多周期MIPS设计 实验报告

17307130178 宁晨然

1. 实验概述

本次实验完成了多周期MIPS设计，实现了多周期处理器的微体系结构。多周期体系结构意味着，时间周期不再受最长时间指令限制，而是分段为多个模块，根据状态机图转换状态，实现多个周期的运转。难点在于整个模块的状态机图，以及每个指令对应的状态。组件基本与单周期类似。

实验使用的是verilog硬件设计语言。

2. 实验展示

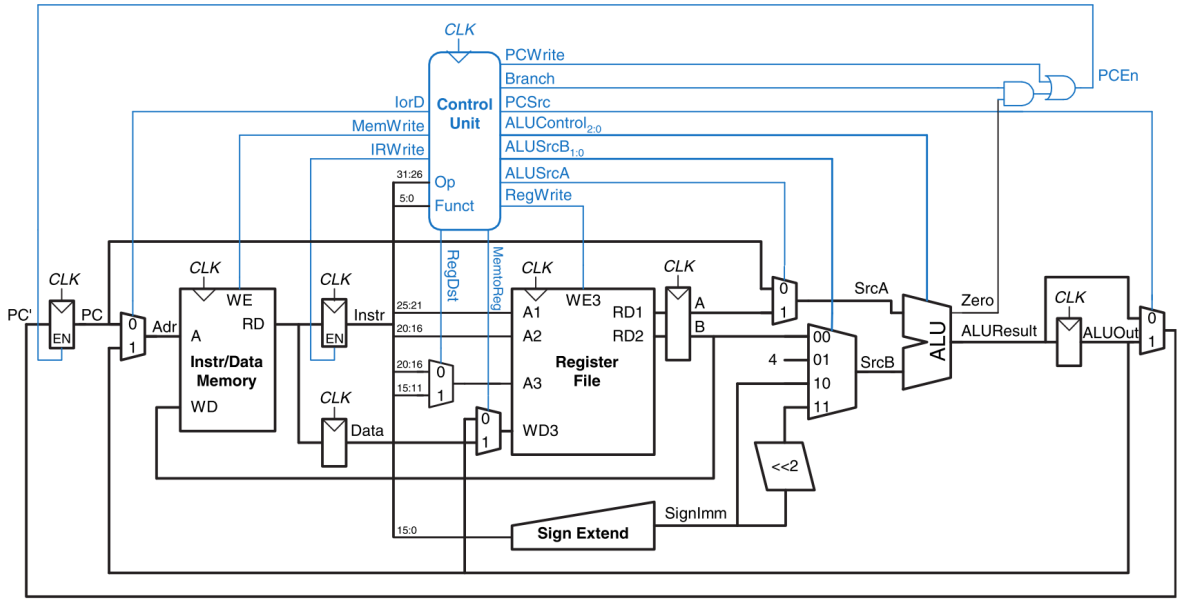
实验完成MIPS多周期处理器的设计，并且增加了指令，修改代码、修改状态机图，并且最终通过了测试。

2.1 实验亮点

本人认为该实验可以达到**A水平**，在此列出本次实验作业的亮点：

- 实现了**MIPS多周期处理器的设计**
- **增加**了指令（具体指令在第五章讲解）
- 实现了**模块化测试和整合仿真测试，并且通过了最终的测试**
- 代码结构清晰，每个函数有**清晰易懂的注释**；**文件结构干净整洁**
- **实验报告清晰易懂，涵盖要点**、提纲挈领

2.2 实验原理



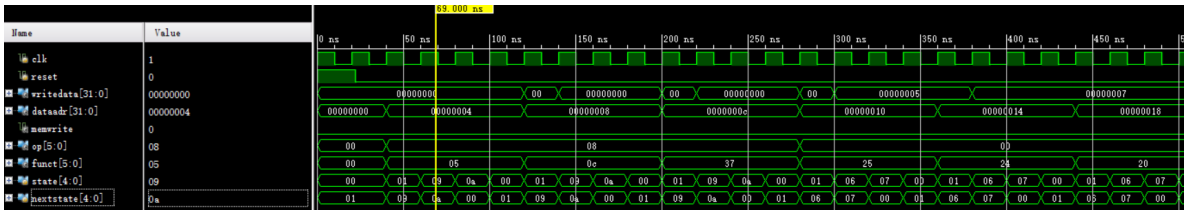
上图是多周期mips设计的模样，整个多周期处理器的构成由两个板块构成：

- MIPS主部件：主要部件由控制通路和数据通路组成，数据通路根据寄存器和储存器中的数据以及控制信号完成指令，控制通路负责根据指令调整控。
- mem：合并了dmem和imem，与单周期类似。

