《FPGA 技术基础》学习报告

设计题目：利用 FPGA 制作一个八位 CPU

院　　系：信息工程与自动化学院自动化系

专　　业：自动化

班　　级：自动化 183

姓　　名：XXX

学　　号：

小组成员：XXX　　XXX

摘要

利用学习到的 FPGA 和微型计算机技术知识，制作一个八位图灵完备的哈佛架构 CPU。通过自行编写的机器码，能实现一般的移位指令，数字运算和逻辑运算指令，call、ret 调用指令，jmp 类转移指令等。可以利用 FPGA 开发板上的开关作为指令码和数据的输入，利用 led 小灯作为输出结果的显示。

关键词

FPGA、哈佛架构、DE2-70

目录

引言

我们两位同学一直都很想做一个 CPU，这个项目既有挑战性，又能很好的锻炼我们的各方面能力。本文是利用 FPGA 制作的一个 8bit 单周期 CPU，包含了一般 CPU 的大部分指令，如移位指令，控制转移指令，算术运算和逻辑运算指令。本次课设运用了 FPGAVerilog 语言和计算机基本结构的知识，帮助我们更深刻的理解所学习的知识。

第一章 概述

1.1 CPU18386 的基本结构

本次课程设计我们利用 FPGA 做一个图灵完备的哈佛结构 CPU，我们用我们的班级名称结合所学习的经典 X86 结构 CPU 将其命名为 18386。受限于 DE2-70 教育开发版的限制，18386 的地址总线只有八位，不能满足冯诺依曼结构 CPU 对程序指令的存储空间要求，于是我们选择了使用哈佛结构。采用哈佛总线体系结构的芯片内部程序空间和数据空间是分开的，这就允许同时取指令和取操作数，从而大大提高了运算能力。参考《深入理解计算机系统》一书中，作者自己定义了一个新的指令集，Y86-64，它其实就是 X86-64 的精简版，来说明 CPU 的体系结构。我们参考该书完成了 18386 的设计。运行过程如下图 1-1 所示。

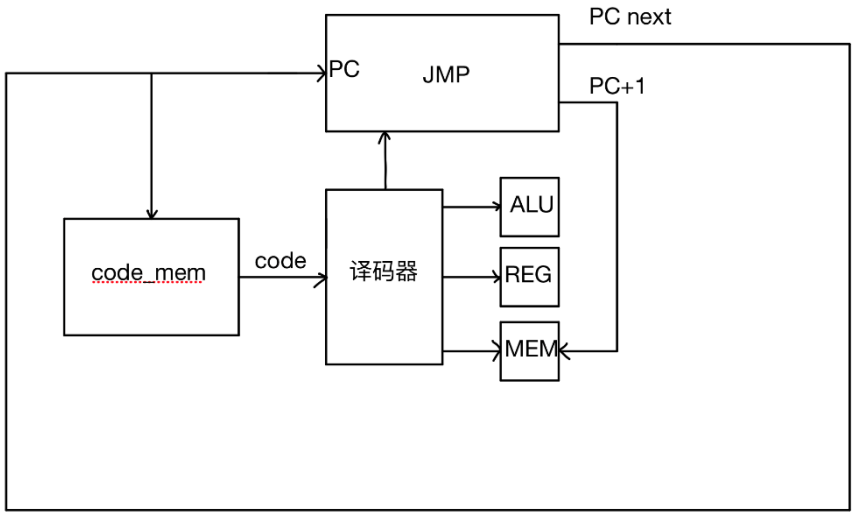

图 1-1 CPU18386 的运行过程

1.2 CPU 的运行过程

18386 的指令运行过程为取指->译码->执行->访存->写回->更新 PC。取指阶段从内存读取指令字节，地址为 PC；译码阶段读入两个操作数；在执行阶段 ALU 执行指令指明的操作；访存阶段将数据写入内存，或从内存读出数据；写回阶段

最多将两个结果写到寄存器文件中；更新 PC 将 PC 设置成下一条指令的地址。

18386 高电平计算低电平写寄存器、存储器。一条指令的地址 PC 写进 JMP 模块进行一次 PC+1，并将结果送入存储器储存，用于 CALL 指令等调用时返回使用。然后开始该程序的执行，PC 指向程序存储器中的一个代码的机器码，将该代码的机器码送入译码器，译码器将结果根据代码的内容，从 REG、MEM 中取出数据，将从 REG、MEM 中取出的数据和可能的立即数送入 ALU 计算；将计算结果根据指令送到 REG、MEM 中。

第二章 硬件系统

2.1 ALU

ALU 是 CPU 的运算器，负责完成算术运算和逻辑运算，18386 的 ALU 代码如下

```
module alu(
    input clk,
    input en,
    input[7:0]a,
    input[7:0]b,
    input[2:0]op,
    output [7:0]e,
    output reg[7:0]flag
);
    reg [8:0]y;
    assign e=y[7:0];
    always@(a or b or op)
    begin
        case(op)
        3'b000:begin y=a+b+9'b0;end//add
        3'b001:begin y=a-b+9'b0;end//sub
        3'b010:begin y=a&b+9'b0;end//and
        3'b011:begin y=a|b+9'b0;end//or
        3'b100:begin y=~a;end//not
        3'b101:begin y=a^b+9'b0;end//xor
        3'b110:begin y=a<<b+9'b0;end//shl
        3'b111:begin y=a>>b+9'b0;end//shr
        endcase
```

```
    end
    always@(negedge clk)
    begin
        if(en==1'b1)
        begin
            flag[0]<=(y==9'b0)?1'b1:1'b0;
            flag[1]<=y[7];
            flag[2]<=y[8];
        end
    end
endmodule
```

ALU 结构图如下图 2-1 所示。18386 的 ALU 有两个数据输入 a、b，一个操作码输入 op，输出数据 e，其中包含一个标志位寄存器，受限于时间问题，我们只添加了 JMP 类指令中最需要的 CF、SF、ZF 三个标志寄存器。其中的 reg [8:0]y 是为了方便提取 CF 进位标[志位而设计的，直接将[8]y 送给 CF 作为进位的标志位。SF 是符号标志位，直接将[7]y 送到 SF 中，always 语句中的 if 语句则是用于判断是否为零，结果送给零标志位 ZF。

图 2-1 ALU 模块

Case 语句的表达式为 op，通过不同的 op 选择不同的运算，ALU 中的算术运算和逻辑运算指令机器码对应如下表 2-1

000	001	010	011	100	101	110	111
ADD	SUB	AND	OR	NOT	XOR	SHL	SHR

表 2-1ALU 指令机器码对照表

标志寄存器 FLAG 内容如下表 2-2

0	0	0	0	0	CF	SF	ZF

表 2-2 FLAG 标志寄存器

2.2 存储器

存储器用于存放计算机工作所需要的程序和数据，18386 的存储器中存放了数据段，代码段，堆栈。存储器原理图如下图 2-2 所示。

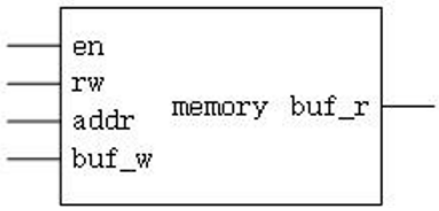

图 2-2 存储器模块

存储器有一个使能端 en；一个读写选择端 rw；选择进行读还是写；addr 为地址输入，一般输入是 PC；一个输入（写）缓存区 buf_w；一个输出缓存区 buf_r。

