实验二报告

07152001 班 1120201198 史桠彬

实验内容

设计并实现一个简单处理器模型完成功能验证。

实验环境

ASUS VivoBook + Windows10 + Vivado2019.2，语言为 Verilog HDL。

实验要求

1. 处理器应基于 6.5 指令集或 MIPS 指令集或 RISCV 指令集进行设计或自行设计指令集。
2. 处理器的控制器结构应为微程序控制器或集硬布线控制器。
3. 处理器需支持算术运算、逻辑运算、存储器读写、寄存器间数据传送等几类指令。
4. 处理器需支持立即数寻址、寄存器寻址等数据寻址方式和顺序寻址、跳跃寻址两种指令的寻址方式。
5. 处理器需支持 8 条以上的指令。
6. 若选择自行设计指令集，处理器需能运行由自己设计的指令系统构成的一段程序，程序执行功能正确。

实验目标

设计模型处理器的总体结构、指令系统，合理安排时序信号。利用 Vivado 软件对该模型进行仿真分析和功能验证结果正确。

实验过程

1. 指令集及控制器结构选择

经查阅资料，我们选择基于 RISCV 指令集进行处理器设计。

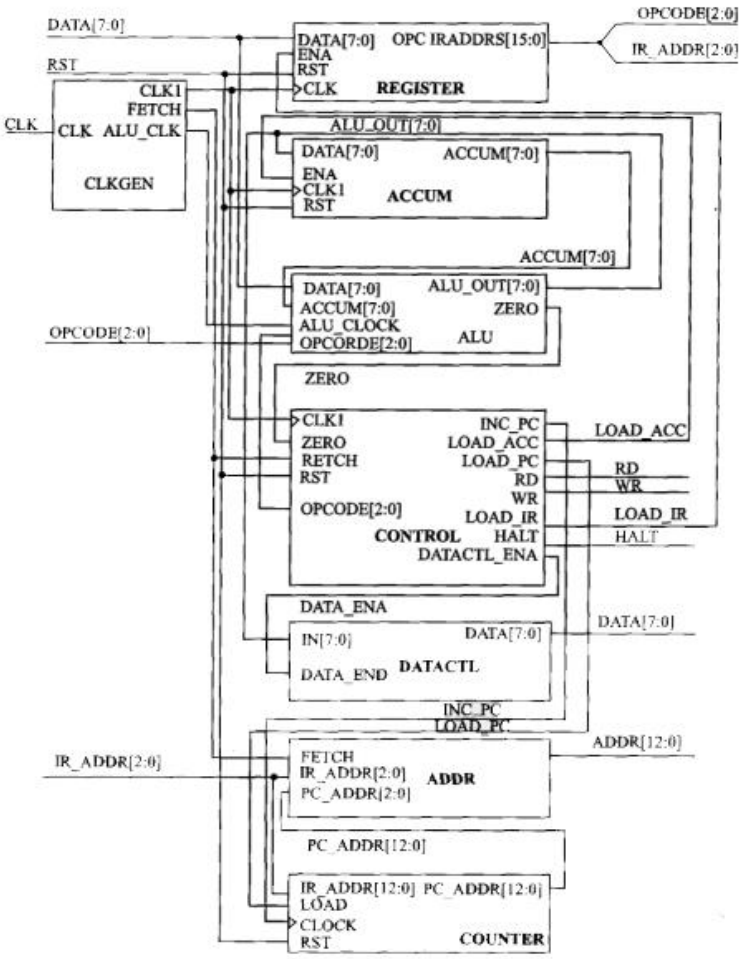
RISC-V 是一个基于精简指令集（RISC）原则的开源指令集架构（ISA）。基于 RISCV 指令集的简单 CPU 模型与一般的 CPU 模型相比，指令系统得到了简化，计算机的结构也更加简单合理，从而提高了运算速度。

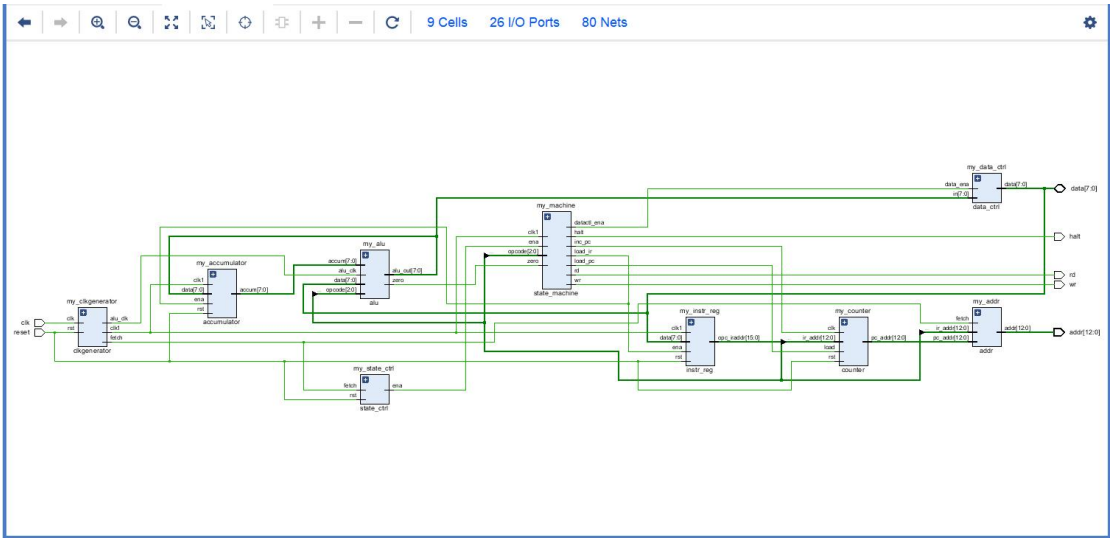
此外，基于 RISCV 指令集的简单 CPU 模型的时序控制信号形成部件是用硬布线逻辑实现的，是用触发器和逻辑门直接连线所构成的状态机和组合逻辑，产生控制序列的速度比用微程序控制方式快得多，因为这样做省去了读取微指令的时间。

