

Universitatea Tehnică din Cluj-Napoca
Facultatea de Automatică și Calculatoare

RAPORT

PROCESOR MIPS CICLU UNIC

Activitate Laborator
ARHITECTURA CALCULATOARELOR

Nume student: Maria-Magdalena Cret
Grupa 30223

Cuprins

1 Obiectivul proiectului	3
2 Instrucțiuni suplimentare	3
2.1 Tabelul cu cele 15 instrucțiuni pentru procesorul MIPS	5
2.2 Diagrama în varianta finală a procesorului MIPS realizat	5
3 Proiectare	6
4 Programul executat de procesor	6
5 Implementare în VHDL	9
5.1 Unitatea de extragere a instrucțiunilor IFetch	10
5.2 Unitatea de decodificare a instrucțiunilor ID	10
5.3 Unitatea de Control UC	11
5.4 Unitatea de execuție EX	11
5.5 Unitatea de memorie MEM	12
5.6 Unitatea de scriere a rezultatului WB	13
6 Testare	16
6.1 Tabelul cu valorile pentru principalele semnale din MIPS (Tabel de trasare):	17
7 RTL Schematic	18
8 MIPS 32 BITI PIPELINE - ACTUALIZARE MIPS SINGLE CICLE	18
8.0.1 Arhitectura MIPS Pipeline-desen	19
8.0.2 Arhitectura MIPS Pipeline-actualizat(la fel desenat, dar cu vizibilitate mai bună)	20
8.0.3 Testare	20

1 Obiectivul proiectului

Obiectivul principal a acestui proiect este acela de a se projecța și implementa un procesor MIPS care să realizeze un program unic, implementat de către student în timpul celor 4 laboratoare dedicate pentru acest subiect. Procesorul a cărui implementare se cere trebuie realizat la fel ca cel studiat la curs, pe 36 de biți. Codificarea Opcode-ului din Instruction, programul din memoria de program, instrucțiunile suplimentare, toate au o abordare proprie deoarece studentul decide fiecare dintre acestea.

2 Instrucțiuni suplimentare

În afară de instrucțiunile de bază pentru implementarea MIPS-ului, care se dău implicit, va trebui ca fiecare student să își aleagă 4 instrucțiuni suplimentare pentru procesor. Două dintre acestea sunt de tipul R, iar celelalte două sunt de tipul I. Dimensiunea instrucțiunilor procesorului este de 32 de biți. Formatul instrucțiunilor pe 32 biți este următorul:

Tip R	6	5	5	5	5	6
	opcode=0	rs	rt	rd	sa	function
	31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
Tip I	6	5	5	16		
	opcode	rs	rt	immediate/offset		
	31 ... 26	25 ... 21	20 ... 16	15	...	0
Tip J	6	26				
	opcode	address				
	31 ... 26	25	...	0		

Tipul și formatul pentru instrucțiuni

Astfel că am ales pentru implementarea procesorului MIPS cu program unic, operațiile suplimentare de tip R: XOR și SLT, unde XOR semnifică *SAU-Exclusiv logic între două registre, memorează rezultatul în alt registru* și SLT semnifică *Set on Less Than (signed)*. De asemenea, pentru XOR și SLT se specifică mai jos operația, sintaxa și formatul pentru acestea; operațiile suplimentare de tip I: ANDI = *ȘI logic între un registru și o valoare imediată, cu rezultatul în alt registru* și BNE = *Branch on Not Equal = Salt conditionat dacă două registre sunt diferite* registru.

Operații de tip R:

XOR \Rightarrow

Operație : $\$d = \$s \hat{\wedge} \$t; PC = PC + 4;$

Sintaxă : xor \$d, \$s, \$t

Format : 000000 ssss tttt dddd 00000 100110

SLT \Rightarrow

Operătie : $PC = PC + 4; \text{ if } \$s < \$t \text{ then } \$d = 1 \text{ else } \$d = 0;$

Sintaxă : slt \$d, \$s, \$t

Format : 000000 ssss tttt dddd 00000 101010

Observație: Pentru operațiile de tip R, primele 6 zerouri reprezintă OpCodul, care va fi la fel pentru toate operațiile de acest fel.

Operații de tip I:

ANDI \Rightarrow

Operatie : \$t = \$s & ZE(imm); PC = PC + 4;

Sintaxă : andi \$t, \$s, imm

Format : 001100 ssss tttt ii iiiiiiiiiiiiiii

BNE \Rightarrow

Operatie : if \$s \neq \$t then PC = (PC + 4) + (SE(offset) << 2) else PC = PC + 4;

Sintaxă : bne \$s, \$t, offset

Format : 000101 ssss tttt ooooooo0oooooooooooo

Observație: Pentru operațiile de tip I, primele 6 zerouri reprezintă OpCodul, care NU va fi la fel pentru toate operațiile de acest fel, este unic pentru fiecare operație, atfel ca prin acesta se diferențiază operația care se realizează, de tipul I.

Explicații implementare: Se adaugă un semnal de control Branch_N pentru a putea implementa instrucțiunea BNE. Astfel că procesorul MIPS după schema clasică prezentată la laborator va mai aveam în plus o poartă logică AND la care se va lega semnalul Branch_N, care este un semnal pe care îl găsim în Main Control cu toate celealte semnale. Se mai adaugă în plus și un inversor pentru Zero pentru că BNE se execută atunci când din ALU, pe semnalul ZERO ieșe 0, astfel ca ieșirea din ALU este diferită de 0. De reținut este că ieșirea Zero este 1 atunci când operațiile din ALU au ca output 0. Branch_N este activ și se realizează pentru instrucțiunea BNE atunci când outputul din ALU este diferit de 0, în caz contrar se activează Branch, într-un singur caz, pentru instrucțiunea BEQ. Operația de BNE sau BEQ se alege pe baza unei porti noi introduse, SAU logic. În acest caz PCSrc va deveni (Zero and Branch) ori ($\overline{\text{Zero}}$ and Branch_N).

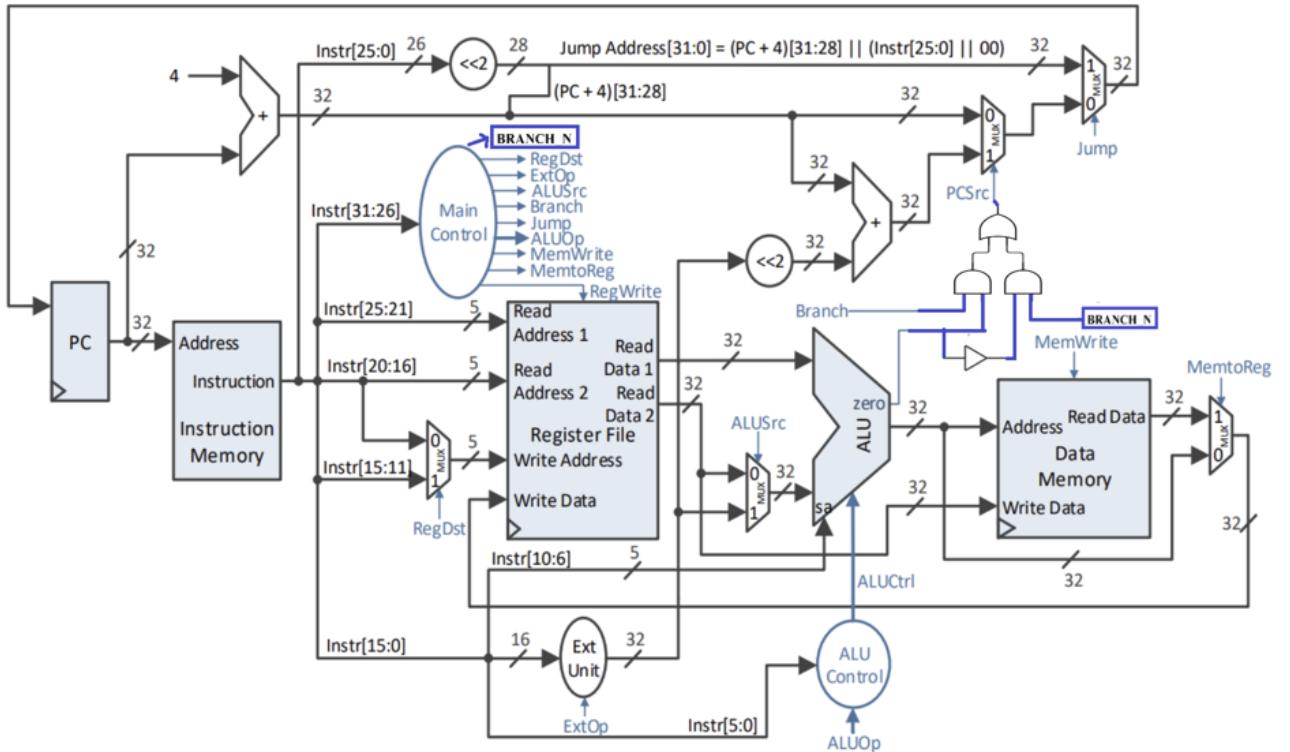
2.1 Tabelul cu cele 15 instrucțiuni pentru procesorul MIPS

