University Politehnica Bucharest
Computers Faculty

CN Project : Verilog implementation of MIPS processor

Coordinator :Decebal Popescu, Teaching Assistant, Ph.D

Students :
Cocorada Sorin
Comanescu George
Manta Mihai
Parpauta Alexandru
Safta Octavian

SUMMARY

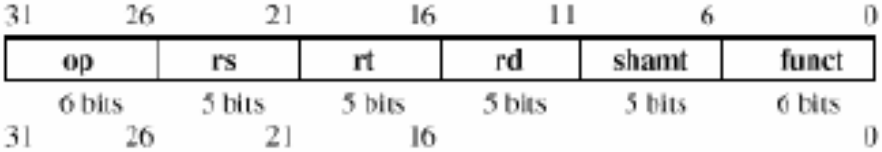
Project purpose: The implementation in verilog of a MIPS processor which can execute the following instructions: ADD and SUB;OR Immediate; LOAD and STORE Word ; BRANCH. The source is also presented with this documentation.

The projection of a processor that works in a single clock cycle.
The performances of the processor are imposed by:
-the number of instructions from the program execution (n);
- the clock rate(period) (T)
-the number of clock cycles for each instruction (CPI)
The projection of the processor (executing unit and command unit) shall determine
- the clock rate(period) (T)
- number of clock cycles for each instruction.
The projection of a processor that executes a single instruction within a clock cycle.
-Advantage: a single clock cycle for each instruction
-Desavantage: the duration of the clock cycle.
The projection phases:
1. Examination of the instructions set from the which result the expectations for the execution unit**:**
-the meaning of each instruction is given by the register transfer;
-the execution unit must include the memory elements;(the registers necessary to the instructions set;
-the execution unit must also assure each register transfer;
2.The selection of the execution unit components and the establishment of the clock sincronisation.
3. The assambly of the execution unit according to the specifications.
4. The analyze of the implementation for each instruction to determine signals and command points, which affect the implementation of all the transfers between the registers.
5.The assembly of the command logic

II.MIPS instructions format

All MIPS instructions are 32 bits instructions. There are 3 different instruction types:

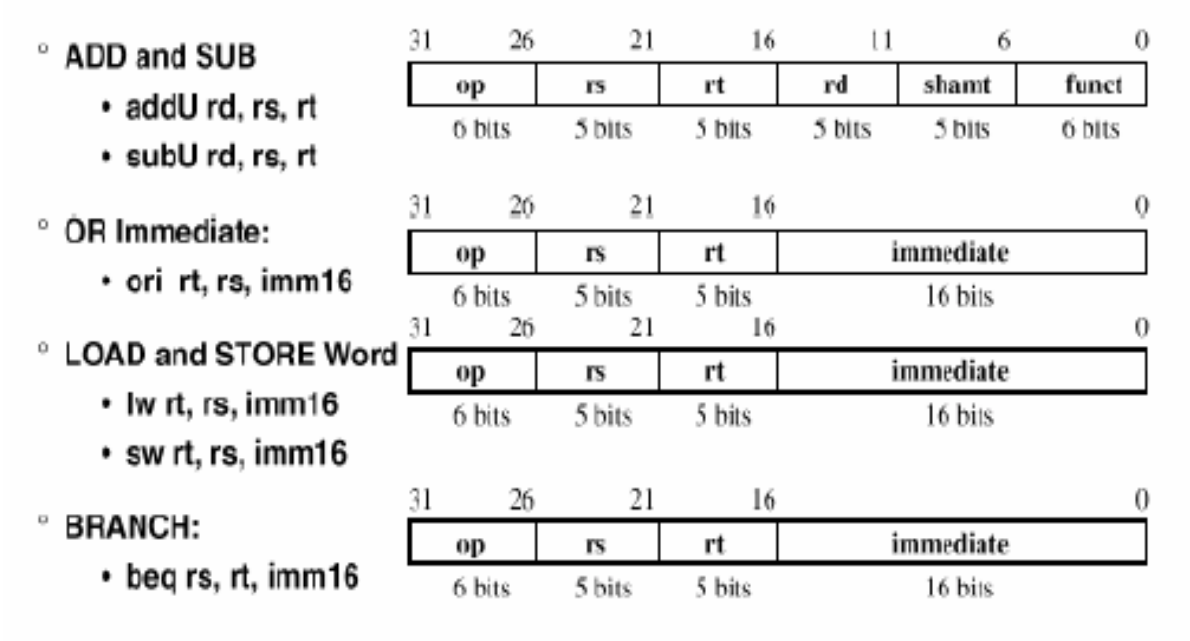
- **R-type**



- **I-type**

The operation codes are these:
-op the operation cod of the instruction
-rs, rt, rd the source and destination registers address
-shamt the quantity/number of bites with the whch the deplasament is made ;
- funct selects the operation specified by op ;
- address /immediately represents the deplassament oft the address(offset)value immediately;
- the target address the deplassament for the target address

