

组成原理课程第五次实验报告

实验名称：存储器与单周期 CPU

学号：22122266 姓名：张恒硕 班次：0416

一、实验目的

1. 了解只读存储器 ROM 和随机存取存储器 RAM 的原理，理解 ROM 读取数据及 RAM 读取、写入数据的过程，理解计算机中存储器地址编址和数据索引方法，理解同步 RAM 和异步 RAM 的区别。
2. 掌握调用 xilinx 库 IP 实例化 RAM 的设计方法。
3. 理解 MIPS 指令结构、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，了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。
4. 熟悉并掌握单周期 CPU 的原理和设计。
5. 进一步加强运用 verilog 语言进行电路设计的能力。

二、实验内容说明

（一）存储器

1. 学习存储器的设计及原理，如 ROM 读地址索引读取数据过程及时序、RAM 读写时序、同步和异步区别等。学习计算机中内存地址编址和数据索引方法。学习 ISE 工具中调用库 IP 的方法。
2. 设计实验方案，画出结构框图，标出输入输出端口，确定存储器宽度、深度和写使能位数。
3. 调用 xilinx 库 IP 实例化一块同步 RAM，设置两个端口，一个用来正常读写，另一个作为调试端口，只使用读功能用于观察存储器内部数据。仿真得到正确的波形图。
4. 将以上设计作为一个单独的模块，设计一个外围模块去调用之。外围模块中需调用封装好的 LCD 触摸屏模块，显示 RAM 的正常端口的地址、待写入的数据和读出的数据，显示调试端口的地址和读出的数据。并且需要利用触摸功能输入正常端口的地址和写数据，以及调试端口的地址。
5. 将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板子上进行演示。

（二）单周期 CPU 的复现与修改

1. 学习 MIPS 指令集，深入理解常用指令的功能和编码，并进行归纳，确定处理器各部件的控制码，比如使用何种 ALU 运算，是否写寄存器堆等。
2. 确定要实现 MIPS 指令（一条 load 指令、一条 store 指令、10 条基础运算指令、一条跳转指令）。对这些指令进行分析，如下表：

指令类型	指令名称	汇编指令	指令码	源操作数 1	源操作数 2	源操作数 3	目的寄存器	功能描述
R 型指令	无符号加法	Addu rd, rs, rt	000000 rs rt rd 00000 100001	[rs]	[rt]		rd	$GPR[rd] = GPR[rs] + GPR[rt]$
	无符号减法	suburd, rs, rt	000000 rs rt r d 00 000 100011	[rs]	[rt]		rd	$GPR[rd] = GPR[rs] - GPR[rt]$

	有符号比较, 小于置位	sltrd, rs, rt	000000 rs rt rd 00000 101010	[rs]	[rt]		rd	$GPR[rd] = (\text{sign}(GPR[rs]) < \text{sign}(GPR[rt]))$
	按位与	andrd, rs, rt	000000 rs rt rd 00000 100100	[rs]	[rt]		rd	$GPR[rd] = GPR[rs] \& GPR[rt]$
	按位或非	norrd, rs, rt	000000 rs rt rd 00000 100111	[rs]	[rt]		rd	$GPR[rd] = \sim(GPR[rs] GPR[rt])$
	按位或	orrd, rs, rt	000000 rs rt rd 00000 100101	[rs]	[rt]		rd	$GPR[rd] = GPR[rs] GPR[rt]$
	按位异或	xorrd, rs, rt	000000 rs rt rd 00000 100110	[rs]	[rt]		rd	$GPR[rd] = GPR[rs] \wedge GPR[rt]$
	逻辑左移	sllrd, rt, shf	000000 00000 rt shf 000000		[rt]		rd	$GPR[rd] = \text{zero}(GPR[rt]) \ll \text{shf}$
	逻辑右移	srlrd, rt, shf	000000 00000 rt shf 000010		[rt]		rd	$GPR[rd] = \text{zero}(GPR[rt]) \gg \text{shf}$
I 型指令	立即数无符号加法	addiur, rs, imm	001001rs rt imm	[rs]	sign_ext(imm)		rt	$GPR[rt] = GPR[rs] + \text{sign_ext}(\text{imm})$
	判断相等跳转	beqrs, rt, offset	000100 rs rt offset	[rs]	[rt]			$\text{if } GPR[rs] = GPR[rt] \text{ then } PC = PC + \text{sign_ext}(\text{offset}) \ll 2$
	判断不等跳转	bners, rt, offset	000101 rs rt offset	[rs]	[rt]			$\text{if } GPR[rs] \neq GPR[rt] \text{ then } PC = PC + \text{sign_ext}(\text{offset}) \ll 2$
	从内存装载字	lwrt, offset(base)	100011 b rt offset	[b]	sign_ext(offset)		rt	$GPR[rt] = \text{Mem}[GPR[\text{base}] + \text{sign_ext}(\text{offset})]$

