哈爾濱工業大學

计算机系统

大作业

题　　目 Y86-64 顺序处理器的 Verilog 实现

专　　业 计算机类

学　　号 1160300903

班　　级 1603009

学　　生 王昭为

指 导 教 师 吴锐

计算机科学与技术学院

2017 年 12 月

目 录

第 1 章 任务基本信息

1.1 作业要求

用 Verilog 实现 Y86-64 的顺序结构-SEQ，要求：

1、 在现有 Y86-64 ISA 基础上增加 iaddq 指令，即 ISA+；

2、 利用一种硬件描述语言如 Verilog，设计能够支持 ISA+的 SEQ；

3、 进行模拟验证；

4、 进行逻辑综合（选做，加分项目）

提示：可参照“Verilog Implementation of a Pipelined Y86-64 Processor”文档（waside-verilog.pdf），写出 SEQ 的 Verilog 实现。

1.2 实验环境与工具

1、Vivado

2、ModelSIM

第 2 章 预备知识

（该章满分 30 分）

2.1 给出 ISA+的指令编码设计

指令	icode 4bit	ifun 4bit	rA 4bit	rB 4bit	valC 64bit
halt	0	0			
nop	1	0			
rrmovq **rA,rB**	2	0	rA	rB	
irmovq **V,rB**	3	0	F	rB	V
rmmovq **rA,D(rB)**	4	0	rA	rB	D
mrmovq **D(rB),rA**	5	0	rA	rB	D
OPq **rA,rB**	6	fn	rA	rB	
jXX **Dest**	7	fn			Dest
cmovxx **rA,rB**	2	fn	rA	rB	
call **Dest**	8	0			Dest
ret	9	0			
pushq **rA**	a	0	rA	F	
popq **rA**	b	0	rA	F	
iaddq **V,rB**	c	0	F	rB	V

整数操作指令		
addq	6	0
subq	6	1
andq	6	2
xorq	6	3

分支操作指令			传送指令		
jmp	7	0	rrmovq	2	0
jle	7	1	cmovle	2	1
jl	7	2	cmovl	2	2
je	7	3	cmove	2	3
jne	7	4	cmovne	2	4
jge	7	5	cmovge	2	5
jg	7	6	cmovg	2	6

2.2 写出指令 iaddq 在 SEQ 中的计算过程

阶段	通用
	iaddq
取指	icode:ifun ← M1[pc] F : rB ← M1[pc+1] valC ← M8[pc+2] valP = pc+10
译码	valA ← valC valB ← R[rB]
执行	valE ← valB+valA
访存	
写回	R[rB] ← valE
更新 pc	pc ← valP

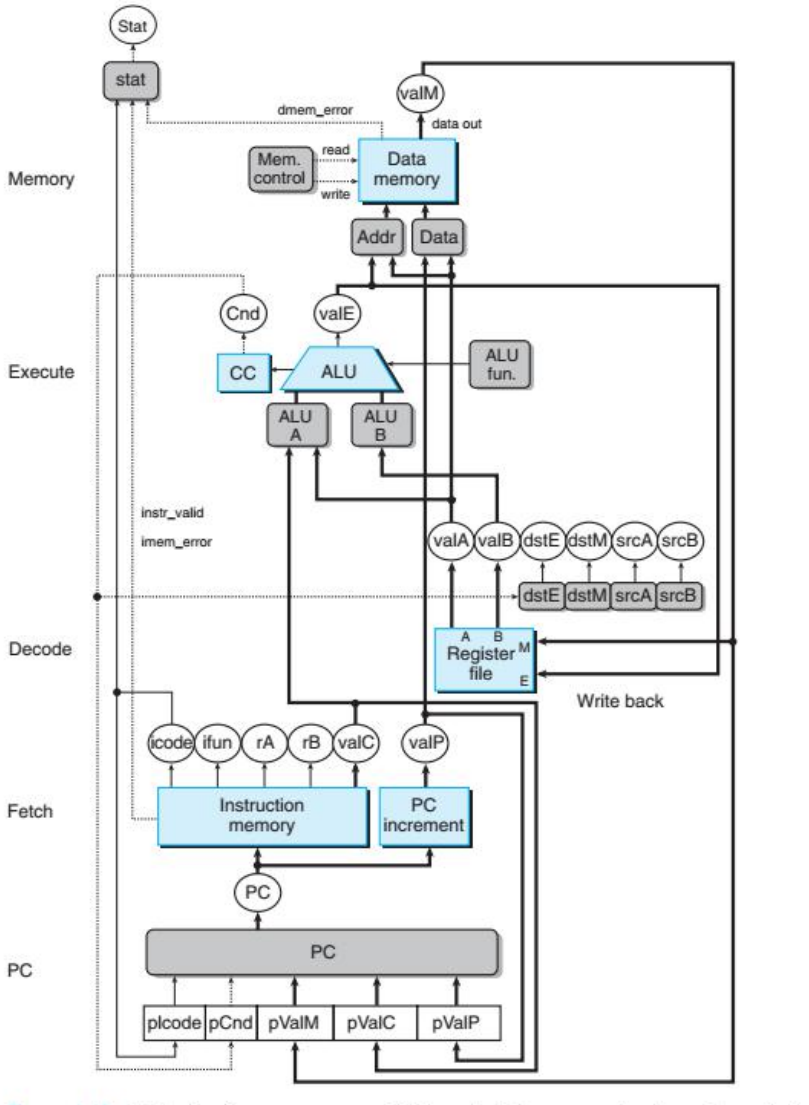
2.3 使用 iaddq 指令重写教材图 4-6 中的 sum 函数

```
sum:
    xorq %rax,%rax
    and %rsi,%rsi
    jmp test
  loop:
    mrmovq (%rdi),%r10
    addq %r10,%rax
    iaddq $8,%rdi
    iaddq -$1,%rsi
  test:
    jne loop
    ret
```