2. 处理器模型结构

根据实验任务中处理机的性能指标和结构细节，可以将处理器分为以下八个部分：

（1）时钟发生器
（2）指令寄存器
（3）累加器
（4）算术逻辑运算单元
（5）数据控制器
（6）状态控制器
（7）程序计数器
（8）地址多路器





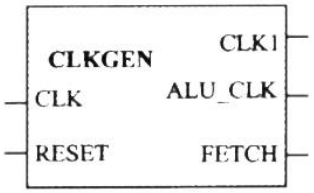
3. 部件结构和逻辑关系

（1） 时钟发生器

- 设计思想

时钟发生器利用外来时钟信号 clk 生成一系列时钟信号 clk1、fetch、alu_clk 并送往 CPU 的其他部件,其中,fetch 是控制信号,clk 的 8 分频信号。当 FETCH 高电平时,使 CLK 能

触发CPU控制器开始执行一条指令;同时 FETCH信号还将控制地址多路器输出指令地址和数据地址。clk信号用作指令寄存器、累加器、状态控制器的时钟信号。ALU-ENA则用于控制算术逻辑运算单元的操作。



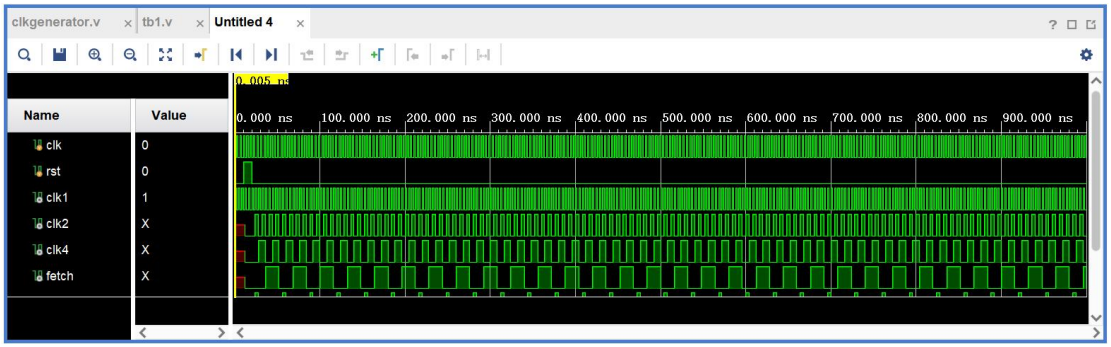
- 设计代码

```
`timescale 1ns / 1ps

module clkgenerator(clk,rst,clk1,clk2,clk4,fetch,alu_clk);
input clk,rst;
output wire clk1;
output reg clk2,clk4,fetch,alu_clk;//三个不同时钟，fetch时钟，alu时钟,全部要是reg类型

reg [7:0] state;
parameter S1=8'b0000_0001,
          S2=8'b0000_0010,
          S3=8'b0000_0100,
          S4=8'b0000_1000,
          S5=8'b0001_0000,
          S6=8'b0010_0000,
          S7=8'b0100_0000,
          S8=8'b1000_0000,
          idle=8'b0000_0000;
//利用状态机来写，提高了代码的可综合性
assign clk1=~clk;

always @(negedge clk)
    if(rst)
        begin
            clk2<=0;
            clk4<=0;
            fetch<=0;
            alu_clk<=0;
            state<=idle;
        end
    else
        begin
            case(state)
```

```
S1:
    begin
    clk2<=~clk2;//clk2 每次都反向，其实是 clk 的二倍
    alu_clk<=~alu_clk;
    state<=S2;
    end
S2:
    begin
    clk2<=~clk2;
    clk4<=~clk4;
    alu_clk<=~alu_clk;//alu 在一个大周期内只有一次
    state<=S3;
    end
S3:
    begin
    clk2<=~clk2;
    state<=S4;
    end
S4:
    begin
    clk2<=~clk2;
    clk4<=~clk4;//clk4 每两个系统时钟变一次，四倍的系统周期
    fetch<=~fetch;//fetch 每个大周期内变一次，是八个系统周期
    state<=S5;
    end
S5:
    begin
    clk2<=~clk2;
    state<=S6;
    end
S6:
    begin
    clk2<=~clk2;
    clk4<=~clk4;
    state<=S7;
    end
S7:
    begin
    clk2<=~clk2;
    state<=S8;
    end
S8:
    begin
    clk2<=~clk2;
```

```
                clk4<=~clk4;
                fetch<=~fetch;
                state<=S1;
                end
            idle:
                begin
                state<=S1;//初始状态
                end
            default://为了避免电路级错误而写 default
                begin
                state<=idle;
                end
        endcase
    end
endmodule
```