2.2.1 mem_code 代码存储区

代码存储区类似于 X86 汇编语言中的 CODE SEGMENT 定义的用于存放代码的区域，18386 的代码段程序如下

```
module mem_code(
    input en,
    input sign_rw,//chose
    input [7:0]addr,//pc
    input [15:0]buf_w,//write cache
    output [15:0]buf_r//read cache
);
    integer i;
    reg [15:0]mem[256];//procedure
    initial
    begin
        $readmemb("code.bin",mem);
    end
    always@(posedge en)
    begin
        if(en==1'b1)
        begin
```

```
            if(sign_rw==1'b1)
                mem[addr]<=buf_w;
        end
    end
    assign buf_r=(sign_rw==1'b0)?mem[addr]:16'd0;
endmodule
```

mem_code 有一个使能端 en，有一个选择器 sign_rw 用于选择是读出操作还是写入操作，有一个地址输入[7:0]addr 用于读入 PC 的内容，最后分别有一个读和写缓冲区 buf_r 和 buf_w。定义的程序段[15:0]mem[256]，可编写 256 条 16 字节的程序。当 sign_rw=0 时为读状态，将存储器单元 mem[addr]中的内容读入到 buf_r, 当 sign_rw=1 时为写状态，将存储器单元 buf_w 中的内容写入到 mem[addr]。

```
    begin
        $readmemb("code.bin",mem);
    end
```

其中的这一段代码表示将 code.txt 中的内容读入到 mem 中，这一段程序决定了我们可以在电脑上将我们像要的程序编写在 code.txt 中，将其送入我们的 18386 执行。

2.2.2 mem_data 数据存储区

数据存储区类似于 X86 汇编语言中的 DATA SEGMENT 定义的用于存放数据的区域，18386 的数据段程序如下

```
module mem_data(
    input clk,
    input en,
    input sign_rw,
    input [7:0]addr,
    input [7:0]buf_w,
    input [15:0]sw,
    output [15:0]led,
    output [7:0]buf_r
);
    integer i;
    reg [7:0]mem[256];
    initial
    begin
        for(i=0;i<256;i=i+1)
            mem[i]<=8'd0;
```

```
    end
    always@(negedge clk)
    begin
        if(en==1'b1)
        begin
            if(clk==1'b0 && sign_rw==1'b1)
            begin
                mem[addr]<=buf_w;
            end
        end
        mem[252]<=sw[7:0];
        mem[253]<=sw[15:8];

    end
    assign buf_r= (sign_rw==1'b0 && en==1'b1)?mem[addr]:8'd0;
    assign led[15:8]=mem[255];
    assign led[7:0]=mem[254];
endmodule
```

mem_data 与 mem_code 类似，不同在于存放内容为程序执行所需要的数据，有一个使能端 en，有一个选择器 sign_rw 用于选择是读出操作还是写入操作，有一个地址输入[7:0]addr 用于读入 PC 的内容，最后分别有一个读和写缓冲区 buf_r 和 buf_w。因为 mem_data 会有输出到 FPGA 版上的可能，所以还有 led 用于输出。具体每个功能的使用会在第三章指令集部分详细讲述。

2.3 JMP 模块

JMP 模块服务于 PC、CALL、RET 和 JMP 类指令，用于跳转到所需的地址，18386 的 JMP 模块代码如下所示。

```
module jmp(
    input clk,
    input en,
    input [2:0]op,
    input [7:0]pc,
    input [7:0]addr,
    input [7:0]flag,
    output reg [7:0]pc_inc,
    output reg [7:0]next_pc
);
    always@(pc)
```

```
    begin
        pc_inc<=pc+8'b1;
    end
    always@(negedge clk)
    begin
        next_pc<=pc+8'b1;
        if(en==1'b1)
        begin
            case(op)
                3'b000:next_pc<=addr;//jmp
                3'b001:begin if(flag[0]==1'b1 && flag[2]==1'b0)
next_pc<=addr;end//je
                3'b010:begin if(flag[0]==1'b0) next_pc<=addr;end//jne
                3'b011:begin if(flag[0]==1'b0 && flag[2]==1'b0)
next_pc<=addr;end//ja
                3'b100:begin if(flag[0]==1'b0 && flag[2]==1'b1)
next_pc<=addr;end//jb
                3'b101:begin if(flag[0]==1'b1) next_pc<=addr;end//jc
                3'b110:begin if(flag[0]==1'b0) next_pc<=addr;end//jnc
            endcase
        end
    end
endmodule
```

JMP 模块原理图如下图 2-3 所示，JMP 模块同样有一个使能端 en；一个代表操作数的 op 输入；表示程序地址的 PC；地址输入 addr，addr 一般是目标地址；flag 代表标志位输入，JMP 指令之中出了 JMP 直接跳转和 JA,JB 根据大小跳转以外，大部分都是依据标志位进行跳转，所以需要有一个标志位的输入。

输出方面一个是 pc_inc 为当前执行过程，next_pc 为下一条指令的地址，存放尽存储器中，当执行 CALL 和 RET 指令时，用于返回。

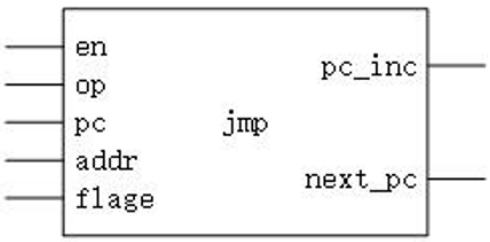

图 2-3

Case 语句的表达式为 op，通过不同的 op 选择不同的指令，JMP 模块中的指

令机器码对应如下表 2-3。

000	001	010	011	100	101	110
JMP	JE	JNE	JA	JB	JC	JNC

表 2-3 控制转移类指令对照表

2.4 register 寄存器模块

寄存器用来存放运算过程中所需要的操作数、操作数的地址、中间结果以及最后的运算结果。18386 的寄存器包含了通用寄存器、堆栈指针寄存器和空寄存器。

通用寄存器为 r0~r5，用来存放 8 位数据，这 6 个通用寄存器没有设置专门的用途，都可以用来存放地址和数据信息。堆栈指针寄存器一直指向栈顶位置，用于设置和访问堆栈。寄存器的代码如下所示。

```
module register(
    input clk,
    input [7:0]buf_w1,
    input [7:0]buf_w2,
    input r_En,
    input w_En,
    input [2:0]addr_r1,
    input [2:0]addr_r2,
    input [2:0]addr_w1,
    input [2:0]addr_w2,
    output [7:0]buf_r1,
    output [7:0]buf_r2
);
    reg [7:0]registers[8];

    integer i;

    initial
    begin
       for(i=0;i<8;i=i+1)
           registers[i]<=8'd0;
    end
    always@(negedge clk)
    begin
          if(clk==1'b0 && w_En==1'b1)
```

```
        begin
            registers[addr_w1]<=buf_w1;
            registers[addr_w2]<=buf_w2;
        end
    end
     assign buf_r1=(r_En==1'b1)?registers[addr_r1]:8'd0;
     assign buf_r2=(r_En==1'b1)?registers[addr_r2]:8'd0;
endmodule
```

寄存器的结构如图 2-4。寄存器模块有写使能端 r_en，写使能端 w_en，写缓存区 buf_w1 和 buf_w2，读缓存区 buf_r1 和 buf_r2，读地址线 addr_r1 和 addr_r2，写地址线 addr_w1 和 addr_w2。寄存器有一共有 8 个，每个 8 位。进行写指令是时钟下降沿，同时读使能端 w_en=1'b1，将 buf_w1 和 buf_w2 中的内容送到写地址 addr_w1 和 addr_w2 所指向的寄存器；读操作同理。

