浙江大学

本科实验报告

课程名称：	计算机组成
姓　名：	Parsa 帕萨
学　院：	计算机科学与技术学院
系：	计算机科学与技术系
专　业：	中加班
学　号：	3170300180
指导教师：	姜晓红

2019 年　05 月　31 日

浙江大学实验报告

课程名称： 计算机组成 实验类型： 综合

实验项目名称： Lab 3: Designing a multi cycle mips cpu with datapath and controllers

学生姓名： Parsa 帕萨 专业： 中加班 学号： 3170300180

同组学生姓名： None 指导老师： 姜晓红

实验地点： 东 4-510 实验日期：2019 年 05 月 31 日

1 – Purpose of the Experiments and Tasks

i. Design the controller of the multi-cycle cpu

ii. Design and implementation of datapath for multi-cycle cpu

2 – Introduction

This experiment is built on top of the previous experiment where we designed a single-cycle mips cpu. The problem with our previous implementation is that each cycle is limited to performing one instruction. This means that each cycle takes as long as the longest instruction, which is in this case *load word*. As their name suggests, multi-cycle cpus perform one instruction in multiple cycles. As a result, shorter instructions take shorter time, while longer instructions would take a similar length to execute. This means faster clock speeds, which further means faster cpus. Although single cycle cpus provide a very simple implementation, but their speed is a big trade off.

3 – Implementation

To overcome the slowness of single cycle cpus, we can divide the instructions into different steps for their execution. We can breakdown these steps into 5 categories:

1. Instruction Fetch: Reading the instruction from memory
2. Instruction Decode: Reading source registers, and decide type of the instruction
3. Execute: Compute an operation
4. Memory: Read or write to the memory
5. Write Back: Store the results into the destination register

What do these steps mean? They mean given an instruction; we break down its execution into smaller steps to speed up the process. Unlike the single cycle CPU, each instruction would take few cycles to be executed. Depending on the type of instruction, it would go through different step and during each step a different control signal is activated in the datapath module which would execute that command. Before we get into the specifications of the controller module, it would be better to review the CPU design.

