南通大學人工智能与计算机学院 (阿里云大数据学院) (人工智能研究院合署)
School of Artificial Intelligence and Computer Science, Nantong University

Computer Composition Principles Course Design

REPORT ON A

MIPS SINGLE-CYCLE CPU IMPLEMENTATION

Supervisor

Dr. Cheng Chun

School of Artificial Intelligence and Computer Science
Nantong University

2024 年 11 月 6 日

I. Purpose

1. Strengthen Understanding of Computer Systems: This course helps students build a solid foundation in the principles of computer organization. Through practical application of concepts, they will better understand how different mechanisms of a computer interact. It also emphasizes the idea of "time and space" in computing, giving students a clearer picture of how the entire system functions as a whole.

2. Improve Design, Problem-Solving, and Teamwork Skills: Students will gain experience in designing and troubleshooting computer systems, developing the ability to conduct independent research. The course offers practical engineering experience and prepares students to solve complex problems in computing. It also emphasizes the importance of teamwork by preparing students to work effectively in groups, take on leadership roles, and complete assigned tasks independently.

II. Team

The team 5 (**TEAM ALPHA)** members and division of labor are shown in the table below.

No.	**Student ID**	**Name**	**Role**	**Contributions**
1	2130130203	AHMED MD SHAKIL	Leader	Coordinated team tasks, supervised CPU design, implemented the control unit, and led troubleshooting.
2	2130130204	VASKAR CHAKMA	Member	Developed the ALU, integrated it into the CPU, set up test environments, and ran simulations.
3	2130130233	AMIN MISBAHUL	Member	Contributed to documentation, finalized project reports, and assisted in testing the CPU.
4	2130130222	ROUF ABDUR	Member	Handled opcode decoding, and tested integration.
5	2130130224	SUNNY BARUA	Member	Integrated the data memory module for load/store instructions.
6	2130130228	PROTIK MD KUDRUTUZZAMAN	Member	Focused on testing and debugging.

III. Tasks

1．Basic design tasks

（1）Build the data path, realize the controller, memory and external storage devices.

（2）Combine the data path and controller implemented by task (1) into a single-cycle CPU with MIPS instructions, and then combine it with the simplified memory and external storage implemented by (1) to form a simple computer, realizing automatic instruction fetching, decoding and execution by the CPU.

（3）Write a simulation stimulus program to test whether the CPU and computer functions are normal and handle errors.

IV. Conditions for Course Design Completion

A computer with EDA tool software (Vivado) installed.

V. Implementation of various functional devices

1. Register stack: Use clk clock signal, rst reset signal and we write enable signal to control the writing of waddr address data wdata, and then read out the corresponding data rdata1 and rdata2 according to the different addresses of raddr1 and raddr2. When rst is valid at low level, the values in the register are set to 0; when rst is invalid at high level and we write enable signal is valid, the data of a register where a waddr address is located can be written, and then the data in the register is read out through the addresses of raddr1 and raddr2, and then the subsequent operations are completed.

2. ALU operator: Use different values of the alucontrol field to specify what operation to perform on the two data. The value of the alucontrol field is generated by decoding from the op field of the controller controller.

3. Shift module: The most important operation used in this experiment is to shift a data left by two bits, that is, it can be completed using the most basic syntax of the Verilog language.

4. Adder: This has been done in the previous digital logic experiment. Here, the main purpose is to complete the +4 operation of pc in order to obtain the address of the next instruction. Another place is to use the adder to calculate pcbranch, that is, if the current instruction is a branch instruction, jump to the corresponding address after calculation.

5. PC counter: This is essentially a simple D flip-flop with a clk clock signal and a rst reset signal. It needs to be initialized and reset at the beginning of the experiment.

6. Two-way selector: Use an s signal to determine which value to output, a or b. As far as the previous experiment is concerned, when s=1, select output b, otherwise select output a. In this comprehensive design, this module will be used in five places.

7. Instruction register: This is essentially a rom read-only memory. It does not require a clock signal. It fetches the instruction at the corresponding position based on the input address and then sends it to other components.

8. Data memory: This is essentially a ram read-write memory, which can be read and written. However, when writing, it is necessary to cooperate with the clk clock signal and the we write signal to control the write operation. The we write signal is determined by the memwrite signal generated by decoding from the controller op field.

9. Controller: A more difficult module in this experiment. It decodes the high five bits of the instruction read from the instruction memory as op to obtain the control signals of the corresponding components (memtoreg, memwrite, branch, alusrc, regdst, regwrite and alucontrol execution operation signal). Note the conditions for the execution of the branch branch jump instruction: the zero signal and the branch signal are valid at the same time to execute.

10. Sign extension: Expand the original 16-bit data to 32-bit data. The expansion method is to use the highest bit as the sign bit and expand according to the sign bit.

Mips single cycle instruction table: MIPS-ISA

7	25:21	20:16	15:11	10:6	5:0	
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	000000	rt	immediate			lui rt, immediate
001101	rs	rt	immediate			ori rs, rt, immediate
000000	000000	rt	rd	sa	000000	sll rd, rt, sa
000000	000000	rt	rd	sa	000010	srl rd, rt, sa
000000	000000	rt	rd	sa	000011	sra rd, rt, sa
000000	rs	rt	rd	000000	000100	sllv rd, rt, rs

000000	rs	rt	rd	000000	000110	srlv rd, rt, rs
000000	rs	rt	rd	000000	000111	srav rd, rt, rs
000000	000000	000000	rd	000000	010000	mfhi rd
000000	000000	000000	rd	000000	010010	mflo rd
000000	rs	000000	000000	000000	010001	mthi rs
000000	rs	000000	000000	000000	010011	mtlo rs

