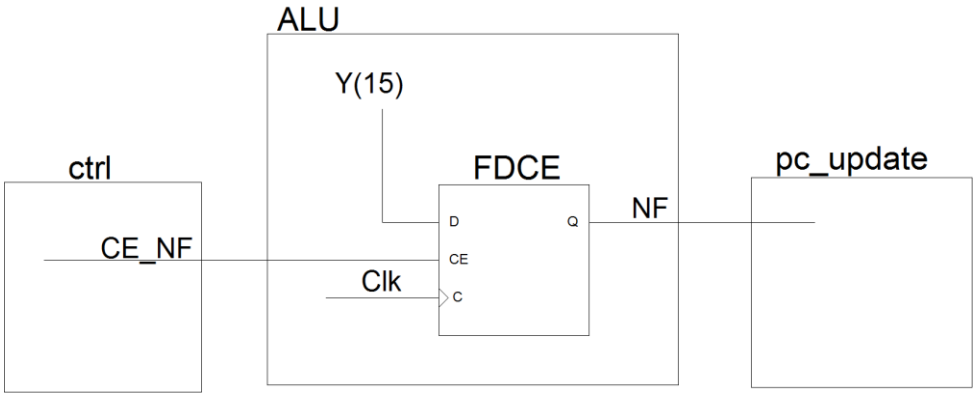
Conform observaţiei de mai sus, unele instrucţiuni nu afectează unii indicatorii de condiţie. Cum implementarea curentă este de tip o instrucţiune – un impuls Clk, avem nevoie de un bistabil care să poată

ignora anumite impulsuri Clk (şi implicit anumite instrucţiuni). Un astfel de bistabil este prevăzut cu încă o intrare numită CE – Clock Enable: dacă frontul activ al lui Clk apare când CE='0' atunci starea bistabilului nu se va schimba.

În VHDL un astfel de bistabil se descrie astfel:

```
Q <= D when rising_edge(Clk) and CE='1';
```

Implementarea de principiu a indicatorului N este prezentată în figura următoare:



CE pentru un indicatorul de condiție N trebuie generat de blocul de control conform coloanei CE_NF din tabelul 2. Informația care se va memora este bitul de semn al rezultatului, adică bitul 15.

Atenţie! Bitul 15 al rezultatului nu se poate folosi aşa cum îl generează ALU pentru a implementa NF. Vom considera un fragment din implementarea ALU:

```
entity ALU is
   port(
     ……
     Y    : out std_logic_vector(15 downto 0);
     NF   : out std_logic_vector
   )
end;

architecture ALU_arch of ALU is
begin
   Y    <= ……
   NF   <= Y(15) when rising_edge(CLK)……    ← EROARE!
end;
```

Implementarea de mai sus va genera o eroare deoarece semnalul Y a fost declarat de tip OUT. Semnalele de tip OUT nu pot fi utilizate în expresii.

Semnalul Y trebuie generat în exterior dar în acelaşi timp avem nevoie de el pentru operaţii interne. Un astfel de semnal ar trebui declarat în modul buffer. Deoarece Modelsim nu se descurcă bine cu buffer vom folosi metoda copiei, după cum urmează:

```
entity ALU is
   port(
     ……
     Y    : out std_logic_vector(15 downto 0);
     NF   : out std_logic_vector
   )
end;
```

```
architecture ALU_arch of ALU is
   signal sY:   std_logic_vector(15 downto 0); --Y va fi o copie a lui sY
begin
   Y   <= sY;
   sY  <= ……..
   NF  <= sY(15) when rising_edge(CLK)……   ← OK!
end;
```

Pentru indicatorul Z se va adăuga un comparator între Y şi zero si se va memora ieşirea comparatorului. **Pentru OV** se va adăuga un SLC care se va implementa conform indicaţiilor din laboratorul 2 SOC, pasul 4 şi se va memora ieşirea acestui SLC.

