南通大學
NANTONG UNIVERSITY

Course Design of Computer Principle of Organization

2024-2025-1


南通大學
NANTONG UNIVERSITY
Course Design
Design your MIPS Single-Cycle CPU

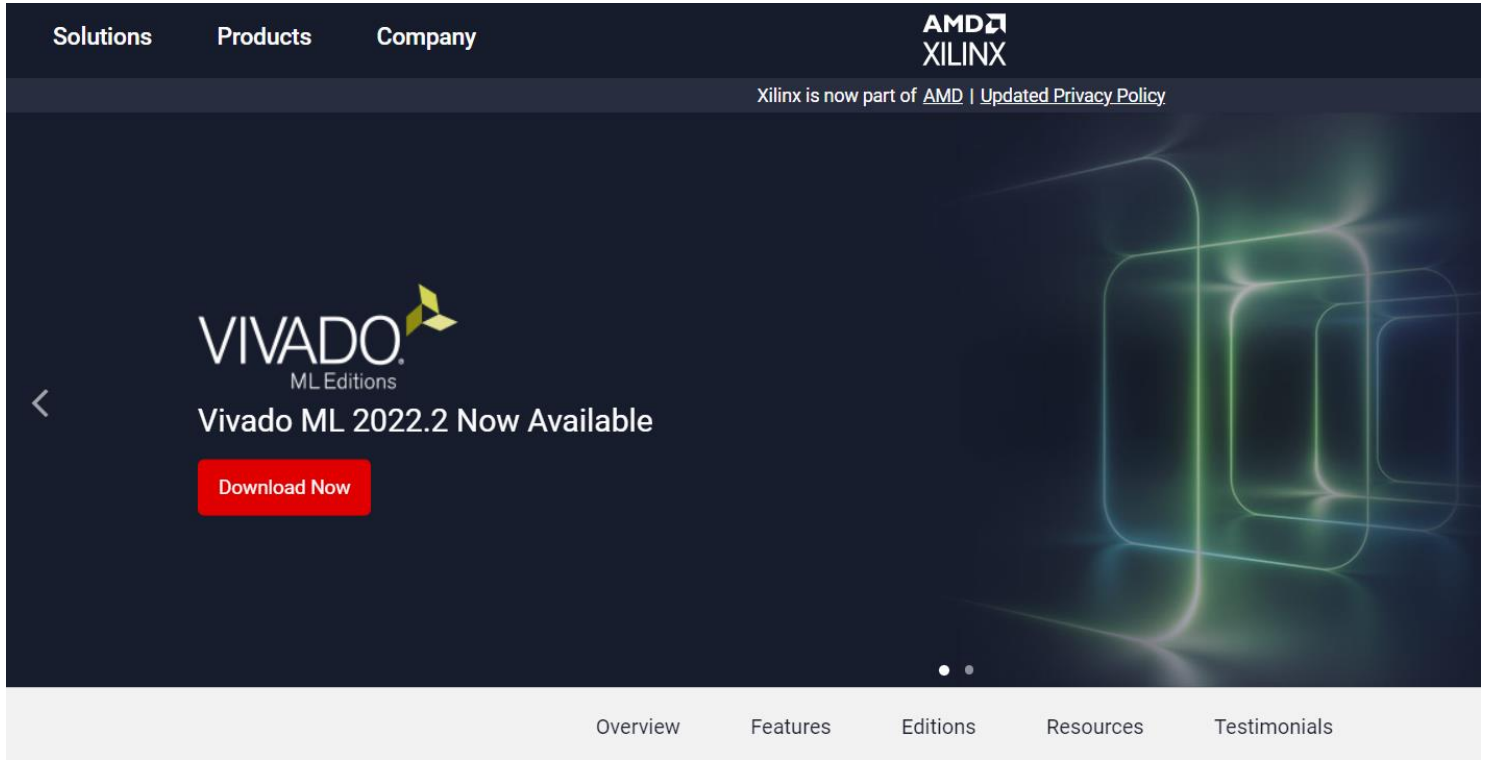
Experimental environment

Software: VIVADO

Download: https://www.xilinx.com/products/design-tools/vivado.html

Programming language : Verilog HDL

Link: https://www.verilog.com/

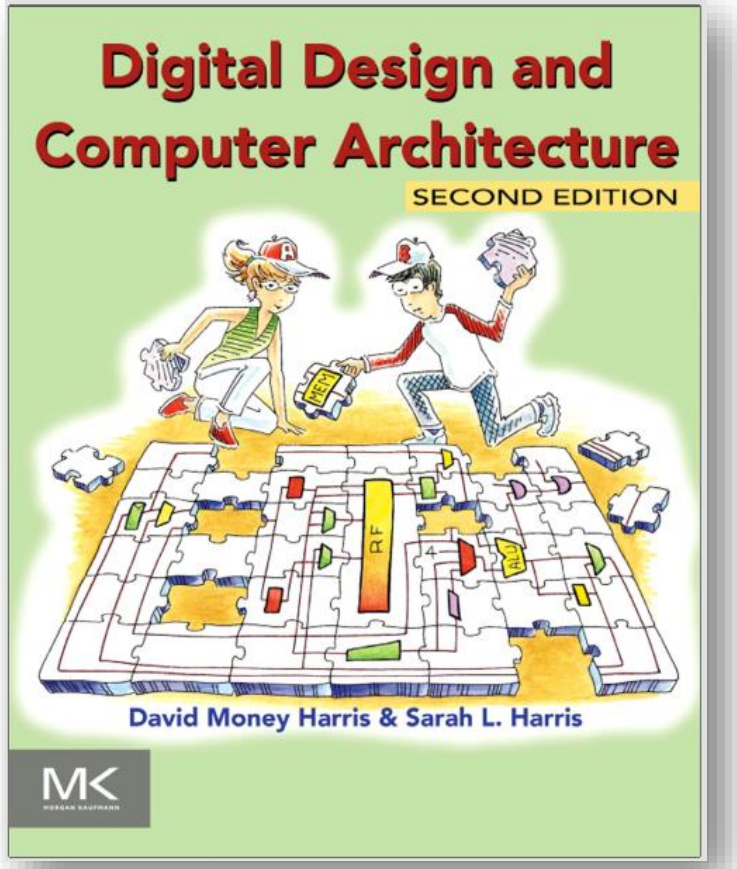


Reference

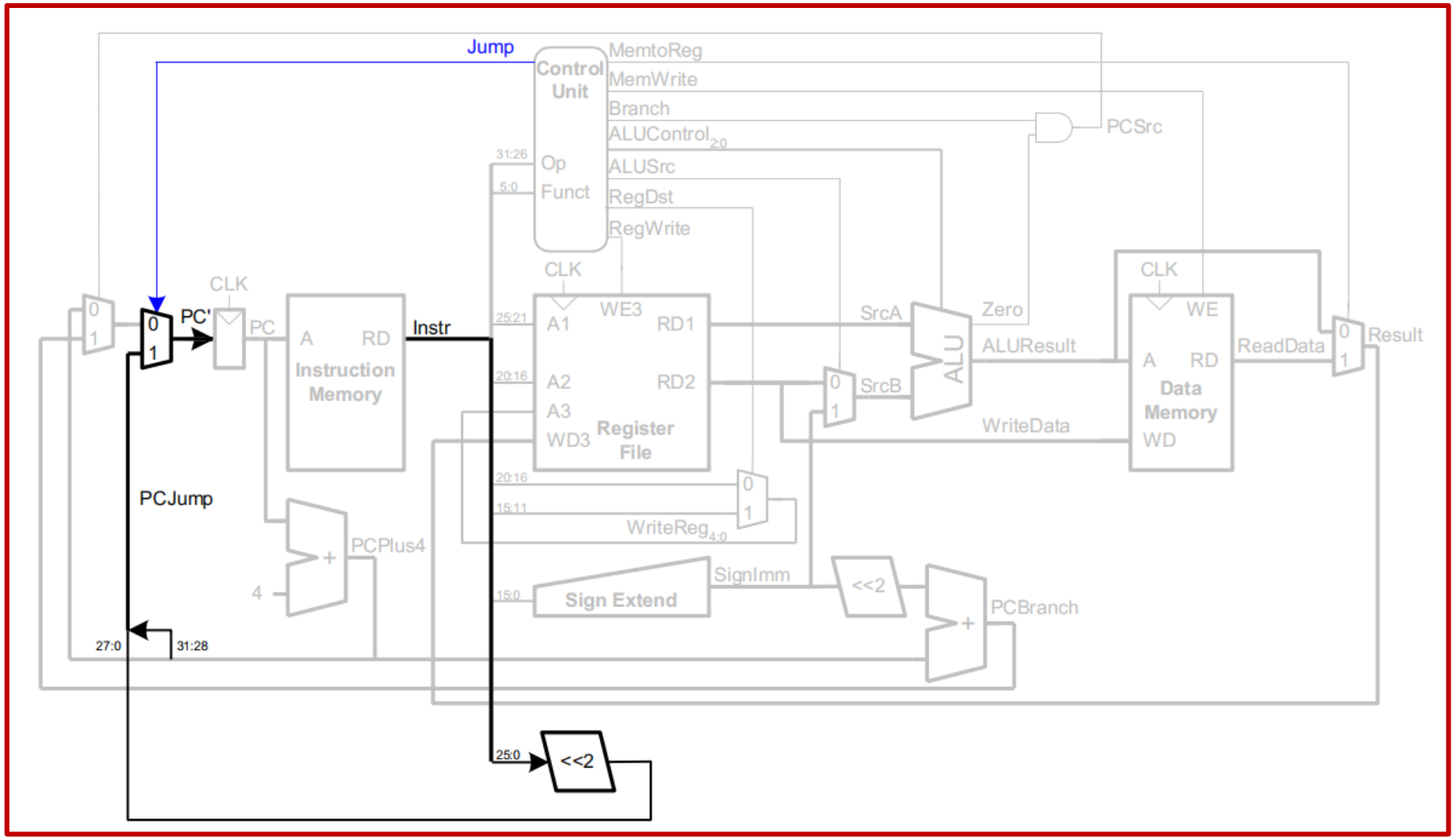
Harris D, Harris S L. **Digital design and computer architecture (2nd Ed**) [M].

- **MIPS ISA** (Instruction set architecture) (see chap 6, p295)
- **MIPS Single-Cycle Architecture** (see chap 7.3, p376-389)

Download:
https://github.com/stevemac321/pdfs/blob/master/Digital%20Design%20and%20Computer%20Architecture%20(2nd%20Ed)(gnv64).pdf



MIPS Single-Cycle CPU



MIPS Single-Cycle CPU

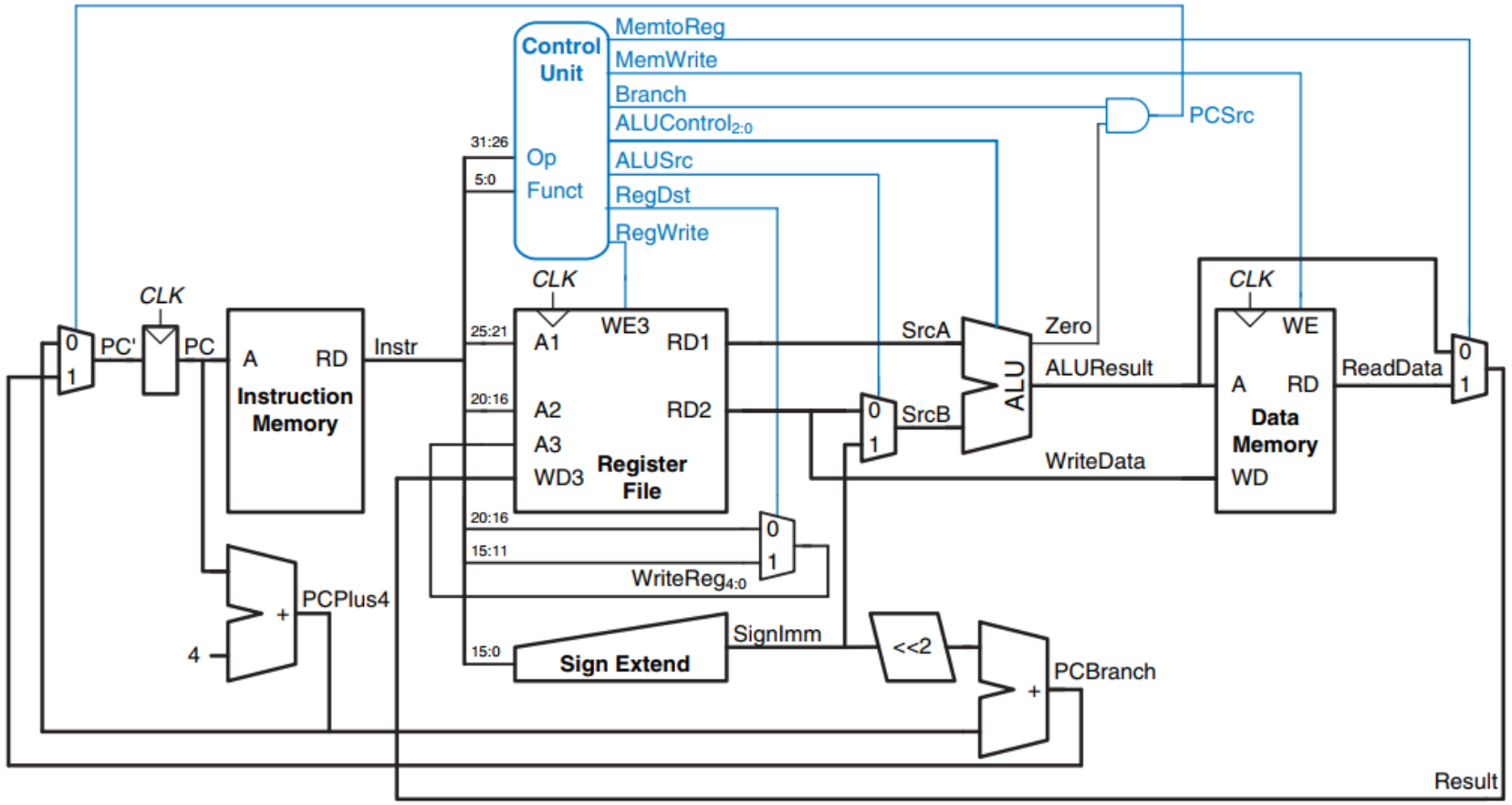


Figure 7.11 Complete single-cycle MIPS processor

Step 1: Build each individual modules

Step 2: Build DataPath

Step 3: Build MIPS-CPU

Step 4: Simulation

南通大學
NANTONG UNIVERSITY

Step 1 Build each individual module

MIPS Single-Cycle CPU

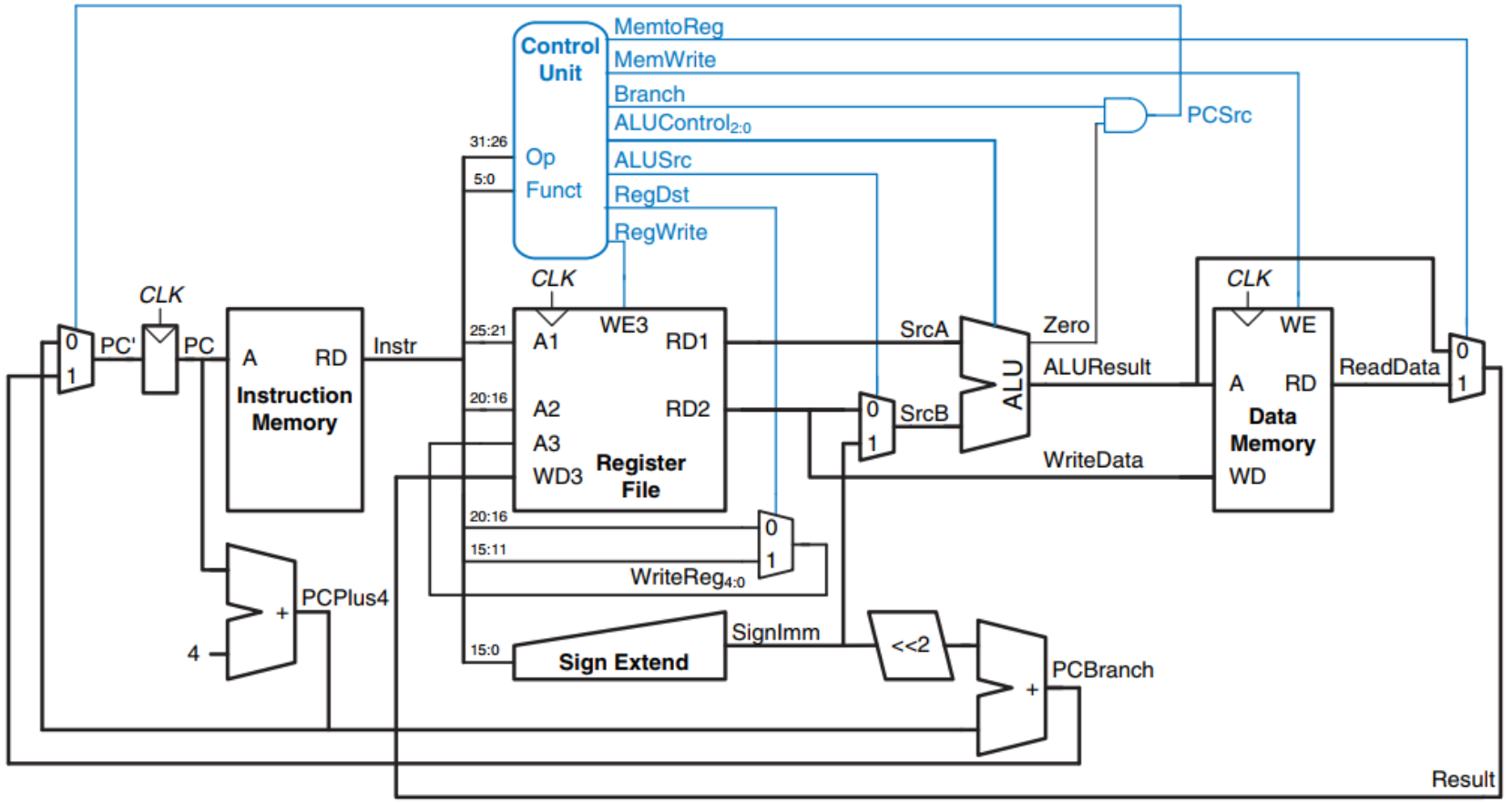


Figure 7.11 Complete single-cycle MIPS processor

Modules:

1. Register file
2. ALU
3. <<2: Shifter
4. Adder
5. PC
6. MUX-2
7. Instruction Memory
8. Data Memory
9. Control unit
10. Sign extend

MIPS Single-Cycle CPU

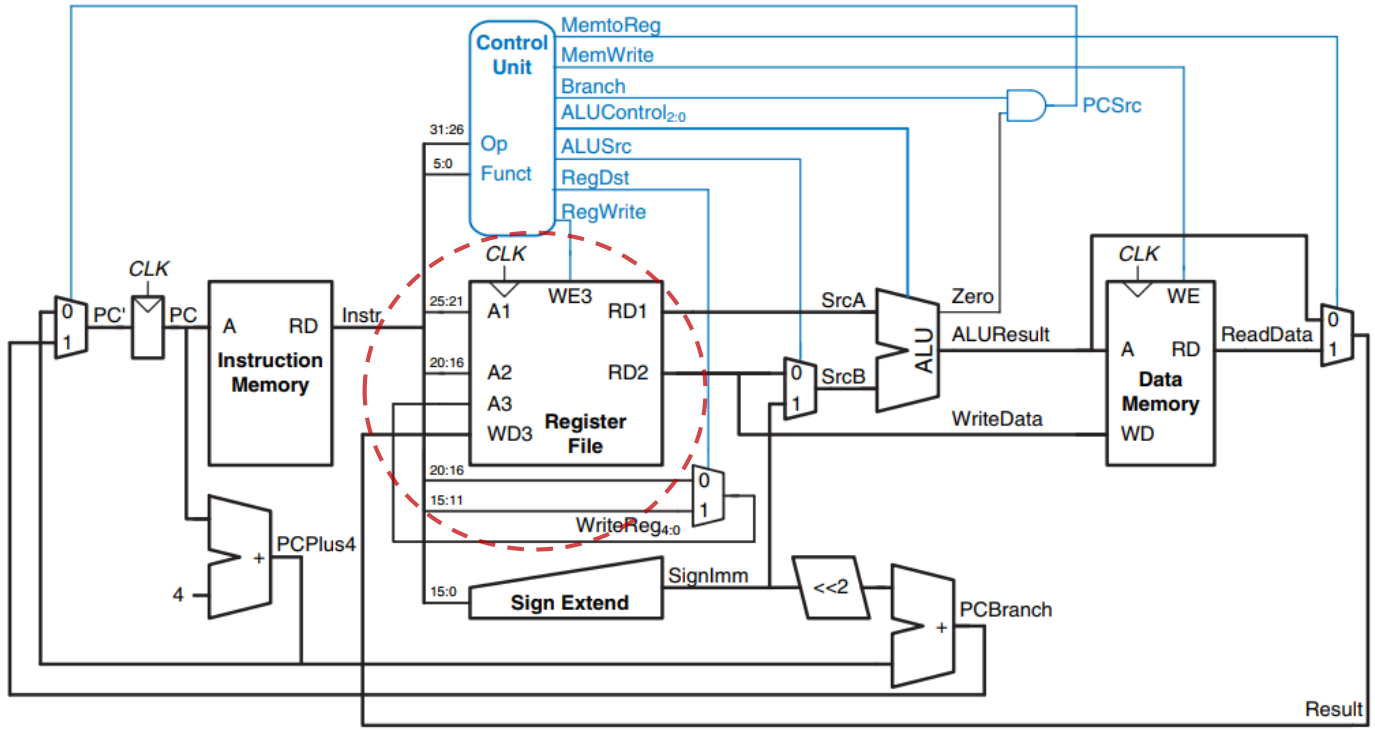


Figure 7.11 Complete single-cycle MIPS processor

Modules:

1. **Register file:**
   - Support two-read and one-write.
   - Read: address A1,A2, output data on RD1,RD2.
   - Write: set RegWrite=1, WE(write enable) signal is valid, with **clk** signal, address A3, input data from WD3.

MIPS Single-Cycle CPU

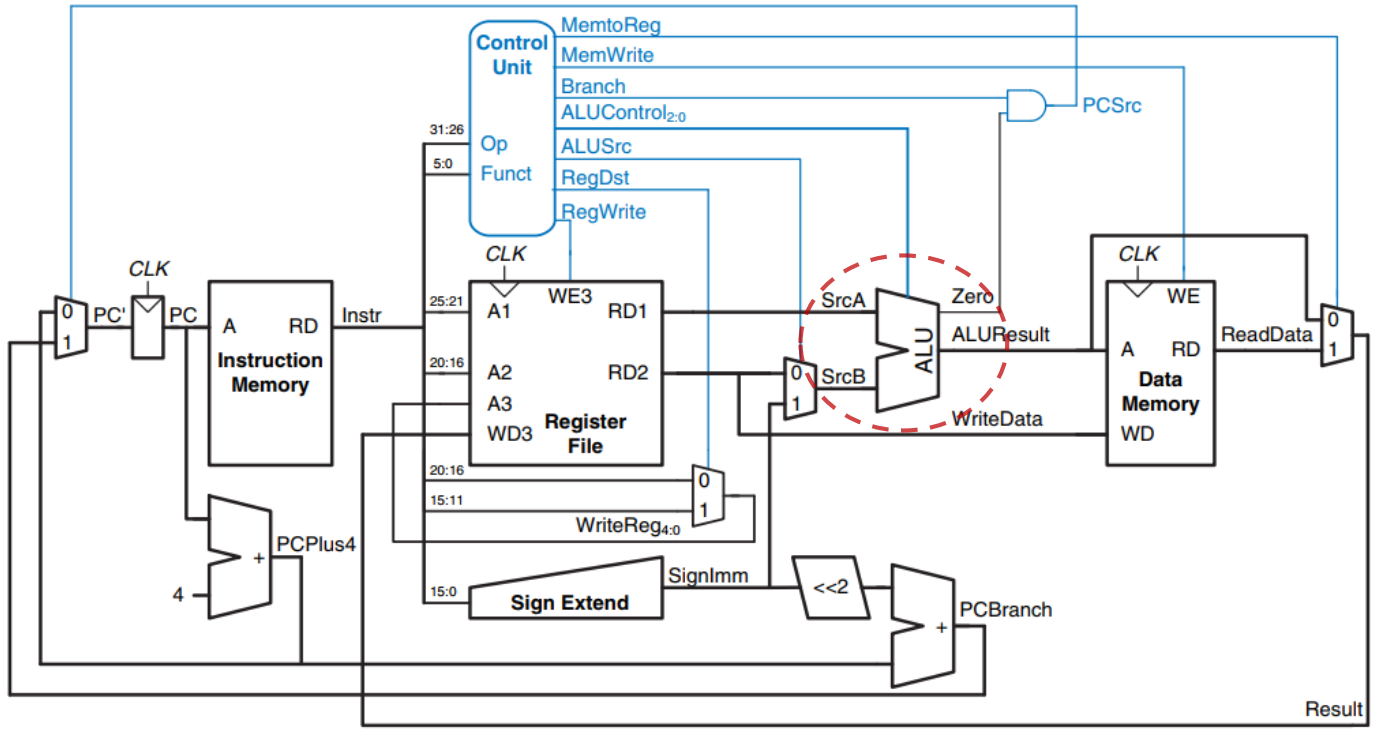


Figure 7.11 Complete single-cycle MIPS processor

Modules:

2. ALU:

- ALUControl: Specify operation type.
- **2:0**, ALU supports a maximum of 2^3=8 operation types.