- 仿真波形图

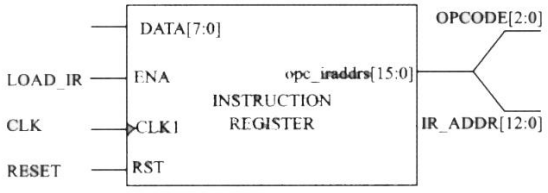


（2） 指令寄存器

- 设计思想

指令寄存器的触发时钟是 clk,在 clk 的正沿触发下,寄存器将数据总线送来的指令存高 8 位或低 8 位寄存器中,但并不是每个 ck 的上升沿都寄存数据总线的数据,因为数据总线上有时传输指令,有时传输数据。什么时候寄存,什么时候不寄存由 CPU 状态控制器的 loadir 信号控制, loadir 信号通过 ena 口输入到指令寄存器,复位后,指令寄存器被清为零。

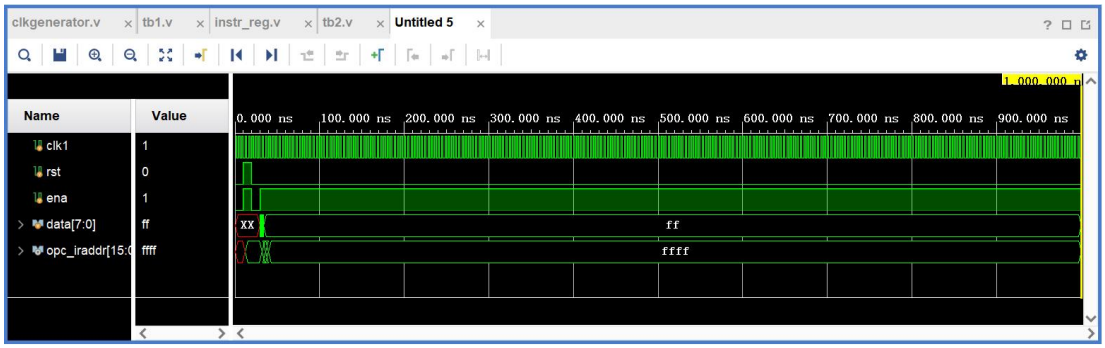
每条指令为两个字节,即 16 位,高 3 位是操作码,低 13 位是地址(CPU 的地址总线为 13 位,寻址空间为 8K 字节)本设计的数据总线为 8 位,所以每条指令需取两次,先取高 8 位,后取低 8 位,而当前取的是高 8 位还是低 8 位,由 state 变量记录 state 为 0 表示取的是高 8 位,存入高 8 位寄存器,同时将变量 state 置为 1. 下次再寄存时,由于 state 为 1,可知取的是低 8 位,存入低 8 位寄存器中。



- 设计代码

```
`timescale 1ns / 1ps

module instr_reg(opc_iraddr,data,ena,clk1,rst);
output [15:0] opc_iraddr;//这个就是输出的 16 进制 opcode 的地址
input [7:0] data;        //这个是输入数据
input ena,clk1,rst;      // load_ir 输入进来，表示这时是否进行指令地址寄存
reg [15:0] opc_iraddr;

reg state;
always @(posedge clk1)
    begin
        if(rst)
            begin
            opc_iraddr<=16'b0000_0000_0000_0000;
            state<=1'b0;
            end
        else
            begin
                if(ena)
                    begin
                        casex(state)
                            1'b0:
                                begin
                                opc_iraddr[15:8]<=data;//先存高八位
                                state<=1;
                                end
                            1'b1:
                                begin
                                opc_iraddr[7:0]<=data;//再存低八位
                                state<=0;
                                end
                            default:
                                begin
                                opc_iraddr[15:0]<=16'bxxxx_xxxx_xxxx_xxxx;
                                state<=1'bx;
                                end
                        endcase
                    end
                else
                    state<=1'b0;//如果总线不是指令，则 state 一直是 0
    end
    end
endmodule
```

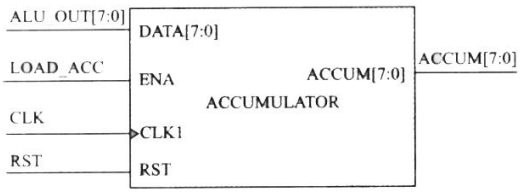
- 仿真波形图



（3） 累加器

- 设计思想

累加器用于存放当前的结果，它也是双目运算中的一个数据来源。复位后，累加器的值是零。当累加器通过 ena 口收到来自 CPU 状态控制器 load_acc 信号时，在 clk1 时钟正跳沿时就收到来自数据总线的数据。



- 设计代码

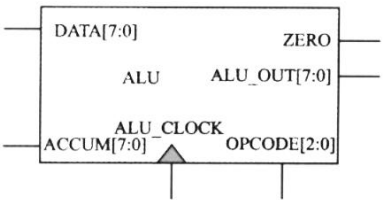
```
`timescale 1ns / 1ps

module accumulator(accum,data,ena,clk1,rst);
output [7:0] accum;
input [7:0] data;
input ena,clk1,rst;
reg [7:0] accum;

always @(posedge clk1 or negedge rst)
    begin
        if(rst)
            accum<=8'b0000_0000;//进行 reset
        else
            if(ena)
                accum<=data;//ena 时，信号正常输出
    end
endmodule
```