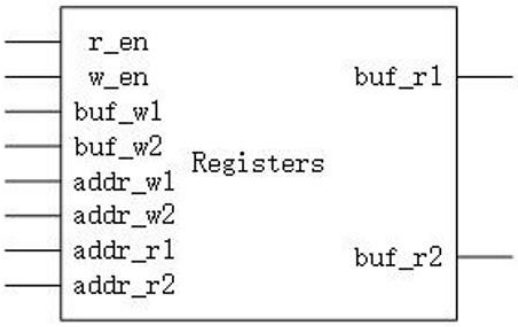

图 2-4 寄存器结构图

寄存器内容如下表 2-4 所示，r0~r5 为通用寄存器，110ps 寄存器相当于 X86 中的 sp 寄存器，用于存放栈顶指针。111 为空寄存器。

REG	000	001	010	011	100	101	110	111
	r0	r1	r2	r3	r4	r5	ps	null

表 2-4 寄存器内容

2.5 decoder 译码器

译码器是电子技术中的一种多输入多输出的组合逻辑电路，负责将二进制代码翻译为特定的对象（如逻辑电平等），功能与编码器相反。我们的译码器就是将输入的机器码译成对应的指令。译码器代码如下。

```
module decoder(
    input [15:0]code,
```

```
    output reg flag_hlt,
    output reg a_en,
    output reg m_en,
    output reg r_en_r,
    output reg r_en_w,
    output reg j_en,
    output reg [2:0]a_op,
    output reg [2:0]j_op,
    output reg mem_rw,
    output reg [2:0]reg1,
    output reg [2:0]reg2,
    output reg [2:0]regw1,
    output reg [2:0]regw2,
    output reg [7:0]num,
    output reg [7:0]alub,
    output reg [1:0]mem_data_addr_selector,
    output reg [1:0]mem_data_buf_w_selector,
    output reg alu_a_selector,
    output reg [1:0]alu_b_selector,
    output reg [1:0]reg_buf_w1_selector,
    output reg [1:0]reg_buf_w2_selector,
    output reg [1:0]jmp_addr_selector
);

    always@(code)
    begin
        flag_hlt=1'b0;
        a_en=1'b0;
        m_en=1'b0;
        r_en_r=1'b0;
        r_en_w=1'b0;
        j_en=1'b0;
        a_op=3'b000;
        j_op=3'b000;
        mem_rw=1'b0;
        reg1=3'b111;
        reg2=3'b111;
        regw1=3'b111;
        regw2=3'b111;
        num=8'b00000000;
        alub=8'b00000000;
        mem_data_addr_selector=2'b00;
        mem_data_buf_w_selector=2'b00;
```

```
alu_a_selector=1'b0;
alu_b_selector=2'b00;
reg_buf_w1_selector=2'b00;
reg_buf_w2_selector=2'b00;
jmp_addr_selector=2'b00;
if(code[15]==1'b1)
begin
    flag_hlt=1'b0;
    a_en=1'b1;
    m_en=1'b0;
    r_en_r=1'b1;
    r_en_w=1'b1;
    j_en=1'b0;
    a_op=code[13:11];
    reg1=code[10:8];
    regw1=code[10:8];
    regw2=3'b111;
    alu_a_selector=1'b0;
    reg_buf_w1_selector=2'b10;
    if(code[14]==1'b1)
    begin
        reg2=code[7:5];
        alu_b_selector=2'b01;
    end
    else
    begin
        reg2=3'b111;
        num=code[7:0];
        alu_b_selector=2'b10;
    end
end
else
begin
    if(code[14]==1'b1)
    begin
        flag_hlt=1'b0;
        j_en=1'b0;
        regw2=3'b111;
        case(code[13:11])
            3'b000:
            begin
                m_en=1'b0;
                r_en_r=1'b0;
```

```
        r_en_w=1'b1;
        reg1=3'b111;
        reg2=3'b111;
        regw1=code[10:8];
        regw2=3'b111;
        num=code[7:0];
        reg_buf_w1_selector=2'b01;
    end
    3'b001:
    begin
        m_en=1'b0;
        r_en_r=1'b1;
        r_en_w=1'b1;
        reg1=code[7:5];
        reg2=3'b111;
        regw1=code[10:8];
        regw2=3'b111;
        reg_buf_w1_selector=2'b11;
    end
    3'b010:
    begin
        m_en=1'b1;
        r_en_r=1'b0;
        r_en_w=1'b1;
        mem_rw=1'b0;
        reg1=3'b111;
        reg2=3'b111;
        regw1=code[10:8];
        regw2=3'b111;
        num=code[7:0];
        mem_data_addr_selector=2'b10;
        mem_data_buf_w_selector=2'b00;
        reg_buf_w1_selector=2'b00;
    end
    3'b011:
    begin
        m_en=1'b1;
        r_en_r=1'b1;
        r_en_w=1'b1;
        mem_rw=1'b0;
        reg1=code[7:5];
        reg2=3'b111;
        regw1=code[10:8];
```

```
                regw2=3'b111;
                mem_data_addr_selector=2'b00;
                mem_data_buf_w_selector=2'b00;
                reg_buf_w1_selector=2'b00;
            end
            3'b110:
            begin
                m_en=1'b1;
                r_en_r=1'b1;
                r_en_w=1'b0;
                mem_rw=1'b1;
                reg1=code[10:8];
                reg2=3'b111;
                regw1=3'b111;
                regw2=3'b111;
                num=code[7:0];
                mem_data_addr_selector=2'b10;
                mem_data_buf_w_selector=2'b01;
            end
            3'b111:
            begin
                m_en=1'b1;
                r_en_r=1'b1;
                r_en_w=1'b0;
                mem_rw=1'b1;
                reg1=code[10:8];
                reg2=code[7:5];
                regw1=3'b111;
                regw2=3'b111;
                mem_data_addr_selector=2'b00;
                mem_data_buf_w_selector=2'b10;
            end
        endcase
    end
    else
    begin
        case(code[13:12])
            2'b00:
            begin
                m_en=1'b0;
                r_en_r=1'b0;
                r_en_w=1'b0;
                j_en=1'b0;
```

```
        reg1=3'b111;
        reg2=3'b111;
        regw1=3'b111;
        regw2=3'b111;
        if(code[9]==1'b1)
            flag_hlt=1'b1;
        else
            flag_hlt=1'b0;
    end
    2'b01:
    begin
        flag_hlt=1'b0;
        m_en=1'b1;
        r_en_r=1'b1;
        r_en_w=1'b1;
        j_en=1'b0;
        reg1=3'b110;
        regw1=3'b110;
        alub=8'b0000_0001;
        alu_a_selector=1'b0;
        alu_b_selector=2'b11;
        reg_buf_w1_selector=2'b10;
        if(code[11:9]==3'b000)
        begin
            if(code[8]==1'b0)
            begin
                a_op=3'b001;
                mem_rw=1'b1;
                reg2=3'b111;
                regw2=3'b111;
                num=code[7:0];
                mem_data_addr_selector=2'b11;
                mem_data_buf_w_selector=2'b11;
            end
            else if(code[8]==1'b1)
            begin
                a_op=3'b001;
                mem_rw=1'b1;
                reg2=code[7:5];
                regw2=3'b111;
                mem_data_addr_selector=2'b11;
                mem_data_buf_w_selector=2'b10;
            end
```

```
        end
        else if(code[11:9]==3'b001)
        begin
            a_op=3'b000;
            mem_rw=1'b0;
            reg2=3'b111;
            regw2=code[7:5];
            mem_data_addr_selector=2'b00;
            reg_buf_w2_selector=00;
        end
    end
    2'b10:
    begin
        flag_hlt=1'b0;
        m_en=1'b1;
        r_en_r=1'b1;
        r_en_w=1'b1;
        j_en=1'b1;
        j_op=3'b000;
        reg1=3'b110;
        regw1=3'b110;
        regw2=3'b111;
        alub=8'b0000_0001;
        mem_data_buf_w_selector=2'b00;
        alu_a_selector=1'b0;
        alu_b_selector=2'b11;
        reg_buf_w1_selector=2'b10;
        reg_buf_w2_selector=2'b00;
        if(code[11:9]==3'b000)
        begin
            a_op=3'b001;
            mem_rw=1'b1;
            mem_data_addr_selector=2'b11;
            if(code[8]==1'b1)
            begin
                reg2=code[7:5];
                jmp_addr_selector=2'b11;
            end
            else
            begin
                reg2=3'b111;
                num=code[7:0];
                jmp_addr_selector=2'b01;
```

```
                    end
                end
                else if(code[11:9]==3'b001)
                begin
                    a_op=3'b000;
                    mem_rw=1'b0;
                    reg2=3'b111;
                    mem_data_addr_selector=2'b00;
                    jmp_addr_selector=2'b00;
                end
            end
            2'b11:
            begin
                flag_hlt=1'b0;
                m_en=1'b0;
                r_en_w=1'b0;
                j_en=1'b1;
                j_op=code[11:9];
                reg2=3'b111;
                regw1=3'b111;
                regw2=3'b111;
                if(code[8]==1'b1)
                begin
                    r_en_r=1'b1;
                    reg1=code[7:5];
                    jmp_addr_selector=2'b10;
                end
                else
                begin
                    r_en_r=1'b0;
                    reg1=3'b111;
                    num=code[7:0];
                    jmp_addr_selector=2'b01;
                end
            end
        endcase
    end
  end
end


endmodule
```

