UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

PROCESORUL MIPS CICLU UNIC- 16 BITI

Student: Hădărău Elena

Grupa:30223

a) Cele 4 instructiuni suplimentare

1. Tipul R

- xor $rd, $rs, $rt RTL: RF[rd] ←RF[rs] ^ RF[rt] //Exclusive OR

- sra $rd, $rs, sa RTL: RF[rd] ←RF[rs] >> sa //Shift right arithmetic

2. Tipul I

- bne $rs, $rt, imm RTL: If(RF[rs] !=RF[rt]) then PC ←PC+1 + S_Ext(imm)

Else PC ←PC+1 // Branch on not equal

- andi $rt, $rs, imm RTL: RF[rt] ←RF[rs] & Z_Ext(imm) // Logical AND unsigned constant

b) Semnale control MIPS16 pentru Anexa 5

Tipuri de operații care se pun în paranteză la ALUOp si ALUCtrl: {(+), (-), (&), (|), (^), (<<*l*), (<<*lv*), (>>*l*), (>>*a*), (<)}, & - AND, | - OR, ^ - *XOR, l - logic, a - aritmetic, v - cu variabilă*

Instruc țiune	Opcode *Instr(15-13)*	RegDst	ExtOp	ALUSrc	Branch	SBne	Jump	Mem Write	Memto Reg	Reg Write	ALUOp (1:0)	func *Instr(2-0)*	ALUCtrl (2:0)
Add	000	1	0(x)	0	0	0	0	0	0	1	00 (tip R)	000	000 (+)
Sub	000	1	0	0	0	0	0	0	0	1	00	001	001(-)
Sll	000	1	0	0	0	0	0	0	0	1	00	010	010(<<l)
Srl	000	1	0	0	0	0	0	0	0	1	00	011	011(>>l)
And	000	1	0	0	0	0	0	0	0	1	00	100	100(&)
Or	000	1	0	0	0	0	0	0	0	1	00	101	101(|)
Xor	000	1	0	0	0	0	0	0	0	1	00	110	110(^)
Sra	000	1	0	0	0	0	0	0	0	1	00	111	111(>>a)
Addi	001	0	1	1	0	0	0	0	0	1	01(+)	000(x)	000 (+)
Lw	010	0	1	1	0	0	0	0	1	1	01(+)	000(x)	000 (+)
Sw	011	0	1	1	0	0	0	1	0	0	01(+)	000(x)	000 (+)
Beq	100	0	1	0	1	0	0	0	0	0	10(-)	000(x)	001(-)
Andi	101	0	0	1	0	0	0	0	0	1	11(&)	000(x)	100(&)
Bne	110	0	1	0	0	1	0	0	0	0	10(-)	000(x)	001(-)
J	111	0	0(x)	0	0	0	1	0	0	0	00	000(x)	000 (x)

c) Program- memoria ROM

Programul implementat pe procesor (instrucțiunile se află pe primele 12 adrese din memoria ROM) are scopul de a contoriza numărul elementelor pare dintr-un vector de 10 elemente. Elementele vectorului sunt stocate în memoria RAM pe primele 10 adrese (adresele: 0-9). Tot în memoria RAM, la adresa 12, se va stoca, la finalul execuției programului, numărul elementelor pare. Pentru a verifica dacă elementul preluat din memorie este par am folosit instrucțiunea **andi** pentru a face un și logic cu constanta 1, deoarece toate numerele pare au bitul cel mai nesemnificativ egal cu 0. Astfel, dacă elementul cu care se face și logic este par voi avea in registrul destinație valoarea 0, iar dacă numărul e impar va fi valoarea 1. În plus, folosesc instrucțiunea **bne,** cu ajutorul căreia compar valoarea din registrul destinație al instrucțiunii andi cu valoarea din registrul 0 (adică chiar valoarea 0). Dacă valorile celor 2 registre sunt diferite, semnalul corespunzător instrucțiunii bne (semnalul sbne=1) și înseamnă că s-a detectat un număr impar, deci se va face un salt peste instrucțiunea următoare în care se va salva numărul elementelor pare.