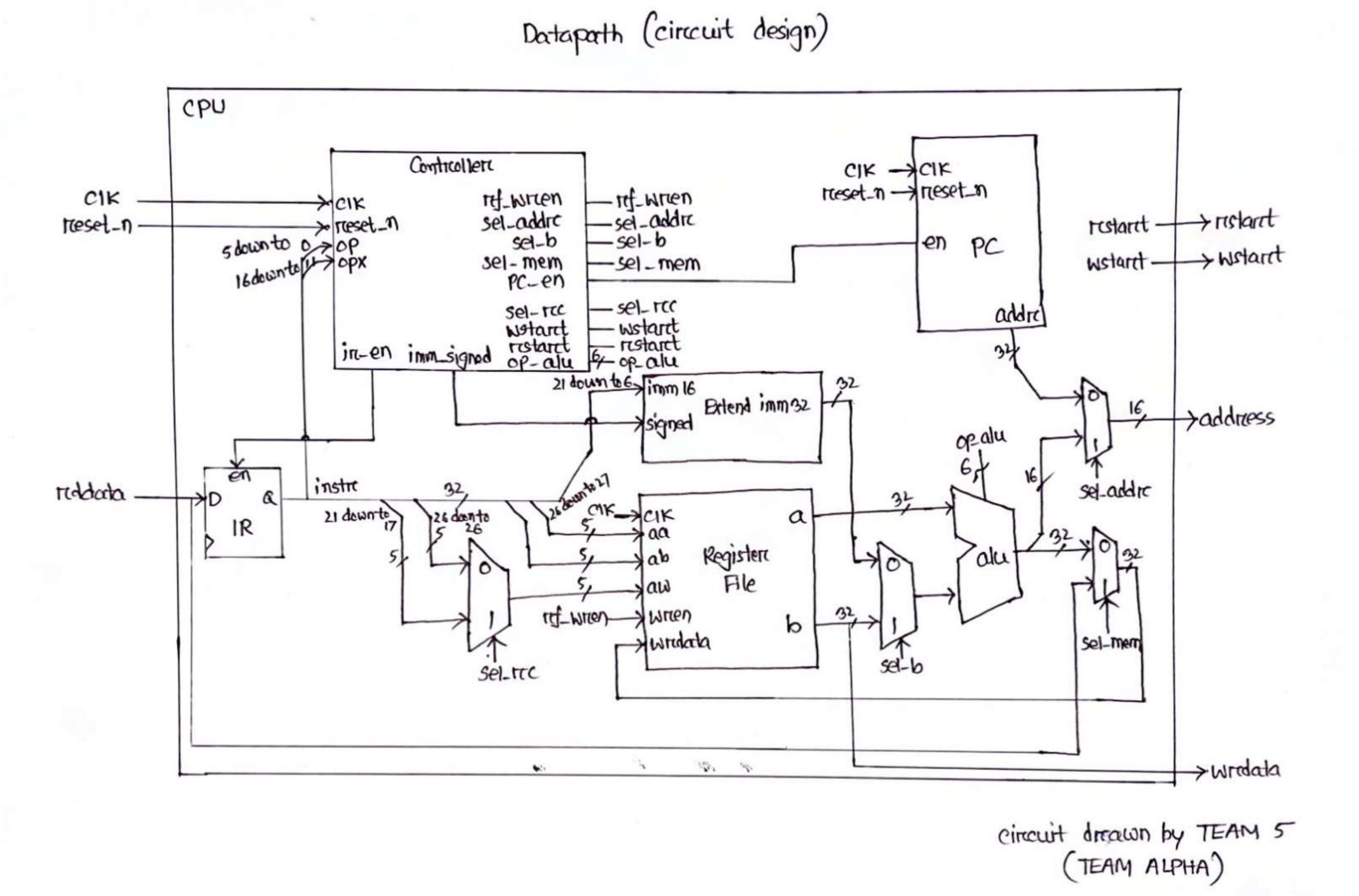
000000	rs	rt	rd	000000	100000	add rd, rs, rt
000000	rs	rt	rd	000000	100001	addu rd, rs, rt
000000	rs	rt	rd	000000	100010	sub rd, rs, rt
000000	rs	rt	rd	000000	100011	subu rd, rs, rt
000000	rs	rt	rd	000000	101010	slt rd, rs, rt
000000	rs	rt	rd	000000	101011	sltu rd, rs, rt
000000	rs	rt	000000	000000	011000	mult rs, rt
000000	rs	rt	000000	000000	011001	multu rs, rt
000000	rs	rt	000000	000000	011010	div rs, rt
000000	rs	rt	000000	000000	011011	divu rs, rt
001000	rs	rt	immediate			addi rt, rt, 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
000000	rs	000000	000000	000000	001000	jr rs
000000	rs	000000	rd	000000	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	000000	offset			bgtz rs, offset
000110	rs	000000	offset			blez rs, offset
000101	rs	rt	offset			bne rs, rt, offset
000001	rs	000000	offset			bltz rs, offset
000001	rs	100000	offset			bltzal rs, offset
000001	rs	000001	offset			bgez rs, offset
000001	rs	100001	offset			bgezal rs, offset

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)

The MIPS instructions are shown in the figure above. The Light Blue instructions are implemented this time, including and, or, add, sub, slt, addi, j, beq, lw, and sw. The following module circuits are designed according to this instruction format.

（1）Datapath

① Circuit design



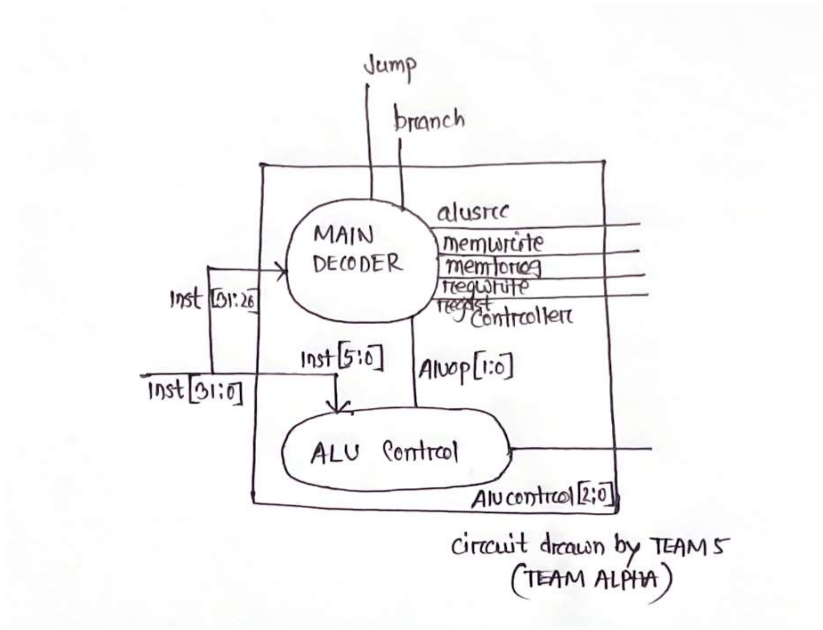
The data path consists of a program counter, a two-way selector, an adder, a register file, a signed extender, an instruction left shift by two bits, and an arithmetic unit. These devices are connected together to implement lw, sw, r-type, branch, jump, and i-type instruction paths, which are the core of the CPU.

② Code implementation

```
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
```

（2）Controller

①Circuit design



The controller consists of two modules: main decoder and alu decoder. It decodes the input instructions and gives various control and operation signals.

