PROCESORUL MIPS 32

CICLU UNIC

TAR IVETT - DORA
UNIVERSITATEA TEHNICĂ CLUJ NAPOCA
Facultatea de Automatică și Calculatoare

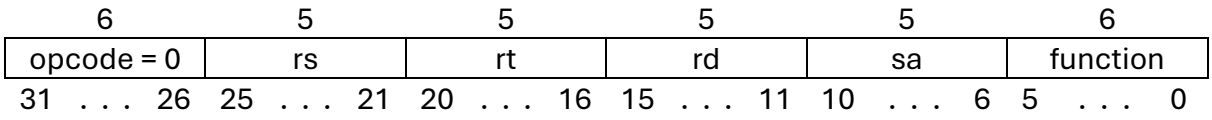
Contents

Instrucțiuni pentru MIPS32

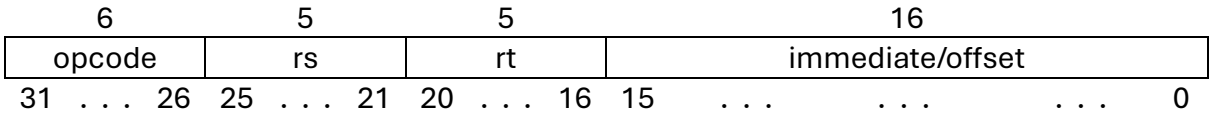
Tipul instrucțiunii	Denumirea instrucțiunii	Prescurtare
Tip R	Addition	add
	Substraction	sub
	Shift Left Logical	sll
	Shift Right Logical	srl
	Logical AND	and
	Logical OR	or
	Logical XOR	xor
	Shift Left Logical Variable	sllv
Tip I	Add Immediate	addi
	Load Word	lw
	Store Word	sw
	Branch on Equal	beq
	Branch on Greater than or Equal to Zero	bgez
	Set on Less Than Immediate(signed)	slti
Tip J	Jump	j

Formatul instrucțiunii

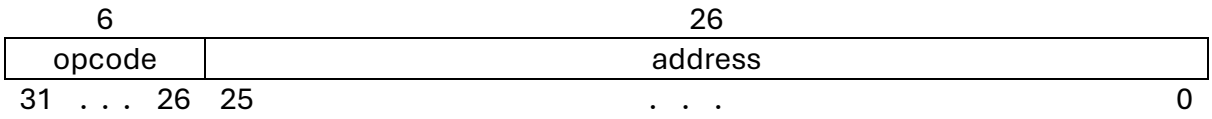
Tip R



Tip I



Tip J



Setul de instrucțiuni ales

Instruction	Type	Assembly	RTL Abstract
add	R	add $rd, $rs, $rt	RF[rd] <= RF[rs] + RF[rt]; PC <= PC + 4
sub	R	sub $rd, $rs, $rt	RF[rd] <= RF[rs] - RF[rt]; PC <= PC + 4
and	R	and $rd, $rs, $rt	RF[rd] <= RF[rs] & RF[rt]; PC <= PC + 4
or	R	or $rd, $rs, $rt	RF[rd] <= RF[rs] | RF[rt]; PC <= PC + 4
xor	R	xor $rd, $rs, $rt	RF[rd] <= RF[rs] ^ RF[rt]; PC <= PC + 4
sll	R	sll $rd, $rt, sa	RF[rd] <= RF[rt] << sa; PC <= PC + 4
srl	R	srl $rd, $rt, sa	RF[rd] <= RF[rt] >> sa; PC <= PC + 4
sllv	R	sllv $rd, $rt, $rs	RF[rd] <= RF[rt] << RF[rs]; PC <= PC + 4
addi	I	addi $rt, $rs, imm	RF[rt] <= RF[rs] + S_Ext(imm); PC <= PC + 4
lw	I	lw $rt, offset($rs)	RF[rt] <= Mem[RF[rs] + S_Ext(offset)]; PC <= PC + 4
sw	I	sw $rt, offset($rs)	Mem[RF[rs] + S_Ext(offset)] <= RF[rt]; PC <= PC + 4
beq	I	beq $rs, $rt, offset	If RF[rs] = RF[rt] then PC <= PC + 4 + S_Ext(offset)<<2
bgez	I	bgez $rs, offset	If RF[rs] >= 0 then PC <= PC + 4 + S_Ext(offset)<<2
slti	I	slti $rt, $rs, imm	If RF[rs] < imm then RF[rt] <= 1 Else RF[rd] <= 0; PC <= PC + 4
j	J	j address	PC <= (PC + 4)[31:28] || address || 00

Function codes

Function	Code
add	100000
sub	100001
and	010001
or	010010
xor	010011
sll	001010
srl	001100
sllv	001110

Opcodes

Operation	Code
addi	000001
lw	110000
sw	110001
beq	011000
bgez	011001
slti	001000
j	111111

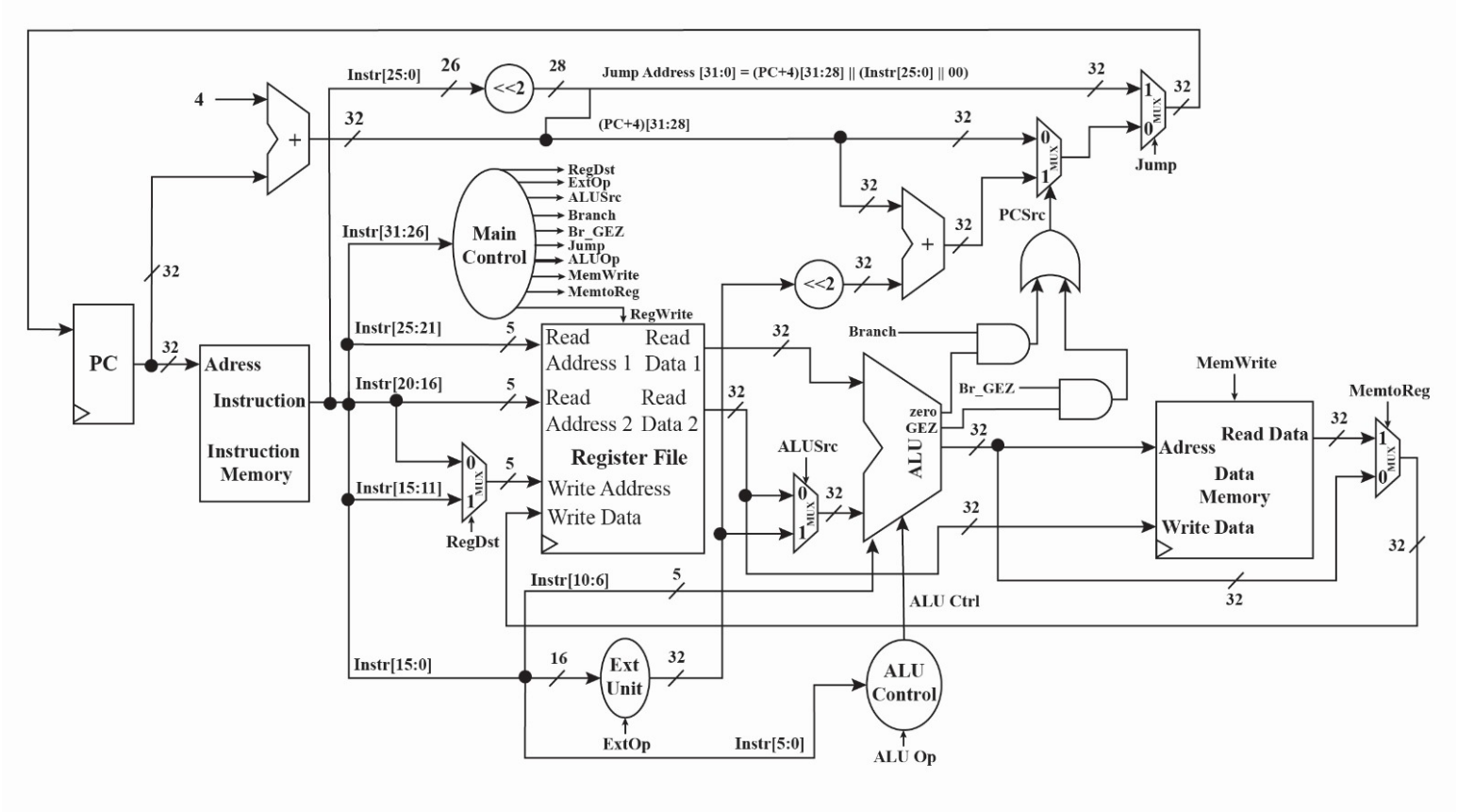
Semnale de control MIPS32

Instrucțiune	Opcode *Instr[31-26]*	Reg Dst	Ext Op	ALU Src	Branch	Br_gez	Jump	Mem Write	Memto Reg	Reg Write	ALU Op[1:0]	function *Instr[5-0]*	ALUCtrl[3:0]
ADD	000000	1	0	0	0	0	0	0	0	1	10(R)	100000	0000(+)
SUB	000000	1	0	0	0	0	0	0	0	1	10(R)	100001	1000(-)
SLL	000000	1	0	0	0	0	0	0	0	1	10(R)	001010	0100(<<l)
SRL	000000	1	0	0	0	0	0	0	0	1	10(R)	001100	0101(>>l)
SLLV	000000	1	0	0	0	0	0	0	0	1	10(R)	001110	0111(<<lv)
AND	000000	1	0	0	0	0	0	0	0	1	10(R)	010001	0001(&)
OR	000000	1	0	0	0	0	0	0	0	1	10(R)	010010	0010(|)
XOR	000000	1	0	0	0	0	0	0	0	1	10(R)	010011	0011(x)
ADDI	000001	0	1	1	0	0	0	0	0	1	00(+)	xxxxxx	0000(+)
SLTI	001000	0	1	1	0	0	0	0	0	1	11	xxxxxx	1011
LW	110000	0	1	1	0	0	0	0	1	1	00(+)	xxxxxx	0000(+)
SW	110001	x	1	1	0	0	0	1	x	0	00(+)	xxxxxx	0000(+)
BEQ	011000	x	1	0	1	0	0	0	x	0	01(-)	xxxxxx	1000(-)
BGEZ	011001	x	1	0	0	1	0	0	x	0	01(-)	xxxxxx	1000(-)
J	111111	x	x	x	x	x	1	0	x	0	xx	xxxxxx	1011

