南京理工大学计算机科学与工程学院

计算机系统结构实验报告

实验名称 单周期CPU上板验证实验

学生姓名 XX

学生学号

实验地点 1003

目录

1 实验目的

1. 理解 MIPS 指令结构，理解 MIPS 指令集中常用指令的功能和编码，学会对这些 指令进行归纳分类。
2. 了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。
3. 熟悉并掌握单周期 CPU 的原理和设计。
4. 进一步加强运用 verilog 语言进行电路设计的能力。
5. 为后续设计多周期 cpu 的实验打下基础。

2 实验设备

1. 装有 Xilinx Vivado 的计算机一台
2. LS-CPU-EXB-002 教学系统实验箱一套

3 实验内容

单周期 CPU 就是在 1 个 CPU 周期内执行完一条指令，执行指令所需的微操作由控 制器产生，结构简单，容易设计。指令系统可包括如下指令类型：

运算类指令：算术运算、逻辑运算、移位运算，如 ADD 、SUB、AND、NOT 、 SLL、SRL 等等。

传送类指令：寄存器之间的传送、立即数传送，如 MOVZ、MOVN、

MFHI 等等。

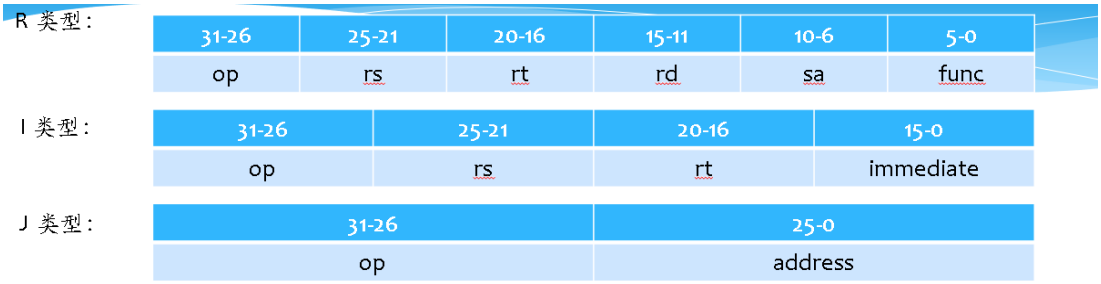
访存类指令：读存储器指令、写存储器指令，如 LB、LW、SB、SW 等等。

控制类指令：无条件转移、有条件转移，如 J、JR、JAL、BEQ 等等。

设计中所有寄存 器和存储器都是异步读同步写的，即读出数据不需要时钟控制，但写入数据需时钟控 制。 故单周期 CPU 的运作即：在一个时钟周期内，根据 PC 值从指令 ROM 中读出相应的 指令，将指令译码后从寄存器堆中读出需要的操作数，送往 ALU 模块，ALU 模块运算 得到结果。 如果是 store 指令，则 ALU 运算结果为数据存储的地址，就向数据 RAM 发出写请 求，在下一个时钟上升沿真正写入到数据存储器。 如果是 load 指令，则 ALU 运算结果为数据存储的地址，根据该值从数据存 RAM 中 读出数据，送往寄存器堆，根据目的寄存器发出写请求，在下一个时钟上升沿真正写入 到寄存器堆中。 如果非 load/store 操作，若有写寄存器堆的操作，则直接将 ALU 运算结果送往寄存器堆，根据目的寄存器发出写请求，在下一个时钟上升沿真正写入到寄存器堆中。 如果是分支跳转指令，则是需要将结果写入到 pc 寄存器中的。

本实验要求：根据提供的 32 位 MIPS 框架，至少实现 5 条指令（运算类、传送类、访存类、 控制转移类等）的上板验证。

指令格式：按 32 位 MIPSCPU 的指令格式(参考 32 位 MIPSCPU 指令集规范) 如：MIPS32 指令的三种格式，如下图：



其中

op：为操作码；

rs：为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt：为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

func：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

immediate：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Laod）/ 数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

address：为地址。

运算类指令

add rd，rs，rt （说明：以助记符表示，是汇编指令；以代码表示，是机器指令）

功能：rd←rs＋rt。reserved 为预留部分，即未用，一般填“0”。

000000	rs(5位)	rt(5位)	rd(5位)	reserved

传送类指令

move rd，rs　　功能：rd←rs 。

000001	rs(5位)	00000	rd(5位)	reserved

存储器读/写指令

sw rt，immediate(rs) 写存储器

功能：memory[rs+ (sign-extend)immediate]←rt；immediate 符号扩展再相加。

000010	rs(5位)	rt(5位)	immediate(16位)

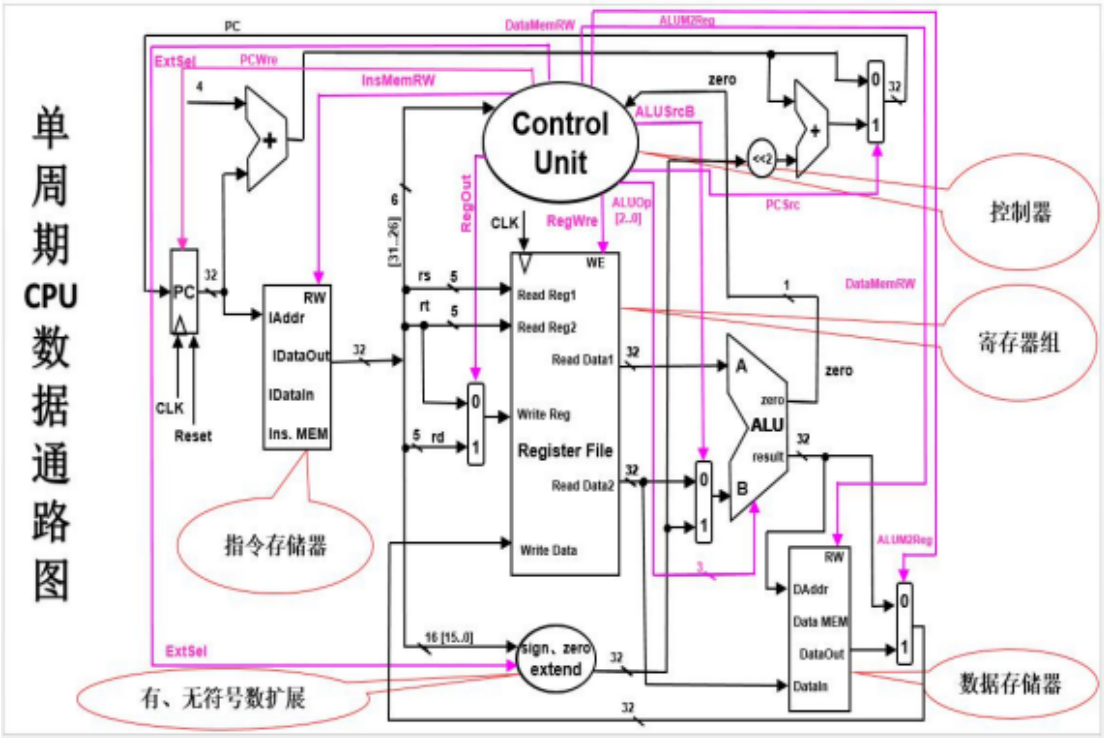
分支指令

beq rs,rt,immediate

功能：if(rs=rt) pc←pc + 4 + (sign-extend)immediate <<2；

000011	rs(5位)	rt(5位)	immediate(位移量，16位)