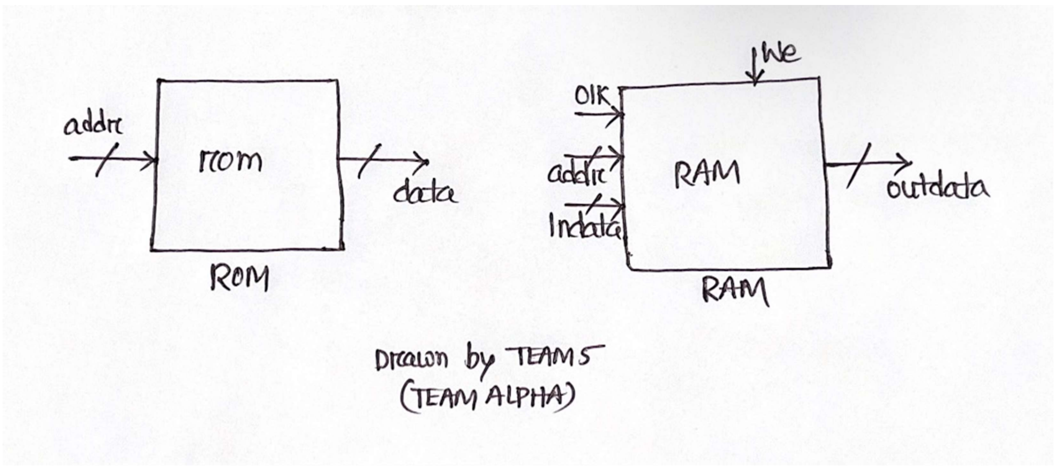
②Code implementation:

```
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
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
```

(3) Instruction RAM and memory RAM

① Circuit design



Used to store instructions and data respectively

② Code implementation

ROM:

```
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;
    5'h0: romdata = 32'h00000020;
    5'h6: romdata = 32'h20020005;
    5'hE: romdata = 32'h00000020;
    5'hB: romdata = 32'h10000004;
    5'hD: romdata = 32'h0000002A;
    5'h2: romdata = 32'h00000022; // sub rd,rs,rt
    5'h12: romdata = 32'hAC220054; // sw rt,offset(base)
    5'h8: romdata = 32'h8C220004; // lw rt,offset(base)
    5'h11: romdata = 32'h00000026; // xor rd,rs,rt
    5'hF: romdata = 32'h00000024; // and rd,rs,rt
    5'h10: romdata = 32'h08000010; // jmp target
   5'h3: romdata=32'h00e22025; //or a0,a3,v0
```

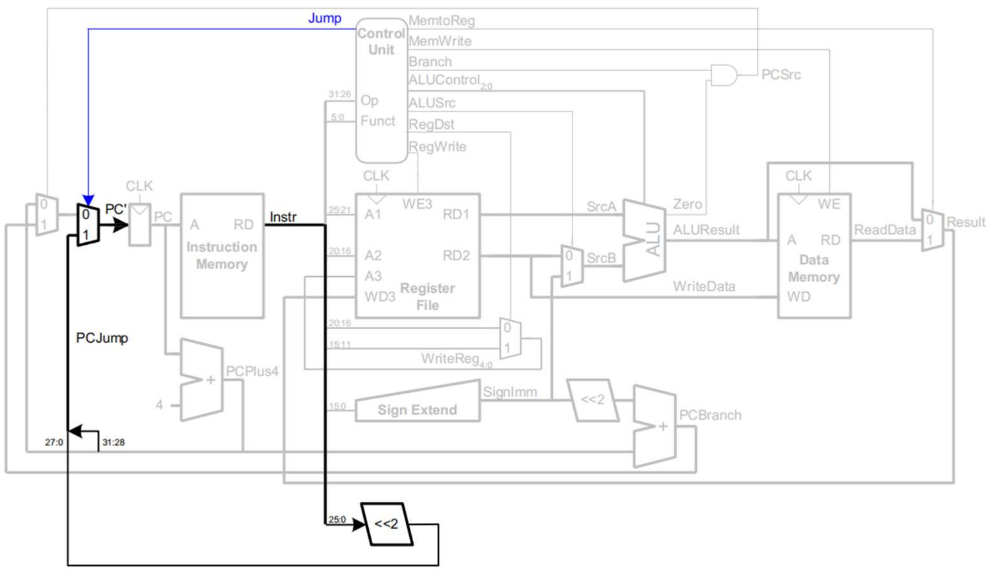
```
  default:romdata=32'h0;
    endcase
    assign data=romdata;
endmodule
```

RAM:

```
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
```

VI. MIPS single cycle CPU overall design

1.CPU circuit design



Connecting the Controller and Datapath

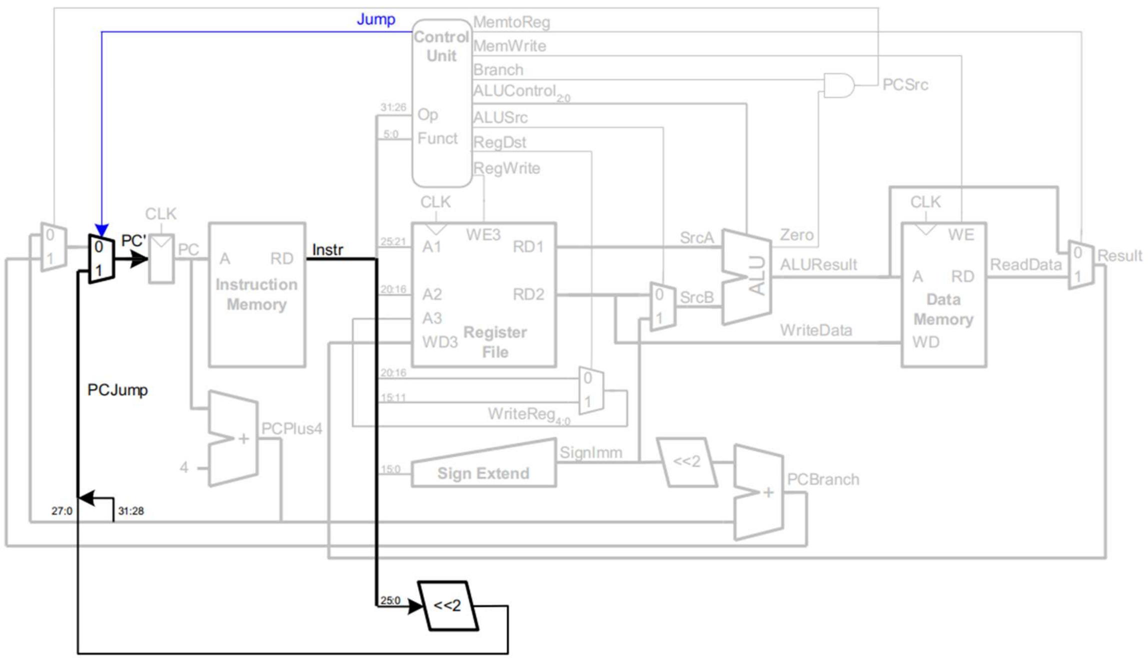
2. Code implementation

```
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),.re
gdst(regdst),
    .branch(branch),.regwrite(regwrite),.alusrc(alusrc),.jump(jump),.memtoreg(memtoreg),.al
ucontrol(alucontrol)
);

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

endmodule
```

3. Overall function implementation

As a module of the computer, the CPU cannot run alone. It must be combined with the instruction memory and data memory to realize a complete instruction execution process of fetching, decoding, executing, and storing values.

