School of Artificial Intelligence and Computer Science Nantong University

­­­­­­­­­­­­­­­­­­­­ **Computer Composition Principles**

**Course Design**

**Report**

**Title Simple single-cycle CPU implementation**

I. Purpose

1．Consolidate the knowledge of the course "Principles of Computer Organization". Through the comprehensive application of knowledge, students will have a deeper understanding of the working principles and mutual connections of each module of the computer system, and deepen their understanding of the concept of "time-space" in computer work, so as to clearly establish the concept of the whole computer.

2．Learn the basic steps and methods of designing and debugging computers, cultivate students' ability to carry out scientific research independently, enable students to obtain the necessary engineering practice experience and ability to solve complex engineering problems in the computer field, and understand the responsibilities of team work. Cultivate students' ability to serve as team members or leaders, and cultivate students' ability to independently complete the work assigned by the team.

II. Team

This course design team consists of 3-6 people. The team members and division of labor are shown in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Student ID | Name | Role | Contribution |
| 1 |  |  | Leader |  |
| 2 |  |  | Members |  |
| 3 |  |  | Members |  |
| 4 |  |  | Members |  |
| 5 |  |  | Members |  |

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.

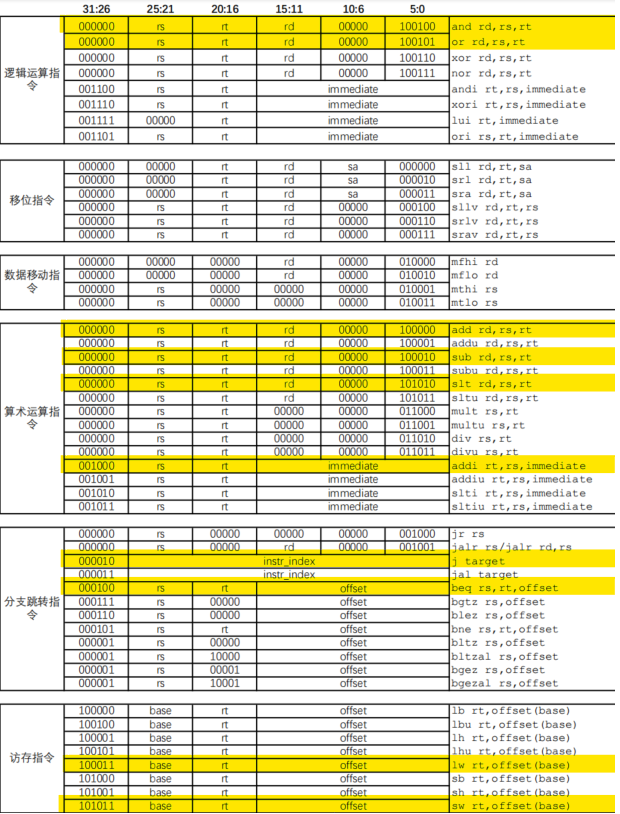
7. Instruction register: This is essentially a rom read-only memory. It does not require a clock signal. It fetches the instruction at the corresponding position based on the input address and then sends it to other components.

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.