（4） 算数运算器

- 设计思想

根据输入的不同的操作码分别实现相应的加、与、异或、跳转等基本运算操作。基于这

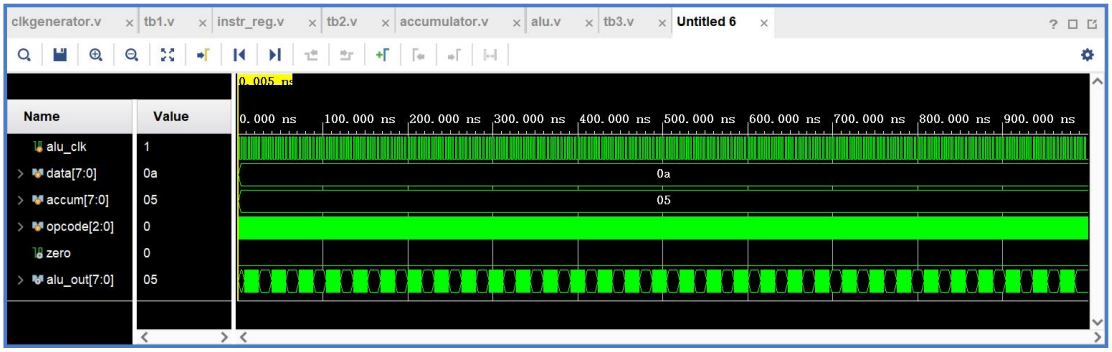
些基本运算，还可以实现很多种其他运算以及逻辑判断操作。



- 设计代码

```
`timescale 1ns / 1ps

module alu(alu_out,zero,data,accum,alu_clk,opcode);
output [7:0] alu_out;
output zero;
input [2:0] opcode;
input [7:0] data,accum;
input alu_clk;
reg [7:0] alu_out;

parameter HLT=3'b000,    //暂停指令，将操作数 accum 传输到输出
          SKZ=3'b001,    //跳过指令，也是将操作数传输到输出
          ADD=3'b010,     //加法
          ANDD=3'b011,    //位与运算
          XORR=3'b100,    //位异或运算
          LDA=3'b101,     //传输指令，将 data 传输到输出
          STO=3'b110,     //存储指令，将 accum 传输到输出
          JMP=3'b111;     //跳转指令，将 accum 传输到输出
assign zero=!accum;//accum 如果是全 0，那么就输出 zero 标识位为 1；
always @(posedge alu_clk)
    begin
        casex(opcode)
            HLT: alu_out<=accum;
            SKZ: alu_out<=accum;
            ADD: alu_out<=data+accum;
            ANDD: alu_out<=data&accum;
            XORR: alu_out<=data^accum;
            LDA: alu_out<=data;
            STO: alu_out<=accum;
            JMP: alu_out<=accum;
            default: alu_out<=8'bxxxx_xxxx;
        endcase
    end
endmodule
```

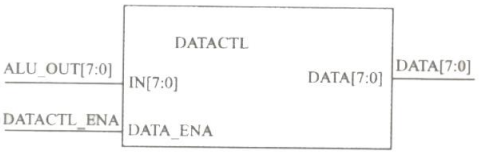
- 仿真波形图



（5） 数据控制器

- 设计思想

数据控制器的作用是控制累加器的数据输出,由于数据总线是各种操作时传送数据的公共通道,不同情况下传送不同的内容。有时要传输指令,有时要传送 RAM 区或接口的数据。累加器的数据只有在需要往 RAM 区或端口写时才允许输出,否则应呈现高阻态,以允许其他部件使用数据总线。所以任何部件往总线上输出数据时,都需要一控制信号。而此控制信号的启、停则由 CPU 状态控制器输出的各信号控制决定,数据控制器何时输出累加器的数据则由状态控制器输出的控制信号 datactl_ena 决定。



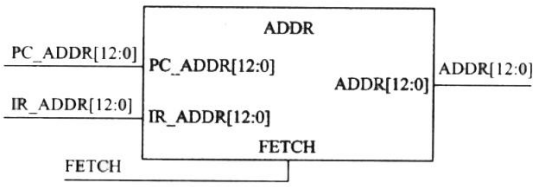
- 设计代码

```
`timescale 1ns / 1ps

module state_ctrl(ena,fetch,rst);
output ena;
input fetch,rst;
reg ena;
always @(posedge fetch or posedge rst)
    begin
        if(rst)
            ena<=0;
        else
            ena<=1;
//state_ctrl 模块的作用就是：在 fetch 上升沿时刻，如果 rst 是高，那就 enable 为低，否则就正常工作
    end
endmodule
```

（6） 地址多路器

- 设计思想

地址多路器用于选择输出的地址是 程序计数器地址还是数据/端口地址。每个指令周期的前 4 个时钟周期用于从 ROM 中读取指令,输出的应是 PC 地址;后 4 个时钟周期用于对 RAM 或端口的读写该地址由指给出。地址的选择输出信号由时钟信号的 8 分频信号 fetch 提供。



- 设计代码

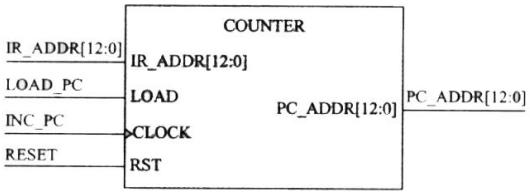
```
`timescale 1ns / 1ps

module addr(addr,fetch,ir_addr,pc_addr);
output [12:0] addr;
input [12:0] ir_addr,pc_addr;
input fetch;

assign addr=fetch? pc_addr:ir_addr;//在 fetch 的高电平期间读 pc 地址，低电平期间读数据或端口的地址
endmodule
```

（7） 程序计数器

- 设计思想

程序计数器用于提供指令地址，以便读取指令。指令按地址顺序存放在存储器中。有两种途径可形成指令地址:其一是顺序执行的情况,其二是遇到要改变顺序执行程序的情况,例如执行 JMP 指令后,需要形成新的指令地址。