Indicatorul C este cel mai complicat. Pentru C vom considera mai întâi adunarea. O primă soluție este soluţia folosită în laboratorul 3. Din păcate această soluție este greu de implementat în VHDL. Chiar dacă am implementa-o, codul rezultat ar fi stufos şi greu de înţeles.

Pentru adunare soluția cea mai simpla în VHDL se bazează pe supraîncărcarea operatorului ‚+' şi conduce la un cod mult mai clar. Soluţia bazată pe supraîncărcare s-a folosit în laboratorul 5. Din păcate această soluție nu implementează transportul de ieşire de la rangul cel mai semnificativ.

Adunarea plus generarea transportului se poate implementa cu operatorul ‚+' dacă adăugăm la stânga fiecare operand un bit de ‚0'.

Pentru a înţelege mai uşor ideea vom considera un sumator pe patru biţi. Să presupunem ca operanzii sunt A=1100 şi B=0101. Adunarea A+B va produce rezultatul:

```
1100+
0101
0001
```

După cum era de aşteptat, transportul se pierde. Să vedem ce se întâmpla dacă adăugam un bit ‚0' ca cel mai semnificativ bit. Operandul A se devine 01100 iar B devine 00101. Rezultatul adunării noilor operanzi este:

```
01100+
00101
10001
```

Rezultatul va avea la rândul său un bit suplimentar în poziția cea mai semnificativă. Acest bit are valoarea transportului de ieşire de la rangul cel mai semnificativ.

Adăugarea unui bit suplimentar la un vector se face cu ajutorul operatorului de concatenare. Simbolul pentru acest operator este ‚&'. Concatenarea presupune doi operanzi; oricare din operanzi este fie vector (std_logic_vector), fie scalar (std_logic). Rezultatul este întotdeauna un vector.

În continuare vom considera un modul VHDL care face adunare pe 4 biţi. Codul pentru modulul sumator fără generarea transportului este:

```
entity sum is
   port(
     A   : in  std_logic_vector(3 downto 0);
     B   : in  std_logic_vector(3 downto 0);
     R   : out std_logic_vector(3 downto 0)
   )
end;

architecture sum_arch of sum is
begin
   R <= A+B;
end;
```

Implementarea transportului modifică codul după cum după cum urmează:

```
entity sumcy is
   port(
     A   : in  std_logic_vector(3 downto 0);
     B   : in  std_logic_vector(3 downto 0);
     R   : out std_logic_vector(3 downto 0);
     Co  : out std_logic
   )
end;

architecture sumcy_arch of sumcy is
   signal CYR  : std_logic_vector(4 downto 0);
begin
   CYR <= (,0' & A) + (,0' & B);
   R   <= CYR(3 downto 0);
   Co  <= R(4);
end;
```

Operanzii sumatorului sunt ‚0'&A şi ‚0'&B. Deoarece lungimea operanzii este de 5 biți şi rezultatul va avea tot 5 biţi. Bitul 4 este transportul iar biţii 3:0 sunt rezultatul adunării.

Atenţie! Operatorul de concatenare are aceeaşi prioritate ca adunarea şi scăderea şi din acest motiv parantezele sunt obligatorii.

Situaţia este diferită în cazul scăderii. Se ştie că de fapt scăderea se implementează prin adunarea complementului. Operatorul ‚-' a fost supraîncărcat pentru a face codul mai clar dar a ascuns modul real în care se face scăderea. La fel ca la supraîncărcarea lui ‚+', nu se generează împrumutul.

În cazul scăderii NU există o intrarea separată pentru împrumut, ci se foloseşte transportul. Semnificaţia transportului este diferită: împrumutul are valoarea transportului negat. Pentru a înţelege mai uşor ideea vom considera scăderea cu operanzi pe patru biţi. Să presupunem ca operanzii sunt A=5=0101 şi B=2=0010. Scăderea A-B se va executa de fapt ca A+not B +1 şi va produce rezultatul:

```
A-B:          0101+
              1101        Not(0010)=1101
                 1
             ------
             10011
```

Se observă că această scădere generează transport dar împrumutul este ‚0'.