第 3 章 Y86-64 顺序处理器设计

（该章满分 50 分）

3.1 Y86-64 顺序处理器结构设计



3.2 Y86-64 顺序处理器模块设计（包含 Verilog 模块接口设计）

3.2.1 时钟控制同步复位寄存器

1.基本功能说明：该模块在 reg 型变量的基础上进行了修改和包装，由同步时钟控制输入数据。数据输出为组合逻辑，不需要时钟控制。该模块带有同步复位端 reset，reset 为 1 时且时钟上升沿来临时，寄存器的值将变成 reVal 的值。

2.主要端口定义：

in(输入)	**寄存器输入**
out(输出)	**寄存器输出**
reset(输入)	**同步复位端**
reVal(输入)	**同步复位值**
clock(输入)	**时钟**
set(输入)	**更新寄存器的使能信号**

3. 时钟时序：

时钟上升沿的时候更新寄存器

4.代码：

```
module cenrreg(in,out,set,reset,reVal,clock);//同步清零的寄存器
    parameter width = 8;//寄存器宽度
    input[width-1:0] in;//更新寄存器的输入
    input set;//更新寄存器的使能信号
    input reset;//复位信号
    input clock;//时钟
    output[width-1:0] out;//寄存器输出
    reg [width-1:0] out;
    input [width-1:0] reVal;
    always@(posedge clock)//时钟上升沿更新
    begin
        if(set)//如果设置为 1
            out <= in;
        if(reset)//如果复位为 1
            out <= reVal;
     end
endmodule
```

3.2.2 寄存器文件模块

1. **基本功能说明：**

1）该模块模拟 cpu 的寄存器文件，由 clock 时钟控制，reset 信号为同步清零端，当 reset 为 1，且时钟上升沿来临时，寄存器文件中的所有寄存器都被清零。srcA,srcB,分别为两个寄存器的标识符，valA,valB 为 srcA，srcB 对应寄存器的值，该输入为组合逻辑，不需要时钟控制。dstE 和 dstM 为两个要更新的寄存器的标识符，更新

的值对应为 valE 和 valM，valE 为执行阶段的计算结果，valM 为访存阶段的结果。rax，rcx，rdx，rdx，rbx，rsp 等寄存器输出端口，用于在 cpu 处于 STATUS_MODE 时输出寄存器的值。

2）对于模块中所用到的参数，说明如下

valSize：用于声明 rax 值的位数

regAddr ：用于声明寄存器 id 的位数

IRAX ：用于声明寄存器 id，其他 15 参数不一一列举

3） 下面语句调用 cenrreg 模块声明了 rax 寄存器，其他 15 个寄存器也是如此

```
cenrreg #(64) raxReg(raxData,rax,raxW,reset,1'b0,clock);
```

2.主要端口定义：

srcA(输入)	**寄存器读端口**
srcB(输入)	**寄存器读端口**
valA(输出)	**寄存器输出**
valB(输出)	**寄存器输出**
dstE(输入)	**寄存器写操作地址**
dstM(输入)	**寄存器写操作地址**
valE(输入)	**寄存器写操作数据**
valM(输入)	**寄存器写操作数据**
reset(输入)	**同步清零**
clock(输入)	**时钟**

