Organizarea Calculatoarelor
LABORATOR 10 – Adăugarea instrucţiunii BEQ

SCOPUL LUCRĂRII

În procesorul MIPS redus se vor face modificările şi adăugările necesare pentru ca acesta să poată executa şi instrucţiunea BEQ. Proiect se va simula iar în final se va implementa pe placa S3.

CHESTIUNI TEORETICE

Formatul instrucţiunii BEQ este:

BEQ rs, rt, offset

BEQ este o instrucţiune de salt condiţionat – **B**ranch on **EQ**ual. Condiţia de care depinde saltul este egalitatea dintre conţinutul registrelor **rs** şi **rt**. În caz de egalitate se va face salt la adresa de destinaţie iar în caz contrar execuţia va continua cu instrucţiunea care urmează după BEQ.

Adresa de destinaţie se calculează prin adunarea câmpului *offset* deplasat stânga cu două poziţii la adresa instrucţiunii care **urmează** după BEQ, nu la adresa BEQ.

În mod formal execuţia instrucţiunii BEQ este descrisă după cum urmează:

```
if GPR[rs]==GPR[rt]
      PC ← PC + 4 + offset<<2
else
      PC ← PC + 4
```

Modul de execuţie descris anterior are scop didactic; modul real de execuţie este cel descris în capitolul „Pipeline”.

Implementarea instrucţiunii BEQ în MIPS redus presupune modificări în trei blocuri: **ALU**, **Ctrl** (control) şi **PC_Update**. Aceste modificări sunt marcate cu roşu în figura 1:

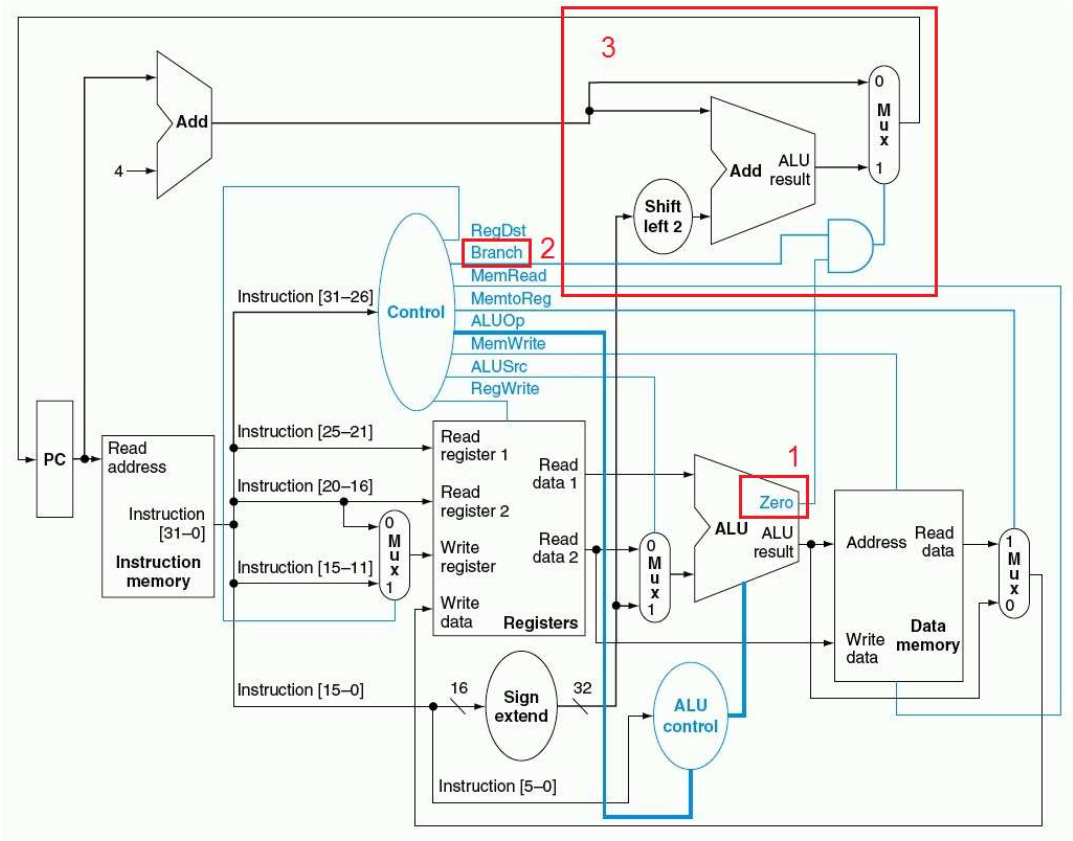

figura 1

Pasul 1: Modificarea ALU, marcaj 1 în figura 1

Se va folosi proiectul mips_sys.