ADD	ADDITION :
SUB	SUBTRACTION :
SLL	SHIFT LEFT LOGICAL:
SRL	SHIFT RIGHT LOGICAL:
AND	LOGICAL AND:
OR	LOGICAL OR:
XOR	LOGICAL XOR:
SLT	SET ON LESS THAN:
LW	LOAD WORD:
SW	STORE WORD:
BEQ	BRANCH ON EQUAL:
ADDI	ADD IMMEDIATE:
BNE	BRANCH ON NOT EQUAL:
ANDI	AND IMMEDIATE:
J	JUMP:

TIP R TIP I TIP J

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 (26) instrucțiuni de tip R și 63 (26-1) instrucțiuni de tip I și J.

2.2 Diagrama în varianta finală a procesorului MIPS realizat



3 Proiectare

Pentru proiectarea acestui procesor a fost întâi nevoie de un tabel cu Semnale de control MIPS32 pentru fiecare componentă, unde s-au inițializat OpCodurile și pentru fiecare instrucțiune din tabelul cu cele 15 instrucțiuni s-au notat ce semnale sunt active (1), inactive(0) sau nu conțează(X). Cum am spus și mai sus am adăugat un semnal de control în plus, Branch_N pentru instrucțiunea BNE pe care l-am adăugat în tabel, care este activ doar pentru instrucțiunea BNE. ALUOp este codificat pe 2 biți deoarece, doar de atâtia este nevoie. ALUCtrl este codificat pe 3 biți, având în dreptul codificării binare și semnul operației pe care o realizează. Am considerat pentru SLT, operația de compare, astfel că am notat acest lucru în dreptul codificării.

Semnale de control MIPS32

Semnale de Branch optionale: ? ∈ {gez, ne, gtz}, se va înlocui ? cu o valoare din paranteză, dacă e cazul
 Tipuri de operații care se pun în paranteză la ALUOp și ALUCtrl:
 (+), (-), (&), (|), (^), (<<), (<<lv), (>>l), (>>a), (<)

Semnificații: & - AND, | - OR, ^ - XOR, l - logic, a - aritmetic, v - cu variabilă

Instrucțiune	Opcode Instr[31-26]	Reg Dst	ExtOp	ALUSrc	Branch	BRANCH_N	Jump	JmpR (optional)	Mem Write	Memto Reg	Reg Write	ALUOp[1:0]	function Instr[5-0]	ALUCtrl[2:0]
ADD	000000	1	X	0	0	0	0		0	0	1	11(R)	000000	001(+)
SUB	000000	1	X	0	0	0	0		0	0	1	11(R)	000001	010(-)
SLL	000000	1	X	X	0	0	0		0	0	1	11(R)	000010	100(<<)
SRL	000000	1	X	X	0	0	0		0	0	1	11(R)	000011	101(>>)
AND	000000	1	X	0	0	0	0		0	0	1	11(R)	000100	011(&)
OR	000000	1	X	0	0	0	0		0	0	1	11(R)	000101	110()
XOR	000000	1	X	0	0	0	0		0	0	1	11(R)	000110	111(^)
SLT	000000	1	X	0	0	0	0		0	0	1	11(R)	000111	000(compare)
LW	100000	0	1	1	0	0	0		0	1	1	10(+)	X	001(+)
SW	100001	X	1	1	0	0	0		1	0	0	10(+)	X	001(+)
BEQ	100010	X	1	0	1	0	0		0	X	0	01(-)	X	010(-)
ADDI	100011	0	1	1	0	0	0		0	0	1	10(+)	X	001(+)
ANDI	100101	0	0	1	0	0	0		0	0	1	00(&)	X	011(&)
BNE	100100	X	1	0	0	1	0		0	X	0	01(-)	X	010(-)
J	111111	X	X	X	X	X	1		0	X	0	X	X	X

4 Programul executat de procesor

Cerință:

Să se înlocuiască toate elementele dintr-un sir cu împărțirea lor la 8 ca întreg, daca sunt mai mici decât X, daca sunt între X si Y cu dublul lor și daca sunt mai mari ca Y se vor înlocui cu 1. Sirul se află în memorie începând cu adresa A ($A \geq 4$) și are N elemente. A, N, X, Y se citesc din memorie de la adresele 0, 1, 2, respectiv 3.

Constrângere: $X \leq Y$

Pentru verificare, se poate adăuga o buclă de citire a elementelor sirului, la final.

```

1 #initializare registrii
2 andi $1 $1 0      # => 100101_00001_00001_0000000000000000
3 andi $2 $2 0      # => 100101_00010_00010_0000000000000000
4 andi $3 $3 0      # => 100101_00011_00011_0000000000000000
5 andi $4 $4 0      # => 100101_00100_00100_0000000000000000
6 andi $10 $10 0    # contor => 100101_01010_01010_0000000000000000
7

```

```

8 # salvare variabile
9 lw $reg1, 0($1)    # adresa A
10                      # => 100011_00001_00001_0000000000000000
11 lw $reg2, 1($2)    # numarul de elemente N
12                      # => 100011_00010_00010_0000000000000001
13 lw $reg3, 2($3)    # valoarea de referinta X
14                      # => 100011_00011_00011_0000000000000010
15 lw $reg4, 3($4)    # valoarea de referinta Y
16                      # => 100011_00100_00100_0000000000000011
17
18 loop:
19 lw $reg5, 0($1)    # reg5 parcurge fiecare element din memorie
20                      # => 100011_00001_00101_0000000000000000
21 andi $reg6 $6 0    # ok => 100101_00110_00110_0000000000000000
22 slt $reg6, $reg5, $reg3 # reg5 < reg3(X) -> reg6=1
23                      # => 000000_00101_00011_00110_00000_000111
24
25 andi $reg7 $7 0    # => 100101_00111_00111_0000000000000000
26 addi $reg7 $7 1    # am pus in reg7 1
27                      # => 100011_00111_00111_0000000000000001
28
29 bne $reg6 $reg7 3  # (et1) daca ok!=1 sare la adresa et1
30                      # => 100100_00110_00111_0000000000000011
31 srl $reg5 $reg5, 3 # altfel sfiftare la dreapta 3 biti, /8
32                      # => 000000_00000_00101_00101_00011_000011
33 sw $reg5, 0($1)    # pun rezultatul final
34                      # => 100001_00001_00101_0000000000000000
35 j 29 (comp_N)      # => 111111_000000000000000000000011101
36
37 et1:                  # elementul > X
38
39 # Pentru Y
40 andi $reg6 $6 0    # ok => 100101_00110_00110_0000000000000000
41 slt $reg6, $reg5, $reg4 # reg5 < reg4(Y) -> reg6=1
42                      # => 000000_00101_00100_00110_00000_000111
43
44 andi $reg7 $7 0    # => 100101_00111_00111_0000000000000000
45 addi $reg7 $7 1    # am pus in reg7 1
46                      # => 100011_00111_00111_0000000000000001
47
48 bne $reg6 $reg7 3 (et2) # daca ok!=1 sare la adresa et2
49                      # => 100100_00110_00111_0000000000000011
50 sll $reg5 $reg5, 1    # altfel sfiftare la stanga 1 bit, *2
51                      # => 000000_00000_00101_00101_00001_000010
52 sw $reg5, 0($1)    # pun rezultatul final
53                      # => 100001_00001_00101_0000000000000000
54 j 29 (comp_N)      # => 111111_000000000000000000000011101
55
56