3.时序说明：

时钟上升沿的时候更新寄存器

4. 代码

```
module
registerFile(srcA,srcB,valA,valB,dstE,valE,dstM,valM,reset,clock,rax,rcx,rdx,rbx,
rsp,rbp,rsi,rdi,r8,r9,r10,r11,r12,r13,r14);
    //寄存器的 id 位宽，和寄存器的位宽
    parameter valSize = 64;
    parameter regAddr = 4;
    //规定寄存器参数
    parameter IRAX = 4'h0;
    parameter IRCX = 4'h1;
    parameter IRDX = 4'h2;
    parameter IRBX = 4'h3;
    parameter IRSP = 4'h4;
    parameter IRBP = 4'h5;
    parameter IRSI = 4'h6;
    parameter IRDI = 4'h7;
    parameter IR8 = 4'h8;
    parameter IR9 = 4'h9;
    parameter IRA = 4'ha;
```

```
    parameter IRB = 4'hb;
    parameter IRC = 4'hc;
    parameter IRD = 4'hd;
    parameter IRE = 4'he;
    parameter RNONE = 4'hf;
    //端口声明
    output[valSize-1:0]rax,rcx,rdx,rbx,rsp,rbp,rdi,rsi,r8,r9,r10,r11,r12,r13,r14;
    input[regAddr-1:0] srcA;
    input[regAddr-1:0] srcB;
    output[valSize-1:0]valA;
    output[valSize-1:0]valB;
    input[regAddr-1:0]dstE;
    input[regAddr-1:0]dstM;
    input[valSize-1:0]valE;
    input[valSize-1:0]valM;
    input reset;
    input clock;
    //声明寄存器输入数据端

wire[valSize-1:0]raxData,rbxData,rcxData,rdxData,rspData,rbpData,rsiData,rdiData,
r8Data,r9Data,r11Data,r12Data,r13Data,r14Data,r10Data;
    //声明寄存器写信号
    wire
raxW,rbxW,rcxW,rdxW,rspW,rbpW,rsiW,rdiW,r8W,r9W,r10W,r11W,r12W,r13W,r14W;
    //调用 cenrreg 模块声明 15 个寄存器
    cenrreg #(64) raxReg(raxData,rax,raxW,reset,64'b0,clock);
    cenrreg #(64) rbxReg(rbxData,rbx,rbxW,reset,64'b0,clock);
    cenrreg #(64) rcxReg(rcxData,rcx,rcxW,reset,64'b0,clock);
    cenrreg #(64) rdxReg(rdxData,rdx,rdxW,reset,64'b0,clock);
    cenrreg #(64) rspReg(rspData,rsp,rspW,reset,64'b1000,clock);
    cenrreg #(64) rbpReg(rbpData,rbp,rbpW,reset,64'b0,clock);
    cenrreg #(64) rsiReg(rsiData,rsi,rsiW,reset,64'b0,clock);
    cenrreg #(64) rdiReg(rdiData,rdi,rdiW,reset,64'b0,clock);
    cenrreg #(64) r8Reg(r8Data,r8,r8W,reset,64'b0,clock);
    cenrreg #(64) r9Reg(r9Data,r9,r9W,reset,64'b0,clock);
    cenrreg #(64) raReg(r10Data,r10,r10W,reset,64'b0,clock);
    cenrreg #(64) rbReg(r11Data,r11,r11W,reset,64'b0,clock);
    cenrreg #(64) rcReg(r12Data,r12,r12W,reset,64'b0,clock);
    cenrreg #(64) rdReg(r13Data,r13,r13W,reset,64'b0,clock);
    cenrreg #(64) reReg(r14Data,r14,r14W,reset,64'b0,clock);
    //选择 valA 的值
    assign valA = srcA==IRAX ? rax:
                  srcA==IRDX ? rdx:
                  srcA==IRCX ? rcx:
                  srcA==IRBX ? rbx:
                  srcA==IRSP ? rsp:
                  srcA==IRBP ? rbp:
```

```
                srcA==IRSI ? rsi:
                srcA==IRDI ? rdi:
                srcA==IR8 ? r8:
                srcA==IR9 ? r9:
                srcA==IRA ? r10:
                srcA==IRB ? r11:
                srcA==IRC ? r12:
                srcA==IRD ? r13:
                srcA==IRE ? r14:
                0;
//选择 valB 的值
assign valB = srcB==IRAX ? rax:
                srcB==IRDX ? rdx:
                srcB==IRCX ? rcx:
                srcB==IRBX ? rbx:
                srcB==IRSP ? rsp:
                srcB==IRBP ? rbp:
                srcB==IRSI ? rsi:
                srcB==IRDI ? rdi:
                srcB==IR8 ? r8:
                srcB==IR9 ? r9:
                srcB==IRA ? r10:
                srcB==IRB ? r11:
                srcB==IRC ? r12:
                srcB==IRD ? r13:
                srcB==IRE ? r14:
                0;
//对寄存器写的输入端赋值
assign raxData = dstM == IRAX ? valM:valE;
assign rdxData = dstM == IRDX ? valM:valE;
assign rcxData = dstM == IRCX ? valM:valE;
assign rbxData = dstM == IRBX ? valM:valE;
assign rspData = dstM == IRSP ? valM : valE;
assign rbpData = dstM == IRBP ? valM : valE;
assign rsiData = dstM == IRSI ? valM : valE;
assign rdiData = dstM == IRDI ? valM : valE;
assign r8Data = dstM == IR8 ? valM : valE;
assign r9Data = dstM == IR9 ? valM : valE;
assign r10Data = dstM == IRA ? valM : valE;
assign r11Data = dstM == IRB ? valM : valE;
assign r12Data = dstM == IRC ? valM : valE;
assign r13Data = dstM == IRD ? valM : valE;
assign r14Data = dstM == IRE ? valM : valE;
//对寄存器写的使能端赋值
 assign raxW = dstM == IRAX | dstE == IRAX;
   assign rcxW = dstM == IRCX | dstE == IRCX;
   assign rdxW = dstM == IRDX | dstE == IRDX;
```

```
        assign rbxW = dstM == IRBX | dstE == IRBX;
        assign rspW = dstM == IRSP | dstE == IRSP;
        assign rbpW = dstM == IRBP | dstE == IRBP;
        assign rsiW = dstM == IRSI | dstE == IRSI;
        assign rdiW = dstM == IRDI | dstE == IRDI;
        assign r8W = dstM == IR8 | dstE == IR8;
        assign r9W = dstM == IR9 | dstE == IR9;
        assign r10W = dstM == IRA | dstE == IRA;
        assign r11W = dstM == IRB | dstE == IRB;
        assign r12W = dstM == IRC | dstE == IRC;
        assign r13W = dstM == IRD | dstE == IRD;
        assign r14W = dstM == IRE | dstE == IRE;
endmodule
```

3.2.3 内存芯片模块

1. 基本功能说明：

1）该模块为内存中的一个内存芯片。它有两个既可以读又可以写的端口，地址输入分别为 addrA，addrB，写的使能端分别为 wEnA，wEnB，写的数据输入分别为 wDataA,wDataB。

2）在本模块中用到了几个参数，它们的作用如下

WORDSIZE ：声明访存时地址和数据的长度

WORDNUM ：声明 ram 中的字节数

ADDRSIZE ：声明 ram 中字节的地址长度

3）时钟下降沿同步控制数据的输入输出，如果 wEnA 有效则写入 wDataA，如果 rEnA 有效则读出 addrA 的内容。端口 B 与端口 A 完全相同。

```
 always@(negedge clock)
     begin
         if(wEnA)
             begin
                 mem[addrA] <= wDataA;
             end
         if(rEnA)
             begin
                 rDataA <= mem[addrA];
             end
         end
```

2.时序说明：

时钟下降沿的时候更新寄存器

3.代码：

```
//随机访问存储器，模拟内存中的一个内存芯片
module
ram(clock,addrA,wEnA,wDataA,rEnA,rDataA,addrB,wEnB,wDataB,rEnB,rDataB);
    //声明字长，芯片的字节数，字节的地址
```

```
    parameter WORDSIZE = 8;
    parameter WORDNUM = 512;
    parameter ADDRSIZE = 9;
    //输入输出端口声明
    //有两个既可以读又可以写的端口，地址输入分别为 addrA，addrB，写的使能端分别为 wEnA，wEnB，写的数据输入分别为 wDataA,wDataB。
    //读的使能端分别为 rEnA，rEnB，读的输出分别为 rDataA，rDataB
    input clock;
    input [ADDRSIZE-1:0] addrA;
    input [WORDSIZE-1:0] wDataA;
    input wEnA;
    input rEnA;
    output [WORDSIZE-1:0] rDataA;
    reg [WORDSIZE-1:0]rDataA;
    input [ADDRSIZE-1:0] addrB;
    input [WORDSIZE-1:0] wDataB;
    input wEnB;
    input rEnB;
    output [WORDSIZE-1:0]rDataB;
    reg[WORDSIZE-1:0]rDataB;
    reg[WORDSIZE-1:0]mem[WORDNUM-1:0];
    //时钟下降沿时进行数据的读写
    always@(negedge clock)
    begin
        if(wEnA)
            begin
                mem[addrA] <= wDataA;
            end
        if(rEnA)
            begin
                rDataA <= mem[addrA];
            end
        end
    always@(negedge clock)
    begin
        if(wEnB)
            begin
                mem[addrB] <= wDataB;
            end
        if(rEnB)
            begin
                rDataB <= mem[addrB];
            end
        end
endmodule
```