2.3 实验结果

实验成功通过了测试文件。仿真结果如下，均符合正确性。

```
run 200 ns
run 200 ns
Simulation succeeded
```

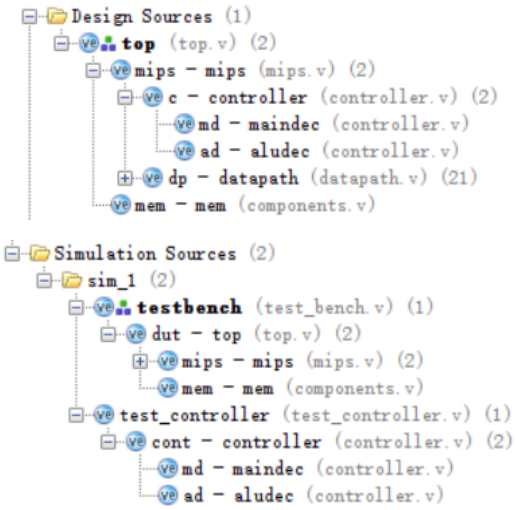


3. 实验代码讲解

先简单介绍本次实验的步骤：

- 第一步：复用单周期mips的构件，重新搭建新的多周期框架
- 第二步：修改dmem和imem，用mem合并两者功能
- 第三步：修改controller和top文件，实现多周期基本框架

- 第四步：实现单元测试，单独对各种模块进行测试，测试无误后再整合文件
- 第五步：实现整合测试，使用test_bench对整个mips的top文件进行测试，测试文件为memfile
- 第六步：添加指令，重复测试流程



下面先简单介绍文件结构。本单元部分介绍代码如何实现和原理，指令的讲解在第五章实验指令详解。

3.1 datapath

数据通路由操作元件和状态元件组合而成。通过分散方式或总线方式连接而成，进行数据存储、处理、传送的路径。下面先分板块讲解。

- 所有数据通路需要调用的元件均写在`components.v`文件中，便于**测试和管理**
- 所有元件均进行了**代码标注**，清晰易懂
- 下面解释很多元件使用了homework2中的代码，实现了功能复用。
- 由于大部分与之前相同，所以不予赘述

