实验项目名称：Multiple-cycle CPU Design

一、 实验目的和要求

1. Design the CPU Controller, Datapath, bring together the basic units into Multiple-cycle CPU
2. Verify the MC CPU with program and observe the execution of program

二、 实验内容和原理

1. Multiple-cycle CPU Controller

The following table shows the output of CPU Controller:

Output Signal	Meaning When 1	Meaning When 0
PCSrc[1:0]	00: PC + 4;01: Branch Instr.;10: jump Instr	
WritePC	Write PC	Not Write PC
IorD	Instruction Addr	Data Addr
WriteMem	Write Mem	Not Write Mem
Write DR	Write Data. Reg	Not Write Data. Reg
Write IR	Write Instr. Reg	Not Write Instr. Reg
MemToReg	From Mem. To Reg	From ALUOut To Reg
RegDest	rd	rt
ALUC	ALU Controller Op	
ALUSrcA	Register rs	PC
ALUSrcB	Selection:00:Reg rt; 01:4; 10:Imm.; 11: branch Address	
WriteA	Write A Reg.	Not Write A Reg.
WriteB	Write B Reg.	Not Write B Reg.
WriteC	Write C Reg.	Not Write C Reg.
WriteReg	Write Reg.	Not Write Reg.

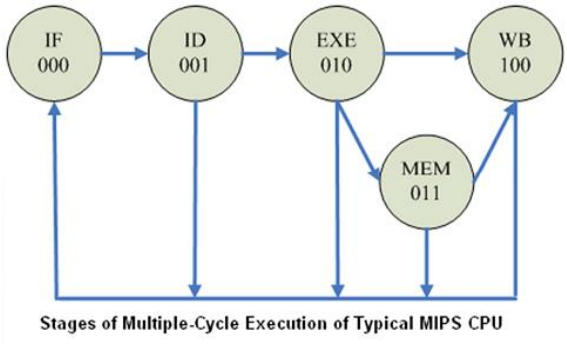
2. The principle of CPU Controller

The typical implement is to use five stages to complete instruction execution:

1) IF：Instruction Fetch
2) ID：Instruction Decode
3) EX：Excution

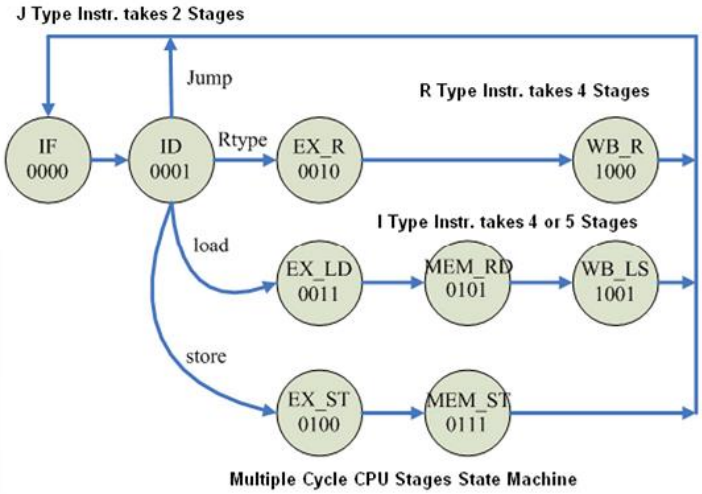
4) MEM：Memory Access（lw/sw）

5) WB：Write Back



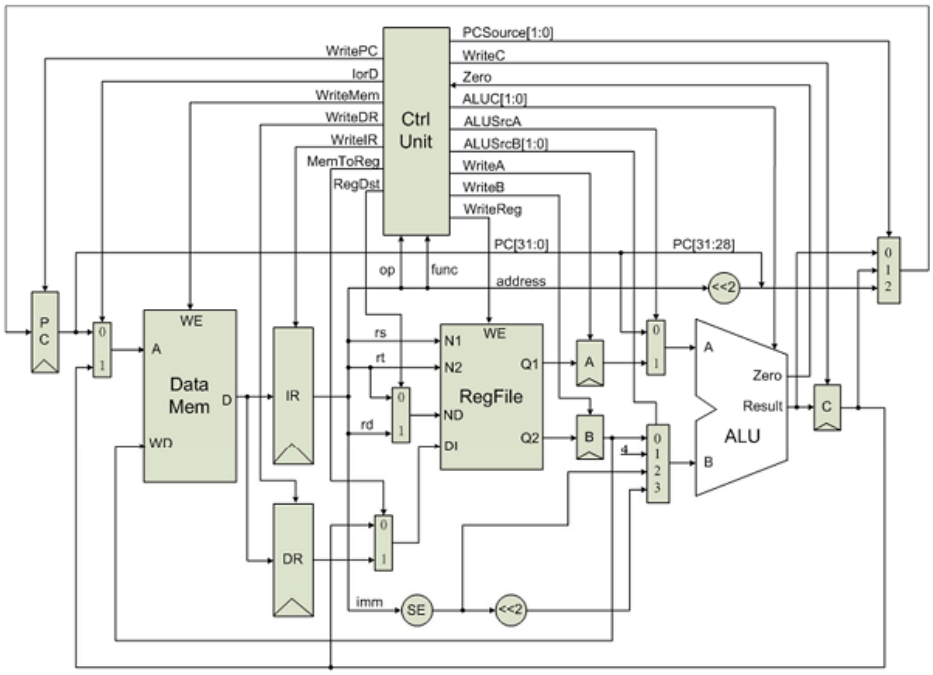
Stages of Multiple-Cycle Execution of Typical MIPS CPU

However, instructions of different types can have different number of stages, thus making the following infinite state machine:



Multiple Cycle CPU Stages State Machine

3. The Datapath of Multiple-cycle CPU



Note that there are some additional registers like IR and DR, and PC+4 is implemented by ALU.