3.2.4 内存模块

1. 基本功能说明：

module myMerory(maddr,wenable,wdata,renable,rdata,m_ok,iaddr,instr,i_ok,clock);

1）模块声明如上，该模块为计算机内存，用于存储数据和指令，使用小字端法储存，该模块有两个端口，分别是数据读写端口和指令读取端口。maddr 指明数据读写的地址，wenable 为 1 时为写操作，renable 为 1 时为读操作，wdata 为写数据，rdata 为读的数据，m_ok 用于表明数据内存访问是否错误，iaddr 为指令访存的地址，instr 为读取的指令，i_ok 用于表明指令访存是否错误，clock 为时钟。

2）该内存的实现调用了 ram 模块，用 16 个 ram 模块组成内存。所以在访存和读取的时候需要从不同的 ram 块中读取一个个字节然后把它们拼接在一起

对内存芯片的声明如下：

ram #(8,memsize/16,60)
bank0(clock,addrI0,1'b0,8'b0,1'b1,outI0,addrD0,dwEn0,inD0,renable,outD0);

下面用数据访存为例说明内存的读写模式：

```
wire[7:0] ib0,ib1,ib2,ib3,ib4,ib5,ib6,ib7,ib8,ib9;
```

首先声明了 10 个 8 位的 wire 型变量，分别用于从不同的 ram 获得一个字节的数据。然后组合在一起形成 instr（指令）ib0 为地址的最低一个字节，ib1 为地址的第二低的字节，依此类推。

iaddr 的后四位指明了第一个内存芯片(bank)的 id，前 60 位为要读取的字节在内存芯片中的地址，iipl 为 iindex 的下个地址。对于不同的 bank 和 ibid 其字节地址有可能是 iipl，有可能是 iindex，例如当 ibid=4’h15 时，对于 bank15，其访存地址为 iindex，而对于 bank0,bank1,bank2.....bank8 其访存地址都是 iipl。

```
wire[3:0]ibid = iaddr[3:0];
wire[59:0] iindex = iaddr[63:4];
wire[59:0] iipl = iindex+1;
```

在代码中还声明了 15 个 wire 型变量，用于输出 bank 的指令输出，之后 ib0 到 ib9 会根据 ibid 选择出正确 bank 的输出。

wire[7:0]
outI0,outI1,outI2,outI3,outI4,outI5,outI6,outI7,outI8,outI9,outI10,outI11,outI12,outI13,outI14,outI15;

以 ib0 为例，说明选择逻辑的思路

当 ibid 为 0 时，访问的第一个 bank 是 bank0，所以指令的最低位 ib0 应该是 outI0，显然当 ibid 为 x 时，第一个访问的 bank 是 bankx，指令的最低位 ib0 就应该是 outIx

```
assign   ib0 = !i_ok ? 0 :
           ibid == 0 ? outI0 :
           ibid == 1 ? outI1 :
           ibid == 2 ? outI2 :
           ibid == 3 ? outI3 :
           ibid == 4 ? outI4 :
           ibid == 5 ? outI5 :
           ibid == 6 ? outI6 :
           ibid == 7 ? outI7 :
```

```
ibid == 8 ? outI8 :
ibid == 9 ? outI9 :
ibid == 10 ? outI10 :
ibid == 11 ? outI11 :
ibid == 12 ? outI12 :
ibid == 13 ? outI13 :
ibid == 14 ? outI14 :
outI15;
```

ibx 都选择完成之后，对 10 个 ibx 进行组合，组合方式如下，形成 instr 输出。

```
assign      instr[7:0] = ib0;
assign       instr[15:8] = ib1;
assign       instr[23:16] = ib2;
assign       instr[31:24] = ib3;
assign       instr[39:32] = ib4;
assign       instr[47:40] = ib5;
assign       instr[55:48] = ib6;
assign       instr[63:56] = ib7;
assign       instr[71:64] = ib8;
assign       instr[79:72] = ib9;
```

另外 i_ok 还要判断内存访问是否越界

```
assign i_ok = (iaddr+9)<memsize;
```

整个访存过程如上所述

3) 代码的输入格式:由于代码输入到内存当中时要保证格式是 ：

icode ifun rA rB valC,的格式，其中 valC 为小字端法。参考 ram 在输入的时候 idata 的低位写在小地址，所以输入的时候 icode ifun 在低位，rA，rB 次之，valC 在最高位。故输入格式为 valC rArB icodeifun。

2. 时序说明：

时钟下降沿的时候更新寄存器

3. 代码：

由于这部分得代码过于冗长，详细代码见附录。

3.2.5split 模块

1. 主要功能说明:

该模块主要用于对指令的第一个字节进行切割,获得指令码和功能码。指令码是该字节的高四位，功能码是该字节的低四位。

2. 端口说明

code(输入)	用于输入第一字节
icode(输出)	指令码
ifun(输出)	功能码

3. 代码

```
//Fetch stage
//split the code into icode and ifun
//get the ifun and icode
```

```
module split(code,icode,ifun);
    input[7:0]code;
    output[3:0]icode;
    output[3:0]ifun;
    assign icode = code[7:4];
    assign ifun = code[3:0];
endmodule
```