```

```

57 et2:                      # elementul > Y
58 andi $reg5 $reg5 0          # => 100101_00101_00101_0000000000000000
59 addi $reg5 $reg5 1          # => 100011_00101_00101_0000000000000001
60 sw $reg5, 0($1)             # => 100001_00001_00101_0000000000000000
61
62
63 comp_N:
64 addi $reg10 $10 1    # => 100011_00111_00111_0000000000000001
65 beq $reg2 $reg10 2 (afisare) # am parcurs tot
66                                # => 100010_00010_01010_0000000000000001
67 addi $reg1 $reg1 1    # => 100011_00001_00001_0000000000000001
68 j 9 (loop)    # se intoarce din nou in loop
69                                # => 111111_000000000000000000000001001
70
71 afisare:
72
73 andi $1 $1 0              # => 100101_00001_00001_0000000000000000
74 andi $10 $10 0             # => 100101_01010_01010_0000000000000000
75 lw $reg1, 0($1)            # => 100000_00001_00001_0000000000000000
76
77 loop_afisare:
78 lw $reg5, 0($1)             # => 100000_00001_00101_0000000000000000
79 addi $reg10 $10 1            # => 100011_01010_01010_0000000000000001
80 beq $reg2 $reg10 3 (final)   # am parcurs tot
81                                # => 100010_00010_01010_0000000000000001
82 addi $reg1 $reg1 1    # incrementare
83                                # => 100011_00111_00111_0000000000000001
84 j 36 (loop_afisare)        # => 111111_00000000000000000000000100100
85 final:

```

Explicație implementare program pentru procesorul MIPS:

Pentru început inițializez registrii pe care doresc să îl utilizez în program pentru a realiza operațiile necesare și inclusiv registrul pe care îl consider contor pentru "for-ul" care parcurge elementele pe care le verific, de la anumite adrese (bucla loop). Apoi salvez în variabilele reg1, reg2, reg3 și reg4, adresa A de la care se pornește programul din memorie, numărul N de elemente, X, Y, iar apoi în bucla loop, cu reg5 parcurg fiecare element din memorie. Verific cu ajutorul instrucțiunii BNE (se creează "if-uri" pe baza acestei instrucțiuni care are rolul de a verifica ce avem nevoie). Verific dacă numărul parcurs din memorie este mai mic decât X și în caz afirmativ îl împart la 8, prin shiftarea la dreapta cu 3 biți, utilizând instrucțiune SRL. Altfel verific dacă este mai mic ca Y. În caz afirmativ înlocuiesc numărul cu dublul său. Acest lucru se realizează utilizând instrucțiunea SLL, shiftez numărul la stânga cu un bit. În cazul în care numărul din memorie este mai mare decât Y, îl înlocuiesc cu 1, asignându-i variabilei reg5 valoarea 1 cu ajutorul instrucțiunii ADDI. Se revine la loop prin strucțiunea de salt J urmată de adresa 9, adresă care reprezintă numărul de instrucțiuni începând de la 0 până la instrucțiunea de start din bucla respectivă. Apoi am implementat o afișare pentru verificare care funcționează pe același principiu cu bucle pentru parcugerea șirului de numere, reg1 fiind contorul pe care îl incrementez cu ajutorul instrucțiunii ADDI. Odată cu ieșirea parcurgerii buclei de afișare programul a luat sfârșit.

Observație: Codurile binare care sunt comentate sub linia de cod în Assembly reprezintă codul instrucțiunii respective, doar că din motivul încadrării în pagină am ales să îl pun la rândul următor. Etichetele au fost înlocuite cu adresele respective, pentru a putea scrie programul în limbaj VHDL, cu biți de 0 și 1. Codul de mai sus translatat în dreapta în biți a fost scris în Unitatea de extragere a instrucțiunilor IFetch în memorie.

Pentru o vizualizare mai bună mai jos este codul C după care s-a realizat implementarea în limbajul Assembly cu instrucțiuniile alese pentru procesor.

```

1 #include <stdio.h>
2 int main() {
3     int A[] = {4, 5, 3, 20, 1, 13, 24};
4     //4 -> Adresa
5     //5 -> Numarul de elemente din sir
6     //3 -> Numarul X
7     //20 > Numarul Y
8     // {1, 13, 24} exemplu de sir de numere
9     int N = A[1] // numarul de elemente din sir
10    int X = A[2] // valoarea de referinta X
11    int Y = A[3] // valoarea de referinta Y
12
13
14    for (int i = A[0]; i < N+A[0]; i++) {
15        if (A[i] < X) {
16            A[i] /= 8; // inlocuim elementul cu impartirea lui la 8
17        } else if (A[i] >= X && A[i] <= Y){
18            A[i] *= 2; // inlocuim elementul cu jumatarea sa
19        }
20    }
21
22 }
23
24
25 printf("Noul sir este: ");
26 for (int i = A[0]; i < N + A[0]; i++) {
27     printf("%d ", A[i]);
28 }
29 printf("\n");
30
31 return 0;
32 }
```

5 Implementare în VHDL

Execuția unei instrucțiuni are următoarele 5 etape:

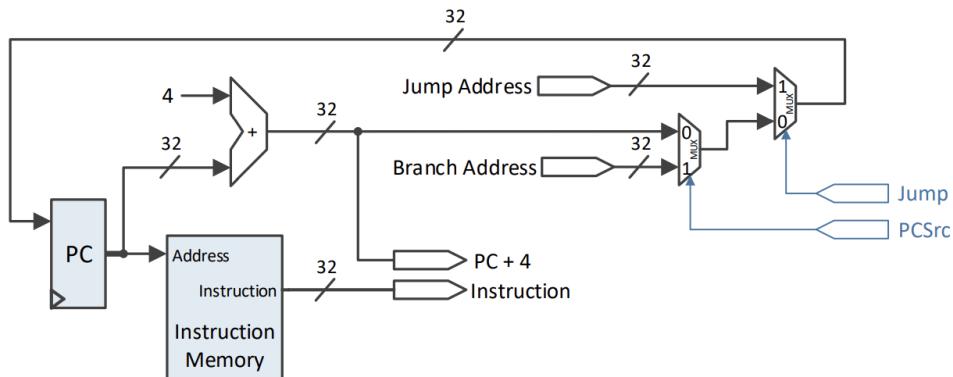
1. IF – Extragerea instrucțiunii (Instruction Fetch);
2. ID/OF – Decodificarea instrucțiunii / extragerea operanzilor (Instruction Decode / Operand Fetch);

3. UC – Unitate de control (Unit Control);
4. EX – Execuție (Execute);
5. MEM – Memorie (Memory);
6. WB – Scriere rezultat (Write-Back).

5.1 Unitatea de extragere a instrucțiunilor IFetch

Unitatea de extragere a instrucțiunilor IFetch 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.