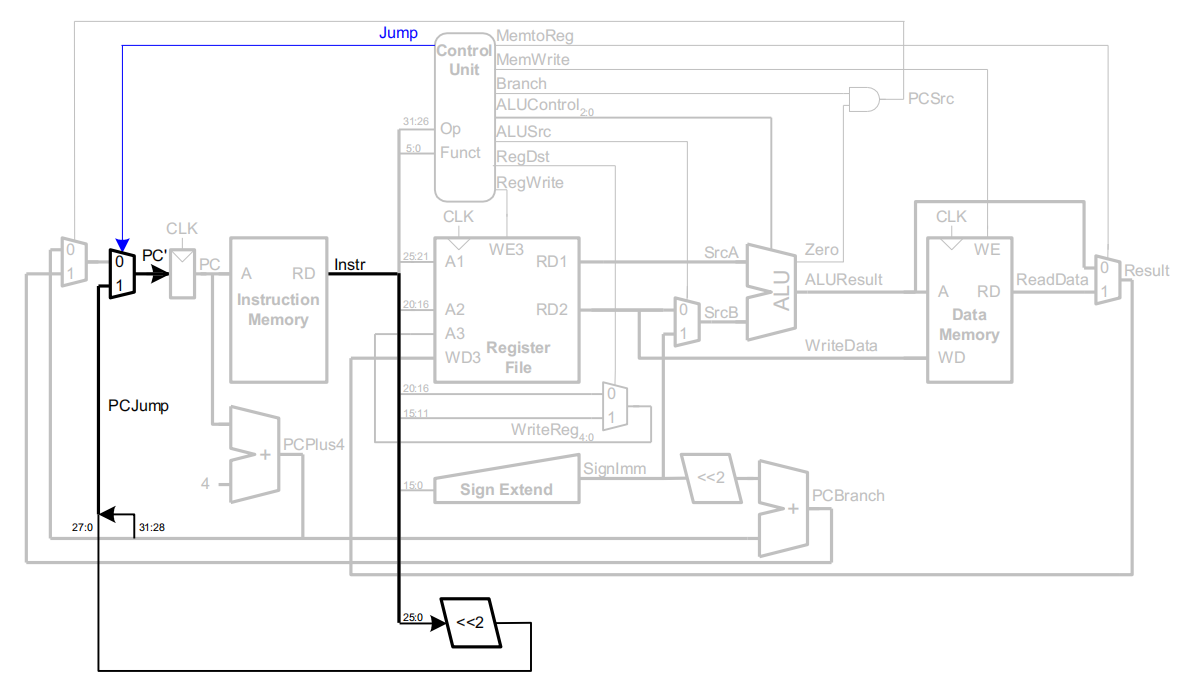
Mips single cycle instruction table:MIPS-ISA



The mips instructions are shown in the figure above. The yellow 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));

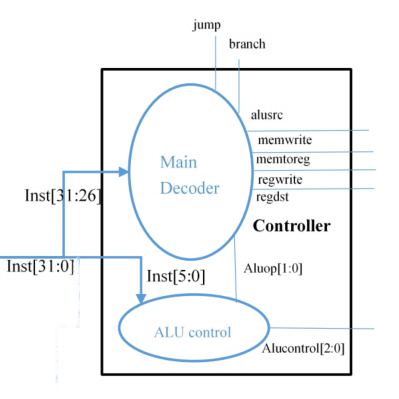
mux2 #(32) mux2\_alu(.s(alusrc),.a(rdata2),.b(imm\_extend),.y(alu\_srcB));

alu alu(.a(rdata1),.b(alu\_srcB),.op(alucontrol),.s(alu\_result),.zero(zero));

endmodule

1. Controller

① Circuit design



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

② Code 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;

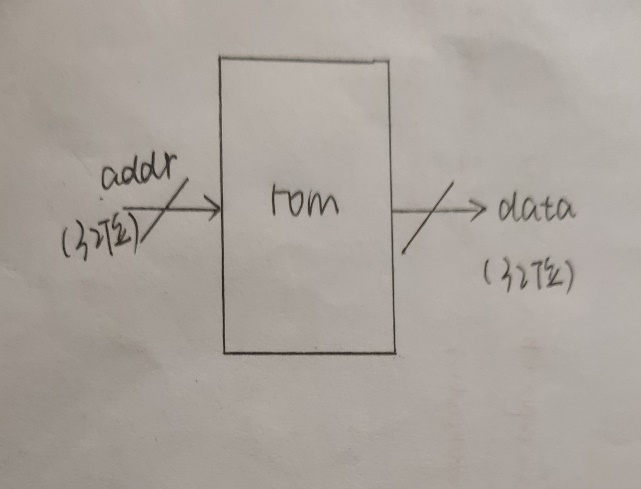
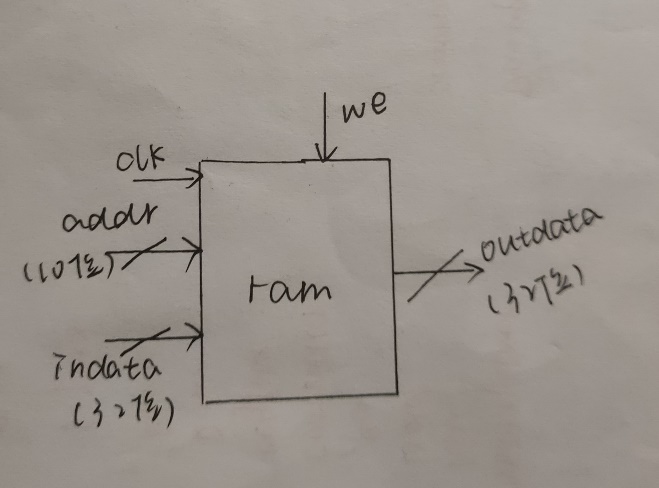
    main\_dec md(op,regdst,branch,regwrite,alusrc,memwrite,jump,memtoreg,aluop);