译码器代码很长但是逻辑几乎是一样的，机器码的第一位和第二位为选择指令集，对应的指令集可以在第三章系统指令集中对应。对于这个译码器，我们在编写的时候整理了一个表格，表格表示了每一个指令在执行过程中需要用到的各个模块的接口。表格在附录中，这里用 CALL 指令来举一个例子。表格表头对应下。

flag_hit//终止
m_en//存储器使能
r_en_r//寄存器读
r_en_w//寄存器写
j_en//jmp 指令使能
[2:0]a_op //alu 操作码
[2:0]j_op //jmp 操作码
mem_rw 存储器读写使能端=1 写=0 读
[2:0]reg1//寄存器读 1
[2:0]reg2 //寄存器读 2
[2:0]regw1 //寄存器写 1
[2:0]regw2 //寄存器写 2
[7:0]num //立即数
[7:0]alub //alu 第二个参数
[1:0]mem_data_addr_selector //存储器数据地址选择器//
[1:0]mem_data_buf_w_selector// 存储器数据缓存区选择器
[1:0]alu_a_selector //alua 类型选择器
[1:0]alu_b_selector// alub 类型选择器
[1:0]reg_buf_w1_selector// 寄存器写入 1 缓冲区选择
[1:0]reg_buf_w2_selector //寄存器写入 2 缓冲区选择
[1:0]jmp_addr_selector //jmp 指令地址选择

表中参数对应接口如下

```
mem_data_en:
    m_en
```

```
mem_data_sign_rw:
    mem_rw
mem_data_addr:
    reg_buf_r1      00
    reg_buf_r2      01
    num             10
    alu_e           11
mem_data_buf_w:
    jmp_pc_inc      00
    reg_buf_r1      01
    reg_buf_r2      10
    num             11

alu_en
    a_en
alu_a:
    reg_buf_r1      0
    reg_buf_r2      1
alu_b:
    reg_buf_r1      00
    reg_buf_r2      01
    num             10
    alub            11
alu_op:
    a_op

reg_en_r:
    r_en_r
reg_en_w:
    r_en_w
reg_addr_r1:
    reg1
reg_addr_r2:
    reg2
reg_addr_w1:
    regw1
reg_addr_w2:
    regw2
reg_buf_w1:
    mem_data_buf_r  00
    num             01
    alu_e           10
    reg_buf_r1      11
```

```
reg_buf_w2:
    mem_data_buf_r  00
    num             01
    alu_e           10
    reg_buf_r2      11

jmp_en:
    j_en
jmp_op:
    j_op
jmp_pc:
    pc
jmp_flage:
    flage
jmp_addr:
    mem_data_buf_r  00
    num             01
    reg_buf_r1      10
    reg_buf_r2      11
```

要执行 CALL 指令，首先 CALL 指令是跳转到一个指定的地址，他不存在暂停执行和空执行，所以 flag_hlt=0。而他需要读入一个内容，这个内容应该是一个地址，需要跳转的目标地址，所以需要读使能。同时他需要将 PC+1 送入堆栈存起来，堆栈是在存储器中的，所以存储器使能端 m_en=1 存储器读写使能端 mem_rw=1 表示写入。这个过程需要改变栈顶指针 ps 的值，所以需要对寄存器进行读写，所以 r_en_r=1,r_en_w=1。CALL 指令是直接跳转到某个地方，所以需要用到 JMP 模块，所以 j_en=1，又因为是直接跳转，所以 j_op=000，表示 JMP 指令。改变 PS 的值需要用到 ALU 做加法运算，所以 a_op=001，表示 ADD 指令。如图 2-5、2-6 所示。

指令	flag_hlt	m_en	r_en_r	r_en_w	j_en	[2:0]a_op	[2:0]j_op
call num	0	1	1	1	1	001	000

图 2-5

因为只用到 ps 寄存器，所以 reg1=110，reg2 寄存器没有使用，所以填入 null 寄存器。因为这是立即寻址的 CALL 指令，操作数为立即数，所以 num 为写入的立即数，alub 也就是 alu 的第二个操作数，上文提到了需要 PC+1，alua=PC，所以 alub=1。

mem_rw	[2:0]reg1	[2:0]reg2	[2:0]regw1	[2:0]regw2	[7:0]num	[7:0]alub
1	110	111	110	111	xxxxxxxx	00000001
1	110	xxx	110	111	????????	00000001

图 2-6

CALL 指令会用到 alu 的结果所以 mem_data_addr_selector=11 如图 2-7 所示表示选择 alue 也就是 alu 的输出结果。Mem_data_buf_w_selector 是存储器数据缓存区选择器，CALL 指令对存储器输入的是下一条指令的地址，所以 Mem_data_buf_w_selector=00 表示输入为 jmp_pc_inc。alu_a_selector 和图 2-8 中的 alu_b_selector 表示 ALU 的两个输入端的类型。

[1:0]mem_data_addr_selector	[1:0]mem_data_buf_w_selector	alu_a_selector
11	00	0

图 2-7

alu_a_selector=0 表示输入为寄存器 1，alu_b_selector=11 表示输入为立即数。reg_buf_w1_selector 是寄存器写入 1 缓冲区 reg_buf_w1_selector=10 表示写入内容为 ALU 的计算结果 alue。reg_buf_w2_selector 是寄存器写入 2 缓冲区选择，由于寄存器 2 为空寄存器，所以此处无意义。最后一个 jmp_addr_selector 是 JMP 指令地址选择。CALL 指令使用到了 JMP，而跳转目的地址是由立即数提供，所以 jmp_addr_selector=01。最终 CALL 指令执行原理如图 2-9 所示。

[1:0]alu_b_selector	[1:0]reg_buf_w1_selector	[1:0]reg_buf_w2_selector	[1:0]jmp_addr_selector
11	10	??	01

图 2-8

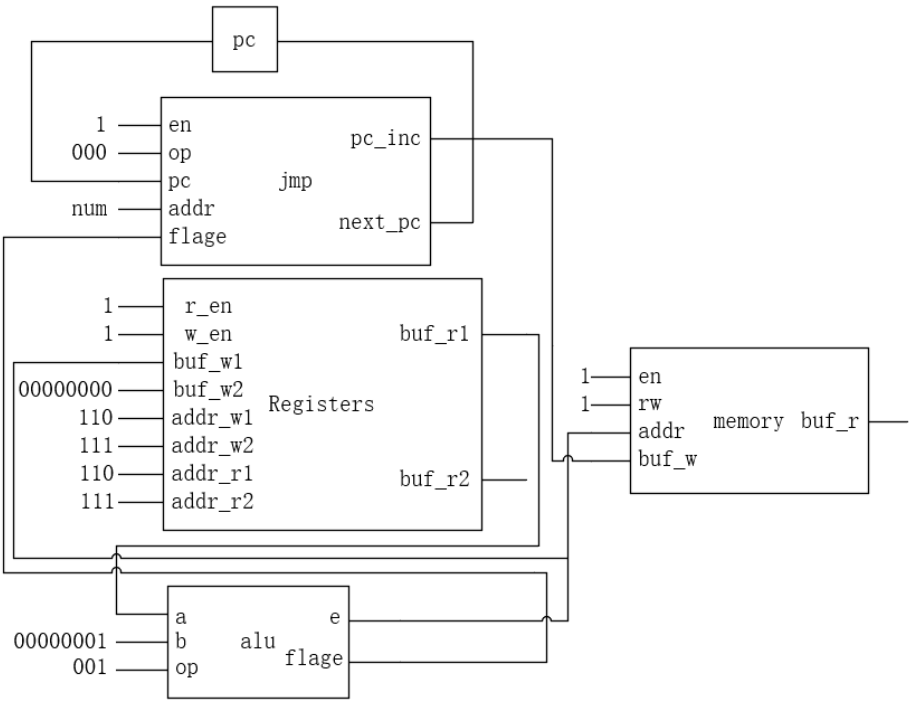

图 2-9 CALL 指令的执行原理

2.6 reset 复位模块

```
module reset(
    input clk_1Hz,
    input button_rst,
    output reg pc_rst,
    output reg clk_rst
);
    reg [2:0]conter=2'd0;
    initial
    begin
        conter<=2'd0;
        pc_rst<=1'b1;
        clk_rst<=1'b1;
    end
    always@(posedge clk_1Hz or negedge button_rst)
    begin
        if(button_rst==1'b0)
        begin
            conter<=2'd0;
            pc_rst<=1'b1;
            clk_rst<=1'b1;
```

```
        end
        else
        begin
            if(conter>=2'd1)
                pc_rst<=1'b0;
            else
                pc_rst<=1'b1;
            if(conter>=2'd2)
                clk_rst<=1'b0;
            else
            begin
                clk_rst<=1'b1;
                conter<=conter+2'd1;
            end
        end


    end
endmodule
```

