



Electrical Engineering Department
Computer Engineering Department
3rd Year

pipelined MIPS

Instructor: Dr. Mohamed Mahmoud Ibrahim

Abdullah Khaled Kamal El Sayed Ali Siam
Abdullah Mohammed Abu Almajd Ali Mohamed Albasony
Abdulrahman Rajab Hashim Ismail
Assem Hossam Mahmoud
Eman Mahmoud Rashwan Rashed
Hussein Mostafa Said Elkholy
Mahmoud Hassan Mohamed Mahmoud
Osama Mohamed Abdel Tawab Ramadan
Saleh Mahmoud Saleh Abdullah Mohammed
Zaghlola Atef Abdel Mawla Abdel Wahab

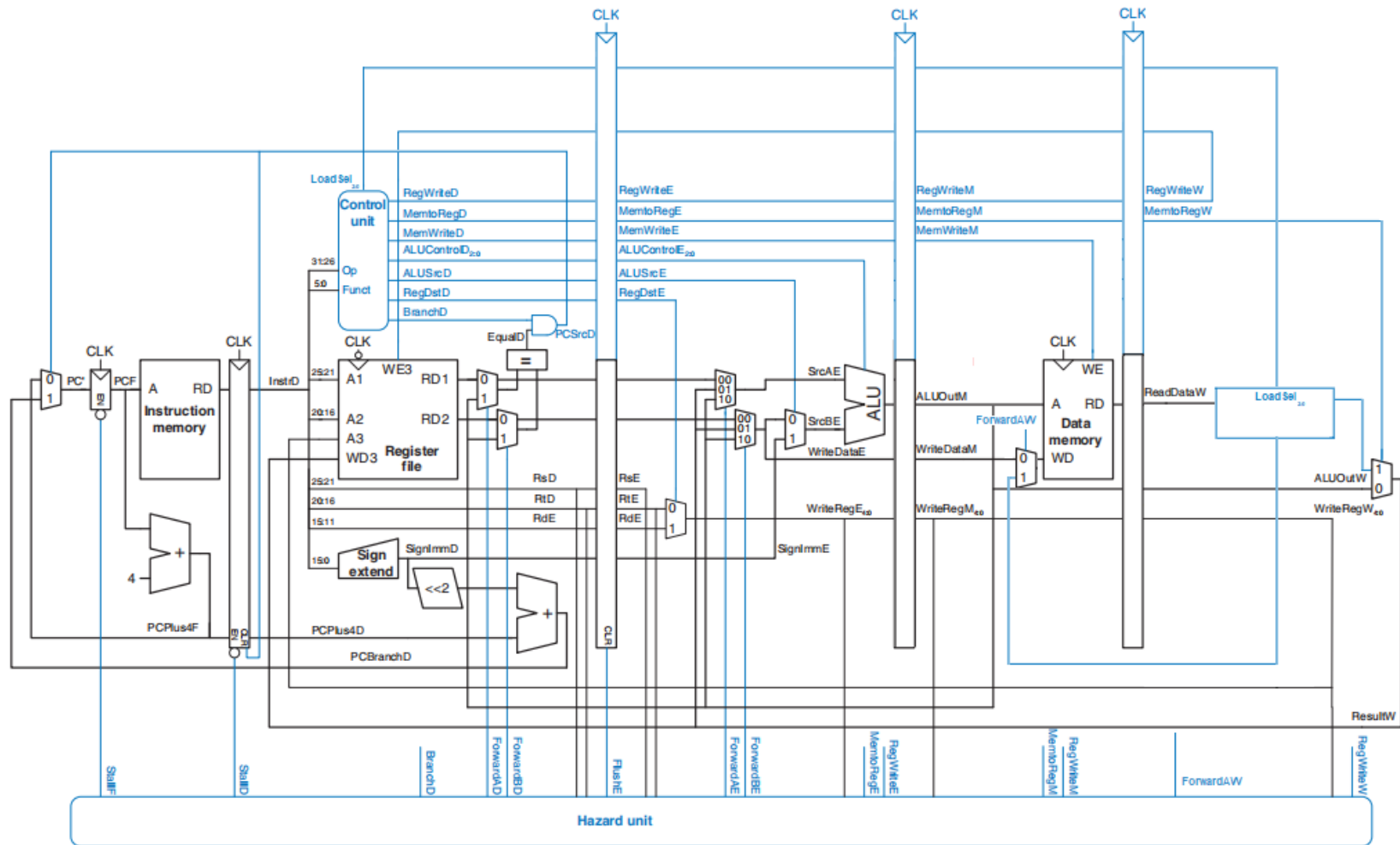
Load half and load byte :

To load a halfword from memory as a signed value

Assembly :

```
lh $storeReg  
imm($regReferringToMemAddress)
```

```
lb $storeReg  
imm($regReferringToMemAddress)
```



Load store hazard

- Store after load causes hazard
 - detection and solving in the previous slide

Components:

Items/Pins:

1. pin_b (byte) : used as a selector for mux[2]
2. half (half-word) : used as a selector for mux[1]
3. mux[1] (multiplexer): a multiplexer provide an option to full word or half word
4. mux[2] (multiplexer): a multiplexer provide an option to mux[1] or one byte

Store half and store byte :

introduction:

To store a halfword or byte to memory.

assembly:

```
sh $Registering value imm($regRefearingToMemAddress)
```

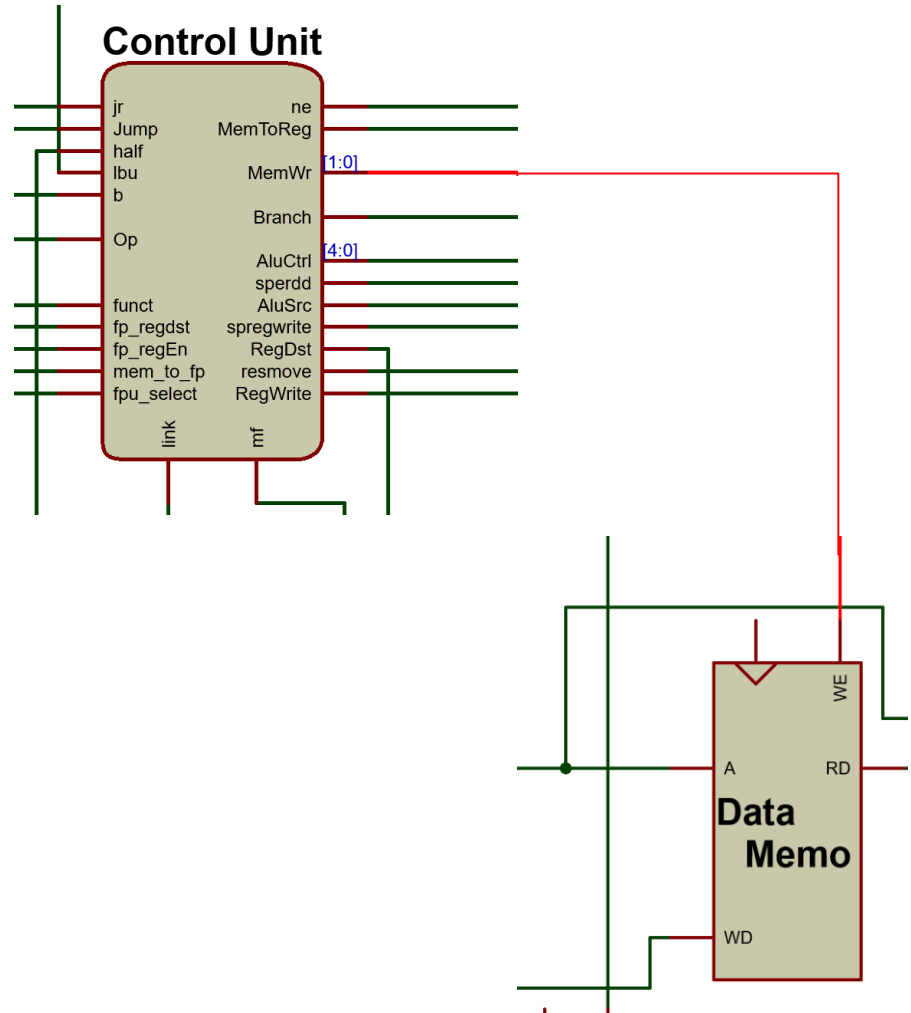
```
sb $Registering value imm($regRefearingToMemAddress)
```



implementation:

- make the MemWr pin 2 bits and WE pin also 2 bits,

in sw the alu result is address [32 bit] of the word and to move to the next word we shift the address left twice to add 4 so we always have 2 bits is 00 , we use this two bits to determined which number of bits in data



option (WE pin)	operation
0 0	don't care
0 1	store word ,RAM[a[31:2]] <= wd;
1 0	store half word , {a[1],4'b0000} uses the second LSB as an indicator to the upper or lower word starting point which is an intuitive approach to reach the half word
1 1	store byte , {a[1:0],3'b000} uses the first and second LSB as an indicator to the specified byte starting point which is an intuitive approach to reach the byte

DIV MOD DIVU MODU Divide Integers

- **Format:** DIV MOD DIVU MODU

DIV rd,rs,rt

MIPS32 Release 6

MOD rd,rs,rt

MIPS32 Release 6

DIVU rd,rs,rt

MIPS32 Release 6

MODU rd,rs,rt

MIPS32 Release 6

ORI Or Immediate

315 The MIPS32® Instruction Set Manual, Revision 6.06

Format: ORI rt, rs, immediate MIPS32

Purpose: Or Immediate

To do a bitwise logical OR with a constant.

Description: GPR[rt] ϕ GPR[rs] or immediate

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical OR operation. The result is placed into GPR *rt*.

Operations:

GPR[rt] ϕ GPR[rs] or zero extend(immediate)

Exceptions:

None

31 26 25 21 20 16 15 0

ORI 001101	<u>rs</u>	<u>rt</u>	immediate
---------------	-----------	-----------	-----------

6 5 5 16

ORI Or Immediate

JAL Jump and Link

The MIPS32® Instruction Set Manual, Revision 6.06 195

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch,

at which location execution continues after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256MB-aligned region.

The low 28 bits of the target address is the instr_index field shifted left 2bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

Operation:

I: \$ra \leftarrow PC + 4

I+1: PC \leftarrow PC

GPRLen-1..28 || instr_index || 02



JR Jump Register

Format: JR rs MIPS32

Assembly idiom MIPS32 Release 6

Purpose: Jump Register

To execute a branch to an instruction address in a register

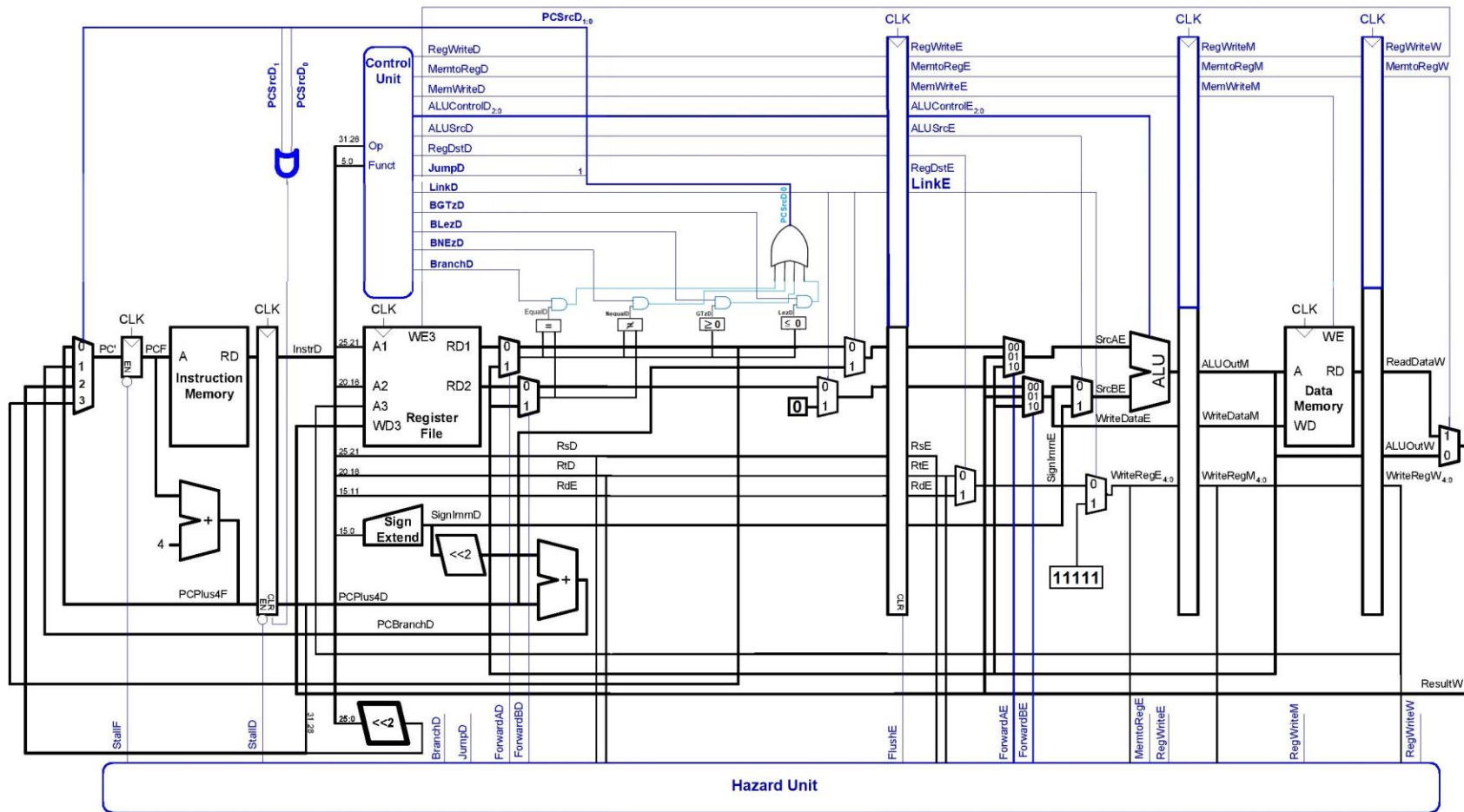
Description: $PC \leftarrow GPR[\text{rs}]$

Jump to the effective target address in GPR rs. Execute the instruction following the jump, in the branch delay slot, before jumping.

For processors that do not implement the MIPS16e or

Branches

- Format : BNE rs, rt, offse
- Format: BEQ rs, rt, offset
- Format: Blez rs
- Format: bgtz rs



MUL MUH MULU MUHU

- **Format:** MUL MUH MULU MUHU

MUL rd,rs,rt

MUH rd,rs,rt

MULU rd,rs,rt

MUHU rd,rs,rt

MIPS32 Release 6

MIPS32 Release 6

MIPS32 Release 6

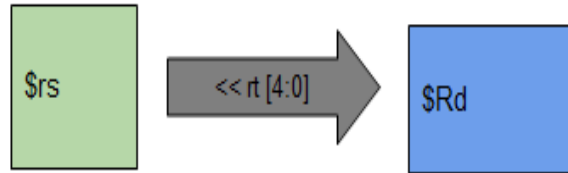
MIPS32 Release 6

Sllv Instruction:

sllv an (R type) instruction for shifting left a word by variable number

Assembly:

```
sllv rd, rs, rt
```



Operation:

sllv would shift the value in reg(rs), by a number stored in low five bits in reg(rt), saving result in reg(rd)

Sllv Instruction:

Implementation:

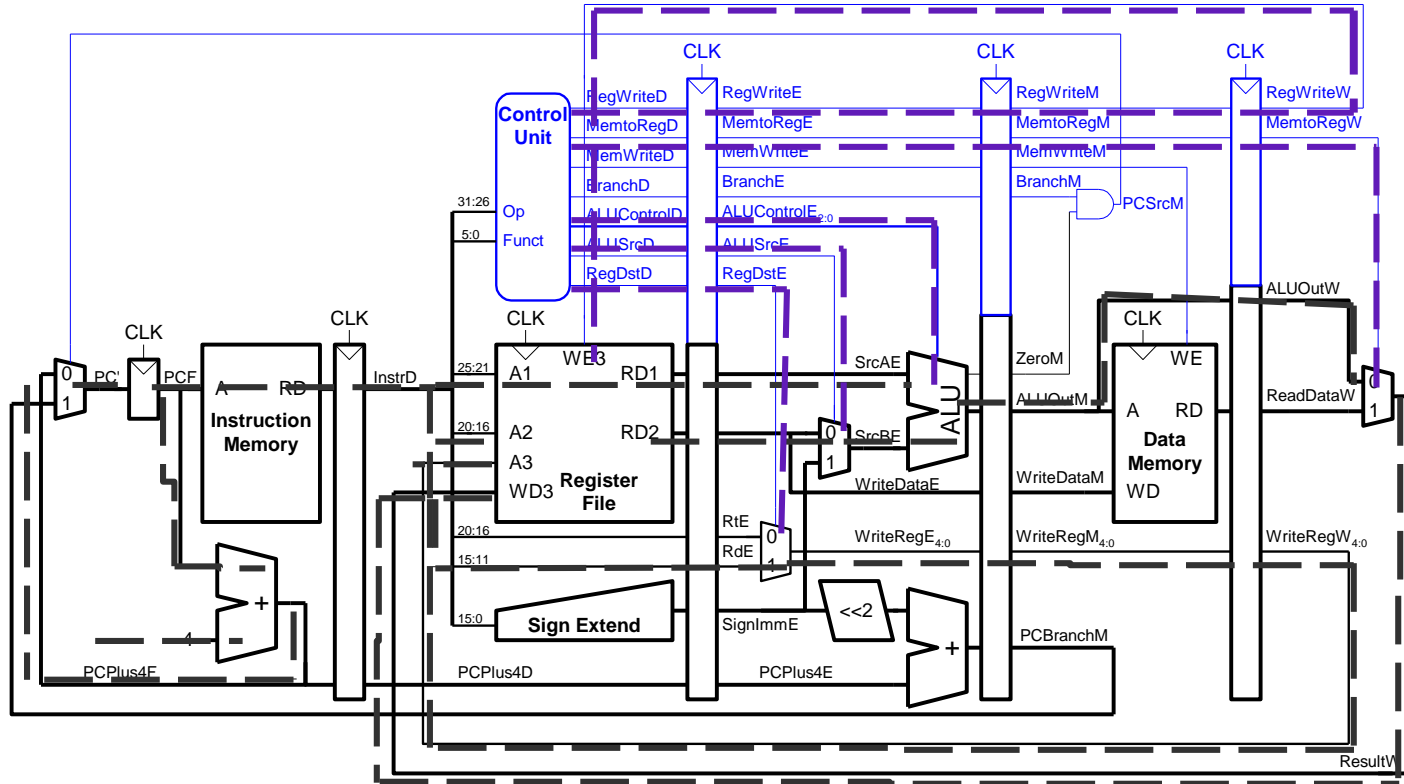
18 5'b01?0?: result = b << shift; //sll 01000/sllv 01001
1-Alu:

Add operation

20 6'b000100: alucontrol <= `EXE_SLLV; //sllv
2-Aludec:

Function : 000100

SLLV Instruction :



Srl Instruction:

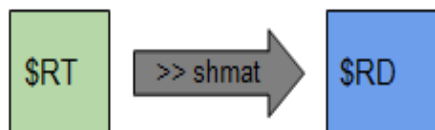
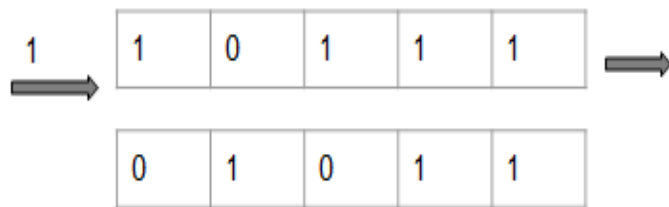
```
srl $rs $rt shift
```

Shift logic operation:

Shifting is to move the 32 bits of a number

Shift right logic:


Insert zero from left



Srl Instruction:

Implementation:

1- We would need to branch Instruction[10:6](the shift amount)

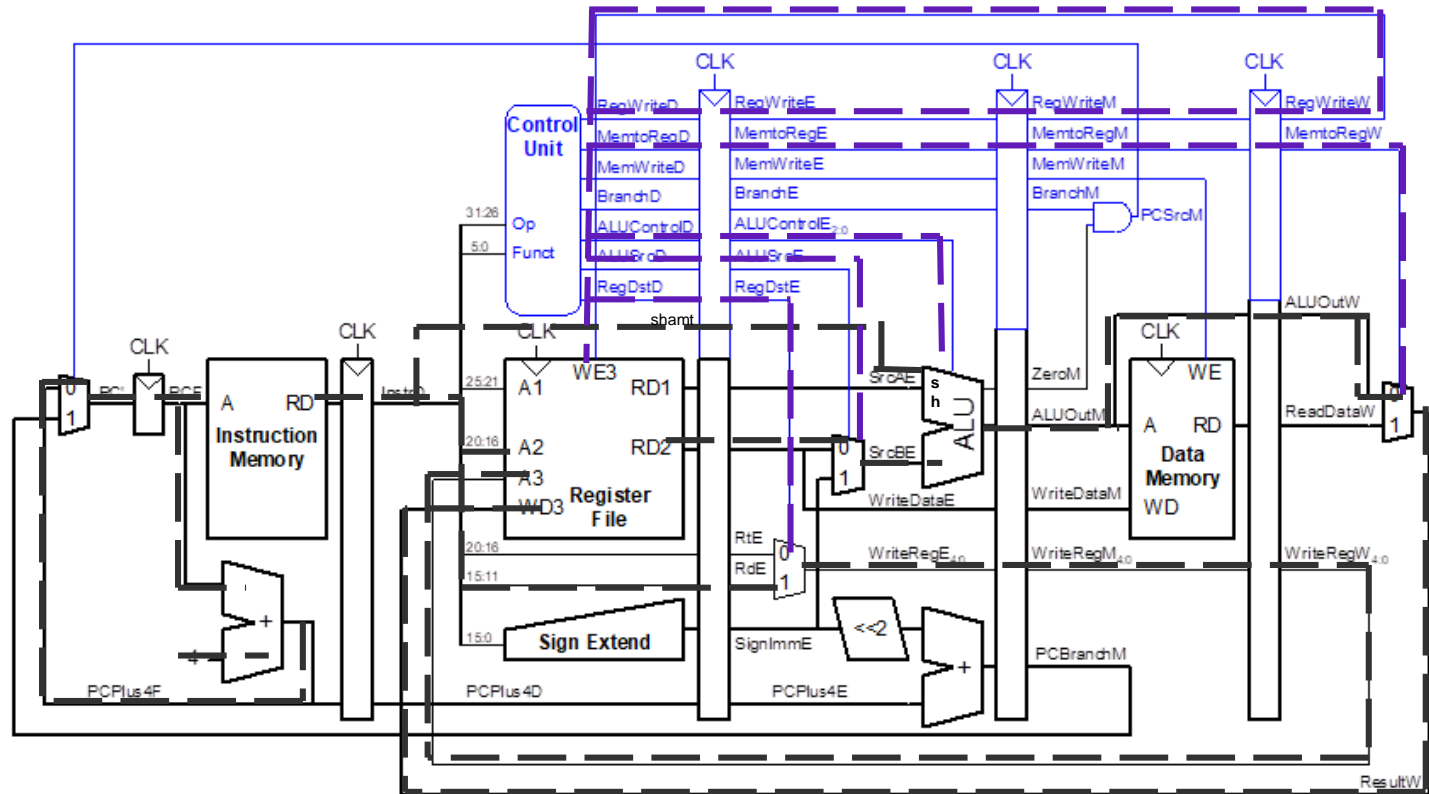
2-Alu:  19 5'b01?1?: result = b >> shift; //srl 01010