The following set of instructions are presented here



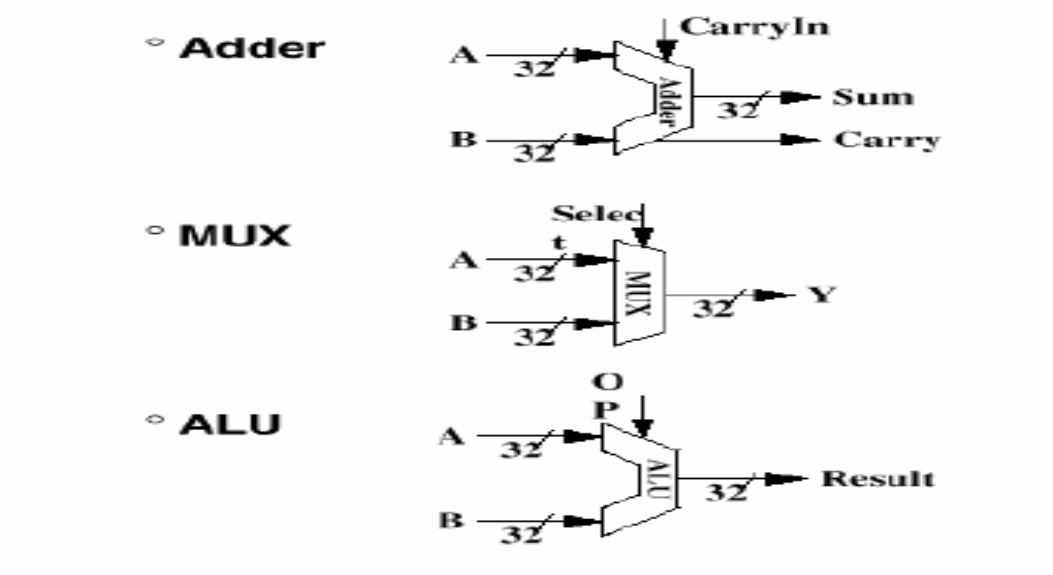
III.The logical transfers between registers (TLR)
- TLR specifies the semnification of instructions
- All begin with the reading of instructions

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

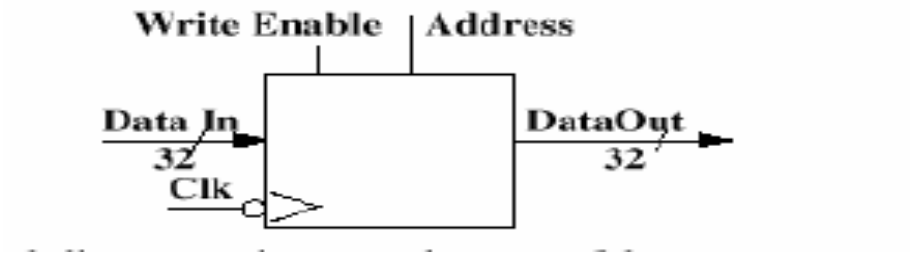
inst	Register Transfers	
ADDU	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
SUBU	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] ‖ 00 else PC <– PC + 4	

1**. The set of instructions:**
Memory :
- instructions and data
Registers:
- capacity 32 words x 32 bits
- read rs, rt;
- write rd, rt.
Counter program (PC)
Signal extender circuit (Extender)
Add/Remove register (offset-ul) extended
Add 4 or Immediately extended to PC
2. The components of the execution unit:
- Combinatory elements
- Memory elements
- Sincronisation methodology



The general registers :
There are 32 registers biport (out)
The register is selected by :
-RA (number) specifies the general registry which is placed on busA
-RB (number) specifies the general registry which is placed on busB
-RW (number) specifies the general registry which is placed on busW
-CLK in is effective only to the writing
The ideal memory :

- Data In
- Data Out

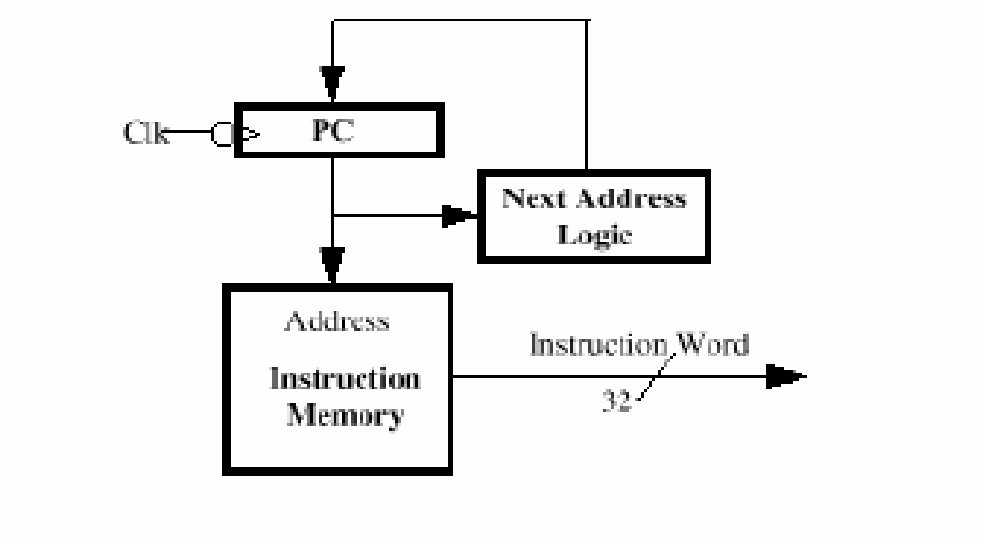


- The word in the meomory is selected using the following steps:
- Address selects the word that will be forced on DataOut
- Write Enable = 1is forced : the word from memory will be forced on DataOut.
- CLK In
- the CLK signal is used only on writing
- During the reading operations the comportament of the bloc is unitary