MIPS Single-Cycle CPU

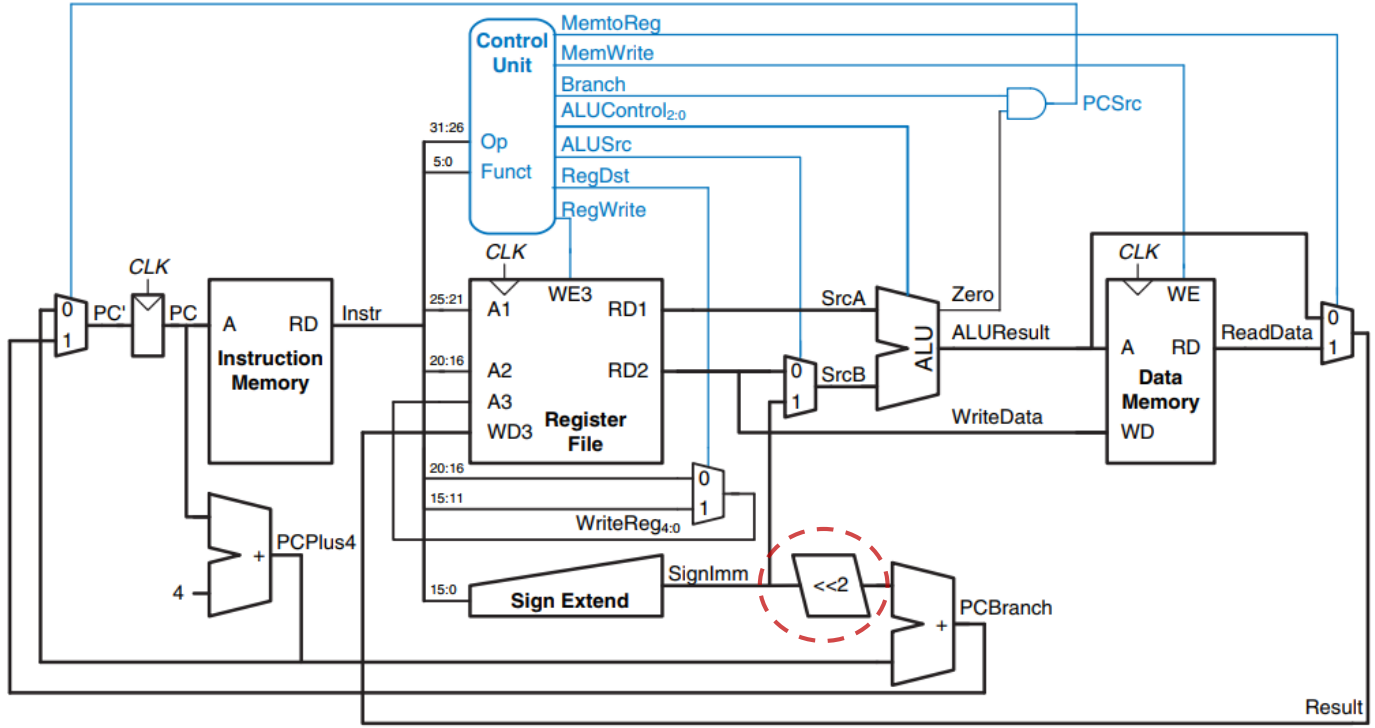


Figure 7.11 Complete single-cycle MIPS processor

Modules:

3. **<<2: Shifter**
   Shift data left with 2 bits.

MIPS Single-Cycle CPU

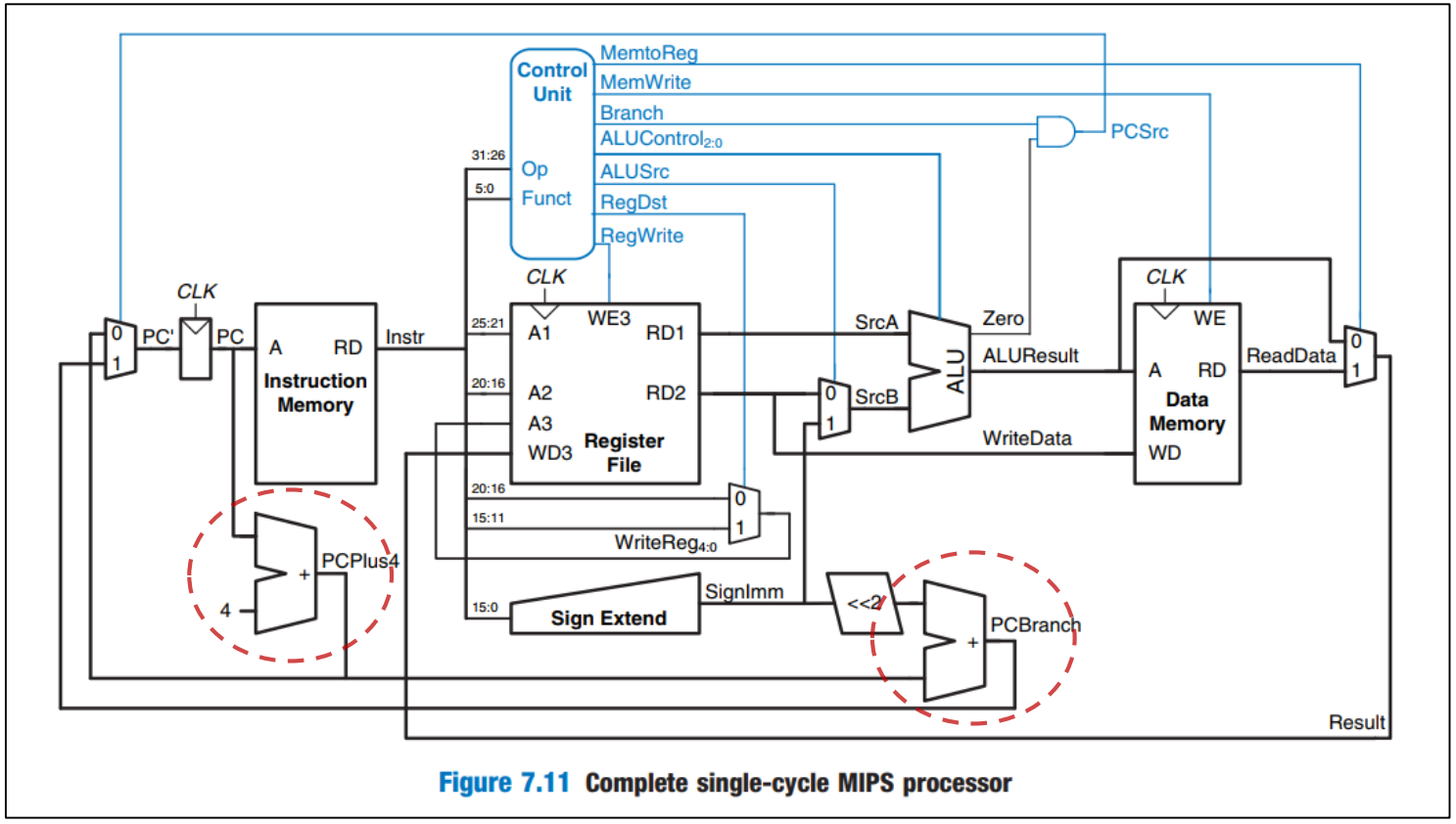


Figure 7.11 Complete single-cycle MIPS processor

Modules:

4. **Adder**
   (1) PC+4
   (2) Pcbranch

MIPS Single-Cycle CPU

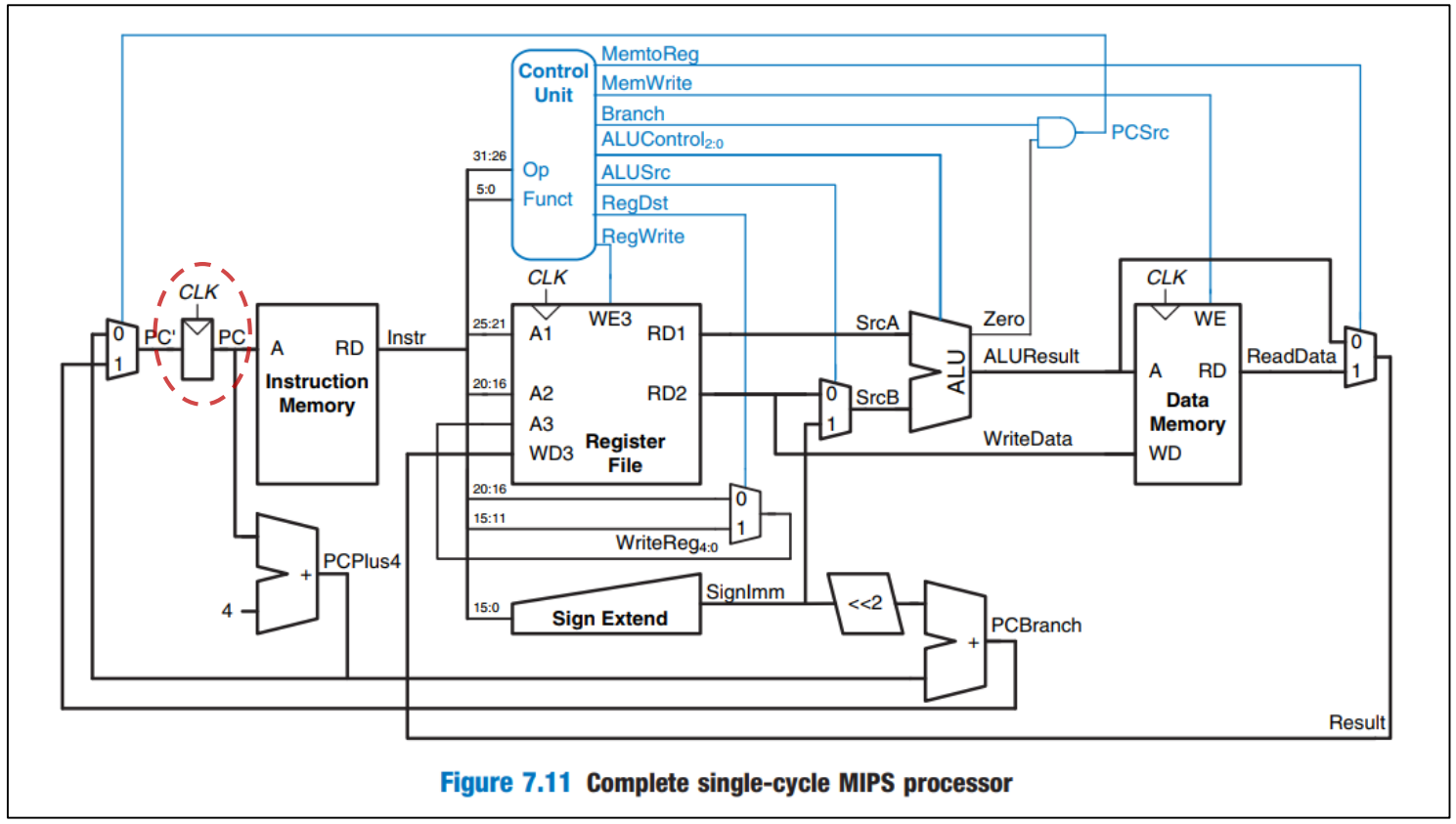


Figure 7.11 Complete single-cycle MIPS processor

Modules:

5. **PC**
   - Store 32-bit instruction address.
   - **Clk** signal.
   - **Rst**: reset signal. The initial reset operation should be carried out at the beginning of the experiment.

MIPS Single-Cycle CPU

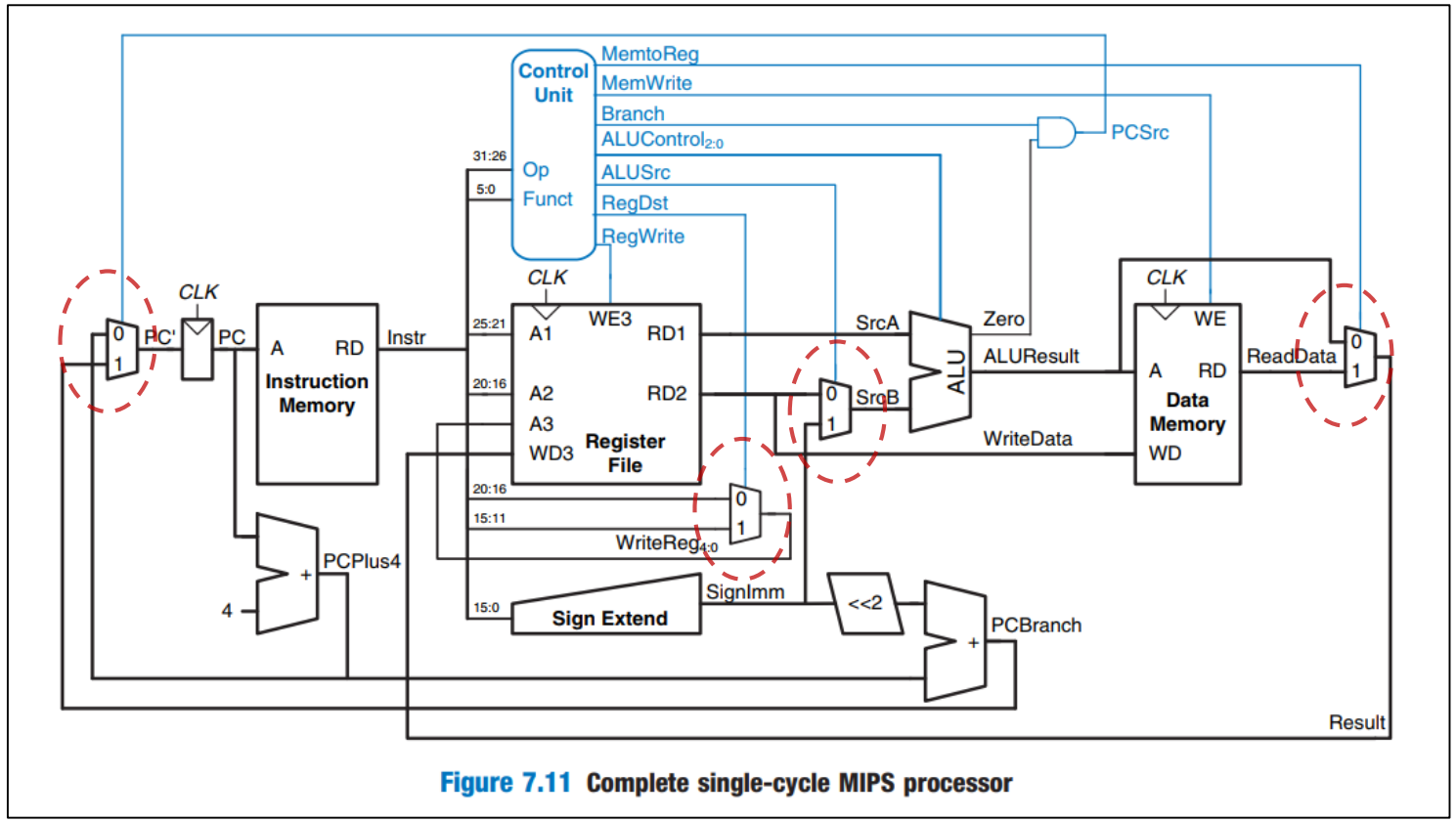


Figure 7.11 Complete single-cycle MIPS processor

Modules:

6. **MUX**
   - Two-way selector: according to the selection signal ***s*** to choose an output from two input channels 0 and 1.

MIPS Single-Cycle CPU

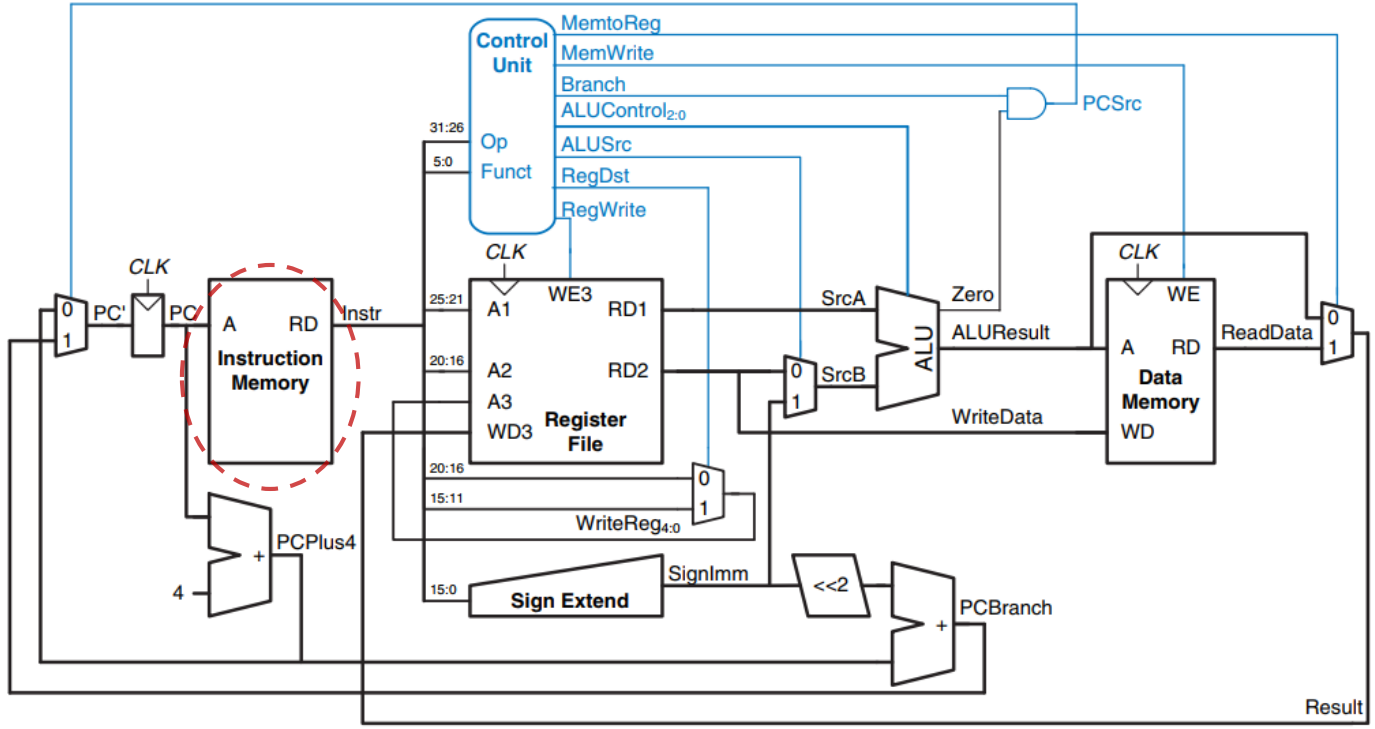


Figure 7.11 Complete single-cycle MIPS processor

Modules:

7. **Instruction Memory**
   - **ROM**: Read only memory.
   - No need the clock signal, according to the input address take out instruction.

MIPS Single-Cycle CPU

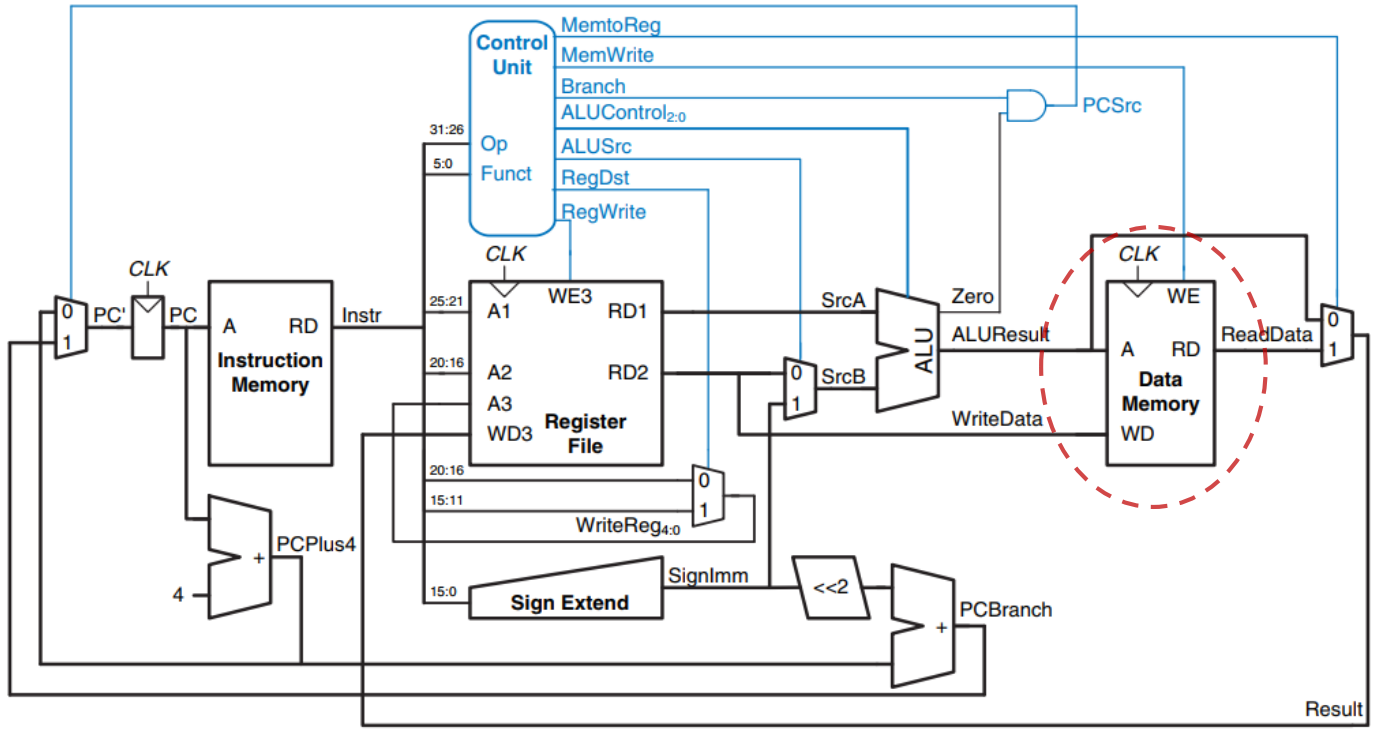


Figure 7.11 Complete single-cycle MIPS processor

Modules:

8. **Data Memory**
   - **RAM**: Read and Write.
   - The write operation is controlled by the **clk** signal and the **WE** write signal, which is determined by the **MemWrite** signal generated by decoding the op field of the controller.

MIPS Single-Cycle CPU

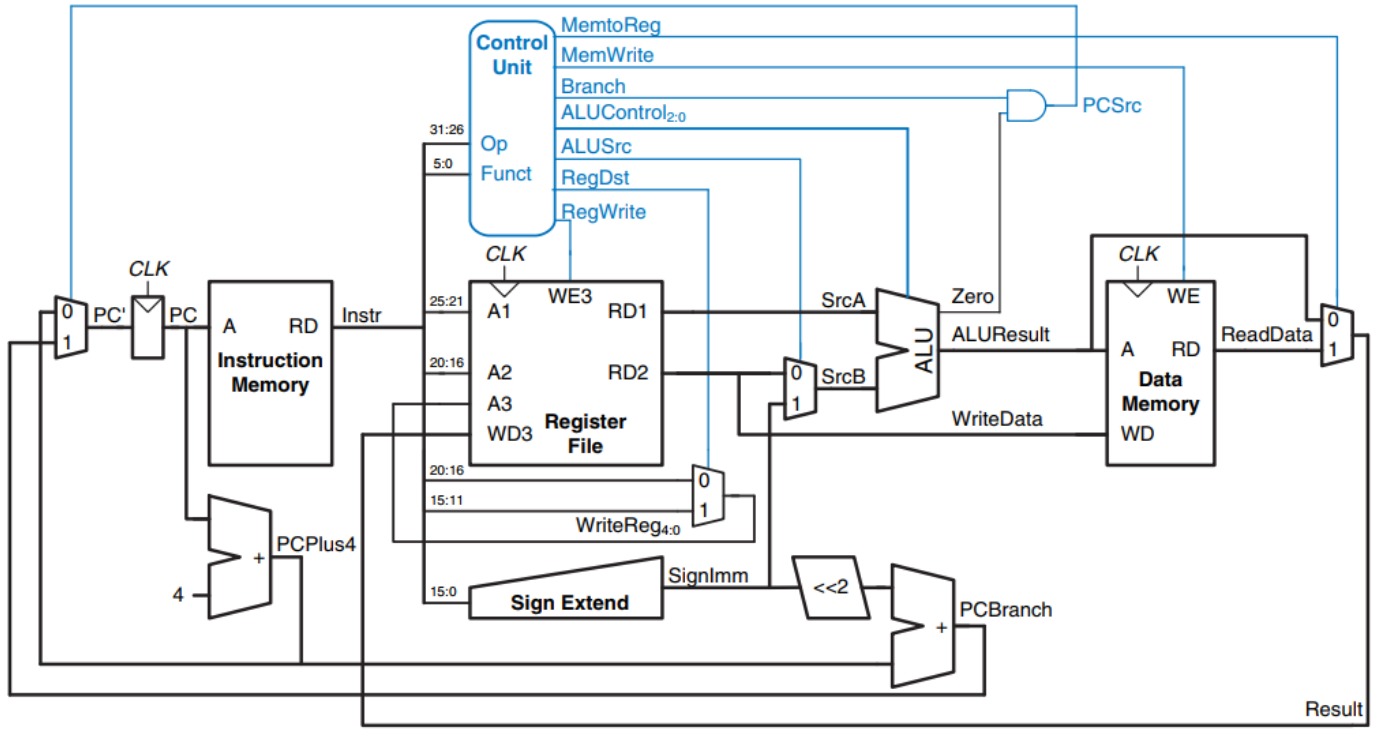


Figure 7.11 Complete single-cycle MIPS processor

Modules:

9. Control unit

- According to the **op** and **func** fields of the instruction, the control signals of each functional component required by each instruction are analyzed and obtained: ***memtoreg***, ***memwrite***, ***branch***, ***alusrc***, ***regdst***, ***regwrite*** and ***alucontrol***.
- Pay attention to the execution condition of the branch instruction ***PCSrc***: only valid when ***zero*** signal and ***branch*** signal are valid at the same time.