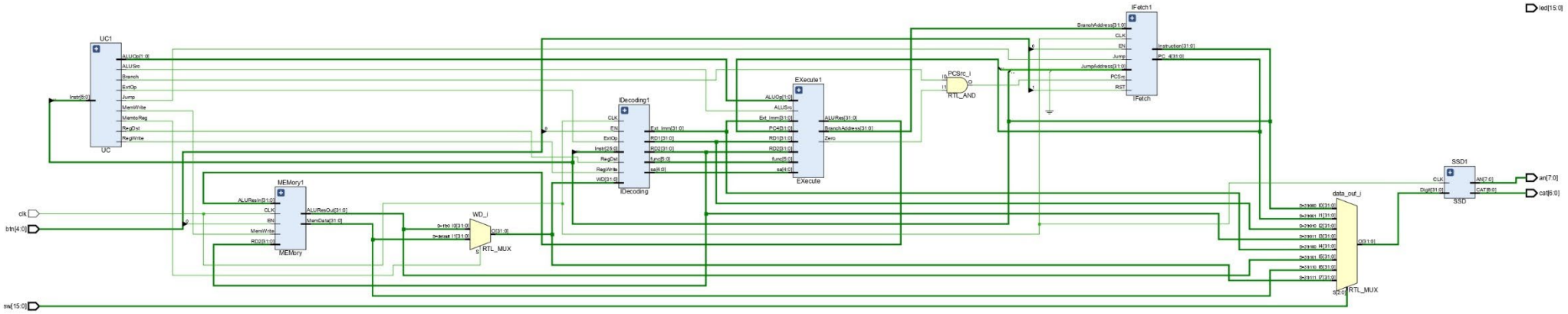
Trasarea execuției

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	LW $1, 0[$0]	X"C0010000"	X"00000004"	X"00000000"	X"00000000"	X"00000000"	X"00000000"	X"00000018"	X"00000018"		
1	LW $2, 4[$0]	X"C0020004"	X"00000008"	X"00000000"	X"0000000C"	X"00000010"	X"00000000"	X"00000010"	X"00000024"		
2	SW $1, 8[$0]	X"C4010008”	X"0000000C"	X"00000000"	X"00000018"	X"0000000C"	X"0000000C"	X"00000018"	X"0000000C"		
3	SW $2, 12[$0]	X"C402000C"	X"00000010"	X"00000000"	X"00000024"	X"00000010"	X"00000010"	X"00000024"	X"00000010"		
4	ADDI $3, 16[$0]	X"04030010"	X"00000014"	X"00000000"	X"00000024"	X"00000014"	X"00000014"	X"0000000C"	X"00000014"		
5	ADDI $8, 1[$0]	X"04080001"	X"00000018"	X"00000000"	X"00000001"	X"00000001"	X"00000001"	X"00000018"	X"00000001"		
6	SUB $4, $1, $2	X"00222021"	X"0000001C"	X"00000018"	X"00000024"	X"00000021"	X"FFFFFFF4"	X"00000000"	X" FFFFFFF4"		
7	SLTI $5, $4, 0	X"20850000"	X"00000020"	X" FFFFFFF4"	X"00000000"	X"00003020"	X"00000024"	X"00000000"	X"00000024"		
8	BEQ $5, $8, 7	X"60A80007"	X"00000024"	X"00000001"	X"00000001"	X"00000007"	X"00000000"	X"00000018"	X"00000000"	X"00000040"	
9	ADD $6, $2, $0	X"00403020"	X"00000044"	X"00000024"	X"00000000"	X"00003020"	X"00000000"	X"00000018"	X"00000018"		
10	ADD $2, $1, $0	X"00201020"	X"00000048"	X"00000018"	X"00000000"	X"00001020"	X"00000018"	X"00000018"	X"00000018"		
11	ADD $1, $5, $0	X"00C00820"	X"0000004C"	X"00000024"	X"00000000"	X"00000820"	X"00000024"	X"00000000"	X"00000024"		
12	J 09	X"FC000009"	X"00000050"	X"00000000"	X"00000000"	X"00000009"	X"00000000"	X"00000018"	X"00000000"		X"00000024"
13	SUB $1, $1, $2	X"00220821"	X"00000028"	X"00000024"	X"00000018"	X"00000821"	X"0000000C"	X"00000018"	X"0000000C"		
14	SW $1, 0[$3]	X"C4610000"	X"0000002C"	X"00000014"	X"0000000C"	X"00000000"	X"00000014"	X"00000005"	X"00000014"		
15	SW $2, 4[$3]	X"C4620004"	X"00000030"	X"00000014"	X"00000018"	X"00000004"	X"00000018"	X"00000006"	X"00000018"		
16	ADDI $3, $3, 8	X"04630008"	X"00000034"	X"00000014"	X"00000014"	X"00000008"	X"0000001C"	X"00000007"	X"0000001C"		
17	SUB $4, $1, $2	X"00222021"	X"00000038"	X"0000000C"	X"00000018"	X"00002021"	X"FFFFFFF4"	X"00000000"	X"FFFFFFF4"		
18	BEQ $4, $0, 5	X"60800005"	X"0000003C"	X"FFFFFFF4"	X"00000000"	X"00000005"	X"FFFFFFF4"	X"00000000"	X"FFFFFFF4"	X"00000050"	
19	J 06	X"FC000006"	X"00000040"	X"00000000"	X"00000000"	X"00000006"	X"00000000"	X"00000018"	X"00000000"		X"00000018"
20	SW $1, 0[$3]	X"C4610000"	X"00000054"	X"00000024"	X"0000000C"	X"00000000"	X"00000024"	X"0000000C"	X"00000024"		

Schema procesorului MIPS 32



RTL schematic



Problema aleasă

Să se determine cel mai mare divizor comun a 2 numere *X* și *Y* citite din memorie de la adresa 0, respectiv 4. Se va implementa algoritmul lui Euclid cu scăderi. Se vor scrie în memorie, la locații consecutive pe 32 de biți, începând cu adresa 8, valorile *X*, *Y,* urmate de cele intermediare generate de algoritmul lui Euclid. Ultimul număr scris va fi cel mai mare divizor comun.

Codul în C++

```
int main()
{
    int n , m;
    std :: cin >> n >> m;
    while(n != m)
        if(n > m)
            n -= m;
        else
            m -= n;
    std :: cout << n << std :: endl;
    return 0;
}
```

Codul în VHDL

```
signal instruction_memory : rom := (
     B"110000_00000_00001_0000000000000000", -
     B"110000_00000_00010_0000000000000100", -
     B"110001_00000_00001_0000000000001000", -
     B"110001_00000_00010_0000000000001100", -
     B"000001_00000_00011_0000000000010000", -
     B"000001_00000_01000_0000000000000001", -
     B"000000_00001_00010_00100_00000_100001",
     B"001000_00100_00101_0000000000000000", -
     B"011000_00101_01000_0000000000000111", -
     B"000000_00001_00010_00001_00000_100001",
     B"110001_00011_00001_0000000000000000", -
     B"110001_00011_00010_0000000000000100", -
     B"000001_00011_00011_0000000000001000", -
     B"000000_00001_00010_00100_00000_100001",
     B"011000_00100_00000_0000000000000101", -
     B"111111_00000000000000000000000000110", --
     B"000000_00010_00000_00110_00000_100000",
     B"000000_00001_00000_00010_00000_100000",
     B"000000_00110_00000_00001_00000_100000",
     B"111111_00000000000000000000000001001", --
     B"110001_00011_00001_0000000000000000", -
     others => X"00000000");
```

Acest program rezolvă în totalitate cerințele problemei, fiind testat atât în simulare, cât și pe placă. Problemele care au apărut au fost rezolvate.

Exemplu

Vom considera în acest exemplu ca la adresa 0 vom avea valoarea 24(18 hexa) și la adresa 4 vom avea valoarea 36(24 hexa) și vom urmări cum calculează programul cel mai mare divizor comun al acestor numere.