Programul în C

```
int A[10]={1,2,3,4,5,6,7,8,9,10};

int catepare=0;

for(int i=0;i<10;i++)

        if(A[i]%2==0)

                catepare++;
```

Programul în limbaj de asamblare

```
0: add $1, $0, $0          #i=0, contorul buclei
1: addi $2,$0,10           #se pune în registrul 2 numărul de iterații (10)
2: add $3, $0,$0           #inițializare indexului locației de memorie
3: add $4,$0,$0            #catepare=0 – în registrul 4 se va reține nr. elementelor pare
4: beq $2,$1,7             #verific dacă s-au făcut 10 iterații
5: lw $5,0($3)             #în carc în registrul 5 elementul curent din șir
6: andi $6, $5, 1          #instr. 6 și 7 – verific dacă elementul e par sau nu; dacă e impar
7: bne $6,$0,1              #se face salt la instr. 9
8: addi $4,$4,1            #dacă elementul verificat e par, atunci se incrementează reg4
9: addi $3, $3, 1          #incrementez indexul locației pt urm. element din vector
10: addi  $1, $1, 1        #actualizare contor buclă – i++
11: j 4                    #salt la începutul buclei
12: sw $4,12($0)           #salvarea în memorie la adresa 12 a numărului de elemente pare
```

Programul în cod mașină

0: b”000_000_000_001_0_000” # instr. in hexa: x”0010”

1: b”001_000_010_0001010” #x”210A”

2: b”000_000_000_011_0_000” #x”0030”

3: b”000_000_000_100_0_000” #x”0040”

4: b”100_010_001_0000111” #x”8887”

5: b”010_011_101_0000000” #x”4E80”

6: b”101_101_110_0000001” #x”B701”

7: b”110_110_000_0000001” #x”D801”

8: b”001_100_100_0000001” #x”3201”

9: b”001_011_011_0000001” #x”2D81”

10: b”001_001_001_0000001” #x”2481”

11: b”111_0000000000100” #x”E004”

12: b”011_000_100_0001100” #x”620C”

d) Trasarea programului

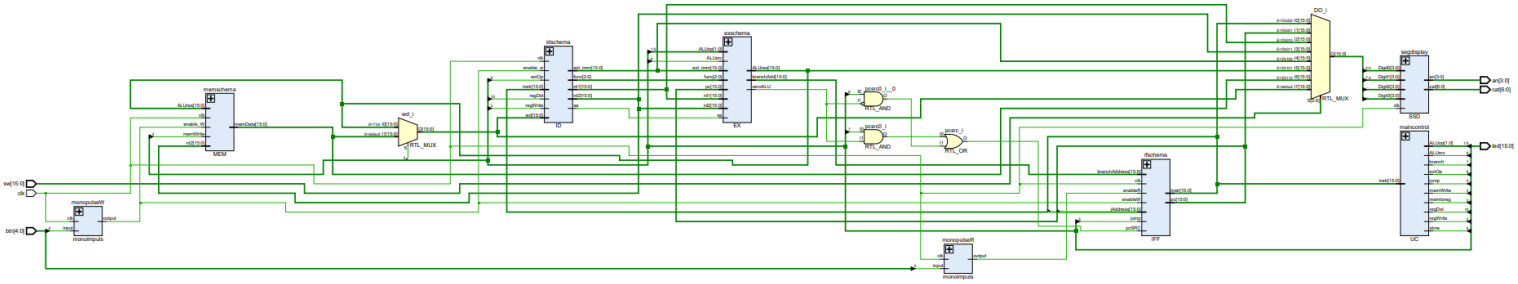
Pas	SW(7:5)	“000”	“001”	“010”	“011”	“100”	“101”	“110”	“111”	Branch	Jump
	Instrucțiune	Instr. (hexa)	PC+1	RD1	RD2	Ext_imm	ALUres	memData	WD	Address	Address
0	add $1, $0, $0	x”0010”	x”0001”	x”0000”	x”0000”	x”0010”	x”0000”	x”0001”	x”0000”		
1	addi $2,$0,10	x”210A”	x”0002”	x”0000”	x”0000”	x”000A”	x”000A”	x”000F”	x”000A”		
2	add $3, $0,$0	x”0030”	x”0003”	x”0000”	x”0000”	x”0030”	x”0000”	x”0001”	x”0000”		
3	add $4,$0,$0	x”0040”	x”0004”	x”0000”	x”0000”	x”0040”	x”0000”	x”0001”	x”0000”		
4	beq $2,$1,7	x”8887”	x”0005”	x”000A”	x”0000”	x”0007”	x”000A”	x”000F”	x”000A”	x”000C”	
5	lw $5,0($3)	x”4E80”	x”0006”	x”0000”	x”0000”	x”0000”	x”0000”	x”0001”	x”0001”		
6	andi $6, $5, 1	x”B701”	x”0007”	x”0001”	x”0000”	x”0001”	x”0001”	x”0002”	x”0001”		