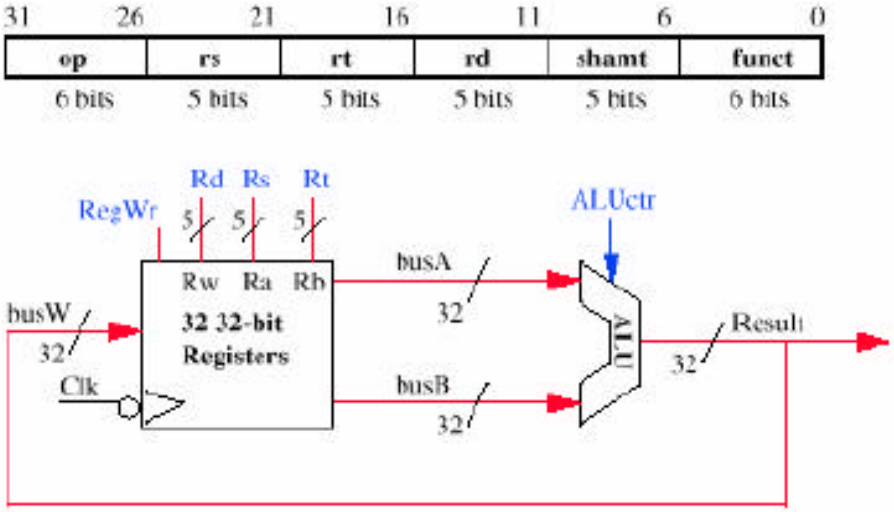
The sincronisation metodology (clocking)
All the memory elements are controlled by the same clock front
- the duration of the cycle equals CLK -> Q + Longest delay + Setup Time + Clock Skew
- CLK -> Q + shortest delay >Hold Time

3.The instructions read unit:
The RTL operations are:

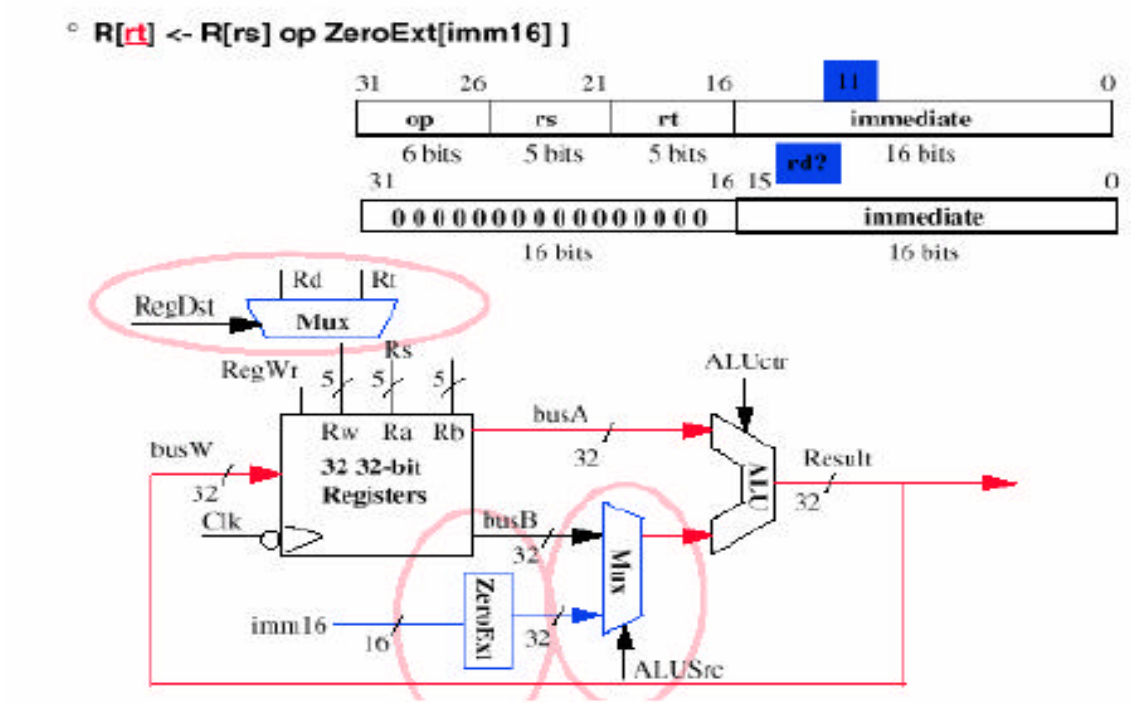
-read the instruction form mem[PC]
-the actualization of the PC counter(PC->PC+4)