	向内存存储字	swrt, offset (base)	101011 b rt offset	[b]	sign_ext(offset)	[rt]	Mem[GPR[base]+sign_ext(offset)]=GPR[rt]
	立即数装载高位	luirt, imm	001111 00000 rt imm		{imm, 16'd0}	rt	GPR[rt]={imm, 16'd0}
J 型指令	直接跳转	jtarget	000010 target	PC	target		跳转, PC={PC[31:28], target, 2'b00}

ps: GPR 表示通用寄存器, [rs]表示寄存器 rs 里存储的值, PC 表示程序计数器; imm 为 16 位立即数, sign_ext(imm)表示对其进行符号扩展; target 为 26 位立即数。

3. 设计实验方案，画出结构框图。

单周期 CPU 的一条指令的所有操作在一个时钟周期内执行完：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。

store 指令：ALU 运算结果为数据存储地址，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；

load 指令：ALU 运算结果为数据存储地址，根据该值从 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；

写寄存器堆的操作：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；

分支跳转指令：将结果写入 pc 寄存器中。

4. 使用 verilog 编写相应代码。

5. 依据实现的指令, 编写汇编程序，内嵌到自行搭建的异步指令 ROM，进行验证。如下表：

表 7.5 单周期 CPU 测试所用汇编程序详述

指令地址	汇编指令	结果描述	机器指令的机器码	
			16 进制	二进制
00H	addiu \$1,\$0,#1	[\$1] = 0000_0001H	24010001	0010_0100_0000_0001_0000_0000_0000_0001
04H	sll \$2,\$1,#4	[\$2] = 0000_0010H	00011100	0000_0000_0000_0001_0001_0001_0000_0000
08H	addu \$3,\$2,\$1	[\$3] = 0000_0011H	00411821	0000_0000_0100_0001_0001_1000_0010_0001
0CH	srl \$4,\$2,#2	[\$4] = 0000_0004H	00022082	0000_0000_0000_0010_0010_0000_1000_0010
10H	subu \$5,\$3,\$4	[\$5] = 0000_000DH	00642823	0000_0000_0110_0100_0010_1000_0010_0011
14H	sw \$5,#19(\$1)	Mem[0000_0014H] = 0000_000DH	AC250013	1010_1100_0010_0101_0000_0000_0001_0011
18H	nor \$6,\$5,\$2	[\$6] = FFFF_FFE2H	00A23027	0000_0000_1010_0010_0011_0000_0010_0111
1CH	or \$7,\$6,\$3	[\$7] = FFFF_FFF3H	00C33825	0000_0000_1100_0011_0011_1000_0010_0101
20H	xor \$8,\$7,\$6	[\$8] = 0000_0011H	00E64026	0000_0000_1110_0110_0100_0000_0010_0110
24H	sw \$8,#28(\$0)	Mem[0000_001CH] = 0000_0011H	AC08001C	1010_1100_0000_1000_0000_0000_0001_1100
28H	slt \$9,\$6,\$7	[\$9] = 0000_0001H	00C7482A	0000_0000_1100_0111_0100_1000_0010_1010
2CH	beq \$9,\$1,#2	跳转到指令 34H	11210002	0001_0001_0010_0001_0000_0000_0000_0010
30H	addiu \$1,\$0,#4	不执行	24010004	0010_0100_0000_0001_0000_0000_0000_0100
34H	lw \$10,#19(\$1)	[\$10] = 0000_000DH	8C2A0013	1000_1100_0010_1010_0000_0000_0001_0011
38H	bne \$10,\$5,#3	不跳转	15450003	0001_0101_0100_0101_0000_0000_0000_0011
3CH	and \$11,\$2,\$1	[\$11] = 0000_0000H	00415824	0000_0000_0100_0001_0101_1000_0010_0100
40H	sw \$11,#28(\$0)	Mem[0000_001CH] = 0000_0000H	AC0B001C	1010_1100_0000_1011_0000_0000_0001_1100
44H	sw \$4,#16(\$0)	Mem[0000_0010H] = 0000_0004H	AC040010	1010_1100_0000_0100_0000_0000_0001_0000
48H	lui \$12,#12	[\$12] = 000C_0000H	3C0C000C	0011_1100_0000_1100_0000_0000_0000_1100
4CH	j 00H	跳转指令 00H	08000000	0000_1000_0000_0000_0000_0000_0000_0000