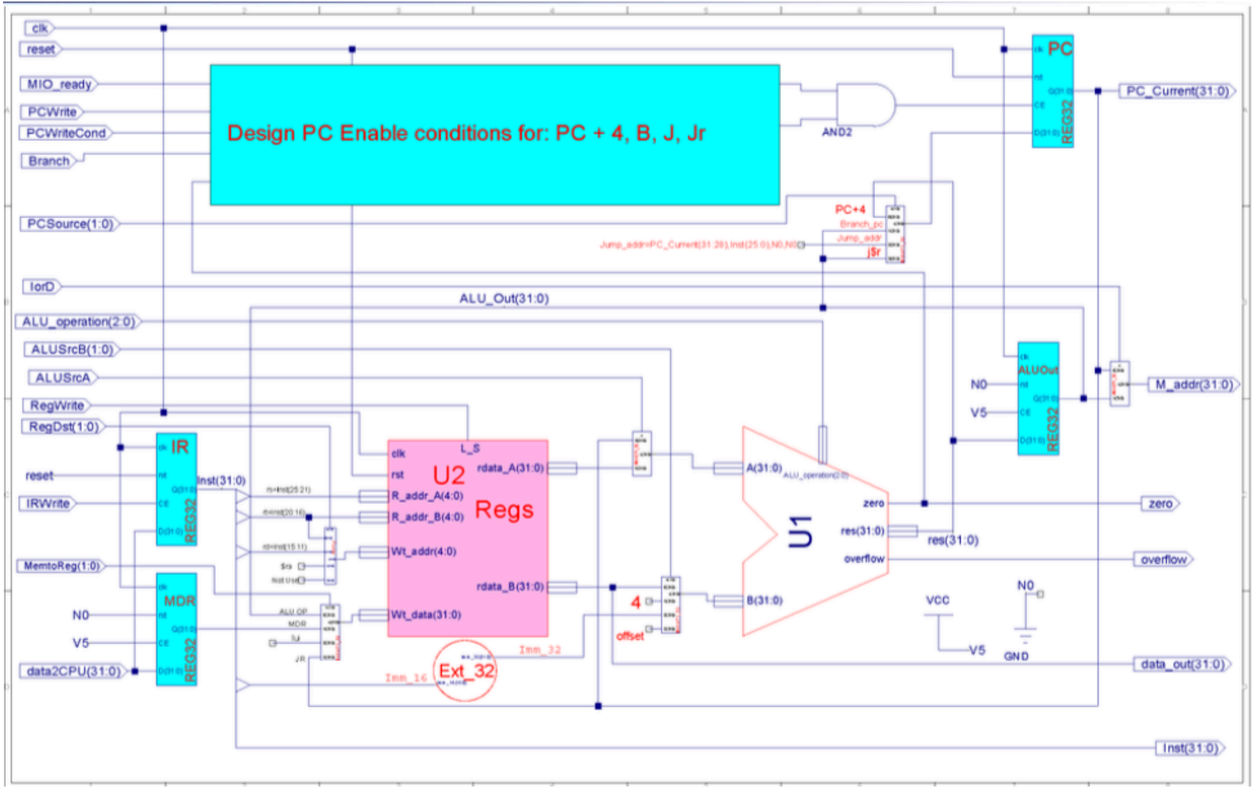


Figure 1 Design of Multi-Cycle MIPS CPU

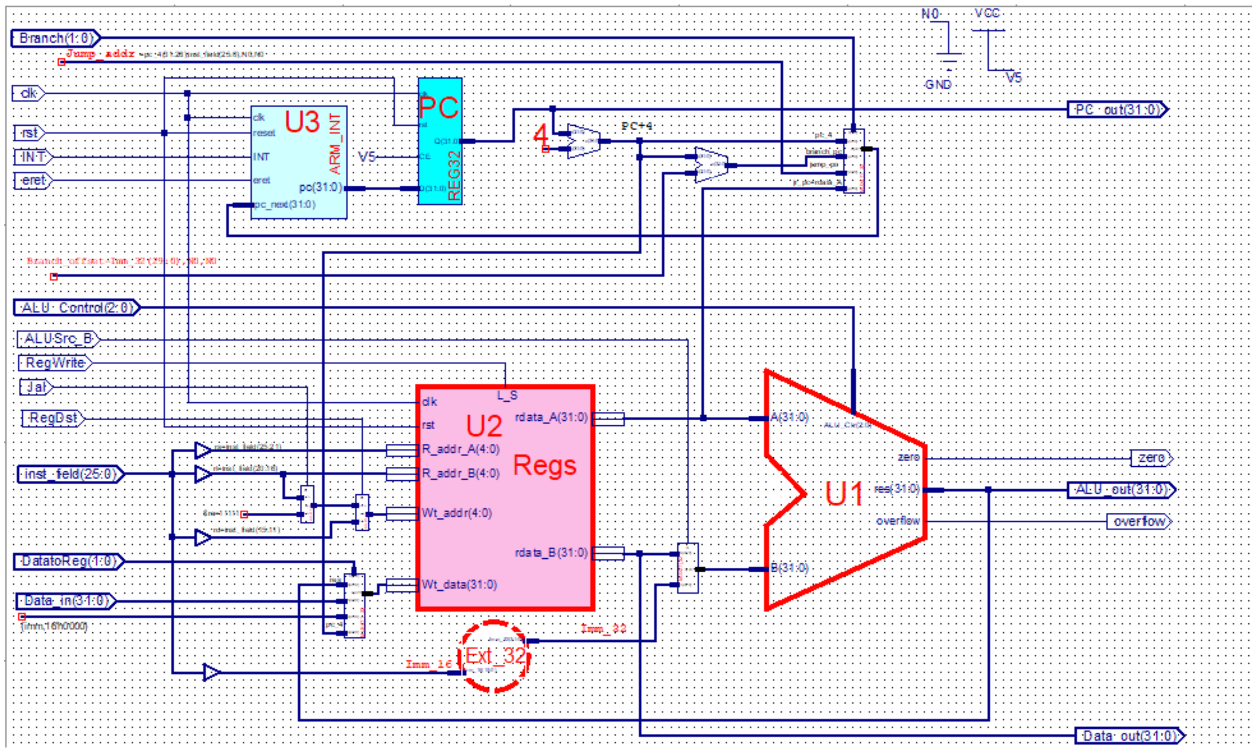


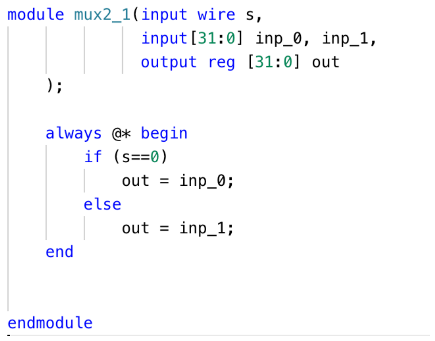
Figure 2 Datapath for single cycle MIPS CPU

As you can see, we removed two adders in the design of multi-cycle, all additions are done in the ALU module. This should be taken care of with our controller module, which commands the datapath module what to do. To control the flow of the instruction, we have a module called reg32, which is activated by IRWrite signal. Here is the implementation of this:

```
module reg32(input clk,
             input rst,
             input CE,
             input [31:0]D,
             output reg[31:0]Q);

    always @(posedge clk or posedge rst)
        if (rst==1)  Q <= 32'h00000000;
        else if (CE) Q <= D;

endmodule
```

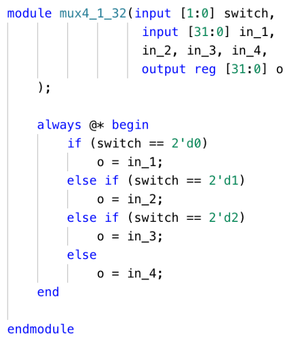
Figure 3 Reg32 module implementation

As it can be seen, the CE is the signal that allows the flow of input, otherwise it’s resetted by rst signal. This module is essential to our multi-cycle design because this way we can control the flow of data. The second instance of this module is used for loading data to the register, assuming that the incoming data2PC is data (used as a buffer for the incoming data), then we use MemtoReg as the switch for our multiplexer to feed the data to Wt_data for our registers; the data comes from a multiplexer that takes the result of our ALU, the buffered data, results of load upper immediate and jr instruction. Similar to our single cycle, we use a multiplexer to determine the address we want to write to; the inputs of this multiplexer are instruction[20:16], instruction[15:11] and return address. The next change takes place right after the register module, where instead of performing additions separately we can take advantage of ALU module and using a simple multiplexer and we can control the entry data, whether is the PC data or read data from register that is going to be fed into the ALU module. The controller for this multiplexer is ALUSrcA. Similarly we have a multiplexer with four inputs; this multiplexer takes read data from register module, 4 (for reading the next instruction), instruction[15:0] and offset. Similarly, we have a signal from the controller module called ALUSrcB for controlling this multiplexer. The implementations of these modules are as such:



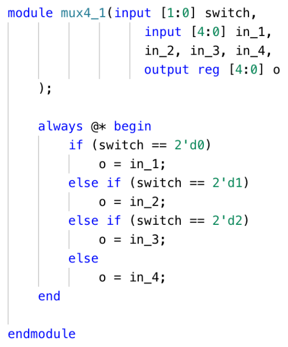
```
module mux2_1(input wire s,
              input[31:0] inp_0, inp_1,
              output reg [31:0] out
    );

    always @* begin
        if (s==0)
            out = inp_0;
        else
            out = inp_1;
    end


endmodule
```

Figure 4Multiplexer 2 -> 1 32bits



```
module mux4_1_32(input [1:0] switch,
                 input [31:0] in_1,
                 in_2, in_3, in_4,
                 output reg [31:0] o
    );

    always @* begin
        if (switch == 2'd0)
            o = in_1;
        else if (switch == 2'd1)
            o = in_2;
        else if (switch == 2'd2)
            o = in_3;
        else
            o = in_4;
    end

endmodule
```

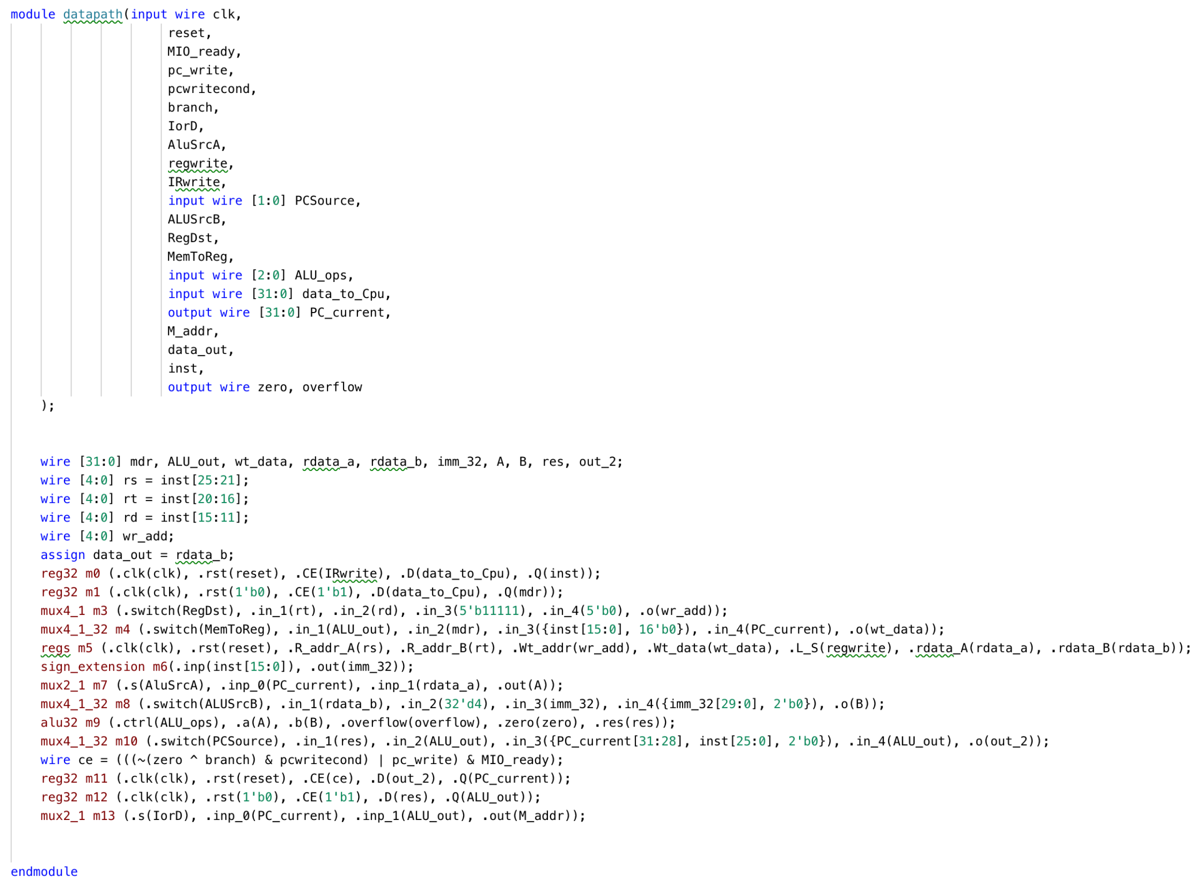
Figure 5 Multiplexer 4 -> 1 32bits



```
module mux4_1(input [1:0] switch,
              input [4:0] in_1,
              in_2, in_3, in_4,
              output reg [4:0] o
    );

    always @* begin
        if (switch == 2'd0)
            o = in_1;
        else if (switch == 2'd1)
            o = in_2;
        else if (switch == 2'd2)
            o = in_3;
        else
            o = in_4;
    end

endmodule
```