3.2.6align 模块

1. 基本功能说明：

该模块主要用于获得 rA，rB，valC，输入 code 为指令的后 9 个字节，然后对 9 个字节进行切割，低 4 位是 srcB，第 7 到 4 位是 srcA，根据 needRegids 是否为真选择不同位成为 valC。

2.端口声明

code(input)	输入的 9 字节指令
rA(output)	寄存器 id
rB(output)	寄存器 id
valC(output)	立即数
needRegids(input)	是否需要寄存器

3.时序说明：

该模块不受始终控制

4.代码

```
module align(code,rA,rB,valC,needRegids);
    input[71:0]code;
    input needRegids;
    output[3:0]rA;
    output[3:0]rB;
    output[63:0]valC;
    assign rB = code[3:0];
    assign rA = code[7:4];
    assign valC = needRegids?code[71:8]: code[63:0];
endmodule
```

3.2.7valP 计算模块

1. 基本功能说明

该模块可以根据是否需要寄存器，是否需要 valC，和当前 pc 值计算 valP。needRedigs 表示是否需要寄存器，needValC 表示是否需要 valC,根据这两个信号再 pc 上加上一定数字获得 valP。

2. 端口说明

pc(input)	当前 pc
needRegids(input)	表明是否需要寄存器
needValC(input)	表明是否需要 valC
valP(output)	valP

3. 时序说明：

该模块不受时钟控制。

4.代码

```
//用于计算 valP 的模块
//根据当前的 pc，needRegids 和 needValC，计算出 valP
module pc_increment(pc,needRegids,needValC,valP);
    input[63:0]pc;
    input needRegids;
    input needValC;
    output[63:0]valP;
    assign valP = pc+1+needRegids+8*needValC;
endmodule
```

3.2.8alu 模块

1. **基本功能说明：**该模块为计算模块，可以进行加、减、与、异或四种操作，输出 valE，并获得 newCC。newCC 用于更新 CC 寄存器

2. 端口说明

aluA(input)	**第一个计算的数字**
aluB(input)	**第二个计算的数字**
ifun(input)	**指令码**
valE(output)	**计算结果**
newCC(outpu)	**新产生的 CC 值**

3. 时序控制：

该模块不需要时钟控制

4.代码：

```
//Execute Stage
//ALU
//可以进行 and,sub,add,xor 四中操作，得到 valE 和 newCC
module alu(aluA,aluB,ifun,valE,newCC);
    //设定四种计算的编码
    parameter ALUADD = 4'h0;
    parameter ALUSUB = 4'h1;
    parameter ALUAND = 4'h2;
    parameter ALUXOR = 4'h3;
    input[63:0] aluA;
    input[63:0] aluB;
    input[3:0] ifun;
    output[63:0] valE;
    output[2:0] newCC;
    //计算 valE
    assign valE = ifun==ALUADD ? aluA+aluB:
```

```
                ifun==ALUSUB ? aluB-aluA:
                ifun==ALUAND ? aluB&aluA:
                aluB^aluA;
    //设置 newCC
    assign newCC[2] = (valE==0);
    assign newCC[1] = valE[63];
    assign                          newCC[0]                          =
(aluA[63]==aluB[63])&(valE[63]!=aluB[63])&(ifun==ALUADD) ? 1:

(aluA[63]!=aluB[63])&(aluB[63]!=valE[63])&(ifun==ALUSUB) ? 1:
                    0;
endmodule
```

3.2.9 条件码寄存器

1. 主要功能说明：

该模块主要储存条件码，newCC 是输入的新的 CC 值。reset 是复位，复位值为 3’h100。clock 为时钟。setCC 是更新 CC 的控制码。

2. 端口声明

newCC(input)	新的 cc
cc(output)	当前的 cc 值
setCC(input)	更新 cc 的控制码
reset(input)	复位 cc
clock(input)	时钟

3. 时序说明：

当时钟上升沿来临时更新寄存器。

4.代码：

```
//条件码寄存器，用于更新条件码
module CC(newCC,cc,setCC,reset,clock);
    output[2:0] cc;
    input[2:0] newCC;
    input setCC;
    input reset;
    input clock;
    cenrreg #(3)ccReg(newCC,cc,setCC,reset,3'b100,clock);
endmodule
```

3.2.10 cond 模块

1.主要功能说明

该模块主要用来计算 cond 码，用来判断 jXX 指令是否进行转跳,comxx 是否进行传递。根据指令功能码 ifun，当前状态码 cc 计算出 Cnd 并输出

2. 端口说明

ifun	指令功能码

cc	当前寄存器状态
Cnd	跳转判断

```
module cond(ifun,cc,Cnd);
    //设定功能的编码
    parameter C_YES = 4'h0;
    parameter C_LE = 4'h1;
    parameter C_L = 4'h2;
    parameter C_E = 4'h3;
    parameter C_NE = 4'h4;
    parameter C_GE = 4'h5;
    parameter C_G = 4'h6;
    input[3:0] ifun;
    input[2:0] cc;
    output Cnd;
    //将条件码分开
    wire zf = cc[2];
    wire sf = cc[1];
    wire of = cc[0];
    //计算 Cnd
    assign Cnd = (ifun==C_YES)|
                 ifun==C_LE &((sf^of)|zf)|
                 (ifun == C_L & (sf^of)) |
                 ifun==C_E & zf|
                 ifun==C_NE & ~zf|
                 ifun==C_GE & (~sf^of)|
                 ifun==C_G & ((~sf^of)&~zf);
endmodule
```

3.2.11 seq+模块

1.主要功能说明

该模块是 cpu-seq+，该模块调用了上述的 10 个模块，来组成了 seq 中的寄存器和部分组合逻辑，另外还有大量的组合逻辑。

2.端口说明

mode(input)	用于控制 cpu 的模式
stat(output)	用于输出 cpu 的状态
odata(output)	用于输出寄存器或内存的值
udaddr(input)	输入或者输出的地址
idata(input)	输入的值
clock(input)	时钟