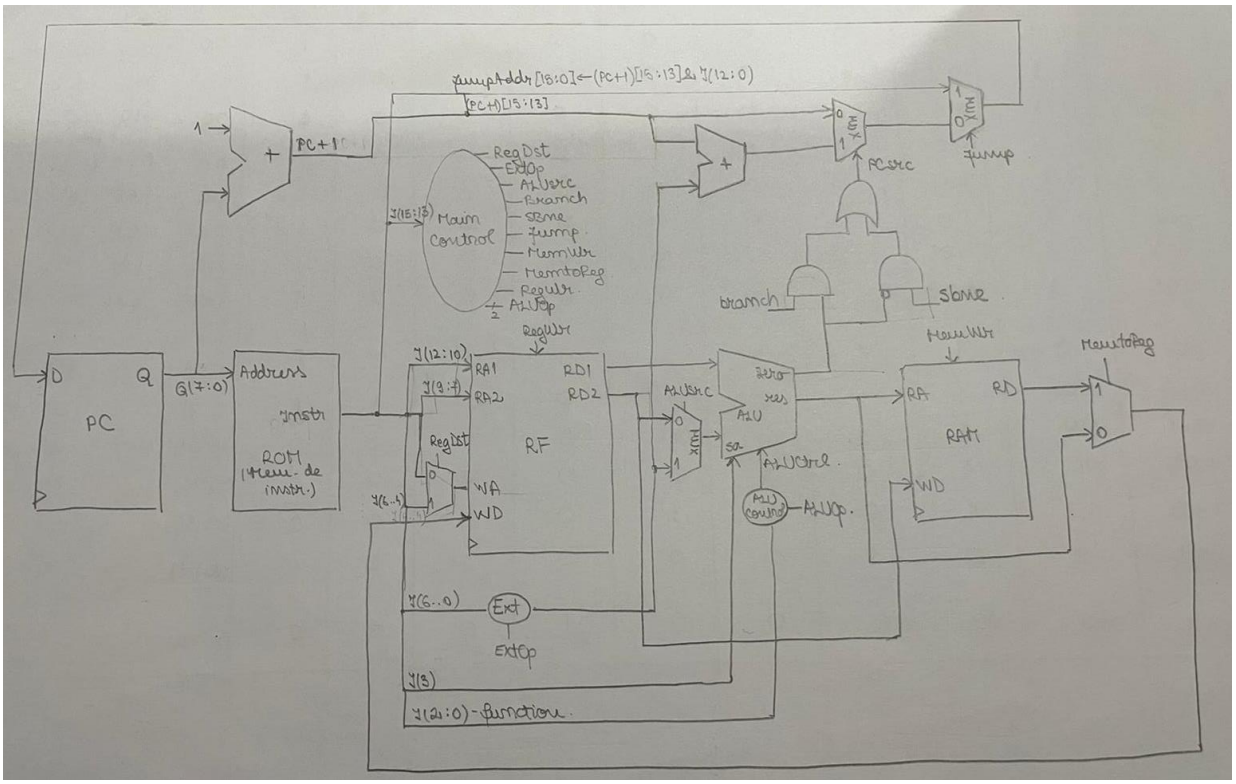
7	bne $6,$0,1	x”D801”	x”0008”	x”0001”	x”0000”	x”0001”	x”0001”	x”0002”	x”0001”	x”0009”	
8	addi $3, $3, 1	x”2D81”	x”000A”	x”0000”	x”0000”	x”0001”	x”0001”	x”0002”	x”0001”		
9	addi $1, $1, 1	x”2481”	x”000B”	x”0000”	x”0000”	x”0001”	x”0001”	x”0002”	x”0001”		
10	j 4	x”E004”	x”000C”	x”0000”	x”0000”	x”0004”	x”0000”	x”0001”	x”0000”		x”0004”
	sw $4,12($0)	x”620C”	x”000D”	x”0000”	x”0005”	x”000C”	x”000C”	x”000F”	x”000C”		

- Linia corespunzătoare instr. de la adresa 12 (ultima linie), este completată pentru a 10-a iterație când se iese din buclă, astfel în registrul 4 voi avea numărul elementelor pare din vector (pentru exemplul dat va fi 5).

e) Nu există părți din procesor incomplete.

f) Nu există erori la implementarea în VHDL.
Schema corespunzătoare procesorului MIPS ciclu unic pe 16 biți cu instrucțiunile suplimentare adăugate este următoarea:





g) Programul a fost testat pe plăcuță FPGA BASYS 3 și funcționează conform așteptărilor.