Figure 6 Multiplexer 4 -> 1 5bits

The next difference is the output results of the ALU, previously we had it directly head out or go to our register write. But in the multi-cycle cpu, the alu output is not only the result of arithmetic result of two registers but also, it is used for the addition of addresses, because we removed the adder. Further, the output is sent to a multiplexer and buffer. The reason it is sent to the multiplexer is for getting the next pc (current pc + 4) and for the buffer it is used as a memory to be read from ram. The mentioned multiplexer also takes in branch address, jump address and register address for jumping, which is then fed into another buffer then it goes to the pc_current, which is our current pc or the next instruction to be ran. And with respect to our ALUout buffer, it goes into a multiplexer with IorD signal which indicates either to set M_addr as PC or ALUout (the input data for lw). Another difference is our data_out output, it is set by data_b output from our register module. The full implementation of the datapath for multi-cycle Mips cpu is as follows:



```
module datapath(input wire clk,
                    reset,
                    MIO_ready,
                    pc_write,
                    pcwritecond,
                    branch,
                    IorD,
                    AluSrcA,
                    regwrite,
                    IRwrite,
                    input wire [1:0] PCSource,
                    ALUSrcB,
                    RegDst,
                    MemToReg,
                    input wire [2:0] ALU_ops,
                    input wire [31:0] data_to_Cpu,
                    output wire [31:0] PC_current,
                    M_addr,
                    data_out,
                    inst,
                    output wire zero, overflow
    );


    wire [31:0] mdr, ALU_out, wt_data, rdata_a, rdata_b, imm_32, A, B, res, out_2;
    wire [4:0] rs = inst[25:21];
    wire [4:0] rt = inst[20:16];
    wire [4:0] rd = inst[15:11];
    wire [4:0] wr_add;
    assign data_out = rdata_b;
    reg32 m0 (.clk(clk), .rst(reset), .CE(IRwrite), .D(data_to_Cpu), .Q(inst));
    reg32 m1 (.clk(clk), .rst(1'b0), .CE(1'b1), .D(data_to_Cpu), .Q(mdr));
    mux4_1 m3 (.switch(RegDst), .in_1(rt), .in_2(rd), .in_3(5'b11111), .in_4(5'b0), .o(wr_add));
    mux4_1_32 m4 (.switch(MemToReg), .in_1(ALU_out), .in_2(mdr), .in_3({inst[15:0], 16'b0}), .in_4(PC_current), .o(wt_data));
    regs m5 (.clk(clk), .rst(reset), .R_addr_A(rs), .R_addr_B(rt), .Wt_addr(wr_add), .Wt_data(wt_data), .L_S(regwrite), .rdata_A(rdata_a), .rdata_B(rdata_b));
    sign_extension m6(.inp(inst[15:0]), .out(imm_32));
    mux2_1 m7 (.s(AluSrcA), .inp_0(PC_current), .inp_1(rdata_a), .out(A));
    mux4_1_32 m8 (.switch(ALUSrcB), .in_1(rdata_b), .in_2(32'd4), .in_3(imm_32), .in_4({imm_32[29:0], 2'b0}), .o(B));
    alu32 m9 (.ctrl(ALU_ops), .a(A), .b(B), .overflow(overflow), .zero(zero), .res(res));
    mux4_1_32 m10 (.switch(PCSource), .in_1(res), .in_2(ALU_out), .in_3({PC_current[31:28], inst[25:0], 2'b0}), .in_4(ALU_out), .o(out_2));
    wire ce = (((~(zero ^ branch) & pcwritecond) | pc_write) & MIO_ready);
    reg32 m11 (.clk(clk), .rst(reset), .CE(ce), .D(out_2), .Q(PC_current));
    reg32 m12 (.clk(clk), .rst(1'b0), .CE(1'b1), .D(res), .Q(ALU_out));
    mux2_1 m13 (.s(IorD), .inp_0(PC_current), .inp_1(ALU_out), .out(M_addr));


endmodule
```

Figure 7 Implementation of datapath for multi-cycle cpu

Wire ce is the activation signal for the buffer that sets the output for PC_Current. The signal is activated by combination of MIO_ready AND (PCWrite OR (PCWriteCond AND (inv(Branch XOR zero)))).

Now that we have the datapath implemented, as discussed, we can divide the instruction execution into 5 steps: IF, De, Ex, Mem and WB. The state diagram below shows how we can set the signals for each step throughout running an instruction. Each type of instruction follows a different path and requires a certain number of steps to be executed. Jump instructions are the shortest instructions to be ran on the multi-cycle cpus with only needing to run through instruction fetch, decoder and execute. The longest instruction is lw which requires to go through all the required steps which include instruction fetch, decoder, execute, memory and finally write back. The diagram below puts all these signals together and shows how the controller module should be designed with respect to each instruction group:

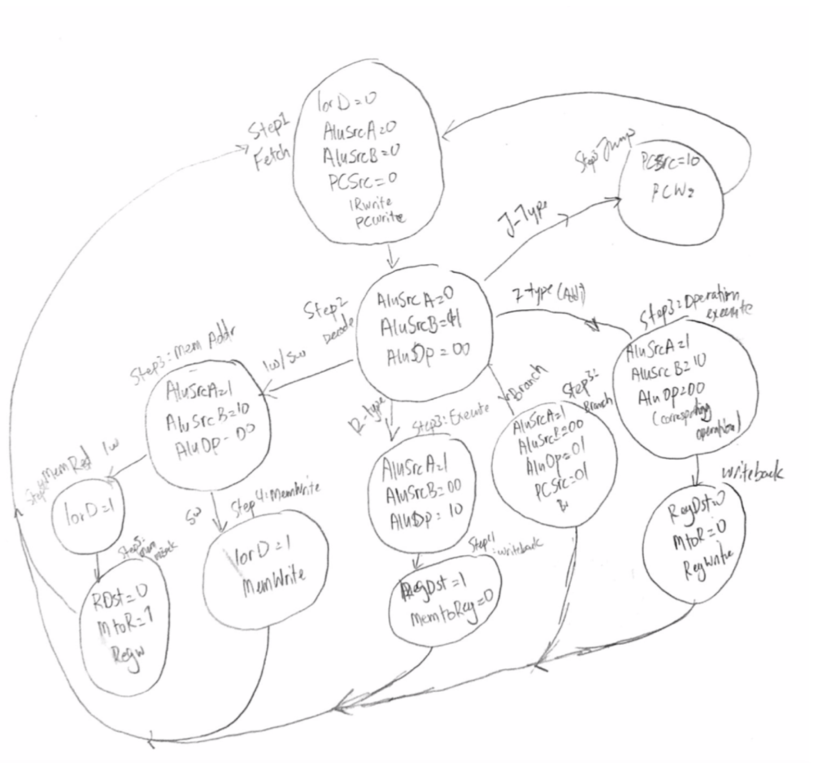
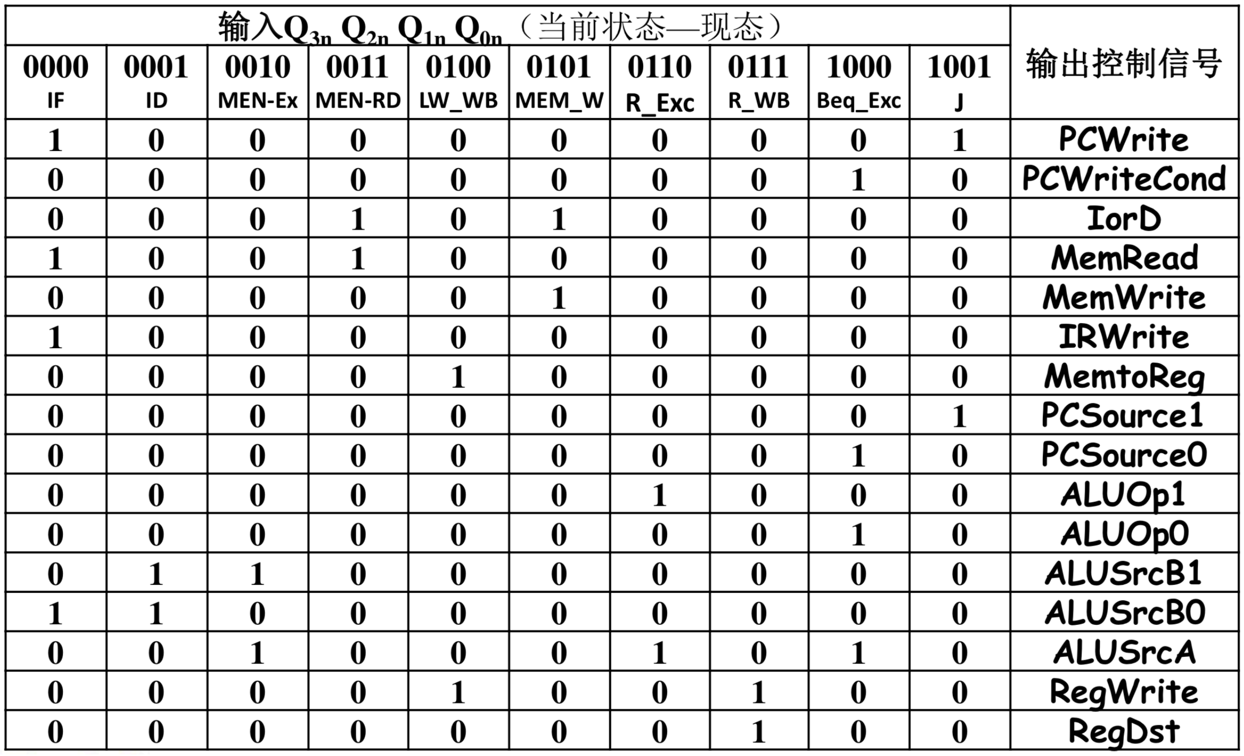


Figure 8 State diagram for instructions and their relative control signals

Here is the truth table for different states of the controllers to control the operations, which it inspired me to do mine; unlike the following table, instead of having 10 states, I have the 5 states, and within each case I set each control signal depending on the instruction.