6. 仿真编写的代码，得到正确的波形图。

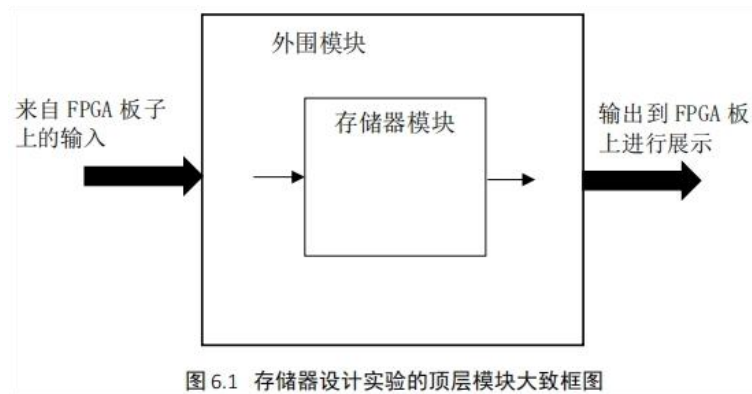
7. 将以上设计作为一个单独的模块，设计一个外围模块去调用之。外围模块中需调用封装好的 LCD 触摸屏模块，观察单周期 CPU 的内部状态，比如 32 个寄存器的值、PC 的值等。并利用触摸功能输入特定数据 RAM 地址，从该 RAM 的调试端口读出数据显示在屏上，以达到实时观察数据存储器内部数据变化的效果。通过这些手段，可以在板上充分验证 CPU 的正确性。

8. 将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板子上进行演示。

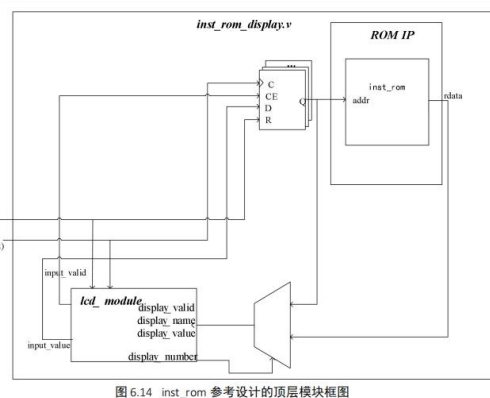
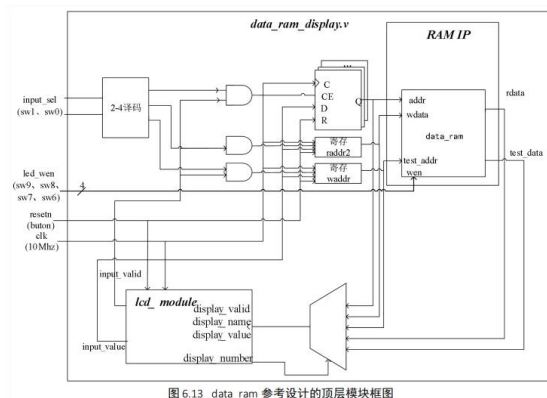
9. 修改代码，添加指令并实现 if-else 判断。