复位模块输出两个复位信号，一个是时钟复位 clk_rst，一个是 pc 复位 pc_rst。通电后，pc 复位和 clk 复位都置 1。过一秒后，pc 复位置 0，再一秒后 clk 复位置 0，CPU 开始正常工作。复位有按钮 button_rst，按下后开始复位，程序重新执行。

2.7 Clock 时钟模块

```
module clock(
    input sw_clk,
    input clock_50MHz,
    input en,
    output reg clock_main,
    output reg clock_1Hz
);
    reg [31:0]conter_main;
    reg [31:0]conter_1Hz;
    initial
    begin
        clock_1Hz=1'b0;
        clock_main=1'b1;
        conter_main=32'd0;
        conter_1Hz=32'd0;
```

```
    end
    always@(posedge clock_50MHz)
    begin
        if(en==1'b0)
        begin
            conter_main<=32'd0;
            clock_main<=1'b1;
        end
        else if((sw_clk==1'b0 && conter_main>=32'd25) || (sw_clk==1'b1
&& conter_main>=32'd25000000))
        begin
            clock_main<=~clock_main;
            conter_main<=32'd0;
        end
        else
            conter_main<=conter_main+32'd1;
    end

    always@(posedge clock_50MHz)
    begin
        if(conter_1Hz>=32'd25000000)
        begin
            conter_1Hz<=32'd0;
            clock_1Hz<=~clock_1Hz;
        end
        else
            conter_1Hz<=conter_1Hz+32'd1;
    end

endmodule
```

时钟模块根据输入的 50MHz 时钟信号输出主时钟信号和一个 1Hz 的复位时钟信号。主时钟信号有两档，一档是 1Hz，还有一档是 1MHz。由 sw_clk，也就是 DE2-70 开发板上的 sw16 控制切换。

2. CPU 框架

```
module cpu(
    input clock_50MHz,

    input [15:0]sw,
    input sw_code_edit,
```

```
    input sw_clk,
    input button_next,
    input button_rst,
    input button_run,
    input button_hlt,
    output [15:0]led,
    output [7:0]led_flag,
    output led_code,
    output led_clk,
    output led_clock,
    output [7:0]seg_0,
    output [7:0]seg_1,
    output [7:0]seg_2,
    output [7:0]seg_3
);

    reg [7:0]code_pointer=8'b0000_0000;
    reg [7:0]mem_code_addr=8'b0000_0000;
    reg [7:0]pc=8'b0000_0000;
    reg flag_hlt_ext=1'b0;

    reg [7:0]mem_data_addr;
    reg [7:0]mem_data_buf_w;

    reg [7:0]alu_a;
    reg [7:0]alu_b;

    reg [7:0]reg_buf_w1;
    reg [7:0]reg_buf_w2;

    reg [7:0]jmp_addr;

    wire reset;

    wire pc_rst;
    wire clk_rst;

    wire clock_en;
    wire clk;
     wire clk_1Hz;
    wire mem_code_en;
    wire [15:0]code;
```

```
wire flag_hlt;
wire a_en;
wire m_en;
wire r_en_r;
wire r_en_w;
wire j_en;
wire [2:0]a_op;
wire [2:0]j_op;
wire mem_rw;
wire [2:0]reg1;
wire [2:0]reg2;
wire [2:0]regw1;
wire [2:0]regw2;
wire [7:0]num;
wire [7:0]alub;
wire [1:0]mem_data_addr_selector;
wire [1:0]mem_data_buf_w_selector;
wire alu_a_selector;
wire [1:0]alu_b_selector;
wire [1:0]reg_buf_w1_selector;
wire [1:0]reg_buf_w2_selector;
wire [1:0]jmp_addr_selector;

wire [7:0]mem_data_buf_r;

wire [7:0]flag;
wire [7:0]jmp_pc_inc;
wire [7:0]jmp_next_pc;

wire [7:0]reg_buf_r1;
wire [7:0]reg_buf_r2;

wire [7:0]alu_e;



reset reset1(
    clk_1Hz,
    button_rst,
    pc_rst,
    clk_rst
);
clock clock_main(
```

```
    sw_clk,
    clock_50MHz,
    clock_en,
    clk,
    clk_1Hz
);
mem_code mem_code1(
    mem_code_en,
    sw_code_edit,
    mem_code_addr,
    sw,
    code
);
decoder decoder1(
    code,
    flag_hlt,
    a_en,
    m_en,
    r_en_r,
    r_en_w,
    j_en,
    a_op,
    j_op,
    mem_rw,
    reg1,
    reg2,
    regw1,
    regw2,
    num,
    alub,
    mem_data_addr_selector,
    mem_data_buf_w_selector,
    alu_a_selector,
    alu_b_selector,
    reg_buf_w1_selector,
    reg_buf_w2_selector,
    jmp_addr_selector
);
mem_data mem_data1(
    clk,
    m_en,
    mem_rw,
    mem_data_addr,
```

```
    mem_data_buf_w,
    sw,
    led,
    mem_data_buf_r
);
jmp jmp1(
    clk,
    j_en,
    j_op,
    pc,
    jmp_addr,
    flag,
    jmp_pc_inc,
    jmp_next_pc
);
register register1(
    clk,
    reg_buf_w1,
    reg_buf_w2,
    r_en_r,
    r_en_w,
    reg1,
    reg2,
    regw1,
    regw2,
    reg_buf_r1,
    reg_buf_r2
);
alu alu1(
    clk,
    a_en,
    alu_a,
    alu_b,
    a_op,
    alu_e,
    flag
);
assign led_clock=clk;
assign led_code=sw_code_edit;
assign led_clk=sw_clk;
assign led_flag=flag;
assign clock_en=~flag_hlt_ext & ~clk_rst;
assign mem_code_en=~button_next;
```

```
    always@(pc or code_pointer or sw_code_edit)//编程模式切换
    begin
        if(sw_code_edit==1'b1)
            mem_code_addr<=code_pointer;
        else
            mem_code_addr<=pc;
    end

    always@(negedge button_next or negedge button_rst)//rst 按钮
    begin
        if(button_rst==1'b0)
            code_pointer<=8'd0;
          else if(button_next==1'b0)
            code_pointer<=code_pointer+8'd1;
    end

    always@(negedge button_run or negedge button_hlt or posedge
sw_code_edit)//运行和停机按钮
    begin
        if(button_run==1'b0 || sw_code_edit==1'b1)
            flag_hlt_ext<=1'b0;
        else if(button_hlt==1'b0)
            flag_hlt_ext<=1'b1;
    end

    always@(posedge clk,posedge pc_rst)//控制 PC
    begin
        if(pc_rst==1'b1)
            pc<=8'b0000_0000;
        else if(clk==1'b1 && flag_hlt==1'b0)
            pc<=jmp_next_pc;
    end

    always@(mem_data_addr_selector or reg_buf_r1 or reg_buf_r2 or num or
alu_e)
    begin
        case(mem_data_addr_selector)
            2'b00:mem_data_addr=reg_buf_r1;
            2'b01:mem_data_addr=reg_buf_r2;
            2'b10:mem_data_addr=num;
            2'b11:mem_data_addr=alu_e;
        endcase
```

```
    end

    always@(mem_data_buf_w_selector or jmp_pc_inc or reg_buf_r1 or
reg_buf_r2 or num)
    begin
        case(mem_data_buf_w_selector)
            2'b00:mem_data_buf_w=jmp_pc_inc;
            2'b01:mem_data_buf_w=reg_buf_r1;
            2'b10:mem_data_buf_w=reg_buf_r2;
            2'b11:mem_data_buf_w=num;
        endcase
    end

    always@(alu_a_selector or reg_buf_r1 or reg_buf_r2)
    begin
        case(alu_a_selector)
            1'b0:alu_a=reg_buf_r1;
            1'b1:alu_a=reg_buf_r2;
        endcase
    end

    always@(alu_b_selector or reg_buf_r1 or reg_buf_r2 or num or alub)
    begin
        case(alu_b_selector)
            2'b00:alu_b=reg_buf_r1;
            2'b01:alu_b=reg_buf_r2;
            2'b10:alu_b=num;
            2'b11:alu_b=alub;
        endcase
    end

    always@(reg_buf_w1_selector or mem_data_buf_r or num or alu_e or
reg_buf_r1)
    begin
        case(reg_buf_w1_selector)
            2'b00:reg_buf_w1=mem_data_buf_r;
            2'b01:reg_buf_w1=num;
            2'b10:reg_buf_w1=alu_e;
            2'b11:reg_buf_w1=reg_buf_r1;
        endcase
    end
```

```
    always@(reg_buf_w2_selector or mem_data_buf_r or num or alu_e or
reg_buf_r2)
    begin
        case(reg_buf_w2_selector)
            2'b00:reg_buf_w2=mem_data_buf_r;
            2'b01:reg_buf_w2=num;
            2'b10:reg_buf_w2=alu_e;
            2'b11:reg_buf_w2=reg_buf_r2;
        endcase
    end

    always@(jmp_addr_selector or mem_data_buf_r or num or reg_buf_r1 or
reg_buf_r2)
    begin
        case(jmp_addr_selector)
            2'b00:jmp_addr=mem_data_buf_r;
            2'b01:jmp_addr=num;
            2'b10:jmp_addr=reg_buf_r1;
            2'b11:jmp_addr=reg_buf_r2;
        endcase
    end
endmodule
```