    alu\_dec ad(funct,aluop,alucontrol);

endmodule

(3) instruction ram and memory ram

① Circuit design

ROM Ram

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;

    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;

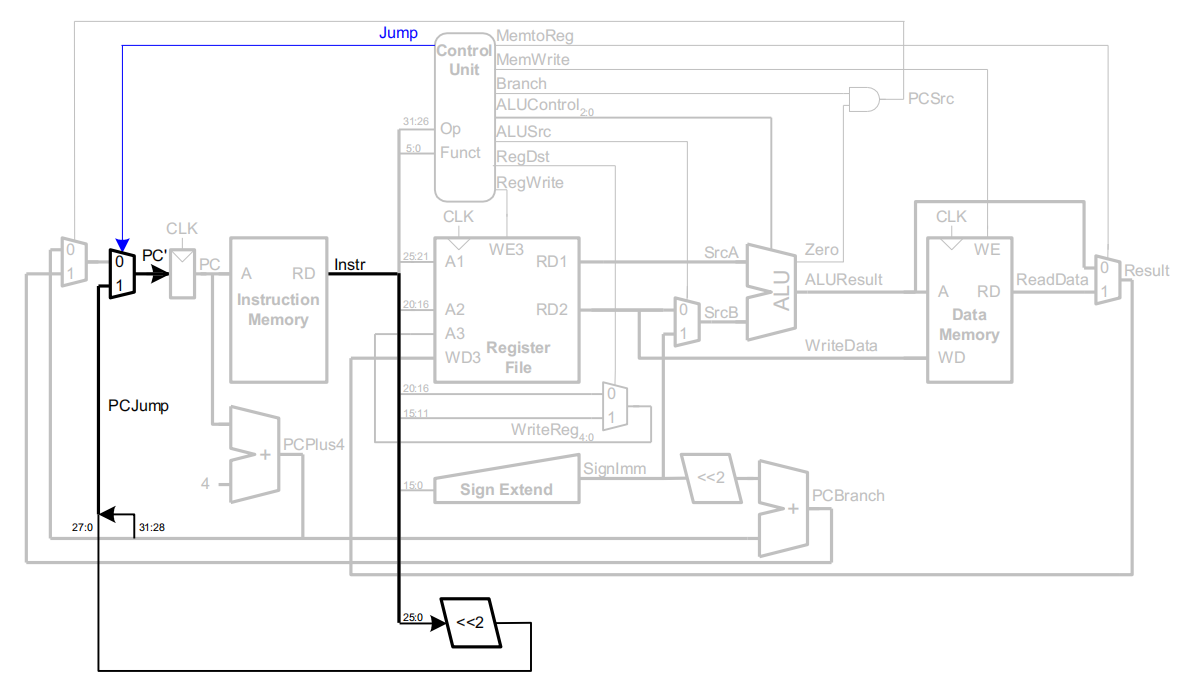
end

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),.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(branch),.regwrite(regwrite),

.alusrc(alusrc),.alucontrol(alucontrol),.memwrite(we),.jump(jump),

.memtoreg(memtoreg)