(1) Circuit design



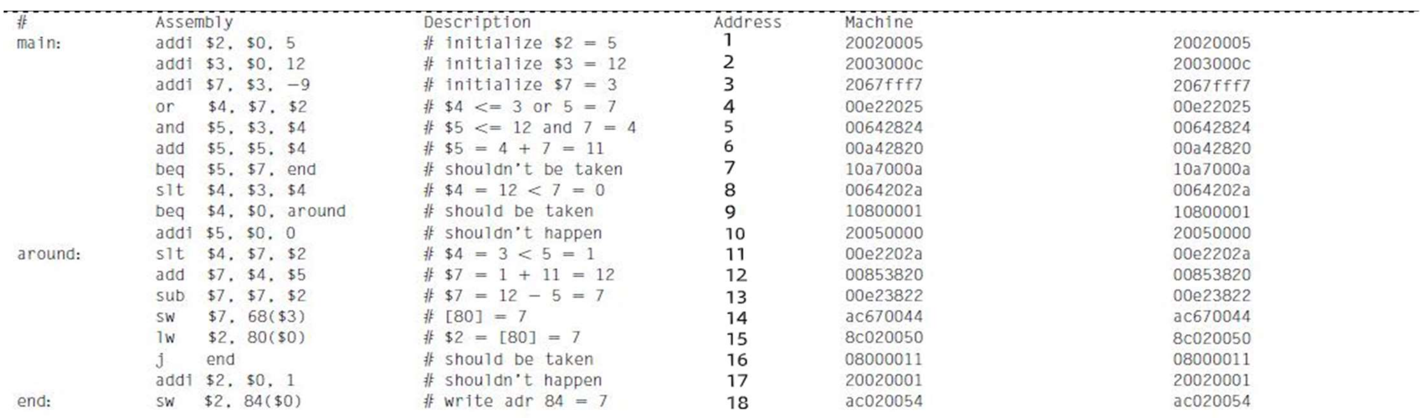
(2) Code implementation

```
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_w data(mem_w data),
.imm_e xtend(imm_e xtend),
.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_r esult[9:0]),
.indata(mem_w data),
.outdata(mem_rdata)
);
endmodule
```

VII. Functional testing

Write a simulation stimulus file to test the correctness of the implemented instructions. Since the instructions are already in the instruction register, you only need to give the top module a clock signal and then check the correctness of the code execution cycle by cycle.

Instructions stored in the instruction register and theoretical running results:



#	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

1. Simulation stimulus code

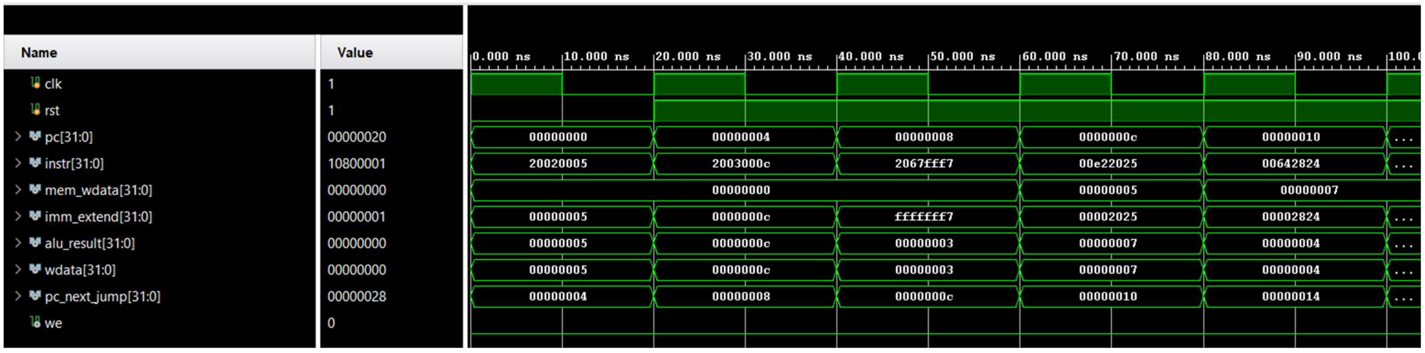
```
module testbanch();
reg clk;
reg rst;
wire [31:0] pc,instr,mem_wdata,imm_extend,alu_result,wdata,pc_next_jump;
wire we;
top uut(clk,rst,pc,instr,mem_wdata,imm_extend,alu_result,wdata,pc_next_jump,we);

initial begin
        rst <= 0;
        #20;
        rst <=1;
end

always begin
        clk <= 1;
        #10;
        clk <= 0;
        #10;
end

always @(negedge clk) begin
        if(we) begin
            /* code */
            if(alu_result === 84 & mem_wdata === 7) begin
                /* code */
                $display("Simulation succeeded");
                $stop;
            end else if(alu_result !== 80) begin
                /* code */
                $display("Simulation Failed");
                $stop;
            end
        end
end
endmodule
```

2. Test results

The simulation waveform input is clk and rst signals. For easy observation, the output of this experiment defines the following signals: pc、instr、mem_wdata、inn_extend、alu_result、wdata、pc_next_jump and we. The specific analysis is as follows:

① **Addi instruction:** Add the value of register rs to the immediate value imm which is sign-extended to 32 bits, and write the result to register rt. If overflow occurs, an IntegerOverflow exception is triggered.