三、实验原理图

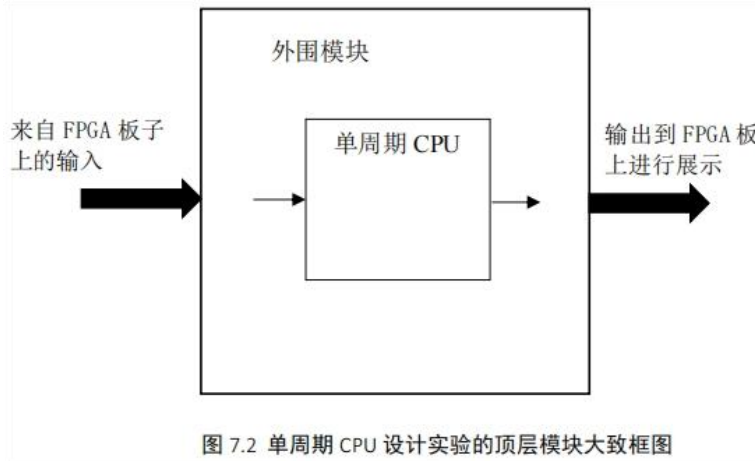
(一) 存储器

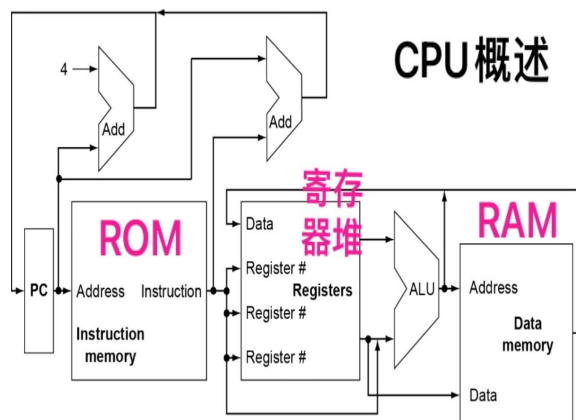


以下依次给出同步 RAM 和同步 ROM 的顶层模块框图：



(二) 单周期 CPU





CPU 概述

寄存器堆

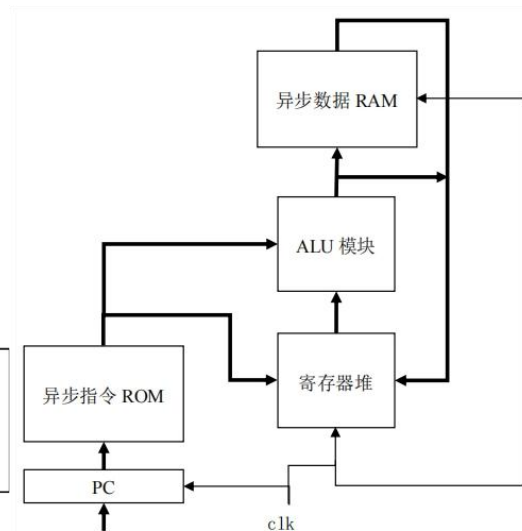


图 7.1 单周期 CPU 的大致框图

CPU 结构概述

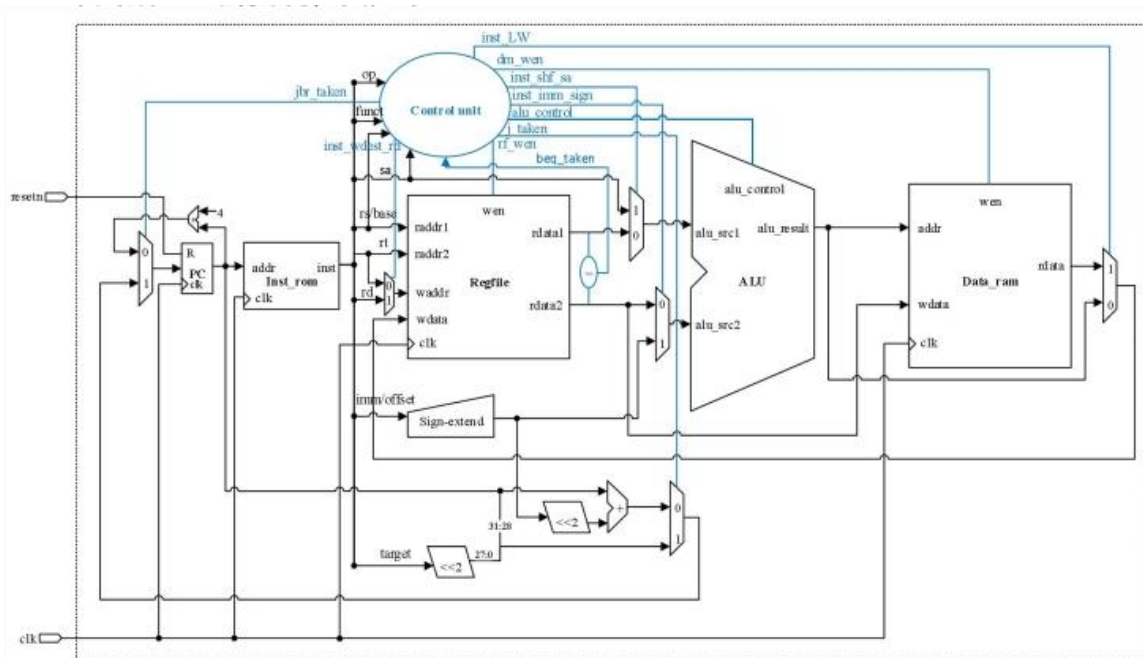


图 7.3 单周期 CPU 的实现框图

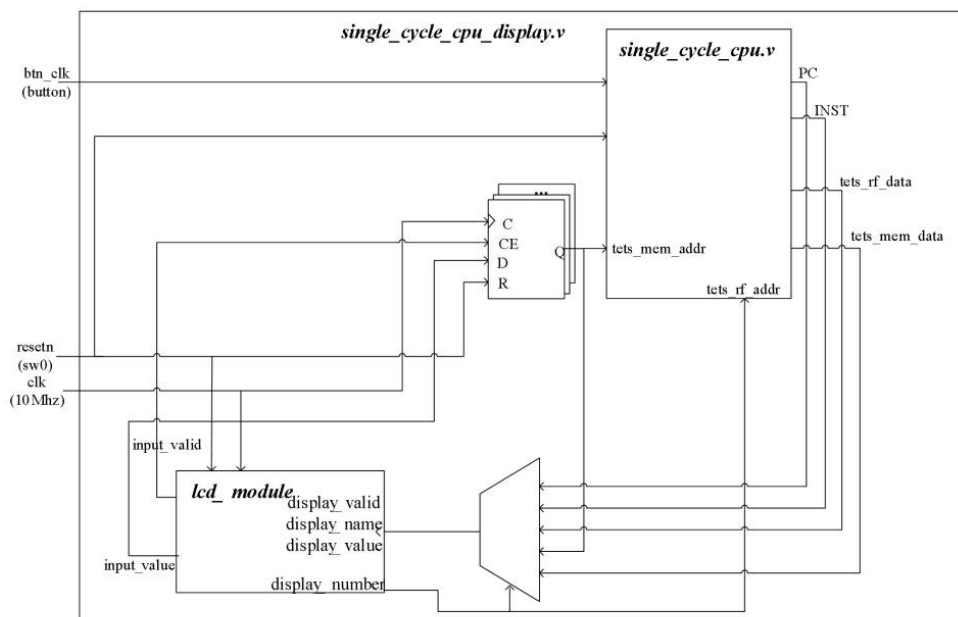


图 7.4 单周期 CPU 参考设计的顶层模块框图

以下给出修改增加指令的部分：

基于R型指令的编码方式：

$$000000 | \underbrace{rs}_{5bits, op2} | \underbrace{rt}_{5bits, op1} | \underbrace{rd}_{5bits, result} | 00000 | XXXXXX$$

其中 rs , rt , rd 对应汇编指令中的顺序是反向的，汇编中顺序为 $result$, $op2$, $op1$

令

1. 同或 $funcnt == 6'b101100$;

2. 与非 $funcnt == 6'b101101$;

均取6号和7号寄存器中的值分别为 rs , rt ，30号寄存器存储与非结果 rd ，31号寄存器存储同或结果 rd

编码为

1. 同或：

$$000000 | 00111 | 00110 | 11111 | 00000 | 101100$$

即十六进制 $00E6F82C$

2. 与非：

$$000000 | 00111 | 00110 | 11110 | 00000 | 101101$$

即十六进制 $00E6F02D$

指令类型	指令名称	汇编指令	指令码	源操作数1	源操作数2	目的寄存器	功能描述
R	按位	Xnor	$000000 rs rt r$	[r]	[r]	r	$GPR[rd] = \sim(GPR[rs]) \wedge$

型 指 令	同或	rd,rs,r t	d 00 000 101100	s]	t]	d	GPR[rt])
	按位 与非	nand rd,rs,r t	000000 rs rt r d 00 000 101101	[r s]	[r t]	r d	GPR[rd] = ~(GPR[rs] & GPR[rt])