00: 110000_00000_00001_0000000000000000 ⇔ lw $1, 0[$0]

- ➔ In R1 se scrie cuvântul de la adresa 0, iar in exemplul nostru x18
- ➔ La acest pas sunt utile : WD=18, WA=1, RD2=0, RD1=0, RA2=1, RA1=0, Address=0

01: 110000_00000_00010_0000000000000100 ⇔ lw $2, 4[$0]

- ➔ In R2 se scrie cuvântul de la adresa 4, iar in exemplul nostru x24
- ➔ La acest pas sunt utile : WD=24, WA=2, RD2=0, RD1=0, RA2=2, RA1=0, Address=4

02: 110001_00000_00001_0000000000001000 ⇔ sw $1, 8[$0]

- ➔ La adresa 8 din memorie se scrie valoarea din R1, la adresa 8 va fi x18
- ➔ La acest pas sunt utile : MemData=18, Address=8, RD2=18, RD1=0, RA2=1, RA1=0

03: 110001_00000_00010_0000000000001100 ⇔ sw $2, 12[$0]

- ➔ La adresa 12 din memorie se scrie valoarea din R2, la adresa 12 va fi x24
- ➔ La acest pas sunt utile : MemData=24, Address=12, RD2=24, RD1=0, RA2=2, RA1=0, Imm=C

04: 000001_00000_00011_0000000000010000 ⇔ addi $3, 16[$0]

- ➔ În R3 se adună valoarea lui R0 cu valoarea imediatului, adică 16
- ➔ La acest pas sunt utile : WD=16, WA=3, RD1=0, RA1=0

05: 000001_00000_01000_0000000000000001 ⇔ addi $8, 1[$0]

- ➔ În R8 se adună valoarea lui R0 cu valoarea imediatului, adică 1
- ➔ La acest pas sunt utile : WD=1, WA=8, RD1=0, RA1=0

06: 000000_00001_00010_00100_00000_100001⇔sub $4,$1,$2

- ➔ În R4 se pune diferența dintre R1 și R2, va fi -12
- ➔ La acest pas sunt utile : WD=1, WA=4, RD2=24, RA2=2, RD1=18, RA1=1

07: 001000_00100_00101_0000000000000000⇔slti $5, $4, 0

- ➔ Compară valoarea lui R4 cu valoarea imediatului și inițiează R5 cu 1 sau 0 în funcție de valoarea lui R4 ; aici va fi -12 comparat cu 0, deci R5=1
- ➔ La acest pas sunt utile : WD=1, WA=5, RD2=0, RA2=5, RD1=-8, RA1=4, imm=0

08: 011000_00101_01000_0000000000000111⇔beq $5, $8, 7

- ➔ Compară valoarea din R5 și R8, iar dacă sunt egale sa sară, iar dacă nu se execută în continuare ; R5=1, R8=1, se va sări
- ➔ La acest pas sunt utile : RD2=1, RA2=5, RD1=1, RA1=8, imm=7

09: 000000_00001_00010_00001_00000_100001⇔sub $1, $1, $2

- ➔ În R1 se scrie diferența dintre R1 și R2, în R1 va fi 12
- ➔ La acest pas sunt utile : WD=C, WA=1, RD2=18, RA2=2, RD1=24, RA1=1

0A: 110001_00011_00001_0000000000000000⇔sw $1, 0[$3]

- ➔ Scriem în memorie se scrie valoarea din R1, la adresa indicată de R3
- ➔ La acest pas sunt utile : MemData=C, RD1=10, RA1=3, RD2=C, RA2=1, Address=16(x10)

0B: 110001_00011_00010_0000000000000100⇔sw $2, 4[$3]

- ➔ Scriem în memorie se scrie valoarea din R2, la adresa indicată de R3+4
- ➔ La acest pas sunt utile : MemData=18, RD1=14, RA1=3, RD2=18, RA2=2, Address=16+4=20(x14)

0C: 000001_00011_00011_0000000000001000⇔addi $3, $3, 8

- ➔ Adunăm 8 la R3 pentru a ajunge la adresa încă necompletată
- ➔ La acest pas sunt utile : WD=18, RD2=10, RA2=3, RD1=10, RA1=3, Imm=8

0D: 000000_00001_00010_00100_00000_100001⇔sub $4,$1,$2

- ➔ Se pune R4 valoarea scăderii R1-R2 pentru a putea verifica ulterior dacă cele două sunt egale, aici R4=12-24=-12
- ➔ La acest pas sunt utile : WD=FFFFFFF4, WA=4, RD2=18, RA2=2, RD1=C, RA1=1

0E: 011000_00100_00000_0000000000000101⇔beq $4, $0, 5

- ➔ Dacă în R4 este 0 atunci se sare la sfârșitul codului
- ➔ La acest pas sunt utile : RD2=0, RA2=0, RD1= FFFFFFF4, RA1=4, Imm=5

0F: 111111_00000000000000000000000110⇔j 06

- ➔ Dacă nu s-a efectuat saltul precedent atunci mai este nevoie de o iterație și se sare înapoi la început
- ➔ La acest pas sunt utile : Imm=6

10: 000000_00010_00000_00110_00000_100000⇔add $6, $2, $0

11: 000000_00001_00000_00010_00000_100000⇔add $2, $1, $0

12: 000000_00110_00000_00001_00000_100000⇔add $1, $5, $0

- ➔ Aceasta este partea la care se sare de la începutul codului dacă valoarea din R1 este mai mică decât R2, și se efectuează o interschimare între R1 și R2
- ➔ Adica R6 <= R2, R2 <= R1, R1 <= R6

13: 111111_00000000000000000000001001⇔j 09

- ➔ S-a efectuat interschimbarea și se sare înapoi pentru a continua execuția
- ➔ La acest pas sunt utile : Imm=9

14: 110001_00011_00001_0000000000000000⇔sw $1, 0[$3]

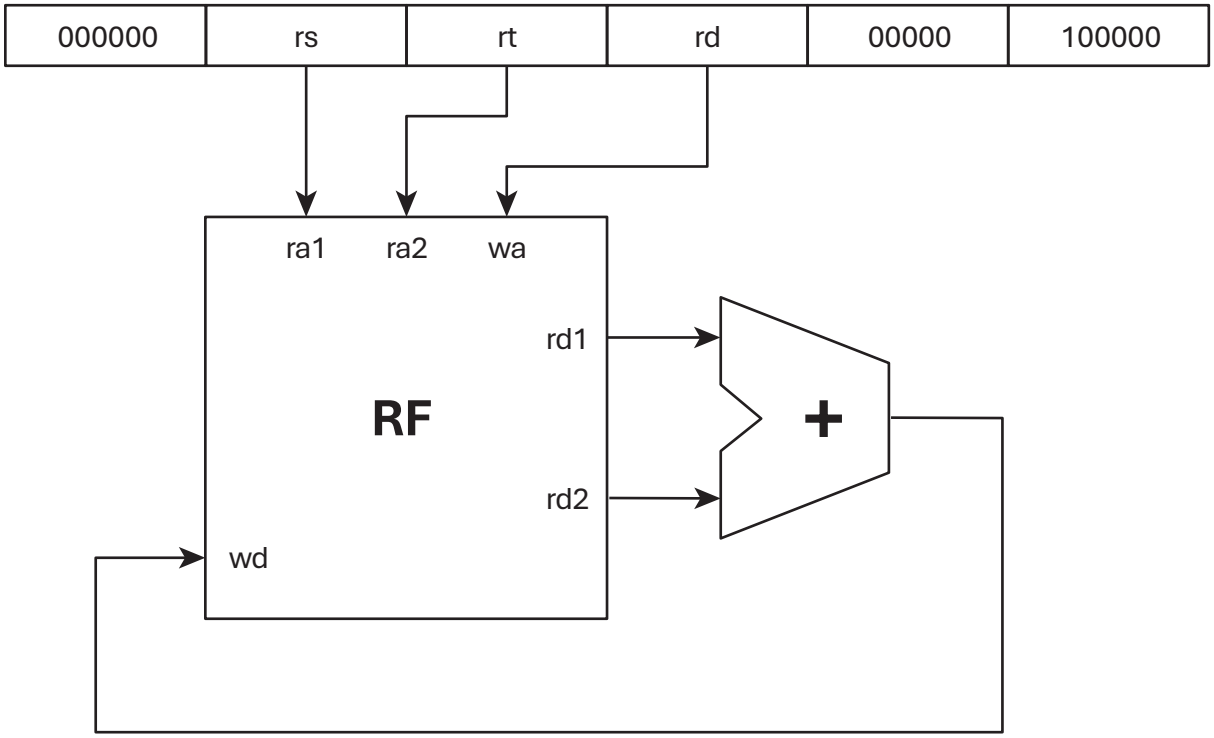
- ➔ Scriem în memorie se scrie valoarea din R1, la adresa indicată de R3
- ➔ La acest pas sunt utile : MemData=C, RD2=X, RA2=3, RD1=C, RA1=1, Address=X
- ➔ Adresa și valoarea lui R3 nu se cunosc, deoarece depinde de numărul de bucle pe care îl efectuează

Descrierea operațiilor

În continuare se vor prezenta schițele detaliate pentru fiecare instrucțiune.