CPU 总框架模块用来连接各个子模块，并控制他们之间的数据传输。另一方面 CPU 总框架模块还连接了 DE2-70 开发板上的开关和按钮与程序。控制整个 CPU 的运行。 通过选择子来选择不同模块的功能。

第三章 系统指令集

3.1 数据传送类指令

数据传送在任何程序中都是最基本、最常用也是最重要的操作指令，它在程序中使用频率最高。18386 的数据传送类指令可以满足立即数与寄存器、寄存器与寄存器、直接寻址下存储器与寄存器、寄存器间接寻址下存储器与寄存器之间的数据传送。数据传送类指令的机器码如表 3-1 所示

|指令集选择|指令集选择|指令|reg1 or num|reg2 or num|

|0|1|000|xxx|xxxxxxxx| mov reg,num

|0|1|001|xxx|xxx|?????|　　mov reg,reg

|0|1|010|xxx|xxxxxxxx|　　mov reg,[num]

|0|1|011|xxx|xxx|?????|　　mov reg,[reg]

|0|1|110|xxx|xxxxxxxx|　　mov [num],reg

|0|1|111|xxx|xxx|?????|　　mov [reg],reg

表 3-1 数据传送类指令机器码与指令对照表

第一位和第二位作为指令集的选择，第一位为 1 代表选择了 ALU 的算术和逻辑运算指令集，第一位为 0 代表选择了转移和传送类指令集，第二位为 1 代表选择了转移和传送类指令集中的传送类指令集，第二位为 0 则代表选择了转移类指令集。第三位代表指令，数据传送类指令的机器码与指令对应如上可知。如 000 对应的指令目的操作数是寄存器，源操作数是立即数，该指令执行效果为将源操作数输入的立即数送入目的操作数所代表的寄存器，第四位代表目的操作数，第五位代表源操作数。‘X’表示待输入的内容，‘？’表示无关的内容。

3.2 运算类指令

运算类指令包含算术运算类指令、逻辑运算类指令、移位指令，算术运算类指令受限于时间限制，我们只制作了加减，乘除很多情况下可以用加减指令替代。逻辑运算类指令我们制作了与、或、非、异或；移位指令我们制作了 SHL、SHR 逻辑左移和逻辑右移指令。运算类指令机器码如下表 3-2 所示。

|指令集选择|指令集选择|指令|reg1|num|

|1|0|000|xxx|xxxxxxxx|　　add reg,num

|1|0|001|xxx|xxxxxxxx|　　sub reg,num cmp reg,num

|1|0|010|xxx|xxxxxxxx|　　and reg,num test reg,num

|1|0|011|xxx|xxxxxxxx|　　or reg,numz

|1|0|101|xxx|xxxxxxxx|　　xor reg,num

|1|0|110|xxx|xxxxxxxx|　　shl reg,num

|1|0|111|xxx|xxxxxxxx|　　shr reg,num

|指令集选择|指令集选择|指令|reg1|reg2|

|1|1|000|xxx|xxx|?????|　　add reg,reg

|1|1|001|xxx|xxx|?????|　　sub reg,reg cmp reg,reg

|1|1|010|xxx|xxx|?????|　　and reg,reg test reg,reg

|1|1|011|xxx|xxx|?????|　　or reg,reg

|1|1|100|xxx|xxx|?????|　　not reg,reg

|1|1|101|xxx|xxx|?????|　　xor reg,reg

|1|1|110|xxx|xxx|?????|　　shl reg,reg

|1|1|111|xxx|xxx|?????|　　shr reg,reg

表 3-2 运算类指令机器码与指令对照表

第一位和第二位作为指令集的选择，第一位为 1 代表选择了 ALU 的算术和逻辑运算指令集；第二位与操作数有关，第二位为 0 时选择源操作数为立即数，第二位为 1 时选择源操作数为寄存器，第三位为操作码，操作码与机器码对应如下表 3-3 所示。第四位为目的操作数，第五位为源操作数。由于通用寄存器寻址只有个三位，所以表 3-2 中下半部分寄存器作为源操作数时只有前三位为有效位，后五为‘？’表示无效。

000	001	010	011	100	101	110	111
ADD	SUB	AND	OR	NOT	XOR	SHL	SHR

表 3-3 ALU 指令机器码对照表

3.3 控制转移类指令

计算机一般是顺序执行，但有的时候需要根据不同条件去做不同的处理，有时候需要跳过几条指令，有时候需要重复执行某段程序，或者转移到另一段程序中。控制转移类指令对于计算机来说也是必不可少的部分，18386 中包含了 JMP、JE、JNE、JA、JB、JC、JNC、CALL、RET、HLT、NOP 指令，PUSH、POP 指令在 18386 中也归类为控制转移类指令。控制转移类指令机器码与指令对照如下表 3-4。

|指令集选择|指令集选择||指令|reg1|reg2 or num|

|0|0|00|000|?|????????|　　nop

|0|0|00|001|?|????????|　　hlt

|0|0|01|000|0|xxxxxxxx|　　push num

|0|0|01|000|1|xxx|?????|　　push reg

|0|0|01|001|?|xxx|?????|　　pop reg

|0|0|10|000|0|????????| call num

|0|0|10|000|1|xxx|?????| call reg

|0|0|10|001|?|????????| ret

|0|0|11|000|0|xxxxxxxx| jmp num

|0|0|11|000|1|xxx|?????| jmp reg

|0|0|11|001|0|xxxxxxxx| je num

|0|0|11|001|1|xxx|?????| je reg

|0|0|11|010|0|xxxxxxxx| jne num

|0|0|11|010|1|xxx|?????| jne reg

|0|0|11|011|0|xxxxxxxx| ja num

|0|0|11|011|1|xxx|?????| ja reg

|0|0|11|100|0|xxxxxxxx| jb num

|0|0|11|100|1|xxx|?????| jb reg

|0|0|11|101|0|xxxxxxxx| jc num

|0|0|11|101|1|xxx|?????| jc reg

|0|0|11|110|0|xxxxxxxx| jnc num

|0|0|11|110|1|xxx|?????| jnc reg

表 3-4 控制转移类指令机器码与指令对照

第一位和第二位作为指令集的选择。第一位 0 表示选择非 ALU 指令集；第二位 0 表示选择控制转移类指令集；第三位 00 表示选择 NOP 空指令、HLT 停机指令，01 表示选择 PUSH、POP 指令、10 表示选择 CALL、RET 指令、11 表示选择 JMP 类指令，机器码与指令对应见表 3-5；第四位表示操作位指令码；第五位 1 表示操作数寄存器，0 表示操作数立即数；第六位为输入的内容。

