**计算机组织结构**

**CPU 设计**



04020120 柏济舟

04020522 凌嘉谦

**信息科学与工程学院**

**东南大学**

2023.05

目录

[一、实验目的 2](#_Toc23977)

[二、任务 3](#_Toc9848)

[1、CPU内部结构 4](#_Toc30294)

[2、内部寄存器和内存： 4](#_Toc3537)

[3、微程序控制单元 7](#_Toc24655)

[三、 仿真结果与分析 10](#_Toc21369)

[四、 实物演示 18](#_Toc22975)

[五、 小组成员与分工（待添加） 20](#_Toc2194)

[附录 20](#_Toc21592)

[Top\_Module 20](#_Toc20663)

[Main\_Memory 24](#_Toc17206)

[ALU 25](#_Toc24793)

[BR 27](#_Toc25788)

[ACC 28](#_Toc20000)

[IR 28](#_Toc20609)

[MBR 30](#_Toc29862)

[PC 30](#_Toc4491)

[Control\_Memory 31](#_Toc15693)

[Instruction\_register 33](#_Toc5987)

[Branch\_logic 34](#_Toc765)

[Instruction\_Decoder 34](#_Toc23754)

[Control\_Address\_Register 35](#_Toc6920)

[Multiplexer 35](#_Toc22804)

[Seg\_Decoder 36](#_Toc24972)

[Seg\_Scan 37](#_Toc14717)

# **一、实验目的**

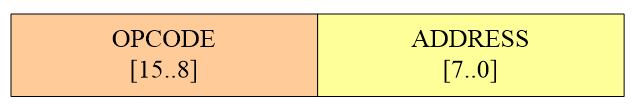
这个项目的目的是设计一个简单的CPU。这种CPU包含基本指令集,我们将利用其指令集生成一个非常简单的程序。为简单起见,我们只会考虑CPU内的关系,寄存器,内存和指令集。我们只需要考虑下列事项:读/写寄存器,读/写内存和执行指令。

CPU工作有五个阶段,即获取指令,译码指令,获取数据、处理数据,写入数据。至少四个部分构成一个简单的CPU:控制单元,内部寄存器,ALU和指令集,这是我们的项目设计的主要研究方面。

# **二、任务**

我们的简单的CPU设计使用单地址指令格式。指令字包含两个部分:操作数(码),它定义了函数的指令;地址部分,大多数指令地址部分包含数据的内存位置操作,我们称之为直接寻址。在一些指令,地址部分是操作数,叫做立即寻址。

简化起见,内存的大小是256\*16。指令有16比特。其中操作码部分8比特，地址段8比特。指令字如图1所示。



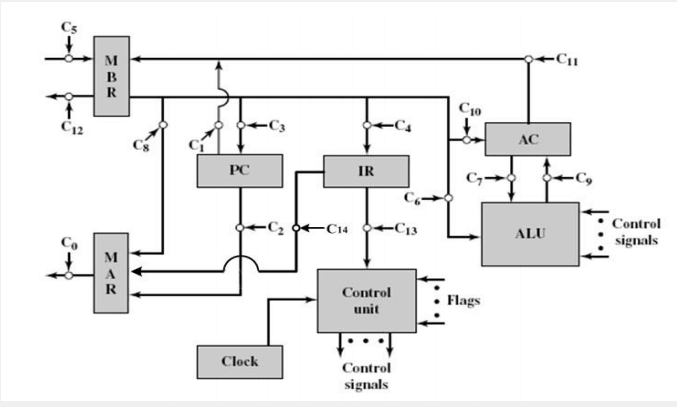
***图 1***

相关的操作码指令表1中列出。

|  |  |  |
| --- | --- | --- |
| INSTRUCTION | OPCODE | COMMENTS |
| STORE X | 00000001 | ACC->[X] |
| LOAD X | 00000010 | [X]->ACC |
| ADD X | 00000011 | ACC+[X]->ACC |
| SUB X | 00000100 | ACC-[X]->ACC |
| JMPGEZ X | 00000101 | If ACC>=0 Then X->PC else PC+1->PC |
| JMP X | 00000110 | X->PC |
| HALT | 00000111 | Halt a program |
| MPY X | 00001000 | ACC\*[X]->MR,ACC |
| AND X | 00001001 | ACC and [X]->ACC |
| OR X | 00001010 | ACC or [X]->ACC |
| NOT X | 00001011 | NOT [X]->ACC |
| SHIFTR | 00001100 | SHIFT [X] to right 1bit, Logic Shift |
| SHIFTL | 00001101 | SHIFT [X] to left 1bit, Logic Shift |