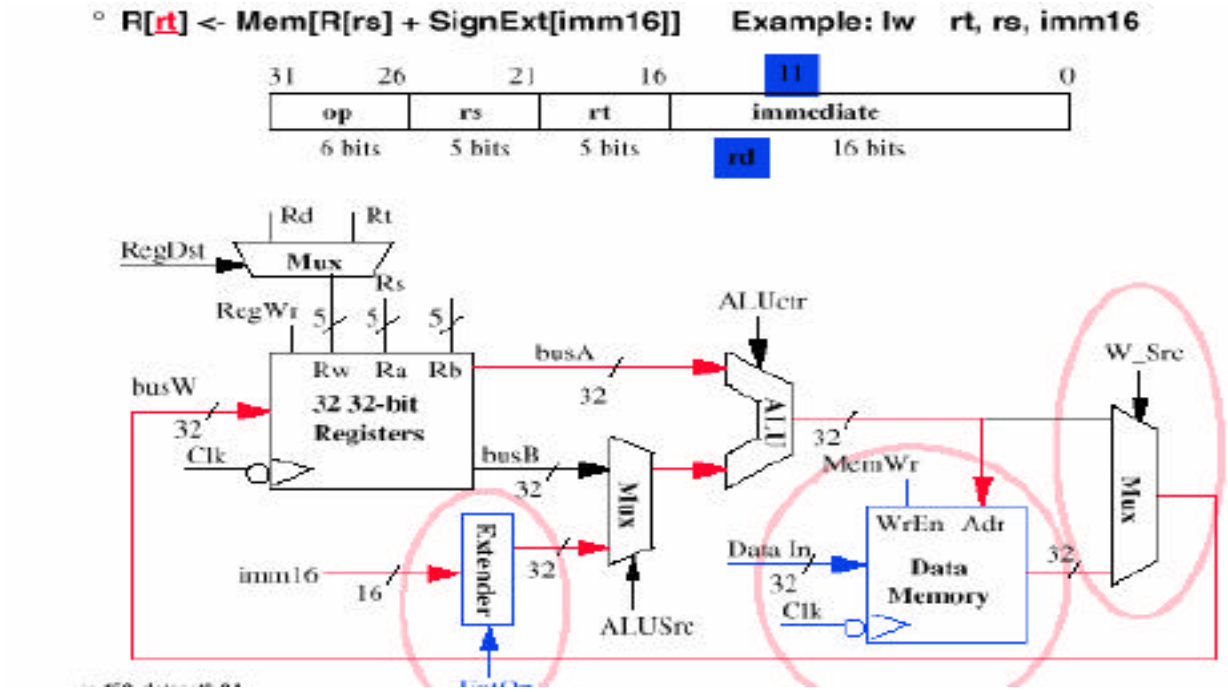
3. Add/ Substract
R[rd] <- R[rs] op R[rt]
- Ra. Rb si Rw
- ALUctr si RegWr: represent the control logic after the instruction decodification



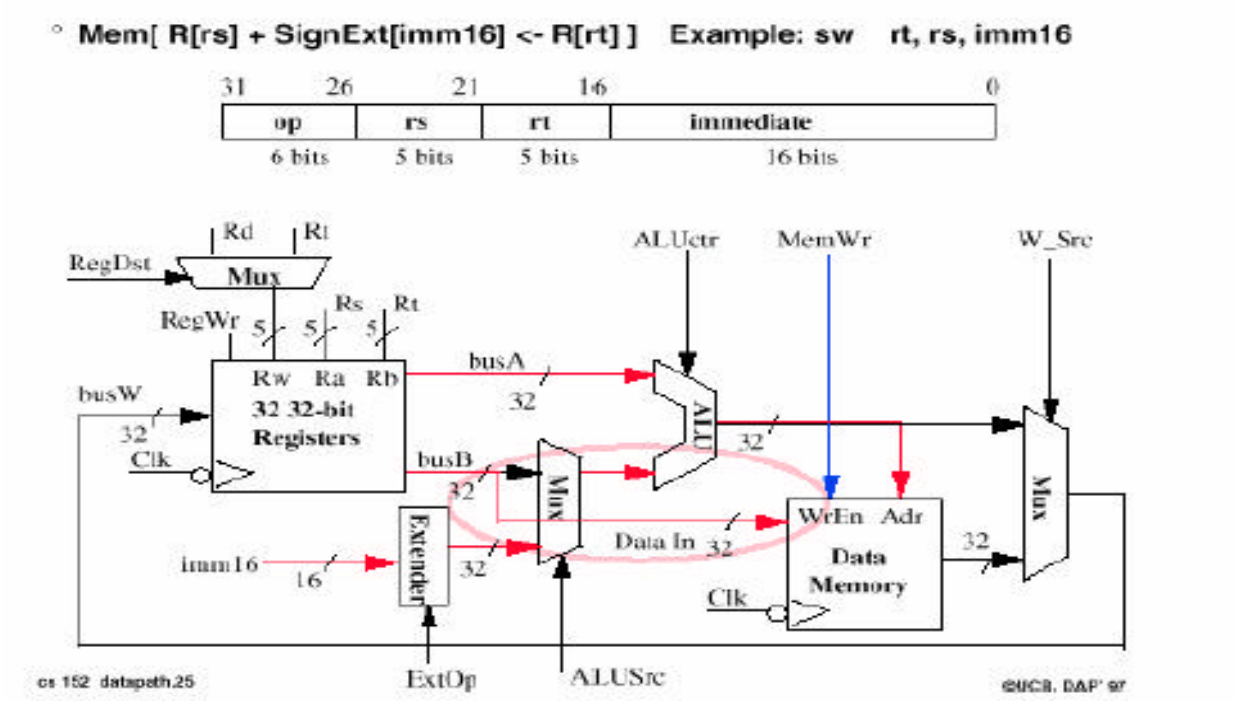
Logical operations with Immediate:



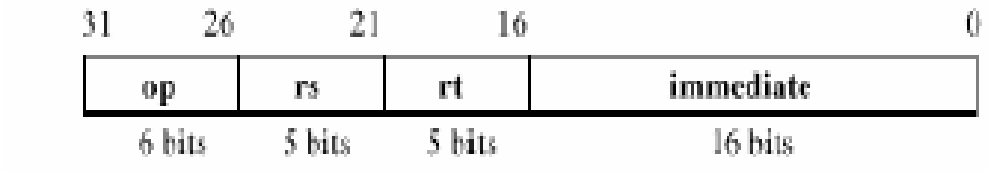
Load operations diagram



Store operations :



Branch operation:



beq rs, rt, imm16
- mem[PC] reads the instruction from memory
- Egal <- R[rs] = = R[rt] Calculates the branch condition
- if(COND = 0) Calculates the next instruction address
- PC <- PC + 4 + (ExtSemn(imm16) x4)
- else
- PC <- PC + 4

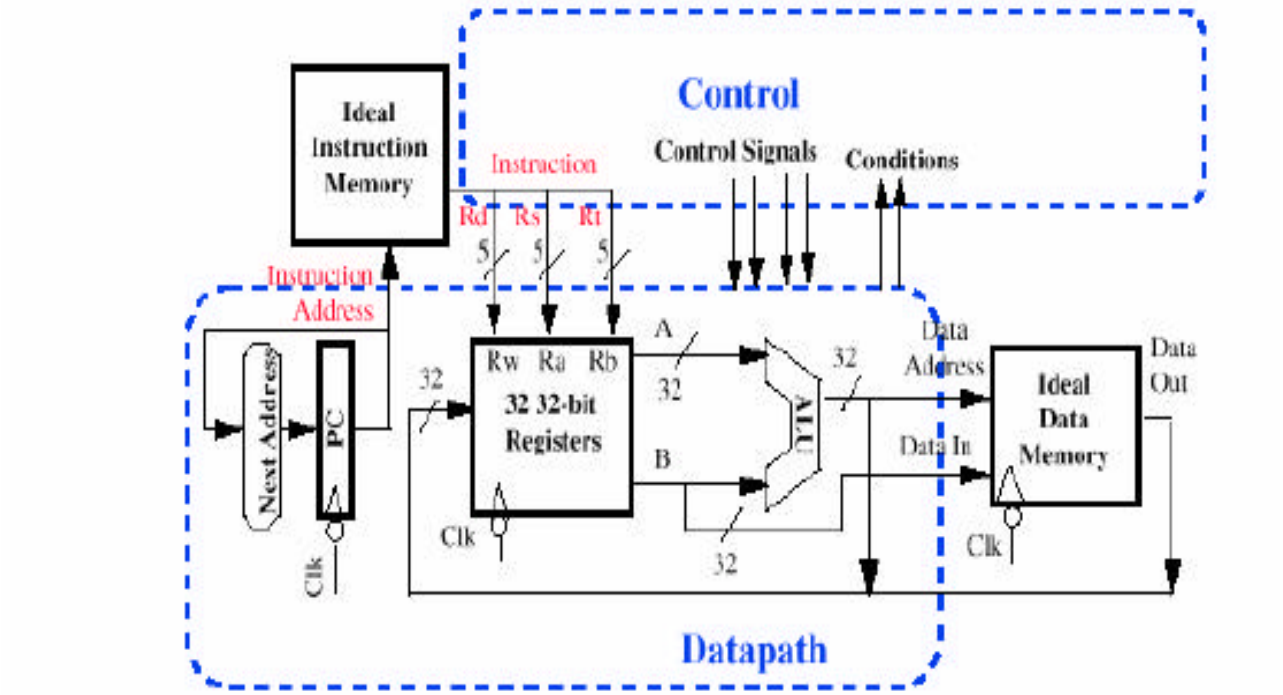
The execution unit for branch command: - generates equal