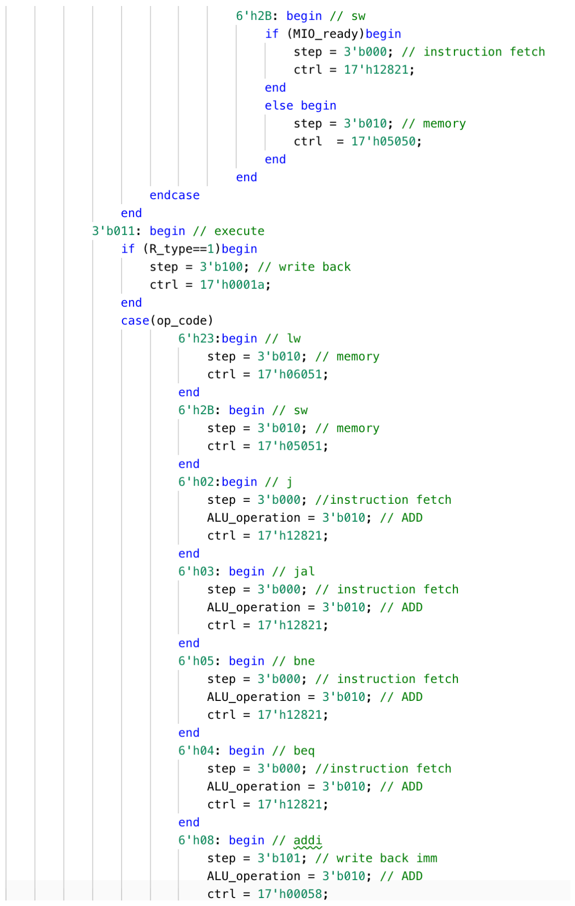
输入Q$_{3n}$ Q$_{2n}$ Q$_{1n}$ Q$_{0n}$（当前状态—现态）										输出控制信号
0000 IF	0001 ID	0010 MEN-Ex	0011 MEN-RD	0100 LW_WB	0101 MEM_W	0110 R_Exc	0111 R_WB	1000 Beq_Exc	1001 J	
1	0	0	0	0	0	0	0	0	1	PCWrite
0	0	0	0	0	0	0	0	1	0	PCWriteCond
0	0	0	1	0	1	0	0	0	0	IorD
1	0	0	1	0	0	0	0	0	0	MemRead
0	0	0	0	0	1	0	0	0	0	MemWrite
1	0	0	0	0	0	0	0	0	0	IRWrite
0	0	0	0	1	0	0	0	0	0	MemtoReg
0	0	0	0	0	0	0	0	0	1	PCSource1
0	0	0	0	0	0	0	0	1	0	PCSource0
0	0	0	0	0	0	1	0	0	0	ALUOp1
0	0	0	0	0	0	0	0	1	0	ALUOp0
0	1	1	0	0	0	0	0	0	0	ALUSrcB1
1	1	0	0	0	0	0	0	0	0	ALUSrcB0
0	0	1	0	0	0	1	0	1	0	ALUSrcA
0	0	0	0	1	0	0	1	0	0	RegWrite
0	0	0	0	0	0	0	1	0	0	RegDst