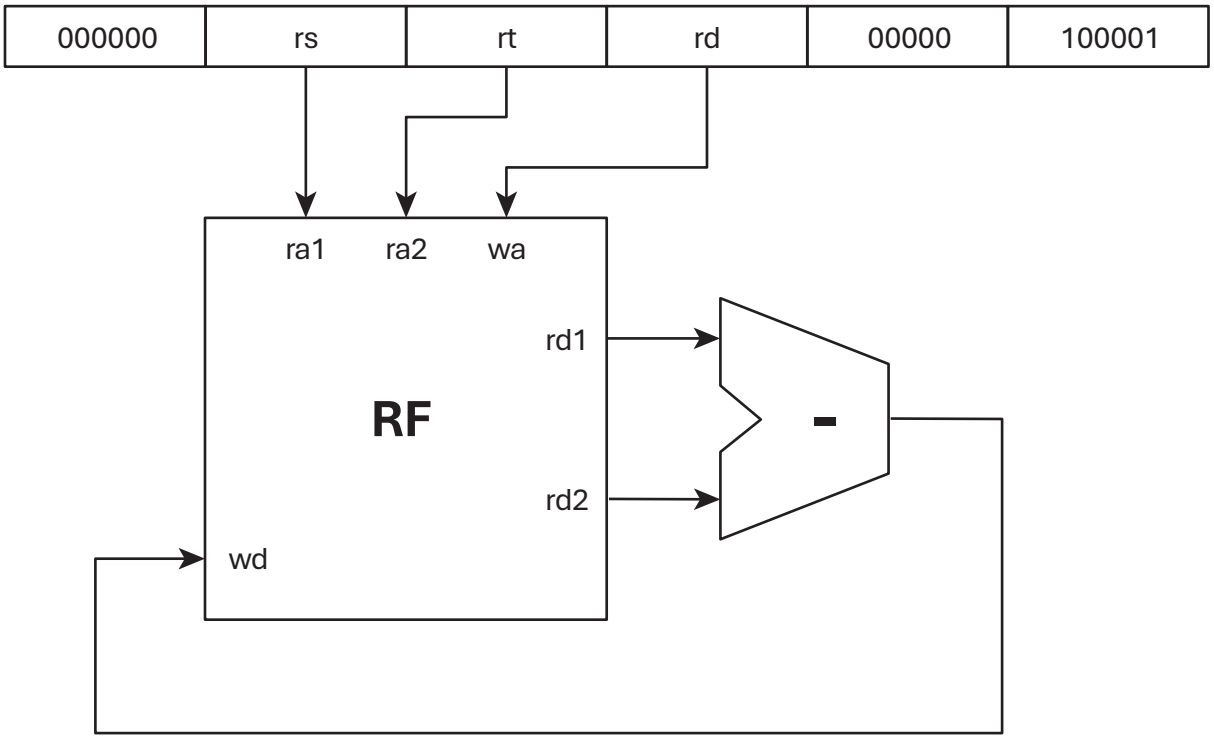
ADD - ADD

Descriere	Adună două registre și memorează rezultatul în al treilea
RTL	$d ← $s + $t; PC ← PC + 4
Sintaxă	add $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100000
Exemplu	add $2, $3, $4 => 000000_00011_00100_00010_00000_100000



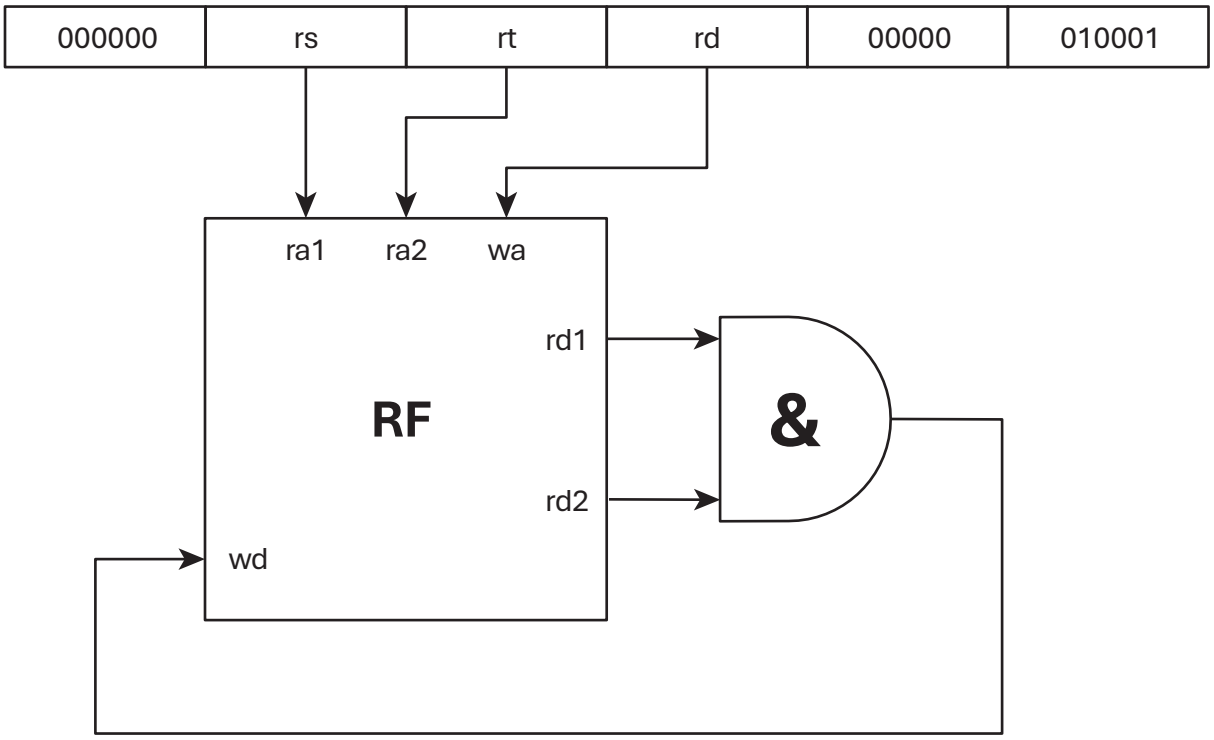
SUB - SUBSTRACT

Descriere	Scade două registre și memorează rezultatul în al treilea
RTL	$d ← $s - $t; PC ← PC + 4
Sintaxă	sub $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100001
Exemplu	sub $3, $4, $5 => 000000_00100_00101_00011_00000_100001



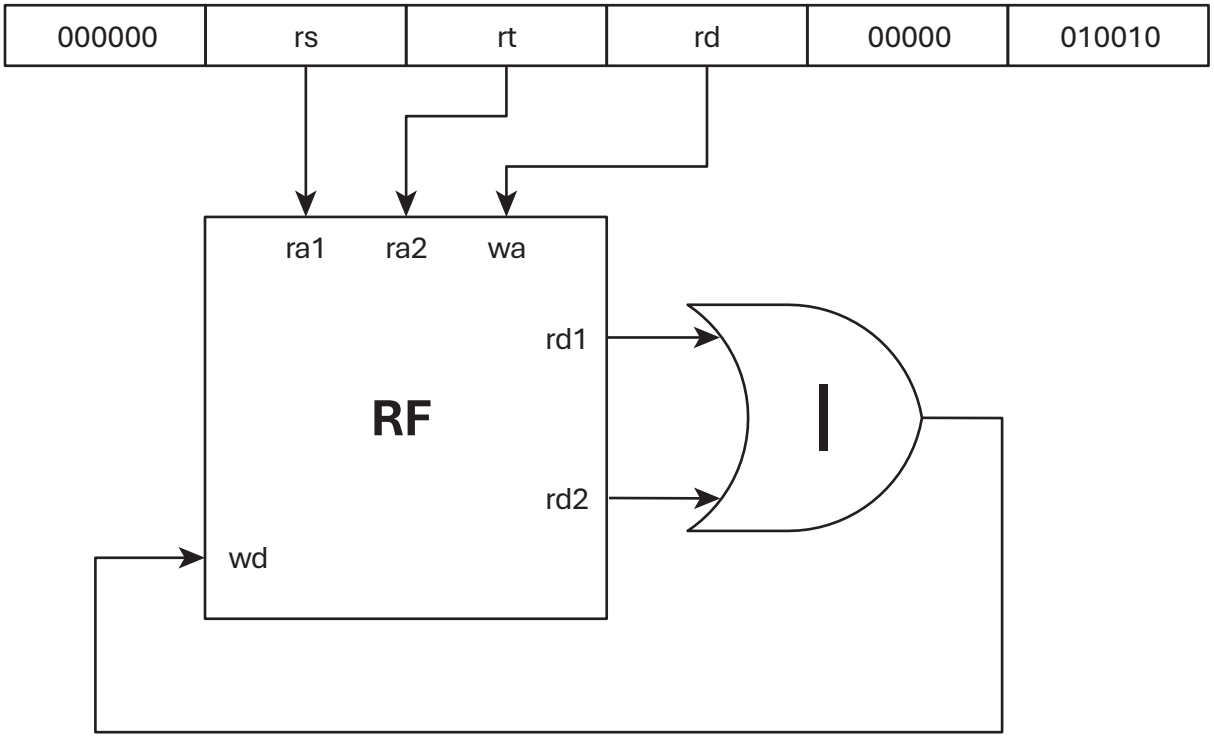
AND - AND Immediate

Descriere	ȘI logic între două registre cu memorarea rezultatului în al treilea
RTL	$d ← $s & $t; PC ← PC + 4
Sintaxă	and $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 010001
Exemplu	and $4, $5, $6 => 000000_00101_00110_00100_00000 010001