The semnifications of the command signals are the following:
- Rs, Rt and Imed16 are cabled in the Execution Unit
- nPC_sel: 0 => PC <- PC + 4; 1=> PC <-PC + 4 + ExtSemn(Imed16) || 00

The logic for each command signal is the following:

```
° nPC_sel    <= if (OP == BEQ) then EQUAL else 0
° ALUsrc     <= if (OP == "000000") then "regB" else "immed"
° ALUctr     <= if (OP == "000000") then  funct
                elseif (OP == ORi) then "OR"
                elseif (OP == BEQ) then "sub"
                else "add"
° ExtOp      <= if (OP == ORi) then "zero" else "sign"
° MemWr      <= (OP == Store)
° MemtoReg   <= (OP == Load)
° RegWr:     <= if ((OP == Store) II (OP == BEQ)) then 0 else 1
° RegDst:    <= if ((OP == Load) II (OP == ORi)) then 0 else 1
```

The abstract representation of the MIPS processor:

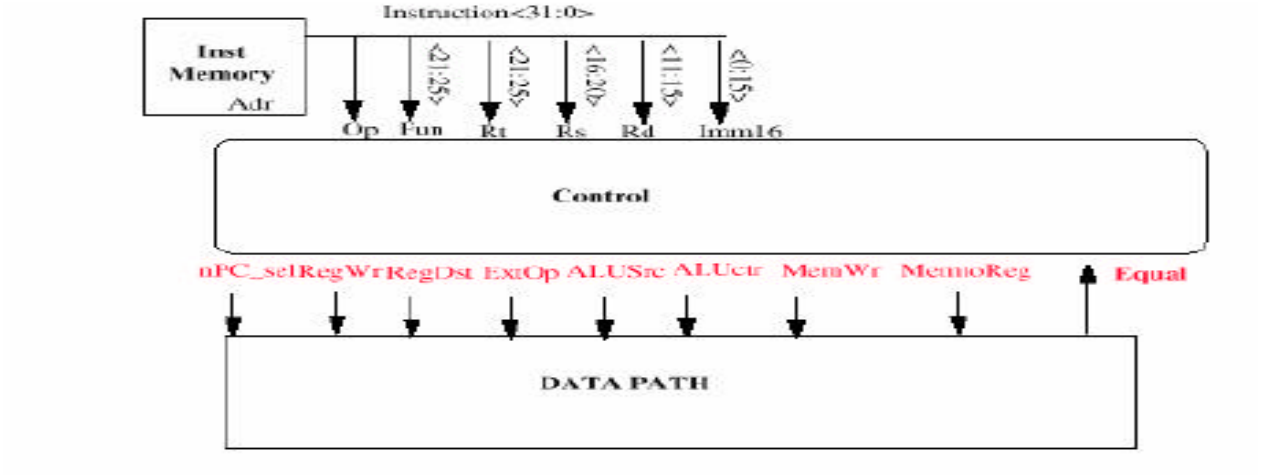


The MIPS structure is characterized by the following elements:

- the instructions have a fixed length
- the source registers are positioned in the same fields
- "immediately" has the same dimension and positioning
- the operations are made with operands from the registers or from the registers field

The Execution Unit works inb a single cycle so CPI=1, and the duration of the cycle is very big.

Project of the command unit;

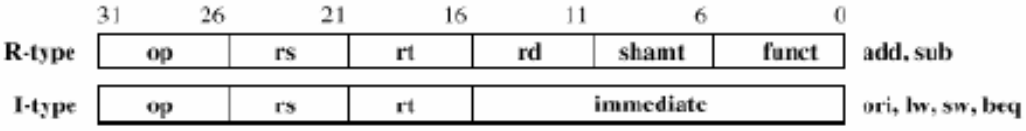


Summary of the command signals:

inst	Register Transfer	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = "add", RegDst = rd, RegWr, nPC_sel = "+4"	
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = "sub", RegDst = rd, RegWr, nPC_sel = "+4"	
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
	ALUsrc = Im, Extop = "Z", ALUctr = "or", RegDst = rt, RegWr, nPC_sel = "+4"	
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
	ALUsrc = Im, Extop = "Sn", ALUctr = "add", MemtoReg, RegDst = rt, RegWr,	nPC_sel = "+4"
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
	ALUsrc = Im, Extop = "Sn", ALUctr = "add", MemWr, nPC_sel = "+4"	
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4	
	nPC_sel = "Br", ALUctr = "sub"	

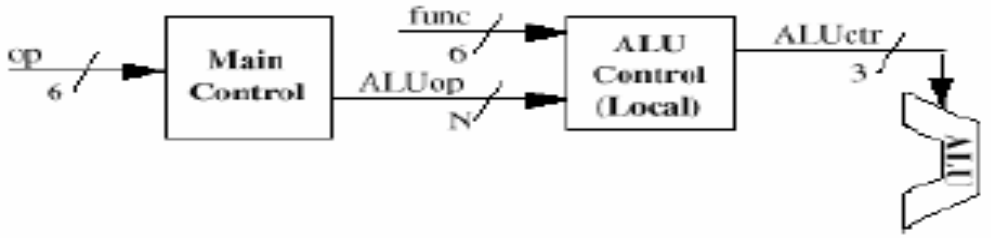
The command signal table (summary)