复位后，指令指针为零，即每次 CPU 重新启动将从 ROM 的零地址开始读取指令并执行。每条指令执行完需两个时钟，这时 pc_addr 已被增 2，指向下一条指令。如果正在执行的指令是跳转语句,这时 CPU 状态控制器将会输出 lad_pc 信号，通过 oad 口进入程序计数器。程序计数器(pc_addr)将装入目标地址(ir_addr),而不是增 2。



- 设计代码

```
`timescale 1ns / 1ps

module counter(pc_addr,ir_addr,load,clk,rst);
```

```
output [12:0] pc_addr;
input [12:0] ir_addr;
input load,clk,rst;
reg [12:0] pc_addr;

always @(posedge clk or posedge rst)
    begin
        if(rst)
            pc_addr<=13'b0000_0000_0000;
        else
            if(load)//load 是 1 时,直接将输入进行传输
                pc_addr<=ir_addr;
            else
                pc_addr<=pc_addr+1;
    end
endmodule
```

（8） 状态控制器

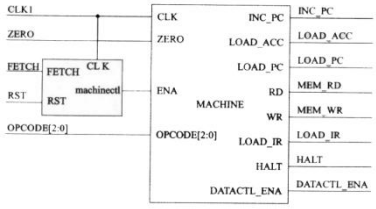
- 设计思想

状态控制器由两部分组成:

① 状态机

② 状态控制器

状态机控制器接收复位信号 RST,当 RST 有效时通过信号 ena 使其为 0 输入到状态机中，以停止状态机的工作。



- 设计代码

```
`timescale 1ns / 1ps

module state_ctrl(ena,fetch,rst);
output ena;
input fetch,rst;
reg ena;
always @(posedge fetch or posedge rst)
    begin
        if(rst)
            ena<=0;
        else
```

```
            ena<=1;
//state_ctrl 模块的作用就是：在 fetch 上升沿时刻，如果 rst 是高，那就 enable 为低，否则就正常工作
    end
endmodule


`timescale 1ns / 1ps

module state_machine(inc_pc,load_acc,load_pc,rd,wr,load_ir,
                        datactl_ena,halt,clk1,zero,ena,opcode);
output inc_pc,load_acc,load_pc;//指示信号
output rd,wr;//指示是进行读还是写状态
output load_ir; //指示是否进行装载（寄存地址）
output datactl_ena,halt;//输出的是 data_ctrl 的使能，以及 halt 信号
input clk1,zero,ena;    //输入的零标志位，主控时钟是 clk1（最快的那个），使能（由 state_ctrl 模块输出）
input [2:0] opcode;//输入的操作码

reg inc_pc,load_acc,load_pc,rd,wr,load_ir;
reg datactl_ena,halt;

parameter   HLT = 3 'b000,
            SKZ = 3 'b001,
            ADD = 3 'b010,
            ANDD = 3 'b011,
            XORR = 3 'b100,
            LDA = 3 'b101,
            STO = 3 'b110,
            JMP = 3 'b111;

reg [2:0] state;
always @(negedge clk1)
    begin
        if(!ena)
            begin
            state<=3'b000;
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;//在 enable 信号为低时，进行全部置零的操作
            end
        else
            ctl_cycle;//在 enable 为 1 的情况下，每次 clk1 的上升沿都进行一次 task
```

```
    end
    //-------ctl_cycle task--------
task ctl_cycle;
    begin
        casex(state)//是一个 state 数量为 8 的状态机，一个完整的操作周期
        3'b000://进来后的第一个状态
            begin
                {inc_pc,load_acc,load_pc,rd}<=4'b0001;//第一次:rd 和 load_ir
置高，寄存器读 rom 传过来的 8 位指令数据（高八位）
                {wr,load_ir,datactl_ena,halt}<=4'b0100;
                state<=3'b001;//按顺序进行下面的状态!
            end
        3'b001:
            begin//第二次：inc_pc 和 rd，load_ir 置高，pc 会加一，并且继续读
rom 的八位指令数据（低八位）
                {inc_pc,load_acc,load_pc,rd}<=4'b1001;
                {wr,load_ir,datactl_ena,halt}<=4'b0100;
                state<=3'b010;
            end
        3'b010:
            begin//第三次：空操作
                {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
                state<=3'b011;
            end
        3'b011:
            begin//第四次：pc 会加一，但前提是操作码不是 HLT
                if(opcode==HLT)
                    begin//如果操作码是 hlt，说明要暂停，这时输出一个 hlt 标
志位，并且是 pc 加一
                    {inc_pc,load_acc,load_pc,rd}<=4'b1000;
                    {wr,load_ir,datactl_ena,halt}<=4'b0001;
                    end
                else
                    begin//如果不是 hlt，pc 正常加一
                    {inc_pc,load_acc,load_pc,rd}<=4'b1000;
                    {wr,load_ir,datactl_ena,halt}<=4'b0000;
                    end
                state<=3'b100;
            end
        3'b100:
            begin//第五次：对不同操作符进行分支赋值
                if(opcode==JMP)
                    begin//如果是 jump，跳过这一条，那么就直接 load_pc，把目
```