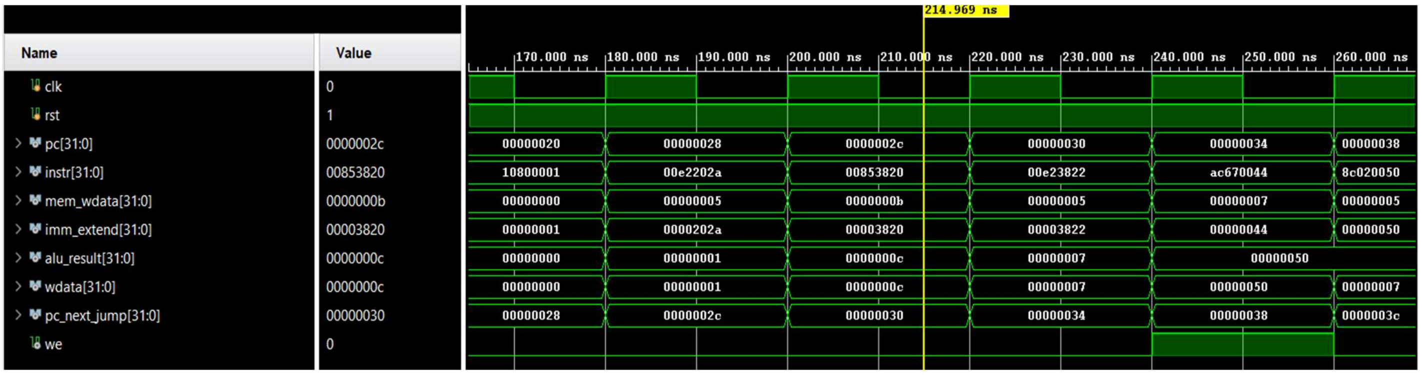
001000	rs	rt	imm
6	5	5	16



```
Name                Value      0.000 ns   10.000 ns  20.000 ns  30.000 ns  40.000 ns  50.000 ns  60.000 ns  70.000 ns  80.000 ns  90.000 ns  100.0
clk                 1
rst                 1
pc[31:0]            00000020   00000000              00000004              00000008              0000000c              00000010              ...
instr[31:0]         10800001   20020005              2003000c              2067fff7              00e22025              00642824              ...
mem_wdata[31:0]     00000000                         00000000                                    00000005              00000007
imm_extend[31:0]    00000001   00000005              0000000c              fffffff7              00002025              00002824              ...
alu_result[31:0]    00000000   00000005              0000000c              00000003              00000007              00000004              ...
wdata[31:0]         00000000   00000005              0000000c              00000003              00000007              00000004              ...
pc_next_jump[31:0]  00000028   00000004              00000008              0000000c              00000010              00000014              ...
we                  0
```

② **or instruction:** The value in register rs is bitwise logically ORed with the value in register rt and the result is written to register rd.

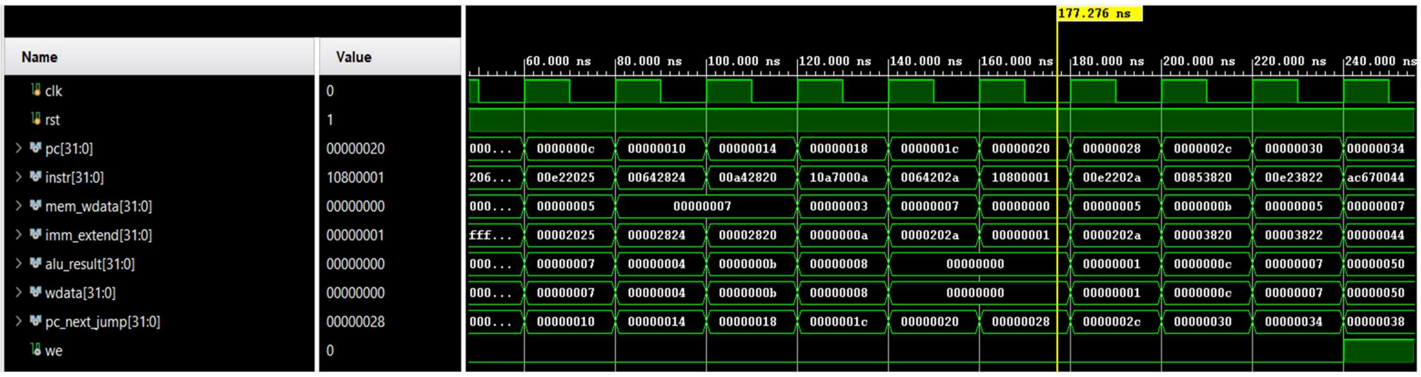
000000	rs	rt	rd	00000	100101
6	5	5	5	5	6



Name	Value
clk	0
rst	1
pc[31:0]	0000002c
instr[31:0]	00853820
mem_wdata[31:0]	0000000b
imm_extend[31:0]	00003820
alu_result[31:0]	0000000c
wdata[31:0]	0000000c
pc_next_jump[31:0]	00000030
we	0

③**and instruction:** The value in register rs is bitwise logically ANDed with the value in register rt and the result is written to register rd.

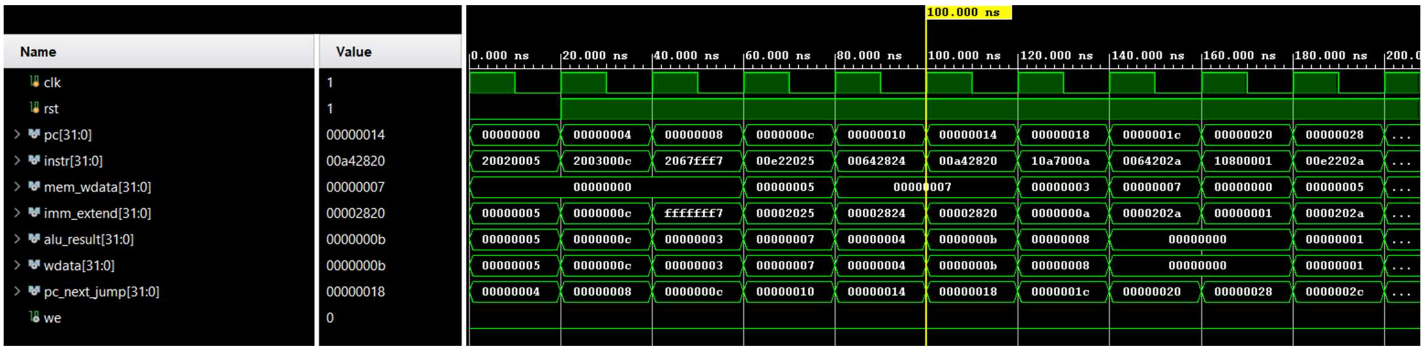
000000	rs	rt	rd	00000	100100
6	5	5	5	5	6



Name	Value
clk	0
rst	1
pc[31:0]	00000020
instr[31:0]	10800001
mem_wdata[31:0]	00000000
imm_extend[31:0]	00000001
alu_result[31:0]	00000000
wdata[31:0]	00000000
pc_next_jump[31:0]	00000028
we	0

④**add instruction:** Add the value of register rs to the value of register rt and write the result to register rd. If overflow occurs, an integer overflow exception (IntegerOverflow) is triggered.