**表1 操作码指令集**

## 1、CPU内部结构

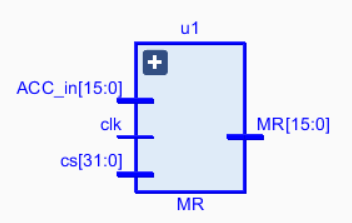


***图 2 CPU体系结构***

图中显示了一个简单的CPU体系结构及其对各种内部数据路径和控制信号的使用。我们的CPU设计应该基于这种架构。

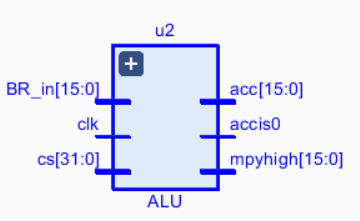
## 2、内部寄存器和内存：

### MR



MR用于实现MPY指令，将乘法器保持在指令的开头。当指令被执行时，它持有结果的一部分。

### ALU

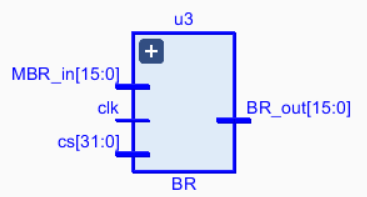


ALU（算术逻辑单元）是完成基本算术和逻辑运算的计算单元。在我们的设计中，必须支持如表2所示操作：

|  |  |
| --- | --- |
| Operations | Explanations |
| ADD | (ACC)  (ACC)  (BR) |
| SUB | (ACC)  (ACC)  (BR) |
| AND | (ACC)  (ACC) and (BR) |
| OR | (ACC)  (ACC) or (BR) |
| NOT | (ACC)  Not (ACC) |
| SRL | (ACC)  Shift (ACC) to Left 1 bit |
| SRR | (ACC)  Shift (ACC) to Right 1 bit |

**表2 ALU单元计算**

### BR

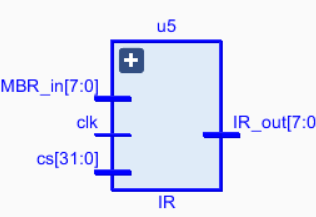


BR作为ALU的一个输入，存放着ALU的一个操作数。本课程中，BR有16比特。

### ACC

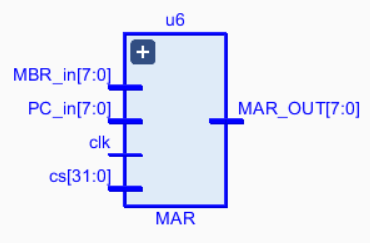
ACC保存着ALU的另一个操作数，而且通常ACC存放着ALU的计算结果。本课程中，ACC有16比特。

### IR



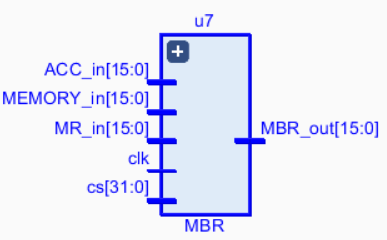
IR存放指令的OPCODE（操作码）部分。本课程中，IR有8比特。

### MAR

****

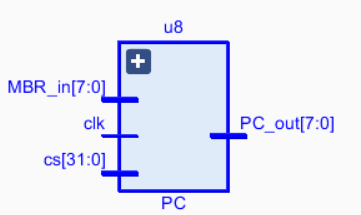
MAR存放着要从存储器中读取或要写入存储器的存储器地址。此处，“读”定义为CPU从内存中读。“写”定义为CPU把数据写入内存。本课程的设计中，MAR拥有8比特，可以存取256个地址。