components主要实现的功能部件有：

- 寄存器文件 register file
- 左移2
- 带符号扩展、零扩展
- 寄存器
- 多路选择器
- ALU

```
// 3. Register file
// 2 read 1 write
module regfile(
    input clk, wen,
    input [4:0] address1,address2,address3,
    input [31:0] wdata,
    output [31:0] rd1,rd2
);
// 32 32-bit registers
reg [63:0] rf [31:0];
integer i;

// initialization
initial
begin
    for(i=0;i<64;i=i+1)
        rf[i] = 32'b0;
end

// write 1
always @(posedge clk)
begin
    if( wen && (address3!=0)) // write and register 0 cannot be changed
        rf[address3] <= wdata;
end

// read two at same time
assign rd1 = (address1!=0) ? rf[address1] : 0;
assign rd2 = (address2!=0) ? rf[address2] : 0;

endmodule

// 4. others


module sl2(input [31:0] a,
           output [31:0] y);
    assign y = {a[29:0], 2'b00}; // shift left by 2
endmodule

module signext(input [15:0] a,
               output [31:0] y);
    assign y = {{16{a[15]}}, a};  // signed extender
endmodule

module zeroext(input [15:0] a,
               output [31:0] y);  // zero extender
    assign y = {16'b0, a};
endmodule
```

```
// zeroext8_32 is needed for LBU
module zeroext8_32(input [7:0] a,
                   output [31:0] y);
assign y = {24'b0, a};
endmodule

// signext8_32 is needed for LB
module signext8_32(input [7:0] a,
                   output [31:0] y);
assign y = {{24{a[7]}}, a};
endmodule

module flopr #(parameter WIDTH=8)(input clk,reset,
            input   [WIDTH-1:0] data_in,
            output reg [WIDTH-1:0] data_out);
always @(posedge clk, posedge reset)
    if (reset) data_out<=0;
    else data_out<=data_in;
endmodule

module flopenr #(parameter WIDTH = 8)(input clk, reset,
                 input  en,
                 input  [WIDTH-1:0] data_in,
                 output reg [WIDTH-1:0] data_out);
always @(posedge clk, posedge reset)
    if (reset) data_out <= 0;
    else if (en) data_out <= data_in;
endmodule

module mux2#(parameter WIDTH=8)
(
    input [WIDTH-1:0] a,b,
    input s,
    output [WIDTH-1:0] y
);
assign y = s?b:a;
endmodule

module mux3 #(parameter WIDTH = 8)
(
    input [WIDTH-1:0] d0, d1, d2,
    input [1:0] s,
    output [WIDTH-1:0] y
);
    assign #1 y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module mux4 #(parameter WIDTH = 8)
(
    input [WIDTH-1:0] d0, d1, d2, d3,
    input [1:0] s,
    output reg [WIDTH-1:0] y
);
    always@(*)
    begin
        case(s)
            2'b00: y <= d0;
```

```
            2'b01: y <= d1;
            2'b10: y <= d2;
            2'b11: y <= d3;
        endcase
    end
endmodule

module mux5 #(parameter WIDTH = 8)
(
    input [WIDTH-1:0] d0, d1, d2, d3, d4,
    input [2:0] s,
    output reg [WIDTH-1:0] y
);
    always@(*)
    begin
        case(s)
            3'b000: y <= d0;
            3'b001: y <= d1;
            3'b010: y <= d2;
            3'b011: y <= d3;
            3'b100: y <= d4;
        endcase
    end
endmodule

module alu(input  [31:0] a, b,
           input  [2:0]  f,
           output reg [31:0] y,
           output reg zero);
wire [31:0] S, Bout;
assign Bout = f[2] ? ~b : b;
assign S = a + Bout + f[2];

    always@(*)
    begin
        case(f[1:0])
            2'b00: y <= a & Bout;
            2'b01: y <= a | Bout;
            2'b10: y <= S;
            2'b11: y <= S[31];
        endcase
        if (y == 32'h00000000) zero = 1;
        else                   zero = 0;
    end
endmodule
```