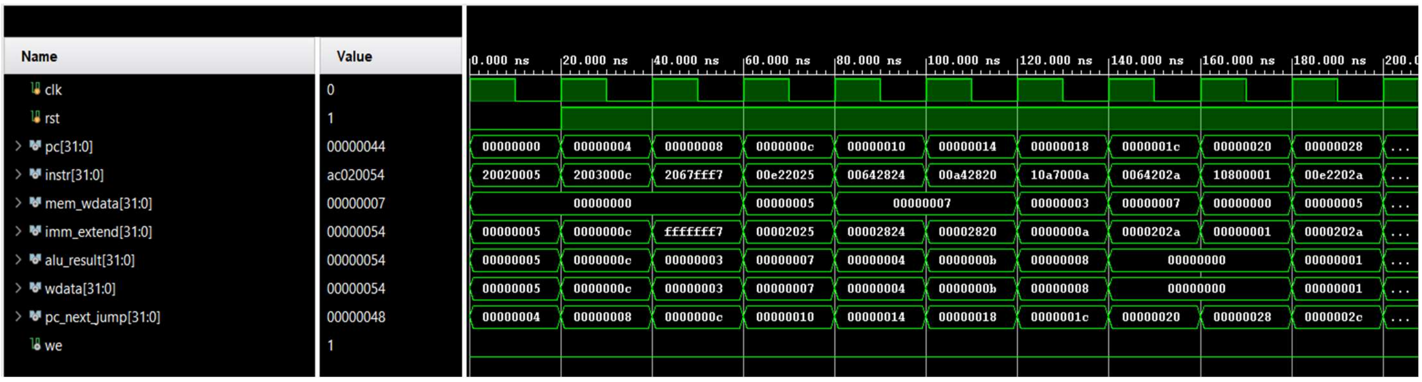
000000	rs	rt	rd	00000	100000
6	5	5	5	5	6



Name	Value
clk	1
rst	1
pc[31:0]	00000014
instr[31:0]	00a42820
mem_wdata[31:0]	00000007
imm_extend[31:0]	00002820
alu_result[31:0]	0000000b
wdata[31:0]	0000000b
pc_next_jump[31:0]	00000018
we	0

⑤**beq instruction:** If the value of register rs is equal to the value of register rt, then branch, otherwise execute sequentially. The branch target is calculated by the value of the immediate value offset shifted left by 2 bits and signed extended plus the PC of the delay slot instruction corresponding to the branch instruction.

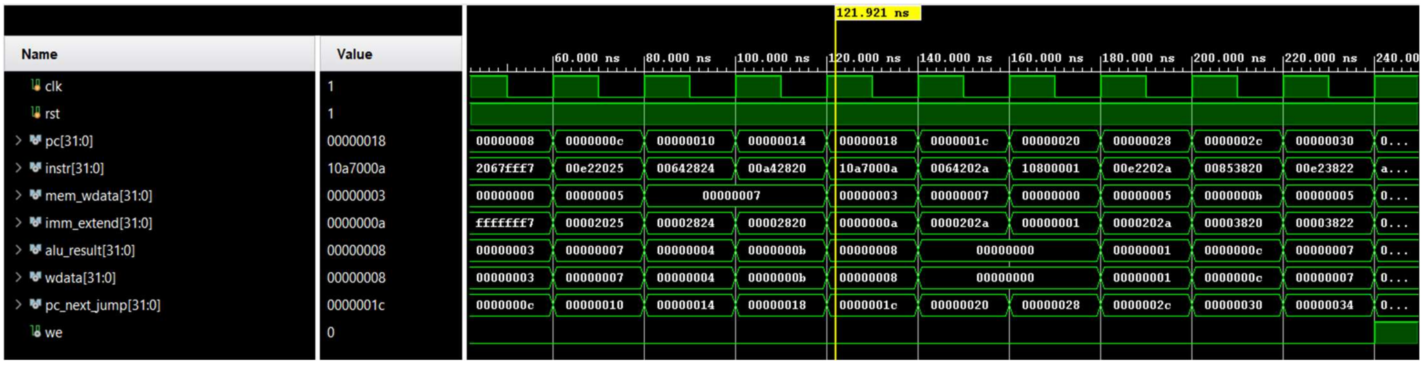
000100	rs	rt	offset
6	5	5	16



Name	Value
clk	0
rst	1
pc[31:0]	00000044
instr[31:0]	ac020054
mem_wdata[31:0]	00000007
imm_extend[31:0]	00000054
alu_result[31:0]	00000054
wdata[31:0]	00000054
pc_next_jump[31:0]	00000048
we	1

⑥**slt instruction:** Compare the value in register rs with the value in register rt as signed numbers. If the value in register rs is smaller, register rd is set to 1; otherwise, register rd is set to 0.

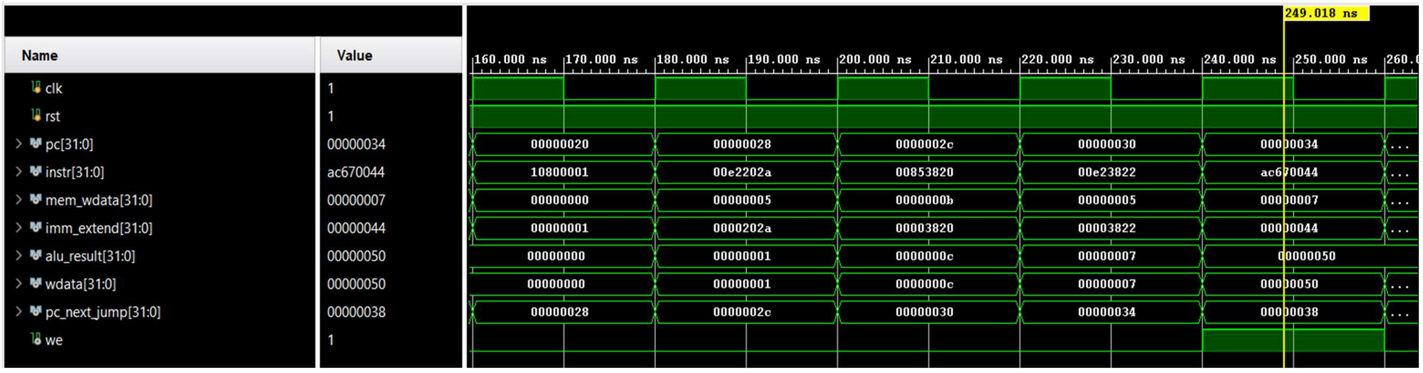
000000	rs	rt	rd	00000	101010
6	5	5	5	5	6



Name	Value
clk	1
rst	1
pc[31:0]	00000018
instr[31:0]	10a7000a
mem_wdata[31:0]	00000003
imm_extend[31:0]	0000000a
alu_result[31:0]	00000008
wdata[31:0]	00000008
pc_next_jump[31:0]	0000001c
we	0

⑦**sub instruction:** Subtract the value of register rs from the value of register rt, and write the result to register rd. If overflow occurs, an integer overflow exception (IntegerOverflow) is triggered.