Add operatio 18 6'b000010: alucontrol <= `EXE_SRL; //srl

3-Aludec:

Function :000010

Srl Instruction:



Sra Instruction:

- R-type instruction with funct=3

Assembly:

```
sra $rd, $rt, $shamt
```

functionality:

Shift to the right arithmetically **rt** by **shamt**
and store the result in **rd**



Sra Instruction:

Implementation:

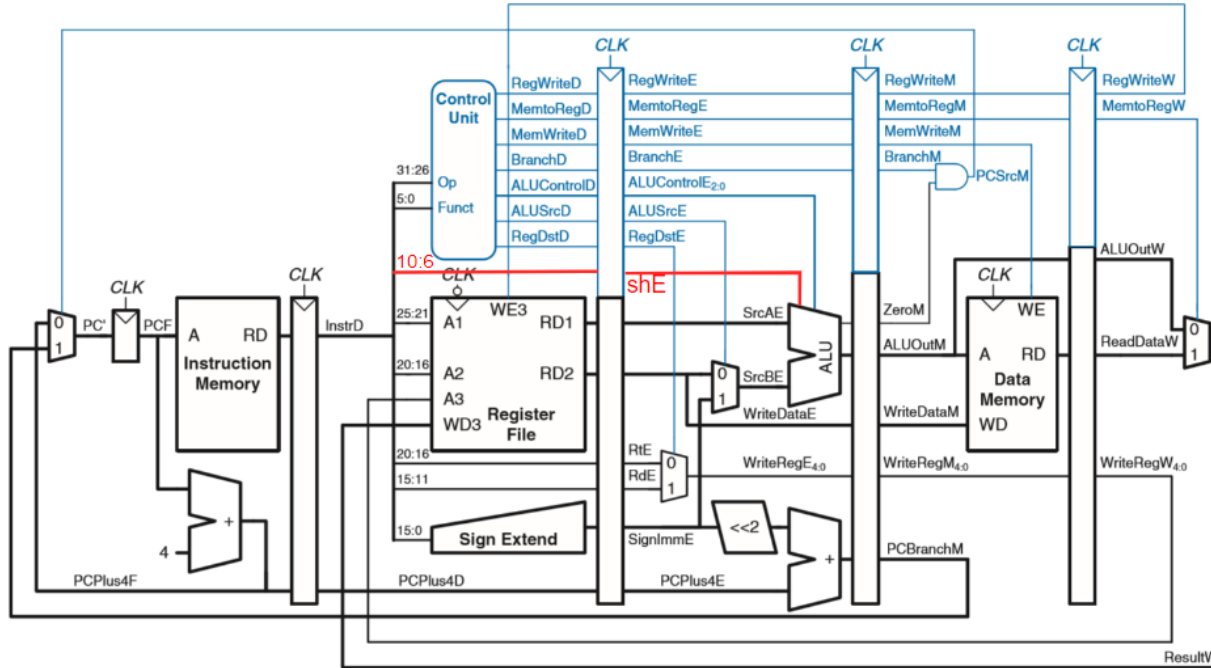
1-We would need to branch Instruction[10:6] (the shift amount)