datapath实现方法如下：

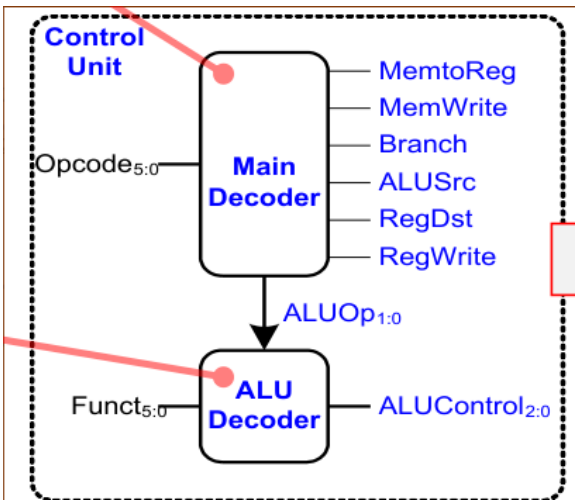
```
module datapath(
    input clk, reset,
    input pcen, irwrite,
    input regwrite,
    input alusrca, iord, memtoreg, regdst,
    input [2:0]  alusrcb, // ANDI
    input [1:0]  pcsrc,
```

```
    input [2:0]  alucontrol,
    input [1:0]  lb, // LB/LBU
    output [5:0]  op, funct,
    output zero,
    output [31:0] adr, writedata,
    input  [31:0] readdata
);
    // internal temperary data
    wire [4:0] writereg;
    wire [31:0] pcnext, pc;
    wire [31:0] instr, data, srca, srcb;
    wire [31:0] a;
    wire [31:0] aluresult, aluout;
    wire [31:0] signimm; // sign-extended immediate
    wire [31:0] zeroimm; // the zero-extended imm
    wire [31:0] signimmsh; // sign-extended immediate
                           // shifted left by 2
    wire [31:0] wd3, rd1, rd2;
    wire [31:0] memdata, membytezext, membytesext; // LB / LBU
    wire [7:0] membyte; // LB / LBU

    // op and funct
    assign op = instr[31:26];
    assign funct = instr[5:0];

    // datapath
    flopenr #(32) pcreg(clk, reset, pcen, pcnext, pc);
    mux2    #(32) adrmux(pc, aluout, iord, adr);
    flopenr #(32) instrreg(clk, reset, irwrite,
                           readdata, instr);
    // changes for LB / LBU
    flopr   #(32) datareg(clk, reset, memdata, data);
    mux4    #(8)  lbmux(readdata[31:24],
                        readdata[23:16], readdata[15:8],
                        readdata[7:0], aluout[1:0],
                        membyte);
    zeroext8_32   lbze(membyte, membytezext);
    signext8_32   lbse(membyte, membytesext);
    mux3    #(32) datamux(readdata, membytezext, membytesext,
                          lb, memdata);

    mux2    #(5)  regdstmux(instr[20:16],
                            instr[15:11], regdst, writereg);
    mux2    #(32) wdmux(aluout, data, memtoreg, wd3);
    regfile       rf(clk, regwrite, instr[25:21],
                     instr[20:16],
                     writereg, wd3, rd1, rd2);
    signext       se(instr[15:0], signimm);
    zeroext       ze(instr[15:0], zeroimm); // ANDI
    sl2           immsh(signimm, signimmsh);
    flopr   #(32) areg(clk, reset, rd1, a);
    flopr   #(32) breg(clk, reset, rd2, writedata);
    mux2    #(32) srcamux(pc, a, alusrca, srca);
    mux5    #(32) srcbmux(writedata, 32'b100,
                          signimm, signimmsh,
                          zeroimm, // ANDI
                          alusrcb, srcb);
    alu           alu(srca, srcb, alucontrol,aluresult, zero);
```

```
    flopr   #(32) alureg(clk, reset, aluresult, aluout);
    mux3    #(32) pcmux(aluresult, aluout,
                        {pc[31:28], instr[25:0], 2'b00},
                        pcsrc, pcnext);
endmodule
```