数据通路 ：单周期 32 位 MIPSCPU 的参考如下图：



4 实验程序

1. adder.v

```
`timescale 1ns / 1ps
//************************************************************************
// > 文件名: adder.v
// > 描述 ：加法器，直接使用"+"，会自动调用库里的加法器
//************************************************************************
module adder(
    input  [31:0] operand1,
    input  [31:0] operand2,
    input         cin,
    output [31:0] result,
    output        cout
    );
    assign {cout,result} = operand1 + operand2 + cin;

endmodule
```

2. alu.v

```
`timescale 1ns / 1ps
//************************************************************************
// > 文件名: alu.v
// > 描述 ：ALU 模块，可做 12 种操作
//************************************************************************
module alu(
    input  [11:0] alu_control, // ALU 控制信号
    input  [31:0] alu_src1,    // ALU 操作数 1,为补码
    input  [31:0] alu_src2,    // ALU 操作数 2，为补码
    output [31:0] alu_result   // ALU 结果
    );

    // ALU 控制信号，独热码
    wire alu_add;  //加法操作
    wire alu_sub;  //减法操作
    wire alu_slt;  //有符号比较，小于置位，复用加法器做减法
    wire alu_sltu; //无符号比较，小于置位，复用加法器做减法
    wire alu_and;  //按位与
    wire alu_nor;  //按位或非
```

```
    wire alu_or;    //按位或
    wire alu_xor;   //按位异或
    wire alu_sll;   //逻辑左移
    wire alu_srl;   //逻辑右移
    wire alu_sra;   //算术右移
    wire alu_lui;   //高位加载

    assign alu_add  = alu_control[11];
    assign alu_sub  = alu_control[10];
    assign alu_slt  = alu_control[ 9];
    assign alu_sltu = alu_control[ 8];
    assign alu_and  = alu_control[ 7];
    assign alu_nor  = alu_control[ 6];
    assign alu_or   = alu_control[ 5];
    assign alu_xor  = alu_control[ 4];
    assign alu_sll  = alu_control[ 3];
    assign alu_srl  = alu_control[ 2];
    assign alu_sra  = alu_control[ 1];
    assign alu_lui  = alu_control[ 0];

    wire [31:0] add_sub_result;
    wire [31:0] slt_result;
    wire [31:0] sltu_result;
    wire [31:0] and_result;
    wire [31:0] nor_result;
    wire [31:0] or_result;
    wire [31:0] xor_result;
    wire [31:0] sll_result;
    wire [31:0] srl_result;
    wire [31:0] sra_result;
    wire [31:0] lui_result;

    assign and_result = alu_src1 & alu_src2;     // 与结果为两数按位与
    assign or_result  = alu_src1 | alu_src2;     // 或结果为两数按位或
    assign nor_result = ~or_result;              // 或非结果为或结果按位取反
    assign xor_result = alu_src1 ^ alu_src2;     // 异或结果为两数按位异或
    assign lui_result = {alu_src2[15:0], 16'd0}; // 立即数装载结果为立即数移位至高半字
节

//-----{加法器}begin
//add,sub,slt,sltu 均使用该模块
    wire [31:0] adder_operand1;
    wire [31:0] adder_operand2;
    wire        adder_cin     ;
```

```
    wire [31:0] adder_result  ;
    wire        adder_cout    ;
    assign adder_operand1 = alu_src1;
    assign adder_operand2 = alu_add ? alu_src2 : ~alu_src2;
    assign adder_cin      = ~alu_add; //减法需要 cin
    adder adder_module(
    .operand1(adder_operand1),
    .operand2(adder_operand2),
    .cin     (adder_cin     ),
    .result  (adder_result  ),
    .cout    (adder_cout    )
    );

    //加减结果
    assign add_sub_result = adder_result;

    //slt 结果
    //adder_src1[31] adder_src2[31] adder_result[31]
    //      0          1          X(0 或 1)      "正-负"，显然小于不成立
    //      0          0           1          相减为负，说明小于
    //      0          0           0          相减为正，说明不小于
    //      1          1           1          相减为负，说明小于
    //      1          1           0          相减为正，说明不小于
    //      1          0          X(0 或 1)      "负-正"，显然小于成立
    assign slt_result[31:1] = 31'd0;
    assign slt_result[0]    = (alu_src1[31] & ~alu_src2[31]) | (~(alu_src1[31]^alu_src2[31]) &
adder_result[31]);

    //sltu 结果
    //对于 32 位无符号数比较，相当于 33 位有符号数（{1'b0,src1}和{1'b0,src2}）的比较，
最高位 0 为符号位
    //故，可以用 33 位加法器来比较大小，需要对{1'b0,src2}取反,即需要
{1'b0,src1}+{1'b1,~src2}+cin
    //但此处用的为 32 位加法器，只做了运算:                    src1  +  ~src2  +cin
    //32 位加法的结果为{adder_cout,adder_result},则 33 位加法结果应该为
{adder_cout+1'b1,adder_result}
    //对比 slt 结果注释，知道，此时判断大小属于第二三种情况，即源操作数 1 符号位
为 0，源操作数 2 符号位为 0
    //结果的符号位为 1，说明小于，即 adder_cout+1'b1 为 2'b01，即 adder_cout 为 0
    assign sltu_result = {31'd0, ~adder_cout};
//-----{加法器}end

//-----{移位器}begin
    // 移位分三步进行，
```

```
  // 第一步根据移位量低 2 位即[1:0]位做第一次移位，
  // 第二步在第一次移位基础上根据移位量[3:2]位做第二次移位，
  // 第三步在第二次移位基础上根据移位量[4]位做第三次移位。
  wire [4:0] shf;
  assign shf = alu_src1[4:0];
  wire [1:0] shf_1_0;
  wire [1:0] shf_3_2;
  assign shf_1_0 = shf[1:0];
  assign shf_3_2 = shf[3:2];

  // 逻辑左移
  wire [31:0] sll_step1;
  wire [31:0] sll_step2;
  assign sll_step1 = {32{shf_1_0 == 2'b00}} & alu_src2             // 若 shf[1:0]="00",不
移位
                   | {32{shf_1_0 == 2'b01}} & {alu_src2[30:0], 1'd0}    // 若 shf[1:0]="01",左移
1 位
                   | {32{shf_1_0 == 2'b10}} & {alu_src2[29:0], 2'd0}    // 若 shf[1:0]="10",左移
2 位
                   | {32{shf_1_0 == 2'b11}} & {alu_src2[28:0], 3'd0};   // 若 shf[1:0]="11",左移
3 位
  assign sll_step2 = {32{shf_3_2 == 2'b00}} & sll_step1            // 若 shf[3:2]="00",不
移位
                   | {32{shf_3_2 == 2'b01}} & {sll_step1[27:0], 4'd0}   // 若 shf[3:2]="01",第一
次移位结果左移 4 位
                   | {32{shf_3_2 == 2'b10}} & {sll_step1[23:0], 8'd0}   // 若 shf[3:2]="10",第一
次移位结果左移 8 位
                   | {32{shf_3_2 == 2'b11}} & {sll_step1[19:0], 12'd0}; // 若 shf[3:2]="11",第一
次移位结果左移 12 位
  assign sll_result = shf[4] ? {sll_step2[15:0], 16'd0} : sll_step2;    // 若 shf[4]="1",第二次
移位结果左移 16 位

  // 逻辑右移
  wire [31:0] srl_step1;
  wire [31:0] srl_step2;
  assign srl_step1 = {32{shf_1_0 == 2'b00}} & alu_src2             // 若 shf[1:0]="00",不
移位
                   | {32{shf_1_0 == 2'b01}} & {1'd0, alu_src2[31:1]}    // 若 shf[1:0]="01",右移
1 位,高位补 0
                   | {32{shf_1_0 == 2'b10}} & {2'd0, alu_src2[31:2]}    // 若 shf[1:0]="10",右移
2 位,高位补 0
                   | {32{shf_1_0 == 2'b11}} & {3'd0, alu_src2[31:3]};   // 若 shf[1:0]="11",右移
3 位,高位补 0
  assign srl_step2 = {32{shf_3_2 == 2'b00}} & srl_step1            // 若 shf[3:2]="00",不
```

```
移位
                | {32{shf_3_2 == 2'b01}} & {4'd0, srl_step1[31:4]}   // 若 shf[3:2]="01",第一次移位结果右移 4 位,高位补 0
                | {32{shf_3_2 == 2'b10}} & {8'd0, srl_step1[31:8]}   // 若 shf[3:2]="10",第一次移位结果右移 8 位,高位补 0
                | {32{shf_3_2 == 2'b11}} & {12'd0, srl_step1[31:12]}; // 若 shf[3:2]="11",第一次移位结果右移 12 位,高位补 0
  assign srl_result = shf[4] ? {16'd0, srl_step2[31:16]} : srl_step2;   // 若 shf[4]="1",第二次移位结果右移 16 位,高位补 0

  // 算术右移
  wire [31:0] sra_step1;
  wire [31:0] sra_step2;
  assign sra_step1 = {32{shf_1_0 == 2'b00}} & alu_src2                    // 若 shf[1:0]="00",不移位
                | {32{shf_1_0 == 2'b01}} & {alu_src2[31], alu_src2[31:1]}       // 若 shf[1:0]="01",右移 1 位,高位补符号位
                | {32{shf_1_0 == 2'b10}} & {{2{alu_src2[31]}}, alu_src2[31:2]}    // 若 shf[1:0]="10",右移 2 位,高位补符号位
                | {32{shf_1_0 == 2'b11}} & {{3{alu_src2[31]}}, alu_src2[31:3]};   // 若 shf[1:0]="11",右移 3 位,高位补符号位
  assign sra_step2 = {32{shf_3_2 == 2'b00}} & sra_step1                   // 若 shf[3:2]="00",不移位
                | {32{shf_3_2 == 2'b01}} & {{4{sra_step1[31]}}, sra_step1[31:4]}  // 若 shf[3:2]="01",第一次移位结果右移 4 位,高位补符号位
                | {32{shf_3_2 == 2'b10}} & {{8{sra_step1[31]}}, sra_step1[31:8]}  // 若 shf[3:2]="10",第一次移位结果右移 8 位,高位补符号位
                | {32{shf_3_2 == 2'b11}} & {{12{sra_step1[31]}}, sra_step1[31:12]}; // 若 shf[3:2]="11",第一次移位结果右移 12 位,高位补符号位
  assign sra_result = shf[4] ? {{16{sra_step2[31]}}, sra_step2[31:16]} : sra_step2;   // 若 shf[4]="1",第二次移位结果右移 16 位,高位补符号位
//-----{移位器}end

  // 选择相应结果输出
  assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
                      alu_slt      ? slt_result :
                      alu_sltu     ? sltu_result :
                      alu_and      ? and_result :
                      alu_nor      ? nor_result :
                      alu_or       ? or_result  :
                      alu_xor      ? xor_result :
                      alu_sll      ? sll_result :
                      alu_srl      ? srl_result :
                      alu_sra      ? sra_result :
```

```
                alu_lui         ? lui_result :
                32'd0;
endmodule
```