2- Alu: `20 5'b11?1?: result = b >>>shift; //sra 11010`

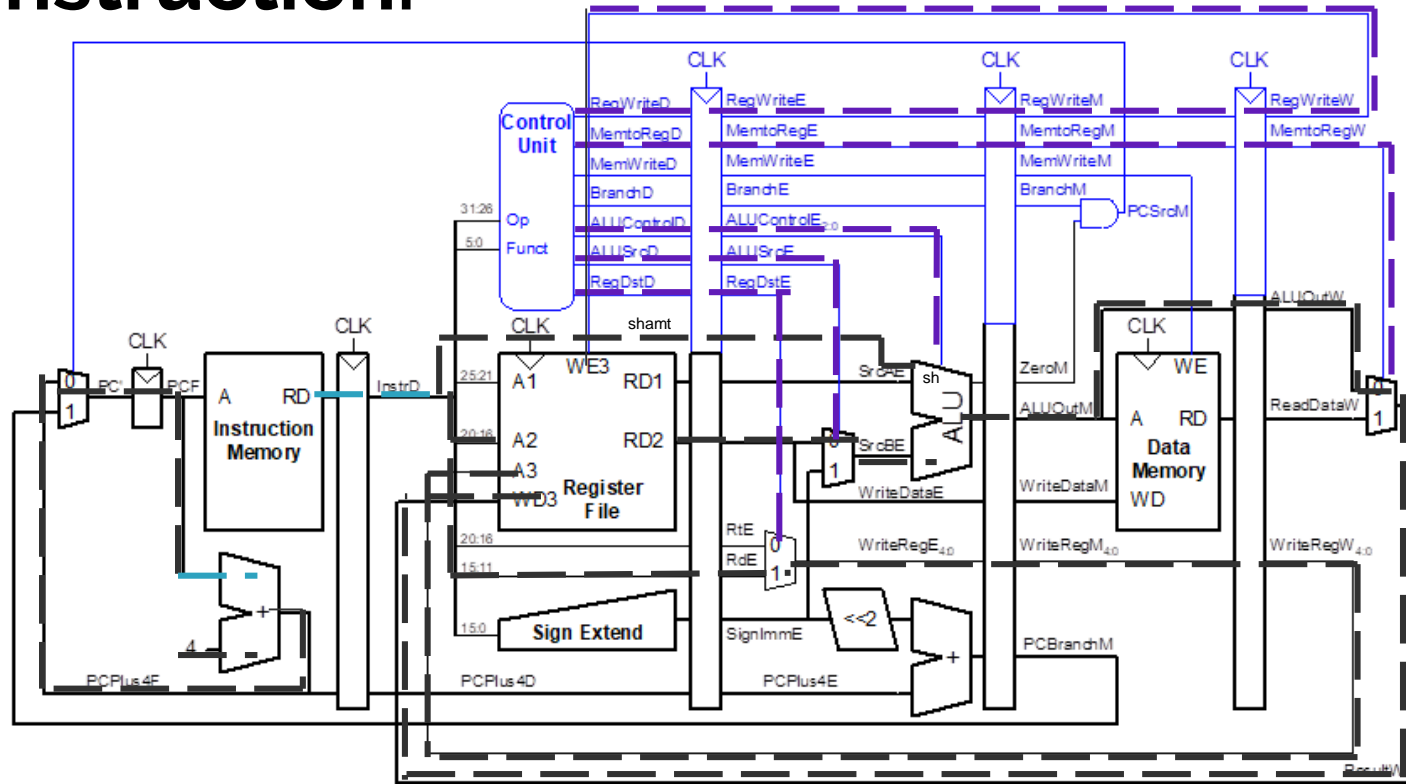
Add operation

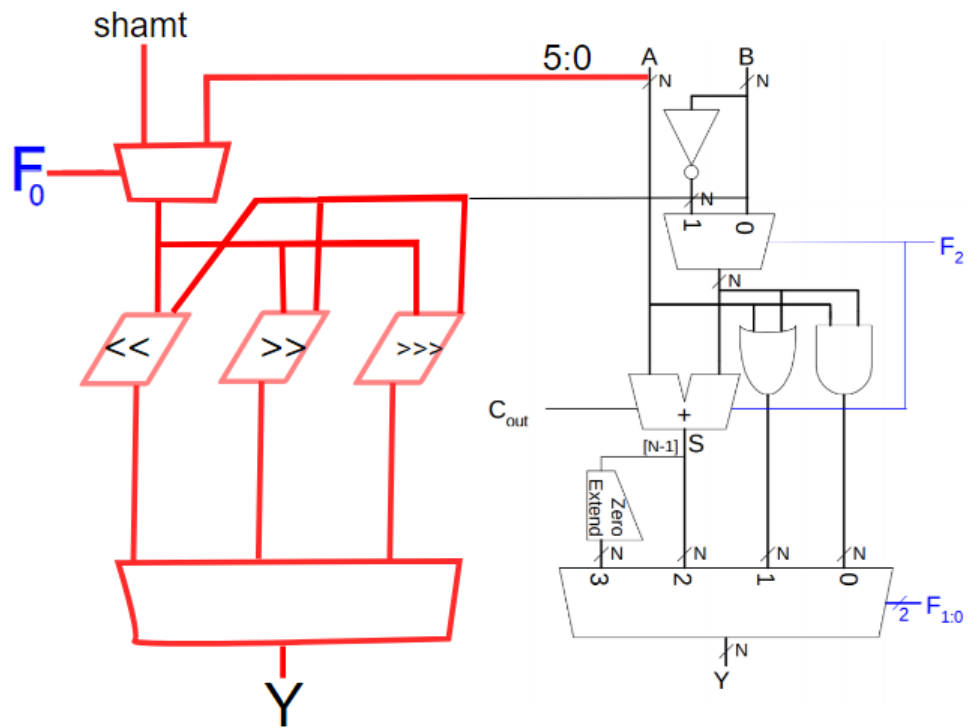
- 3-Aludec: `19 6'b000011: alucontrol <= `EXE_SRA; //sra`
- Function:000011

Datapath edits



Sra Instruction:



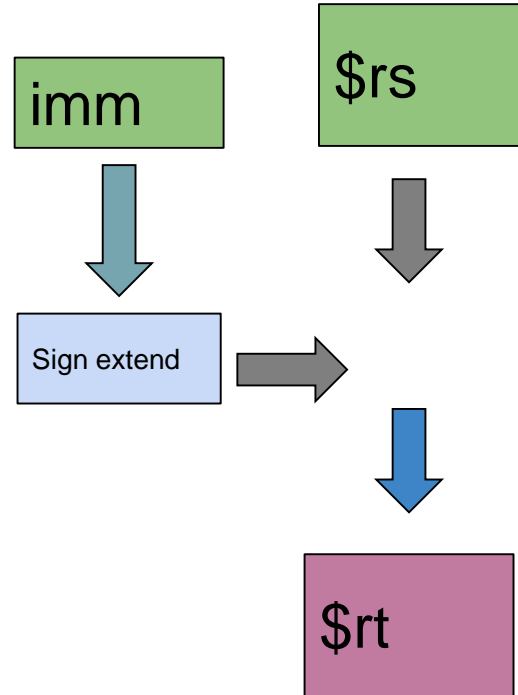


Andi

The andi instruction performs the bitwise and of the contents of Rt and sign extended value of imm and store the result in the register Rs.

Assembly :

```
andi $rt, $rs, imm
```

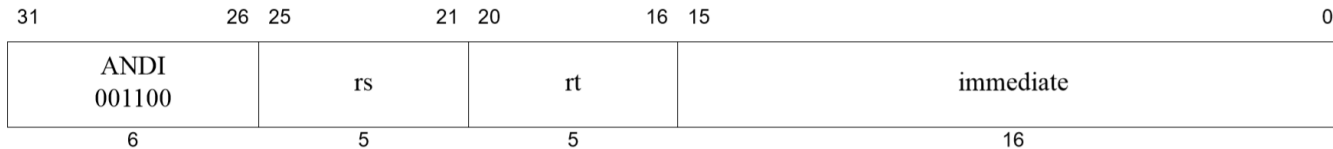
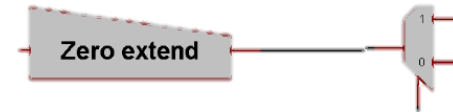


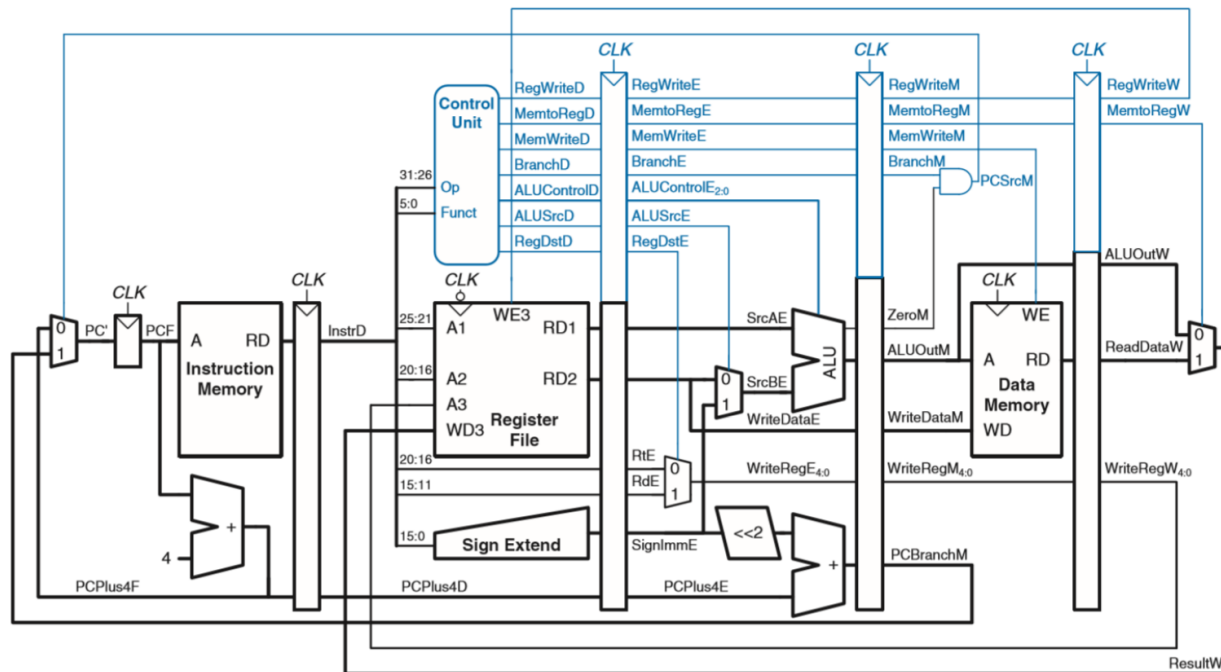
Implementation :

Alu:

- changing ALUSel(Alu op)

function operation
001100 $y = a + \text{zero_extend}(\text{immd})$;





addiu

- To add a constant to a 32-bit integer.

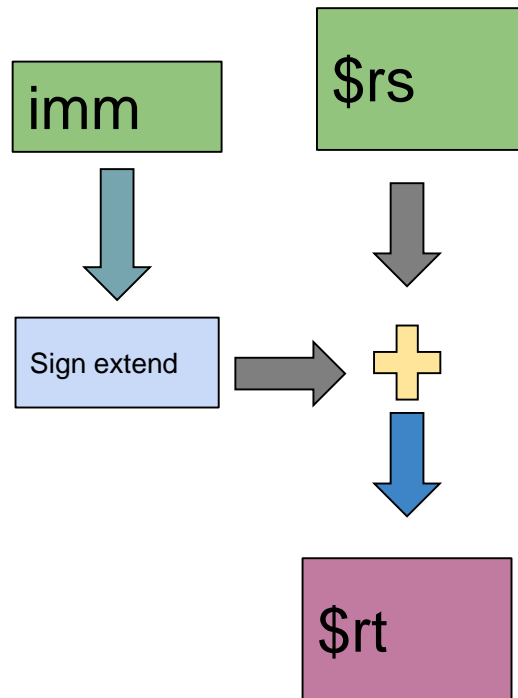
Assembly:

```
addiu $rt, $rs, imm
```

Functionality :

Add **\$rs** to sign extended **immed**

Store result in **\$rt**



Note

- The term “unsigned” in the instruction name is a misnomer;
- this operation is 32-bit modulo arithmetic that does **not** trap on **overflow**.