Să efectuăm scăderea B-A ca B+not A +1:

```
B-A:          0010+
              1010        Not(0101)=1010
                 1
             ------
             01101
```

Această scădere NU generează transport dar împrumutul trebuie să fie ‚1'.

În concluzie, la scădere transportul negat are semnificaţia de împrumut.

Ultima întrebare la care trebuie răspuns priveşte valoarea care trebuie adăugată în faţă operanților pentru generarea corectă a transportului. Considerăm următoarea implementare:

```
entity difcy is
   port(
     A    : in  std_logic_vector(3 downto 0);
     B    : in  std_logic_vector(3 downto 0);
     R    : out std_logic_vector(3 downto 0);
     Co   : out std_logic
   )
end;

architecture difcy _arch of difcy is
   signal CYR  : std_logic_vector(4 downto 0);
begin
   CYR <= (X & A) + (Y & B);
   R   <= CYR(3 downto 0);
   C0  <= R(4);
end;
```

Găsiţi valoarea constantelor X şi Y pentru ca rezultatul să fie corect. Nu uitaţi că scăderea se face prin adunarea A+notB+1. Consideraţi exemplele anterioare.

Memoria de date

Memoria de date la PIC24E are organizarea 32Kx16. 16 biţi formează un cuvânt. Memoria poate fi adresată atât pe cuvânt cât şi pe octet. Din acest motiv în asamblor se specifică adresa de octet atât pentru octeţi cât şi pentru cuvinte. Corespondenţa între adresa de cuvânt şi adresa de octet este detaliată în următorul tabel:

Adresă Cuvânt	Adresa Octet Superior (biţii 15..8)	Adresa Octet Inferior (biţii 7..0)
0000h	0001h	0000h
0001h	0003h	0002h
0002h	0005h	0004h
...	...	...
7ffeh	fffdh	fffch
7fffh	ffffh	fffeh

În asamblor adresa unui cuvânt este adresa octetului sau inferior. Pentru exemplificare vom considera instrucţiunea **MOV f,Wnd**:

Syntax:	{label:}	MOV	f,	Wnd		
Operands:	f ∈ [0 ... 65534] Wnd ∈ [W0 ... W15]					
Operation:	(f) →Wnd					
Status Affected:	None					
Encoding:	1000	0fff	ffff	ffff	ffff	dddd
Description:	Move the word contents of the specified file register to Wnd. The file					

Pentru instrucţiunea MOV 0x1020,W1 asamblorul generează codul maşină următor:

```
000204  808101      MOV 0x1020, W1
```

Se observă că în instrucţiune adresa este 1020h iar in codul maşină generat adresa este 1020h/2=810h.

Blocul de memorie pentru PIC24 este foarte asemănător cu blocul de memorie de la MIPS.

Blocul memoriei de date conţine o memorie RAM alcătuită din 16 locaţii de 16 de biţi plus 3 locaţii speciale. Acest bloc foloseşte 5 adrese. Adresele A4-A0 generate de procesor (dreptunghiul albastru din figura de mai sus) se conectează pinii Addr(4:0) ai blocului de memorie. Celelalte adrese procesor nu contează. Din acest motiv blocul memoriei de date ocupă multiple spaţii de adresă procesor.