的地址送给程序计数器

```
                {inc_pc,load_acc,load_pc,rd}<=4'b0010;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
                end
            else if(opcode==ADD || opcode==ANDD ||
                    opcode==XORR|| opcode==LDA)
                begin//如果是 ADD,ANDD,XORR,LDA,那就正常进行 read,计算得到数据
                {inc_pc,load_acc,load_pc,rd}<=4'b0001;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
                end
            else if(opcode==STO)
                begin//如果是 STO，那就将 datactl_ena（数据控制模块使能）置高，输出累加器的数据
                {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                {wr,load_ir,datactl_ena,halt}<=4'b0010;
                end
            else
                begin//否则，就全部为 0
                {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
                end
            state<=3'b101;
        end
    3'b101:
        begin//第六次：
            if(opcode==ADD || opcode==ANDD ||
                opcode==XORR || opcode==LDA)
                begin//如果是上面这些操作，那就要继续进行这些操作，与累加器的输出进行运算
                {inc_pc,load_acc,load_pc,rd}<=4'b0101;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
                end
            else if(opcode==SKZ && zero==1)
                begin//如果是 SKz，先判断是否是零，如果是零就 pc+1，否则就全零空操作
                {inc_pc,load_acc,load_pc,rd}<=4'b1000;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
                end
            else if(opcode==JMP)
                begin//如果是 JMP，那就 pc+1，然后 load_pc，锁定目的地址
                {inc_pc,load_acc,load_pc,rd}<=4'b1010;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
                end
```

```
        else if(opcode==STO)
            begin//如果是STO，就将数据写入地址处
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b1010;
            end
        else
            begin
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;
            end
        state<=3'b110;
    end
3'b110:
    begin//第七次，空操作
        if(opcode==STO)
            begin//如果是STO，那么需要使数据控制模块enable
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b0010;
            end
        else if(opcode==ADD || opcode==ANDD ||
            opcode==XORR || opcode==LDA)
            begin//如果是这些操作码，那就进行read
            {inc_pc,load_acc,load_pc,rd}<=4'b0001;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;
            end
        else
            begin
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;
            end
        state<=3'b111;
    end
3'b111:
    begin//第八次
        if(opcode==SKZ && zero==1)
            begin//如果是SKZ，并且是零，那么就pc+1,否则空操作
            {inc_pc,load_acc,load_pc,rd}<=4'b1000;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;
            end
        else
            begin
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;
            end
```

```
                state<=3'b000;
            end
        default:
            begin
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;
            state<=3'b000;
            end
        endcase
    end
endtask
//--------end of task--------
endmodule
```

4. 外围模块

（1） 地址译码器

- 设计代码

```
`timescale 1ns / 1ps

module decoder(addr,rom_sel,ram_sel);
output rom_sel,ram_sel;
input [12:0] addr;
reg rom_sel,ram_sel;
always @(addr)
    begin
        casex(addr)
            13'b1_1xxx_xxxx_xxxx:{rom_sel,ram_sel}<=2'b01;
            13'b0_xxxx_xxxx_xxxx:{rom_sel,ram_sel}<=2'b10;
            13'b1_0xxx_xxxx_xxxx:{rom_sel,ram_sel}<=2'b10;
            default:{rom_sel,ram_sel}<=2'b00;
       endcase
//只有地址码的前两位全 1，才选通 ram，否则一直是 rom
    end
endmodule
```

（2） RAM

- 设计代码

```
`timescale 1ns / 1ps

module ram(data,addr,read,write,ena);
    output [7:0] data;
    input [9:0] addr;
    input read,write,ena;
```

```
    reg [7:0] ram [10'h3ff: 0];
    assign data=(read && ena)?ram[addr]:8'hzz;
    always @(posedge write)
        begin
        ram[addr]<=data;
        end
endmodule
```

（3） ROM

- 设计代码

```
`timescale 1ns / 1ps

module rom(data,addr,read,ena);
    output [7:0] data;
    input [12:0] addr;
    input read,ena;
    reg [7:0] memory [13'h1fff : 0];
    wire [7:0] data;
    assign data=(read && ena)?memory[addr]:8'bzzzz_zzzz;
endmodule
```