### MBR

****

MBR存储着将要被存入内存或者最后一次从内存中读出来的数值。本课程的设计中，MBR有16比特。

### PC

****

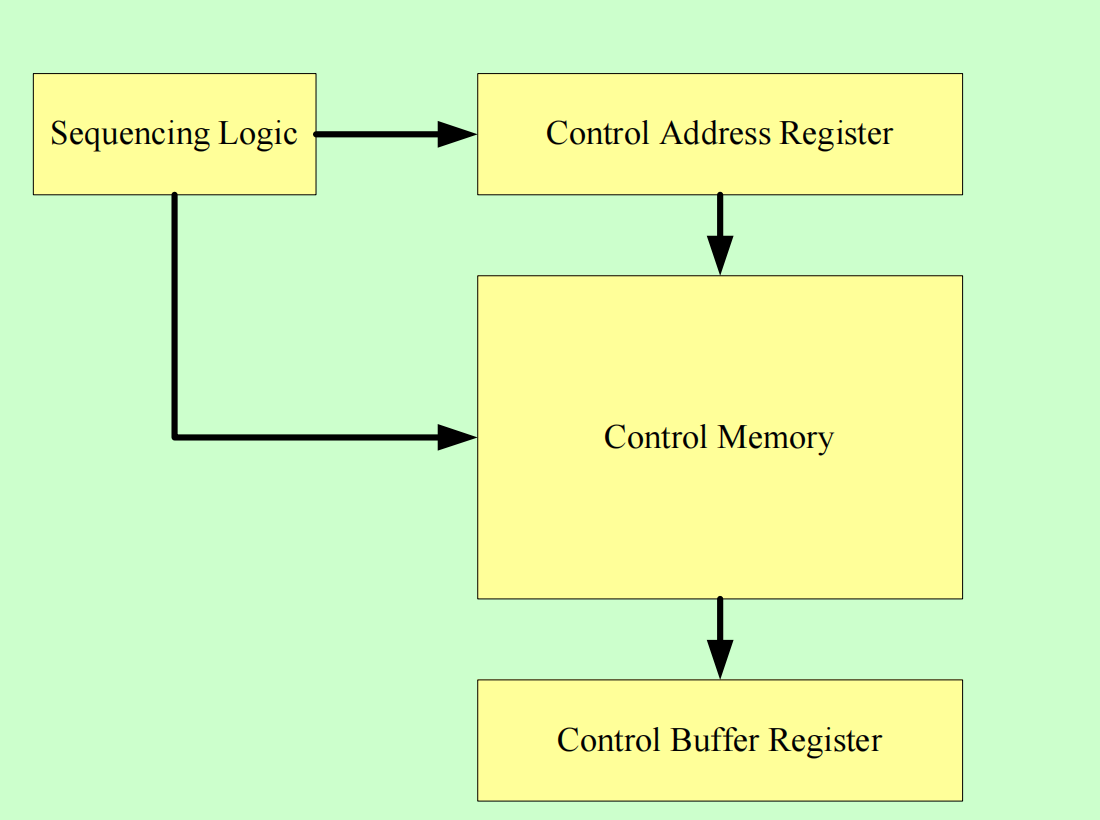
PC寄存器用来跟踪程序中将要使用的指令。本课程中，PC有8比特。

## 3、微程序控制单元

我们已经学习了微程序控制单元的知识。这里，我们只回顾一些术语和基本结构。

在微程序控制中，微程序由一些微指令组成，微程序存储在控制存储器中，控制存储器生成正确执行指令集所需的所有控制信号。微指令包含一些同时执行的微操作。

图3显示了这样实现的关键要素。微指令集存储在控制存储器中。控制地址寄存器包含要读取的下一条微指令的地址。当从控制存储器读取微指令时，它被传输到控制缓冲寄存器。寄存器连接到控制单元发出的控制线。因此，从控制存储器读取微指令与执行该微指令相同。图中所示的第三个元素是一个排序单元，它加载控制地址寄存器并发出读取命令。