机器码	00000	00001	01000	01001	10000	10001	11000
指令	NOP	HLT	PUSH	POP	CALL	RET	JMP
11001	11010	11011	11100	11101	11110		
JE	JNE	JA	JB	JC	JNC		

表 3-5 控制转移类指令对照表

第四章 程序编写与执行结果

4.1 程序编写

为了测试 cpu 功能，编写测试代码如下:

```
mov ps,f0h        ;初始化栈顶指针
mov r0,00h        ;r0 清零
add r0,01h        ;r0+1
mov [ffh],r0      ;显示 r0 到 led 灯组
call 8d           ;delay(1.8s)
jmp 2d            ;跳转到第三行继续执行
nop               ;空指令
nop               ;空指令
push r0           ;保护 r0
push r1           ;保护 r1
push r2           ;保护 r2
mov r2,200d       ;r2=200
mov r1,200d       ;r1=200
mov r0,22d        ;r0=22
sub r0,1d         ;r0-1
jnz 14d           ;若 r0 不为 0,跳转到第 15 行
sub r1,1d         ;r1-1
jnz 13d           ;若 r1 不为 0,跳转到第 14 行
sub r2,1d         ;r2-1
jnz 12d           ;若 r2 不为 0,跳转到第 13 行
pop r2            ;恢复 r2
pop r1            ;恢复 r1
pop r0            ;恢复 r0
ret               ;回到上一层函数
```

转换为机器码如下:

```
0100011011110000 //mov ps,f0h
0100000000000000 //mov r0,00h
1000000000000001 //add r0,01h
0111000011111111 //mov [ffh],r0
0010000000001000 //call 8d
0011000000000010 //jmp 2d
0000000000000000 //nop
0000000000000000 //nop
```

```
0001000100000000 //push r0
0001000100100000 //push r1
0001000101000000 //push r2
0100001011001000 //mov r2,200d
0100000111001000 //mov r1,200d
0100000000010110 //mov r0,22d
1000100000000001 //sub r0,1d
0011010000001110 //jnz 14d
1000100100000001 //sub r1,1d
0011010000001101 //jnz 13d
1000101000000001 //sub r2,1d
0011010000001100 //jnz 12d
0001001001000000 //pop r2
0001001000100000 //pop r1
0001001000000000 //pop r0
0010001000000000 //ret
```

4.2 结果展示

如图所示，cpu 成功在 1MHz 主频下执行了给定代码，暂未测试更高的主频

结论

本次课程设计的 CPU 完成了上述内容的构建，所设计的 CPU 可以完成上述的指令，我们在 code.bin 文件中对指令逐个进行了测试，均能完美完成。

心得体会

很早以前，在学习 x86 汇编语言的时候就有想过自己制作一个简单的 cpu，所以在得知该课程结课作业可以自选题目时，我第一个想到的便是自制一个 cpu。经过我和 XXX 同学接近一个周的努力，在历经了漫长的 Debug 和测试过程之后，我们的 cpu 终于成功运行。尽管这个 cpu 的功能十分简单，只能够执行少量的指令集，而且因为没有设计流水线结构，也注定了它的执行速度不会太快，但依旧是我们努力的结果。

在制作的过程中，我们也遇到了许多的问题，比如因为对 verilog 语言特性的不了解，我们的一些线路被生成了锁存器，导致时序错误，最后排查了好久才发现，事实上在编译产生的警告中已经提示了我们的某些语句生成了锁存器，但我们觉得有警告正常，所以没仔细看，也因此浪费了许多时间。另外，由于我在设计指令集的时候经验不足，一些本该实现的指令没有实现，比如 CMP 指令和 TEXT 指令。这样的错误还有很多很多，这里也就不值得一一列出。

总之，通过这次实验，我更深入的认识了硬件描述语言的特性，也是对这个课程所学知识的一次完美应用。

参考资料

[1]兰德尔 E 布莱恩特 大卫 R 奥哈拉论.深入理解计算机系统 [M].北京:机械工业出版社,2016:245-282

[2]魏家明.Verilog 编程艺术 [M].北京:电子工业出版社,2014

[3]周荷琴 冯焕清.微信计算机原理与接口技术[M].安徽:中国科学技术大学出版社.1996

[4]黄勤.微信计算机原理与接口技术[M].北京:机械工业出版社.2015

[5] 深入探討 DE2-70 的『Error：Can't place pins assigned to pin location Pin_AD25 (IOC_X95_Y2_N1)』錯誤訊息的原因與解決方式[I]. https://blog.csdn.net/Silent_Majority/article/details/72829524,2017

[6] 如何解決 DE2-70 的『Error: Can't place pins assigned to pin location Pin_AD25 (IOC_X95_Y2_N1)』的錯誤訊息? (SOC) (Quartus II) (DE2-70) [I]. https://www.cnblogs.com/oomusou/archive/2008/10/21/de2_70_pin_ad25.html

附录

附录一

指令	flag_hlt	a_en	m_en	r_en_r	r_en_w	j_en	[2:0]a_op
nop	0	0	0	0	0	0	???
hlt	1	0	0	0	0	0	???
push num	0	0	1	1	1	0	001
push reg	0	0	1	1	1	0	001
pop reg	0	0	1	1	1	0	000
call num	0	0	1	1	1	1	001
call reg	0	0	1	1	1	1	001
ret	0	0	1	1	1	1	000
jmp num	0	0	0	0	0	1	???
jmp reg	0	0	0	1	0	1	???
je num	0	0	0	0	0	1	???
je reg	0	0	0	1	0	1	???
jne num	0	0	0	0	0	1	???
jne reg	0	0	0	1	0	1	???
ja num	0	0	0	0	0	1	???
ja reg	0	0	0	1	0	1	???
jb num	0	0	0	0	0	1	???
jb reg	0	0	0	1	0	1	???
jc num	0	0	0	0	0	1	???
jc reg	0	0	0	1	0	1	???
jnc num	0	0	0	0	0	1	???
jnc reg	0	0	0	1	0	1	???
mov reg,num	0	0	0	0	1	0	???
mov reg,reg	0	0	0	1	1	0	???
mov reg,[num]	0	0	1	0	1	0	???
mov reg,[reg]	0	0	1	1	1	0	???
mov [num],reg	0	0	1	1	0	0	???
mov [reg],reg	0	0	1	1	0	0	???

add reg,num	0	1	0	1	1	0	xxx
sub reg,num cmp reg,num	0	1	0	1	1	0	xxx
and reg,num test reg,num	0	1	0	1	1	0	xxx
or reg,num	0	1	0	1	1	0	xxx
not reg	0	1	0	1	1	0	xxx
xor reg,num	0	1	0	1	1	0	xxx
shl reg,num	0	1	0	1	1	0	xxx
shr reg,num	0	1	0	1	1	0	xxx
add reg,reg	0	1	0	1	1	0	xxx
sub reg,reg cmp reg,reg	0	1	0	1	1	0	xxx
and reg,reg test reg,reg	0	1	0	1	1	0	xxx
or reg,reg	0	1	0	1	1	0	xxx
not reg,reg	0	1	0	1	1	0	xxx
xor reg,reg	0	1	0	1	1	0	xxx
shl reg,reg	0	1	0	1	1	0	xxx
shr reg,reg	0	1	0	1	1	0	xxx