三、 实验过程和数据记录

The code of the main part is as follows:

```
// Multiple-cycle CPU control module, implemented by a finite state machine
module ctrl(clk, rst, ir_data, zero, write_pc, iord, write_mem, write_dr,
write_ir, memtoreg, regdst,
pcsource, write_c, alu_ctrl, alu_srcA, alu_srcB, write_a, write_b,
write_reg, state, insn_type, insn_code, insn_stage);
    //states
    parameter IF = 4'b0000, ID = 4'b0001, EX_R = 4'b0010,
        EX_LD = 4'b0011, EX_ST = 4'b0100, MEM_RD = 4'b0101,
        MEM_ST = 4'b0111, WB_R = 4'b1000, WB_LS = 4'b1001, EX_BEQ = 4'b1010,
EX_J = 4'b1011, OTHER = 4'b1111;
    //ALU control signals
    parameter ADD = 2'b00, SUB = 2'b01, AND = 2'b11, NOR = 2'b10;
    //
    parameter LED_IF = 4'b0001, LED_ID = 4'b0010, LED_EX = 4'b0011, LED_MEM
=4'b0100, LED_WB = 4'b0101;
    parameter LED_R = 4'b0001, LED_J = 4'b0010, LED_I = 4'b0011, LED_N
= 4'b0000;
    parameter LED_LD = 4'b0001, LED_ST = 4'b0010, LED_AD = 4'b0011,
        LED_SU = 4'b0100, LED_AN = 4'b0101, LED_NO = 4'b0110,
        LED JP = 4'b0111, LED NI = 4'b0000;
    //stages
    parameter STAGE_IF = 3'b000, STAGE_ID = 3'b001, STAGE_EXE = 010,
STAGE_WB = 3'b100, STAGE_MEM = 3'b011;
    // input signals
    input       clk;      //clock
    input       rst;      //reset
    input [31:0] ir_data;    //instruction
    input       zero;     //zero signal
    // control signals
```

```
    output      write_pc;    //write PC enable
    output       iord;       //IorD signal to use PC or the ALU result as
the address of memory
    output      write_mem;   //write memory enable
    output       write_dr;   //write the data register enable
    output       write_ir;   //write the instruction register enable
    //memtoreg signal decides the data written to register is from the
data register or the ALU result
    output      memtoreg;
    output       regdst; //regdst decides whether the written register
is rt or rd
    output [1:0]  pcsource;  //pcsource controls the PC assignment
//write_c decides whether to write the ALU result into registerC
    output       write_c;
    output [1:0]  alu_ctrl;  //ALU control signal
    output      alu_srcA;    //ALU source A port
    output [1:0]  alu_srcB;  //ALU source B port
//write_a decides whether to write RegFile's output A port into registerA
    output       write_a;
//write_b decides whether to write RegFile's output B port into registerB
    output       write_b;
    output       write_reg;  //write register enable
    output [3:0]  state;     //status of state machine
    output [3:0]  insn_type; //type of instruction
    output [3:0]  insn_code; //operation code type of instruction
    output [3:0]  insn_stage;//instruction stage

    reg [3:0]  state;
    reg [3:0]  insn_type;
    reg [3:0]  insn_code;
    reg [3:0]  insn_stage;
    reg      write_pc;
    reg       iord;
    reg      write_mem;
    reg       write_dr;
    reg       write_ir;
    reg      memtoreg;
    reg       regdst;
    reg [1:0]  pcsource;
    reg       write_c;
    reg [1:0]  alu_ctrl;
    reg      alu_srcA;
    reg [1:0]  alu_srcB;
    reg       write_a;
    reg       write_b;
    reg       write_reg;
    initial begin
        state <= OTHER;
        write_pc <= 1'b0;
        write_mem <= 1'b0;
        write_dr <= 1'b0;
        write_ir <= 1'b0;
        memtoreg <= 1'b0;
        regdst <= 1'b0;
        pcsource <= 2'b00;   //PC+4
        write_c <= 1'b0;
        alu_ctrl <= 2'b00;   //add
        alu_srcA <= 1'b0; //PC
        alu_srcB <= 2'b01;   //4
        write_a <= 1'b0;
        write_b <= 1'b0;
```

```
    write_reg <= 1'b0;
end
always @ (posedge clk or posedge rst)
begin
    if (rst == 1)
    begin
        state <= IF;
        insn_stage <= STAGE_IF;
    end
    else
    case (state)
        IF:               // Instruction Fetch
        begin
            write_reg <= 1'b0;
            write_mem <= 1'b0;
            write_pc <= 1'b1; //next state PC will be PC+4
            write_ir <= 1'b1; //fetch instruction
            pcsource <= 2'b00;
            //jump to the next state and stage
            state <=ID;
            insn_stage <= STAGE_ID;
        end
        ID:               // Instruction Decode
        begin
            write_pc <= 1'b0; //reset
            write_ir <= 1'b0; //reset
            //write RegFile's output A port into registerA
            write_a <= 1'b1;
            //write RegFile's output B port into registerB
            write_b <= 1'b1;
            alu_srcA <= 1'b0; //PC