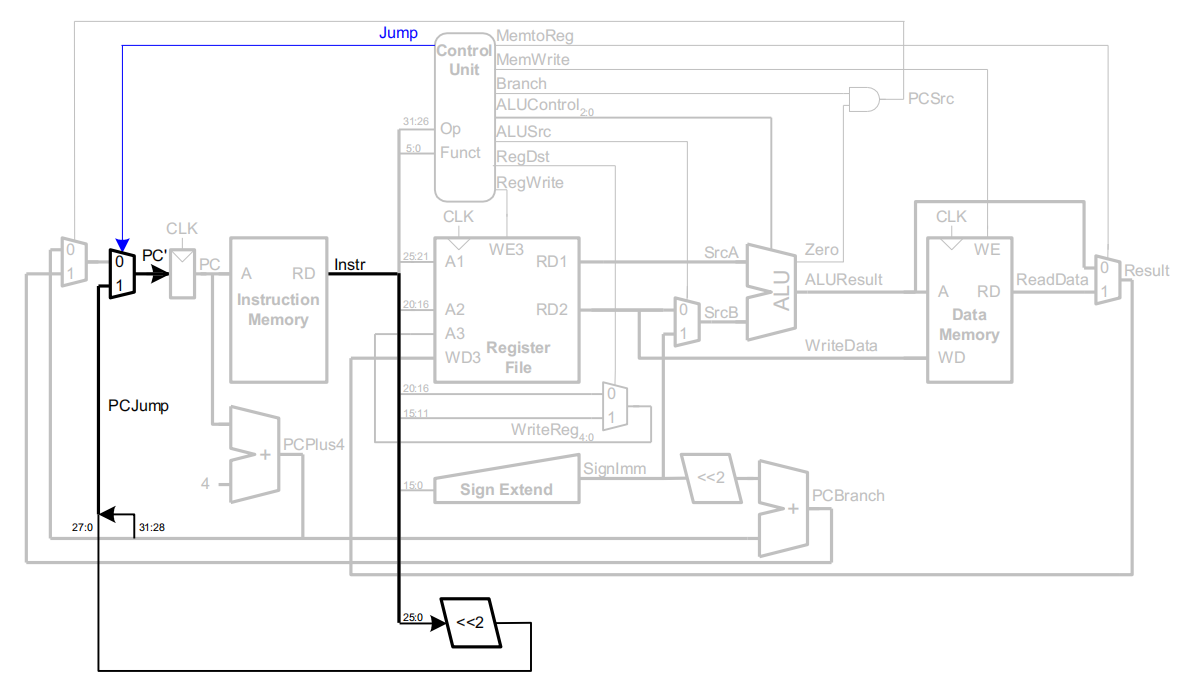
);

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\_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)

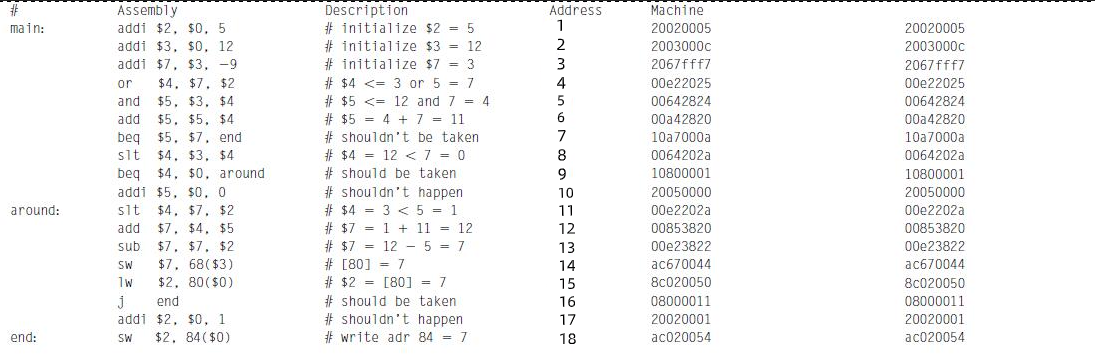
);

endmodule

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:



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

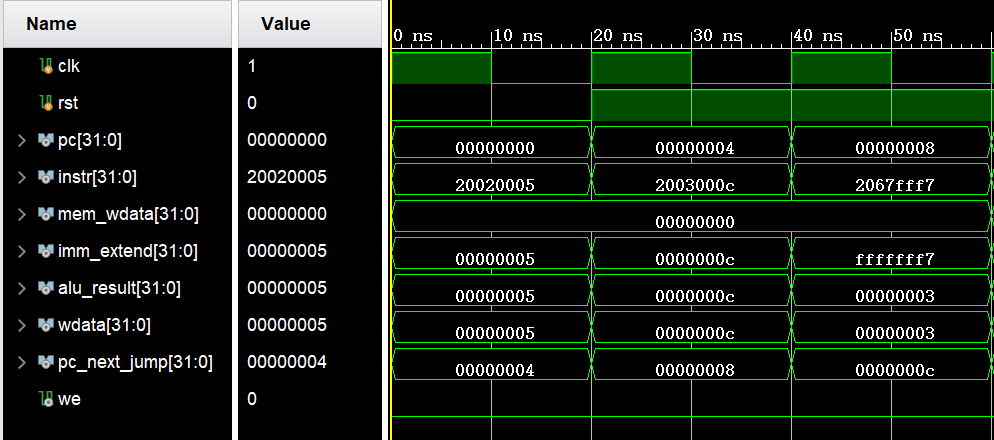
2. Test results

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 |

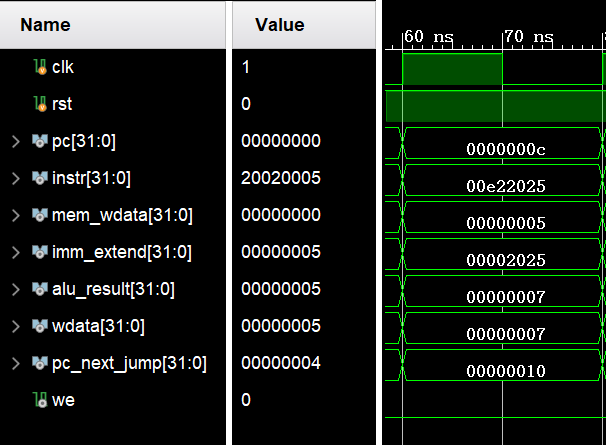
6 5 5 16



1. 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 |

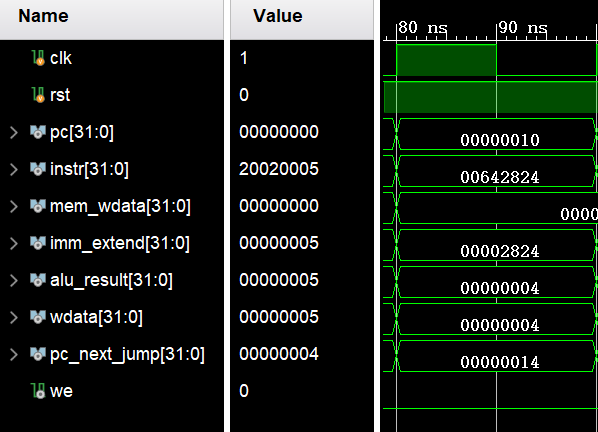
6 5 5 5 5 6



③and 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 |