四、实验步骤

(一) 存储器

以下给出了 ROM 的代码。

```
module inst_rom(
    input      [4 :0] addr, // 指令地址
    output reg [31:0] inst   // 指令
);

    wire [31:0] inst_rom[21:0]; // 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
        //添加同或和与非指令
        assign inst_rom[20] = 32'h00E7402E; // 50H: xnor   $15,$7,$6    | $15 =
FFFF_FFEH
        assign inst_rom[21] = 32'h00C7402E; // 54H: nand   $15,$7,$6    | $15 =
FFFF_FFEH

        //读指令,取4字节
        always @(*)
        begin
            case (addr)
                // 原有指令地址不变
                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];
                // 添加新指令地址
                5'd20 : inst <= inst_rom[20]; // 同或指令
                5'd21 : inst <= inst_rom[21]; // 与非指令
            endcase
        end

```

```

        default: inst <= 32'd0;
    endcase
end
endmodule

```

代码定义了不同的指令 inst 及其地址 addr，并给出了具体的读操作。
以下给出了 RAM 的代码。

```

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

```

代码定义了读、写数据 rdata 和 wdata，地址 addr，字节写使能 wen 以及时钟 clk。当写控制信号为 1 时，数据写入内存；并可以读取指定地址的数据。

（二）单周期 CPU

以下给出了单周期 CPU 的代码。

改动

注释

使用不同颜色标注了出来。

```
`timescale 1ns / 1ps
single_cycle_cpu.v
`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;
    reg [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;
    // 新指令: 选择下一条指令地址 (if-else 逻辑), 与上一行等效
    always @*
    begin
        if (jbr_taken) begin
            next_pc = jbr_target; // 跳转指令
        end else begin
            next_pc = seq_pc; // 不跳转, 继续顺序执行
        end
    end

    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
    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]), // 1, 5, 指令地址
        .inst     (inst            ) // 0, 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;
wire inst_XNOR, inst_NAND; // 新增同或和与非指令

```

```

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

```

```

assign inst_XNOR = op_zero & sa_zero & (funct == 6'b101100);// 逻辑同或运算
assign inst_NAND = op_zero & sa_zero & (funct == 6'b101101);// 逻辑与非运算
//以上为机器码指令构造的低六位，用于之后的机器码编址

```

```

// 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      ), // 1, 1
    .wen      (rf_wen   ), // 1, 1
    .raddr1   (rs       ), // 1, 5
    .raddr2   (rt       ), // 1, 5
    .waddr    (rf_waddr ), // 1, 5
    .wdata    (rf_wdata ), // 1, 32
    .rdata1   (rs_value ), // 0, 32
    .rdata2   (rt_value ), // 0, 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;
wire inst_xnor, inst_nand; // 新增控制信号
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
assign inst_xnor = inst_XNOR; // 逻辑同或

```



```

assign inst_nand = inst_NAND; // 逻辑与非
// 新增同或、与非两种指令

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 [13:0] alu_control; // 基于独热编码，新增两条指令应同时扩大 2 位位宽
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,
                    inst_xnor,
                    inst_nand};
// 同步新增同或、与非编码
//-----{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           ), // 1, 1, 时钟
    .wen    (dm_wen        ), // 1, 1, 写使能
    .addr   (dm_addr[6:2]), // 1, 32, 读地址
    .wdata  (dm_wdata      ), // 1, 32, 写数据
    .rdata  (dm_rdata      ), // 0, 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 | inst_xnor | inst_nand;
// 同步新增同或、与非作为寄存器堆写入地址为 rd 的指令
// 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

`timescale 1ns / 1ps
inst_rom.v
module inst_rom(
    input      [4:0] addr, // 指令地址
    output reg [31:0] inst // 指令
);

wire [31:0] inst_rom[21:0]; // 指令存储器, 字节地址 7'b000_0000~7'b111_1111
// 同步增加 2 个指令寄存器, 用于存储同或、与非的指令操作
//----- 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

```

//添加同或和与非指令

```

assign inst_rom[19] = 32'h00E6F82C; // 50H: xnor   $31,$7,$6    | $15 = FFFF_FFEEH
assign inst_rom[20] = 32'h00E6F02D; // 54H: nand   $30,$7,$6    | $15 = 0000_001DH
assign inst_rom[21] = 32'h08000000; // 4CH: j      00H          | 跳转指令 00H

```

// 机器码的对应如下，结果转换见表：

//读指令,取 4 字节

always @(*)

begin

case (addr)

// 原有指令地址不变

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];

```

```
// 添加新指令地址
```

```

5'd19 : inst <= inst_rom[19]; // 同或指令
5'd20 : inst <= inst_rom[20]; // 与非指令
5'd21 : inst <= inst_rom[21];

```