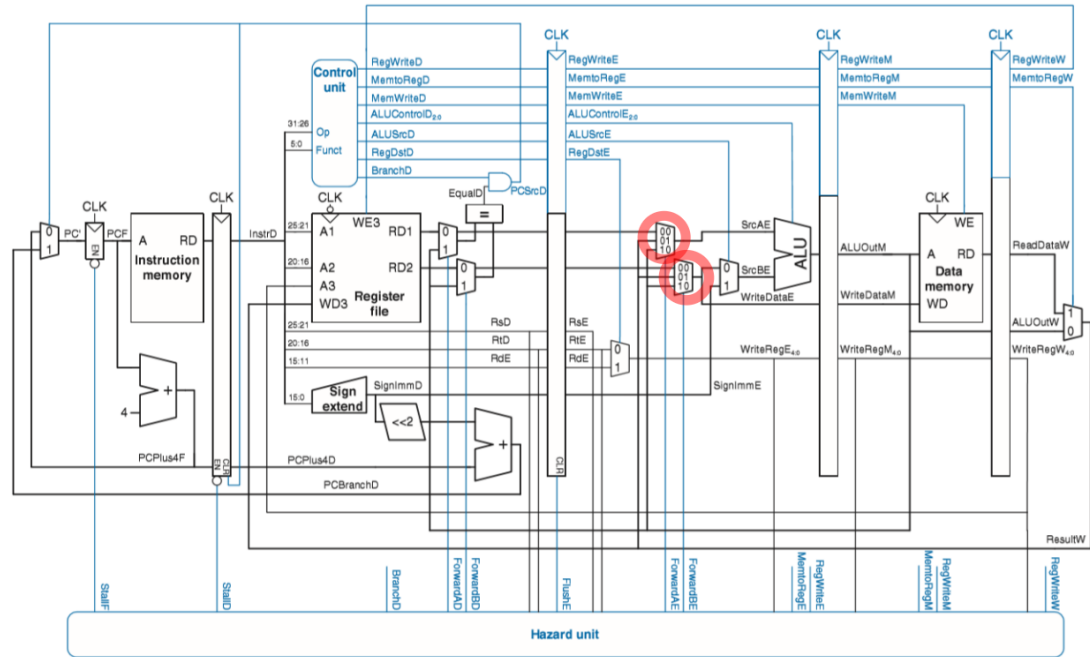
This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that **ignore** overflow, such as **C** language arithmetic.

implementation:

addiu is identical to **addi** so **no** required edit in datapath

Just assign controls for opcode

instruction	andiu
REgWrite	1
RegDST	0
ALUSrc	01
Branch	0
MemWrite	0
MemtoReg	0
Jump	0
jr	0
aluop	0000



```

16 always_comb
17     case(r)
18         0'b000000: y=0b0;
19         0'b000001: y=a|b;
20         0'b000010: y=a&b;
21         0'b000011: y=(b<(<shamt)); //sll
22         0'b000100: y=a+(-b);
23         0'b000101: y=a+(<b);
24         0'b000110: y=a&b;
25         0'b000111: //slt
26         begin
27             y=a-b;
28             y=y[31]?b1:b0;
29         end
30         0'b010000: y = (b<0); //lwi
31         0'b010001: y = 0'b0; //nop
32         0'b010010: begin //bltz
33             if((a[31]==1)|(a==0))
34                 y=0;
35             else
36                 y=1;
37             end
38         end
39         0'b010011: y = ( b >> (a[4:0]) ); //srlv
40         0'b011000: y=(b>>shamt); //srl
41         0'b011001: y = ( Signed(b) >>> (a[4:0]) ); //sra
42         0'b011010: y = a&b; //slilu
43         0'b011011: y = ( b << (a[4:0]) ); //srlv
44         0'b100000: begin //bgtz
45             if (a>0)
46                 y=1;
47             else
48                 y=0;
49             end
50         end
51         0'b100001: y=(b>>>shamt); //sra
52         0'b100010: y = ( b << (a[4:0]) ); //sllv
53         0'b100011: result = Signed(a) * Signed(b); //mult
54         0'b101001: result = a * b; //multu
55         0'b10110: begin result[31:0] <= Q; result[63:32] <= R; end //div
56         0'b10111: begin result[31:0] <= Q1; result[63:32] <= R1; end //divu
57
58     default: y=0;
59 endcase
60

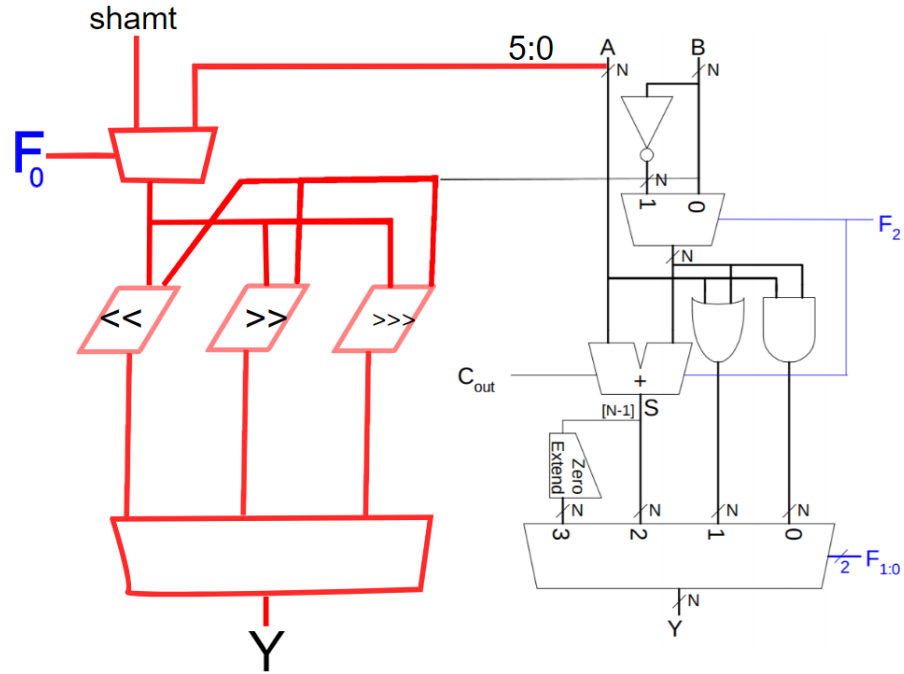
```

```

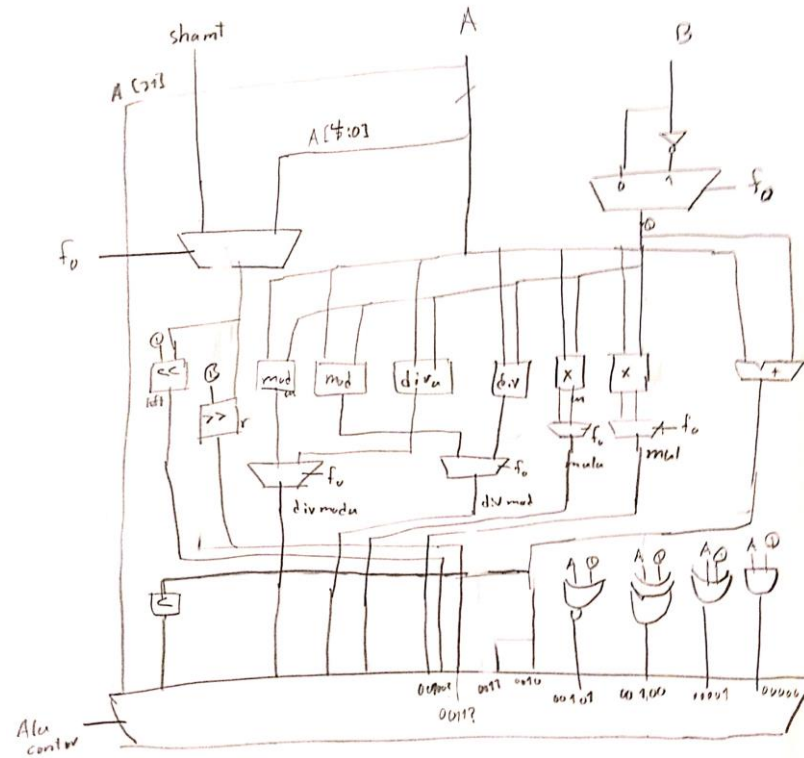
1 module alu(input logic [31:0] a, b,
2 + input logic [4:0] shamt,
3 + input logic [4:0] alucontrol,
4 output logic [31:0] result);
5
6 logic [31:0] bout, sum;
7 + logic [4:0] shift;
8 + assign bout = alucontrol[2] ? ~b : b; //for subtraction set alucontrol[2] to 1
9 assign sum = a + bout + alucontrol[2];
10 + assign shift = alucontrol[0] ? shamt : a[4:0]; //for shift variable alucontrol[0]
    = 0
11
12 always_comb
13 + case (alucontrol) inside
14 + 0'b000000: result = a & bout; //and 000000/
15 + 0'b000001: result = a | bout; //or 000001
16 + 0'b000010: result = sum; //sub 000110/add 000010
17 + 0'b000011: result = sum[31]; //slt
18 + 0'b000100: result = b << shift; //sll 010000/sllv 010001
19 + 0'b000101: result = b >> shift; //srl 010100
20 + 0'b000110: result = b >>>shift; //sra 110100
21 endcase
22