3.2 controller

控制器由两个译码器构成，通过`op`，`funct`，`zero`的值译码得出所有操作的所需要的值。



实现方法与之前类似，但是将整个执行过程变成了状态机过程，具体指令讲解在后面。

```
// controller

module controller
(
    input clk, reset,
    input [5:0] op, funct,
    input zero,
    output pcen, memwrite, irwrite, regwrite,
    output alusrca, iord, memtoreg, regdst,
    output [2:0] alusrcb,       // ANDI
    output [1:0] pcsrc,
    output [2:0] alucontrol,
    output [1:0] lb             // LB/LBU
);
wire [1:0] aluop;
wire branch,nbranch, pcwrite;

// Main Decoder and ALU Decoder
maindec md(clk,reset,op,pcwrite,memwrite,irwrite,regwrite,
               alusrca, branch, iord, memtoreg, regdst,
               alusrcb, pcsrc, aluop, nbranch, lb); // BNE LBU
aludec ad(funct, aluop, alucontrol);
assign pcen = pcwrite | (branch & zero) | (nbranch & ~zero); // BNE
endmodule


//maindecoder
module maindec
(
    input clk, reset,
    input [5:0] op,
```

```
    output pcwrite, memwrite, irwrite, regwrite,
    output alusrca, branch, iord, memtoreg, regdst,
    output [2:0] alusrcb,   // ANDI
    output [1:0] pcsrc,
    output [1:0] aluop,
    output bne,             // BNE
    output [1:0] lb         // LB/LBU
);

    parameter       FETCH=5'b00000, DECODE=5'b00001, MEMADR=5'b00010,
                    MEMRD=5'b00011, MEMWB=5'b00100,  MEMWR=5'b00101,
                    RTYPEEX=5'b00110, RTYPEWB=5'b00111, BEQEX=5'b01000,
                    ADDIEX=5'b01001, ADDIWB=5'b01010, JEX=5'b01011,
                    ANDIEX=5'b01100, ANDIWB=5'b01101,BNEEX=5'b01110,
                    LBURD=5'b01111, LBRD=5'b10000;

    reg [4:0] state, nextstate;
    reg [18:0] controls;

// different instructions
    parameter RTYPE = 6'b000000;
    parameter LW    = 6'b100011;
    parameter ADDI  = 6'b001000;
    parameter BNE   = 6'b000101;
    parameter LBU   = 6'b100100;
    parameter LB    = 6'b100000;
    parameter ANDI  = 6'b001100;
    parameter J     = 6'b000010;
    parameter SW    = 6'b101011;
    parameter BEQ   = 6'b000100;

    // state register
    always@(posedge clk or posedge reset)
        if (reset) state <= FETCH;
        else state <= nextstate;

    // next state logic
    always@(*)
    begin
        case(state)
            FETCH:   nextstate <= DECODE;
            DECODE: case(op)
                        LW:      nextstate <= MEMADR;
                        SW:      nextstate <= MEMADR;
                        LB:      nextstate <= MEMADR; // LB
                        LBU:     nextstate <= MEMADR; // LBU
                        RTYPE:   nextstate <= RTYPEEX;
                        BEQ:     nextstate <= BEQEX;
                        ADDI:    nextstate <= ADDIEX;
                        J:       nextstate <= JEX;
                        BNE:     nextstate <= BNEEX; // BNE
                        ANDI:    nextstate <= ADDIEX; // ANDI
                        default: nextstate <= FETCH;
                    endcase
            MEMADR: case(op)
                        LW:      nextstate <= MEMRD;
                        SW:      nextstate <= MEMWR;
                        LBU:     nextstate <= LBURD; // LBU
```

```
                        LB:      nextstate <= LBRD; // LB
                        default: nextstate <= FETCH;
                        // default should never happen
                    endcase
            MEMRD:   nextstate <= MEMWB;
            MEMWB:   nextstate <= FETCH;
            MEMWR:   nextstate <= FETCH;
            RTYPEEX: nextstate <= RTYPEWB;
            RTYPEWB: nextstate <= FETCH;
            BEQEX:   nextstate <= FETCH;
            ADDIEX:  nextstate <= ADDIWB;
            ADDIWB:  nextstate <= FETCH;
            JEX:     nextstate <= FETCH;
            ANDIEX:  nextstate <= ANDIWB; // ANDI
            ANDIWB:  nextstate <= FETCH; // ANDI
            BNEEX:   nextstate <= FETCH; // BNE
            LBURD:   nextstate <= MEMWB; // LBU
            LBRD:    nextstate <= MEMWB; // LB
            default: nextstate <= FETCH;
            // default should never happen
        endcase
    end

    // output logic
    assign {pcwrite, memwrite, irwrite, regwrite,
            alusrca, branch, iord, memtoreg, regdst,
            nbranch, // BNE
            alusrcb, pcsrc, aluop,
            lb} = controls; // LBU

    always@(*)
    begin
        case (state)
            FETCH:   controls <= 19'b1010_00000_0_00100_00_00;
            DECODE:  controls <= 19'b0000_00000_0_01100_00_00;
            MEMADR:  controls <= 19'b0000_10000_0_01000_00_00;
            MEMRD:   controls <= 19'b0000_00100_0_00000_00_00;
            MEMWB:   controls <= 19'b0001_00010_0_00000_00_00;
            MEMWR:   controls <= 19'b0100_00100_0_00000_00_00;
            RTYPEEX: controls <= 19'b0000_10000_0_00000_10_00;
            RTYPEWB: controls <= 19'b0001_00001_0_00000_00_00;
            BEQEX:   controls <= 19'b0000_11000_0_00001_01_00;
            ADDIEX:  controls <= 19'b0000_10000_0_01000_00_00;
            ADDIWB:  controls <= 19'b0001_00000_0_00000_00_00;
            JEX:     controls <= 19'b1000_00000_0_00010_00_00;
            ANDIEX:  controls <= 19'b0000_10000_0_10000_11_00; // ANDI
            ANDIWB:  controls <= 19'b0001_00000_0_00000_00_00; // ANDI
            BNEEX:   controls <= 19'b0000_10000_1_00001_01_00; // BNE
            LBURD:   controls <= 19'b0000_00100_0_00000_00_01; // LBU
            LBRD:    controls <= 19'b0000_00100_0_00000_00_10; // LB
            default: controls <= 19'b0000_xxxxx_x_xxxxx_xx_xx;
        endcase
    end

endmodule

module aludec(input [5:0] funct,
              input [1:0] aluop,
```

```
            output reg [2:0] alucontrol);
    always@(*)
    begin
        case(aluop)
            2'b00: alucontrol <= 3'b010; // add
            2'b01: alucontrol <= 3'b110; // sub
            default: case(funct) // RTYPE
                6'b100000: alucontrol <= 3'b010; // ADD
                6'b100010: alucontrol <= 3'b110; // SUB
                6'b100100: alucontrol <= 3'b000; // AND
                6'b100101: alucontrol <= 3'b001; // OR
                6'b101010: alucontrol <= 3'b111; // SLT
                default: alucontrol <= 3'bxxx; // ???
            endcase
        endcase
    end

endmodule
```