În asamblare se vor folosi adresele procesor specificate mai jos:

- Cele 16 locaţii de 16 biţi ocupă spaţiul de adrese de octet **1000h -101Fh**
- Cuvântul de la adresa de octet **1020h** este de tipul **„Read only”**. **Întotdeauna** data citită din această locaţie are valoarea marcherului I/O de intrare **INW0**. Această locaţie este folosită pentru introducerea de date. Scrierea acestei locaţii nu are efect: data scrisă se va pierde.
- Locaţia de la adresa **1022h** este de tipul **„Read only”**. **Întotdeauna** data citită din această locaţie are valoarea marcherului I/O de intrare **INW1**. Această locaţie este folosită pentru introducerea de date. Scrierea acestei locaţii nu are efect: data scrisă se va pierde.
- Locaţia de la adresa **1024h** este de tipul **„Write only”**. Data scrisă în această locaţie va apare în exteriorul sistemului prin intermediul marcherului I/O de ieşire **OUTW0**. Această locaţie este folosită pentru extragerea de date. **Întotdeauna** data citită din această locaţie este nedefinită. Data citită din această locaţie **NU** coincide cu data scrisă anterior.

Blocul memoriei de date se conectează cu o mică diferenţă faţă de MIPS. La MIPS adresa este generată de ALU deoarece adresarea este bazată pe când la PIC24 (pentru proiect) modul de adresare este direct, aşa că adresa se ia din instrucţiune

Plasaţi Memoria de date pe planșa de desenare şi conectaţi-o.

Blocul de control

Se implementează după indicaţiile de la MIPS. O primă diferenţă se referă la opcode: la MIPS codul instrucţiunii se află în câmpul OPCODE pentru instrucţiunile în format I sau în câmpurile OPCODE şi FUNCTION pentru instrucţiunile în format R.

La PIC24, pentru majoritatea instrucţiunilor de implementat, opcodul se află pe primii 5 biți ai instrucţiunii. Construiţi tabelul următor şi verificaţi că primii 5 biți ai instrucţiunii sunt diferiţi pentru instrucţiunile de implementat. Dacă 5 biți nu sunt suficienţi, consideraţi mai mulți biţi!

	Opcode	CE_NF	ALUOP	MemWr	Mem2Reg	RegWr	Alte semnale de control…
ADD	01000	1					
SUB	01010	1					
AND	01100	1					
IOR		1					
MOV f, wnd	10000	0					
MOV wns, f		0					
BRA expr		0					
alte instrucţiuni		?					

În tabel se trec toate semnalele de control ce au rezultat din implementarea căii de date. Unele semnale sunt identice ca nume şi funcţionalitate cu cele de la MIPS, ca de exemplu: ALUOP, MemWR, Mem2Reg şi RegWr. Apar semnale de control noi cum ar fi CE_flag iar alte semnale dispar, cum ar fi ALUSrc.

Scrieţi codul VHDL conform indicaţiilor, generaţi simbolul, plasaţi-l pe planşă şi conectaţi-l.

Şedința de proiect 4

Se va detalia procedura de obținere a codului maşină pentru PIC24.

Pentru obținere a codului maşină, urmați paşii de mai jos:

1. Instalaţi mediul de dezvoltare MPLAB® X IDE. Pentru a obține acest mediu accesaţi pagina http://www.microchip.com/mplabx-ide-windows-installer
2. Instalaţi compilatorul MPLAB® XC16 Compiler. Pentru a obține compilatorul accesaţi pagina http://www.microchip.com/mplabxc16windows
3. Copiaţi pe calculatorul dumneavoastră folderul **test1.X** în care se găseşte proiectul.
4. Lansați în execuţie **Mplab** prin intermediul icoanei: 

5. Deschideți proiectul test1 selectând **File →Open Project…** Navigaţi până la folderul **test1.X** (pe care l-aţi copiat la pasul 3), selectaţi-l şi apoi apăsaţi **Open Project**
6. Deschideți fişierul test1.s făcând dublu click pe numele său. Trebuie să obţineţi situaţia din figura următoare:

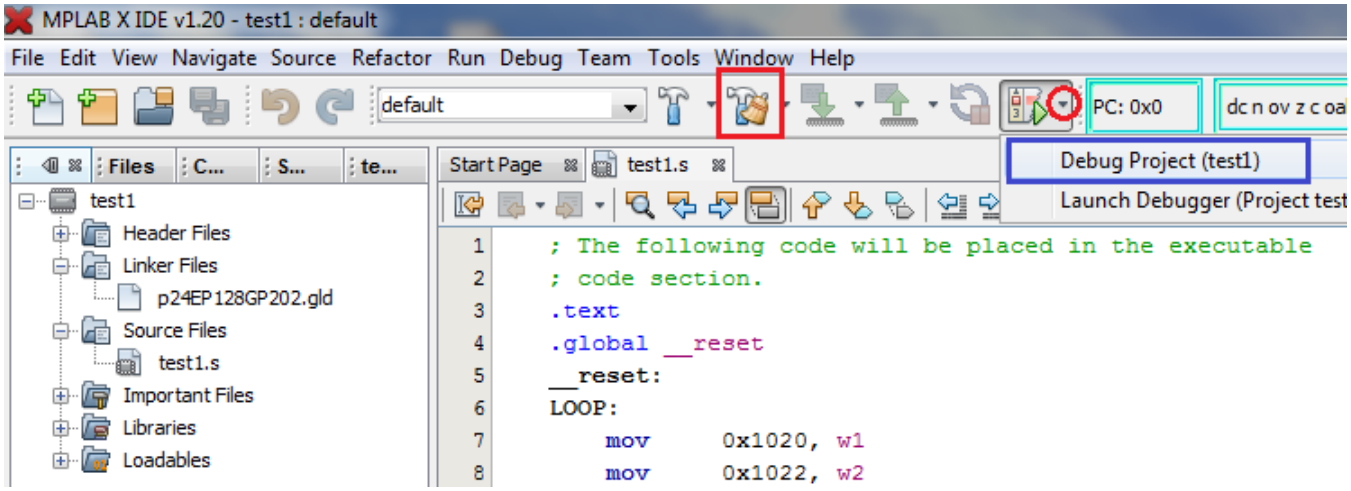
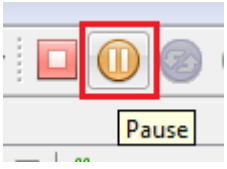
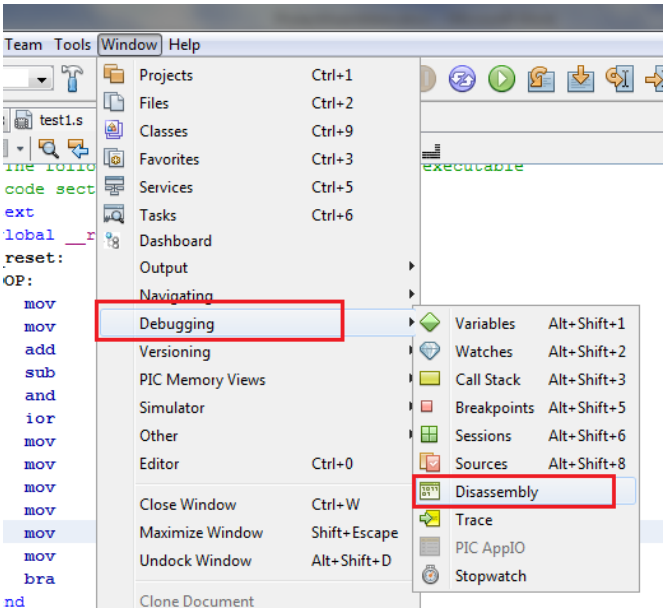