See Appendix A → func	10 0000	10 0010	We Don't Care :-)				
→ op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	**sub**	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
nPCsel	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

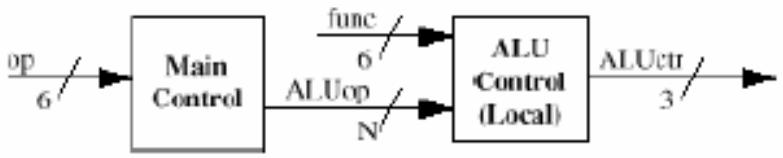


Concept of local decodification:

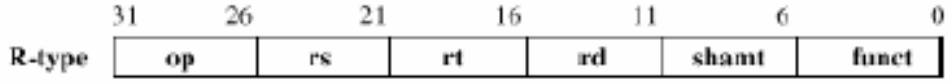
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



The decodification of "func":



	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx



funct<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than

The logical equations for ALUctr<2> are the following:

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<2>
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

This makes func<3> a don't care

- **ALUctr<2> = !ALUop<2> & ALUop<0> +**
  ALUop<2> & !func<2> & func<1> & !func<0>

The logical equations for ALUctr<1> are the following:

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<1>
0	0	0	x	x	x	x	1
0	x	1	x	x	x	x	1
1	x	x	0	0	0	0	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

- ALUctr<1> = !ALUop<2> & !ALUop<0> +
  ALUop<2> & !func<2> & !func<0>

The logical equations for ALUctr<0> are the following:

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
0	1	x	x	x	x	x	1
1	x	x	0	1	0	1	1
1	x	x	1	0	1	0	1

- ALUctr<0> = !ALUop<2> & ALUop<0>
  + ALUop<2> & !func<3> & func<2> & !func<1> & func<0>
  + ALUop<2> & func<3> & !func<2> & func<1> & !func<0>

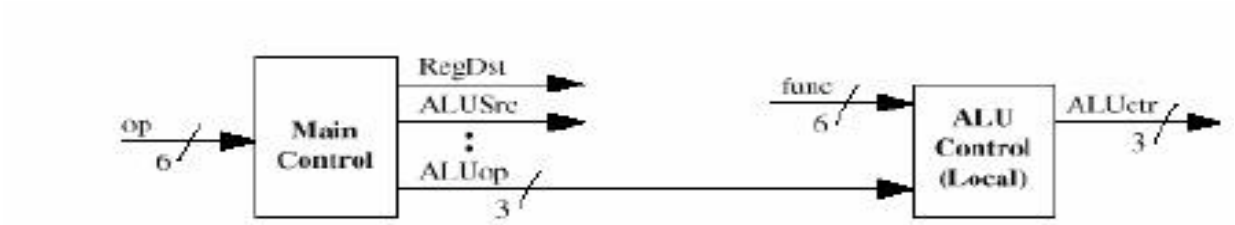
The command bloc for the UAL is the following:

- ALUctr<2> = !ALUop<2> & ALUop<0> +
  ALUop<2> & !func<2> & func<1> & !func<0>
- ALUctr<1> = !ALUop<2> & !ALUop<0> +
  ALUop<2> & !func<2> & !func<0>
- ALUctr<0> = !ALUop<2> & ALUop<0>
  + ALUop<2> & !func<3> & func<2> & !func<1> & func<0>
  + ALUop<2> & func<3> & !func<2> & func<1> & !func<0>