***图 3 控制单元微结构***

**+1**

## CAR

Control

Memory

CBR

S2,S1,S0

CAR

Control Signals Jump Conditions Address

Field

Zero

Op\_Map

Int

0

Ind

U

**…**

#### C0, C1, …

2

1

S2

**…**

Branch

Logic

Multi-

plexer

S1

ALU Flags

Interrupt

S0

**Address selection**

0

Opcode Mapping

Decoder

DI

Indirect

Instruction Register

***图 4 微程序控制单元***

微指令与对应的微操作、控制内存的关系如下表所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Micro-Operations | | Control Memory | | | |
| Current  Address | Address  Field | Jump  Condition | Control  Signals |
| FETCH | MAR🡨PC  MBR🡨Memory, PC🡨PC+1  IR🡨MBR  IF (Indirect=1) Jump to INDIRECT  Jump to EXECUTE | 0  1  2  3  4 | ---  ---  ---  5  12 | 0  0  0  Ind  U | C2  C0 C5CR  C4  ---  --- |
| INDIRECT | MAR🡨IRaddress  MBR🡨Memory  IRaddress🡨MBR  Jump to EXECUTE | 5  6  7  8 | ---  ---  ---  12 | 0  0  0  U | C14  C0 C5CR  C4  --- |
| INTERRUPT | MAR🡨Save\_Add, MBR🡨PC  Memory🡨MBR, PC🡨Routine\_Addr  Jump to FETCH | 9  10  11 | ---  ---  0 | 0  0  U | C1  C0 C12 Cw  --- |
| EXECUTE | Jump to Opcode routine | 12 | --- | Op-Map | C13 |
| Detect interrupt | IF (Interrupt=1) Jump to INTERRUPT  Jump to FETCH | 13  14 | 9  0 | Int  U | ---  --- |
| ADD AC,x | MAR🡨IRaddress  MBR🡨Memory  BR🡨MBR  AC🡨AC+BR | 15  16  17  18 | ---  ---  ---  13 | 0  0  0  U | C14  C0 C5 CR  C6  CALU3 C7 C9  C15 |
| SUB AC,x | MAR🡨IRaddress  MBR🡨Memory  BR🡨MBR  AC🡨AC+BR | 19  20  21  22 | ---  ---  ---  13 | 0  0  0  U | C14  C0 C5 CR  C6  CALU4 C7 C9 C15 |
| MUL AC,x | MAR🡨IRaddress  MBR🡨Memory  BR🡨MBR  [MR,AC]🡨AC×BR | 23  24  25  26 | ---  ---  ---  13 | 0  0  0  U | C14  C0 C5 CR  C6  CALU5 C7 C9 C15 |
| AND AC,x | MAR🡨IRaddress  MBR🡨Memory  BR🡨MBR  AC🡨AC AND BR | 27  28  29  30 | ---  ---  ---  13 | 0  0  0  U | C14  C0 C5 CR  C6  CALU6 C7 C9 C15 |
| OR AC,x | MAR🡨IRaddress  MBR🡨Memory  BR🡨MBR  AC🡨AC OR BR | 31  32  33  34 | ---  ---  ---  13 | 0  0  0  U | C14  C0 C5 CR  C6  CALU7 C7 C9 C15 |
| NOT AC,x | MAR🡨IRaddress  MBR🡨Memory  BR🡨MBR  AC🡨NOT BR | 35  36  37  38 | ---  ---  ---  13 | 0  0  0  U | C14  C0 C5 CR  C6  CALU8 C9 C15 |
| SHIFTR x | MAR🡨IRaddress  MBR🡨Memory  BR🡨MBR  AC🡨SHIFTR BR | 39  40  41  42 | ---  ---  ---  13 | 0  0  0  U | C14  C0 C5 CR  C6  CALU9 C9 C15 |
| SHIFTL x | MAR🡨IRaddress  MBR🡨Memory  BR🡨MBR  AC🡨SHIFTL BR | 43  44  45  46 | ---  ---  ---  13 | 0  0  0  U | C14  C0 C5 CR  C6  CALU10 C9 C15 |
| STORE x,AC | MAR🡨IRaddress,MBR🡨AC  Memory🡨MBR | 47  48 | ---  13 | 0  U | C11 C14  C0 C12 CW |
| LOAD AC,x | MAR🡨IRaddress  MBR🡨Memory  AC🡨MBR | 49  50  51 | ---  ---  13 | 0  0  U | C14  C0 C5 CR  C10 |
| JMPGEZ X | MAR🡨IRaddress  MBR🡨Memory  IF (ACC≥0) PC🡨MBR  Else PC🡨PC+1 | 52  53  54 | ---  ---  0 | 0  0  U | C14  C0 C5 CR  C3 |
| JMP X | MAR🡨IRaddress  MBR🡨Memory  PC🡨MBR | 55  56  57 | ---  ---  0 | 0  0  U | C14  C0 C5 CR  C3 |
| HALT | (The program is ended) | 58 | 58 | U |  |