3.3 mem

将单周期中的dmem和imem合并成了mem。

```
module mem(input  clk, we,
           input  [31:0] a, wd,
           output [31:0] rd);

reg [31:0] RAM[63:0];

    initial
        begin
            $display("read data begin");
            $readmemh("D:/memfile.dat",RAM);
            $display("done");
//            $readmemh("testANDI.dat",RAM);
            // $readmemh("testBNE.dat",RAM);
        end
assign rd = RAM[a[31:2]]; // word aligned
always @(posedge clk)
    if (we)
        RAM[a[31:2]] <= wd;
endmodule
```

3.4 mips

除了传参有不同之外，基本与单周期类似。

```
module mips(
    input clk, reset,
    output [31:0] adr, writedata,
    output memwrite,
    input [31:0] readdata);

    wire        zero, pcen, irwrite, regwrite,
                alusrca, iord, memtoreg, regdst;
    wire [2:0] alusrcb; // ANDI
```

```
    wire [1:0] pcsrc;
    wire [2:0] alucontrol;
    wire [5:0] op, funct;
    wire [1:0] lb; // LB/LBU

    controller c(clk, reset, op, funct, zero,
                 pcen, memwrite, irwrite, regwrite,
                 alusrca, iord, memtoreg, regdst,
                 alusrcb, pcsrc, alucontrol, lb); // LB/LBU
    datapath dp(clk, reset,
                pcen, irwrite, regwrite,
                alusrca, iord, memtoreg, regdst,
                alusrcb, pcsrc, alucontrol,
                lb, // LB/LBU
                op, funct, zero,
                adr, writedata, readdata);
endmodule
```

3.5 top

top文件与单周期类似，去掉了dmem和imem，替换成了mem。

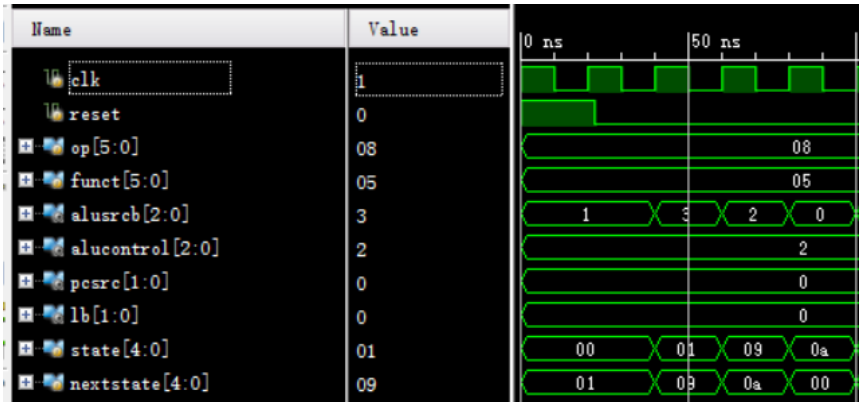
```
module top(
    input clk,reset,
    output [31:0] writedata,dataadr,
    output memwrite
    );
wire [31:0] instr,readdata;
mips mips(clk,reset,dataadr,writedata,memwrite,readdata);
mem mem(clk,memwrite,dataadr,writedata,readdata);
endmodule
```