MIPS Single-Cycle CPU

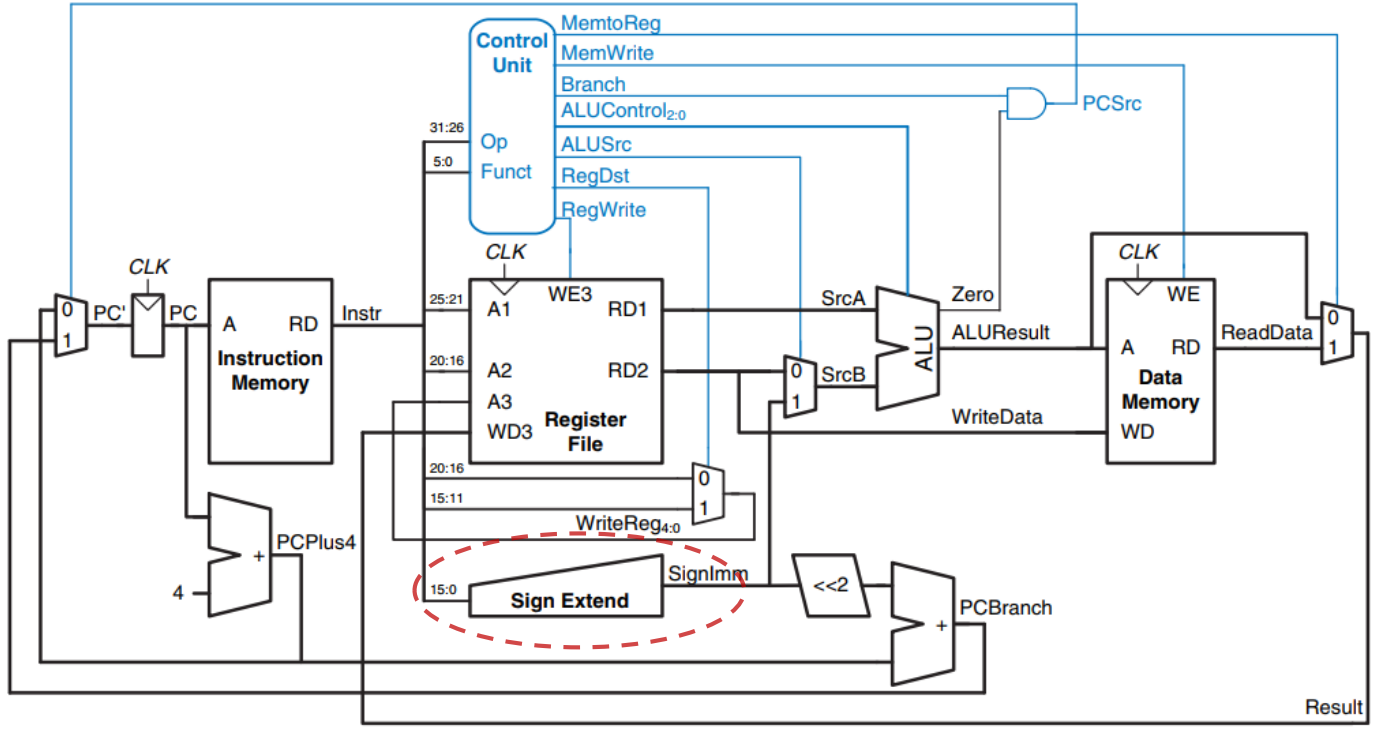


Figure 7.11 Complete single-cycle MIPS processor

Modules:

10. Sign extend

- The original **16-bit data** is extended to **32-bit data**.
- The expansion method is to take the highest bit as the symbol bit and expand according to the symbol bit.

Build each individual modules

- Design a 32-bit **PC** counter module
- Design a 32-bit **Adder**, for PC+4 to PC_next
- Design a 128B **ROM** with 32-bit word length

- Design a 32-bit **Register File**
- Design a 32-bit **ALU**
- Design a 32-bit "2 to 1" data selector, **MUX**

- Design a 128B **RAM** with 32-bit word length
- Design a 32-bit **Shifter**, shift left with 2 bits
- Design a **Sign Extend**, 16-bit data is extended to 32-bit data.

- Design a **Controller**, for instruction decode and control signals.

Experiment 1：**Fetch**
(PC、Adder、Instruction Memory)

Experiment 2：**Operation**
(RegFile、ALU、MUX)

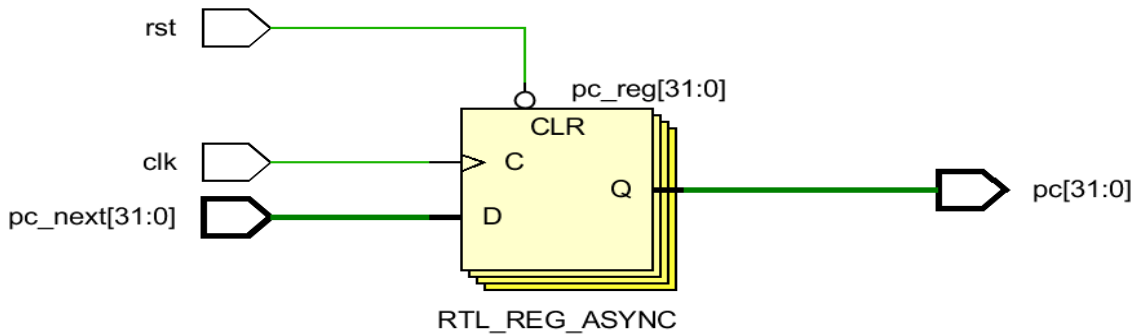
Experiment 3：**Memory Access**
(SignExt、Data Memory、Shifter)

Experiment 4：**Controller**

Build each individual modules(1)

- **Design a 32-bit PC counter module**

	Signal name	Signal function
Input signal	clk	Data input **clk**, 1-bit width
	rst	Data input **rst**, 1-bit width
	pc_next	Data input **pc_next**, 32-bit width
Output signal	pc	Data output **pc**, 32-bit width

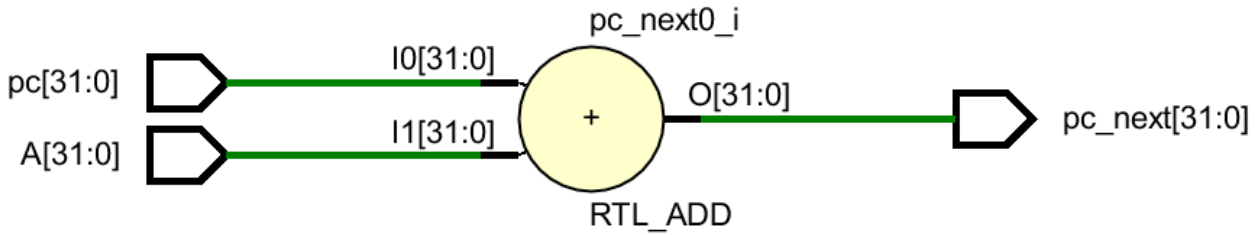



```
module pc(
  input wire clk,
  input wire rst,
  input[31:0] pc_next,
  output reg[31:0] pc
  );
  initial begin
      pc  <= 0;
  end
  always @(posedge clk or neged
ge rst)  begin
      if (!rst) begin
      pc <= 0;
      end
      else   begin
      pc <=  pc_next;
      end
  end
```

Build each individual modules(2)

- **Design a 32-bit Adder, for PC+4 to PC_next**

	Signal name	Signal function
Input signal	pc	Input **pc**, 32-bit width
	A	Input **A**, 32-bit width, A=00000004**H**
Output signal	pc_next	Output **pc_next**, 32-bit width

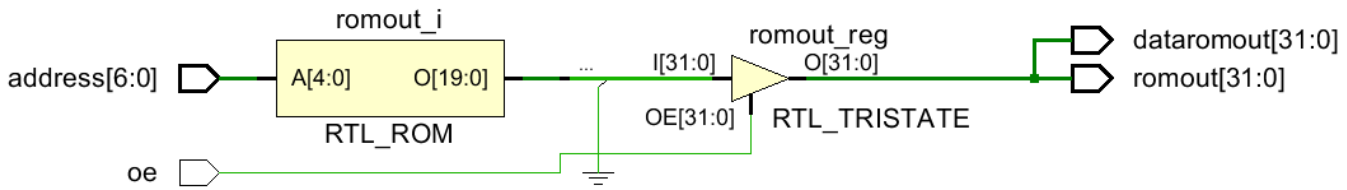



```
module Add(
    input wire [31:0] pc,
    input wire [31:0] A,
    output reg [31:0] pc_next
    );
    always @(pc or A)begin
        {pc_next}=pc+A;
    end
endmodule
```

Build each individual modules(3)

- **Design a 128B ROM with 32-bit word length**

	Signal name	Signal function
Input signal	address	Address input, 7-bit width
	oe	Output enable signal, 1-bit width, oe=1 is valid
Output signal	dataromout	Output **dataromout**, 32-bit width



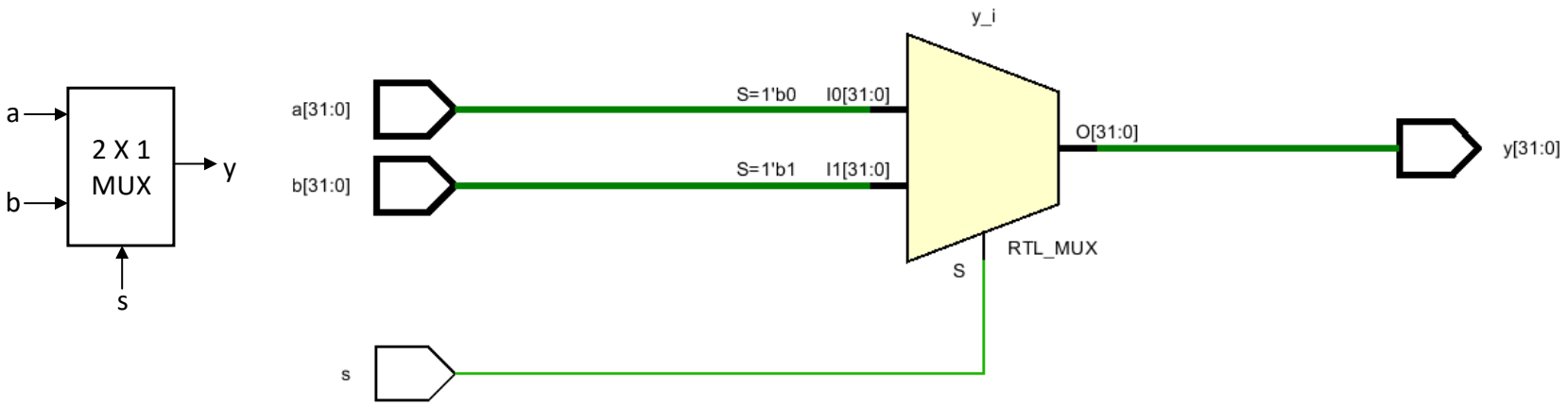

```
module Rom(
input [6:0] address,
input oe,
output [31:0] dataromout,
reg [31:0] romout
);
always @(*) begin
    if(oe)
        case(address[6:2])
        5'b00001:romout <= 32'h428c8;
        5'b00010:romout <= 32'h92008;
        5'b00011:romout <= 32'h63788;
        5'b01000:romout <= 32'h36389;
```

```
    5'b00100:romout <= 32'heeef8;
    5'b00101:romout <= 32'hbbcad;
    5'b00110:romout <= 32'h83665;
    5'b00111:romout <= 32'h8defc;
    5'b01000:romout <= 32'h9aade;
default: romout <= 32'h0;
endcase
else romout <= 32'hzz;
end
assign dataromout = romout;
endmodule
```

Build each individual modules(4)

- **Design a 32-bit "2 to 1" data selector, MUX**

	Signal name	Signal function
Input signal	a	Channel 1 data input, 32-bit width
	b	Channel 2 data input, 32-bit width
	s	Channel selection, 1-bit width
Output signal	y	Data output **y**, 32-bit width

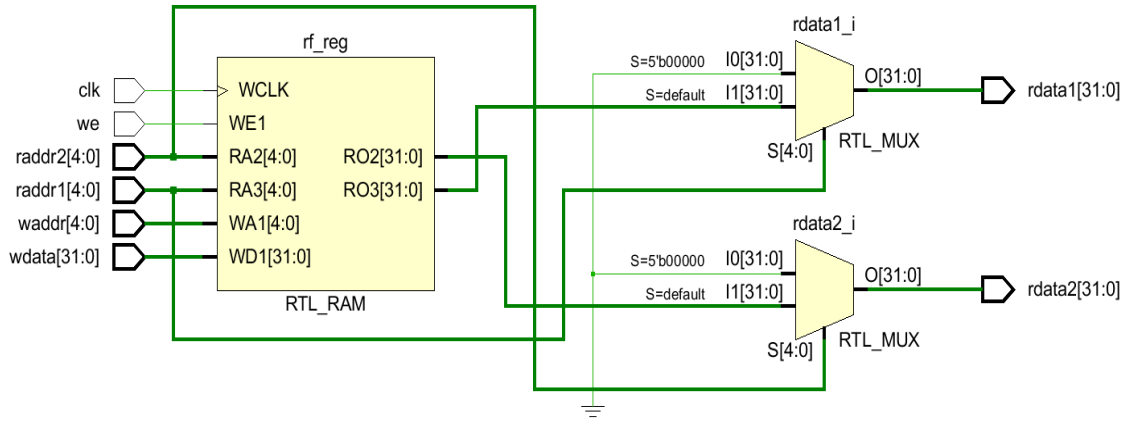



```
module Multiplexer(
    input wire[31:0] s,
    input wire[31:0] a,
    input wire[31:0] b,
    output reg[31:0] y
    );
    always @(*) begin
        case(s)
            32'h0:begin
                y <= a;
            end
            32'h1:begin
                y <= b;
            end
        endcase
    end
endmodule
```

Build each individual modules(5)

◆ Design a 32-bit Register File

	Signal name	Signal function
Input signal	clk	Clock signal, 1-bit
	we	WE signal, 1-bit
	raddr1	Address of data 1, 5-bit
	raddr2	Address of data 2, 5-bit
	waddr	Address for write, 5-bit
	wdata	Write data, 32-bit
Output signal	rdata1	Output data 1, 32-bit
	rdata2	Output data 2, 32-bit

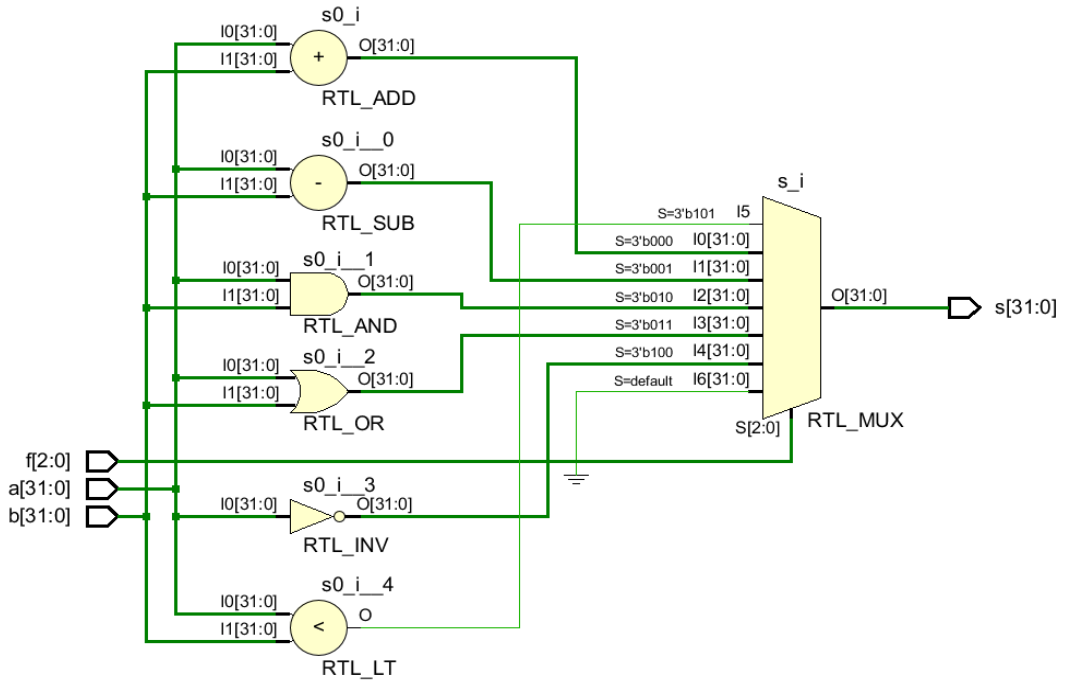



```
module regfile(
    input  clk,
    input  [4:0] raddr1,
    output [31:0] rdata1,
    input  [4:0] raddr2,
    output  [31:0] rdata2,
    input   we,
    input   [4:0] waddr,
    input   [31:0] wdata
);
    integer i;
    reg [31:0] rf[31:0];
    always @(posedge clk ) begin

        if (we)  begin
            rf[waddr]<= wdata;
        end
    end

    assign rdata1 = (raddr1==5'h0) ? 32'h0 : rf[raddr1];
    assign rdata2 = (raddr2==5'h0) ? 32'h0 : rf[raddr2];

endmodule
```

Build each individual modules(6)

- **Design a 32-bit ALU**

	Signal name	Signal function
Input signal	a	Data input a, 32-bit
	b	Data input b, 32-bit
	f	Data input f, 3-bit
Output signal	s	Data output s, 32-bit

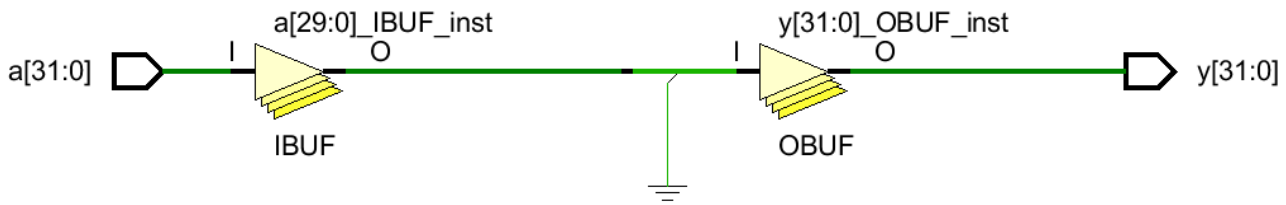



```
module ALU(
    input wire [31:0] a,
    input wire [31:0] b,
    input wire [2:0] f,
    output reg [31:0] s
    );

    always @(*) begin
        case(f)
            3'b000: begin
                s <= a + b;
            end
            3'b001: begin
                s <= a - b;
            end
            3'b010: begin
                s <= a & b;
            end
            3'b011: begin
                s <= a | b;
            end
            3'b100: begin
                s <= ~a;
            end
            3'b101: begin
                s <= (a < b);
            end
            default: begin
                s <= 32'b0;
            end
        endcase
    end
endmodule
```

Build each individual modules(7)

- **Design a 32-bit Shifter, shift left with 2 bits**

	Signal name	Signal function
Input signal	a	Data input a, 32-bit
Output signal	y	Data output y, 32-bit



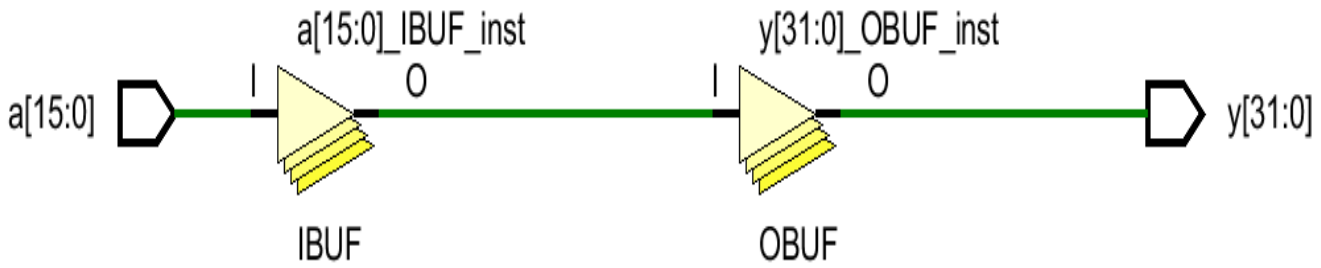

```
module shift(
    input wire[31:0] a,
    output [31:0] y
    );
    assign y = {a[29:0],2'b00};
endmodule
```

Build each individual modules(8)

- **Design a Sign Extend, 16-bit data is extended to 32-bit data.**

	Signal name	Signal function
Input signal	a	Data input a, 16-bit
Output signal	y	Data output y, 32-bit

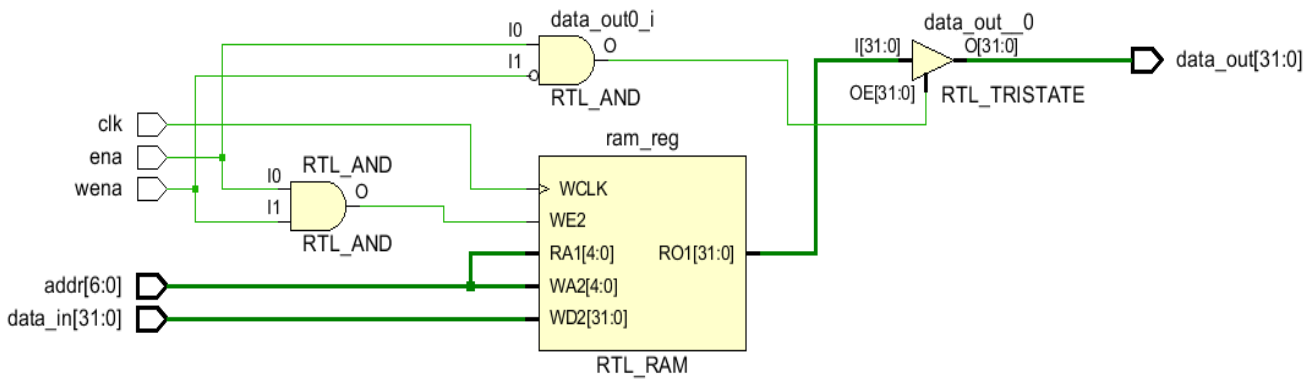
```
module signex(
    input wire[15:0] a,
    output [31:0] y
    );

    assign y = {{16{a[15]}},a};
endmodule
```



Build each individual modules(9)