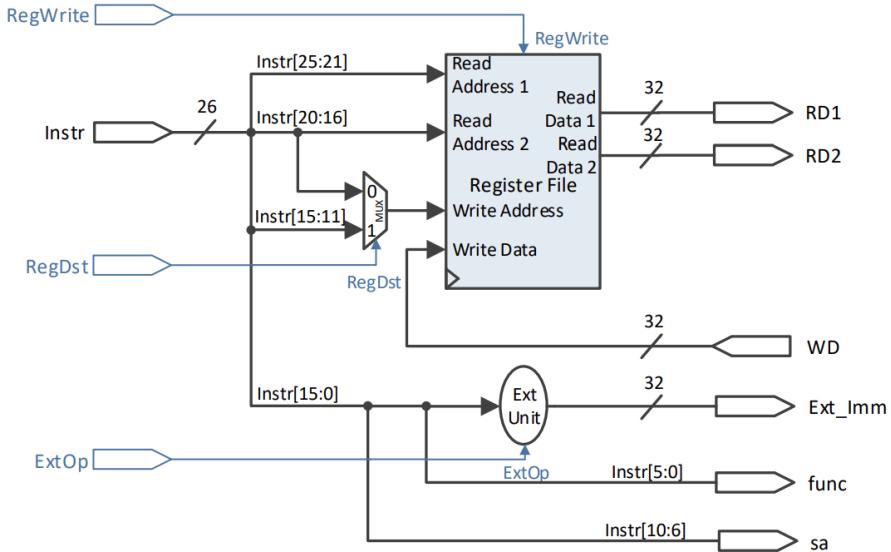


Unitatea IFetch primește pe intrările de date adresele de salt și pune la dispoziție, pe ieșiri, adresa imediat următoare ($PC + 4$), respectiv conținutul instrucțiunii curente. Adresele pot să fie de salt condiționat (branch) sau necondiționat (jump).

5.2 Unitatea de decodificare a instrucțiunilor ID

Unitatea de decodificare a instrucțiunilor ID realizează extragerea operanzilor și conține următoarele elemente:

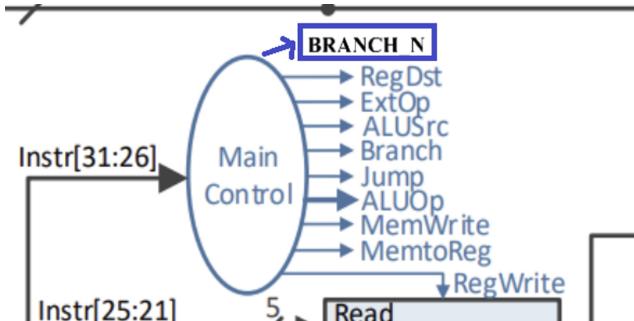
- Register File (RF) – bloc de 32 registre pe 32 de biți
- Multiplexor MUX 2:1 – stabilește adresa de scriere în RF;
- Unitate de extindere (Ext Unit) – extinde valoarea câmpului immediate la 32 de biți (immediatul extins).



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.

5.3 Unitatea de Control UC

Unitatea de Control UC generează semnalele care determină funcționalitatea unităților din calea de date. 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. În plus în Main Control am adăugat semnalul de Branch_N pentru instrucțiunea BNE.



Main Control actualizat cu semnalul Branch_N

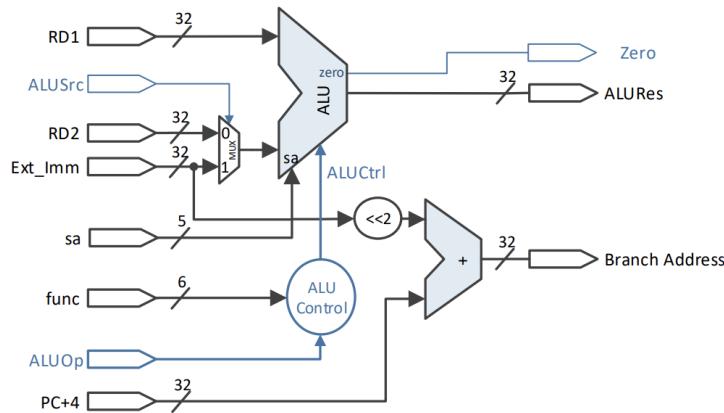
5.4 Unitatea de execuție EX

Unitatea de execuție EX realizează operațiile aritmetice și logice necesare instrucțiunii. Are în componență următoarele elemente:

- Unitatea Aritmetică-Logică

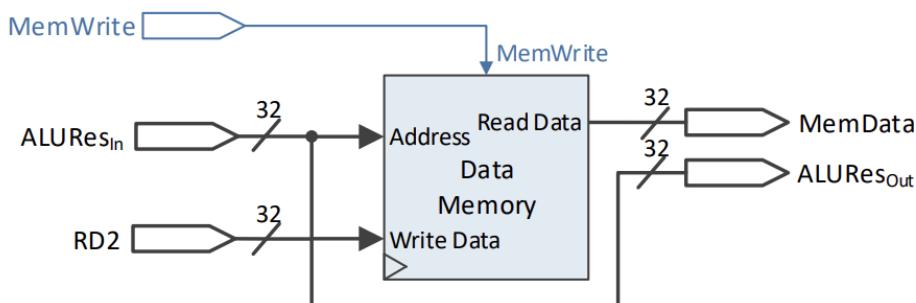
- Unitatea de Control pentru ALU (ALU Control) – generează codul operației pentru ALU;
- Multiplexor MUX 2:1 – stabilește sursa celui de-al 2-lea operand pentru ALU, între Read Data 2 (RD2) și imediatul extins (Ext_Imm);
- Unitatea de deplasare la stânga cu 2 poziții a imediatului extins și sumatorul 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.



5.5 Unitatea de memorie MEM

Unitatea de memorie MEM 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.



```

entity MemoryUnit is
  Port
  (
    clk: in std_logic;
    enable: in std_logic;
    MemWrite: in std_logic;
    ALURes_in: in std_logic_vector(31 downto 0);
    RD2: in std_logic_vector(31 downto 0);
    MemData: out std_logic_vector(31 downto 0);
    AluRes_out: out std_logic_vector(31 downto 0)
  );
end MemoryUnit;

architecture Behavioral of MemoryUnit is
type RAM is array (0 to 63) of std_logic_vector(31 downto 0);
signal ram_memory : RAM := (
  B"00000000_00000000_00000100", --A = adresa : 4 =>HEXA: 4 --ADR 0
  B"00000000_00000000_00000011", --N = numarul de elemente : 3 =>HEXA: 5 --ADR 1
  B"00000000_00000000_00000011", --X : 3 =>HEXA: 3 --ADR 2
  B"00000000_00000000_00010100", --Y : 20 =>HEXA: 14 --ADR3
  B"00000000_00000000_00000001", --I =>HEXA: 1 --ADR4
  B"00000000_00000000_00001101", --13 =>HEXA: D --ADR 5
  B"00000000_00000000_00001100", --24 =>HEXA: 18 --ADR 6
  others => X"00000000");
begin

  AluRes_out <= ALURes_in;
  MemData <= ram_memory(conv_integer(ALURes_in));

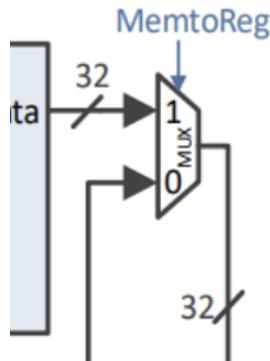
  process(clk)
  begin
    if clk'event and clk = '1' then
      if enable = '1' then
        if MemWrite = '1' then
          ram_memory(conv_integer(ALURes_in)) <= rd2;
        end if;
      end if;
    end if;
  end process;
end Behavioral;

```

Implementarea Memoriei în limbaj VHDL

5.6 Unitatea de scriere a rezultatului WB

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



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

```

entity WBUnit is
Port
(
    MemToReg: in std_logic;
    MemData: in std_logic_vector(31 downto 0);
    ALURes_out: in std_logic_vector(31 downto 0);
    WD: out std_logic_vector(31 downto 0)
);
end WBUnit;

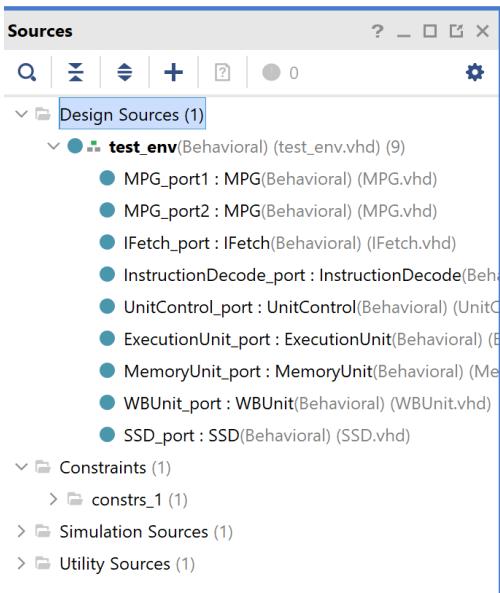
architecture Behavioral of WBUnit is
begin
    WD <= ALURes_out when MemToReg = '0' else MemData;
end Behavioral;

```

Implementarea WB în limbaj VHDL

Componentele procesorului MIPS sunt:

În ceea ce ține de implementare am ales să realizez fiecare componentă treptat, urmând a le lega pentru testare în fișierul test.env.