Proiectare al ALU s-a explicat în laboratorul 7, pasul 9.

Mai întâi trebuie adăugat semnalul **Zero** la nivelul entității. După adăugarea acestui semnal, entitatea ALU va arăta astfel:

```
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);
        Zero      : out std_logic
    );
end ALU;
```

Specificaţi semnalul **Zero** folosind conditional signal assignment: dacă **RdData1** este egal cu **RdData2** atunci semnalul **Zero** va fi ‚1' iar în caz contrar va fi ‚0'.

În final regeneraţi simbolul asociat ALU folosind **Design Utilities →Create Schematic Symbol** din fereastra **Processes** şi editaţi-l dacă este necesar. Noul simbol al ALU trebuie să arate ca în figura următoare:

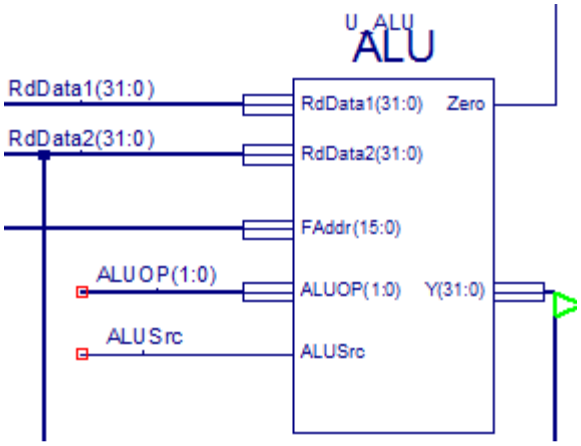

figura 2

Pasul 2: Modificarea blocului de control Ctrl, marcaj 2 în figura 1.

Proiectare al controlului s-a explicat în laboratorul 7, pasul 12.

Instrucţiunea BEQ este codificată conform formatului I:

	6	5	5	16
BEQ	00_0100 (OPCODE)	Rs	Rt	offset

Această instrucţiune se adaugă la celelalte instrucţiuni deja implementate. Codificarea instrucţiunilor deja implementate la care se adaugă BEQ este detaliată în tabelul următor:

	6	5	5	5	5	6
	OPCODE					**Function**
ADD	0	Rs	Rt	Rd	0	0x20=10_0000
SUB	0	Rs	Rt	Rd	0	0x22=10_0010
AND	0	Rs	Rt	Rd	0	0x24=10_0100
OR	0	Rs	Rt	Rd	0	0x25=10_0101
LW	0x23=10_0011	Rs	Rt	Imm16		
SW	0x2b=10_1011	Rs	Rt	Imm16		
BEQ	0x04=00_0100	Rs	Rt	Imm16		

În laboratorul 7 semnalele de control s-au sintetizat conform tabelului următor:

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

Adăugarea instrucţiunii **BEQ** înseamnă o linie suplimentară şi o coloană suplimentară în această tabelă. Linia şi coloana suplimentară sunt marcate cu roşu în tabelul următor:

tabelul 1

	Opcode	**Function**	**Branch**	**ALUSrc**	**ALUOP**	**MemWr**	**Mem2Reg**	**RegWr**	**RegDest**
ADD	00_0000	10_0000	**0**	**0**	**00**	**0**	**0**	**1**	**1**
SUB	00_0000	10_0010	**0**	**0**	**01**	**0**	**0**	**1**	**1**
AND	00_0000	10_0100	**0**	**0**	**10**	**0**	**0**	**1**	**1**
OR	00_0000	10_0101	**0**	**0**	**11**	**0**	**0**	**1**	**1**
LW	10_0011	-	**0**	**1**	**00**	**0**	**1**	**1**	**0**
SW	10_1011	-	**0**	**1**	**00**	**1**	**0**	**0**	**0**
BEQ	00_0100	-	**1**	**x**	**xx**	**0**	**x**	**0**	**x**

Coloana suplimentară corespunde noului semnal de control **Branch**. Acest semnal este activ (‚1’ logic) numai când se execută instrucţiunea BEQ, fiind ‚0’ pentru orice altă instrucţiune.