指令	[2:0]j_op	mem_rw	[2:0]reg1	[2:0]reg2	[2:0]regw1
nop	???	?	111	111	111
hlt	???	?	111	111	111
push num	???	1	110	111	110
push reg	???	1	110	xxx	110
pop reg	???	0	110	111	110
call num	000	1	110	111	110
call reg	000	1	110	xxx	110
ret	000	0	110	111	110
jmp num	xxx	?	111	111	111
jmp reg	xxx	?	xxx	111	111
je num	xxx	?	111	111	111
je reg	xxx	?	xxx	111	111
jne num	xxx	?	111	111	111
jne reg	xxx	?	xxx	111	111
ja num	xxx	?	111	111	111
ja reg	xxx	?	xxx	111	111
jb num	xxx	?	111	111	111
jb reg	xxx	?	xxx	111	111
jc num	xxx	?	111	111	111
jc reg	xxx	?	xxx	111	111
jnc num	xxx	?	111	111	111
jnc reg	xxx	?	xxx	111	111
mov reg,num	???	?	111	111	xxx
mov reg,reg	???	?	xxx	111	xxx
mov reg,[num]	???	0	111	111	xxx
mov reg,[reg]	???	0	xxx	111	xxx
mov [num],reg	???	1	xxx	111	111
mov [reg],reg	???	1	xxx	xxx	111
add reg,num	???	?	xxx	111	xxx
sub reg,num cmp reg,num	???	?	xxx	111	xxx
and reg,num test reg,num	???	?	xxx	111	xxx
or reg,num	???	?	xxx	111	xxx

not reg	???	?	xxx	111	xxx
xor reg,num	???	?	xxx	111	xxx
shl reg,num	???	?	xxx	111	xxx
shr reg,num	???	?	xxx	111	xxx
add reg,reg	???	?	xxx	xxx	xxx
sub reg,reg cmp reg,reg	???	?	xxx	xxx	xxx
and reg,reg test reg,reg	???	?	xxx	xxx	xxx
or reg,reg	???	?	xxx	xxx	xxx
not reg,reg	???	?	xxx	xxx	xxx
xor reg,reg	???	?	xxx	xxx	xxx
shl reg,reg	???	?	xxx	xxx	xxx
shr reg,reg	???	?	xxx	xxx	xxx

指令	[2:0]regw2	[7:0]num	[7:0]alub	[1:0]mem_data_addr_selector
nop	111	????????	????????	??
hlt	111	????????	????????	??
push num	111	xxxxxxxx	00000001	11
push reg	111	????????	00000001	11
pop reg	xxx	????????	00000001	00
call num	111	xxxxxxxx	00000001	11
call reg	111	????????	00000001	11
ret	111	????????	00000001	00
jmp num	111	xxxxxxxx	????????	??
jmp reg	111	????????	????????	??
je num	111	xxxxxxxx	????????	??
je reg	111	????????	????????	??
jne num	111	xxxxxxxx	????????	??
jne reg	111	????????	????????	??
ja num	111	xxxxxxxx	????????	??
ja reg	111	????????	????????	??
jb num	111	xxxxxxxx	????????	??
jb reg	111	????????	????????	??
jc num	111	xxxxxxxx	????????	??
jc reg	111	????????	????????	??
jnc num	111	xxxxxxxx	????????	??
jnc reg	111	????????	????????	??
mov reg,num	111	xxxxxxxx	????????	??
mov reg,reg	111	????????	????????	??
mov reg,[num]	111	xxxxxxxx	????????	10
mov reg,[reg]	111	????????	????????	00
mov [num],reg	111	xxxxxxxx	????????	10
mov [reg],reg	111	????????	????????	00
add reg,num	111	xxxxxxxx	????????	??
sub reg,num cmp reg,num	111	xxxxxxxx	????????	??
and reg,num test reg,num	111	xxxxxxxx	????????	??
or reg,num	111	xxxxxxxx	????????	??

not reg	111	xxxxxxxx	????????	??
xor reg,num	111	xxxxxxxx	????????	??
shl reg,num	111	xxxxxxxx	????????	??
shr reg,num	111	xxxxxxxx	????????	??
add reg,reg	111	????????	????????	??
sub reg,reg cmp reg,reg	111	????????	????????	??
and reg,reg test reg,reg	111	????????	????????	??
or reg,reg	111	????????	????????	??
not reg,reg	111	????????	????????	??
xor reg,reg	111	????????	????????	??
shl reg,reg	111	????????	????????	??
shr reg,reg	111	????????	????????	??

指令	[1:0]mem_data_buf_w_selector	alu_a_selector
nop	??	?
hlt	??	?
push num	11	0
push reg	10	0
pop reg	00	0
call num	00	0
call reg	00	0
ret	00	0
jmp num	??	?
jmp reg	??	?
je num	??	?
je reg	??	?
jne num	??	?
jne reg	??	?
ja num	??	?
ja reg	??	?
jb num	??	?
jb reg	??	?
jc num	??	?
jc reg	??	?
jnc num	??	?
jnc reg	??	?
mov reg,num	??	?
mov reg,reg	??	?
mov reg,[num]	00	?
mov reg,[reg]	00	?
mov [num],reg	01	?
mov [reg],reg	10	?
add reg,num	??	0
sub reg,num cmp reg,num	??	0
and reg,num test reg,num	??	0
or reg,num	??	0
not reg	??	0

xor reg,num	??	0
shl reg,num	??	0
shr reg,num	??	0
add reg,reg	??	0
sub reg,reg cmp reg,reg	??	0
and reg,reg test reg,reg	??	0
or reg,reg	??	0
not reg,reg	??	0
xor reg,reg	??	0
shl reg,reg	??	0
shr reg,reg	??	0

指令	[1:0]alu_b_selector	[1:0]reg_buf_w1_selector
nop	??	??
hlt	??	??
push num	11	10
push reg	11	10
pop reg	11	10
call num	11	10
call reg	11	10
ret	11	10
jmp num	??	??
jmp reg	??	??
je num	??	??
je reg	??	??
jne num	??	??
jne reg	??	??
ja num	??	??
ja reg	??	??
jb num	??	??
jb reg	??	??
jc num	??	??
jc reg	??	??
jnc num	??	??
jnc reg	??	??
mov reg,num	??	01
mov reg,reg	??	11
mov reg,[num]	??	00
mov reg,[reg]	??	00
mov [num],reg	??	??
mov [reg],reg	??	??
add reg,num	10	10
sub reg,num cmp reg,num	10	10
and reg,num test reg,num	10	10
or reg,num	10	10

not reg	10	10
xor reg,num	10	10
shl reg,num	10	10
shr reg,num	10	10
add reg,reg	01	10
sub reg,reg cmp reg,reg	01	10
and reg,reg test reg,reg	01	10
or reg,reg	01	10
not reg,reg	01	10
xor reg,reg	01	10
shl reg,reg	01	10
shr reg,reg	01	10

指令	[1:0]reg_buf_w2_selector	[1:0]jmp_addr_selector
nop	??	??
hlt	??	??
push num	??	??
push reg	??	??
pop reg	00	??
call num	??	01
call reg	??	11
ret	??	00
jmp num	??	01
jmp reg	??	10
je num	??	01
je reg	??	10
jne num	??	01
jne reg	??	10
ja num	??	01
ja reg	??	10
jb num	??	01
jb reg	??	10
jc num	??	01
jc reg	??	10
jnc num	??	01
jnc reg	??	10
mov reg,num	??	??
mov reg,reg	??	??
mov reg,[num]	??	??
mov reg,[reg]	??	??
mov [num],reg	??	??
mov [reg],reg	??	??
add reg,num	??	??
sub reg,num cmp reg,num	??	??
and reg,num test reg,num	??	??
or reg,num	??	??

not reg	??	??
xor reg,num	??	??
shl reg,num	??	??
shr reg,num	??	??
add reg,reg	??	??
sub reg,reg cmp reg,reg	??	??
and reg,reg test reg,reg	??	??
or reg,reg	??	??
not reg,reg	??	??
xor reg,reg	??	??
shl reg,reg	??	??
shr reg,reg	??	??