- **Design a 128B RAM with 32-bit word length**

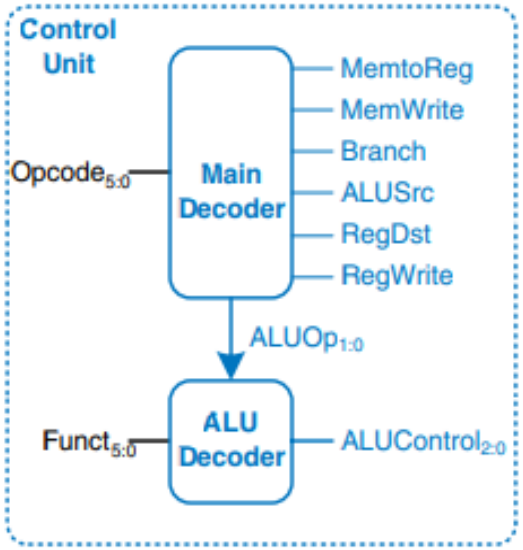
	Signal name	Signal function
Input signal	clk	Clock signal, 1-bit
	ena	Memory effective signal, 1-bit
	wena	Write enable signal, 1-bit
	Addr	Address, 7-bit
	data_in	Data for write, 32-bit
Output signal	data_out	Data for read, 32-bit




```
module Ram(
    input clk,  //存储器时钟信号，上升沿时向 ram 内部写入数据
    input ena,  //存储器有效信号，高电平时存储器才运行，否则输出 z
    input wena, //存储器读写有效信号，高电平为写有效，低电平为读有效，与
ena 同时有效时才可对存储器进行读写
    input [6:0] addr,       //输入地址，指定数据读写的地址
    input [31:0] data_in,   //存储器写入的数据，在 clk 上升沿时被写入
    output [31:0] data_out  //存储器读出的数据，
    );

    reg [31:0] ram [31:0];

    always @(posedge clk)
    begin
        if (ena & wena)  // clk ena wena 同时有效“写”
            ram[addr] <= data_in;
    end

    assign data_out = ( ena & ~wena) ? ram[addr] : 32'hz;  //ena
高 wena低 读（不考虑 clk ?)
endmodule
```

Build each individual modules(10)

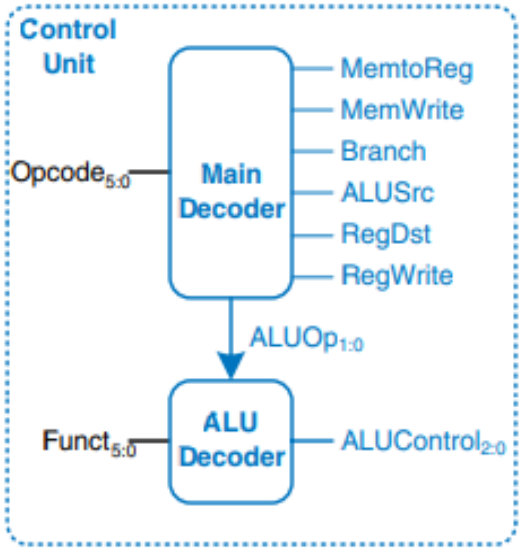
◆**Design a Controller, for instruction decode and control signals.**

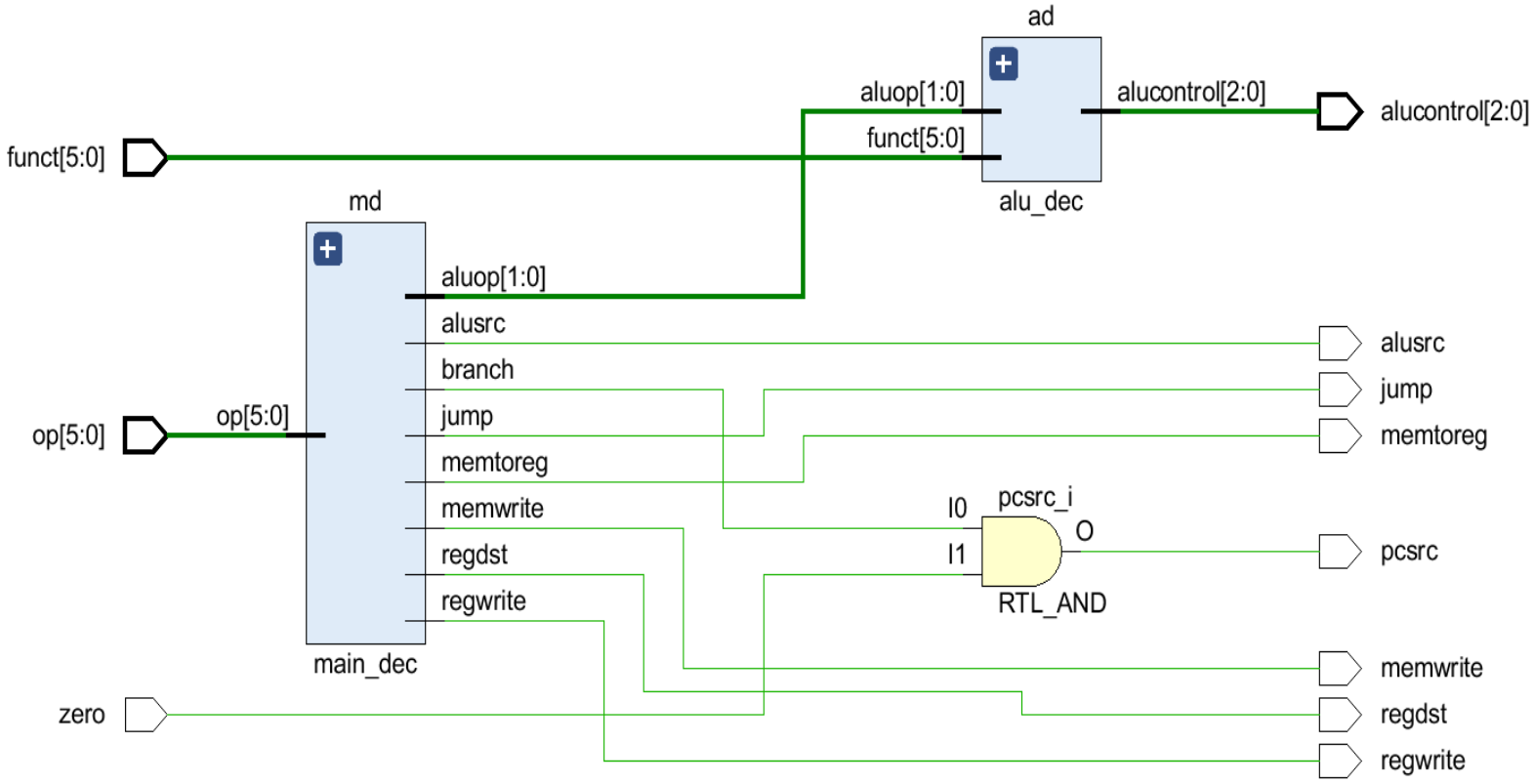


	Signal name	Signal function
Input signal	Op	Opcode, 6-bit
	funct	function code, specifies the function of the instruction, and **aluop** operation code used together, 6-bit
	Zero	Operation result flag, determine whether the result of subtracting two numbers is 0, related to the instruction branch, 1-bit
Output signal (Control signals)	regwrite	Regfile write control, 1-bit
	regdst	Register(for write) selection, 1-bit
	alusrc	ALU input selection, 1-bit
	memwrite	Data Memory write control, 1-bit
	memtoreg	Data input selection, 1-bit
	jump	Jump address, 1-bit
	pcsrc	Branch for *beq*, 1-bit
	alucontrol	ALU control, specify operation type, 3-bit

Build each individual modules(10)

- **Design a Controller, for instruction decode and control signals.**





Build each individual modules(10)

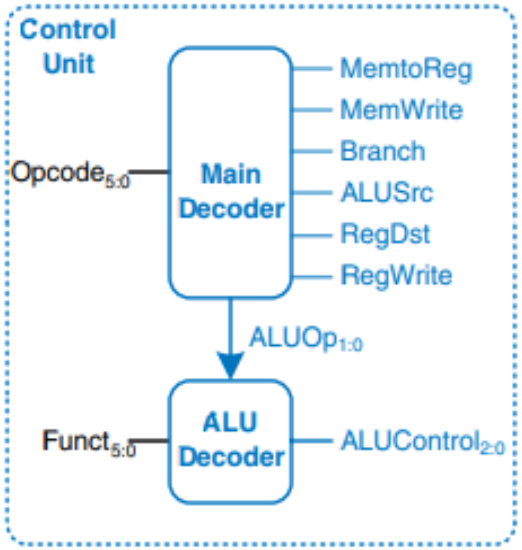
- **Design a Controller, for instruction decode and control signals.**

```
module main_dec(
    input wire[5:0] op,
    output wire memtoreg,memwrite,
    output wire branch,alusrc,
    output wire regdst,regwrite,
    output wire jump,
    output wire[1:0] aluop
    );
    reg[8:0] controls;
    assign {regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump,aluop} = controls;
    always @(*) begin
        case (op)
            6'b000000:controls <= 9'b110000010;//R-TYRE
            6'b100011:controls <= 9'b101001000;//LW
            6'b101011:controls <= 9'b001010000;//SW
            6'b000100:controls <= 9'b000100001;//BEQ
            6'b001000:controls <= 9'b101000000;//ADDI
            6'b000010:controls <= 9'b000000100;//J
            default:  controls <= 9'b000000000;//illegal op
        endcase
    end
endmodule
```

```
module alu_dec(
    input wire[5:0] funct,
    input wire[1:0] aluop,
    output reg[2:0] alucontrol
    );
    always @(*) begin
        case (aluop)
            2'b00: alucontrol <= 3'b010;//add (for lw/sw/addi)
            2'b01: alucontrol <= 3'b110;//sub (for beq)
            default : case (funct)
                6'b100000:alucontrol <= 3'b010; //add
                6'b100010:alucontrol <= 3'b110; //sub
                6'b100100:alucontrol <= 3'b000; //and
                6'b100101:alucontrol <= 3'b001; //or
                6'b101010:alucontrol <= 3'b111; //slt
                default:  alucontrol <= 3'b000;
            endcase
        endcase
```

Build each individual modules(10)

◆**Design a Controller, for instruction decode and control signals.**



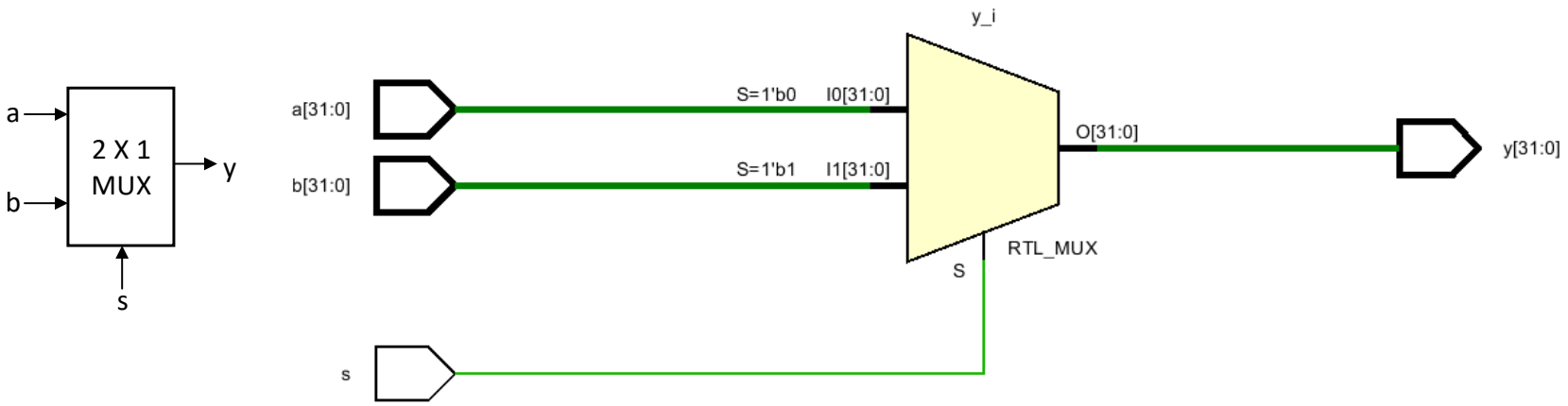

```
module controller(
    input wire[5:0] op,funct,
    input wire zero,
    output wire memtoreg,memwrite,
    output wire pcsrc,alusrc,
    output wire regdst,regwrite,
    output wire jump,
    output wire[2:0] alucontrol
    );
    wire[1:0] aluop;
    wire branch;
    main_dec md(op,memtoreg,memwrite,branch,alusrc,regdst,regwrite,jump,aluop);
    alu_dec ad(funct,aluop,alucontrol);
    assign pcsrc = branch & zero;
endmodule
```

**Eg. Simulation of module (MUX2)

MUX

- **Design a 32-bit "2 to 1" data selector, MUX**

	Signal name	Signal function
Input signal	a	Channel 1 data input, 32-bit width
	b	Channel 2 data input, 32-bit width
	s	Channel selection, 1-bit width
Output signal	y	Data output **y**, 32-bit width



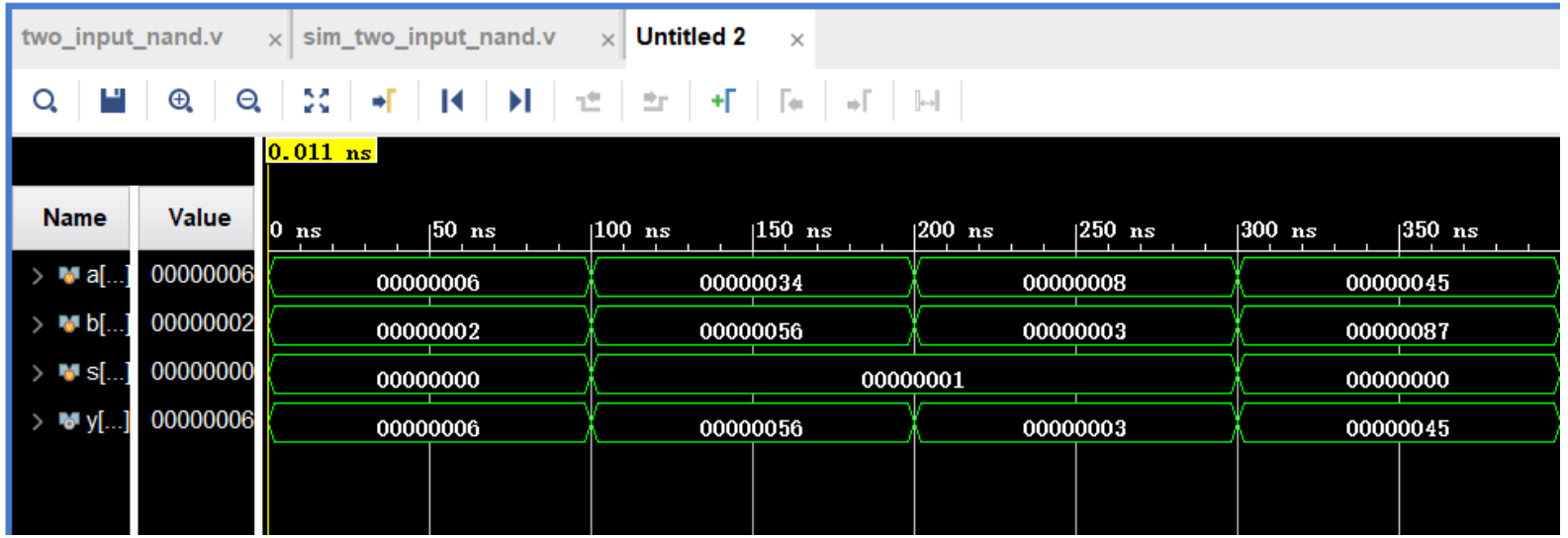

```
module Multiplexer(
    input wire[31:0] s,
    input wire[31:0] a,
    input wire[31:0] b,
    output reg[31:0] y
    );
    always @(*) begin
        case(s)
            32'h0:begin
                y <= a;
            end
            32'h1:begin
                y <= b;
            end
        endcase
    end
endmodule
```

Simulation

- **Design a 32-bit "2 to 1" data selector, MUX**

“2 to 1” MUX

No	a	b	s
1	32’h6	32’h2	32’h0
2	32’h34	32’h56	32’h1
3	32’h8	32’h3	32’h1
4	32’h45	32’h87	32’h0

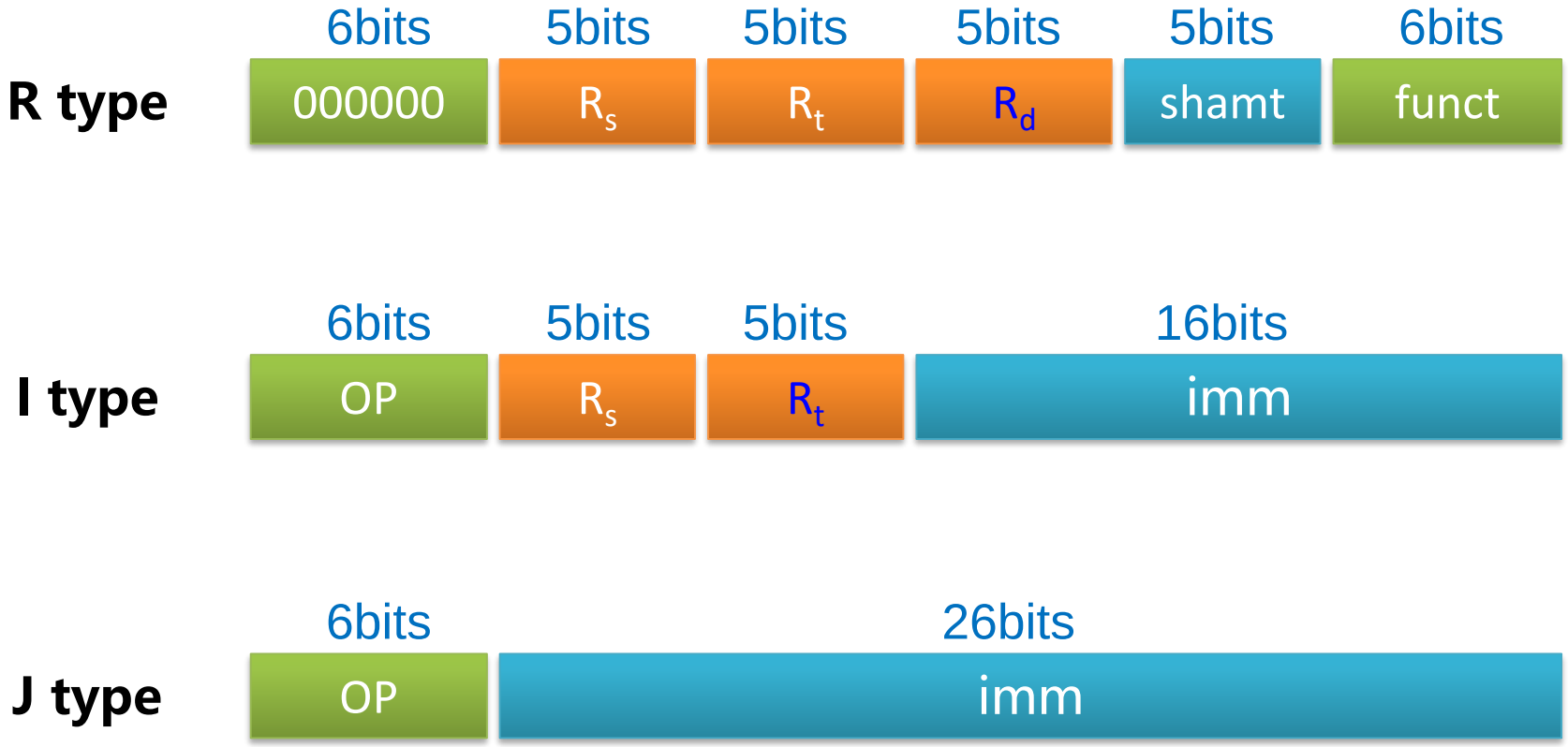



```
module sim_Multiplexer();
    reg [31:0] a;
    reg [31:0] b;
    reg [31:0] s;
    wire [31:0] y;
//要测试的实例化模块
    Multiplexer uut(.a(a), .b(b),
 .s(s), .y(y));
    always begin
        a=32'h6;b=32'h2;s=32'h0;#
100;
        a=32'h34;b=32'h56;s=32'h1
;#100;
        a=32'h8;b=32'h3;s=32'h1;#
100;
        a=32'h45;b=32'h87;s=32'h0
;#100;
    end
endmodule
```

南通大學
NANTONG UNIVERSITY

Step 2: Build DataPath

MIPS-ISA


R type
6bits
5bits
5bits
5bits
5bits
6bits
000000
R_s
R_t
R_d
shamt
funct
I type
6bits
5bits
5bits
16bits
OP
R_s
R_t
imm
J type
6bits
26bits
OP
imm

MIPS-ISA

Funct	Instruction mnemonics	Function
32	add rd,rs,rt	R[rd]=R[rs]+R[rt]
34	sub rd,rs,rt	R[rd]=R[rs]-R[rt]
36	and rd,rs,rt	R[rd]=R[rs]&R[rt]
37	or rd,rs,rt	R[rd]=R[rs] | R[rt]
42	slt rd,rs,rt	R[rd]=(R[rs]<R[rt])?1:0

MIPS-ISA

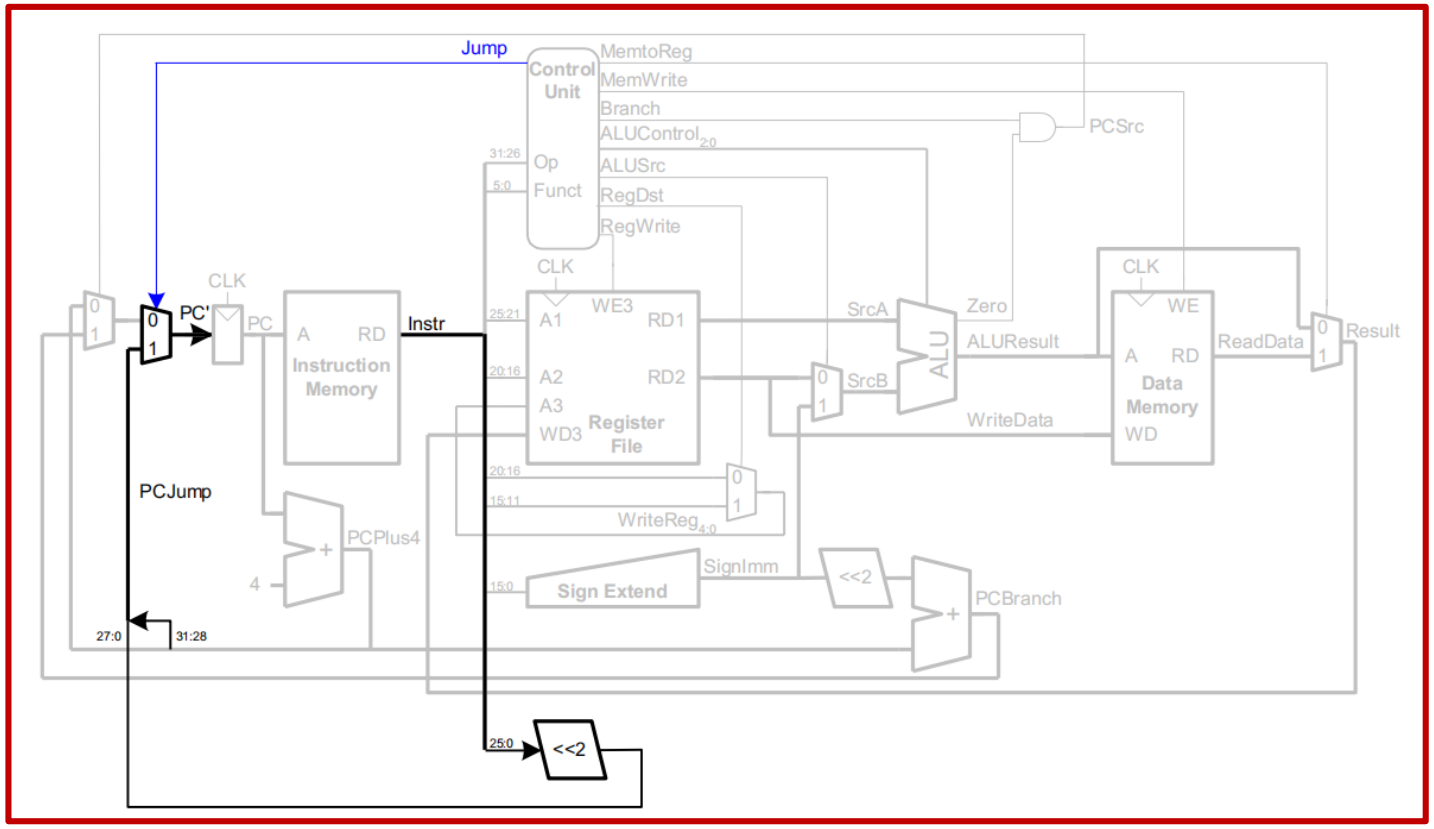
I type

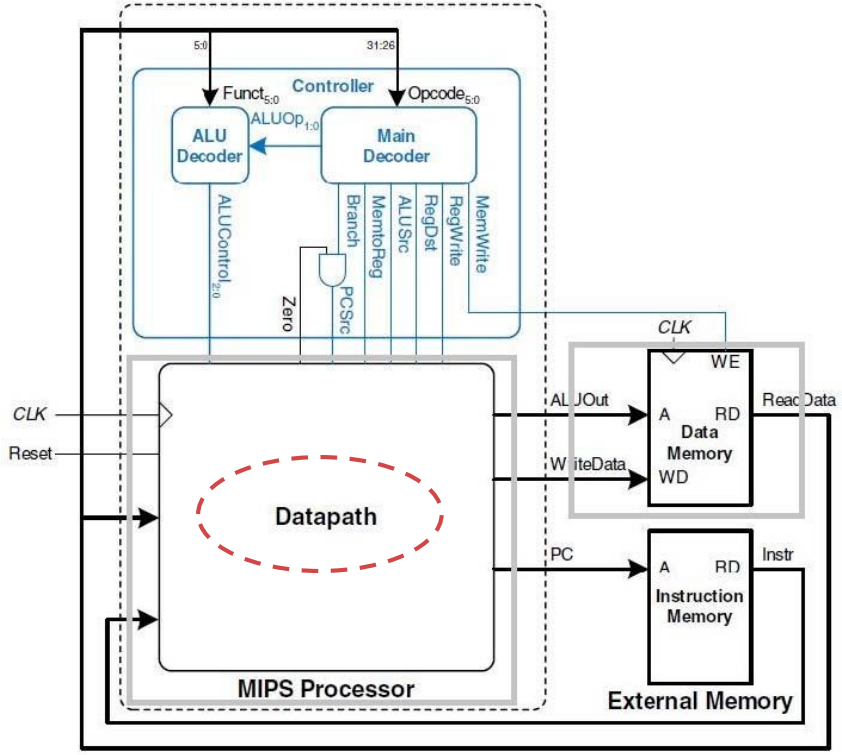
OP	Instruction mnemonics	Function
04	beq rs,rt,imm	if(R[rs]==R[rt]), PC=PC+4+imm<<2
08	addi rt,rs,imm	R[rt]=R[rs]+imm
35	lw rt,imm(rs)	R[rt]=M[R[rs]+imm]
43	sw rt,imm(rs)	M[R[rs]+imm]=R[rt]