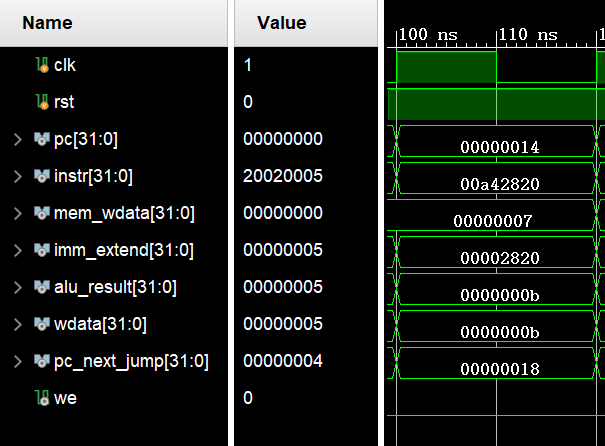
6 5 5 5 5 6



④add 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 |

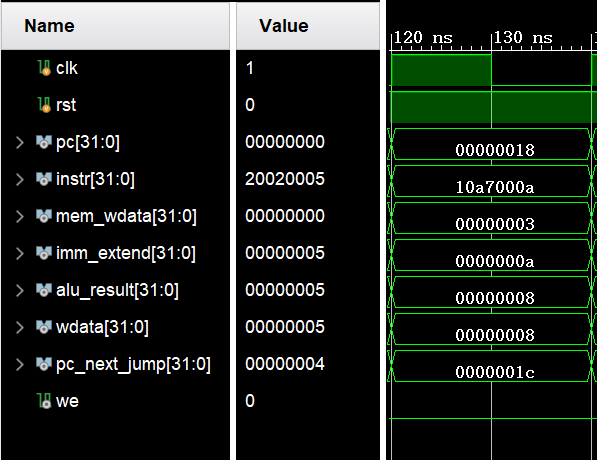
6 5 5 5 5 6



⑤beq 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 |

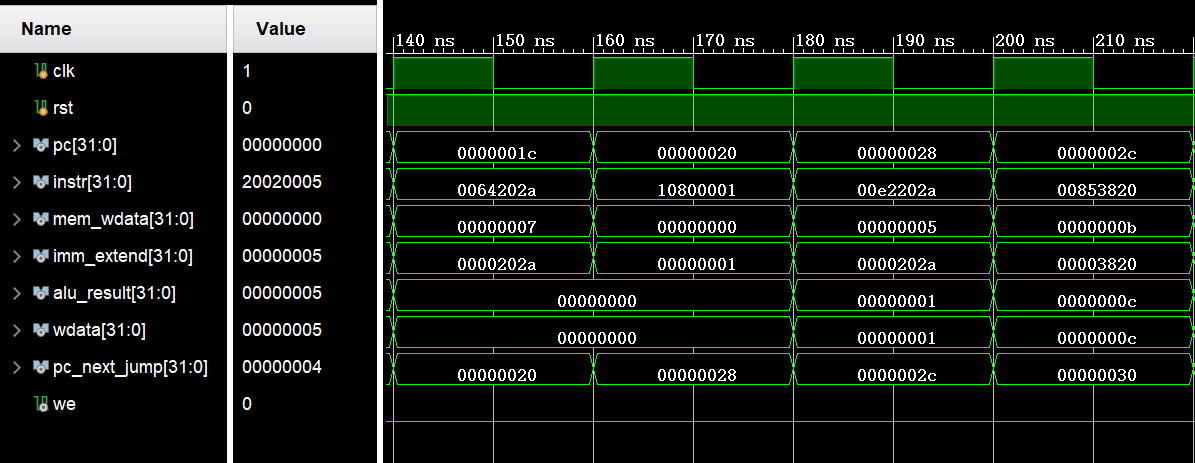
6 5 5 16



⑥slt 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 |

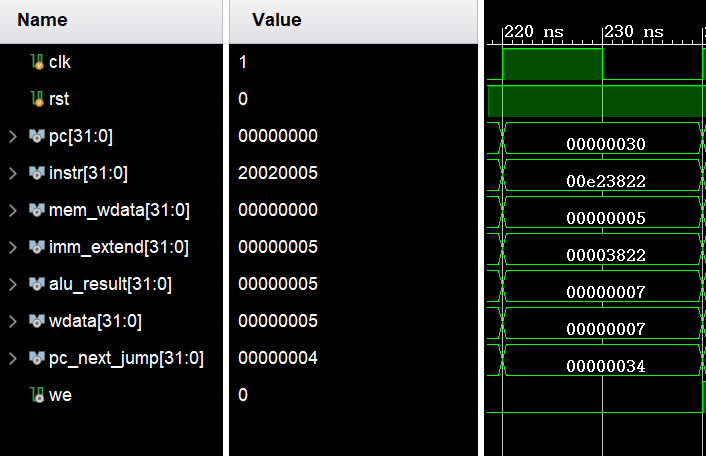
6 5 5 5 5 6