Linia suplimentară corespunde instrucţiunii **BEQ**. Pentru această instrucţiune semnalele de control definite anterior în laboratoarele 6 şi 7 trebuie implementate după cum urmează:

- **ALUSrc** nu contează deoarece în ALU s-a alocat un comparat de egalitate pentru conţinutul registrelor rs şi rt.
- **ALUOP** nu contează deoarece nu contează ce operaţie se face; contează numai rezultatul comparaţiei.
- **MemWr** trebuie să fie ‚0’ deoarece BEQ nu scrie în memoria de date.
- **RegWr** trebuie să fie ‚0’ deoarece BEQ nu scrie nici un registru.
- **Mem2Reg** şi **RegDest** nu contează deoarece **RegWr** este ‚0’

In concluzie, pentru modificarea blocul de control procedaţi după cum urmează:

1. La nivelul entităţii ctrl adăugaţi semnalul de ieşire **Branch**.
2. Specificaţi semnalul **Branch** cu un selected signal assignment ca în tabelul 1.

3. Verificaţi corectitudinea implementării tuturor celorlalte semnale de control din tabelul 1; este posibil ca implementările să nu mai fie aceleaşi. Unde este cazul, modificaţi. **Atenţie**: în loc de valoarea x folosiţi valoarea 0.
4. Regeneraţi simbolul blocului de control. Acest simbol va arăta că în figura următoare :

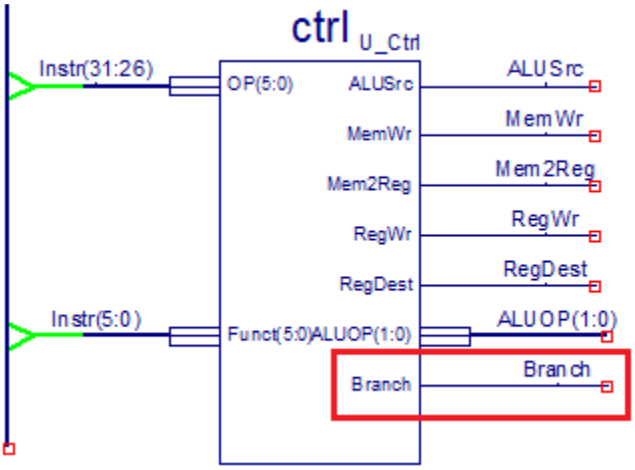

figura 3

Pasul 3: Modificarea blocului PC_Update, marcaj 3 în figura 1

Secţiunea din schema din figura 1 în care se calculează noua valoare a PC este detaliată în figura 4:

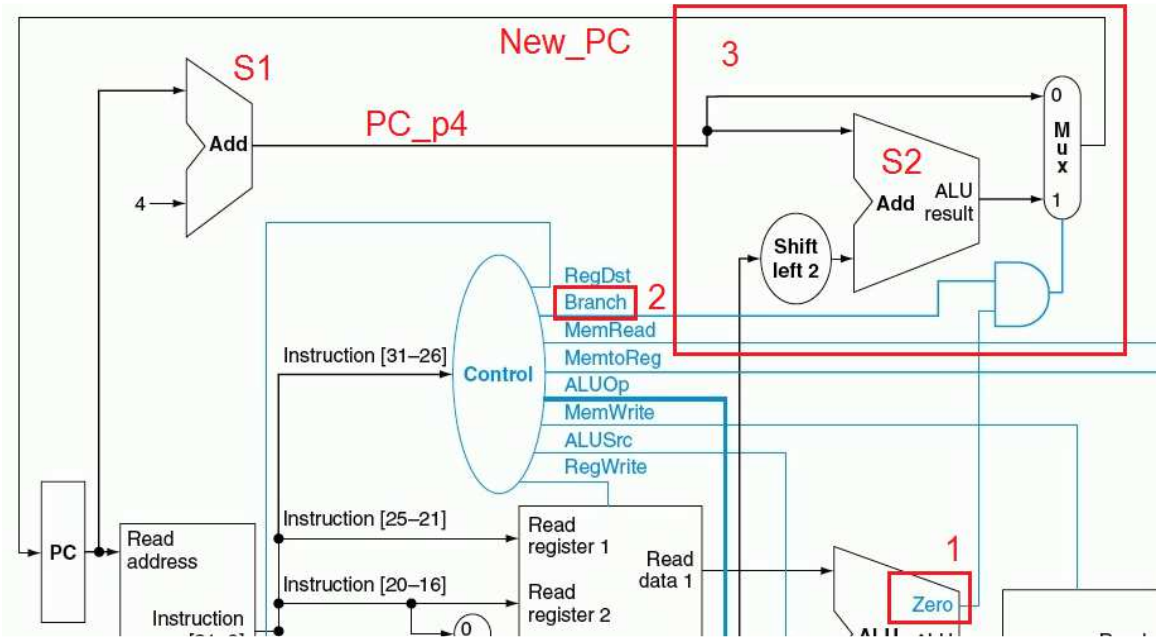

figura 4

La nivelul entităţii trebuie declarate semnalele generate anterior, şi anume **Branch** generat de control şi **Zero** generat de ALU. Se mai adaugă şi offsetul din instructiune. Entitatea blocului PC_Update va avea structura următoare:

```
entity PC_Update is
    Port (
      Zero      : in  STD_LOGIC;
      Offset    : in  STD_LOGIC_VECTOR (15 downto 0);
      Branch    : in  STD_LOGIC;
      PC        : in  STD_LOGIC_VECTOR (31 downto 0);
      New_PC    : out STD_LOGIC_VECTOR (31 downto 0)
    );
end PC_Update;
```

Blocul PC_Update definit în laboratorul 6 constă numai din sumatorul S1 din figura 4. Acest sumator a

fost descris prin declaraţia:

```
New_PC <= PC+4;
```

După cum se vede în figura 4 adăugarea instrucţiunii BEQ face ca noua valoarea PC să nu mai fie PC+4, dar PC+4 este în continuare necesar pentru calculul valori lui `New_PC`. Din acest motiv mai întâi declarăm semnalul PC_p4 **la nivelul arhitecturii** şi apoi îl generăm astfel:

```
PC_p4 <= PC+4;
```

Noua valoare a PC depinde de semnalele `Branch` şi `Zero`. Dacă `Branch` şi `Zero` sunt simultan active (‚1' logic) atunci `New_PC` este `PC_p4` adunat cu **offsetul extins ca semn şi deplasat stânga cu două poziţii**. În caz contrar `New_PC` este `PC_p4`.

Problema constă în extinderea semnului offsetul şi deplasarea stânga 2 poziţii. Offsetul extins ca semn şi deplasat stânga cu doi se va numi in continuare **deplasament - depl** deoarece reprezintă cu cât se „deplasează" PC-ul înainte sau înapoi faţă de valoarea curentă. Pentru aceasta se va proceda asemănător cu generarea semnalului SEAddr în ALU. Mai întâi vom declara la nivelul arhitecturii vectorul `depl` cu 32 de elemente:

```
signal depl      : std_logic_vector(31 downto 0);
```

Vom specifica deplasamentul pe grupe de biţi:

1. Deplasarea cu doi înseamnă că biţi zero şi unu ai deplasamentului sunt ‚0'. Folosiţi slice sau asignarea element cu element. Conceptul de slice a fost introdus în Laboratorul 7, pasul 9
2. Biţii 17 – 2 ai deplasamentului sunt chiar offsetul.
3. Celelalte elemente ale deplasamentului au toate valoarea semnului lui `Offset`, adică `Offset(15)`. În laboratorul 7 s-a prezentat o varianta de extindere a semnului atunci când s-a calculat SEAddr. O altă variantă, probabil mai simplă, pentru **a iniţializa toate elementele unui vector la aceeaşi valoare** este construcţia

   vector `<= (others =>` *valoare*`);`

   unde *valoare* poate fi o constantă sau un semnal de tip std_logic. De exemplu, pentru a iniţializa cu zero toate elementele unui vector se foloseşte `vector <= (others => ‚0');` Pentru extinderea semnului folosiţi această idee sau ideea de la SEaddr!