3. cpu 模式说明

模式	功能	编码
run	cpu 正常运行的模式	000
reset	重置 cpu 中的寄存器	001
download	对内存进行输入	010
upload	查看内存的值	011
status	查看寄存器文件的值	100

4. cpu 状态说明

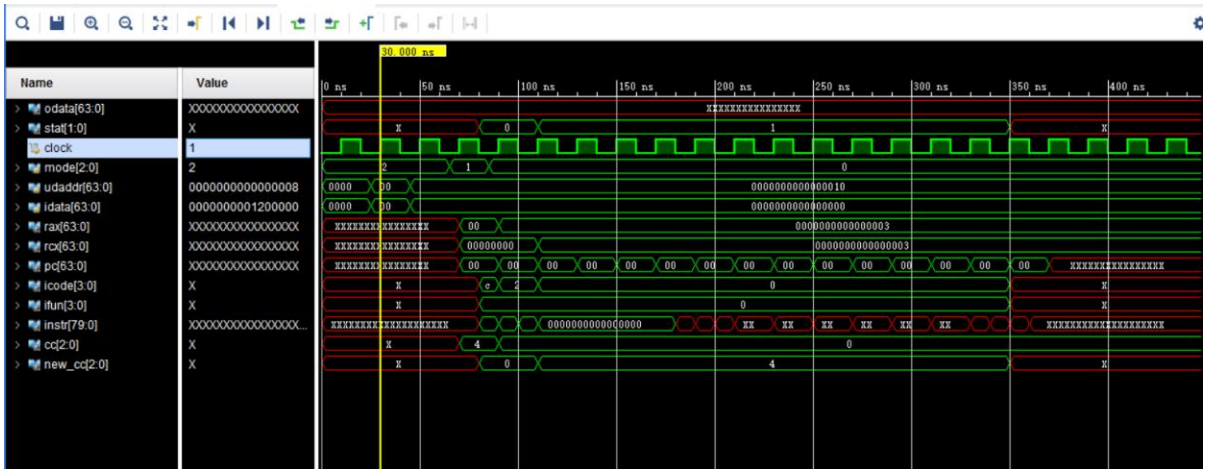
状态	功能	编码
SAOK	cpu 正常状态	00
SHLT	cpu 停机状态	01
SADR	cpu 数据访存错误	10
SINS	cpu 指令访存错误	11

第 4 章 设计验证

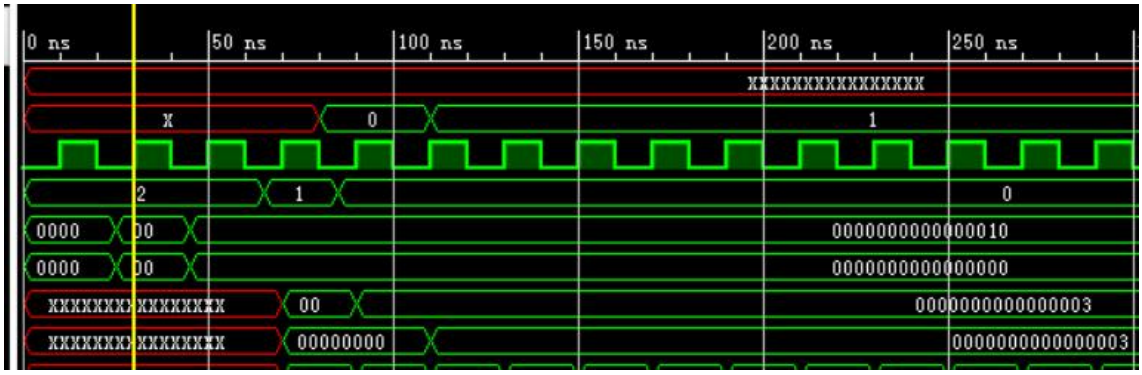
（该章满分 20 分）

4.1 仿真验证

仿真波形如下
测试指令为
iaddq $3,%rax
rrmovq %rax,%rcx
halt
机器码为：
0cf0_0000_0000_0000_0003
2001_0000_0000_0000_0000
（不足 8 的倍数用 halt 指令补齐）
3 条指令执行之后，rax 将为 3，rcx 将为 3，cpu 状态为停机状态。



可以看到在 90ns 时，时钟上升沿来临，然后 rax 寄存器（第 7 行波形）被更新为 3，110ns 时时钟上升沿来临 rcx 寄存器（第 8 行波形）被更新为 3，然后 cpu 读取下一条指令（halt），cpu 进入停机状态（cpu 状态为第二行波形）（状态代码为 1）



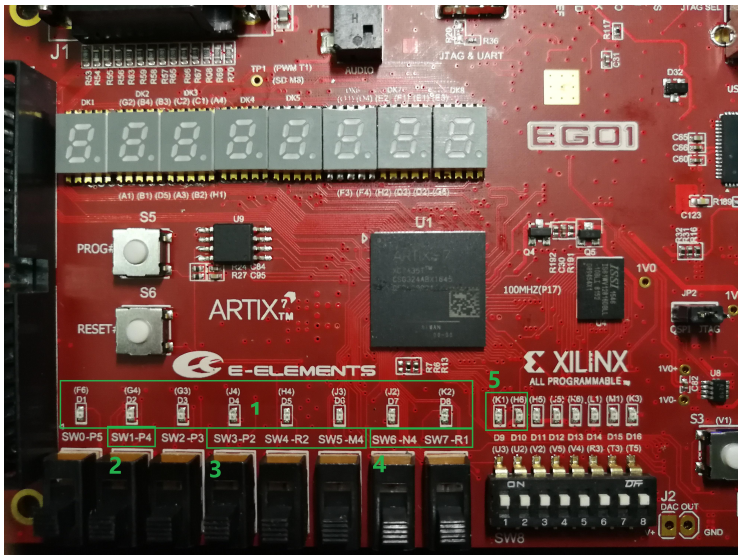
4.2 综合（synthesis）验证（选做，加分项，最多加分值为作业项总成绩的 5 分，作业项分值加满为止）