Figure 9 Truth table for controllers

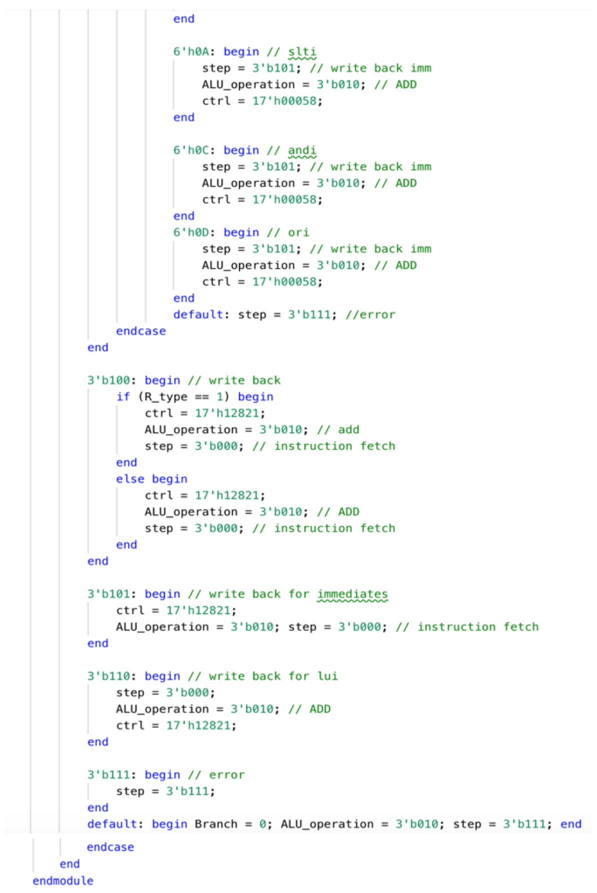
Here is the full implementation of controller module for our multi-cycle CPU:

```verilog
module controller(input  clk,
                    input  reset,
                    input  [31:0] Inst_in,
                    input  zero,
                    input  overflow,
                    input  MIO_ready,
                    output reg MemRead,
                    output reg MemWrite,
                    output reg[2:0]ALU_operation,
                    output reg CPU_MIO,
                    output reg IorD,
                    output reg IRWrite,
                    output reg [1:0]RegDst,
                    output reg RegWrite,
                    output reg [1:0]MemtoReg,
                    output reg ALUSrcA,
                    output reg [1:0]ALUSrcB,
                    output reg [1:0]PCSource,
                    output reg PCWrite,
                    output reg PCWriteCond,
                    output reg Branch);


    wire [5:0] op_code = Inst_in[31:26];
    wire R_type = (op_code == 6'b0) ? 1:0;
    reg [2:0] step;
    reg [16:0] ctrl;
    always @(posedge clk or posedge reset) begin
        {PCWrite,PCWriteCond,IorD,MemRead,MemWrite,IRWrite,MemtoReg,PCSource,ALUSrcB,ALUSrcA,RegWrite,RegDst, CPU_MIO} = ctrl;
        if (reset == 0)begin
            ctrl = 17'h12821;
            ALU_operation = 3'b010;
            step = 3'b0; // instruction fetch
        end
        case (step)
            3'b000: begin // instruction fetch
                if (MIO_ready == 1)begin
                    ALU_operation = 3'b010;
                    step = 3'b001;
                end else step = 3'b000;
            end
            3'b001: begin //instruction decode
                if (R_type == 1)begin
                    step = 3'b011; // execute
                    ctrl = 17'h00010;
                    case (Inst_in[5:0])
                        6'd32: ALU_operation = 3'b010; // ADD
                        6'd34: ALU_operation = 3'b110; // SUB
                        6'd36: ALU_operation = 3'b000; // AND
                        6'd37: ALU_operation = 3'b001; // OR
                        6'd38: ALU_operation = 3'b011; // XOR
                        6'd39: ALU_operation = 3'b100; // NOR
                        6'd42: ALU_operation = 3'b111; // SLT
                        6'd02: ALU_operation = 3'b101; // SRL
                        6'd08: begin  // JR
                            ctrl = 17'h10010;
                            ALU_operation = 3'b010; step = 3'b010; //execute
                            end
                        default: ALU_operation = 3'b010; // ADD
                    endcase
                    end
                else
                    case(op_code)
                        6'h23:begin // lw
                            ctrl = 17'h00050;
                            ALU_operation = 3'b010; // ADD
                            step = 3'b011; // execute
                        end
                        6'h2B: begin // sw
                            ctrl = 17'h00050;
                            ALU_operation = 3'b010; //ADD
                            step = 3'b011; // execute
                        end
                        6'h02:begin // j
                            ctrl = 17'h10160;
                            step = 3'b011; //execute
                        end
                        6'h03: begin // jal
                            ctrl = 17'h1076c;
                            step = 3'b011; // execute
                        end
                        6'h05: begin // bne
                            ctrl = 17'h08090;
                            Branch = 0;
                            ALU_operation = 3'b110; //SUB
                            step = 3'b011; // execute
                        end
                        6'h04: begin // beq
```

```verilog
                            ctrl = 17'h08090;
                            Branch = 1;
                            ALU_operation = 3'b110; //SUB
                            step = 3'b011; //execute
                        end
                        6'h08: begin // addi
                            ctrl = 17'h00050;
                            ALU_operation = 3'b010; // ADD
                            step = 3'b011; // execute
                        end

                        6'h0A: begin // slti
                            ctrl =  17'h00050;
                            ALU_operation = 3'b111; // SLT
                            step = 3'b011; // execute
                        end

                        6'h0C: begin // andi
                            ctrl = 17'h00050;
                            ALU_operation = 3'b0; //AND
                            step = 3'b011; // execute
                        end
                        6'h0D: begin // ori
                            ctrl = 17'h00050;
                            ALU_operation = 3'b001; // or
                            step = 3'b011; // execute
                        end
                        6'h0F: begin // lui
                            ctrl = 17'h00468;
                            step = 3'b101; // write back lui
                        end
                        default: step = 3'b111; //error
                    endcase

            end
        3'b010: begin // memory
                case (op_code)
                        6'h23:begin // lw
                            if (MIO_ready)begin
                                step = 3'b100; // write back
                                ctrl = 17'h00208;
                            end
                            else begin
                                step = 3'b010; // memory
                                ctrl = 17'h06050;
                            end
                        end
```



```
                        6'h2B: begin // sw
                            if (MIO_ready)begin
                                step = 3'b000; // instruction fetch
                                ctrl = 17'h12821;
                            end
                            else begin
                                step = 3'b010; // memory
                                ctrl  = 17'h05050;
                            end
                        end
                endcase
            end
        3'b011: begin // execute
            if (R_type==1)begin
                step = 3'b100; // write back
                ctrl = 17'h0001a;
            end
            case(op_code)
                    6'h23:begin // lw
                        step = 3'b010; // memory
                        ctrl = 17'h06051;
                    end
                    6'h2B: begin // sw
                        step = 3'b010; // memory
                        ctrl = 17'h05051;
                    end
                    6'h02:begin // j
                        step = 3'b000; //instruction fetch
                        ALU_operation = 3'b010; // ADD
                        ctrl = 17'h12821;
                    end
                    6'h03: begin // jal
                        step = 3'b000; // instruction fetch
                        ALU_operation = 3'b010; // ADD
                        ctrl = 17'h12821;
                    end
                    6'h05: begin // bne
                        step = 3'b000; // instruction fetch
                        ALU_operation = 3'b010; // ADD
                        ctrl = 17'h12821;
                    end
                    6'h04: begin // beq
                        step = 3'b000; //instruction fetch
                        ALU_operation = 3'b010; // ADD
                        ctrl = 17'h12821;
                    end
                    6'h08: begin // addi
                        step = 3'b101; // write back imm
                        ALU_operation = 3'b010; // ADD
                        ctrl = 17'h00058;
```