⑦sub 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 |

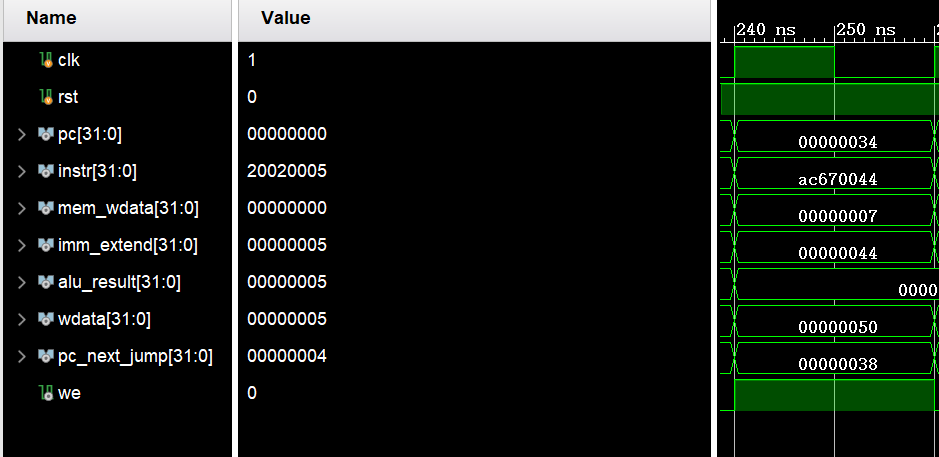
6 5 5 5 5 6



⑧sw 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 |

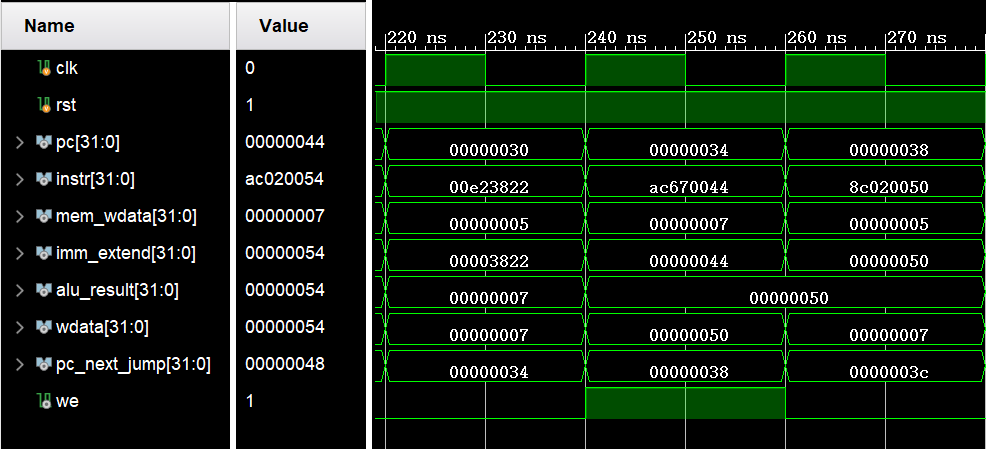
6 5 5 16



⑨lw 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 |

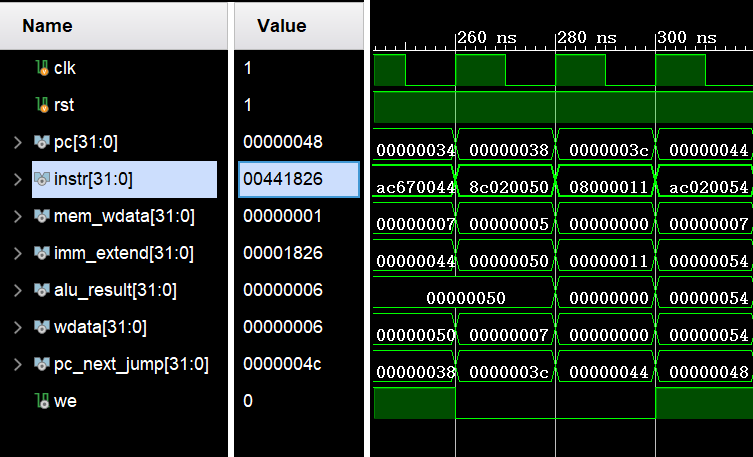
6 5 5 16



⑩jmp 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.