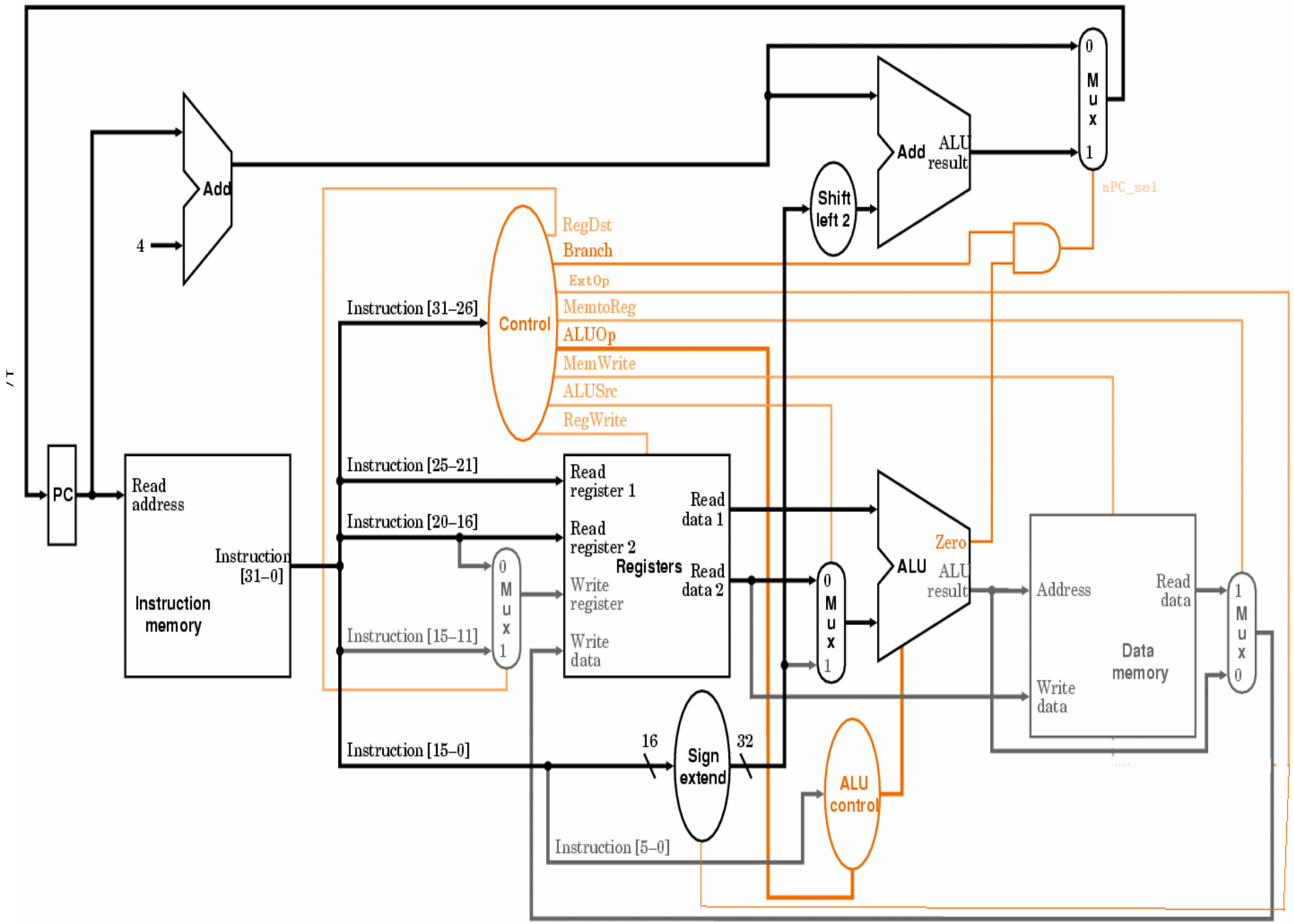
Pasul 5 : Logica pentru fiecare semnal de comanda

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == "Rtype") then "regB" else "immed"
- ALUctr <= if (OP == "Rtype") then funct
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub"
  else "add"

The logic for the main command:



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x


Add
4
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [31–26]
Control
RegDst
Branch
ExtOp
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
0 M u x 1
16
Sign extend
32
Shift left 2
Add
ALU result
0 M u x 1
nPC_sel
ALU
Zero
ALU result
ALU control
Address
Read data
Data memory
Write data
1 M u x 0

The main problems of the single clock cycle processor are the following:
-big work cycle
The work cycle must be big enough for the load instruction
CP clock->Q+
Memory of Instructions access time
General Registers access time(register file R) +
UAL delay(when calculating the address)+
Time to access the data memory+
Time to establish the general registers+
Clock errors.
-the duration of the cycle for the load instruction – is much bigger than that taken for the other instructions.
The command general syntax is :

R-type

6	5	5	5	5	6
Op	**Rs**	**Rt**	**Rd**	**Shamt**	**Func**

Add
R[rd]<-R[rs]+R[rt]
000000.yyyyy.yyyyy.yyyyy.xxxxx.100000

Sub
R[rd]<-R[rs]-R[rt]
000000.yyyyy.yyyyy.yyyyy.xxxxx.100010

I-type

Op	Rs	Rt	Imm16

Ori
R[rt]<-R[rs]+zero_ext(imm16)
001101.yyyyy.yyyyy.iiiiiiiiiiiiiiii

Load
R[rt]<-Mem[R[rs]+sign_ext(imm16)]
10011.yyyyy.yyyyy imm16

Store
Mem[R[rs]+sign_ext(imm16)]<-R[rt]]
101011.yyyyy.yyyyy imm16

Test program

- We load into memory at address 4 the value $(11)_{10}$.
- The register 0 is not used in writing operations but it is used to maintain in it zero value.
- In the register 1 we keep $(11)_{10}$ value.
- In the register 2 we load 1 .
- We use register 3 as index register
- we load in register 4 the memory location referred by register 3
- we compare the values form registers 1 and 4

in case of equality the program ends

else it continues with the next instruction

-we increment register 3

- we make an unconditional loop to the memory load instruction

0) ori r0,r1, $(11)_{10}$
1) ori r0,r2, $(1)_{10}$
2) lw r4,r3 (0000)
3) beq r4,r1, $(2)_{10}$
4) add r3,r2,r1
5) beq r0,r0,$(-4)_{10}$

Bibliography

CN Courses –Course 5 from the 2'nd semester

-Course 8 from the 1'st semester

On-line documentation : http://www.deeps.org
http://www.csit-sun.pub.ro