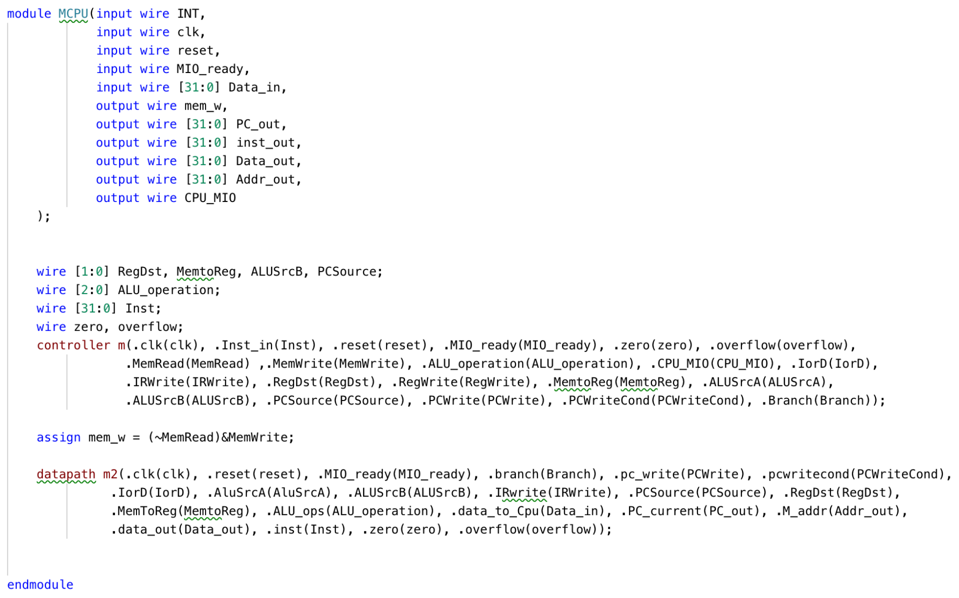


```
                    end

                    6'h0A: begin // slti
                        step = 3'b101; // write back imm
                        ALU_operation = 3'b010; // ADD
                        ctrl = 17'h00058;
                    end

                    6'h0C: begin // andi
                        step = 3'b101; // write back imm
                        ALU_operation = 3'b010; // ADD
                        ctrl = 17'h00058;
                    end
                    6'h0D: begin // ori
                        step = 3'b101; // write back imm
                        ALU_operation = 3'b010; // ADD
                        ctrl = 17'h00058;
                    end
                    default: step = 3'b111; //error
                endcase
            end

            3'b100: begin // write back
                if (R_type == 1) begin
                    ctrl = 17'h12821;
                    ALU_operation = 3'b010; // add
                    step = 3'b000; // instruction fetch
                end
                else begin
                    ctrl = 17'h12821;
                    ALU_operation = 3'b010; // ADD
                    step = 3'b000; // instruction fetch
                end
            end

            3'b101: begin // write back for immediates
                ctrl = 17'h12821;
                ALU_operation = 3'b010; step = 3'b000; // instruction fetch
            end

            3'b110: begin // write back for lui
                step = 3'b000;
                ALU_operation = 3'b010; // ADD
                ctrl = 17'h12821;
            end

            3'b111: begin // error
                step = 3'b111;
            end
            default: begin Branch = 0; ALU_operation = 3'b010; step = 3'b111; end
        endcase
    end
endmodule
```

Figure 10 Implementation of the controller module for Multi-Cycle CPU

As mentioned, the whole implementation is divided differently compared to the proposed one. The Instruction fetch, decoding and execution are just one single case and within each I specify the controllers separately depending on the instruction, where my implementation differs is the writeback; there are three write backs, one for the R-type instructions and lw and sw instructions, one for I-type instructions and the last one for lui.

Now that we have our controller module, we can set up our Multi-Cycle CPU. Here is the module that puts these two modules (datapath and controller) together:



```
module MCPU(input wire INT,
            input wire clk,
            input wire reset,
            input wire MIO_ready,
            input wire [31:0] Data_in,
            output wire mem_w,
            output wire [31:0] PC_out,
            output wire [31:0] inst_out,
            output wire [31:0] Data_out,
            output wire [31:0] Addr_out,
            output wire CPU_MIO
    );


    wire [1:0] RegDst, MemtoReg, ALUSrcB, PCSource;
    wire [2:0] ALU_operation;
    wire [31:0] Inst;
    wire zero, overflow;
    controller m(.clk(clk), .Inst_in(Inst), .reset(reset), .MIO_ready(MIO_ready), .zero(zero), .overflow(overflow),
                .MemRead(MemRead) ,.MemWrite(MemWrite), .ALU_operation(ALU_operation), .CPU_MIO(CPU_MIO), .IorD(IorD),
                .IRWrite(IRWrite), .RegDst(RegDst), .RegWrite(RegWrite), .MemtoReg(MemtoReg), .ALUSrcA(ALUSrcA),
                .ALUSrcB(ALUSrcB), .PCSource(PCSource), .PCWrite(PCWrite), .PCWriteCond(PCWriteCond), .Branch(Branch));

    assign mem_w = (~MemRead)&MemWrite;

    datapath m2(.clk(clk), .reset(reset), .MIO_ready(MIO_ready), .branch(Branch), .pc_write(PCWrite), .pcwritecond(PCWriteCond),
              .IorD(IorD), .AluSrcA(AluSrcA), .ALUSrcB(ALUSrcB), .IRwrite(IRWrite), .PCSource(PCSource), .RegDst(RegDst),
              .MemToReg(MemtoReg), .ALU_ops(ALU_operation), .data_to_Cpu(Data_in), .PC_current(PC_out), .M_addr(Addr_out),
              .data_out(Data_out), .inst(Inst), .zero(zero), .overflow(overflow));


endmodule
```

Figure 11 Implementation of CPU module