            alu_srcB <= 2'b11;   //signed-extended immediate<<2
            alu_ctrl <= ADD;
            write_c <= 1'b1;  //write the ALU result into registerC
            case (ir_data[31:26])
                6'b000000:        // R type insn
                begin
                    //jump to the next state and stage
                    state <= EX_R;
                    insn_stage <= STAGE_EXE;
                end
                6'b000010:   // Jump insn
                begin
                    //jump to the next state and stage
                    state <= EX_J;
                    insn_stage <= STAGE_EXE;
                    pcsource <= 2'b10;   //jump address
                end
                6'b000100:   // Beq  insn
                begin
                    //jump to the next state and stage
                    state <= EX_BEQ;
                    insn_stage <= STAGE_EXE;
                    pcsource <= 2'b01;   //registerC
                end
                6'b100011:  //   Load
                begin
                    //jump to the next state and stage
                    state <= EX_LD;
                    insn_stage <= STAGE_EXE;
                end
```

```
                6'b101011:   // Store
                begin
                    //jump to the next state and stage
                    state <= EX_ST;
                    insn_stage <= STAGE_EXE;
                end
                default:
                begin
                state <= EX_R;
                insn_stage <= STAGE_EXE;
                end
            endcase
        end
        EX_R:         // Excution of R-type
        begin
            write_a <= 1'b0;  //reset
            write_b <= 1'b0;  //reset
            alu_srcA <= 1'b1; //registerA
            alu_srcB <= 2'b00;   //registerB
            write_c <= 1'b1;  //write the ALU result into registerC
            case(ir_data[5:0])
            //generate ALU control signal according to opcode
                6'b100000: alu_ctrl <= ADD;
                6'b100010: alu_ctrl <= SUB;
                6'b100100: alu_ctrl <= AND;
                6'b100111: alu_ctrl <= NOR;
                default:   alu_ctrl <= ADD;
            endcase
            //jump to the next state and stage
            state <=WB_R;
            insn_stage <= STAGE_WB;
            memtoreg <= 1'b0;    //registerC
            write_reg <= 1'b1;   //enable
            regdst <= 1'b1;      //rd
        end
        WB_R:             // Write Back of R-type
        begin
            write_c <= 1'b0;
            //jump to the next state and stage
            state <=IF;
            insn_stage <= STAGE_IF ;
            iord <= 1'b0;        //PC
            alu_srcA <= 1'b0;    //PC
            alu_srcB <= 2'b01;   //4
            alu_ctrl <= ADD;
            write_reg <= 1'b0;   //reset
        end
        EX_LD:            // Excution of Load
        begin
        write_a <= 1'b0;     //reset
        write_b <= 1'b0;     //reset
        iord <= 1'b1;        //registerC
        alu_srcA <= 1'b1;    //registerA
        alu_srcB <= 2'b10;   //signed-extended immediate
        alu_ctrl <= ADD;
        write_c <= 1'b1;  //write the ALU result into registerC
        //jump to the next state and stage
        state <= MEM_RD ;
        insn_stage <= STAGE_MEM ;
        end
        MEM_RD:              // Memory Access of Load
```

```
            begin
                write_c <= 1'b0;  //reset
                write_dr <= 1'b1; //enable to write the data register
                //jump to the next state and stage
                state <= WB_LS;
                insn_stage <= STAGE_WB;
                write_reg <= 1'b1;   //enable
                memtoreg <= 1'b1;    //the data register
                regdst <= 1'b0;      //rt
            end
            WB_LS:            // Write Back of Load
            begin
                write_dr <= 1'b0; //reset
                //jump to the next state and stage
                state <=IF;
                insn_stage <= STAGE_IF ;
                iord <= 1'b0;     //PC
                alu_srcA <= 1'b0; //PC
                alu_srcB <= 2'b01;   //4
                alu_ctrl <= ADD;
                write_reg <= 1'b0;   //reset
            end
            EX_ST:            // Excution of Store
            begin
                write_a <= 1'b0;     //reset
                write_b <= 1'b0;     //reset
                iord <= 1'b1;        //registerC