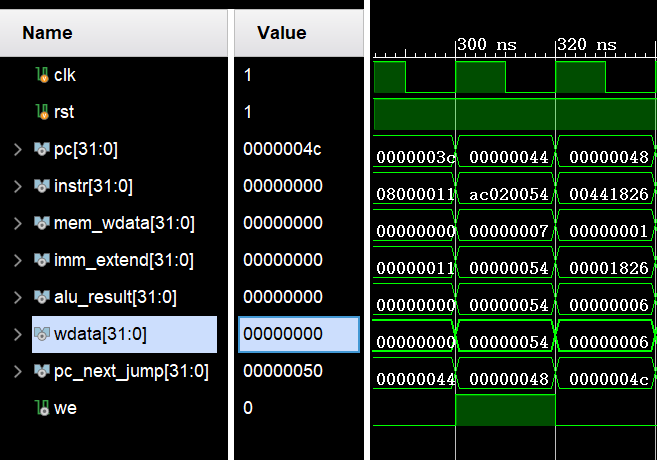
|  |  |
| --- | --- |
| 000010 | instr\_index |

6 26

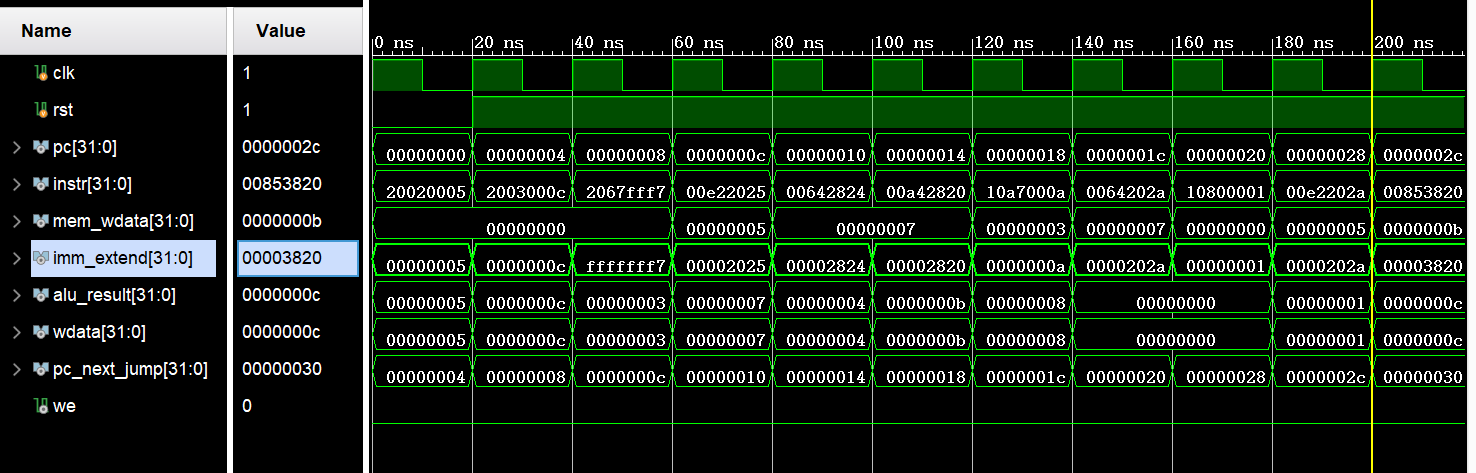


⑩+①: xor instruction: The value in register rs is bitwise logically exclusive ORed with the value in register rt, and the result is written to register rd.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 000000 | rs | rt | rd | 00000 | 100110 |

6 5 5 5 5 6

Overall result:



VIII. Problems encountered and solutions

Problem 1:

Solution 1:

Problem 2:

Solution 2:

IX. Conclusion