3. data_ram.v

```
`timescale 1ns / 1ps
//*************************************************************************
// > 文件名: data_mem.v
// > 描述 ：异步数据存储器模块，采用寄存器搭建而成，类似寄存器堆
// >       同步写，异步读
//*************************************************************************
module data_ram(
    input         clk,        // 时钟
    input  [3:0]  wen,        // 字节写使能
    input  [4:0] addr,        // 地址
    input  [31:0] wdata,      // 写数据
    output reg [31:0] rdata,      // 读数据

    //调试端口，用于读出数据显示
    input  [4 :0] test_addr,
    output reg [31:0] test_data
);
    reg [31:0] DM[31:0]; //数据存储器，字节地址 7'b000_0000~7'b111_1111

    //写数据
    always @(posedge clk)   // 当写控制信号为 1，数据写入内存
    begin
        if (wen[3])
        begin
            DM[addr][31:24] <= wdata[31:24];
        end
    end
    always @(posedge clk)
    begin
        if (wen[2])
        begin
            DM[addr][23:16] <= wdata[23:16];
        end
    end
    always @(posedge clk)
    begin
        if (wen[1])
        begin
            DM[addr][15: 8] <= wdata[15: 8];
```

```
    end
end
always @(posedge clk)
begin
    if (wen[0])
    begin
        DM[addr][7 : 0] <= wdata[7 : 0];
    end
end

//读数据,取 4 字节
always @(*)
begin
    case (addr)
        5'd0 : rdata <= DM[0 ];
        5'd1 : rdata <= DM[1 ];
        5'd2 : rdata <= DM[2 ];
        5'd3 : rdata <= DM[3 ];
        5'd4 : rdata <= DM[4 ];
        5'd5 : rdata <= DM[5 ];
        5'd6 : rdata <= DM[6 ];
        5'd7 : rdata <= DM[7 ];
        5'd8 : rdata <= DM[8 ];
        5'd9 : rdata <= DM[9 ];
        5'd10: rdata <= DM[10];
        5'd11: rdata <= DM[11];
        5'd12: rdata <= DM[12];
        5'd13: rdata <= DM[13];
        5'd14: rdata <= DM[14];
        5'd15: rdata <= DM[15];
        5'd16: rdata <= DM[16];
        5'd17: rdata <= DM[17];
        5'd18: rdata <= DM[18];
        5'd19: rdata <= DM[19];
        5'd20: rdata <= DM[20];
        5'd21: rdata <= DM[21];
        5'd22: rdata <= DM[22];
        5'd23: rdata <= DM[23];
        5'd24: rdata <= DM[24];
        5'd25: rdata <= DM[25];
        5'd26: rdata <= DM[26];
        5'd27: rdata <= DM[27];
        5'd28: rdata <= DM[28];
        5'd29: rdata <= DM[29];
```

```
        5'd30: rdata <= DM[30];
        5'd31: rdata <= DM[31];
    endcase
  end
  //调试端口，读出特定内存的数据
  always @(*)
  begin
    case (test_addr)
      5'd0 : test_data <= DM[0 ];
      5'd1 : test_data <= DM[1 ];
      5'd2 : test_data <= DM[2 ];
      5'd3 : test_data <= DM[3 ];
      5'd4 : test_data <= DM[4 ];
      5'd5 : test_data <= DM[5 ];
      5'd6 : test_data <= DM[6 ];
      5'd7 : test_data <= DM[7 ];
      5'd8 : test_data <= DM[8 ];
      5'd9 : test_data <= DM[9 ];
      5'd10: test_data <= DM[10];
      5'd11: test_data <= DM[11];
      5'd12: test_data <= DM[12];
      5'd13: test_data <= DM[13];
      5'd14: test_data <= DM[14];
      5'd15: test_data <= DM[15];
      5'd16: test_data <= DM[16];
      5'd17: test_data <= DM[17];
      5'd18: test_data <= DM[18];
      5'd19: test_data <= DM[19];
      5'd20: test_data <= DM[20];
      5'd21: test_data <= DM[21];
      5'd22: test_data <= DM[22];
      5'd23: test_data <= DM[23];
      5'd24: test_data <= DM[24];
      5'd25: test_data <= DM[25];
      5'd26: test_data <= DM[26];
      5'd27: test_data <= DM[27];
      5'd28: test_data <= DM[28];
      5'd29: test_data <= DM[29];
      5'd30: test_data <= DM[30];
      5'd31: test_data <= DM[31];
    endcase
  end
endmodule
```