figura 1

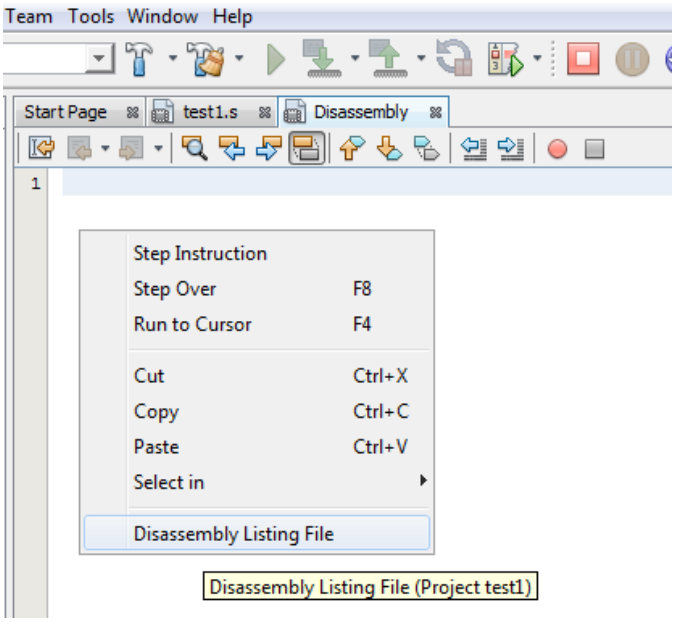
Apoi lansaţi **Clean And Build** prin intermediul icoanei marcată cu dreptunghi roşu în figură. Această operaţie este necesară doar o singură dată, după ce aţi mutat proiectul pe calculatorul dumneavoastră.

7. Lansaţi în execuţie programul test1.s prin apăsarea săgeţii marcate cu cerc roşu în figura 1. Din meniul ca va apare selectaţi **Debug Project** (vezi marcajul cu dreptunghi albastru în figură). Programul va începe să ruleze în simulator şi va apare bara de depanare.
8. Apăsaţi icoana **Pause** pentru a opri execuţia programului: 

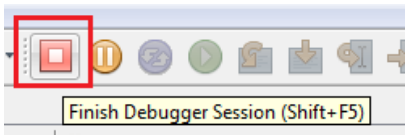
9. Pentru a obține fişierul cu codul maşină activaţi fereastra **Disassembly** conform figurii următoare:



Apoi în fereastra Disassembly faceți clic dreapta şi din meniul contextual ce va apare selectaţi **Disassembly Listing File**, ca în figura următoare:



Fişierul listing va apare într-o fereastră separată şi va fi stocat în folderul proiectului, în subfolderul **Disassembly.** Acest fişier se numeşte **listing.disasm**.

Opriţi execuţia programului cu prin intermediul butonului **Finish**..: 

Copiați codul maşină în fişierul ROM32x16.vhd după procedura explicată la laborator.

Execuţia programului de test în simulator

În afară de asamblare, IDE-ul MPLAB conține şi un simulator. În simulator se pot defini breakpoint-uri, se poate face execuție pas cu pas, se pot vizualiza memoria, registrele şi indicatorii de condiţie, etc. În principiu toate opţiunile de depanare din mediile IDE pe care deja le-aţi utilizat sunt disponibile în MPLAB.

Pentru a executa pas cu pas programul **test1**, procedaţi după cum urmează:

I. Se presupune că aţi executat paşii necesari pentru obținerea listingului ce conține codul maşină. Aceasta este a doua rulare a programului.

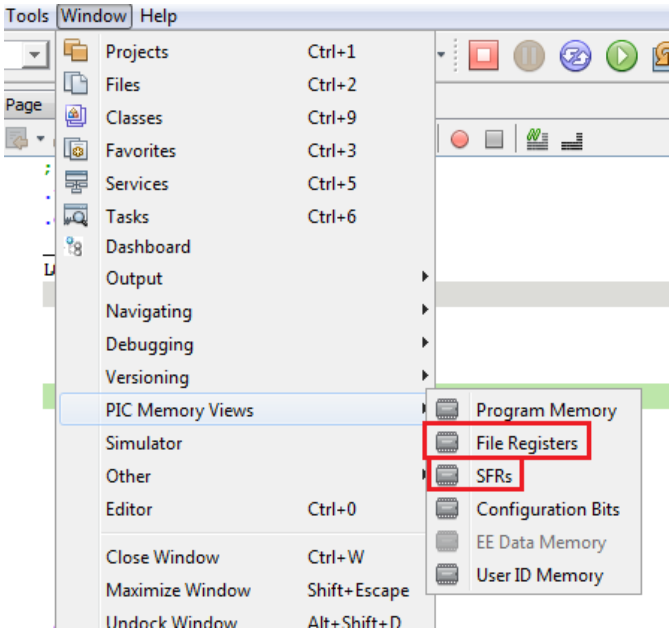
II. Lansați în execuţie **Mplab.** Proiectul test1 ar trebui să se deschidă automat. Dacă nu, procedaţi ca la pasul **5**.