***表3 控制信号及*每个指令的微操作**

# 仿真结果与分析

1. 下面对算式“Not（（2+6+...+202）\*5）SHL 2bit\*（-13）”进行仿真。

高级语言：

sum1=0;

temp1=202;

LOOP1:sum1=sum1+temp1;

temp1=temp1-4;

if(temp1≥0) goto LOOP1;

result1=sum1\*5;

result2=!(result1);

result3=result<<2;

result=result3\*(-13);

汇编语言：

LOAD A0

STORE B0//存储循环加法的结果

LOAD A1

STORE B2//存储循环变量

LOOP1:LOAD B0

ADD B2

STORE B0

LOAD B2

SUB B4

STORE B2

JMPGEZ LOOP1

LOAD B0

MUL A3

STORE B0

NOT BO

STORE BO

SHL BO

STORE BO

SHL BO

STORE BO

LOAD BO

MUL A4

STORE BO//最后运算的结果存储在地址为B0的存储单元中

HALT

各个存储单元的初始值如下（除特殊标注外，地址和内容均为十六进制，左三列为指令，最右列为初始数据）

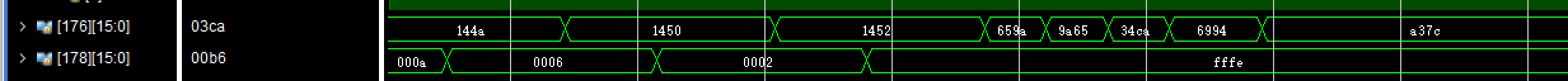
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 地址 | 内容 | 地址 | 内容 | 地址 | 内容 | 地址 | 内容 |
| 00 | 02A0 | 0C | 08A3 | … | … | A0 | 0 |
| 01 | 01B0 | 0D | 01B0 |  |  | A1 | CA |
| 02 | 02A1 | 0E | 0CB0 |  |  | A2 | 4 |
| 03 | 01B2 | 0F | 01B0 |  |  | A3 | 5 |
| 04 | 02B0 | 10 | 0EB0 |  |  | A4 | (-13)10 |
| 05 | 03B2 | 11 | 01B0 |  |  | A5 | 0 |
| 06 | 01B0 | 12 | 0EB0 |  |  | A6 | 0 |
| 07 | 02B2 | 13 | 01B0 |  |  | A7 | 4 |
| 08 | 04A2 | 14 | 02B0 |  |  | A8 | 0 |
| 09 | 01B2 | 15 | 08A4 |  |  |  |  |
| 0A | 05A7 | 16 | 01B0 |  |  |  |  |
| 0B | 02B0 | 17 | 07FF |  |  |  |  |

仿真过程如下：

（1）循环加法的仿真

如图为2+6+...+202的循环加法仿真图，每加一次memory[178]的存储变量大小减4，直到减为-2后跳出循环，最终得到的循环加法结果为5202，十六进制为1452，与预期结果一致。