În concluzie `New_PC` este `PC_p4 + depl` dacă `Branch` şi `Zero` sunt simultan active şi `PC_p4` în caz contrar. Descrieţi pe `New_PC` cu un conditional signal assignment.

După ce actualizaţi simbolul pentru PC_Update faceţi conexiunile din figura 5. Etichetaţi firul care uneşte terminalul Zero de la ALU cu terminalul Zero de la PC_Update cu Zero, pentru a-l identifica mai uşor în simulare.

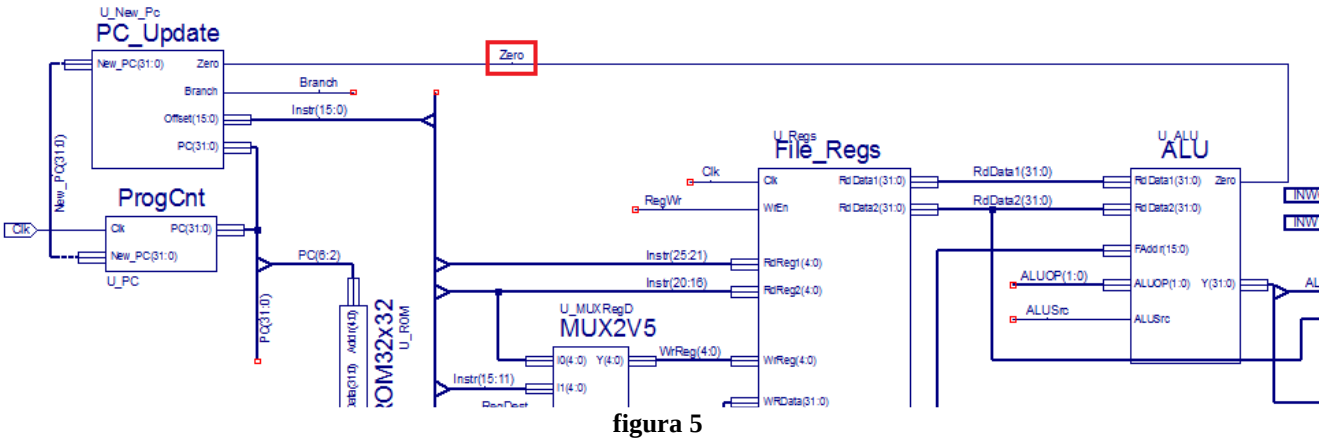

figura 5

Pasul 4: Simularea MIPS

Simularea se va face pentru următorul program:

```
rep:  lw    $2, 0x40($0)      # INW0=0xAAAA_AAAB
      lw    $3, 0x44($0)      # INW1=0x5555_5555
      add   $4, $2, $3
      sub   $5, $2, $3
      and   $6, $2, $3
      or    $7, $2, $3
      sw    $2, 0x48($0)
      sw    $3, 0x48($0)
      sw    $4, 0x48($0)
      sw    $5, 0x48($0)
      sw    $6, 0x48($0)
      sw    $7, 0x48($0)
      beq   $2, $3, rep       # Reg $2 ≠ Reg $3  => Saltul nu se face.
      beq   $0, $0, rep       # Reg $0 = Reg $0  => Saltul se face.
```

Acest program este programul **test1** din laboratorul 9 la care s-au adăugat instrucţiunile BEQ.