4. 实验测试

按照软件开发的原理，所有的部件应该先模块化测试再进行整合测试。本次实验延续了这种分开再整合测试的原理，贯穿整个实验。首先`test_controller`的单元测试。最终使用`test_bench`进行整合测试。

4.1 test_controller

为了测试控制器是否正确产生有限状态机的正确转换，单独对controller进行测试。



- op = 6'b001000; funct = 6'b000101;
- 状态机转变为：00-01-09-0a，对应了正确的操作。

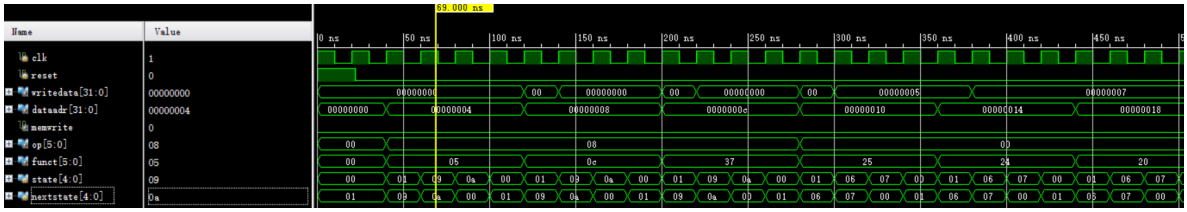
代码如下：

```
module test_controller();
reg clk;
reg reset;
reg [5:0] op,funct;
reg zero;
wire pcen, memwrite, irwrite, regwrite,
    alusrca, iord, memtoreg, regdst;
wire [2:0] alusrcb, alucontrol;
wire [1:0] pcsrc, lb;

initial
        begin
            reset<=1;
            #22;
            reset<=0;
        end

    // generate clk
    always
        begin
            clk<=1;#10;
            clk<=0;#10;
        end
// controller
cont(op,funct,zero,memtoreg,memwrite,pcsrc,alusrc,regdst,regwrite,jump,alucontro
l,immext);
controller cont(clk, reset, op, funct, zero,
                pcen, memwrite, irwrite, regwrite,
                alusrca, iord, memtoreg, regdst,
                alusrcb, pcsrc, alucontrol, lb);

initial begin
   op = 6'b001000;
   funct = 6'b000101;
end

endmodule
```

4.2 test_bench

在跑了test_controller之后就直接测试的test_bench因为之前的代码与上次类似，只有controller变化了很大。

- 目标：测试mips是否正确运转
- 结果：运转正确，仿真结果符合预期。



- 代码：

```
module testbench();
    reg clk;
    reg reset;

    wire [31:0] writedata,dataadr;
    wire memwrite;

    top dut(clk,reset,writedata,dataadr, memwrite);

    initial
        begin
            reset<=1;
            #22;
            reset<=0;
        end

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

    always @(negedge clk)
        begin
            if(memwrite) begin
                if(dataadr === 84 & writedata === 7) begin
                    $display("Simulation succeeded");
                    $stop;
                end
                else if (dataadr !== 80) begin
                    $display("Simulation failed");
                    $stop;
                end
            end
        end

endmodule
```

5. 指令

与上次单周期扩展指令的方法类似，也是在controller中修改对应的状态机变化。

- 添加状态机图，确定每个指令的状态流转过程。
- 必要的时候再添加新的部件设计加入mips多周期框架中

由于根据书上和ppt上的代码、上次单周期的代码基本能够跑通多周期，下面着重讲解重要的指令的原理。

5.1 bne指令

这个添加与上次类似，多加一个nbranch的判断语句。状态机跳转如branch

- alusrca = 1，说明rs为第一个参数
- alusrcb = 000，说明rt作为第二个参数
- nbranch = 1，说明为bne指令，当不相等的时候跳转

- pcsrc = 01，新的pc为aluout，作为跳转地址
- aluop = 01，说明计算操作数减法

bne指令需要在controller中逻辑添加：

```
assign pcen = pcwrite | (branch & zero) | (nbranch & ~zero); // BNE
```