4. inst_rom.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// > 文件名: inst_rom.v
// > 描述 ：异步指令存储器模块，采用寄存器搭建而成，类似寄存器堆
// >       内嵌好指令，只读，异步读
//////////////////////////////////////////////////////////////////////////////////
module inst_rom(
    input      [4 :0] addr, // 指令地址
    output reg [31:0] inst      // 指令
    );

    wire [31:0] inst_rom[19:0]; // 指令存储器，字节地址 7'b000_0000~7'b111_1111
    //------------ 指令编码 ---------|指令地址|--- 汇编指令 -----|- 指令结果 -----//
    assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1   | $1 = 0000_0001H
    assign inst_rom[ 1] = 32'h00011100; // 04H: sll   $2 ,$1,#4   | $2 = 0000_0010H
    assign inst_rom[ 2] = 32'h00411821; // 08H: addu  $3 ,$2,$1   | $3 = 0000_0011H
    assign inst_rom[ 3] = 32'h00022082; // 0CH: srl   $4 ,$2,#2   | $4 = 0000_0004H
    assign inst_rom[ 4] = 32'h00642823; // 10H: subu  $5 ,$3,$4   | $5 = 0000_000DH
    assign inst_rom[ 5] = 32'hAC250013; // 14H: sw    $5 ,#19($1) | Mem[0000_0014H] =
0000_000DH
    assign inst_rom[ 6] = 32'h00A23027; // 18H: nor   $6 ,$5,$2   | $6 = FFFF_FFE2H
    assign inst_rom[ 7] = 32'h00C33825; // 1CH: or    $7 ,$6,$3   | $7 = FFFF_FFF3H
    assign inst_rom[ 8] = 32'h00E64026; // 20H: xor   $8 ,$7,$6   | $8 = 0000_0011H
    assign inst_rom[ 9] = 32'hAC08001C; // 24H: sw    $8 ,#28($0) | Mem[0000_001CH] =
0000_0011H
    assign inst_rom[10] = 32'h00C7482A; // 28H: slt   $9 ,$6,$7   | $9 = 0000_0001H
    assign inst_rom[11] = 32'h11210002; // 2CH: beq   $9 ,$1,#2   | 跳转到指令 34H
    assign inst_rom[12] = 32'h24010004; // 30H: addiu $1 ,$0,#4   | 不执行
    assign inst_rom[13] = 32'h8C2A0013; // 34H: lw    $10,#19($1) | $10 = 0000_000DH
    assign inst_rom[14] = 32'h15450003; // 38H: bne   $10,$5,#3   | 不跳转
    assign inst_rom[15] = 32'h00415824; // 3CH: and   $11,$2,$1   | $11 = 0000_0000H
    assign inst_rom[16] = 32'hAC0B001C; // 40H: sw    $11,#28($0) | Men[0000_001CH] =
0000_0000H
    assign inst_rom[17] = 32'hAC040010; // 44H: sw    $4 ,#16($0) | Mem[0000_0010H] =
0000_0004H
    assign inst_rom[18] = 32'h3C0C000C; // 48H: lui   $12,#12     | [R12] = 000C_0000H
    assign inst_rom[19] = 32'h08000000; // 4CH: j     00H         | 跳转指令 00H
    //读指令,取 4 字节
    always @(*)
    begin
        case (addr)
            5'd0 : inst <= inst_rom[0 ];
            5'd1 : inst <= inst_rom[1 ];
            5'd2 : inst <= inst_rom[2 ];
```

```
            5'd3 : inst <= inst_rom[3 ];
            5'd4 : inst <= inst_rom[4 ];
            5'd5 : inst <= inst_rom[5 ];
            5'd6 : inst <= inst_rom[6 ];
            5'd7 : inst <= inst_rom[7 ];
            5'd8 : inst <= inst_rom[8 ];
            5'd9 : inst <= inst_rom[9 ];
            5'd10: inst <= inst_rom[10];
            5'd11: inst <= inst_rom[11];
            5'd12: inst <= inst_rom[12];
            5'd13: inst <= inst_rom[13];
            5'd14: inst <= inst_rom[14];
            5'd15: inst <= inst_rom[15];
            5'd16: inst <= inst_rom[16];
            5'd17: inst <= inst_rom[17];
            5'd18: inst <= inst_rom[18];
            5'd19: inst <= inst_rom[19];
            default: inst <= 32'd0;
        endcase
    end
endmodule
```

5. regfile.v

```
`timescale 1ns / 1ps
//************************************************************************
// > 文件名: regfile.v
// > 描述 ：寄存器堆模块，同步写，异步读
//************************************************************************
module regfile(
    input           clk,
    input           wen,
    input     [4 :0] raddr1,
    input     [4 :0] raddr2,
    input     [4 :0] waddr,
    input     [31:0] wdata,
    output reg [31:0] rdata1,
    output reg [31:0] rdata2,
    input     [4 :0] test_addr,
    output reg [31:0] test_data
    );
    reg [31:0] rf[31:0];