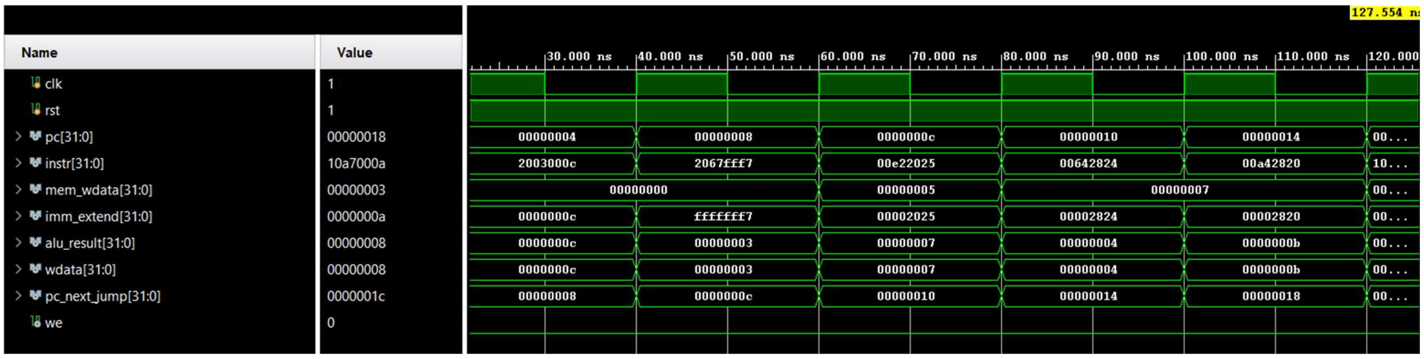
000000	rs	rt	rd	00000	100010
6	5	5	5	5	6



Name	Value						
clk	1						
rst	1						
pc[31:0]	00000034	00000020	00000028	0000002c	00000030	00000034	...
instr[31:0]	ac670044	10800001	00e2202a	00853820	00e23822	ac670044	...
mem_wdata[31:0]	00000007	00000000	00000005	0000000b	00000005	00000007	...
imm_extend[31:0]	00000044	00000001	0000202a	00003820	00003822	00000044	...
alu_result[31:0]	00000050	00000000	00000001	0000000c	00000007	00000050	
wdata[31:0]	00000050	00000000	00000001	0000000c	00000007	00000050	...
pc_next_jump[31:0]	00000038	00000028	0000002c	00000030	00000034	00000038	...
we	1						

⑧**sw instruction:** Add the value of the base register to the sign-extended immediate value offset to get the virtual address of the memory access. If the address is not an integer multiple of 4, an address error exception is triggered. Otherwise, the rt register is stored in the memory according to this virtual address.

101011	base	rt	offset
6	5	5	16

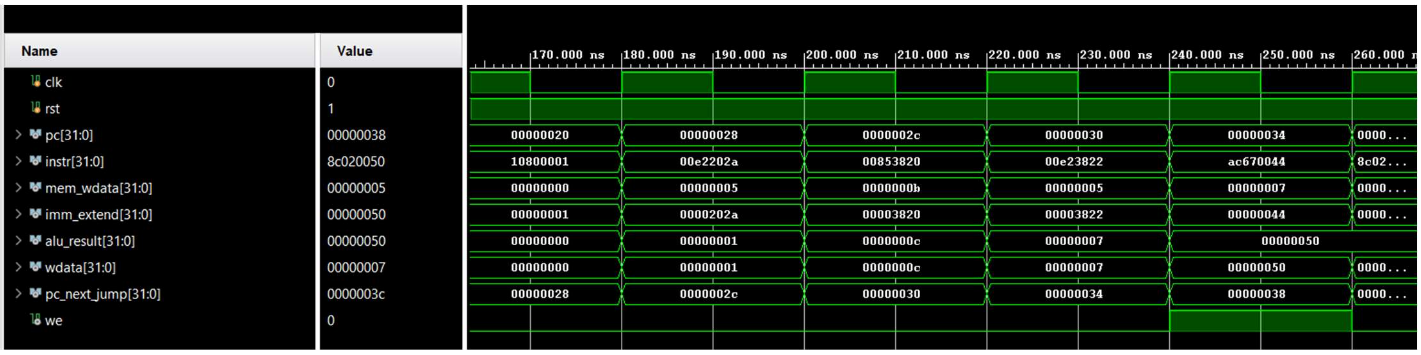


Name	Value						
clk	1						
rst	1						
pc[31:0]	00000018	00000004	00000008	0000000c	00000010	00000014	00...
instr[31:0]	10a7000a	2003000c	2067fff7	00e22025	00642824	00a42820	10...
mem_wdata[31:0]	00000003	00000000		00000005	00000007		00...
imm_extend[31:0]	0000000a	0000000c	fffffff7	00002025	00002824	00002820	00...
alu_result[31:0]	00000008	0000000c	00000003	00000007	00000004	0000000b	00...
wdata[31:0]	00000008	0000000c	00000003	00000007	00000004	0000000b	00...
pc_next_jump[31:0]	0000001c	00000008	0000000c	00000010	00000014	00000018	00...
we	0						

⑨**lw instruction:** Add the value of the base register to the sign-extended immediate value offset to get the virtual address of the memory access. If the address is not an integer multiple of 4, an address error exception is triggered. Otherwise, 4 consecutive bytes of value are read from the memory according to the virtual address, sign-extended, and written to the rt register.

100011	base	rt	offset

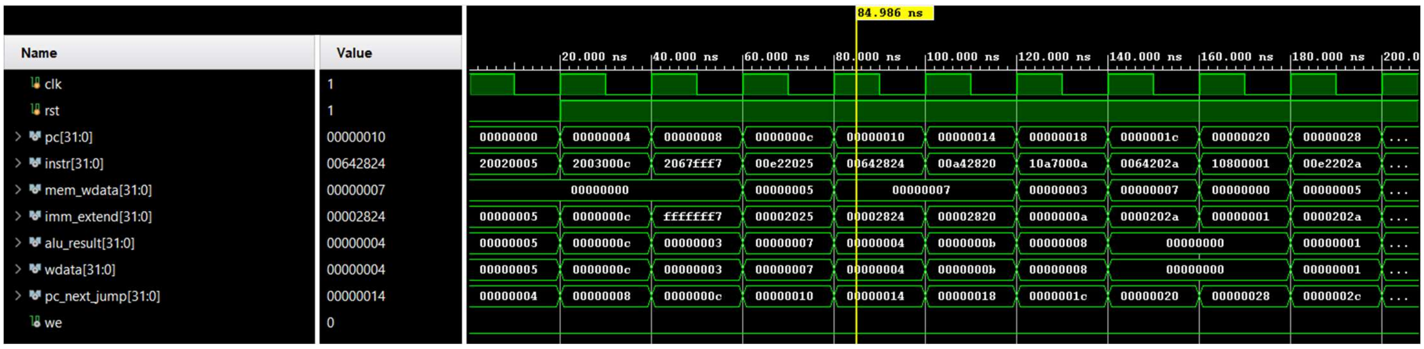
6	5	5	16



Name	Value
clk	0
rst	1
pc[31:0]	00000038
instr[31:0]	8c020050
mem_wdata[31:0]	00000005
imm_extend[31:0]	00000050
alu_result[31:0]	00000050
wdata[31:0]	00000007
pc_next_jump[31:0]	0000003c
we	0

⑩**jmp instruction:** unconditional jump. The jump target is obtained by concatenating the highest 4 bits of the PC of the delay slot instruction corresponding to the branch instruction and the value of the immediate value instr_index shifted left by 2 bits.