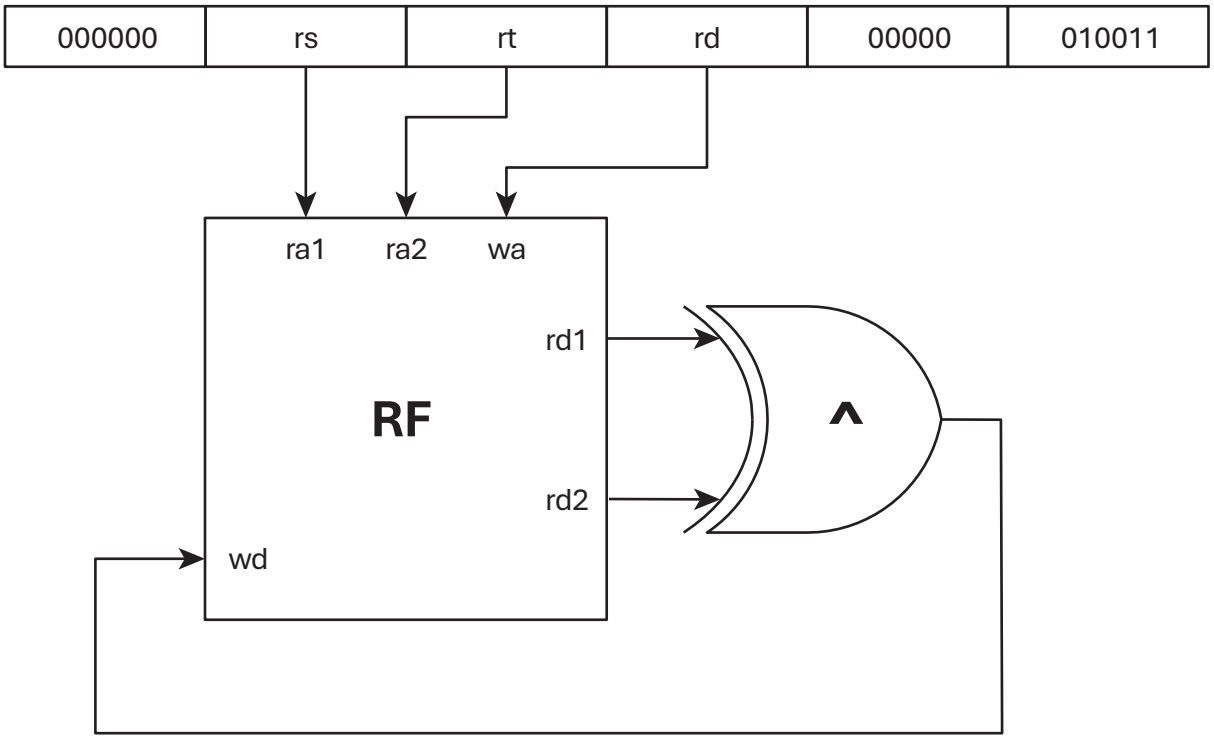
OR - bitwise OR

Descriere	SAU logic între două registre și memorează rezultatul în al treilea
RTL	$d ← $s | $t; PC ← PC + 4
Sintaxă	or $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 010010
Exemplu	or $5, $6, $7 => 000000_00110_00111_00101_00000 010010



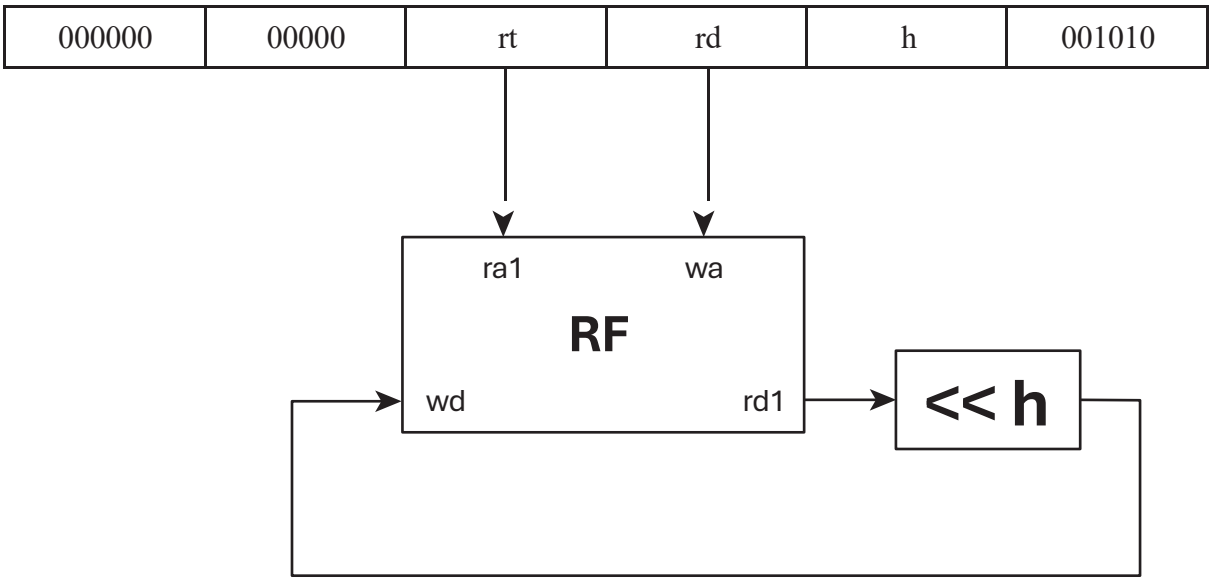
XOR - bitwise eXclusive-OR

Descriere	SAU-Exclusiv logic între două registre, memorează rezultatul în alt registru
RTL	$d ← $s ^ $t; PC ← PC + 4
Sintaxă	xor $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 010011
Exemplu	xor $6, $7, $8 => 000000_00111_01000_00110_00000_010011



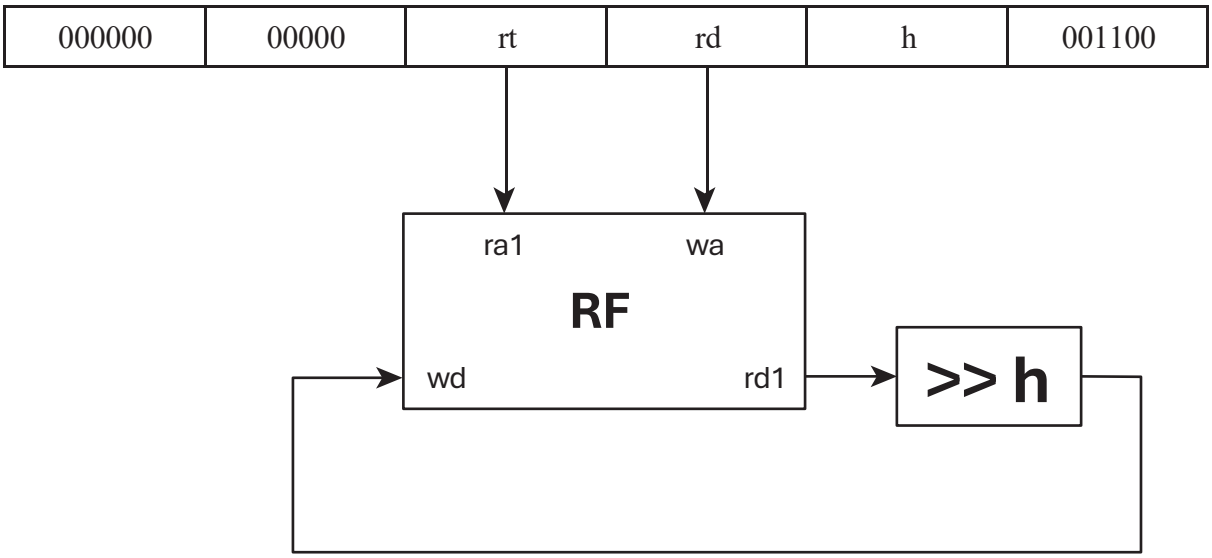
SLL - Shift-Left Logical

Descriere	Deplasare logica la stanga pentru un registru, rezultatul este memorat în alt registru, se introduc zerouri
RTL	$d ← $t << h; PC ← PC + 4
Sintaxă	sll $d, $t, h
Format	000000 00000 ttttt ddddd hhhhh 001010
Exemplu	sll $7, $8, 5 => 000000_00000_01000_00111_00101_001010