J type

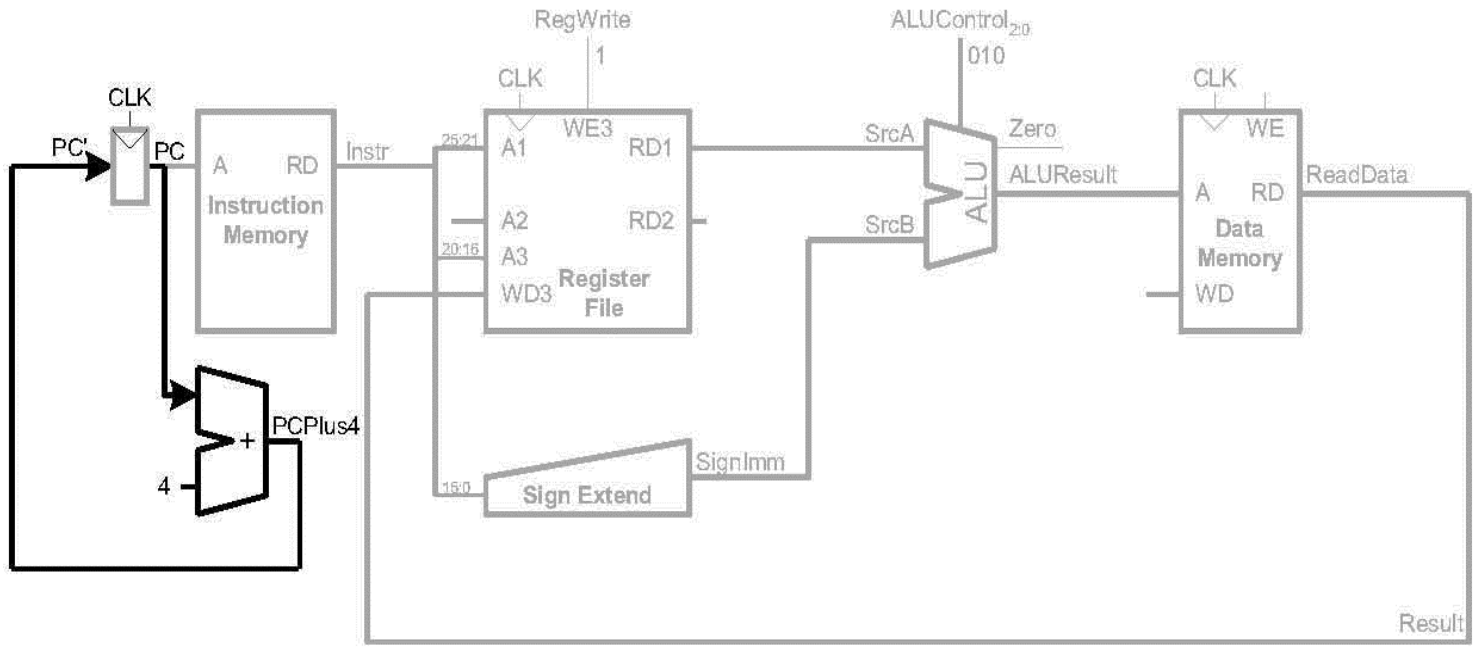
OP	Instruction mnemonics	Function
02	j address	PC ← $\{(PC+4)_{31:28}$, address,00$\}$

MIPS Single-Cycle CPU

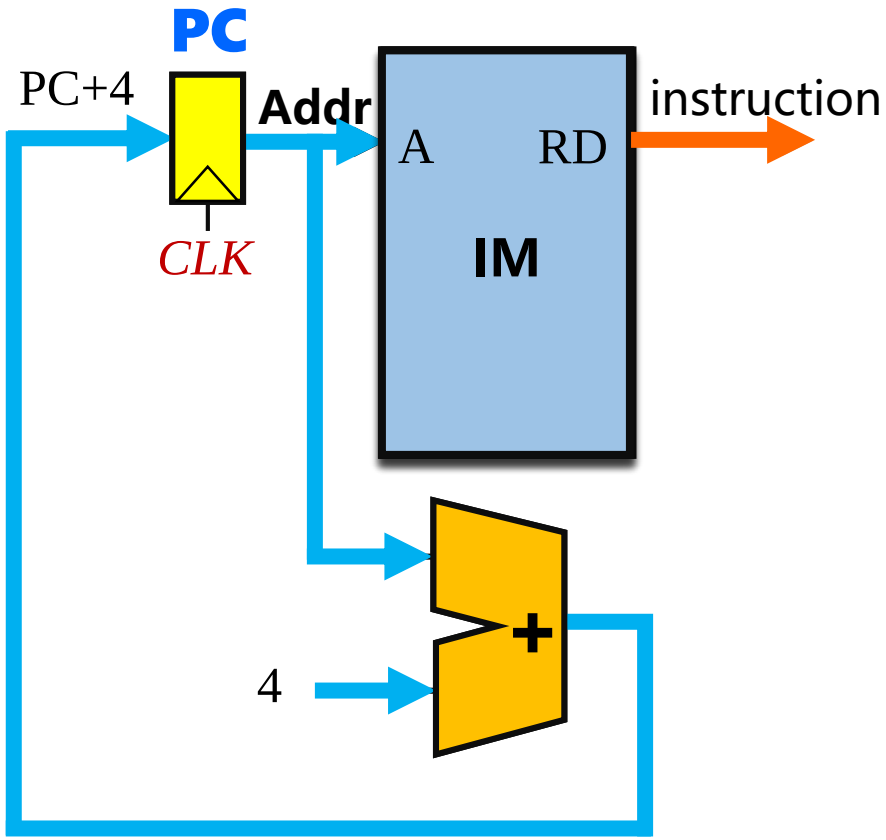




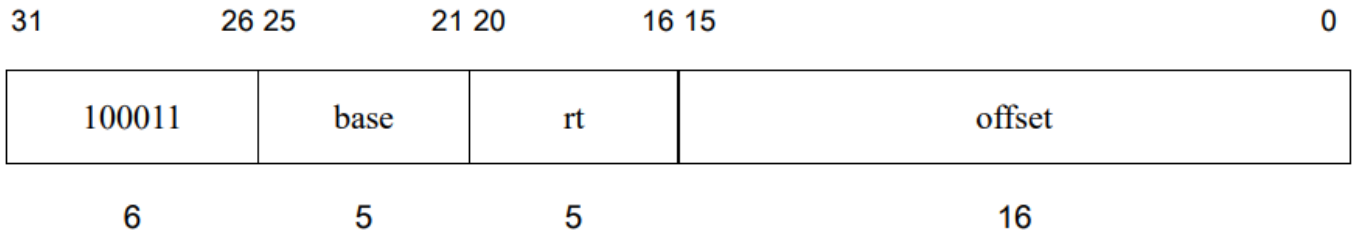
Fetch module



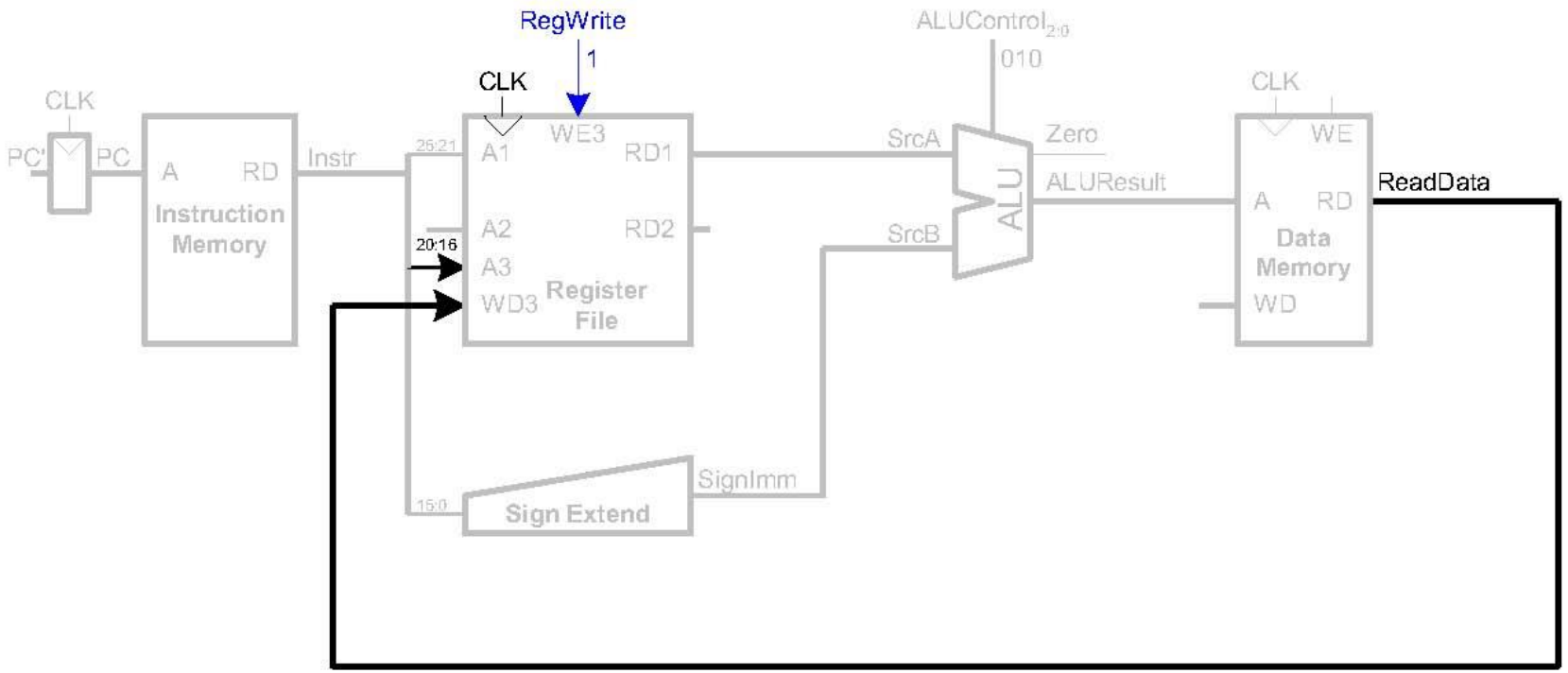
Fetch module



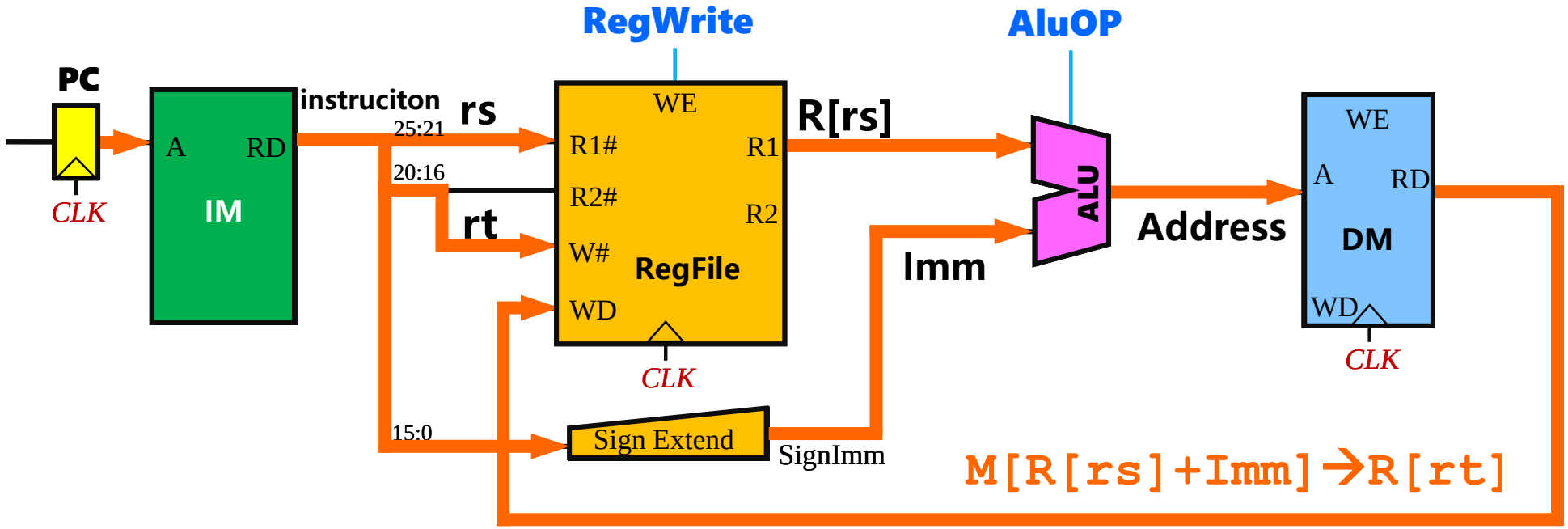
lw instruction (load, memory access)



汇编格式： LW rt, offset(base)

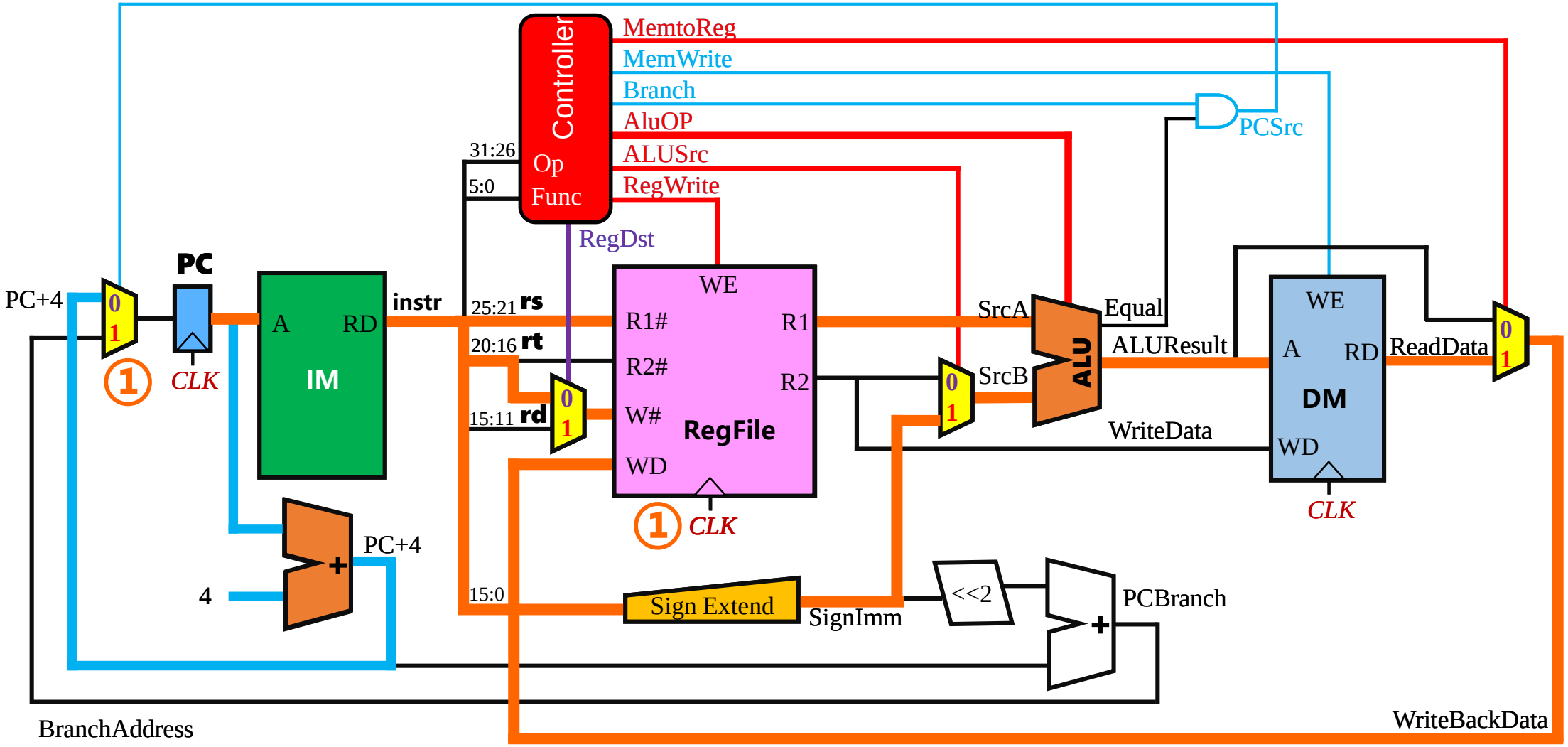


lw instruction (load, memory access)

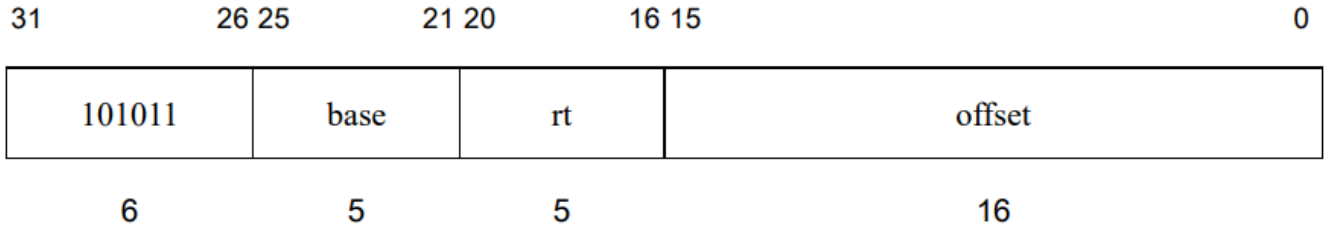


lw DataPath

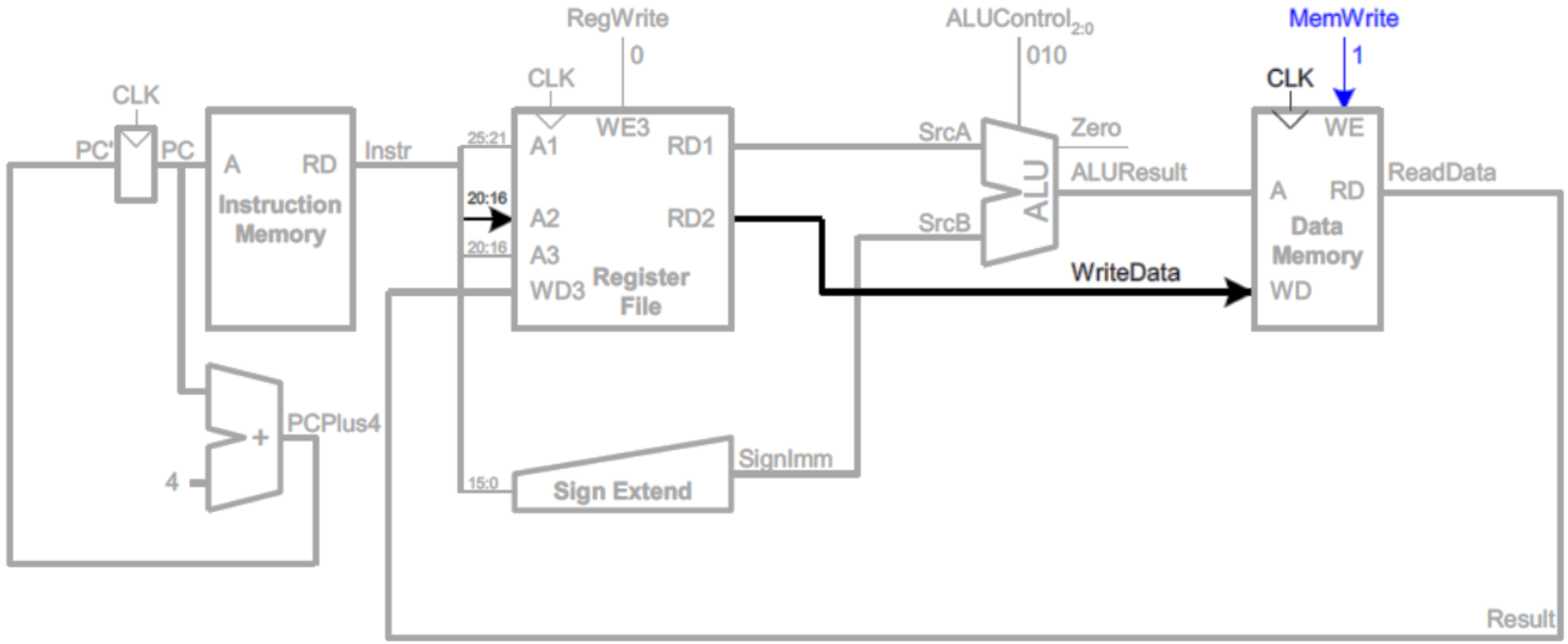
lw rt,imm(rs)	R[rt]=M[R[rs]+imm]



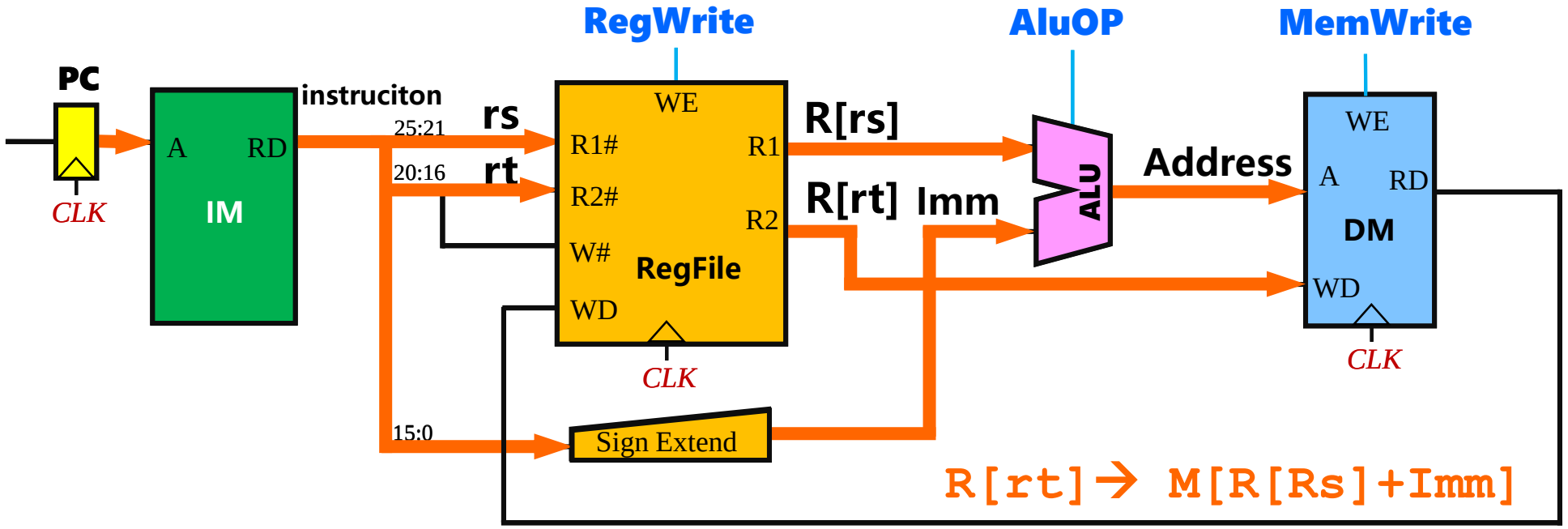
sw instruction (store, memory access)



汇编格式：SW　rt, offset(base)