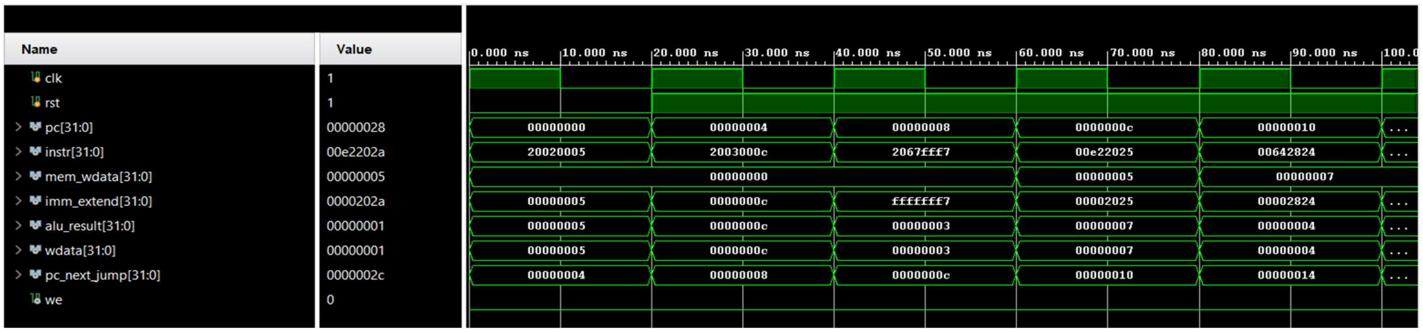
000010	instr_index
6	26



Name	Value
clk	1
rst	1
pc[31:0]	00000010
instr[31:0]	00642824
mem_wdata[31:0]	00000007
imm_extend[31:0]	00002824
alu_result[31:0]	00000004
wdata[31:0]	00000004
pc_next_jump[31:0]	00000014
we	0

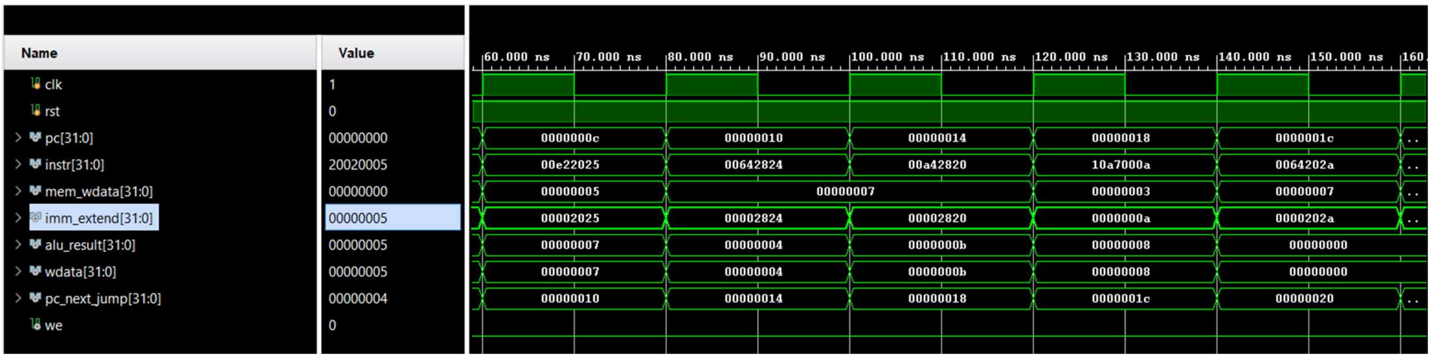
⑩+①**: xor instruction:** The value in register rs is bitwise logically exclusive ORed with the value in register rt, and the result is written to register rd.

000000	rs	rt	rd	00000	100110
6	5	5	5	5	6



Name	Value
clk	1
rst	1
pc[31:0]	00000028
instr[31:0]	00e2202a
mem_wdata[31:0]	00000005
imm_extend[31:0]	0000202a
alu_result[31:0]	00000001
wdata[31:0]	00000001
pc_next_jump[31:0]	0000002c
we	0

Overall result:



Name	Value						
clk	1						
rst	0						
pc[31:0]	00000000	0000000c	00000010	00000014	00000018	0000001c	..
instr[31:0]	20020005	00e22025	00642824	00a42820	10a7000a	0064202a	..
mem_wdata[31:0]	00000000	00000005	00000007		00000003	00000007	..
imm_extend[31:0]	00000005	00002025	00002824	00002820	0000000a	0000202a	..
alu_result[31:0]	00000005	00000007	00000004	0000000b	00000008	00000000	
wdata[31:0]	00000005	00000007	00000004	0000000b	00000008	00000000	
pc_next_jump[31:0]	00000004	00000010	00000014	00000018	0000001c	00000020	..
we	0						

VIII. Problems encountered and solutions

Problem 1: Syntax Errors in Verilog Code

While compiling the Verilog code for the *`rom`* module, we encountered syntax errors due to minor mistakes, like missing semicolons or incorrect case statement formatting.

Solution 1: We carefully reviewed the Verilog code line by line to find syntax errors. We used the Verilog compiler's error messages to locate and fix each mistake, ensuring all statements were correctly formatted. After resolving these issues, the code compiled successfully.

Problem 2: Incorrect Instruction Output

Some instructions, such as *`addi`* and *`sw`*, were not producing the correct output values when tested. This led to unexpected results in the program.

Solution 2: We checked the instruction encoding for each problematic instruction to ensure it matched the MIPS specification. After identifying minor errors in the instruction values, we corrected them in the *`rom`* module. After these adjustments, the instructions executed as expected, producing the correct output.

IX. Conclusion

In this project, we successfully designed and implemented a MIPS Single Cycle CPU, demonstrating a foundational understanding of CPU architecture and instruction execution. By constructing a single-cycle datapath and control unit, we were able to execute a set of MIPS instructions, including arithmetic, logical, memory, and branching operations, within a single clock cycle for each instruction. This project provided hands-on experience and gave us understandings into the complexities of CPU design, especially in terms of data flow, control signals, and timing. Through testing and debugging, we overcame challenges such as encoding instructions correctly and managing control signals to ensure accurate instruction execution.

The final design effectively processes a basic set of MIPS instructions, showcasing the functionality of a single-cycle CPU. This project has strengthened our understanding of digital design principles and foundation for exploring more complex CPU designs, such as pipelined and multi-cycle processors, in future work. Overall, building this MIPS Single Cycle CPU has been a valuable learning experience in computer organization.