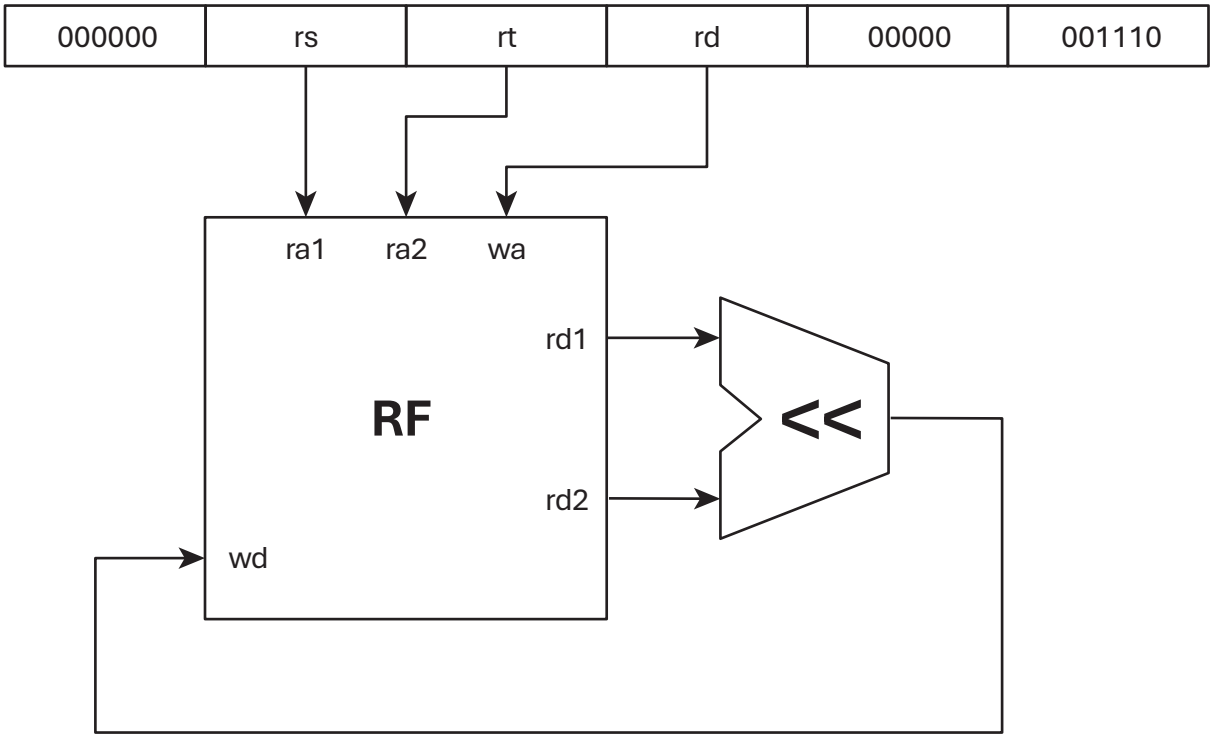
SRL - Shift-Right Logical

Descriere	Deplasare logica la dreapta pentru un registru, rezultatul este memorat în altul, se introduc zerouri
RTL	$d ← $t >> h; PC ← PC + 4
Sintaxă	srl $d, $t, h
Format	000000 00000 ttttt ddddd hhhhh 001100
Exemplu	srl $8, $9, 3 => 000000_00000_01001_01000_00011_001100



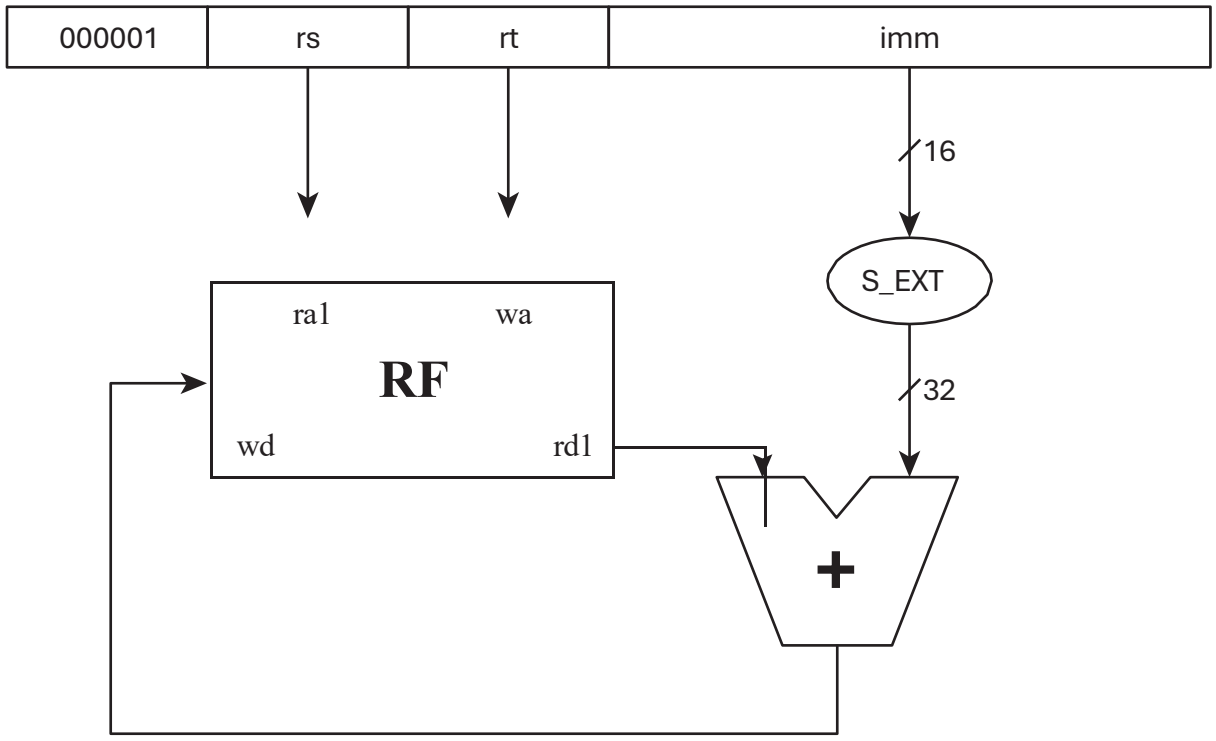
SLLV - Shift-Left Logical Variable

Descriere	Deplasare logica la stanga pentru un registru, cu un numar de pozitii indicat de alt registru, iar rezultatul este memorat într-un al treilea
RTL	$d ← $t << $s; PC ← PC + 4
Sintaxă	sllv $d, $t, $s
Format	000000 sssss ttttt ddddd 00000 001110
Exemplu	sllv $9, $10, $11 => 000000_01011_01010_01001_ 0000_001110



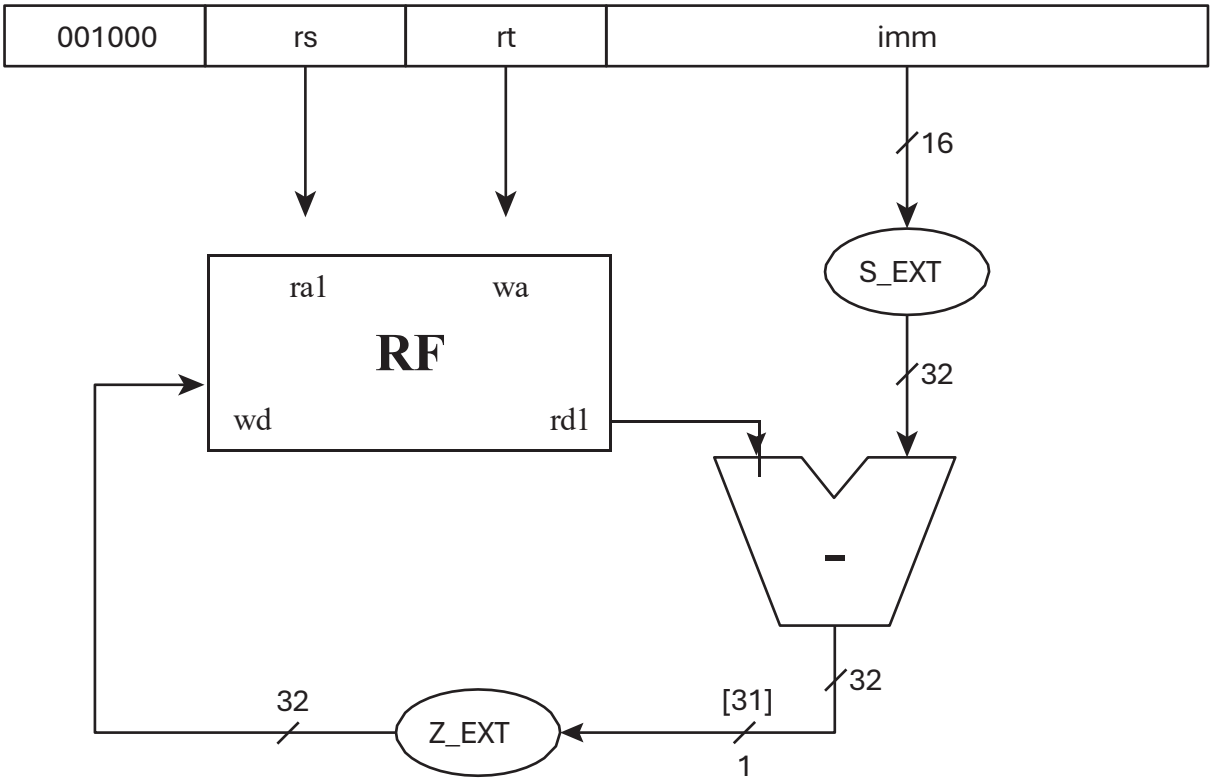
ADDI - ADD Immediate

Descriere	Adună un registru cu o valoare imediată și memorează rezultatul în alt registru
RTL	$t ← $s + SE(imm); PC ← PC + 4
Sintaxă	addi $t, $s, imm
Format	000001 sssss ttttt iiiiiiiiiiiiiiii
Exemplu	addi $10, $3, 32 => 000001_00011_01010_0000000000100000