Pentru a nu modifica mereu memoria de program şi pentru a repeta cu uşurinţă unele teste când este necesar anterior s-a creat un fişier ROM32x32 pentru fiecare program. În acest moment există fişierele ROM32x32_test1.vhd pentru testul iniţial şi ROM32x32_3a2b.vhd pentru programul care calculează 3a-2b în laboratorul 10. Numele entităţii şi cel al arhitecturii sunt identice în toate fişierele.

Programul de mai sus se va memora în fişierul ROM32x32_BEQ.vhd. acest fişier se poate obţine uşor dacă la o copie a fişierul ROM32x32_test1.vhd se adaugă cele două linii pentru BEQ. Pentru a testa un anume cod în proiectul ISE se face Remove pentru fişierul ROM curent şi se face Add pentru fişierul cu care se vrea a se lucra.

Folosiţi testbenchul **tb_mips**. După cum s-a precizat în laboratoarele anterioare, un testbench se poate genera pentru orice modul din proiect, nu numai pentru modulul din vârful ierarhiei. Mai mult, într-un proiect pot exista mai multe testbenchuri, pentru diverse module. Fiecare testbench are propriul fişier cu forme de undă. Formele de undă salvate pentru tb_sys nu pot fi folosite cu tb_mips şi nici invers.

Lansaţi Modelsim. Simulaţi. Vizualizaţi semnalele din figura 6.

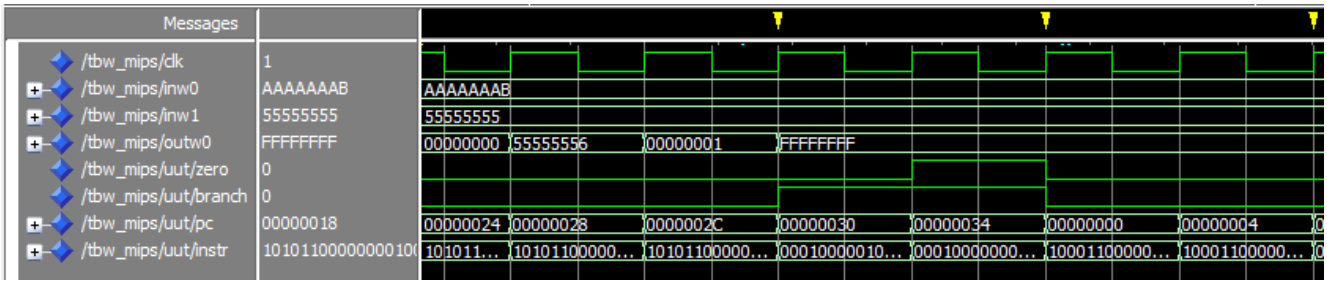

figura 6

Prima instrucţiune BEQ, cea pentru care saltul nu se face, se identifică în simulare după valoarea PC-ului şi a lui instr. Primul BEQ se execută când PC=`0x00000030`. Deoarece conținutul registrul 2 diferă de conținutul registrului 3, saltul nu se va executa şi PC va avansa la valoarea `0x00000034`. Aici se află `beq $0,$0, rep`. Deoarece reg 0=reg 0, saltul se va face şi următoarea valoare a lui PC va fi `0x00000000`.

Dacă implementarea BEQ este corectă veţi obţine comportamentul din figura de mai sus. Dacă nu, depanaţi. Vizualizaţi toate semnale din modulul pc_update. Inspiraţi-vă din laboratorul 8, pasul 5 subpasul 9.

Daca BEQ funcţionează, chemaţi profesorul pentru validare! Dacă aţi ajuns aici aveţi notă 5.

Pasul 5: Execuţia pe placa S3

Creaţi ROM32x32_X.vhd pentru următorul program:

```
rep:        lw    $2, 0x40($0)
            lw    $3, 0x44($0)
            sub   $4, $2, $3
            beq   $3, $2, display
            add   $4, $2, $3
display:    sw    $4 ,0x48($0)
            beq   $0, $0, rep
```

Ce trebuie să se întâmple când rulează acest cod?

Răspundeţi la întrebarea de mai sus şi apoi configurați placa S3. Fişierul de constrângeri este deja creat. **Dacă funcționează, chemaţi profesorul pentru validare! Dacă ați ajuns aici aveți notă o notă între 5 şi 10.**