                alu_srcA <= 1'b1;    //registerA
                alu_srcB <= 2'b10;   //signed-extended immediate
                alu_ctrl <= ADD;
                write_mem <= 1'b1;   //enable
                write_c <= 1'b1;  //write the ALU result into registerC
                //jump to the next state and stage
                state <= MEM_ST;
                insn_stage <= STAGE_MEM ;
            end
            MEM_ST:              // Memory Access of Store
            begin
                write_c <= 1'b0;  //reset
                //jump to the next state and stage
                state <=IF;
                insn_stage <= STAGE_IF;
                iord <= 1'b0;        //PC
                alu_srcA <= 1'b0;    //PC
                alu_srcB <= 2'b01;   //4
                alu_ctrl <= ADD;
            end
            EX_J:             // Excution of Jump
            begin
                write_a <= 1'b0;  //reset
                write_b <= 1'b0;  //reset
                write_c <= 1'b0;  //reset
                write_pc <= 1'b1; //enable
                //jump to the next state and stage
                state <=IF;
                insn_stage <= STAGE_IF;
                pcsource <= 2'b10;   //jump address
                alu_srcA <= 1'b0;    //PC
                alu_srcB <= 2'b01;   //4
                alu_ctrl <= ADD;
            end
```

```
        EX_BEQ:              // Excution of Beq
        begin
            write_a <= 1'b0;     //reset
            write_b <= 1'b0;     //reset
            write_c <= 1'b0;     //reset
            alu_srcA <= 1'b1;    //registerA
            alu_srcB <= 2'b00;   //registerB
            alu_ctrl <= SUB;
            pcsource <= 2'b01;   //registerC
            //zero signal decides whether to write PC
            write_pc <= zero;
            //jump to the next state and stage
            state <=IF;
            insn_stage <= STAGE_IF;
            iord <= 1'b0;        //PC
            alu_srcA <= 1'b0;    //PC
            alu_srcB <= 2'b01;   //4
            alu_ctrl <= ADD;
        end
        default:
        begin
            state <= IF;
            insn_stage <= STAGE_IF;
            iord <= 1'b0;        //PC
            alu_srcA <= 1'b0;    //PC
            alu_srcB <= 2'b01;   //4
            alu_ctrl <= ADD;
        end
    endcase
end
always @ (ir_data)
    case (ir_data[31:26])
        6'b000000:    // R type insn
        begin
        case(ir_data[5:0])
        6'b100000: insn_code <= LED_AD;
        6'b100010: insn_code <= LED_SU;
        6'b100100: insn_code <= LED_AN;
        6'b100111: insn_code <= LED_NO;
        default:  insn_code <= LED_AD;
        endcase
        insn_type <= LED_R;
        end
        6'b000010:  // Jump insn
        begin
            insn_code <= LED_JP;
            insn_type <= LED_J;
        end
        6'b100011:  //   Load
        begin
            insn_code <= LED_LD;
            insn_type <= LED_I;
        end
        6'b101011:  // Store
        begin
            insn_code <= LED_ST;
            insn_type <= LED_I;
        end
        default:  // Not an instruction
        begin
            insn_type <= LED_N;
```

```
            insn_code <= LED_NI;
          end
       endcase
endmodule
//Register File, which encapsulates the registers and some multiplexors
module reg_wrapper(clk, rst, ir_data, dr_data, sw,c_data, memtoreg,
regdst, write_reg,
                   rdata_A, rdata_B, rdata_C, r6out);
    input clk;    //clock
    input rst;    //reset
    input [4:0] sw;//displayed register address, decided by switch input
    input [31:0] ir_data;    //instruction
    input [31:0] dr_data;    //the data register
    input [31:0] c_data;     //registerC
    //control signals
    input memtoreg;
    input regdst;
    input write_reg;

    output [31:0] rdata_A;   //registerA's data
    output [31:0] rdata_B;   //registerB's data
    output [31:0] rdata_C;   //registerC's data
    output [7:0] r6out;      //the lowest 8 bit of register6's data
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [4:0] nd;
    wire [31:0] ni;

    assign rs = ir_data[25:21];
    assign rt = ir_data[20:16];
    assign rd = ir_data[15:11];
    assign nd = regdst? rd : rt;
    assign ni = memtoreg? dr_data : c_data;
    regs  x_regs( .clk(clk),
                  .rst(rst),
                  .rnum_A(rs),
                  .rnum_B(rt),
                  .rnum_C(sw),
                  .wnum(nd),
                  .wdata(ni),