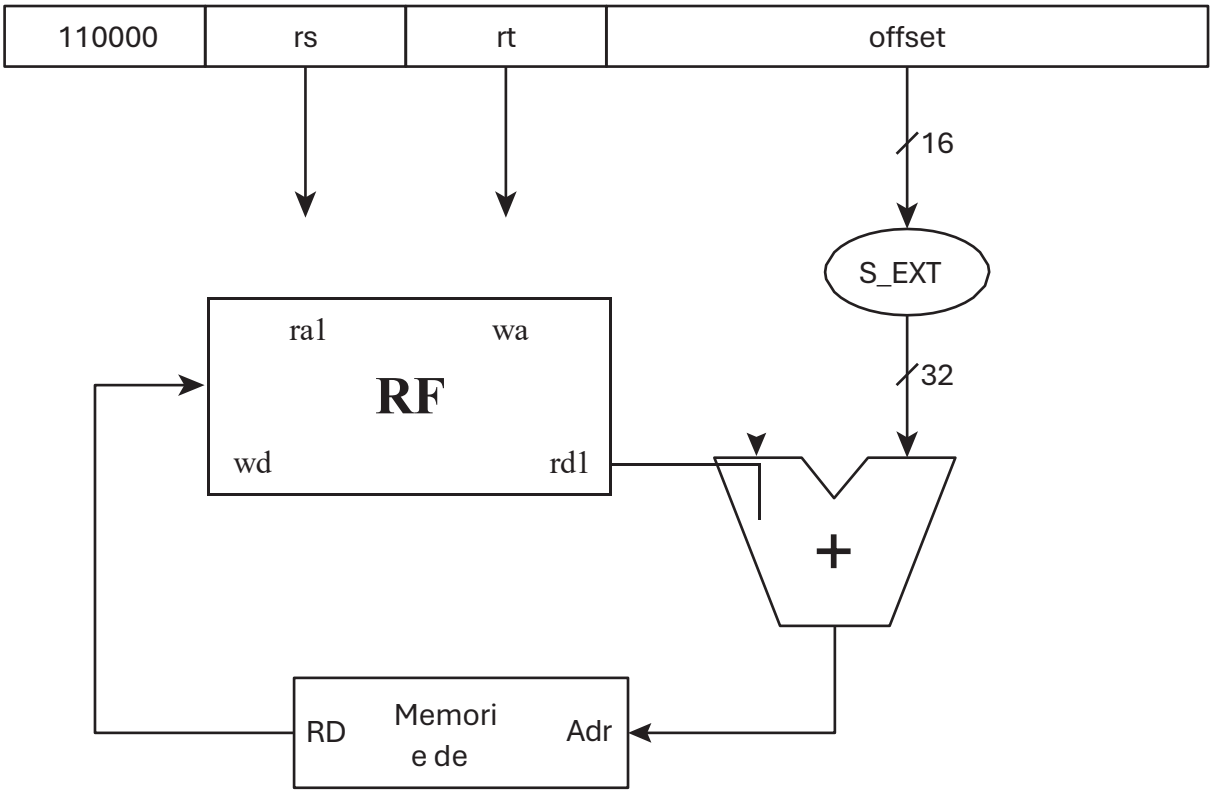
SLTI - Set on Less Than Immediate (signed)

Descriere	Dacă $s este mai mic decât un imediat, $t este initializat cu 1, altfel cu 0
RTL	PC ← PC + 4; if $s < SE(imm) then $t ← 1 else $t ← 0;
Sintaxă	slti $t, $s, imm
Format	001000 sssss ttttt iiiiiiiiiiiiiiii
Exemplu	slti $16, $5, 127 => 001000_00101_10000_0000000001111111



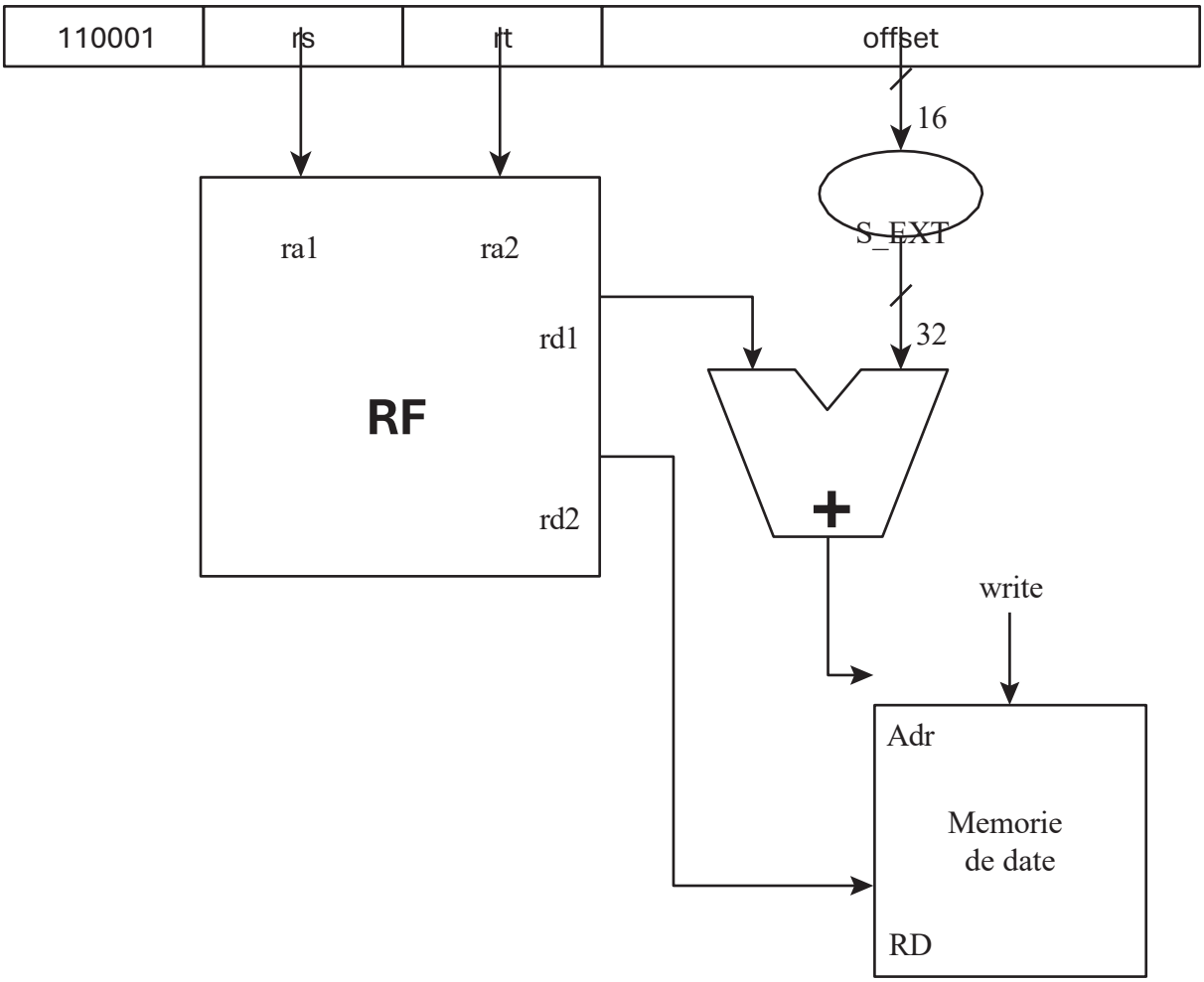
LW - Load Word

Descriere	Un cuvânt din memorie este încărcat într-un registru
RTL	$t ← MEM[$s + SE(offset)]; PC ← PC + 4
Sintaxă	lw $t, offset($s)
Format	110000 sssss ttttt oooooooooooooooo
Exemplu	lw $7, 1($5) => 110000_00101_00111_0000000000000001