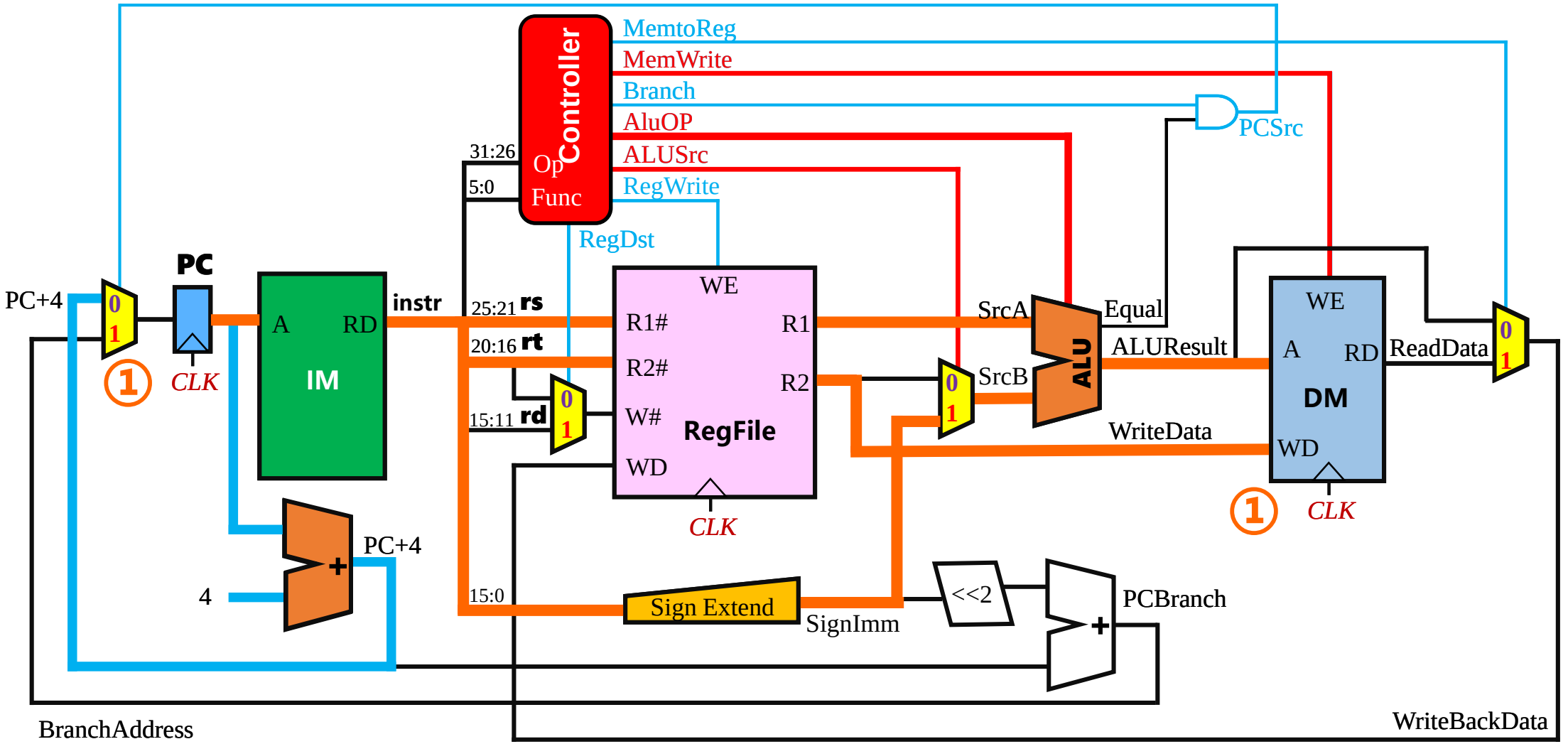
sw instruction (store, memory access)



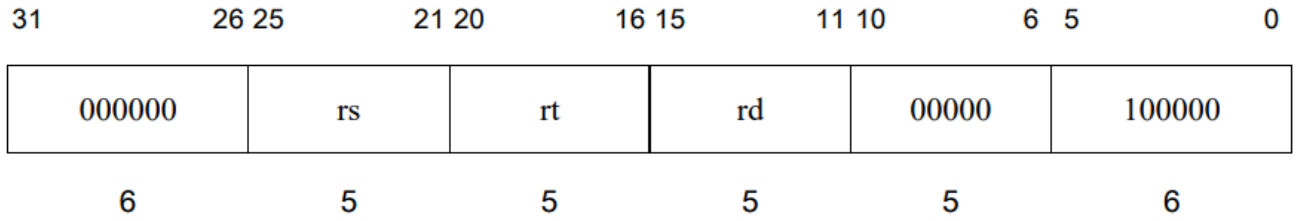
- `sw $s0,32($s1)`

Sw DataPath

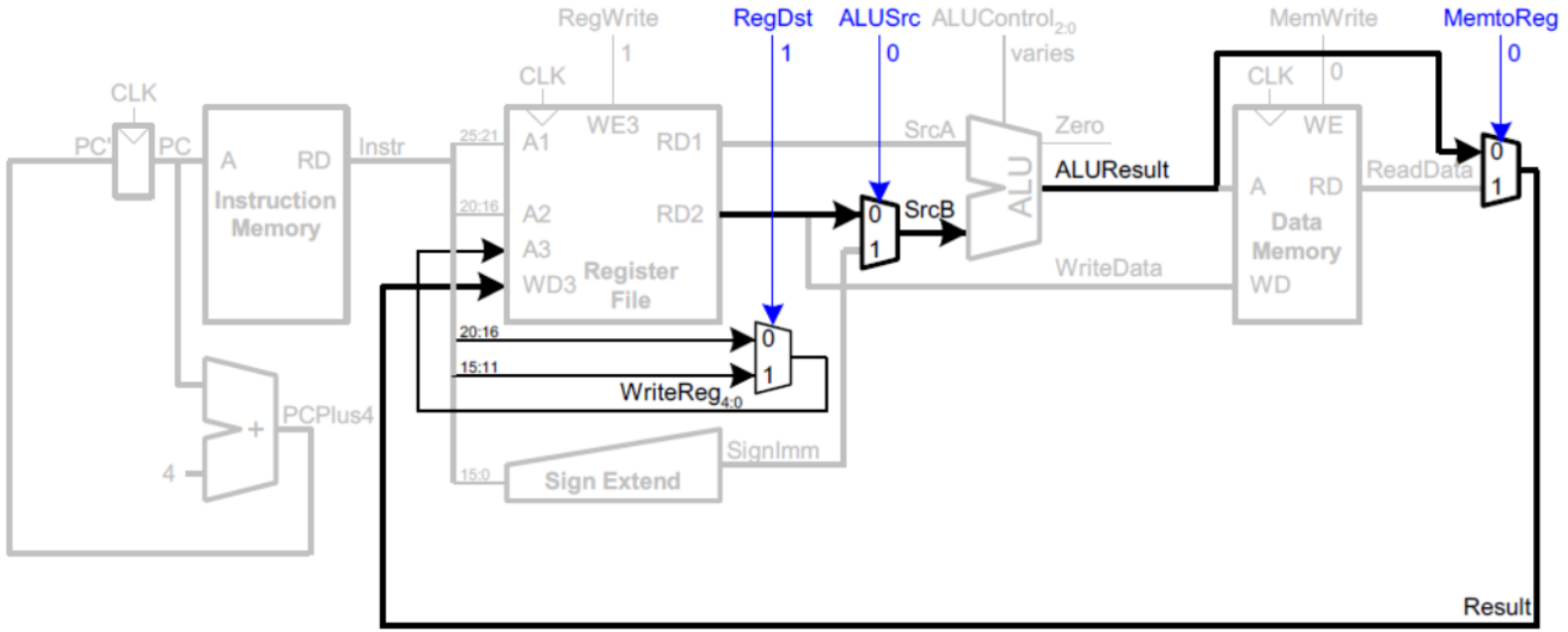
sw rt.imm(rs)	M[R[rs]+imm]=R[rt]



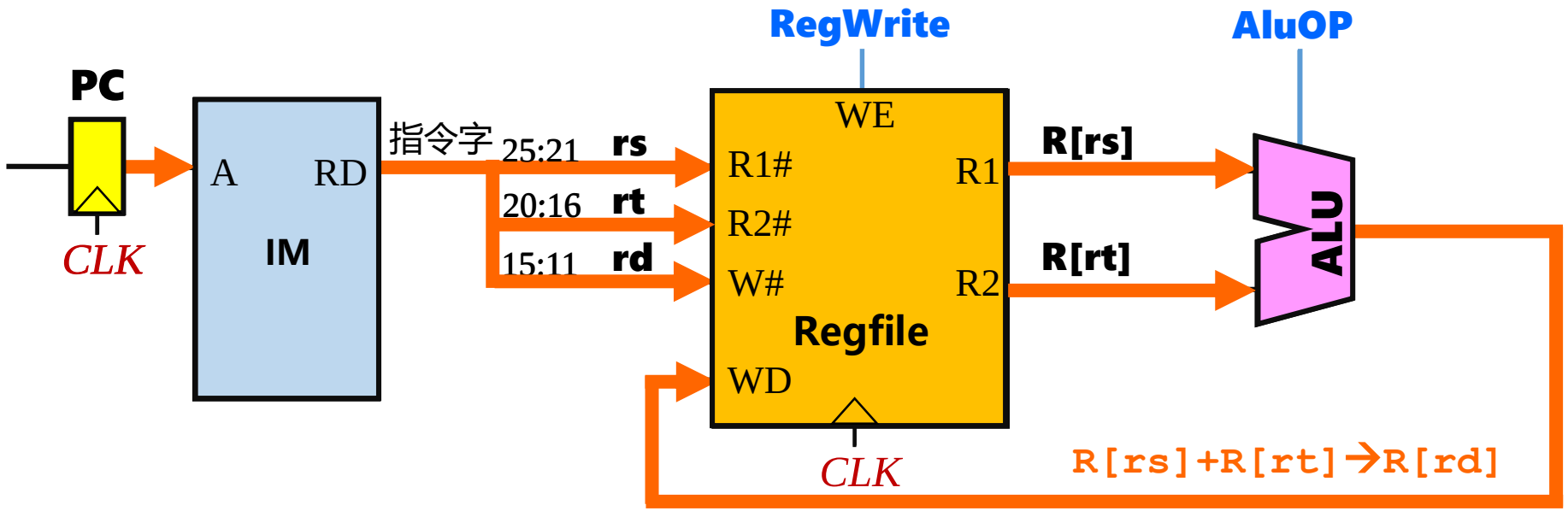
R-type instructions



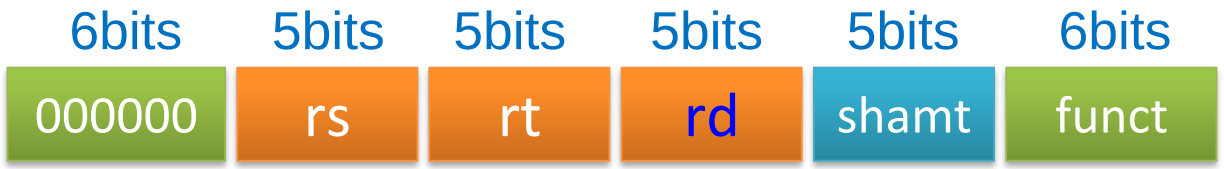
汇编格式： ADD rd, rs, rt



R-type instructions



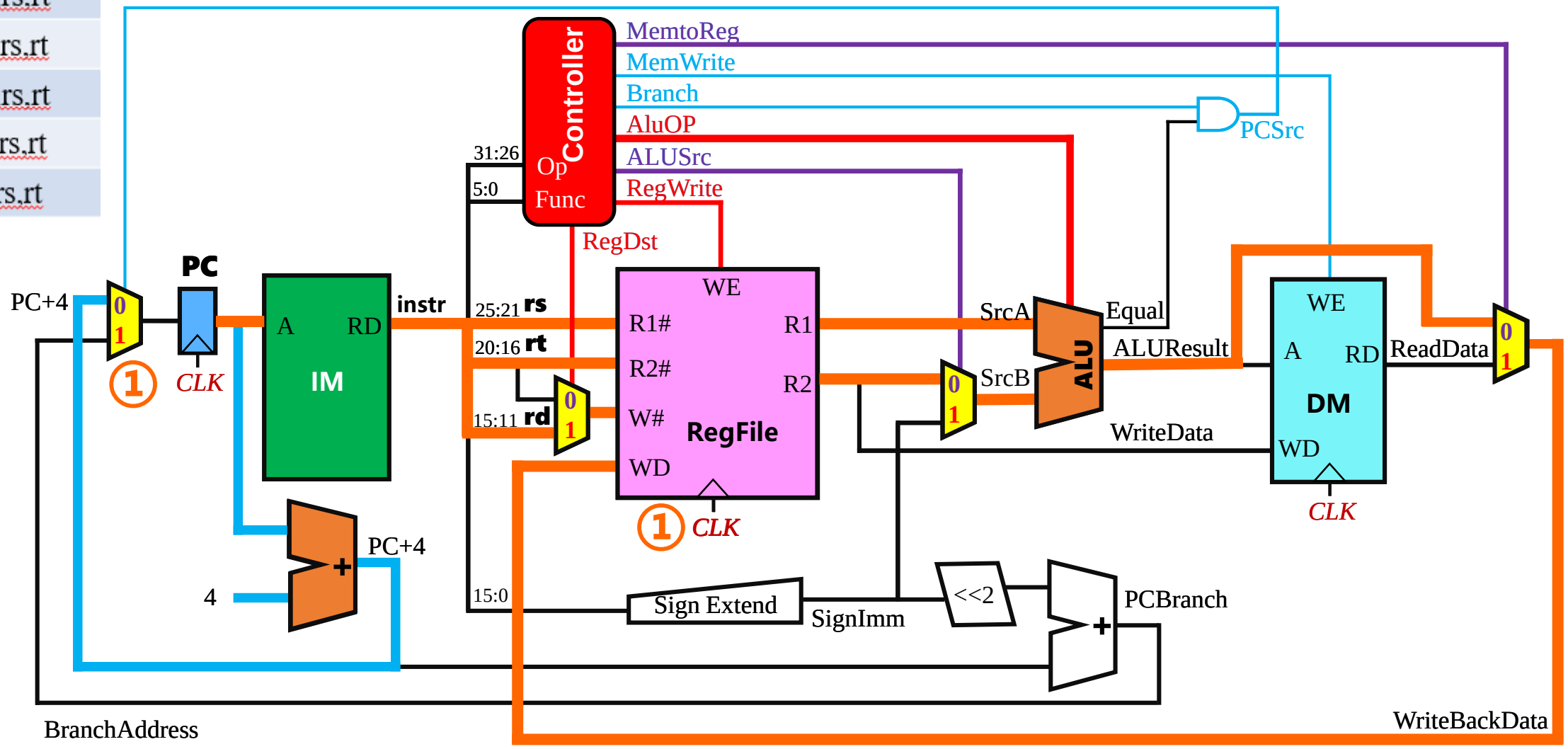

```
add $s0,$s1,$s2
```



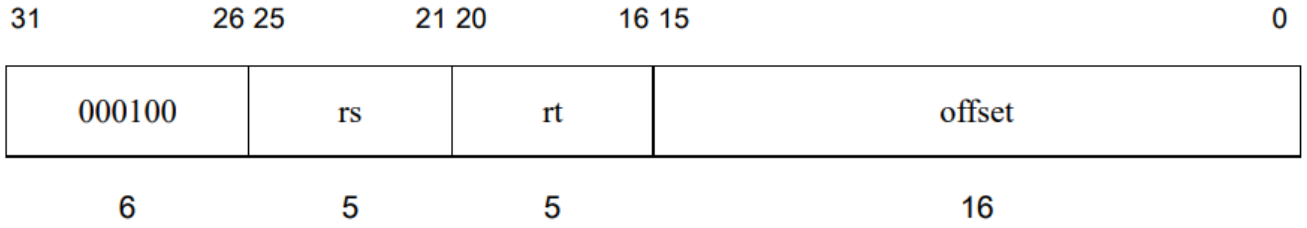
R-type DataPath

6bits	5bits	5bits	5bits	5bits	6bits
R-type	OP	rs	rt	rd	shamt	funct

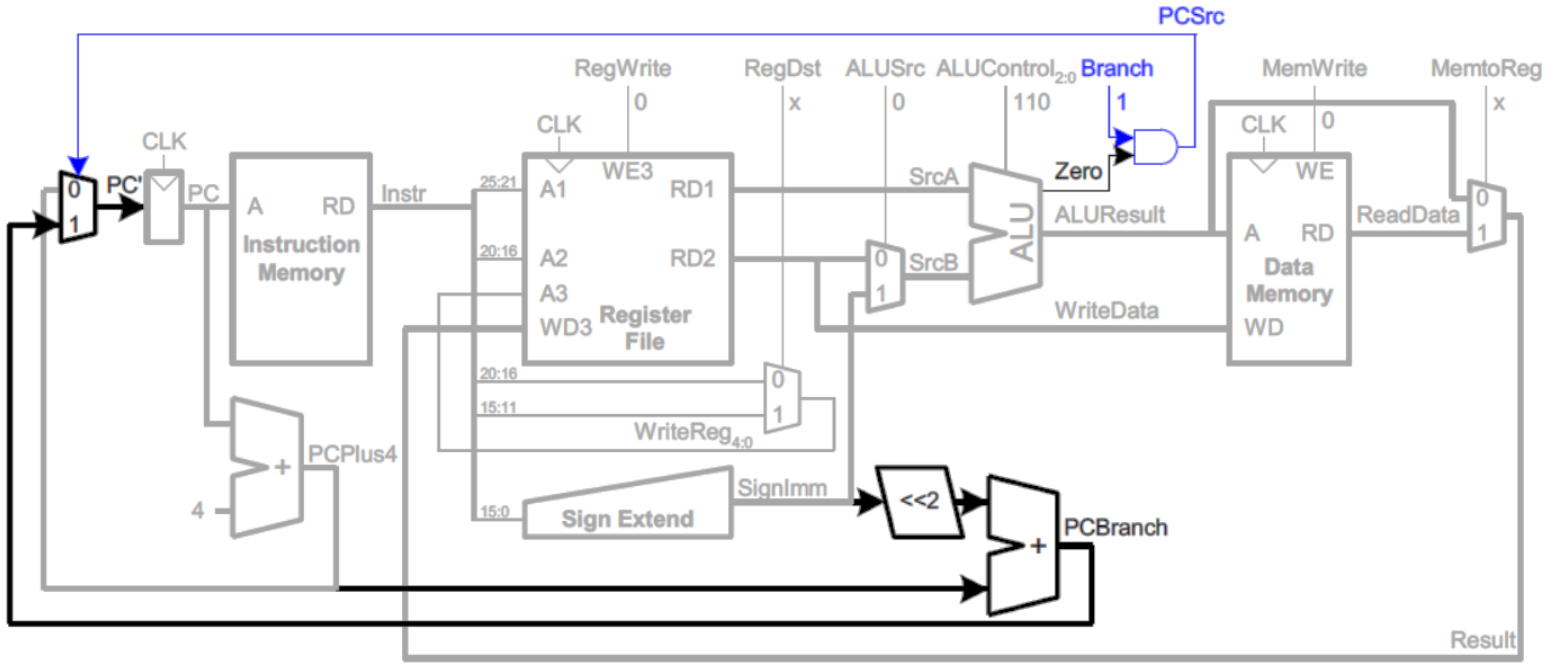
- add rd,rs,rt
- sub rd,rs,rt
- and rd,rs,rt
- or rd,rs,rt
- slt rd,rs,rt



Branch instructions

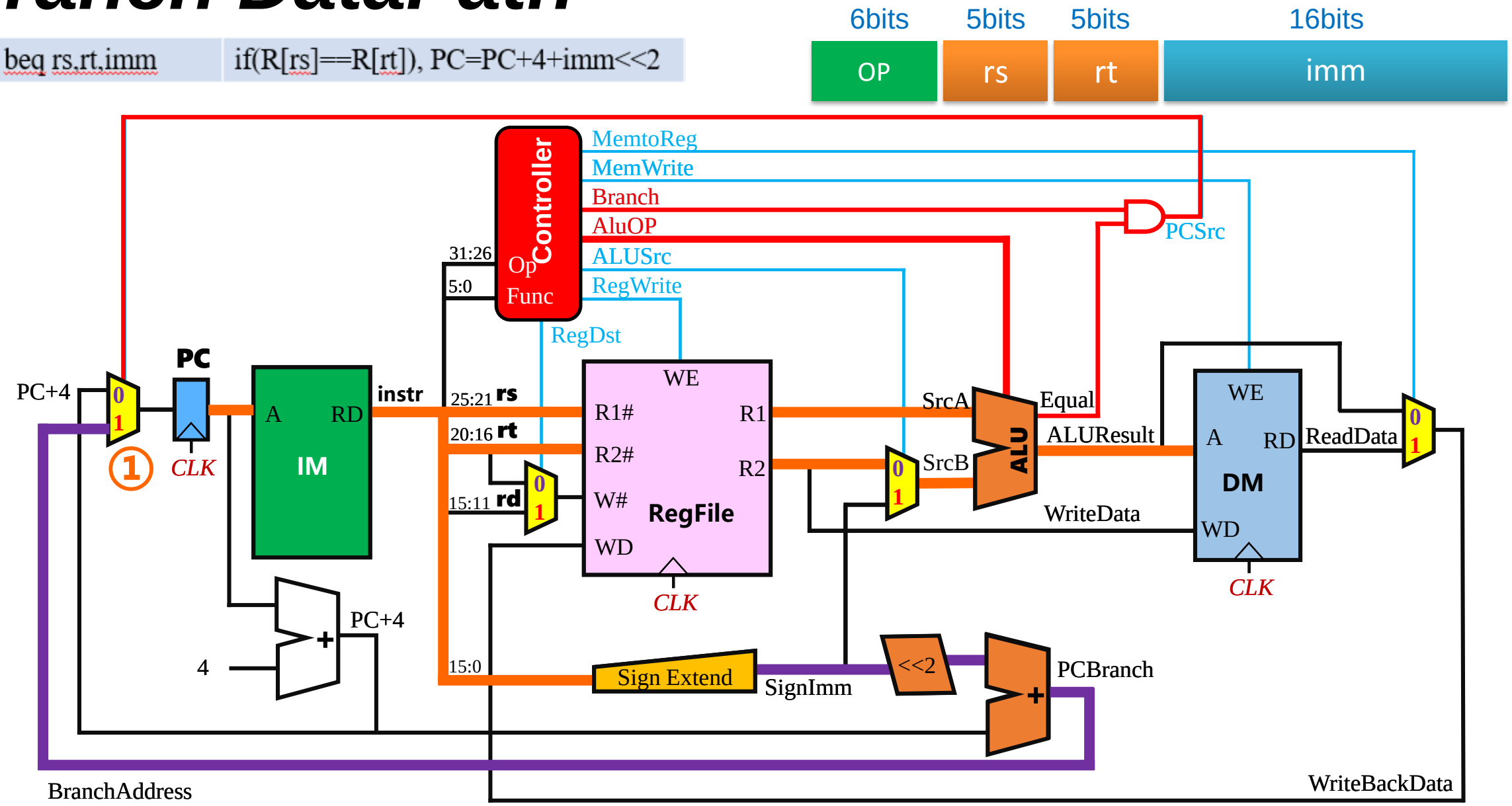


汇编格式： BEQ rs, rt, offset

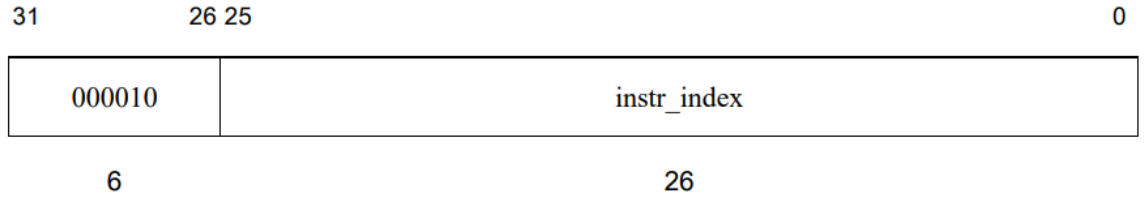


Branch DataPath

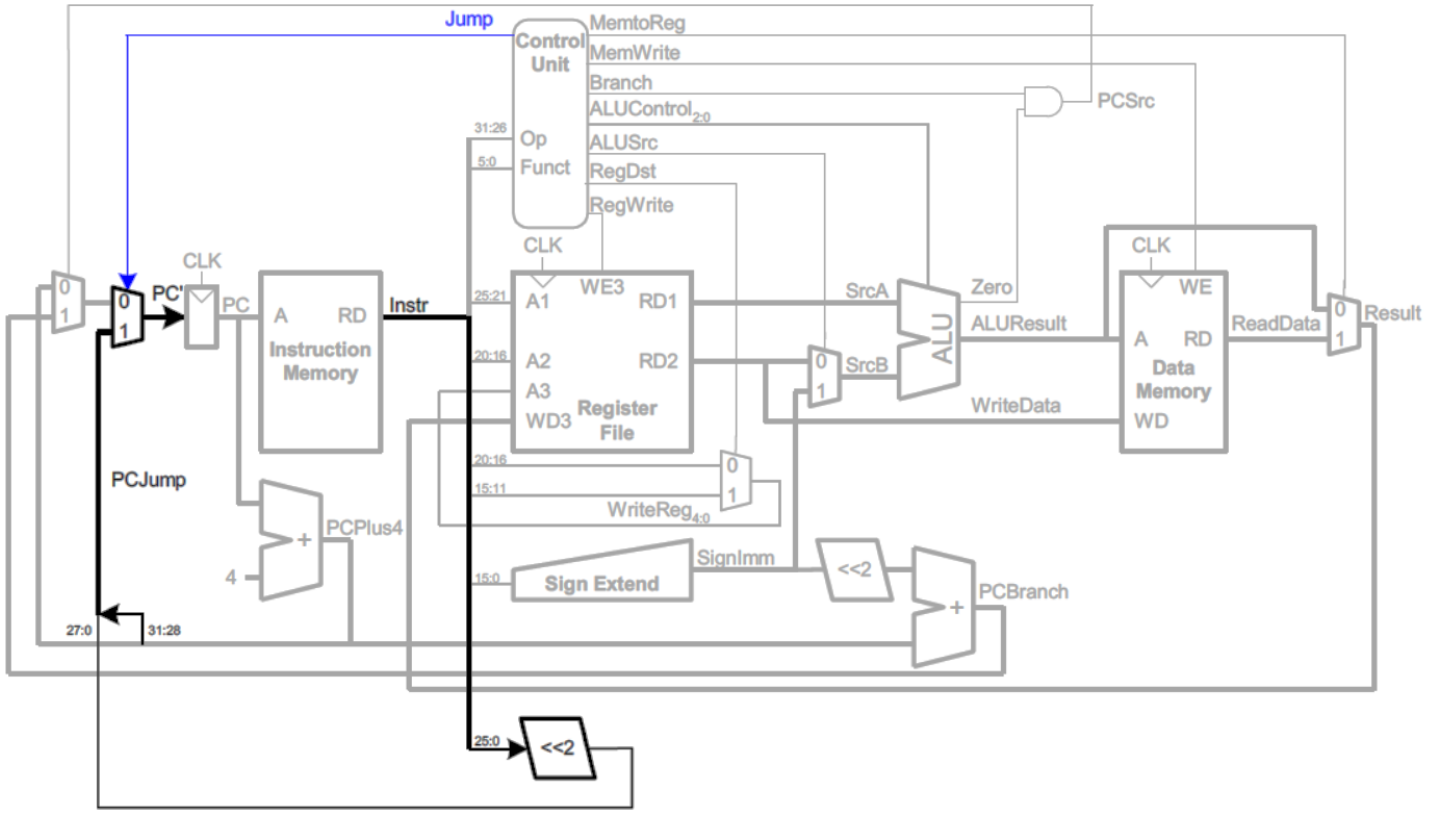
beq rs,rt,imm	if(R[rs]==R[rt]), PC=PC+4+imm<<2