                  .we(write_reg),
                  .rdata_A(rdata_A),
                  .rdata_B(rdata_B),
                  .rdata_C(rdata_C),
                  .r6out(r6out));
endmodule
//regs module
module regs(clk, rst, rnum_A, rnum_B, rnum_C, wnum, wdata, we, rdata_A,
rdata_B, rdata_C, r6out);
    input       clk;         //clock
    input       rst;         //reset
    input [4:0]  rnum_A;     //registerA's address
    input [4:0]  rnum_B;     //registerB's address
    input [4:0]  rnum_C;     //registerC's address
    input [4:0]  wnum;       //writtten register
    input [31:0] wdata;      //writtten data
    input       we;          //write enable
    output [31:0] rdata_A;   //registerA's data
    output [31:0] rdata_B;   //registerB's data
    output [31:0] rdata_C;   //registerC's data
```

```
output [6:0] r6out;      //the lowest 7 bit of register6's data
wire        clk;
wire        rst;
wire [4:0]   rnum_A;
wire [4:0]   rnum_B;
wire [4:0]  rnum_C;
wire [4:0]  rnum_C_old;
wire [4:0]   wnum;
wire [31:0]  wdata;
wire        we;
reg [31:0]   rdata_C;
reg [31:0]   rdata_A;
reg [31:0]   rdata_B;
wire [7:0] r6out;

reg [31:0]   r0;
reg [31:0]   r1;
reg [31:0]   r2;
reg [31:0]   r3;
reg [31:0]   r4;
reg [31:0]   r5;
reg [31:0]   r6;
reg [31:0]   r7;
reg [31:0]   r8;
reg [31:0]   r9;
reg [31:0]   r10;
reg [31:0]   r11;
reg [31:0]   r12;
reg [31:0]   r13;
reg [31:0]   r14;
reg [31:0]   r15;

assign r6out = r6[6:0];
//if registerC's content is changed, renew the output
always @ (rnum_C)
begin
    case (rnum_C)
         5'b00000: rdata_C <= r0;
         5'b00001: rdata_C <= r1;
         5'b00010: rdata_C <= r2;
         5'b00011: rdata_C <= r3;
         5'b00100: rdata_C <= r4;
         5'b00101: rdata_C <= r5;
         5'b00110: rdata_C <= r6;
         5'b00111: rdata_C <= r7;
         5'b01000: rdata_C <= r8;
         5'b01001: rdata_C <= r9;
         5'b01010: rdata_C <= r10;
         5'b01011: rdata_C <= r11;
         5'b01100: rdata_C <= r12;
         5'b01101: rdata_C <= r13;
         5'b01110: rdata_C <= r14;
         5'b01111: rdata_C <= r15;
         default:  rdata_C <= r0;
    endcase
end
always @ (posedge clk or posedge rst)
begin
    if (rst == 1)
    begin
        r0 <= 0;
```

```
        r1 <= 0;
        r2 <= 0;
        r3 <= 0;
        r4 <= 0;
        r5 <= 0;
        r6 <= 0;
        r7 <= 0;
        r8 <= 0;
        r9 <= 0;
        r10 <= 0;
        r11 <= 0;
        r12 <= 0;
        r13 <= 0;
        r14 <= 0;
        r15 <= 0;
    end
    else if (we == 1)
    begin
        case (wnum)
            5'b00000: r0 <= 0;
            5'b00001: r1 <= wdata;
            5'b00010: r2 <= wdata;
            5'b00011: r3 <= wdata;
            5'b00100: r4 <= wdata;
            5'b00101: r5 <= wdata;
            5'b00110: r6 <= wdata;
            5'b00111: r7 <= wdata;
            5'b01000: r8 <= wdata;
            5'b01001: r9 <= wdata;
            5'b01010: r10 <= wdata;
            5'b01011: r11 <= wdata;
            5'b01100: r12 <= wdata;
            5'b01101: r13 <= wdata;
            5'b01110: r14 <= wdata;
            5'b01111: r15 <= wdata;
          default:  r0 <= 0;
        endcase
end
else
begin
   case(rnum_A)
        5'b00000: rdata_A <= r0;
        5'b00001: rdata_A <= r1;
        5'b00010: rdata_A <= r2;
        5'b00011: rdata_A <= r3;
        5'b00100: rdata_A <= r4;
        5'b00101: rdata_A <= r5;
        5'b00110: rdata_A <= r6;
        5'b00111: rdata_A <= r7;
        5'b01000: rdata_A <= r8;
        5'b01001: rdata_A <= r9;
        5'b01010: rdata_A <= r10;
        5'b01011: rdata_A <= r11;
        5'b01100: rdata_A <= r12;
        5'b01101: rdata_A <= r13;
        5'b01110: rdata_A <= r14;
        5'b01111: rdata_A <= r15;
        default:  rdata_A <= r0;
   endcase
   case(rnum_B)
      5'b00000: rdata_B <= r0;
```

```
            5'b00001: rdata_B <= r1;
            5'b00010: rdata_B <= r2;
            5'b00011: rdata_B <= r3;
            5'b00100: rdata_B <= r4;
            5'b00101: rdata_B <= r5;
            5'b00110: rdata_B <= r6;
            5'b00111: rdata_B <= r7;
            5'b01000: rdata_B <= r8;
            5'b01001: rdata_B <= r9;
            5'b01010: rdata_B <= r10;
            5'b01011: rdata_B <= r11;
            5'b01100: rdata_B <= r12;
            5'b01101: rdata_B <= r13;
            5'b01110: rdata_B <= r14;
            5'b01111: rdata_B <= r15;
            default:  rdata_B <= r0;
        endcase
     end
   end
endmodule
```