1. 约束文件

```
set_property IOSTANDARD LVCMOS33 [get_ports {addr[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {addr[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {mode[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {mode[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {mode[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {odata[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {odata[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {odata[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {odata[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {odata[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {odata[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {odata[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {odata[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {stat[1]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {stat[0]}]
set_property PACKAGE_PIN N4 [get_ports {addr[1]}]
set_property PACKAGE_PIN R1 [get_ports {addr[0]}]
set_property PACKAGE_PIN P2 [get_ports {mode[2]}]
set_property PACKAGE_PIN R2 [get_ports {mode[1]}]
set_property PACKAGE_PIN M4 [get_ports {mode[0]}]
set_property PACKAGE_PIN F6 [get_ports {odata[7]}]
set_property PACKAGE_PIN G4 [get_ports {odata[6]}]
set_property PACKAGE_PIN G3 [get_ports {odata[5]}]
set_property PACKAGE_PIN J4 [get_ports {odata[4]}]
set_property PACKAGE_PIN H4 [get_ports {odata[3]}]
set_property PACKAGE_PIN J3 [get_ports {odata[2]}]
set_property PACKAGE_PIN J2 [get_ports {odata[1]}]
set_property PACKAGE_PIN K2 [get_ports {odata[0]}]
set_property PACKAGE_PIN K1 [get_ports {stat[1]}]
set_property PACKAGE_PIN H6 [get_ports {stat[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports clock]
set_property PACKAGE_PIN P4 [get_ports clock]
```

2. 端口说明



所绑管脚如图所示，1 区域的灯用于输出寄存器文件中某个寄存器的后 8 位的输出。2 区域的一个输入为时钟，

3 区域的 3 个输入为 cpu 模式，不同输入对应的模式如下表

模式	功能	编码
run	cpu 正常运行的模式	000
reset	重置 cpu 中的寄存器	001
download	对内存进行输入	010
upload	查看内存的值	011
status	查看寄存器文件的值	100

4 区域的两个管脚是地址控制端口。端口为 00 时对应的地址是 64'h0，端口为 01 时对应的地址为 64'h0000_0000_0000_0008。每次加 8 一次类推。输入指令的时候，

将端口设为 00 会输入 64’h0000_0000_0000_0000 到 64’h0000_0000_0000_0007 的指令。当端口设为 01 会输入 64’h0000_0000_0000_0008 到 64’h0000_0000_0000_000f 的指令。

区域 5 是输出 cpu 的状态

状态	功能	编码
SAOK	cpu 正常状态	00
SHLT	cpu 停机状态	01
SADR	cpu 数据访存错误	10
SINS	cpu 指令访存错误	11

5. 综合顶层模块

```
module seq_synthesis(mode,stat,clock,addr,odata);
input[2:0]mode;
output[1:0]stat;
input clock;
input[1:0]addr;
output[7:0]odata;
wire[63:0] idata;
wire[63:0] udaddr;
wire[63:0]data;
assign odata = data[7:0];
seq mySeq(mode,stat,data,udaddr,idata,clock);
assign udaddr = addr == 2'b00 ? 64'h0000_0000_0000_0000:
                                addr == 2'b01 ? 64'h0000_0000_0000_0008:
                                64'h0;
assign idata = addr== 2'b00 ? 64'h0000_0000_0003_f0c0:
                        addr== 2'b01 ? 64'h0000_0000_0120_0000:
                        addr==2'b10?
64'h0000_0000_0000_0000:64'h0000_0000_0003_f0c0;
endmodule
```

6. 端口说明

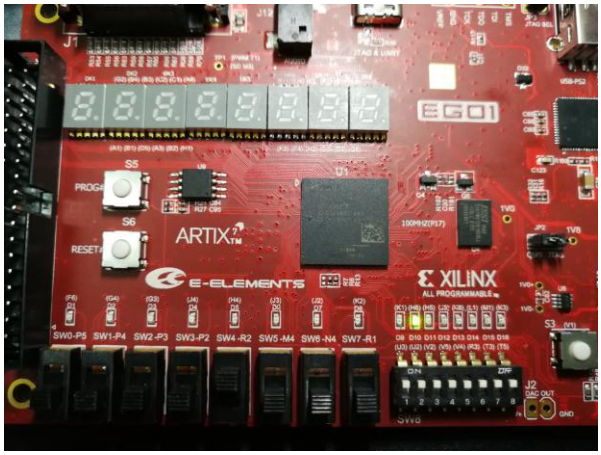
mode(input)	控制 cpu 的模式
stat(output)	输出 cpu 的状态
odata(output)	输出 cpu 寄存器的状态
addr(input)	输入地址
clock	时钟

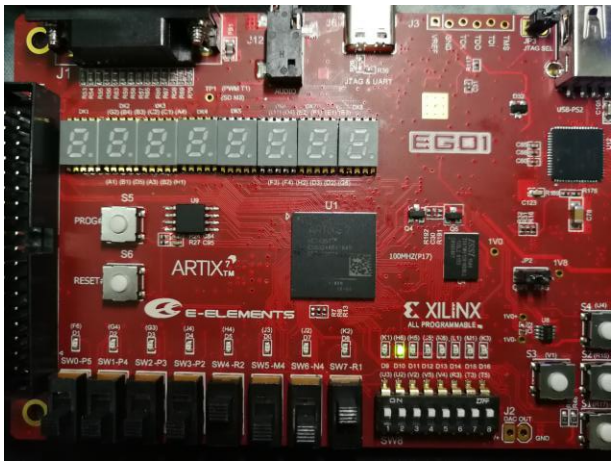
7. 输入指令

```
iaddq $3,%rax
rrmovq %rax,%rcx
halt
```