```

alu with shift



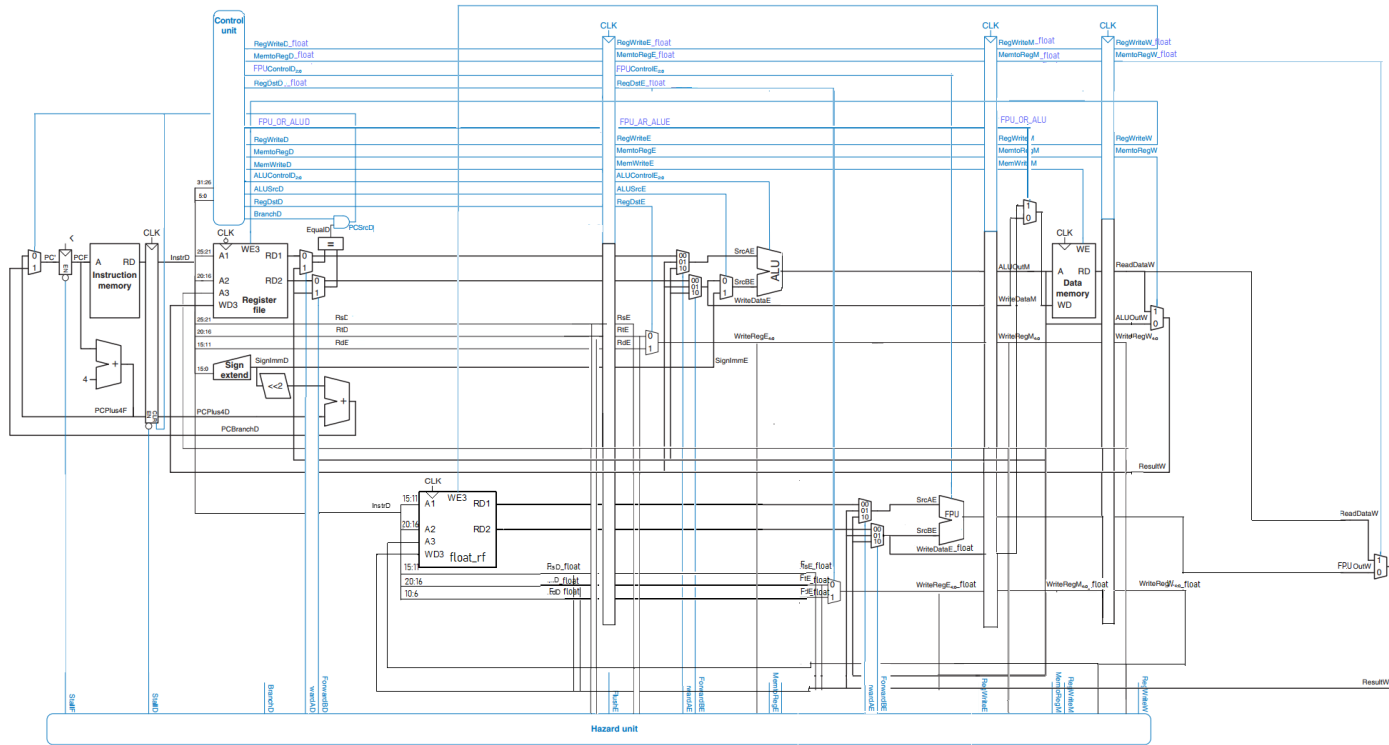
full alu



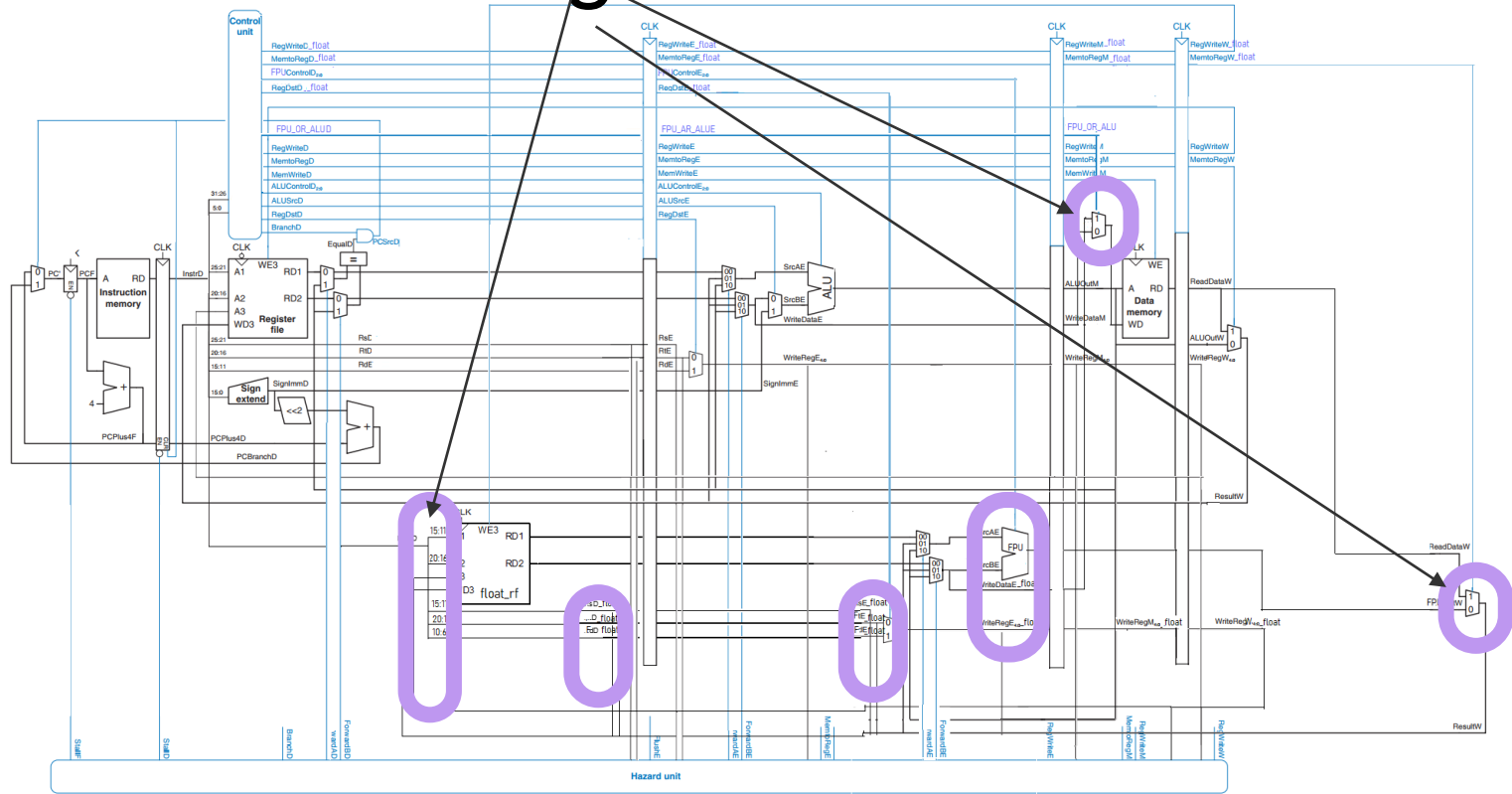
Floating point instructions :