Jump instructions

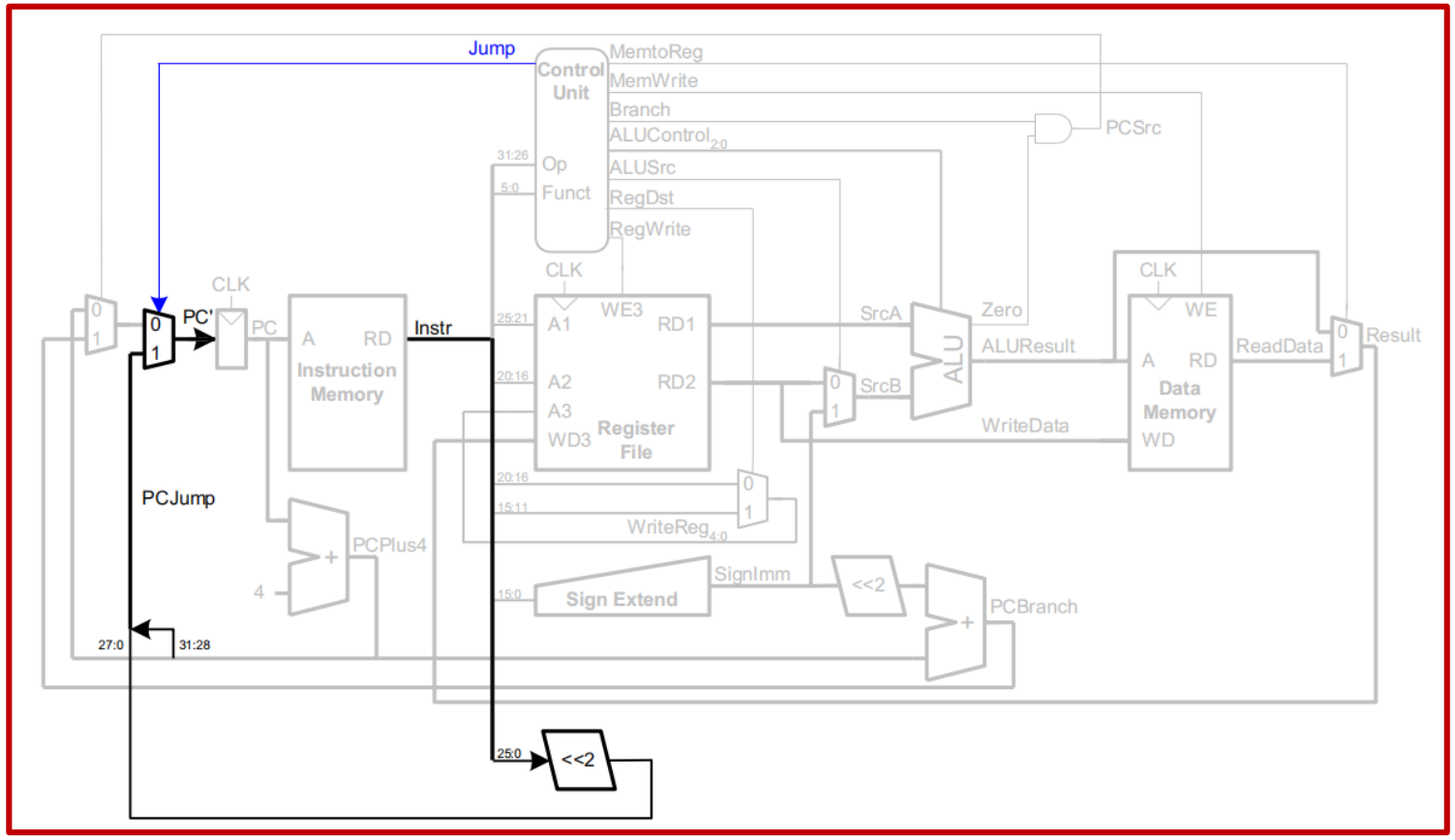


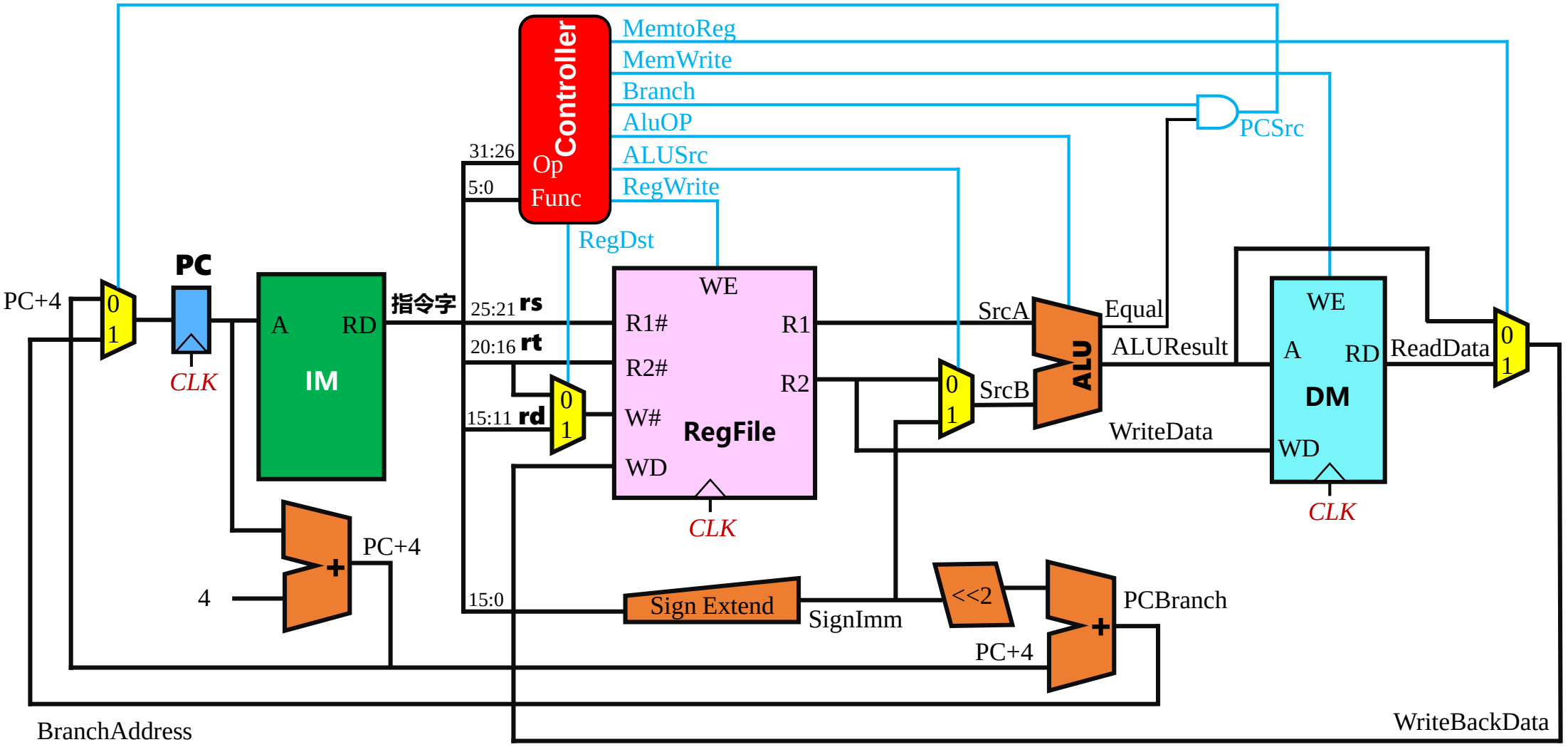
汇编格式：J target



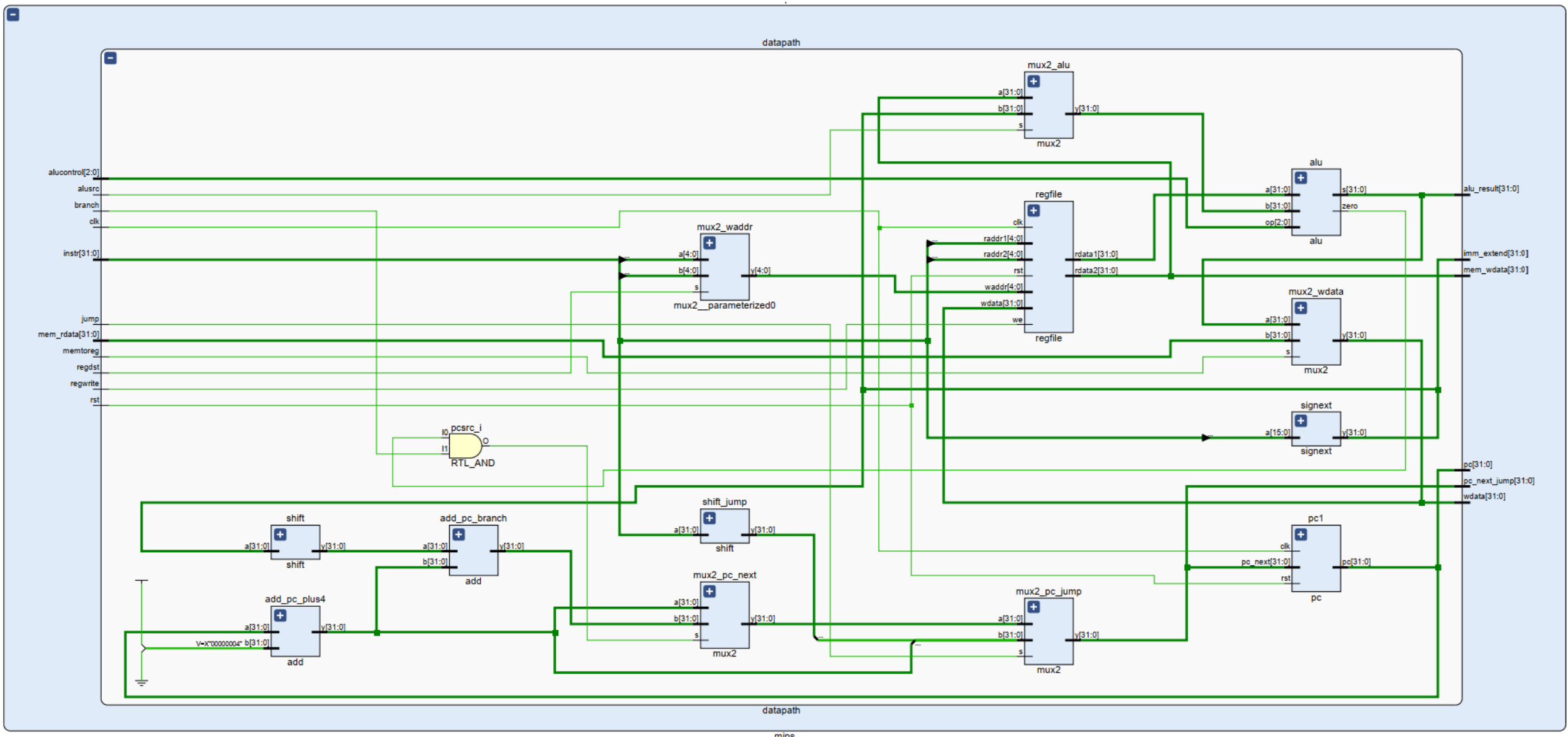
Build DataPath

The **DataPath** is composed of a **PC**, a two-way selector(**MUX**), an **Adder**, a **Register File**, a **Sign extend**, a **shifter**, and an **ALU**. These devices are connected together to realize the ***lw, sw, R-type, branch, jump, I-type*** instruction path.




Controller
Op
Func
MemtoReg
MemWrite
Branch
AluOP
ALUSrc
RegWrite
RegDst
PCSrc
31:26
5:0
PC
PC+4
CLK
A
RD
IM
指令字
25:21 rs
20:16 rt
15:11 rd
WE
R1#
R2#
W#
WD
RegFile
R1
R2
CLK
SrcA
SrcB
ALU
Equal
ALUResult
WriteData
WE
A
RD
DM
WD
CLK
ReadData
PC+4
4
15:0
Sign Extend
SignImm
<<2
PC+4
PCBranch
BranchAddress
WriteBackData

Build DataPath



Build DataPath

```
module datapath(
input wire clk,rst,
input wire [31:0] instr,mem_rdata,
output wire [31:0] pc,alu_result,mem_wdata,imm_extend,wdata,pc_next_jump,
input wire regdst,branch,regwrite,alusrc,jump,memtoreg,
input wire [2:0] alucontrol
);

wire [31:0] pc_plus4,pc_next,rdata1,rdata2,alu_srcB,imm_sl2,pc_branch,instr_sl2;
wire [4:0] write2reg;
wire zero,pcsrc;

assign mem_wdata=rdata2;
assign pcsrc = zero & branch;

mux2 #(32) mux2_pc_next(.s(pcsrc),.a(pc_plus4),.b(pc_branch),.y(pc_next));

shift shift_jump(.a(instr),.y(instr_sl2));
```

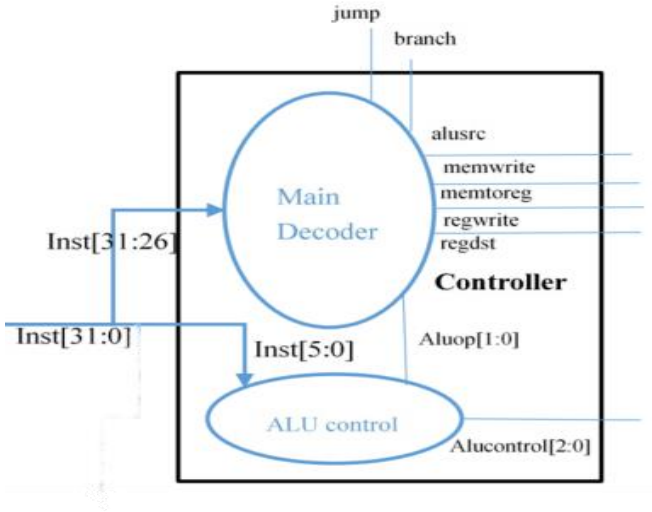
```
mux2 #(32) mux2_pc_jump(.s(jump),.a(pc_next),.b({pc_plus4[31:28],instr_sl2[27:0]}),
.y(pc_next_jump));

pc pc1(.clk(clk),.rst(rst),.pc_next(pc_next_jump),.pc(pc));

add add_pc_plus4(.a(pc),.b(32'h4),.y(pc_plus4));

shift shift(.a(imm_extend),.y(imm_sl2));

add add_pc_branch(.a(imm_sl2),.b(pc_plus4),.y(pc_branch));

signext signext(.a(instr[15:0]),.y(imm_extend));

mux2 #(32) mux2_wdata(.s(memtoreg),.a(alu_result),.b(mem_rdata),.y(wdata));

mux2 #(5) mux2_waddr(.s(regdst),.a(instr[20:16]),.b(instr[15:11]),.y(write2reg));

regfile regfile(.clk(clk),.rst(rst),.raddr1(instr[25:21]),.rdata1(rdata1),
.raddr2(instr[20:16]),.rdata2(rdata2),
.we(regwrite),.waddr(write2reg),.wdata(wdata));


mux2 #(32) mux2_alu(.s(alusrc),.a(rdata2),.b(imm_extend),.y(alu_srcB));

alu alu(.a(rdata1),.b(alu_srcB),.op(alucontrol),.s(alu_result),.zero(zero));

endmodule
```


南通大學
NANTONG UNIVERSITY
Step 3: MIPS CPU

Build Controller

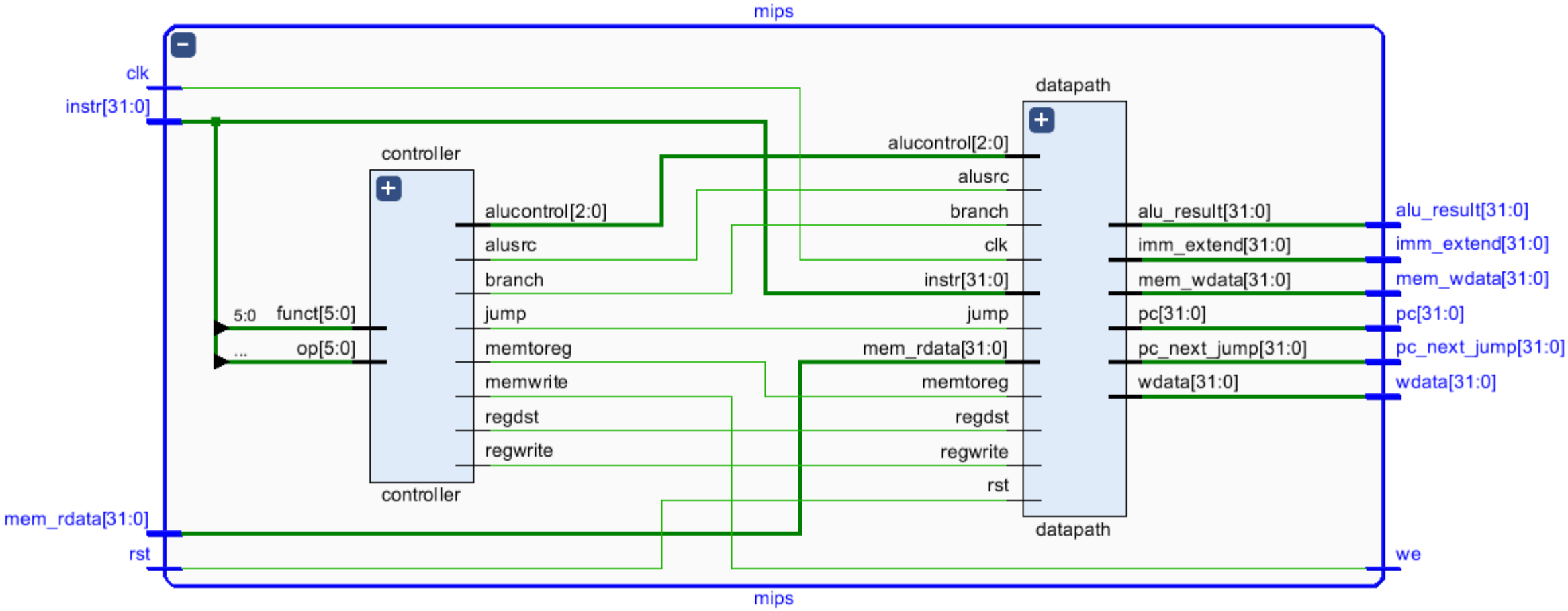
```
module main_dec(
    input wire [5:0] op,
    output wire regdst,branch,regwrite,alusrc,memwrite,jump,memtoreg,
    output wire [1:0] aluop
    );
    reg[8:0] controls;
    assign {regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump,aluop} = controls;
    always @(*) begin
        case (op)
            6'b000000:controls <= 9'b110000010;//R-TYRE
            6'b100011:controls <= 9'b101001000;//LW
            6'b101011:controls <= 9'b001010000;//SW
            6'b000100:controls <= 9'b000100001;//BEQ
            6'b001000:controls <= 9'b101000000;//ADDI
            6'b000010:controls <= 9'b000000100;//J
            default:  controls <= 9'b000000000;//illegal op
        endcase
    end
endmodule
module alu_dec(
    input wire[5:0] funct,
    input wire[1:0] aluop,  output reg[2:0] alucontrol
    );
    always @(*) begin
        case (aluop)
            2'b00: alucontrol <= 3'b010;//add (for lw/sw/addi)
            2'b01: alucontrol <= 3'b110;//sub (for beq)
            default : case (funct)
                6'b100000:alucontrol <= 3'b010; //add
                6'b100010:alucontrol <= 3'b110; //sub
                6'b100100:alucontrol <= 3'b000; //and
                6'b100101:alucontrol <= 3'b001; //or
                6'b101010:alucontrol <= 3'b111; //slt
                6'b100110:alucontrol <= 3'b011; //xor
                default:  alucontrol <= 3'b000;
            endcase
        endcase
    end
endmodule
```

Controller consists of **Main Decoder** and **ALU decoder** modules, which decodes input instructions and gives various control and operation signals.




```
module controller(
    input wire [5:0] op,funct,
    output wire regdst,branch,regwrite,alusrc,memwrite,jump,memtoreg,
    output [2:0] alucontrol
    );
    wire [1:0] aluop;
    main_dec md(op,regdst,branch,regwrite,alusrc,memwrite,jump,memtoreg,aluop);
    alu_dec ad(funct,aluop,alucontrol);
endmodule
```