```
default: inst <= 32'd0;
```

```
endcase
```

```
end
```

```
Endmodule
```

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

```
alu.v
```

```
module alu(
```

```

input  [13:0] alu_control, // ALU 控制信号，同样扩宽 2 位
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
```

```
// 新增同或、与非信号
```

```
wire alu_xnor; //按位同或
wire alu_nand; //按位与非
```

```
// 独热编码，理论上应该与之前的 alu_control 中的顺序一一对应
```

```
assign alu_add  = alu_control[13];
assign alu_sub  = alu_control[12];
assign alu_slt  = alu_control[11];
assign alu_sltu = alu_control[10];
assign alu_and  = alu_control[ 9];
assign alu_nor  = alu_control[ 8];
assign alu_or   = alu_control[ 7];
assign alu_xor  = alu_control[ 6];
assign alu_sll  = alu_control[ 5];
assign alu_srl  = alu_control[ 4];
assign alu_sra  = alu_control[ 3];
assign alu_lui  = alu_control[ 2];
assign alu_xnor = alu_control[ 1];
assign alu_nand = 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;
```

```
// 新增同或、与非结果信号
```

```
wire [31:0] xnor_result;
wire [31:0] nand_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
```

```
// 同或、与非运算
```

```
assign xnor_result = ~(alu_src1 ^ alu_src2); // 同或结果为异或结果按位取反
assign nand_result = ~(alu_src1 & alu_src2); // 与非结果为与结果按位取反
```

```
//----{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 :

```

// 逻辑判断，无对应关系，因此直接加到后面

```

alu_xnor         ? xnor_result :
alu_nand         ? nand_result :

```

```
32'd0;
```

endmodule

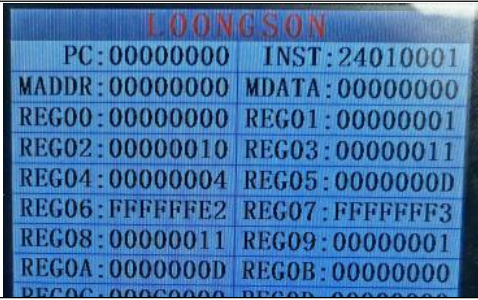
五、实验结果分析

(一) 存储器

	ROM	RAM
1		
2		
3		
4		
5		
6		

以上分别给出了测试只读存储器 ROM 和随机存取存储器 RAM 的样例。

(二) 单周期 CPU

START	REG00:0 00000000	00H: addiu \$1,\$0,#1	REG01:0 00000001	
04H: sll \$2,\$1,#4	REG02:0 00000010	08H: addu \$3,\$2,\$1	REG03:0 00000011	
0CH: srl \$4,\$2,#2	REG04:0 00000004	10H: subu \$5,\$3,\$4	REG05:0 0000000D	
18H: nor	REG06:F	1CH: or	REG07:F	

\$6 , \$5, \$2	FFFFFE2	\$7 , \$6, \$3	FFFFFFF3
20H: xor \$8 , \$7, \$6	REG08:0 0000011	28H: slt \$9 , \$6, \$7	REG09:0 0000001
34H: lw \$10, #19(\$ 1)	REG0A:0 000000D	3CH: and \$11, \$2, \$1	REG0B:0 0000000
48H: lui \$12, #12	REG0C:0 00C0000		REG0D:0 0000000
	REG0E:0 0000000		REG0F:0 0000000
	REG10:0 0000000		REG09:0 0000000
	REG12:0 0000000		REG13:0 0000000
	REG14:0 0000000		REG15:0 0000000
	REG16:0 0000000		REG17:0 0000000
	REG18:0 0000000		REG19:0 0000000
	REG1A:0 0000000		REG1B:0 0000000
	REG1C:0 0000000		REG1D:0 0000000
54H: nand \$30, \$7, \$6	REG1E:0 000001D	50H: xnor \$31, \$7, \$6	REG1F:F FFFFFFEE

六、总结感想

本次实验亲手搭建实现了 ROM、RAM 存储器，并在此基础上实现单周期 CPU，再进行改进。本次实验是对本学期前面实现的各项内容的一次综合应用，是对计算机组成原理课程的理解的一次综合检验。作为本课程的最后一个实验，本次实验全面地考察了我对学科知识的学习成果。