    // three ported register file
    // read two ports combinationally
```

```
// write third port on rising edge of clock
// register 0 hardwired to 0

always @(posedge clk)
begin
   if (wen)
   begin
      rf[waddr] <= wdata;
   end
end
//读端口 1
always @(*)
begin
   case (raddr1)
      5'd1 : rdata1 <= rf[1 ];
      5'd2 : rdata1 <= rf[2 ];
      5'd3 : rdata1 <= rf[3 ];
      5'd4 : rdata1 <= rf[4 ];
      5'd5 : rdata1 <= rf[5 ];
      5'd6 : rdata1 <= rf[6 ];
      5'd7 : rdata1 <= rf[7 ];
      5'd8 : rdata1 <= rf[8 ];
      5'd9 : rdata1 <= rf[9 ];
      5'd10: rdata1 <= rf[10];
      5'd11: rdata1 <= rf[11];
      5'd12: rdata1 <= rf[12];
      5'd13: rdata1 <= rf[13];
      5'd14: rdata1 <= rf[14];
      5'd15: rdata1 <= rf[15];
      5'd16: rdata1 <= rf[16];
      5'd17: rdata1 <= rf[17];
      5'd18: rdata1 <= rf[18];
      5'd19: rdata1 <= rf[19];
      5'd20: rdata1 <= rf[20];
      5'd21: rdata1 <= rf[21];
      5'd22: rdata1 <= rf[22];
      5'd23: rdata1 <= rf[23];
      5'd24: rdata1 <= rf[24];
      5'd25: rdata1 <= rf[25];
      5'd26: rdata1 <= rf[26];
      5'd27: rdata1 <= rf[27];
      5'd28: rdata1 <= rf[28];
      5'd29: rdata1 <= rf[29];
      5'd30: rdata1 <= rf[30];
```

```
        5'd31: rdata1 <= rf[31];
        default : rdata1 <= 32'd0;
    endcase
end
//读端口 2
always @(*)
begin
    case (raddr2)
        5'd1 : rdata2 <= rf[1 ];
        5'd2 : rdata2 <= rf[2 ];
        5'd3 : rdata2 <= rf[3 ];
        5'd4 : rdata2 <= rf[4 ];
        5'd5 : rdata2 <= rf[5 ];
        5'd6 : rdata2 <= rf[6 ];
        5'd7 : rdata2 <= rf[7 ];
        5'd8 : rdata2 <= rf[8 ];
        5'd9 : rdata2 <= rf[9 ];
        5'd10: rdata2 <= rf[10];
        5'd11: rdata2 <= rf[11];
        5'd12: rdata2 <= rf[12];
        5'd13: rdata2 <= rf[13];
        5'd14: rdata2 <= rf[14];
        5'd15: rdata2 <= rf[15];
        5'd16: rdata2 <= rf[16];
        5'd17: rdata2 <= rf[17];
        5'd18: rdata2 <= rf[18];
        5'd19: rdata2 <= rf[19];
        5'd20: rdata2 <= rf[20];
        5'd21: rdata2 <= rf[21];
        5'd22: rdata2 <= rf[22];
        5'd23: rdata2 <= rf[23];
        5'd24: rdata2 <= rf[24];
        5'd25: rdata2 <= rf[25];
        5'd26: rdata2 <= rf[26];
        5'd27: rdata2 <= rf[27];
        5'd28: rdata2 <= rf[28];
        5'd29: rdata2 <= rf[29];
        5'd30: rdata2 <= rf[30];
        5'd31: rdata2 <= rf[31];
        default : rdata2 <= 32'd0;
    endcase
end
 //调试端口，读出寄存器值显示在触摸屏上
always @(*)
```