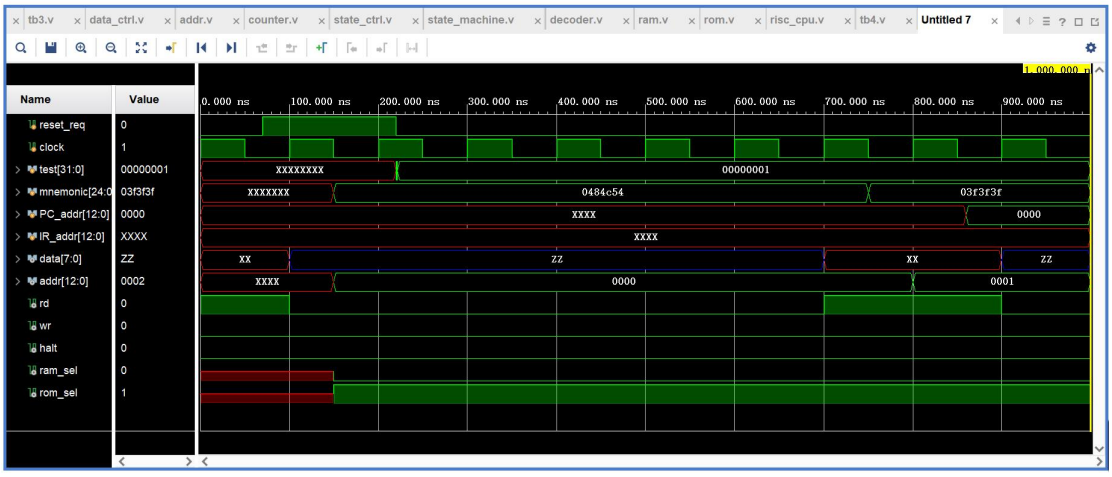
5. CPU 例化主模块

- 设计代码

```
`timescale 1ns / 1ps

module risc_cpu(clk,reset,halt,rd,wr,addr,data);

 input clk,reset;
 output rd,wr,addr,halt;
 inout data;
 wire clk,reset,halt;
 wire [7:0] data;
 wire [12:0] addr;
 wire rd,wr;
 wire clk1,fetch,alu_clk;
 wire [2:0] opcode;
 wire [12:0] ir_addr,pc_addr;
 wire [7:0] alu_out,accum;
 wire zero,inc_pc,load_acc,load_pc,load_ir,data_ena,contr_ena;

//1 时钟生成器
clkgenerator my_clkgenerator(.clk(clk),.clk1(clk1),.fetch(fetch),
                                .alu_clk(alu_clk),.rst(reset));
```

```
//2 指令寄存器
instr_reg
my_instr_reg(.opc_iraddr({opcode,ir_addr[12:0]}),.data(data),.ena(load_ir),
                        .clk1(clk1),.rst(reset));
//3 累加器
accumulator my_accumulator(.accum(accum),.data(alu_out),.ena(load_ir),
                              .clk1(clk1),.rst(reset));
//4 算数逻辑单元：
alu my_alu(.alu_out(alu_out),.zero(zero),.data(data),.accum(accum),
          .alu_clk(alu_clk),.opcode(opcode));
//5 状态机控制模块：
state_ctrl my_state_ctrl(.ena(contr_ena),.fetch(fetch),.rst(reset));
//6 状态机模块：
state_machine                                                    my_machine
(.inc_pc(inc_pc),.load_acc(load_acc),.load_pc(load_pc),
             .rd(rd), .wr(wr), .load_ir(load_ir), .clk1(clk1),
             .datactl_ena(data_ena), .halt(halt), .zero(zero),
             .ena(contr_ena),.opcode(opcode));
//7 数据控制器（累加器）
data_ctrl my_data_ctrl(.data(data),.in(alu_out),.data_ena(data_ena));
//8 地址多路器
addr
my_addr(.addr(addr),.fetch(fetch),.ir_addr(ir_addr),.pc_addr(pc_addr));
//9 程序计数器
counter my_counter(.pc_addr(pc_addr),.ir_addr(ir_addr),
                    .load(load_pc),.clk(inc_pc),.rst(reset));
endmodule
```

- 仿真波形图



6. 指令周期

根据实验要求，我们选择定长指令周期，指令周期由 8 个时钟周期组成,每个时钟周期

都要完成固定的操作。

(1)第 0 个时钟 CPU 状态控制器的输出 rd 和 load ir 为高电平其余均为低电平。指令寄存器寄存由 ROM 送来的高 8 位指令代码。

(2)第 1 个时钟与上一时钟相比只是 inc_pc 从 0 变为 1PC 增 1ROM 送来低 8 位指令代码,指令寄存器寄存该 8 位代码。

(3)第 2 个时钟，空操作。

(4)第 3 个时钟,PC 增 1,指向下一条指令。若操作符为 HLT,则输出信号 HLT 为高。如果操作符不为 HLT 除了 PC 增一外(指向下一条指令)其他各控制线输出为零。

(5)第 4 个时钟若操作符为 AND,ADDXOR 或 LDA,读相应地址的数据;若为 JMP 将目的地址送给程序计数器;若为 STO 输出累加器数据。

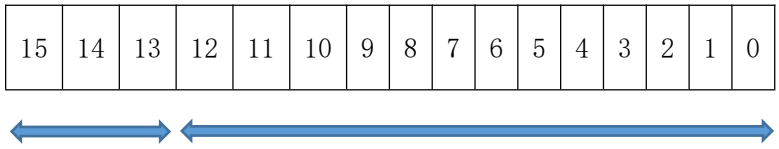
(6)第 5 个时钟若操作符为 ANDD,ADD 或 XORR 算术运算器就进行相应的运算;若为 LDA,就把数据通过算术运算器送给累加器;若为 SKZ,先判断累加器的值是否为 0,如果为 0,PC 就增 1,否则保持原值;若为 JMP,锁存目的地址;若为 STO,将数据写入地址处。