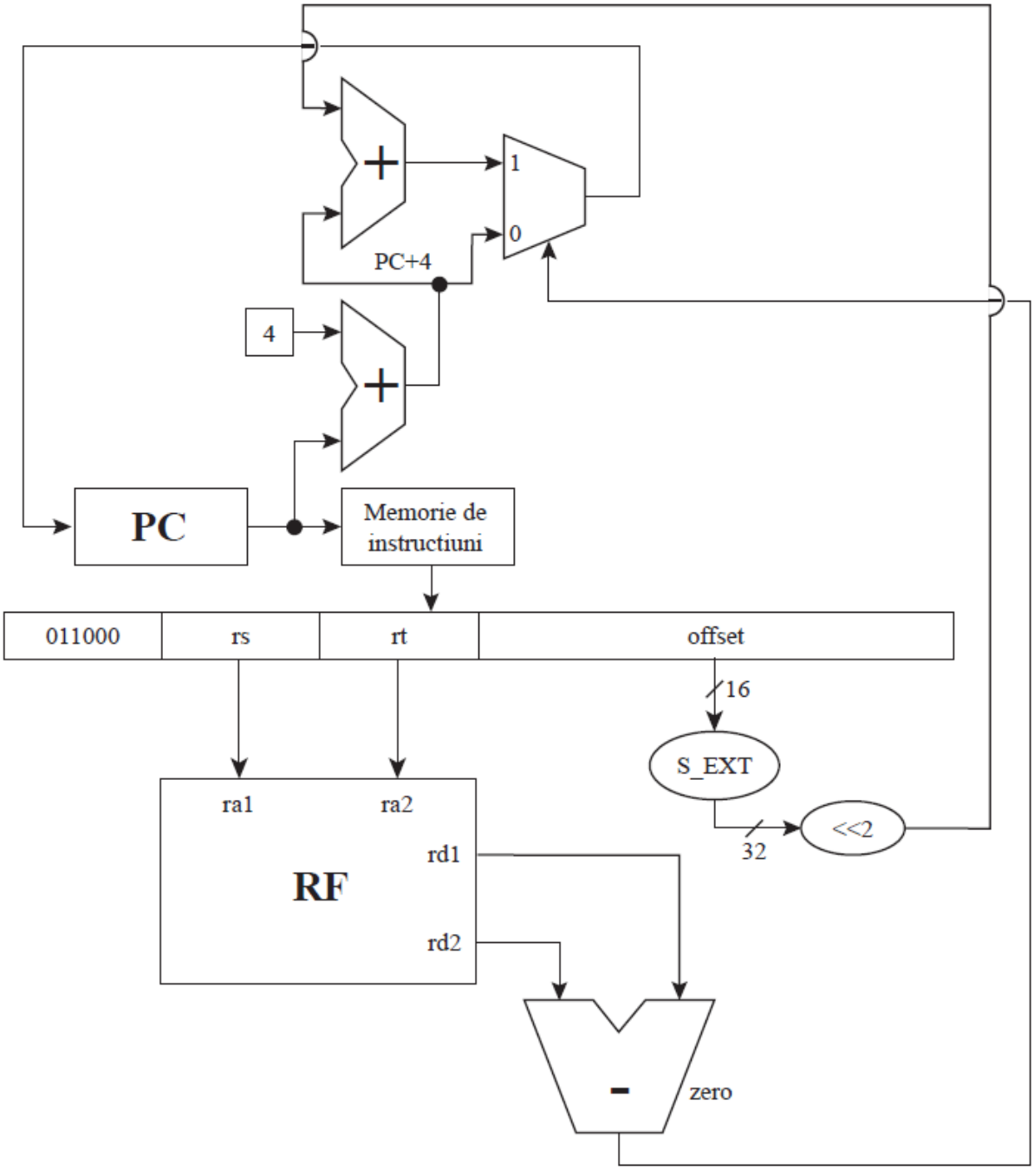
SW - Store Word

Descriere	Valoarea unui registru este stocată în memorie la o anumită adresă
RTL	MEM[$s + SE(offset)] ← $t; PC ← PC + 4
Sintaxă	sw $t, offset($s)
Format	110001 sssss ttttt oooooooooooooooo
Exemplu	sw $10, -1($6) => 110001_00110_01010_1111111111111111



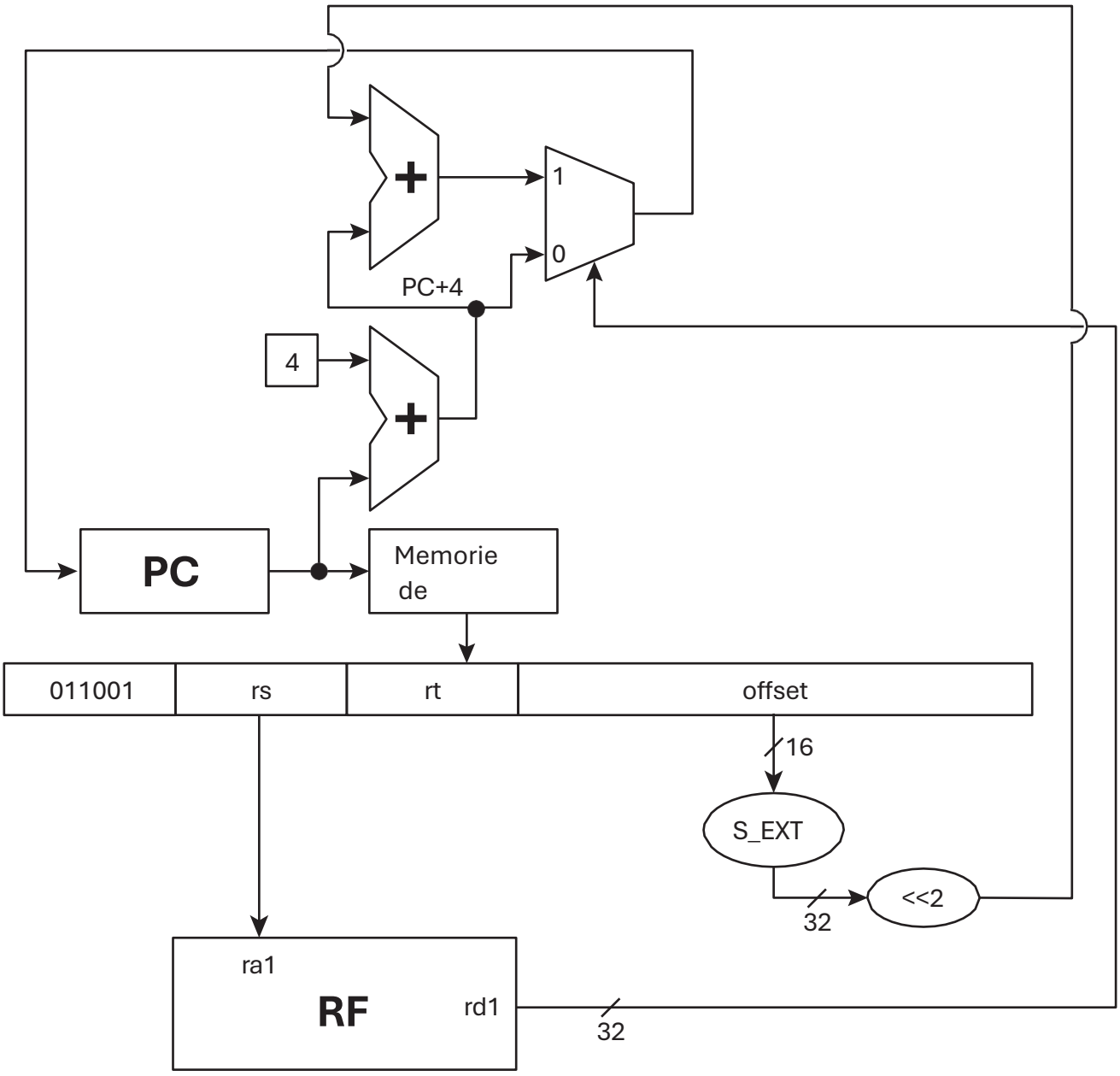
BEQ - Branch on EQual

Descriere	Salt conditionat dacă este egalitate între două registre
RTL	if $s = $t then PC ← (PC + 4) + (SE(offset) << 2) else PC ← PC + 4;
Sintaxă	beq $s, $t, offset
Format	011000 sssss ttttt oooooooooooooooo
Exemplu	beq $2, $3, -1 => 011000_00011_00010_1111111111111111



BGEZ - Branch on Greater than or Equal to Zero

Descriere	Salt conditionat dacă un registru este mai mare sau egal cu 0
RTL	if $s ≥ 0 then PC ← (PC + 4) + (SE(offset) << 2) else PC ← PC + 4;
Sintaxă	bgez $s, offset
Format	011001 sssss 00000 oooooooooooooooo
Exemplu	bgez $5, 4 => 011001_00101_00000_0000000000000100



J - Jump

Descriere	Salt la adresă absolută
RTL	PC ← (PC + 4)[31:28] || (addr << 2);
Sintaxă	j addr
Format	111111 aaaaaaaaaaaaaaaaaaaaaaaaaa
Exemplu	j 10 => 111111_00000000000000000000001010