（2）乘法运算的仿真



乘上5后memory[176]存储的变量变为26010，十六进制为659a，结果符合预期。

（3）取反运算的仿真



（26010）10 =（0110 0101 1001 1010）2

取反后变为（1001 1010 0110 0101）2=（9a65）16

结果符合预期

（4）逻辑移位的仿真



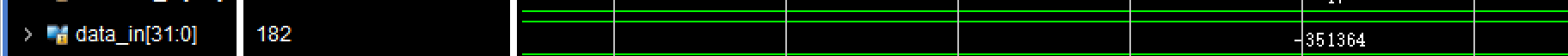
第一次移位变为（0011 0100 1100 1010）2=（34CA）16

第二次移位变为（0110 1001 1001 0100）2=（6994）16

结果符合预期

1. 最终结果的仿真

最终结果如下：



（6994）=（27028）10

（27028）10\*（-13）10=（-351364）10

与理论计算一致

主存模块代码如下：

module main\_memory(

//input clk,

input rst,

input [7:0] address\_in,

input signed [31:0] data\_in,

input R\_control,

input W\_control,

output reg signed [15:0] data\_out,

output signed [31:0] data\_show

);

wire [7:0] M\_address;

reg signed [15:0] memory [255:0];

always@ (\*) begin

//always@ (posedge clk or negedge rst) begin

if(rst==0)

begin

memory[0]=16'h02A0;

memory[1]=16'h01B0;

memory[2]=16'h02A1;

memory[3]=16'h01B2;

memory[4]=16'h02B0;

memory[5]=16'h03B2;

memory[6]=16'h01B0;

memory[7]=16'h02B2;

memory[8]=16'h04A2;

memory[9]=16'h01B2;

memory[10]=16'h05A7;

memory[11]=16'h02B0;

memory[12]=16'h08A3;

memory[13]=16'h01B0;

memory[14]=16'h0CB0;

memory[15]=16'h01B0;

memory[16]=16'h0EB0;

memory[17]=16'h01B0;

memory[18]=16'h0EB0;

memory[19]=16'h01B0;

memory[20]=16'h02B0;

memory[21]=16'h08A4;

memory[22]=16'h01B0;

memory[23]=16'h07FF;

memory[160]=16'h0000;

memory[161]=202;

memory[162]=16'h0004;

memory[163]=5;

memory[164]=-13;

memory[165]=16'h0000;

memory[166]=16'h0000;

memory[167]=16'h0004;

memory[168]=16'h0000;

memory[169]=16'h0000;

memory[170]=16'h0000;

memory[171]=16'h0000;

end

else begin

if(R\_control==1) begin//读

data\_out=memory[M\_address];

end

else if(W\_control==1) begin//写

memory[M\_address]=data\_in[15:0];

memory[M\_address+1]=data\_in[31:16];

end

else ;

end

end

assign M\_address=address\_in;

assign data\_show={memory[177],memory[176]};

2、下面对算式“（（2+8+14+...+302）OR（3+5+7+...+99））SHR 3bit”进行仿真。

高级语言：

sum1=0;

sum2=0;

temp1=302;

temp2=97;

LOOP1:sum1=sum1+temp1;

temp1=temp1-6;

if(temp1≥0) goto LOOP1;

LOOP2:temp2=temp2+2;

sum2=sum2+temp2;

temp2=temp2-4;

if(temp2≥0) goto LOOP2;

result1=sum1|sum2;

result=result1>>3;