(7)第 6 个时钟，空操作。

(8)第 7 个时钟，操作符为 SKZ 且累加器值为 0,则 PC 值再增 1,跳过一条指令,否则 PC 无变化。

7. 指令系统

我们统一指令格式为：

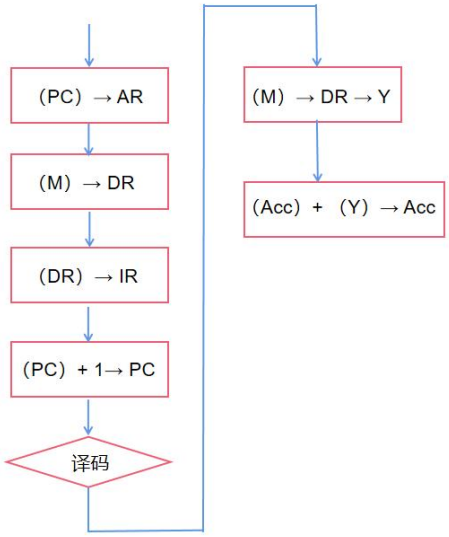


整个指令系统的指令举例：

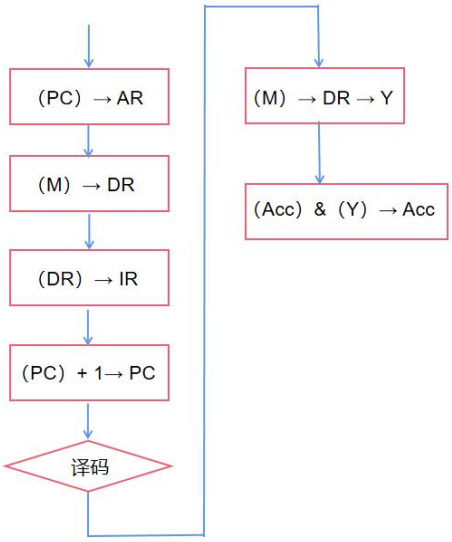
(1) HLT:停机操作。该操作将空一个指令周期,即 8 个时钟周期。

(2) SKZ:为零跳过下一条语句。该操作先判断当前 alu 中的结果是否为零,若是零就跳过下一条语句,否则继续执行。

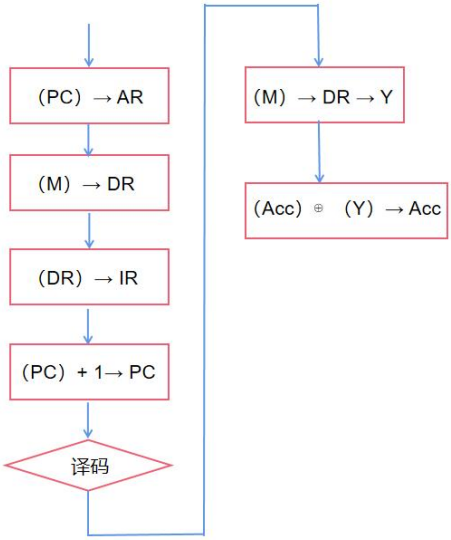
(3) ADD 相加:该操作将累加器中的值与地址所指的存储器或端口的数据相加，结果仍送回累加器中。



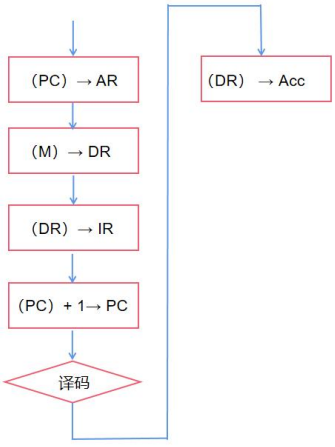
(4) AND 相与:该操作将累加器的值与地址所指的存储器或端口的数据相与,结果仍送回累加器中。



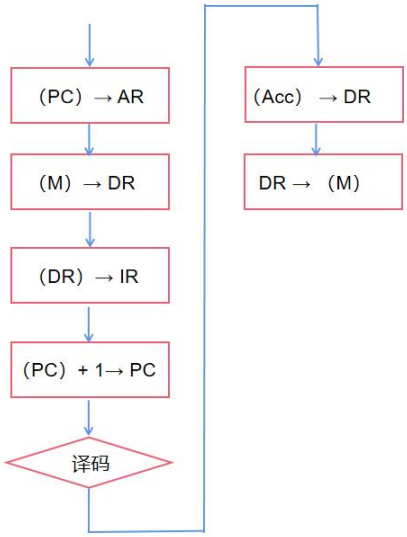
(5) XOR 异或:该操作将累加器的值与指令中给出地址的数据异或,结果仍送回累加器中。



(6) LDA 读数据:该操作将指令中给出地址的数据放入累加器。



(7) STO 写数据:该操作将累加器的数据放入指令中给出的地址。

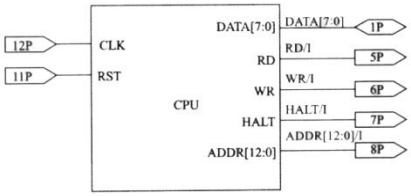


(8) JMP 无条件跳转语句:该操作将跳转至指令给出的目的地址，继续执行。

8. 布线、调试、验收

为了对所设计的处理器进行验证，我们把各部件包装在一个模块下,这样其内部连线就隐蔽起来,从系统的角度看就显得简洁。此外，我们建立了一些必要的外围器件模型,例如储存程序用的 ROM 模型、储存数据用的 RAM 和地址译码器等，它们不需要综合成具体的电路,只要保证功能和接口信号正确就能用于仿真。我们用虚拟器件来代替真实的器件对所设计的处理器进行验证,检查各条指令是否执行正确,与外围电路的数据交换是否正常。

在对所设计的处理器进行验证后，可以进行综合工作。第一阶段先对各个子模块分别加以综合以检查其可综合性。第二阶段把要综合的模块从仿真测试信号模块和虚拟外围电路模块中分离出来,组成一个独立的模块,其中包括了所有需要综合的模块。第三阶段把需要的综合的模块加载到综合器中。



通过自动综合工具，我们可以将 RTL 级的 Verilog 源代码模块进行综合，最后通过布局布线工具让其更加具体化。

实验心得

在本次实验过程中，我学习到了基于 RISCV 指令集的处理器的基本结构和原理，进一步深入了使用 Verilog HDL 进行仿真的过程，了解了常用的 Verilog 语法和验证方法，将书本中学到的内容进行了实践与应用，进一步掌握了 Vivado 的使用及设计代码、仿真代码的书写，对知识进行了巩固，本次实验让我收获颇丰。

实验期间，我遇到了许多难题。本次实验对我来说是一个巨大的挑战，从课本中线段相

连而成的模型示意图到真正能够在软件上仿真运行的实际模型，这中间的鸿沟属实巨大。

本次实验设计的处理器模型是参考北京航空航天大学夏宇闻教授编著的《Verilog 数字系统设计教程第二版》中的 RISCV-CPU 和众多 CSDN 博客教程进行改进后得到的，夏宇闻教授的教程对我的帮助颇大，特此感谢。