```
    begin
        case (test_addr)
            5'd1 : test_data <= rf[1 ];
            5'd2 : test_data <= rf[2 ];
            5'd3 : test_data <= rf[3 ];
            5'd4 : test_data <= rf[4 ];
            5'd5 : test_data <= rf[5 ];
            5'd6 : test_data <= rf[6 ];
            5'd7 : test_data <= rf[7 ];
            5'd8 : test_data <= rf[8 ];
            5'd9 : test_data <= rf[9 ];
            5'd10: test_data <= rf[10];
            5'd11: test_data <= rf[11];
            5'd12: test_data <= rf[12];
            5'd13: test_data <= rf[13];
            5'd14: test_data <= rf[14];
            5'd15: test_data <= rf[15];
            5'd16: test_data <= rf[16];
            5'd17: test_data <= rf[17];
            5'd18: test_data <= rf[18];
            5'd19: test_data <= rf[19];
            5'd20: test_data <= rf[20];
            5'd21: test_data <= rf[21];
            5'd22: test_data <= rf[22];
            5'd23: test_data <= rf[23];
            5'd24: test_data <= rf[24];
            5'd25: test_data <= rf[25];
            5'd26: test_data <= rf[26];
            5'd27: test_data <= rf[27];
            5'd28: test_data <= rf[28];
            5'd29: test_data <= rf[29];
            5'd30: test_data <= rf[30];
            5'd31: test_data <= rf[31];
            default : test_data <= 32'd0;
        endcase
    end
endmodule
```

6. single_cycle_cpu.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
```

```
// > 文件名: single_cycle_cpu.v
// > 描述 :单周期 CPU 模块，共实现 16 条指令
// >      指令 rom 和数据 ram 均采用异步读数据，以便单周期 CPU 好实现
//*************************************************************************
`define STARTADDR 32'd0 // 程序起始地址
module single_cycle_cpu(
    input clk,   // 时钟
    input resetn, // 复位信号，低电平有效

    //display data
    input  [ 4:0] rf_addr,
    input  [31:0] mem_addr,
    output [31:0] rf_data,
    output [31:0] mem_data,
    output [31:0] cpu_pc,
    output [31:0] cpu_inst
    );

//--------------------------------{取指}begin-----------------------------------//
    reg  [31:0] pc;
    wire [31:0] next_pc;
    wire [31:0] seq_pc;
    wire [31:0] jbr_target;
    wire jbr_taken;

    // 下一指令地址：seq_pc=pc+4
    assign seq_pc[31:2]    = pc[31:2] + 1'b1;
    assign seq_pc[1:0]     = pc[1:0];
    // 新指令：若指令跳转，为跳转地址；否则为下一指令
    assign next_pc = jbr_taken ? jbr_target : seq_pc;
    always @ (posedge clk)   // PC 程序计数器
    begin
        if (!resetn) begin
            pc <= `STARTADDR; // 复位，取程序起始地址
        end
        else begin
            pc <= next_pc;   // 不复位，取新指令
        end
    end

    wire [31:0] inst_addr;
    wire [31:0] inst;
    assign inst_addr = pc; // 指令地址：指令长度 32 位
    inst_rom inst_rom_module(       // 指令存储器
```

```
        .addr     (inst_addr[6:2]),  // I, 5,指令地址
        .inst     (inst         )   // O, 32,指令
    );
    assign cpu_pc = pc;       //display pc
    assign cpu_inst = inst;
//---------------------------------{取指}end---------------------------------//

//---------------------------------{译码}begin---------------------------------//
    wire [5:0] op;
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [4:0] sa;
    wire [5:0] funct;
    wire [15:0] imm;
    wire [15:0] offset;
    wire [25:0] target;

    assign op     = inst[31:26];  // 操作码
    assign rs     = inst[25:21];  // 源操作数 1
    assign rt     = inst[20:16];  // 源操作数 2
    assign rd     = inst[15:11];  // 目标操作数
    assign sa     = inst[10:6];   // 特殊域，可能存放偏移量
    assign funct  = inst[5:0];    // 功能码
    assign imm    = inst[15:0];   // 立即数
    assign offset = inst[15:0];   // 地址偏移量
    assign target = inst[25:0];   // 目标地址

    wire op_zero;  // 操作码全 0
    wire sa_zero;  // sa 域全 0
    assign op_zero = ~(|op);
    assign sa_zero = ~(|sa);

    // 实现指令列表
    wire inst_ADDU, inst_SUBU , inst_SLT, inst_AND;
    wire inst_NOR , inst_OR   , inst_XOR, inst_SLL;
    wire inst_SRL , inst_ADDIU, inst_BEQ, inst_BNE;
    wire inst_LW  , inst_SW   , inst_LUI, inst_J;

    assign inst_ADDU  = op_zero & sa_zero    & (funct == 6'b100001);// 无符号加法
    assign inst_SUBU  = op_zero & sa_zero    & (funct == 6'b100011);// 无符号减法
    assign inst_SLT   = op_zero & sa_zero    & (funct == 6'b101010);// 小于则置位
    assign inst_AND   = op_zero & sa_zero    & (funct == 6'b100100);// 逻辑与运算
    assign inst_NOR   = op_zero & sa_zero    & (funct == 6'b100111);// 逻辑或非运算
```

```
assign inst_OR    = op_zero & sa_zero    & (funct == 6'b100101);// 逻辑或运算
assign inst_XOR   = op_zero & sa_zero    & (funct == 6'b100110);// 逻辑异或运算
assign inst_SLL   = op_zero & (rs==5'd0) & (funct == 6'b000000);// 逻辑左移
assign inst_SRL   = op_zero & (rs==5'd0) & (funct == 6'b000010);// 逻辑右移
assign inst_ADDIU = (op == 6'b001001);              // 立即数无符号加法
assign inst_BEQ   = (op == 6'b000100);              // 判断相等跳转
assign inst_BNE   = (op == 6'b000101);              // 判断不等跳转
assign inst_LW    = (op == 6'b100011);              // 从内存装载
assign inst_SW    = (op == 6'b101011);              // 向内存存储
assign inst_LUI   = (op == 6'b001111);              // 立即数装载高半字节
assign inst_J     = (op == 6'b000010);              // 直接跳转

// 无条件跳转判断
wire        j_taken;
wire [31:0] j_target;
assign j_taken  = inst_J;
// 无条件跳转目标地址：PC={PC[31:28],target<<2}
assign j_target = {pc[31:28], target, 2'b00};

//分支跳转
wire        beq_taken;
wire        bne_taken;
wire [31:0] br_target;
assign beq_taken = (rs_value == rt_value);      // BEQ 跳转条件：GPR[rs]=GPR[rt]
assign bne_taken = ~beq_taken;                  // BNE 跳转条件：GPR[rs]≠GPR[rt]
assign br_target[31:2] = pc[31:2] + {{14{offset[15]}}, offset};
assign br_target[1:0]  = pc[1:0];   // 分支跳转目标地址：PC=PC+offset<<2

//跳转指令的跳转信号和跳转目标地址
assign jbr_taken = j_taken          // 指令跳转：无条件跳转 或 满足分支跳转条件
                 | inst_BEQ & beq_taken
                 | inst_BNE & bne_taken;
assign jbr_target = j_taken ? j_target : br_target;

// 寄存器堆
wire rf_wen;
wire [4:0] rf_waddr;
wire [31:0] rf_wdata;
wire [31:0] rs_value, rt_value;

regfile rf_module(
    .clk    (clk      ), // I, 1
    .wen    (rf_wen   ), // I, 1
    .raddr1 (rs       ), // I, 5
```

```
    .raddr2 (rt      ),  // I, 5
    .waddr  (rf_waddr ),  // I, 5
    .wdata  (rf_wdata ),  // I, 32
    .rdata1 (rs_value ),  // O, 32
    .rdata2 (rt_value ),   // O, 32

    //display rf
    .test_addr(rf_addr),
    .test_data(rf_data)
);


// 传递到执行模块的 ALU 源操作数和操作码
wire inst_add, inst_sub, inst_slt,inst_sltu;
wire inst_and, inst_nor, inst_or, inst_xor;
wire inst_sll, inst_srl, inst_sra,inst_lui;
assign inst_add = inst_ADDU | inst_ADDIU | inst_LW | inst_SW; // 做加法运算指令
assign inst_sub = inst_SUBU; // 减法
assign inst_slt = inst_SLT;  // 小于置位
assign inst_sltu= 1'b0;      // 暂未实现
assign inst_and = inst_AND;  // 逻辑与
assign inst_nor = inst_NOR;  // 逻辑或非
assign inst_or  = inst_OR;   // 逻辑或
assign inst_xor = inst_XOR;  // 逻辑异或
assign inst_sll = inst_SLL;  // 逻辑左移
assign inst_srl = inst_SRL;  // 逻辑右移
assign inst_sra = 1'b0;      // 暂未实现
assign inst_lui = inst_LUI;  // 立即数装载高位

wire [31:0] sext_imm;
wire   inst_shf_sa;   //使用 sa 域作为偏移量的指令
wire   inst_imm_sign;  //对立即数作符号扩展的指令
assign sext_imm      = {{16{imm[15]}}, imm};// 立即数符号扩展
assign inst_shf_sa   = inst_SLL | inst_SRL;
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW;

wire [31:0] alu_operand1;
wire [31:0] alu_operand2;
wire [11:0] alu_control;
assign alu_operand1 = inst_shf_sa ? {27'd0,sa} : rs_value;
assign alu_operand2 = inst_imm_sign ? sext_imm : rt_value;
assign alu_control = {inst_add,       // ALU 操作码，独热编码
                      inst_sub,
                      inst_slt,
```

```
                    inst_sltu,
                    inst_and,
                    inst_nor,
                    inst_or,
                    inst_xor,
                    inst_sll,
                    inst_srl,
                    inst_sra,
                    inst_lui};
//---------------------------------{译码}end-----------------------------------//

//---------------------------------{执行}begin---------------------------------//
    wire [31:0] alu_result;

    alu alu_module(
        .alu_control  (alu_control ),  // I, 12, ALU 控制信号
        .alu_src1     (alu_operand1),  // I, 32, ALU 操作数 1
        .alu_src2     (alu_operand2),  // I, 32, ALU 操作数 2
        .alu_result   (alu_result  )   // O, 32, ALU 结果
    );
//---------------------------------{执行}end-----------------------------------//

//---------------------------------{访存}begin---------------------------------//
    wire [3 :0] dm_wen;
    wire [31:0] dm_addr;
    wire [31:0] dm_wdata;
    wire [31:0] dm_rdata;
    assign dm_wen   = {4{inst_SW}} & resetn;   // 内存写使能,非 resetn 状态下有效
    assign dm_addr  = alu_result;              // 内存写地址，为 ALU 结果
    assign dm_wdata = rt_value;                // 内存写数据，为 rt 寄存器值
    data_ram data_ram_module(
        .clk   (clk        ),  // I, 1,  时钟
        .wen   (dm_wen     ),  // I, 1,  写使能
        .addr  (dm_addr[6:2]),  // I, 32, 读地址
        .wdata (dm_wdata   ),  // I, 32, 写数据
        .rdata (dm_rdata   ),  // O, 32, 读数据

        //display mem
        .test_addr(mem_addr[6:2]),
        .test_data(mem_data    )
    );
//---------------------------------{访存}end-----------------------------------//

//---------------------------------{写回}begin---------------------------------//
```

```
    wire inst_wdest_rt;   // 寄存器堆写入地址为 rt 的指令
    wire inst_wdest_rd;   // 寄存器堆写入地址为 rd 的指令
    assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI;
    assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
                         | inst_OR   | inst_XOR  | inst_SLL | inst_SRL;
    // 寄存器堆写使能信号，非复位状态下有效
    assign rf_wen   = (inst_wdest_rt | inst_wdest_rd) & resetn;
    assign rf_waddr = inst_wdest_rd ? rd : rt;       // 寄存器堆写地址 rd 或 rt
    assign rf_wdata = inst_LW ? dm_rdata : alu_result;// 写回结果，为 load 结果或 ALU 结
果
//---------------------------------{写回}end-----------------------------------//
endmodule
```