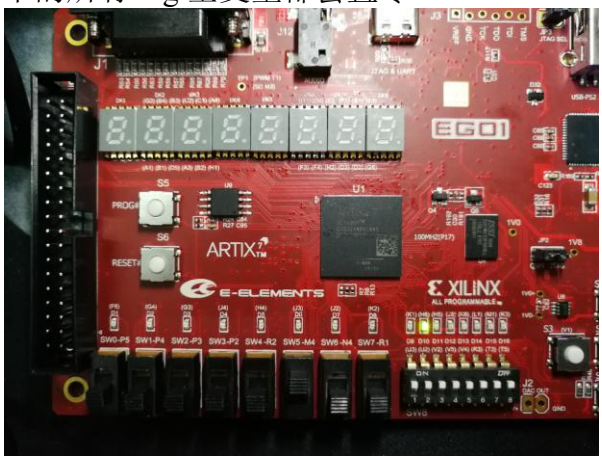
8. 测试

一开始先输入指令，将 mode 拨到 2 输入指令的模式，将地址管脚分别拨到 00（0x0）和 01（0x8），同时拨动时钟管脚输入指令。

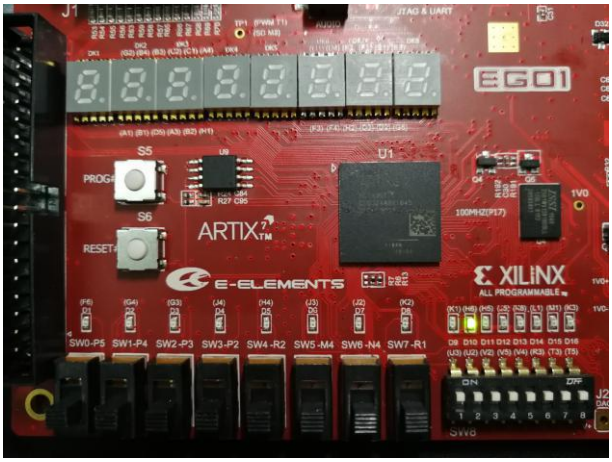




指令执行后如下图，之后将 mode 拨到 1（resetting 模式），拨动一次时钟之后，cpu 中的所有 reg 型变量都会置零。

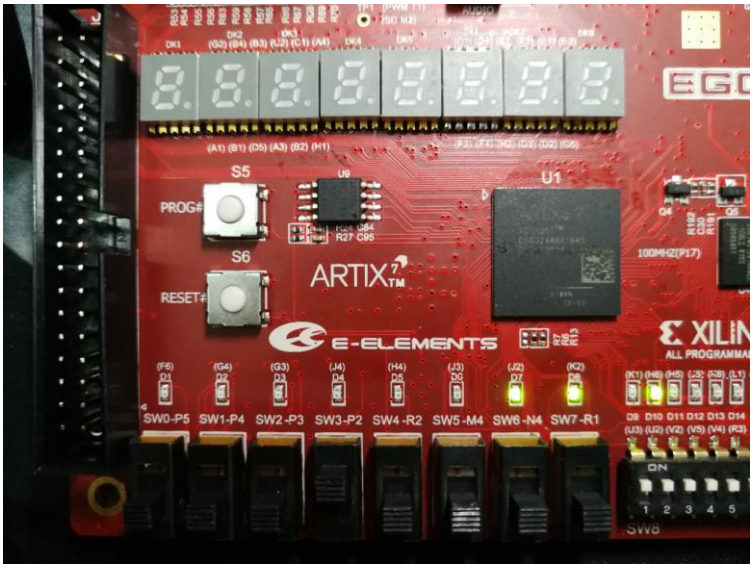


然后 mode 拨到 0（running 模式），拨动时钟，cpu 开始运行，运行之后的状态如下图：

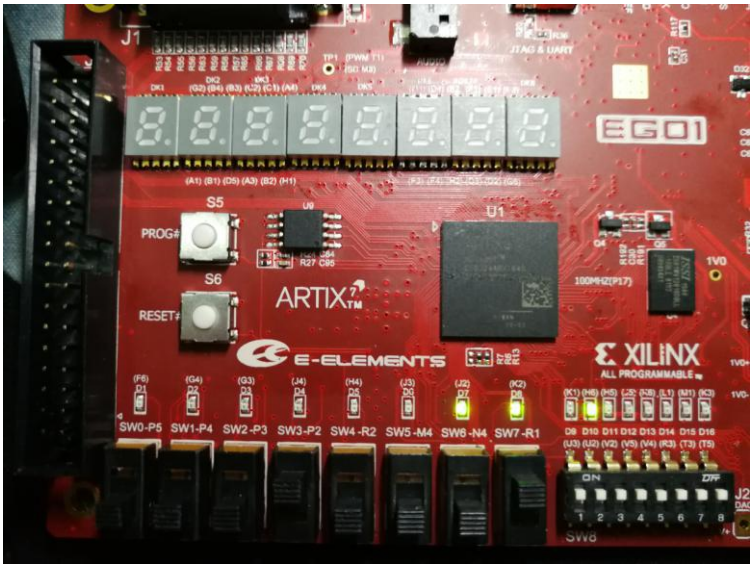


可以看到执行完三条执行之后 cpu 的状态变成了 01，代表停机指令，即运行 halt 指令的结果。

将 mode 设置为 4，用来查看寄存器的值，然后地址设置为 0，8 个灯将表示 rax 的低 8 位，可以发现 rax 的值已经为 3。



之后拨动地址端口为 01(对应%rcx)，发现%rcx 的值也为 3。



第 5 章 总结

5.1 请总结本次实验的收获

本次大作业，自学了 verilog 语言和 vivado 的使用，从写代码，到仿真和综合，花了很大的力气，感觉自己已经基本掌握了 fpga 编程的流程。

5.2 请给出对本次实验内容的建议

无

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 《深入理解计算机系统》，作者 Randal E. Bryant、David R. O’Hallaron，2016-11-15

[2] Randal E,Bryant & David R.O’Hallaron.cs:app3e Web Site ARCH:VLOG Verilog Implementation of a Pipelined Y86 Processor.December 29.2014