Organizarea componentelor MIPS

Codul binar pentru programul pe care îl realizează procesorul s-a scris în memoria din *IFetch*, având în dreapta comentarii cu instrucțiunea pe care o execută, respectiv cu codificarea în HEXA, pentru a putea fi mai ușor de realizat testarea în cele din urmă. Pe placuța de la laborator, valorile se afișează în HEXA pentru fiecare dintre semnalele pe care dorim să le vizualizăm. Adresele de unde se citesc datele, A, numărul de elemente, X, Y și elementele pentru care dorim realizarea programului de către MIPS au fost scrise în memoria *MemoryUnit*. Precum se poate observa mai jos:

D:/Facultate/Anul 2 - Sem 2/Arhitectura Calculatoarelor/Procesor MIPS/Mips/Mips.srsc/sources_1/new/IFetch.vhd

```

132      B"100101_00010_00010_0000000000000000", --andi $2 $2 0    =>HEXA: 94420000
133      B"100101_00011_00011_0000000000000000", --andi $3 $3 0    =>HEXA: 94630000
134      B"100101_00100_00100_0000000000000000", --andi $4 $4 0    =>HEXA: 94840000
135      B"100101_01010_01010_0000000000000000", --andi $10 $10 0   =>HEXA: 954A0000
136      B"100000_00001_00001_0000000000000000", --lw $reg1, 0($1)  =>HEXA: 80210000
137      B"100000_00010_00010_0000000000000001", --lw $reg2, 1($2)  =>HEXA: 80420001
138      B"000000_00011_00011_0000000000000010", --lw $reg3, 2($3)  =>HEXA: 80630002
139      B"100000_00100_00100_0000000000000011", --lw $reg4, 3($4)  =>HEXA: 80840003
140      B"100000_00001_00010_0000000000000000", --loop: lw $reg5, 0($1) =>HEXA: 80250000
141      B"100101_00110_00110_0000000000000000", --andi $reg6 $6 0   =>HEXA: 94C60000
142      B"000000_00101_00011_00110_00000_000111", --slt $reg6, $reg5, $reg3 =>HEXA: 00A33007
143      B"100101_00111_00111_0000000000000000", --andi $reg7 $7 0   =>HEXA: 94E70000
144      B"100001_00111_00111_0000000000000001", --addi $reg7 $7 1   =>HEXA: 8CE70001
145      B"100100_00110_00111_0000000000000011", --bne $reg6 $reg7 3 =>HEXA: 90C70003
146      B"000000_00000_00101_00101_00011_000011", --srl $reg5 $reg5, 3 =>HEXA: 000528C3
147      B"100001_00001_00101_0000000000000000", --sw $reg5, 0($1)  =>HEXA: 84250000
148      B"111111_00000000000000000000000000000000", --j 29 (comp_N)  =>HEXA: FC00001D
149      B"100101_00110_00110_0000000000000000", --andi $reg6 $6 0   =>HEXA: 94C60000
150      B"000000_00101_00100_00110_00000_000111", --slt $reg6, $reg5, $reg4 =>HEXA: 00A43007
151      B"100101_00111_00111_0000000000000000", --andi $reg7 $7 0   =>HEXA: 94E70000
152      B"100001_00111_00111_0000000000000001", --addi $reg7 $7 1   =>HEXA: 8CE70001
153      B"100100_00110_00111_0000000000000011", --bne $reg6 $reg7 3 =>HEXA: 90C70003
154      B"000000_00000_00101_00101_00001_000010", --sll $reg5 $reg5, 1 =>HEXA: 00052842
155      B"100001_00001_00101_0000000000000000", --sw $reg5, 0($1)  =>HEXA: 84250000
156      B"111111_00000000000000000000000000000000", --j 29   =>HEXA: FC00001D
157      B"100101_00101_00101_0000000000000000", --andi $reg5 $reg5 0 =>HEXA: 94A50000
158      B"100011_00101_00101_0000000000000001", --addi $reg5 $reg5 1 =>HEXA: 8CA50001
159      B"100001_00001_00101_0000000000000000", --sw $reg5, 0($1)  =>HEXA: 84250000
160      B"100001_01010_01010_0000000000000001", --comp N: addi $reg10 $10 1 =>HEXA: 8D4A0001
161      B"100001_00010_01010_0000000000000010", --beq $reg2 $reg10 2 =>HEXA: 884A0002
162      B"100001_00001_00001_0000000000000001", --addi $reg1 $reg1 1 =>HEXA: 8C210001
163      B"111111_00000000000000000000000000000000", --j 9 (loop)  =>HEXA: FC000009
164      B"100101_00001_00001_0000000000000000", --afisare: andi $1 $1 0 =>HEXA: 94210000
165      B"100101_01010_01010_0000000000000000", --andi $10 $10 0   =>HEXA: 954A0000
166      B"100000_00001_00001_0000000000000000", --lw $reg1, 0($1)  =>HEXA: 80210000
167      B"000000_00001_00010_0000000000000000", --loop_afisare: lw $reg5, 0($1)  =>HEXA: 80250000
168      B"100011_01010_01010_0000000000000001", --addi $reg10 $10 1 =>HEXA: 8D4A0001
169      B"100001_00010_01010_0000000000000011", --beq $reg2 $reg10 3 =>HEXA: 884A0003
170      B"100011_00001_00001_0000000000000001", --addi $reg1 $reg1 1 =>HEXA: 8C210001
171      B"111111_00000000000000000000000000000000", --j 36 (loop_afisare) =>HEXA: FC000024
172      others => (others => '0')
173

```

Programul traslatat în biți pentru procesorul MIPS

```

entity MemoryUnit is
  Port
  (
    clk: in std_logic;
    enable: in std_logic;
    MemWrite: in std_logic;
    ALURes_in: in std_logic_vector(31 downto 0);
    RD2: in std_logic_vector(31 downto 0);
    MemData: out std_logic_vector(31 downto 0);
    AluRes_out: out std_logic_vector(31 downto 0)
  );
end MemoryUnit;

architecture Behavioral of MemoryUnit is

type RAM is array (0 to 63) of std_logic_vector(31 downto 0);
signal ram_memory : RAM := (
  B"00000000_00000000_00000000_00000100", --A = adresa : 4 =>HEXA: 4 --ADR 0
  B"00000000_00000000_00000000_00000111", --N = numarul de elemente : 3 =>HEXA: 5 --ADR 1
  B"00000000_00000000_00000000_00000011", --X : 3 =>HEXA: 3 --ADR 2
  B"00000000_00000000_00000000_00010100", --Y : 20 =>HEXA: 14 --ADR3
  B"00000000_00000000_00000000_00000001", --I =>HEXA: 1 --ADR4
  B"00000000_00000000_00000000_00001101", --I3 =>HEXA: D --ADR 5
  B"00000000_00000000_00000000_00011000", --I24 =>HEXA: 18 --ADR 6
  others => X"00000000");

```

Datele din memorie pentru realizarea programului pe acestea

6 Testare

Am ales leduri pentru toate semnalele semnificative ale procesorului MIPS, leduri pe care le-am initializat în fișierul test_env.

```
selection <= sw(2)&sw(1)&sw(0);
led(1 downto 0) <= ALUOp;
led(2) <= RegDst;
led(3) <= EXTOp;
led(4) <= ALUSrc;
led(5) <= MemWrite;
led(6) <= MemToReg;
led(7) <= RegWrite;
led(8) <= Branch;
led(9) <= Branch_N;
led(10) <= PCSrc;
led(11) <= Jump;
led(15 downto 12) <= "0000";
```

Am ales două leduri pentru ALUOp dat fiind faptul că este un semnal pe 2 biți care are diverse valori în funcție de operația pe care o execută instrucțiunea (Am notat în tabelul cu instrucțiuni acest lucru). Restul ledurilor vor rămâne neaprinse, pentru că nu sunt utilizate, fiind astfel inițializate cu 0.