7. single_cycle_cpu_display.v

```
`timescale 1ns / 1ps
//************************************************************************
// > 文件名: single_cycle_cpu_display.v
// > 描述 ：单周期 CPU 显示模块，调用 FPGA 板上的 IO 接口和触摸屏
//************************************************************************
module single_cycle_cpu_display(
    //时钟与复位信号
    input clk,
    input resetn,   //后缀"n"代表低电平有效

    //脉冲开关，用于产生脉冲 clk，实现单步执行
    input btn_clk,

    //触摸屏相关接口，不需要更改
    output lcd_rst,
    output lcd_cs,
    output lcd_rs,
    output lcd_wr,
    output lcd_rd,
    inout[15:0] lcd_data_io,
    output lcd_bl_ctr,
    inout ct_int,
    inout ct_sda,
    output ct_scl,
    output ct_rstn
    );
//-----{时钟和复位信号}begin
//不需要更改，用于单步调试
```

```
    wire cpu_clk;   //单周期 CPU 里使用脉冲开关作为时钟，以实现单步执行
     reg btn_clk_r1;
     reg btn_clk_r2;
    always @(posedge clk)
    begin
       if (!resetn)
       begin
          btn_clk_r1<= 1'b0;
       end
       else
       begin
          btn_clk_r1 <= ~btn_clk;
       end

       btn_clk_r2 <= btn_clk_r1;
    end

     wire clk_en;
    assign clk_en = !resetn || (!btn_clk_r1 && btn_clk_r2);
    BUFGCE cpu_clk_cg(.I(clk),.CE(clk_en),.O(cpu_clk));
//-----{时钟和复位信号}end

//-----{调用单周期 CPU 模块}begin

    //用于在 FPGA 板上显示结果
    wire [31:0] cpu_pc;    //CPU 的 PC
    wire [31:0] cpu_inst;  //该 PC 取出的指令
    wire [ 4:0] rf_addr;   //扫描寄存器堆的地址
    wire [31:0] rf_data;   //寄存器堆从调试端口读出的数据
    reg  [31:0] mem_addr;  //要观察的内存地址
    wire [31:0] mem_data;  //内存地址对应的数据
    single_cycle_cpu cpu(
       .clk    (cpu_clk   ),
       .resetn  (resetn    ),

       .rf_addr (rf_addr ),
       .mem_addr(mem_addr),
       .rf_data (rf_data ),
       .mem_data(mem_data),
       .cpu_pc  (cpu_pc  ),
       .cpu_inst(cpu_inst)
    );
//-----{调用单周期 CPU 模块}end
```

```
//--------------------{调用触摸屏模块}begin--------------------//
//-----{实例化触摸屏}begin
//此小节不需要更改
   reg        display_valid;
   reg  [39:0] display_name;
   reg  [31:0] display_value;
   wire [5 :0] display_number;
   wire       input_valid;
   wire [31:0] input_value;

   lcd_module lcd_module(
      .clk          (clk          ),  //10Mhz
      .resetn       (resetn       ),

      //调用触摸屏的接口
      .display_valid  (display_valid ),
      .display_name   (display_name  ),
      .display_value  (display_value ),
      .display_number (display_number),
      .input_valid    (input_valid   ),
      .input_value    (input_value   ),

      //lcd 触摸屏相关接口，不需要更改
      .lcd_rst       (lcd_rst      ),
      .lcd_cs        (lcd_cs       ),
      .lcd_rs        (lcd_rs       ),
      .lcd_wr        (lcd_wr       ),
      .lcd_rd        (lcd_rd       ),
      .lcd_data_io   (lcd_data_io  ),
      .lcd_bl_ctr    (lcd_bl_ctr   ),
      .ct_int        (ct_int       ),
      .ct_sda        (ct_sda       ),
      .ct_scl        (ct_scl       ),
      .ct_rstn       (ct_rstn      )
   );
//-----{实例化触摸屏}end

//-----{从触摸屏获取输入}begin
//根据实际需要输入的数修改此小节，
//建议对每一个数的输入，编写单独一个 always 块
   always @(posedge clk)
   begin
      if (!resetn)
      begin
```

```
            mem_addr <= 32'd0;
        end
        else if (input_valid)
        begin
            mem_addr <= input_value;
        end
    end
    assign rf_addr = display_number-6'd5;
//-----{从触摸屏获取输入}end

//-----{输出到触摸屏显示}begin
//根据需要显示的数修改此小节，
//触摸屏上共有 44 块显示区域，可显示 44 组 32 位数据
//44 块显示区域从 1 开始编号，编号为 1~44，
    always @(posedge clk)
    begin
        if (display_number >6'd4 && display_number <6'd37 )
        begin  //块号 5~36 显示 32 个通用寄存器的值
            display_valid <= 1'b1;
            display_name[39:16] <= "REG";
            display_name[15: 8] <= {4'b0011,3'b000,rf_addr[4]};
            display_name[7 : 0] <= {4'b0011,rf_addr[3:0]};
            display_value       <= rf_data;
         end
        else
        begin
            case(display_number)
                6'd1 : //显示 PC 值
                begin
                    display_valid <= 1'b1;
                    display_name  <= "   PC";
                    display_value <= cpu_pc;
                end
                6'd2 : //显示 PC 取出的指令
                begin
                    display_valid <= 1'b1;
                    display_name  <= " INST";
                    display_value <= cpu_inst;
                end
                6'd3 : //显示要观察的内存地址
                begin
                    display_valid <= 1'b1;
                    display_name  <= "MADDR";
                    display_value <= mem_addr;
```

```
            end
            6'd4 : //显示该内存地址对应的数据
            begin
               display_valid <= 1'b1;
               display_name  <= "MDATA";
               display_value <= mem_data;
            end
            default :
            begin
               display_valid <= 1'b0;
            end
         endcase
      end
   end
//-----{输出到触摸屏显示}end
//----------------------{调用触摸屏模块}end---------------------//
endmodule
```