四、 实验结果分析

Program for verification is as follows:

MIPS code	Instruction data	Comment
LW r1, 20($0)	0x8C01_0014	R1=0xBEEF_0000
LW r2, 21($0)	0x8C02_0015	R2=0x0000_BEEF
ADD r3, r1, r2	0x0022_1820	R3=0xBEEF_BEEF
SUB r4, r1, r2	0x0022_2022	R4=0xBEEE_4111
AND r5, r3, r4	0x0064_2824	R5=0xBEEE_0001
NOR r6, r4, r5	0x0085_3027	R6=0x4111_BEEE
SW r6, 22($0)	0xAC06_0016	22=0x4111_BEEE
J 0	0x0800_0000	PC=0

The result of execution is as follows:

MIPS code	Instruction data	Type	Code	State	Execution
LW r1, 20($0)	0x8C01_0014	3	1 (LD)	0	IF, PC=0
				1	ID, PC=1
				3	EX_LD
				5	MEM_RD,ReadMemAddr=0x14=20
				9	WB_LS, R1=0xBEEF_0000
LW r2, 21($0)	0x8C02_0015	3	1 (LD)	0	IF, PC=1
				1	ID, PC=2
				3	EX_LD

				5	MEM_RD,ReadMemAddr=0x15=21
				9	WB_LS, R2=0x0000_BEEF
ADD r3, r1, r2	0x0022_1820	1	3 (AD)	0	IF, PC=2
				1	ID, PC=3
				2	EX_R
				8	WB_R, R3=0xBEEF_BEEF
SUB r4, r1, r2	0x0022_2022	1	4 (SU)	0	IF, PC=3
				1	ID, PC=4
				2	EX_R
				8	WB_R, R4=0xBEEE_4111
AND r5, r3, r4	0x0064_2824	1	5 (AN)	0	IF, PC=4
				1	ID, PC=5
				2	EX_R
				8	WB_R, R5=0xBEEE_0001
NOR r6, r4, r5	0x0085_3027	1	6 (NO)	0	IF, PC=5
				1	ID, PC=6
				2	EX_R
				8	WB_R, R6=0x4111_BEEE
SW r6, 22($0)	0xAC06_0016	3	2 (ST)	0	IF, PC=6
				1	ID, PC=7
				4	EX_ST, WriteMemAddr=0x16=22
				7	MEM_ST,Mem(22)=0x4111_BEEE
J 0	0x0800_0000	2	7 (JP)	0	IF, PC=7
				1	ID, PC=8
				b	EX_J, PC=0

The result indicates that the code implements the required functions correctly.

五、 讨论与心得

What problem you met and how do you solve it.

I’ve added some codes in the EX_J state in the ctrl module compared to the give reference code:

```
alu_srcA <= 1'b0;     //PC
alu_srcB <= 2'b01;    //4
alu_ctrl <= ADD;
```

If the codes above is not added, when a jmp instruction finishes, the state transfers to IF. Note that `alu_srcA`、`alu_srcB`、`alu_ctrl` have not been initialized(and nor of them have been initialized appropriately before), so `write_pc <= 1'b1;` will make the value of PC wrong, that is to say, not equal to PC+4.