S-au utilizat butoane pentru enable și reset. Enable reprezintă butonul din mijloc aşa cum am ales să testez, reset este butonul deasupra de enable. Trecerea de la o instrucțiune la alta are loc prin apăsarea butonului reprezentativ pentru enable. Alegerea butoanelor s-a declarat în cod în următoarele linii:

```
MPG_port1: MPG port map(btn => btn(0), enable=>enable, clk => clk);
MPG_port2: MPG port map(enable => reset, btn => btn(1), clk => clk);
```

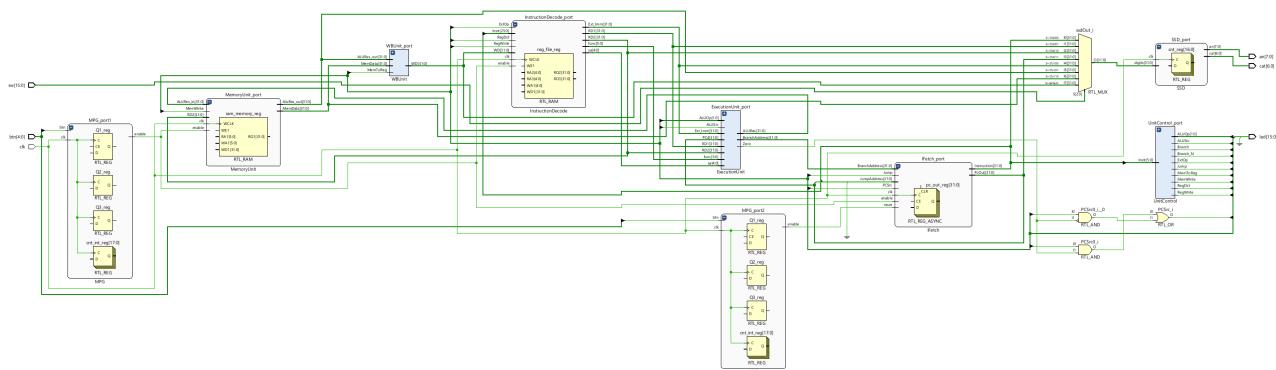
Pentru a putea observa mai eficient dacă cumva procesorul nu se execuță cum ar trebui, este necesară trasarea execuției programului pentru a face un Debug mai ușor. Astfel la fiecare instrucțiune se poate testa dacă ceea ce arată procesorul corespunde cu ceea ce ar trebui să facă. Pentru aceasta am scris într-un tabel pentru fiecare instrucțiune ce valori ar trebui să aibă principalele semnale din MIPS.

6.1 Tabelul cu valorile pentru principalele semnale din MIPS (Tabel de trasare):

Pas	SW(7:5)	"000"	"001"	"010"	"011"	"100"	"101"	"110"	"111"	De completat numai pentru instrucțiuni de salt	
	Instr (în asamblare)	Instr (hexa)	PC+4	RD1	RD2	Ext_Imm	ALURes	MemData	WD	BranchAddr	JumpAddr
0	ANDI	X"94210000"	X"00000004"	X"00000000"	X"00000000"	X"00000000"	X"00000000"	X"00000004"	X"00000000"	X""	X""
1	ANDI	X"94420000"	X"00000008"	X"00000000"	X"00000000"	X"00000000"	X"00000000"	X"00000004"	X"00000000"		
2	ANDI	X"94630000"	X"0000000C"	X"00000000"	X"00000000"	X"00000000"	X"00000000"	X"00000004"	X"00000000"		
3	ANDI	94840000	00000010	00000000	00000000	00000000	00000000	00000004	00000000		
4	ANDI	954A0000	00000014	00000000	00000000	00000000	00000000	00000004	00000000		
5	LW	80210000	00000018	00000000	00000000	00000000	00000000	00000004	00000004		
6	LW	80420001	0000001C	00000000	00000000	00000001	00000001	00000003	00000003		
7	LW	80630002	00000020	00000000	00000000	00000002	00000002	00000003	00000003		
8	LW	80840003	00000024	00000000	00000000	00000003	00000003	00000014	00000014		
9	LW	80250000	00000028	00000004	00000000	00000000	00000004	00000001	00000001		
10	ANDI	94C60000	0000002C	00000000	00000000	00000000	00000000	00000004	00000000		
11	SLT	00A33007	00000030	00000001	00000003	00003007	00000001	00000003	00000001		
12	ANDI	94E70000	00000034	00000000	00000000	00000000	00000000	00000004	00000000		
13	ADDI	8CE70001	00000038	00000000	00000000	00000001	00000001	00000003	00000001		
14	BNE	90C70003	0000003C	00000001	00000001	00000003	00000000	00000004	00000000	00000048	
15	SRL	000528C3	00000040	00000000	00000001	000028C3	00000000	00000004	00000000		
16	SW	84250000	00000044	00000004	00000000	00000000	00000004	00000001	00000004		
17	J	FC00001D	00000048	00000000	00000000	00000001D	00000000	00000004	00000000		00000074
18	ADDI	8D4A0001	00000078	00000000	00000000	00000001	00000001	00000003	00000001		
19	BEQ	884A0002	0000007C	00000003	00000001	00000002	00000002	00000003	00000002	00000084	
20	ADDI	8C210001	00000080	00000004	00000004	00000001	00000005	0000000D	00000005		
21	J	FC000009	00000084	00000000	00000000	00000009	00000000	00000004	00000000		00000024
22	LW	80250000	00000028	00000005	00000000	00000000	00000005	0000000D	0000000D		
24	ANDI	94C60000	0000002C	00000001	00000001	00000000	00000000	00000004	00000000		
25	SLT	00A33007	00000030	0000000D	00000003	00003007	00000000	00000004	00000000		
26	ANDI	94E70000	00000034	00000001	00000000	00000000	00000004	00000000	00000000		
27	ADDI	8CE70001	00000038	00000000	00000000	00000001	00000001	00000003	00000001		
28	BNE	90C70003	0000003C	00000000	00000001	00000003	FFFFFFF	00000000	FFFFFFF	00000048	
29	ANDI	94C60000	0000004C	00000000	00000000	00000000	00000000	00000004	00000000		
30	SLT	00A43007	00000050	0000000D	00000014	00003007	00000001	00000003	00000001		
31	ANDI	94E70000	00000054	00000001	00000001	00000000	00000000	00000004	00000000		
32	ADDI	8CE70001	00000058	00000000	00000000	00000001	00000001	00000003	00000001		
33	BNE	90C70003	0000005C	00000001	00000001	00000003	00000000	00000004	00000000	00000060	
34	SLL	00052842	00000060	00000000	00000000	0002842	0000001A	00000000	0000001A		
35	SW	84250000	00000064	00000005	00000005	0000001A	00000000	00000005	0000000D	00000005	
36	J	FC00001D	00000068	00000000	00000000	00000001D	00000000	00000004	00000000		00000074
37	ADDI	8D4A0001	00000078	00000001	00000001	00000001	00000002	00000003	00000002		
38	BEQ	884A0002	0000007C	00000003	00000002	00000002	00000001	00000003	00000001	00000084	
39	ADDI	8C210001	00000080	00000005	00000005	00000001	00000006	000000018	00000006		
40	J	FC000009	00000084	00000000	00000000	00000009	00000000	00000004	00000000		00000024
41	LW	80250000	00000028	00000006	00000001A	00000000	00000006	000000018	000000018		
42	ANDI	94C60000	0000002C	00000001	00000001	00000000	00000000	00000004	00000000		
43	SLT	00A33007	00000030	00000018	00000003	00003007	00000000	00000004	00000000		
44	ANDI	94E70000	00000034	00000001	00000001	00000000	00000000	00000004	00000001		
45	ADDI	8CE70001	00000038	00000000	00000000	00000001	00000001	00000003	00000001		
46	BNE	90C70003	0000003C	00000000	00000001	00000003	FFFFFFF	00000000	FFFFFFF	00000048	
47	ANDI	94C60000	0000004C	00000000	00000000	00000000	00000000	00000004	00000000		
48	SLT	00A43007	00000050	00000018	00000014	00003007	00000000	00000004	00000000		
49	ANDI	94E70000	00000054	00000001	00000001	00000000	00000000	00000004	00000000		
50	ADDI	8CE70001	00000058	00000000	00000000	00000001	00000001	00000003	00000001		
51	BNE	90C70003	0000005C	00000000	00000001	00000003	FFFFFFF	00000000	FFFFFFF	00000060	
52	ANDI	94A50000	0000006C	00000018	00000018	00000000	00000000	00000004	00000000		
53	ADDI	8CA50001	00000070	00000000	00000000	00000001	00000001	00000003	00000001		
54	SW	84250000	00000074	00000006	00000001	00000000	00000006	000000018	00000006		
55	ADDI	8D4A0001	00000078	00000002	00000002	00000001	00000003	00000014	00000003		
56	BEQ	884A0002	0000007C	00000003	00000003	00000002	00000000	00000004	00000000	00000084	
57	ANDI	94210000	00000088	00000006	00000006	00000000	00000000	00000004	00000000		
58	ANDI	954A0000	0000008C	00000003	00000003	00000000	00000000	00000004	00000000		
59	LW	80210000	00000090	00000000	00000000	00000000	00000000	00000004	00000004		
60	LW	80250000	00000094	00000004	00000001	00000000	00000004	00000000	00000000		