5 实验结果

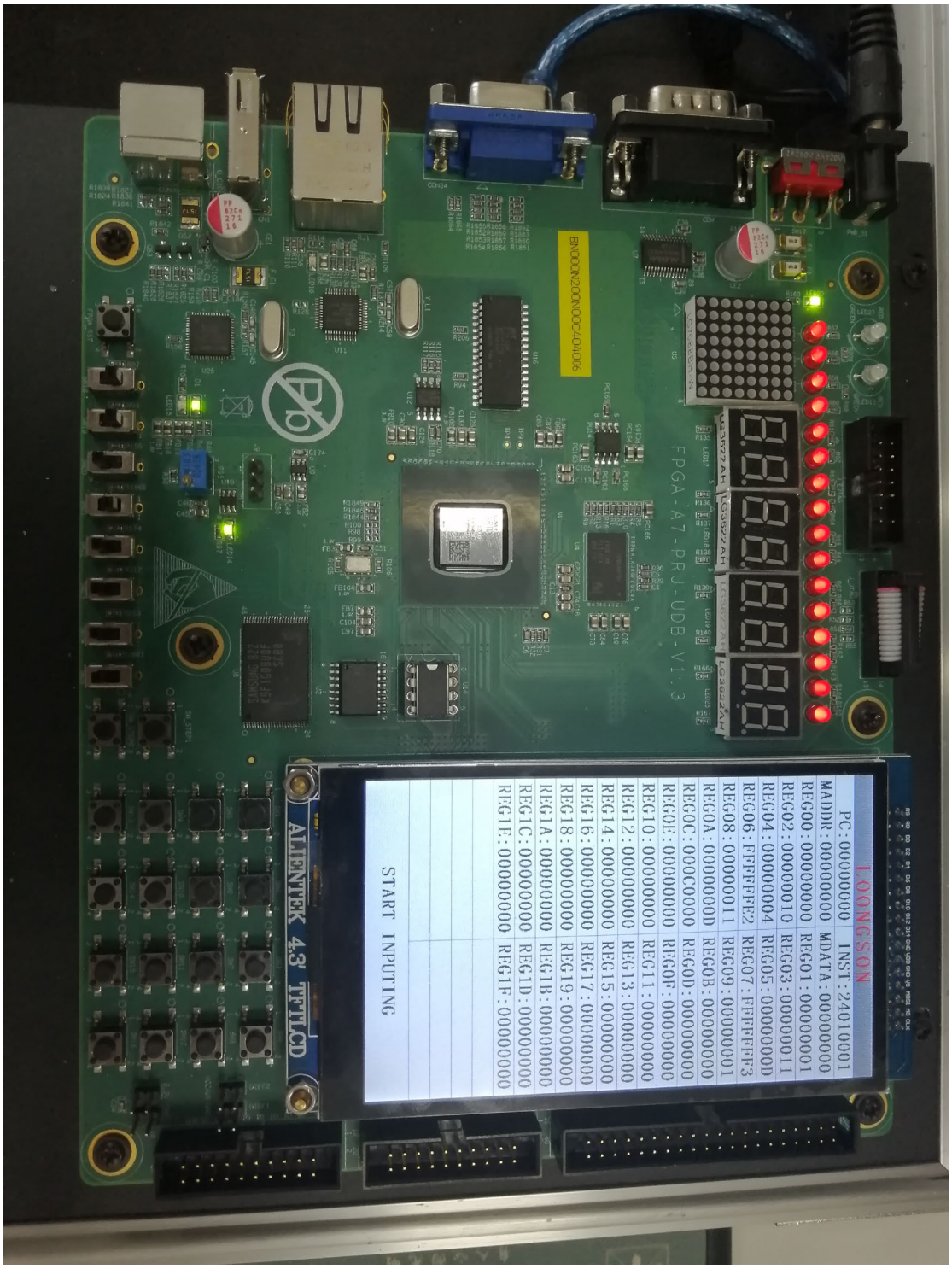
```
//-------------- 指令编码 ----------|指令地址|--- 汇编指令 -----|- 指令结果 -----//
assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1    | $1 = 0000_0001H
assign inst_rom[ 1] = 32'h00011100; // 04H: sll   $2 ,$1,#4    | $2 = 0000_0010H
assign inst_rom[ 2] = 32'h00411821; // 08H: addu  $3 ,$2,$1    | $3 = 0000_0011H
assign inst_rom[ 3] = 32'h00022082; // 0CH: srl   $4 ,$2,#2    | $4 = 0000_0004H
assign inst_rom[ 4] = 32'h00642823; // 10H: subu  $5 ,$3,$4    | $5 = 0000_000DH
assign inst_rom[ 5] = 32'hAC250013; // 14H: sw    $5 ,#19($1)  | Mem[0000_0014H] = 0000_000DH
assign inst_rom[ 6] = 32'h00A23027; // 18H: nor   $6 ,$5,$2    | $6 = FFFF_FFE2H
assign inst_rom[ 7] = 32'h00C33825; // 1CH: or    $7 ,$6,$3    | $7 = FFFF_FFF3H
assign inst_rom[ 8] = 32'h00E64026; // 20H: xor   $8 ,$7,$6    | $8 = 0000_0011H
assign inst_rom[ 9] = 32'hAC08001C; // 24H: sw    $8 ,#28($0)  | Mem[0000_001CH] = 0000_0011H
assign inst_rom[10] = 32'h00C7482A; // 28H: slt   $9 ,$6,$7    | $9 = 0000_0001H
assign inst_rom[11] = 32'h11210002; // 2CH: beq   $9 ,$1,#2    | 跳转到指令34H
assign inst_rom[12] = 32'h24010004; // 30H: addiu $1 ,$0,#4    | 不执行
assign inst_rom[13] = 32'h8C2A0013; // 34H: lw    $10,#19($1)  | $10 = 0000_000DH
assign inst_rom[14] = 32'h15450003; // 38H: bne   $10,$5,#3    | 不跳转
assign inst_rom[15] = 32'h00415824; // 3CH: and   $11,$2,$1    | $11 = 0000_0000H
assign inst_rom[16] = 32'hAC0B001C; // 40H: sw    $11,#28($0)  | Men[0000_001CH] = 0000_0000H
assign inst_rom[17] = 32'hAC040010; // 44H: sw    $4 ,#16($0)  | Mem[0000_0010H] = 0000_0004H
assign inst_rom[18] = 32'h3C0C000C; // 48H: lui   $12,#12      | [R12] = 000C_0000H
assign inst_rom[19] = 32'h08000000; // 4CH: j     00H          | 跳转指令00H
```

图表 1 指令堆

PC 初值为第一条指令地址，通过按键，PC=PC+4，直到第二十条指令，再跳转到第一条指令。

解析第一条指令，addiu $1,$0,#1，所以指令结果得到 1 号寄存器值为

1.从下图的 REG01 值为 1 可见。其余指令类似



图表 2 结果