5.2 andi指令

由于andi指令需要修改状态机，则添加ANDIEX=5'b01100, ANDIWB=5'b01101，即andi过程的执行过程和写回阶段。

- ANDIEX：
  - andi的执行阶段
  - alusrca = 1 ： 第一个参数为rs
  - alusrcb = 100：接受零扩展imm作为第二个参数
  - aluop = 11：alu的操作为and操作
- ANDIWB：
  - regwrite = 1：说明需要写回寄存器
  - memtoreg = 0：需要从aluout读取信息
  - regdst = 0：写回的寄存器为rt

同样为了操作满足，需要对srcbmux进行扩展：

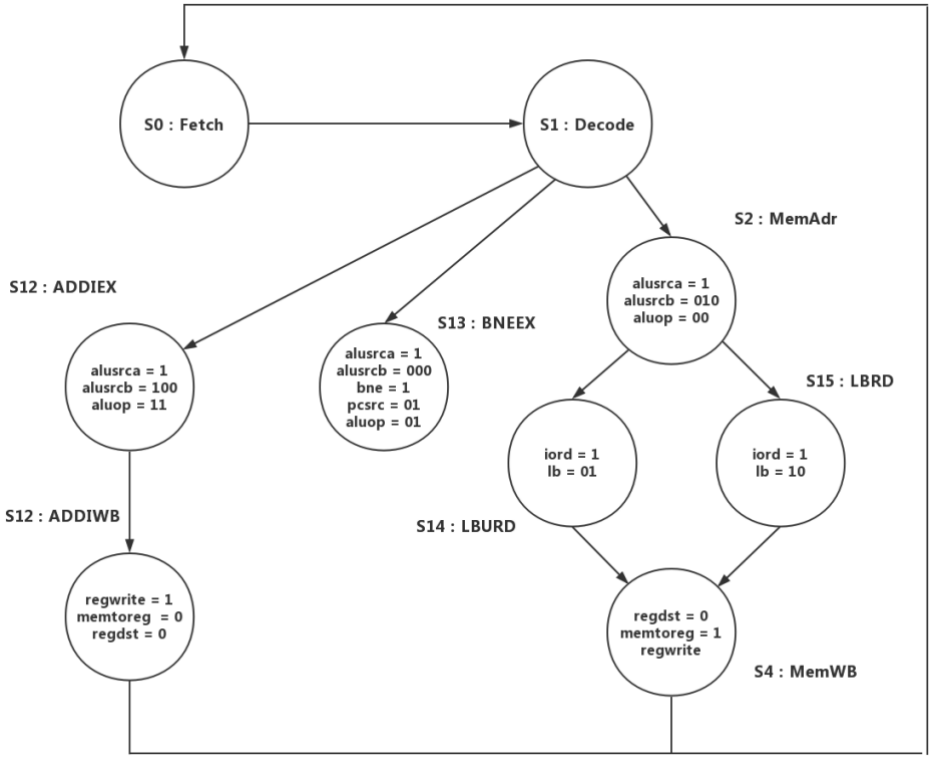
- 添加了新的符号扩展和零扩展

5.3 lbu/lb指令

尝试了添加单周期中没有实现的指令，同样需要扩展状态机图。

- lb指令：装载1字节数据
- lbu指令：装载无符号1字节数据
- 两者都类似于load指令，所以分别重新设置新的状态LBED和LBURD
- LBRD：
  - iord = 1：说明从存储器中读取数据
  - lb = 01：选择符号扩展
- LBRD：
  - iord = 1：说明从存储器读取数据
  - lb = 10： 选择无符号扩展（零扩展）

状态机图如下



6. 实验感想

本次实验实现了多周期mips框架，基于上次单周期的设计，主要对controller进行了大量修改，并且实现了几个基本指令和添加了几个指令，对原来实验进行了扩展。

本次实验也遇到了很多困难，比如在封装代码的时候，没有进行单元测试就直接跑test_bench结果出了很大的问题，花了半天时间才找到组件错误是一个实例化过程传参出错，浪费了很多时间。然后还是老老实实做了单元测试test_controller。然后实验中也添加了很多自己的注释和感想，对原来代码进行了很多扩展。

这是做的最后一个实验，也是本学期成就感比较足的实验，感谢！