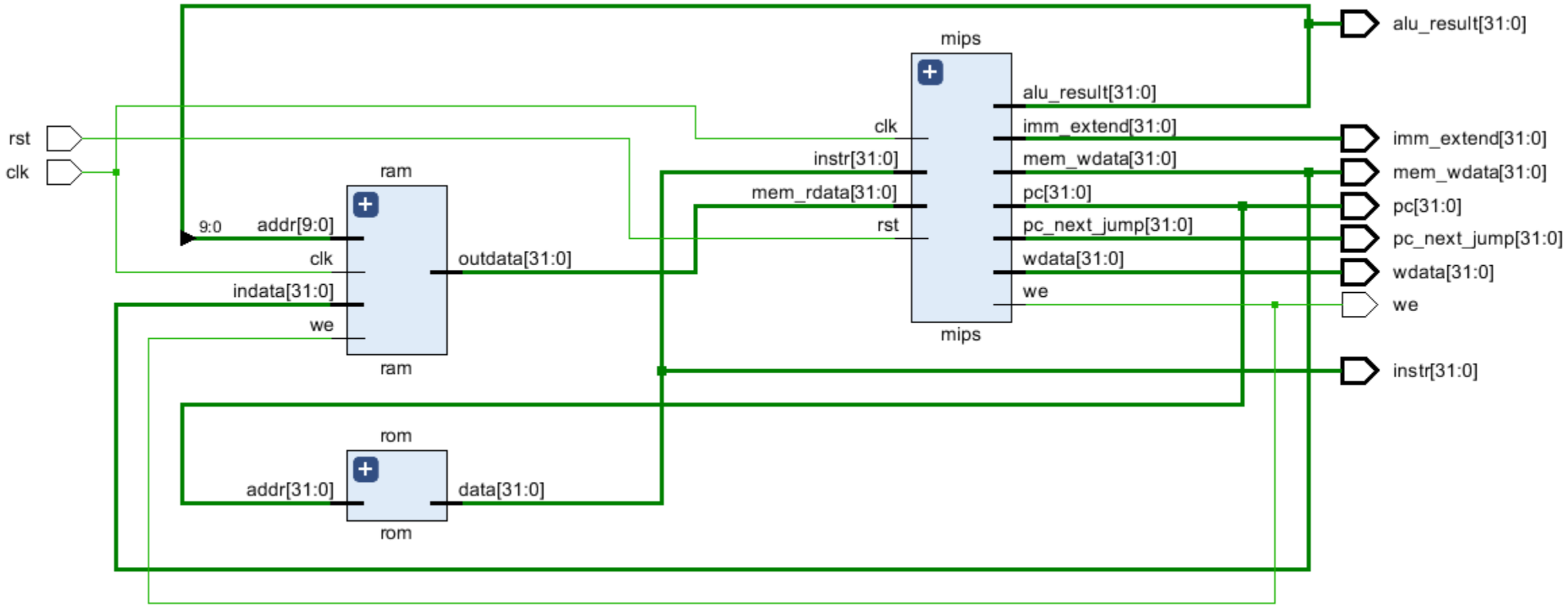
Build MIPS-CPU



Build MIPS-CPU

```
module mips(
    input wire clk,rst,
    output wire we,
    output wire [31:0] pc,alu_result,mem_wdata,imm_extend,pc_next_jump,wdata,
    input wire [31:0] instr,mem_rdata
);
wire regdst,branch,regwrite,alusrc,jump,memtoreg,memwrite;
wire [2:0] alucontrol;

datapath datapath(.clk(clk),.rst(rst),.instr(instr),.mem_rdata(mem_rdata),.pc(pc),
.alu_result(alu_result),
.mem_wdata(mem_wdata),.imm_extend(imm_extend),.pc_next_jump(pc_next_jump),.wdata(wdata)
,.regdst(regdst),
    .branch(branch),.regwrite(regwrite),.alusrc(alusrc),.jump(jump),.memtoreg(memtoreg)
,.alucontrol(alucontrol)
);

controller controller(.op(instr[31:26]),.funct(instr[5:0]),.regdst(regdst),.branch(bran
ch),.regwrite(regwrite),
    .alusrc(alusrc),.alucontrol(alucontrol),.memwrite(we),.jump(jump),
    .memtoreg(memtoreg)
);

endmodule
```

Build TOP module (Model machine)



Build TOP module (Model machine)

```
module top(
    input wire clk,rst,
    output wire [31:0] pc,instr,mem_wdata,imm_extend,alu_result,wdata,pc_next_jump,
    output we
    );

wire [31:0] mem_rdata;
mips mips(.clk(clk),.rst(rst),.we(we),.pc(pc),.alu_result(alu_result),
    .mem_wdata(mem_wdata),.imm_extend(imm_extend),.pc_next_jump(pc_next_jump),
    .wdata(wdata),.instr(instr),.mem_rdata(mem_rdata)
);

rom rom(.addr(pc),.data(instr)
);

ram ram(.clk(clk),.we(we),.addr(alu_result[9:0]),.indata(mem_wdata),.outdata(mem_rdata)
);
endmodule

```

南通大學
NANTONG UNIVERSITY

Step 4: Simulation

MIPS ISA

Logical operation instruction

Arithmetic operation instruction

Shift instruction

Data move instruction

Branch instruction

Memory access instruction

	31:26	25:21	20:16	15:11	10:6	5:0	
逻辑运算指令	000000	rs	rt	rd	00000	100100	and rd,rs,rt
	000000	rs	rt	rd	00000	100101	or rd,rs,rt
	000000	rs	rt	rd	00000	100110	xor rd,rs,rt
	000000	rs	rt	rd	00000	100111	nor rd,rs,rt
	001100	rs	rt	immediate			andi rt,rs,immediate
	001110	rs	rt	immediate			xori rt,rs,immediate
	001111	00000	rt	immediate			lui rt,immediate
	001101	rs	rt	immediate			ori rs,rt,immediate
移位指令	000000	00000	rt	rd	sa	000000	sll rd,rt,sa
	000000	00000	rt	rd	sa	000010	srl rd,rt,sa
	000000	00000	rt	rd	sa	000011	sra rd,rt,sa
	000000	rs	rt	rd	00000	000100	sllv rd,rt,rs
	000000	rs	rt	rd	00000	000110	srlv rd,rt,rs
	000000	rs	rt	rd	00000	000111	srav rd,rt,rs
数据移动指令	000000	00000	00000	rd	00000	010000	mfhi rd
	000000	00000	00000	rd	00000	010010	mflo rd
	000000	rs	00000	00000	00000	010001	mthi rs
	000000	rs	00000	00000	00000	010011	mtlo rs

算术运算指令	000000	rs	rt	rd	00000	100000	add rd,rs,rt
	000000	rs	rt	rd	00000	100001	addu rd,rs,rt
	000000	rs	rt	rd	00000	100010	sub rd,rs,rt
	000000	rs	rt	rd	00000	100011	subu rd,rs,rt
	000000	rs	rt	rd	00000	101010	slt rd,rs,rt
	000000	rs	rt	rd	00000	101011	sltu rd,rs,rt
	000000	rs	rt	00000	00000	011000	mult rs,rt
	000000	rs	rt	00000	00000	011001	multu rs,rt
	000000	rs	rt	00000	00000	011010	div rs,rt
	000000	rs	rt	00000	00000	011011	divu rs,rt
	001000	rs	rt	immediate			addi rt,rs,immediate
	001001	rs	rt	immediate			addiu rt,rs,immediate
	001010	rs	rt	immediate			slti rt,rs,immediate
	001011	rs	rt	immediate			sltiu rt,rs,immediate
分支跳转指令	000000	rs	00000	00000	00000	001000	jr rs
	000000	rs	00000	rd	00000	001001	jalr rs/jalr rd,rs
	000010	instr_index					j target
	000011	instr_index					jal target
	000100	rs	rt	offset			beq rs,rt,offset
	000111	rs	00000	offset			bgtz rs,offset
	000110	rs	00000	offset			blez rs,offset
	000101	rs	rt	offset			bne rs,rt,offset
	000001	rs	00000	offset			bltz rs,offset
	000001	rs	10000	offset			bltzal rs,offset
	000001	rs	00001	offset			bgez rs,offset
	000001	rs	10001	offset			bgezal rs,offset
访存指令	100000	base	rt	offset			lb rt,offset(base)
	100100	base	rt	offset			lbu rt,offset(base)
	100001	base	rt	offset			lh rt,offset(base)
	100101	base	rt	offset			lhu rt,offset(base)
	100011	base	rt	offset			lw rt,offset(base)
	101000	base	rt	offset			sb rt,offset(base)
	101001	base	rt	offset			sh rt,offset(base)
	101011	base	rt	offset			sw rt,offset(base)

MIPS ISA

Funct	Instruction	Function
32	add rd,rs,rt	R[rd]=R[rs]+R[rt]
34	sub rd,rs,rt	R[rd]=R[rs]-R[rt]
36	and rd,rs,rt	R[rd]=R[rs]&R[rt]
37	or rd,rs,rt	R[rd]=R[rs] | R[rt]
42	slt rd,rs,rt	R[rd]=(R[rs]<R[rt])?1:0

OP	Instruction	Function
04	beq rs,rt,imm	if(R[rs]==R[rt]), PC=PC+4+imm<<2
08	addi rt,rs,imm	R[rt]=R[rs]+imm
35	lw rt,imm(rs)	R[rt]=M[R[rs]+imm]
43	sw rt,imm(rs)	M[R[rs]+imm]=R[rt]

OP	Instruction	Function
02	j address	PC {$(PC+4)_{31:28}$, address,00}

Test program

#	Assembly	Description	Address	Machine	
main:	addi $2, $0, 5	# initialize $2 = 5	1	20020005	20020005
	addi $3, $0, 12	# initialize $3 = 12	2	2003000c	2003000c
	addi $7, $3, −9	# initialize $7 = 3	3	2067fff7	2067fff7
	or $4, $7, $2	# $4 <= 3 or 5 = 7	4	00e22025	00e22025
	and $5, $3, $4	# $5 <= 12 and 7 = 4	5	00642824	00642824
	add $5, $5, $4	# $5 = 4 + 7 = 11	6	00a42820	00a42820
	beq $5, $7, end	# shouldn't be taken	7	10a7000a	10a7000a
	slt $4, $3, $4	# $4 = 12 < 7 = 0	8	0064202a	0064202a
	beq $4, $0, around	# should be taken	9	10800001	10800001
	addi $5, $0, 0	# shouldn't happen	10	20050000	20050000
around:	slt $4, $7, $2	# $4 = 3 < 5 = 1	11	00e2202a	00e2202a
	add $7, $4, $5	# $7 = 1 + 11 = 12	12	00853820	00853820
	sub $7, $7, $2	# $7 = 12 − 5 = 7	13	00e23822	00e23822
	sw $7, 68($3)	# [80] = 7	14	ac670044	ac670044
	lw $2, 80($0)	# $2 = [80] = 7	15	8c020050	8c020050
	j end	# should be taken	16	08000011	08000011
	addi $2, $0, 1	# shouldn't happen	17	20020001	20020001
end:	sw $2, 84($0)	# write adr 84 = 7	18	ac020054	ac020054

ROM & RAM

```
#             Assembly
main:       addi $2, $0, 5
            addi $3, $0, 12
            addi $7, $3, -9
            or   $4, $7, $2
            and  $5, $3, $4
            add  $5, $5, $4
            beq  $5, $7, end
            slt  $4, $3, $4
            beq  $4, $0, around
            addi $5, $0, 0
around:     slt  $4, $7, $2
            add  $7, $4, $5
            sub  $7, $7, $2
            sw   $7, 68($3)
            lw   $2, 80($0)
            j    end
            addi $2, $0, 1
end:        sw   $2, 84($0)
```

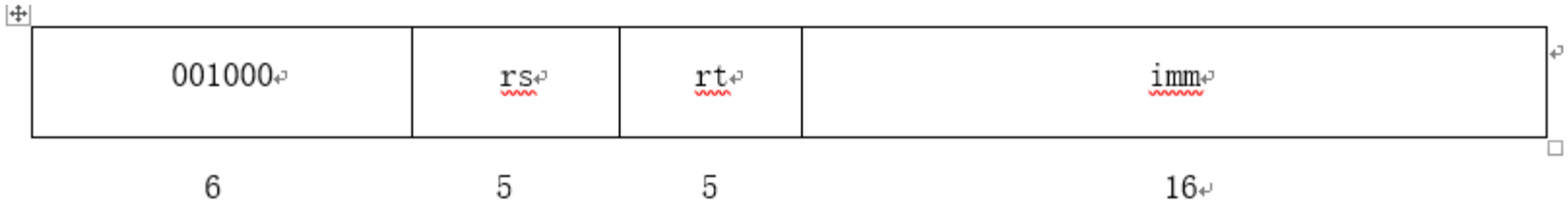
```
module rom(
    input [31:0] addr,
    output [31:0] data
    );
    reg[31:0] romdata;
    always @(*)
    case(addr[31:2])
    5'h0:romdata=32'h20020005;
    5'h1:romdata=32'h2003000c;
    5'h2:romdata=32'h2067fff7;
    5'h3:romdata=32'h00e22025;
    5'h4:romdata=32'h00642824;
    5'h5:romdata=32'h00a42820;
    5'h6:romdata=32'h10a7000a;
    5'h7:romdata=32'h0064202a;
    5'h8:romdata=32'h10800001;
    5'h9:romdata=32'h20050000;
    5'ha:romdata=32'h00e2202a;
    5'hb:romdata=32'h00853820;
    5'hc:romdata=32'h00e23822;
    5'hd:romdata=32'hac670044;
    5'he:romdata=32'h8c020050;
    5'hf:romdata=32'h08000011;
    5'h10:romdata=32'h20020001;
    5'h11:romdata=32'hac020054;       5'h12:romdata=32'h00441826;
    default:romdata=32'h0;
    endcase
    assign data=romdata;
endmodule
```

```
module ram(clk,we,addr,indata,outdata);
input clk,we;
input [31:0] indata;
input [9:0] addr;
output  [31:0]outdata;
reg [31:0] ram[1023:0];
integer i;
initial begin
    for(i=0;i<1024;i=i+1)
        ram[i]=32'b0;
end
always@(posedge clk)
if(we) begin
    ram[addr]<=indata;
end
assign outdata=ram[addr];
endmodule
```

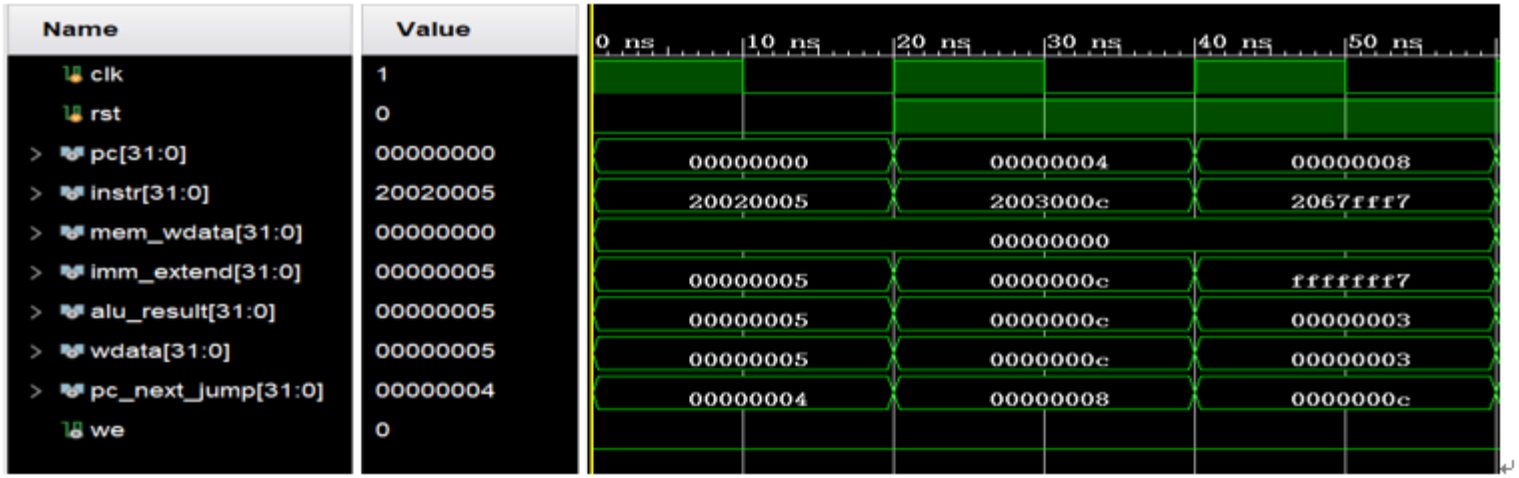
Test program

```
#           Assembly
main:       addi $2, $0, 5
            addi $3, $0, 12
            addi $7, $3, -9
            or   $4, $7, $2
            and  $5, $3, $4
            add  $5, $5, $4
            beq  $5, $7, end
            slt  $4, $3, $4
            beq  $4, $0, around
            addi $5, $0, 0
around:     slt  $4, $7, $2
            add  $7, $4, $5
            sub  $7, $7, $2
            sw   $7, 68($3)
            lw   $2, 80($0)
            j    end
            addi $2, $0, 1
end:        sw   $2, 84($0)
```

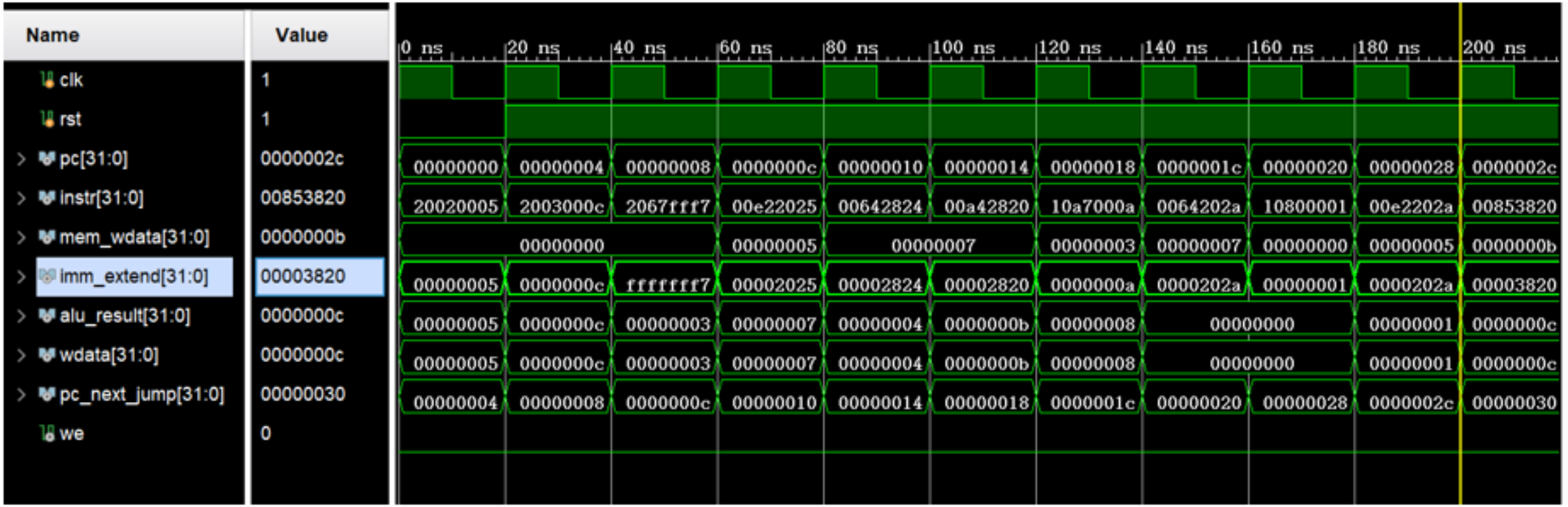
addi instruction: Add the value of register rs to the immediate value imm extended to 32 bits with a sign, and write the result to register rt. If an overflow occurs, an integer overflow exception (IntegerOverflow) is triggered.



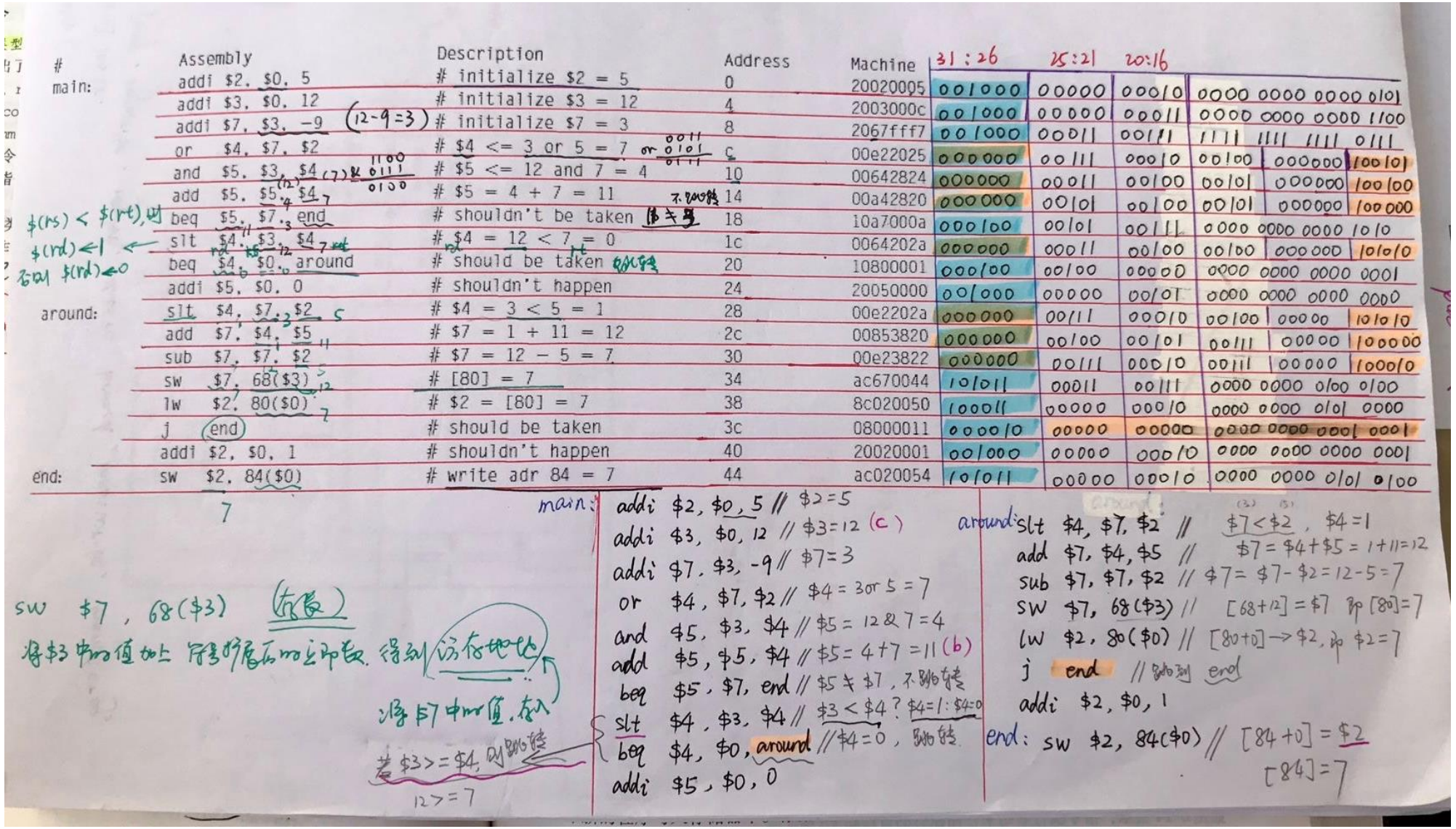
Simulation result analysis



All simulation results



All simulation results



#	Assembly	Description	Address	Machine
main:	addi $2, $0, 5	# initialize $2 = 5	0	20020005
	addi $3, $0, 12	# initialize $3 = 12	4	2003000c
	addi $7, $3, -9	# initialize $7 = 3	8	2067fff7
	or $4, $7, $2	# $4 <= 3 or 5 = 7	c	00e22025
	and $5, $3, $4	# $5 <= 12 and 7 = 4	10	00642824
	add $5, $5, $4	# $5 = 4 + 7 = 11	14	00a42820
	beq $5, $7, end	# shouldn't be taken	18	10a7000a
	slt $4, $3, $4	# $4 = 12 < 7 = 0	1c	0064202a
	beq $4, $0, around	# should be taken	20	10800001
	addi $5, $0, 0	# shouldn't happen	24	20050000
around:	slt $4, $7, $2	# $4 = 3 < 5 = 1	28	00e2202a
	add $7, $4, $5	# $7 = 1 + 11 = 12	2c	00853820
	sub $7, $7, $2	# $7 = 12 - 5 = 7	30	00e23822
	sw $7, 68($3)	# [80] = 7	34	ac670044
	lw $2, 80($0)	# $2 = [80] = 7	38	8c020050
	j end	# should be taken	3c	08000011
	addi $2, $0, 1	# shouldn't happen	40	20020001
end:	sw $2, 84($0)	# write adr 84 = 7	44	ac020054

31:26 | 25:21 | 20:16

```
main: addi $2, $0, 5 // $2=5
      addi $3, $0, 12 // $3=12 (c)
      addi $7, $3, -9 // $7=3
      or   $4, $7, $2 // $4=3 or 5=7
      and  $5, $3, $4 // $5=12 & 7=4
      add  $5, $5, $4 // $5=4+7=11 (b)
      beq  $5, $7, end // $5≠$7, 不跳转
      slt  $4, $3, $4 // $3<$4? $4=1:$4=0
      beq  $4, $0, around // $4=0, 跳转
      addi $5, $0, 0
around: slt  $4, $7, $2 // $7<$2, $4=1
      add  $7, $4, $5 // $7=$4+$5=1+11=12
      sub  $7, $7, $2 // $7=$7-$2=12-5=7
      sw   $7, 68($3) // [68+12]=$7 即[80]=7
      lw   $2, 80($0) // [80+0]->$2, 即 $2=7
      j    end // 跳到 end
      addi $2, $0, 1
end:  sw   $2, 84($0) // [84+0]=$2
                      [84]=7
```

Schedule

Week 6
2024.10.9, jx03-103,18:30-21:00

- Course design content and requirements；
- Course Design Student Grouping.

Week 7
2024.10.16, jsj5-511,18:00-21:00

- Install VIVADO software;
- Learn Verilog HDL;
- Build each individual module;
- Verification of each module by simulation.

Week 8
2024.10.23, jsj5-511,18:00-21:00

- **Build DataPath;**
- **Build MIPS-CPU.**

Week 9
2024.10.30, jsj5-511,18:00-21:00

- Complete the simulation test;
- Complete the writing of the course project report.
  （Each design team submits one report）

Week 10
2024.11.6, jsj5-511,18:00-21:00

- **Course design project acceptance (project explanation, code demonstration, simulation test, report reading)**