61	ADDI	8D4A0001	00000098	00000000	00000000	00000001	00000001	00000003	00000001		
62	BEQ	884A0003	0000009C	00000003	00000001	00000003	00000002	00000003	00000002	000000A8	
63	ADDI	8C210001	000000A0	00000004	00000004	00000001	00000005	0000001A	00000005		
64	J	FC000024	000000A4	00000000	00000000	00000024	00000000	00000004	00000000		00000090
65	LW	80250000	00000094	00000005	00000000	00000005	00000005	0000001A	0000001A		
66	ADDI	8D4A0001	00000098	00000001	00000001	00000001	00000002	00000003	00000002		
67	BEQ	884A0003	0000009C	00000003	00000002	00000003	00000001	00000003	00000001	000000A8	
68	ADDI	8C210001	000000A0	00000005	00000005	00000001	00000006	00000001	00000006		
69	J	FC000024	000000A4	00000000	00000000	00000024	00000000	00000004	00000000		00000090
70	LW	80250000	00000094	00000006	0000001A	00000000	00000006	00000001	00000001		
71	ADDI	8D4A0001	00000098	00000002	00000002	00000001	00000003	00000014	00000003		
72	BEQ	884A0003	0000009C	00000003	00000003	00000003	00000000	00000004	00000000	000000A8	
73	0	0	0	0	0	0	0	0	0		
74											

7 RTL Schematic



8 MIPS 32 BIȚI PIPELINE - ACTUALIZARE MIPS SINGLE CICLE

La fel ca în cazul Mips Ciclu Unic, am adăugat un semnal de branch suplimentar pentru instrucțiune BNE, Branch_N, care va trece prin trei registrii pentru Pipeline. Alte modificări față de Pipeline-ul normal, prezentat la laborator nu au fost făcute, s-a respectat Arhitectura pentru 32 de biți. Totodată am modificat scrierea în Register File pe front descrescător, din front crescător cum era înainte, pentru a rezolva hazardurile structurale.

Pentru rezolvarea hazardurilor am adăugat NoOp-uri acolo unde a fost cazul, având un număr minim de NoOp-uri. Au fost prezente hazarduri de date și hazarduri de control(în cazul instrucțiunilor de Jump și de Branch). Am ales instrucțiunea SLL \$0 \$0 \$0, care nu modifică programul în niciun fel. Programul inițial nu ar putea funcționa pentru Pipeline din cauza hazardurilor prezente. Ex. de hazard de date - lw \$reg5, 0(\$1) și slt \$reg6, \$reg5, \$reg3, respectiv ex. de hazard de control - j 29, după care se adaugă un NoOp. Prin adăugarea de NoOp-uri pentru un anumit hazard, s-au rezolvat posibile alte hazarduri,

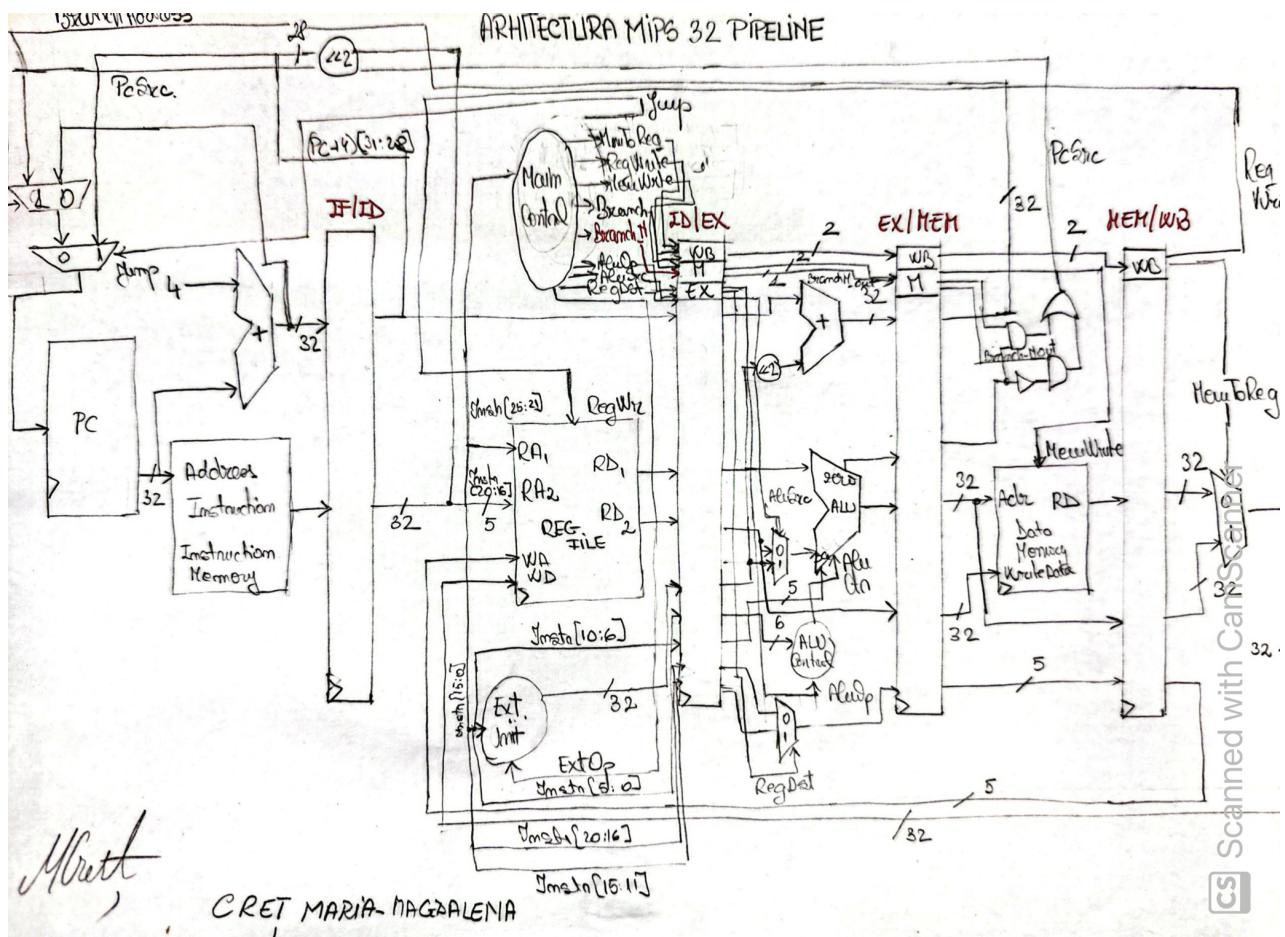
astfel că am verificat aceste cazuri, de fiecare dată când am adăugat un alt NoOp. Nu am interschimbat instrucțiunea de Jump J cu instrucțiunea următoare, pentru că fie se sărea din alte instrucțiuni la instrucțiunea fix următoare, fie era hazard cu această instrucțiune. Am respectat astfel ordinea instrucțiunilor de la Mips Ciclu Simplu, care realizează programul cu cerința:

Să se înlocuiască toate elementele dintr-un sir cu împărțirea lor la 8 ca întreg, dacă sunt mai mici decât X, dacă sunt între X și Y cu dublul lor și dacă sunt mai mari ca Y se vor înlocui cu 1. Sirul se află în memorie începând cu adresa A (A ≥ 4) și are N elemente. A, N, X, Y se citesc din memorie de la adresele 0, 1, 2, respectiv 3.

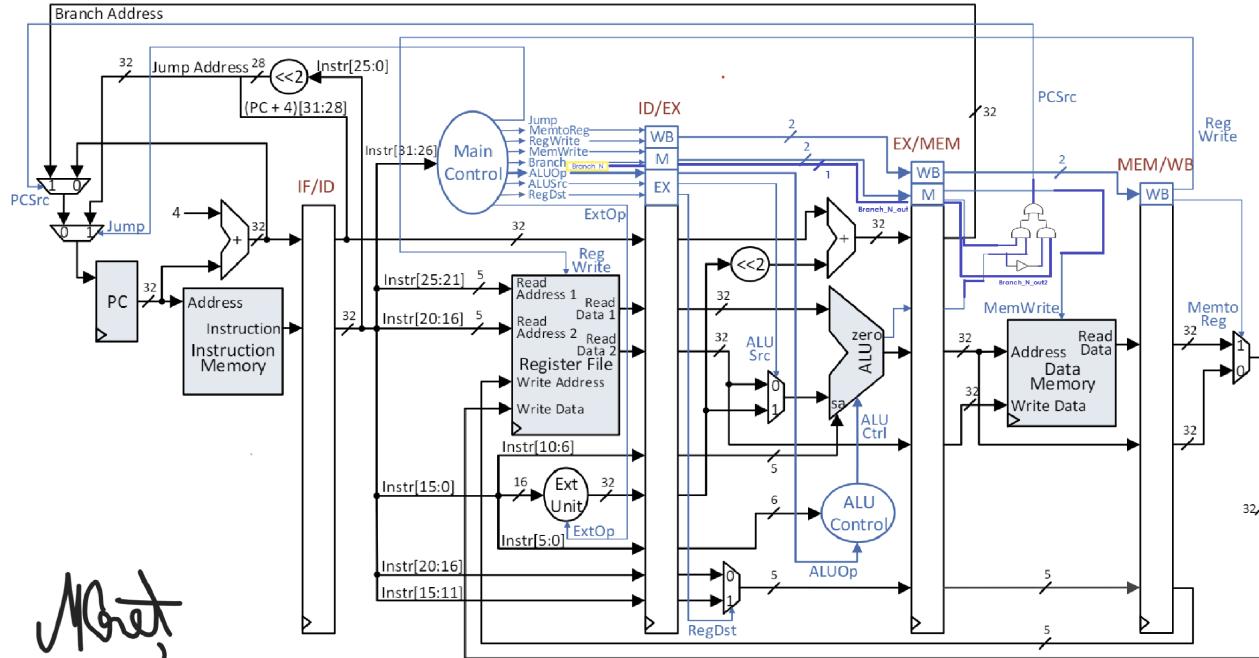
Constrângere: $X \leq Y$

Pentru verificare, se poate adăuga o buclă de citire a elementelor sirului, la final.

8.0.1 Arhitectura MIPS Pipeline-desen



8.0.2 Arhitectura MIPS Pipeline-actualizat (la fel desenat, dar cu vizibilitate mai bună)



CREȚ MARIA-MAGDALENA

8.0.3 Testare

S-a testat inițial pe o placă Basys3, pentru care s-a modificat SSD-ul, inclusiv și afișările (Am adăugat un switch pentru a vedea pentru fiecare instrucțiune parcursă, atât primii 16 biți (cei mai semnificativi), cât și ultimii 16 biți (cei mai puțini semnificativi), din punct de vedere Hexazecimal). Ulterior s-a testat și pe o placă Nexys7. Programul este funcțional, parcurge fiecare instrucțiune corespunzător cu tabelul de trasare de la Mips Ciclu Simplu. $Pc+4$ are loc la același clock ca și instrucțiunea.

Diferențele față de Mips Single Cicle sunt următoarele (*astfel că valorile nu corespund cu tabelul de trasare pentru că se întâmplă la clock-uri diferite, din cauza introducerii regiștrilor pentru Pipeline*):

- $Pc+4$ se afișează diferit pentru instrucțiunile de jump și branch, pentru că odată cu introducerea NoOp-urilor, a trebuit să se modifice pentru aceste instrucțiuni de salt și adresa la care săr, care nu corespunde cu cea anterioară de la Mips-ul Ciclu Simplu. Acest lucru se observă spre exemplu la instrucțiunea J, care va avea altă codificare pentru salt.
- Vor apărea la testare pe afișorul plăcuței și No-Op-uri, adică 0 peste tot pentru toate semnalele Mips-ului Pipeline. NoOp apare atunci când este adăugat după o anumită instrucțiune.
- RD1, respectiv RD2 se vor vedea pe placă cu un clock întârziere (pentru că trec prin primul regisztr de la Pipeline), deci pentru o anumită instrucțiune atunci când verifici RD1 și RD2, pe placă se va afișa RD1, respectiv RD2 pentru instrucțiunea dinainte de

instructiunea curentă.

- Ext_Imm se va vedea pe placă cu un clock întârziere(pentru că trece prin primul registru de la Pipeline), deci pentru o anumită instructiune atunci când verifici Ext_Imm, pe placă se va afisa Ext_Imm pentru instructiunea dinainte de instructiunea curentă.
- AluRes se va vedea pe placă cu două clock-uri(pentru că trece prin primele două registre de la Pipeline). Astfel că pentru o anumită instructiune curentă citită de pe placă, AluRes va fi rezultatul celei de-a doua instructiune în urmă.
- MemData se va vedea pe placă cu trei clock-uri(pentru că trece prin primele trei registre de la Pipeline). Astfel că pentru o anumită instructiune curentă citită de pe placă, MemData va fi rezultatul celei de-a treia instructiune în urmă.
- WriteData se va vedea pe placă cu patru clock-uri(pentru că trece prin toate cele 4 registre de la Pipeline). Astfel că pentru o anumită instructiune curentă citită de pe placă, WriteData va fi rezultat celei de-a patra instructiune în urmă.