- Register file with 32 registers added to data path with FPU
- We repeated same control signals for hazard for floating instructions
- Control signals added :

<i>Signal</i>	<i>Description</i>
FPU control	select operation of FPU
float_reg_dst	choose reg to write
float_regwrite	write to fpu reg
mem_to_fpu	write from memory to fpu Reg
Fpu_mem_write(fpu or alu	write from fpu to memory



Changes added



F-type

op	cop	ft	fs	fd	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Table B.3 F-type instructions (fop = 16/17)

Funct	Name	Description	Operation
000000 (0)	add.s fd, fs, ft / add.d fd, fs, ft	FP add	$[fd] = [fs] + [ft]$
000001 (1)	sub.s fd, fs, ft / sub.d fd, fs, ft	FP subtract	$[fd] = [fs] - [ft]$
000010 (2)	mul.s fd, fs, ft / mul.d fd, fs, ft	FP multiply	$[fd] = [fs] \times [ft]$
000011 (3)	div.s fd, fs, ft / div.d fd, fs, ft	FP divide	$[fd] = [fs] / [ft]$
000101 (5)	abs.s fd, fs / abs.d fd, fs	FP absolute value	$[fd] = ([fs] < 0) ? [-fs] : [fs]$
000111 (7)	neg.s fd, fs / neg.d fd, fs	FP negation	$[fd] = [-fs]$
111010 (58)	c.seq.s fs, ft / c.seq.d fs, ft	FP equality comparison	fpcond = $([fs] == [ft])$
111100 (60)	c.lt.s fs, ft / c.lt.d fs, ft	FP less than comparison	fpcond = $([fs] < [ft])$
111110 (62)	c.le.s fs, ft / c.le.d fs, ft	FP less than or equal comparison	fpcond = $([fs] \leq [ft])$

We implemented **single precision arithmetic instructions** as following :

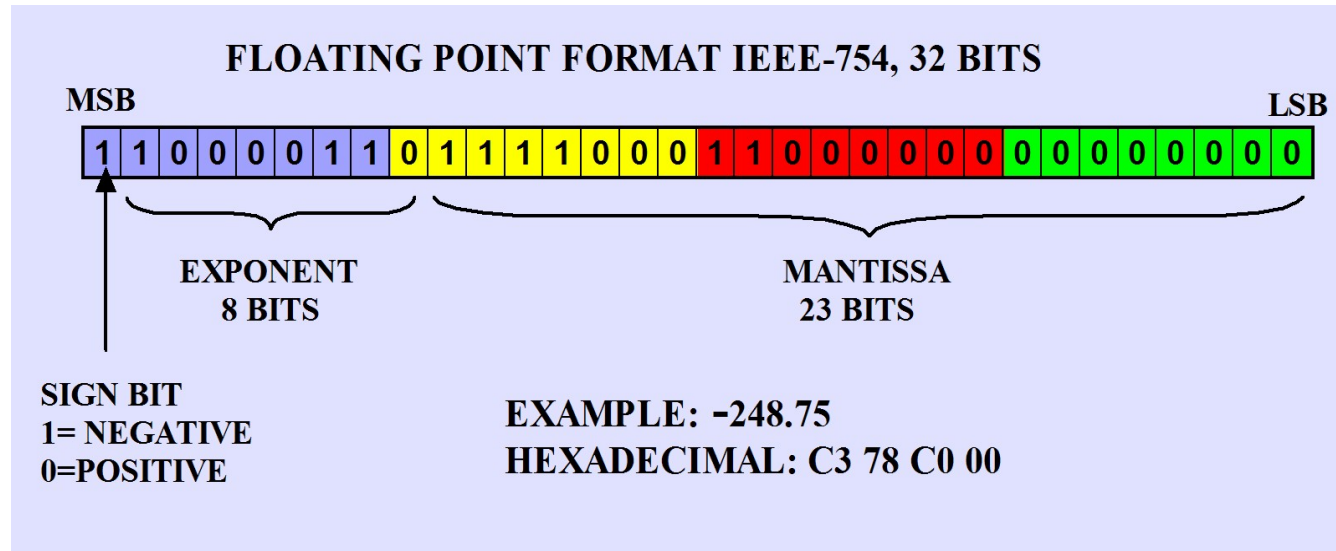
Other control signals are set to 0, operation performed as fpu_selector is set by controller and signals related to fpu are set as following

Fpu_reg_dst	FPU_regregwtite	mem_to_fpu
0	1	0

Fpu selector	Fpu operation	Function
0	$fs+ft$	0
1	$fs-ft$	1
2	$fs*ft$	2
3	fs/ft	3
7	$-fs$	5
5	$ fs $	7

FP negation : only we need complement the MSB of rs in FPU

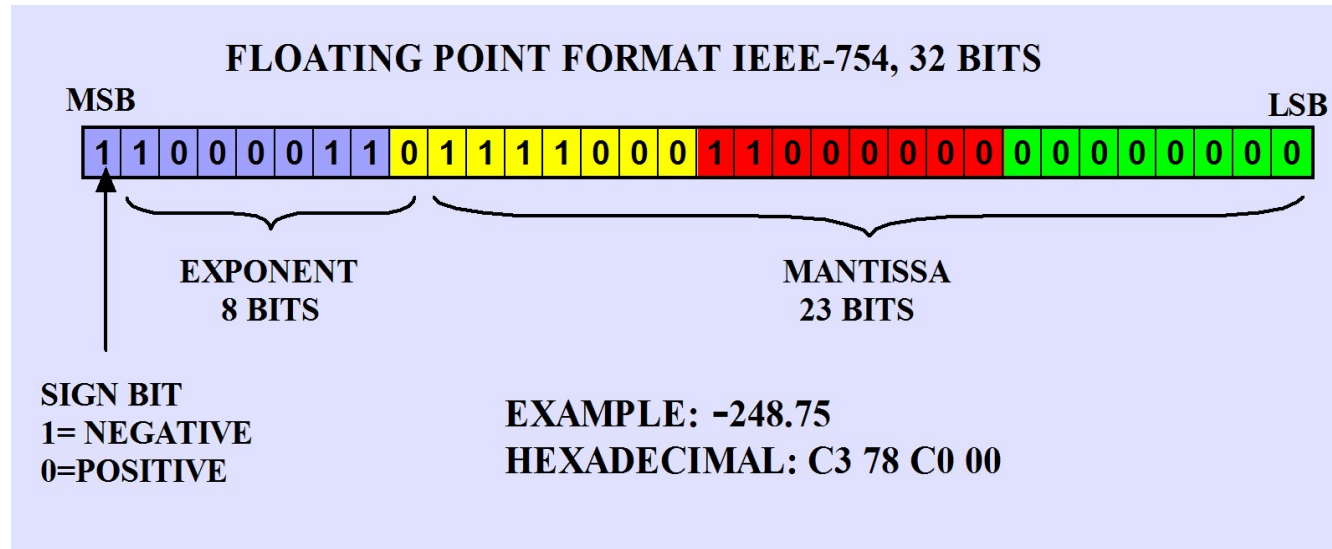
neg.s fd, fs



FPU_out={~rs[31],rs[30:0]}

FP absolute : only we need set the MSB of rs in FPU to zero

`abs.s fd, fs`



`FPU_out={0,rs[30:0]}`

instr	Fpu_reg_dst	FPU_regEn	mem_to_fpu	fpu_memWrite	Alusrc	memWrite
Lwc1.s	0	1	1	0	1	0
Swc1.s	0	0	0	1	1	1