III. Dacă este necesar modificaţi sursa şi apoi executaţi **Build**

IV. Lansaţi simulatorul (Debug) conform indicaţiilor anterioare.

V. Apăsaţi icoana **Pause** pentru a opri execuţia programului.

VI. Pentru a vizualiza registrele şi memoria de date deschideți ferestrele **SFRs** şi **File Registers**, conform figurii următoare:



VII. Iniţializaţi simulatorul prin apăsarea butonului **Reset**, marcat cu 1 în figura următoare:

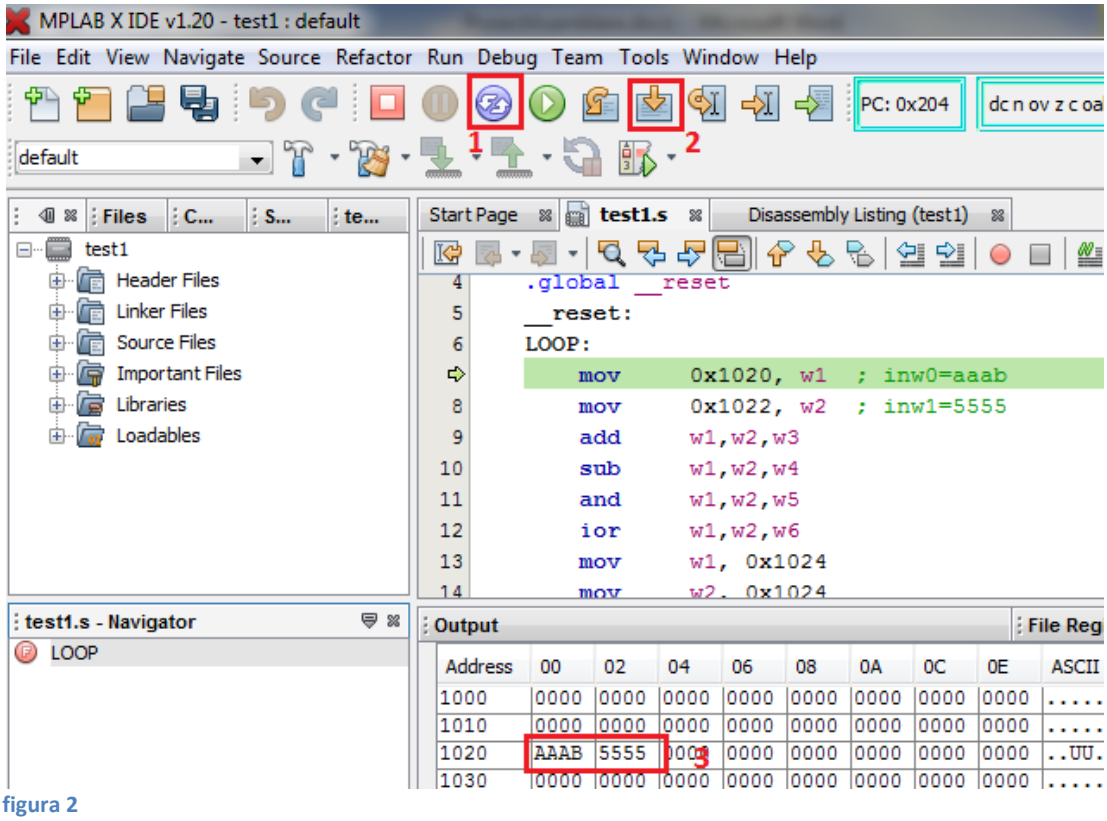
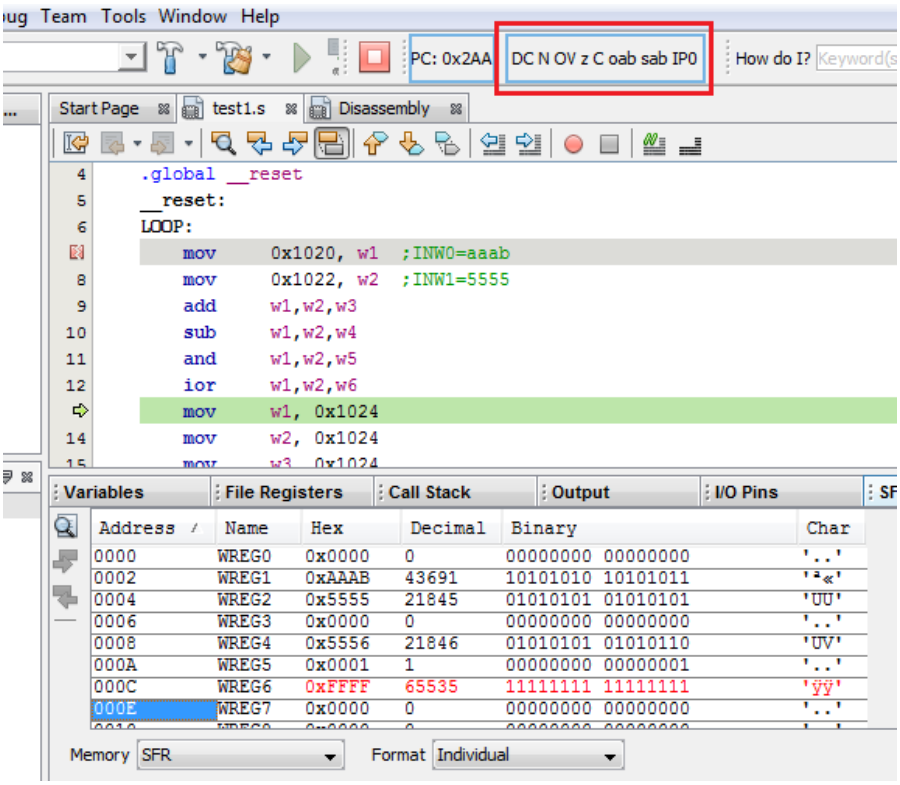


figura 2

VIII. Faceți scroll în fereastra **File register** până când adresa ajunge la valoarea 0x1000. Apoi modificaţi conţinutul locaţiilor 0x1020 (INW0) şi 0x1022 (INW1) ca în figura 2, marcaj 3.

IX. Pentru a executa programul pas cu pas folosiţi butonul **Step Into**, marcat cu 2 în figura 2.

X. Pentru a vedea conținutul registrelor W0-W15 folosiţi cealaltă fereastră **SFRs**. După executarea primelor 4 instrucţiuni se obține situaţia din figura următoare:



Simulatorul afişează şi starea flag-urilor în zona încadrată de dreptunghiul roşu din figura de mai sus. Dacă un flag este ‚0' este scris cu litere mici iar dacă este ‚1' este scris cu litere mari. În figură flagul **z**ero este ‚0' iar **C**arry este ‚1'.

Dacă modificaţi fişierul sursă opriți execuţia cu butonul **Finish**, modificaţi sursa, faceți **Build** şi apoi **Debug**.