汇编语言：

LOAD A0

STORE B0//存储第一个循环加法的结果

LOAD A1

STORE B2//存储第一个循环变量

LOOP1:LOAD B0

ADD B2

STORE B0

LOAD B2

SUB B4

STORE B2

JMPGEZ LOOP1

LOAD A2

STORE C0//存储第二个循环加法的结果

LOAD A3

STORE C2//存储第二个循环变量

LOOP2:LOAD C2

ADD A5

STORE C2

LOAD C0

ADD C2

STORE C0

LOAD C2

SUB A6

STORE C2

JMPGEZ LOOP2

LOAD B0

OR C0

STORE D0

SHR DO

STORE DO

SHR DO

STORE DO

SHR DO

STORE DO//最后运算的结果存储在地址为D0的存储单元中

HALT

各个存储单元的初始值如下（地址和内容均为十六进制，左三列为指令，最右列为初始数据）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 地址 | 内容 | 地址 | 内容 | 地址 | 内容 | 地址 | 内容 |
| 00 | 02A0 | 0C | 01C0 | 18 | 05A8 | A0 | 0 |
| 01 | 01B0 | 0D | 02A3 | 19 | 02B0 | A1 | 12E |
| 02 | 02A1 | 0E | 01C2 | 1A | 0BC0 | A2 | 0 |
| 03 | 01B2 | 0F | 02C2 | 1B | 01D0 | A3 | 61 |
| 04 | 02B0 | 10 | 03A5 | 1C | 0DD0 | A4 | 6 |
| 05 | 03B2 | 11 | 01C2 | 1D | 01D0 | A5 | 2 |
| 06 | 01B0 | 12 | 02C0 | 1E | 0DD0 | A6 | 4 |
| 07 | 02B2 | 13 | 03C2 | 1F | 01D0 | A7 | 4 |
| 08 | 04A4 | 14 | 01C0 | 20 | 0DD0 | A8 | E |
| 09 | 01B2 | 15 | 02C2 | 21 | 01D0 |  |  |
| 0A | 05A7 | 16 | 04A6 | 22 | 07FF |  |  |
| 0B | 02A2 | 17 | 01C2 | … | … |  |  |

仿真过程如下：

（1）第一个循环加法的仿真

如图，每一次循环变量加到保存结果的内存单元中，加法每进行一次后循环变量减6，直到循环变量减为-4后跳出循环，最终得到的循环加法结果为7752，与实验预期结果一致。





（2）第二个循环加法的仿真

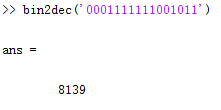
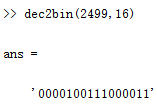
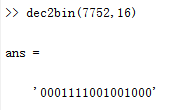
如图，每一次循环中，循环变量先加2，再加到保存结果的内存单元中，加法完成后循环变量减4，直到循环变量减为-1后跳出循环，最终得到的循环加法结果为2499，与理论计算结果一致。



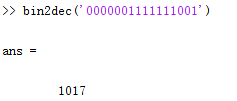


（3）按位取或和逻辑右移的仿真

如图，用MATLAB计算两个循环加法结果7752和2499的16位二进制数表示，可以得到7752的二进制表示为0001111001001000,2499的二进制表示为0000100111000011，这两串二进制码按位取或得到0001111111001011，它的十进制表示为8139。



按位取或得到的结果逻辑右移3位，得到0000001111111001，它的十进制表示为1017。



如下图，由仿真结果可知，按位取或后得到的结果为8139，逻辑右移3位后得到的结果为1017，与理论分析结果相同。



主存模块代码如下：

module main\_memory(

//input clk,

input rst,

input [7:0] address\_in,

input signed [31:0] data\_in,

input R\_control,

input W\_control,

output reg signed [15:0] data\_out,

output signed [31:0] data\_show

);

wire [7:0] M\_address;

reg signed [15:0] memory [255:0];

always@ (\*) begin

//always@ (posedge clk or negedge rst) begin

if(rst==0)

begin

memory[0]=16'h02A0;

memory[1]=16'h01B0;

memory[2]=16'h02A1;

memory[3]=16'h01B2;

memory[4]=16'h02B0;

memory[5]=16'h03B2;

memory[6]=16'h01B0;

memory[7]=16'h02B2;

memory[8]=16'h04A4;

memory[9]=16'h01B2;

memory[10]=16'h05A7;

memory[11]=16'h02A2;

memory[12]=16'h01C0;

memory[13]=16'h02A3;

memory[14]=16'h01C2;

memory[15]=16'h02C2;

memory[16]=16'h03A5;

memory[17]=16'h01C2;

memory[18]=16'h02C0;

memory[19]=16'h03C2;

memory[20]=16'h01C0;

memory[21]=16'h02C2;

memory[22]=16'h04A6;

memory[23]=16'h01C2;

memory[24]=16'h05A8;

memory[25]=16'h02B0;

memory[26]=16'h0BC0;//OR

memory[27]=16'h01D0;

memory[28]=16'h0DD0;//SHR

memory[29]=16'h01D0;

memory[30]=16'h0DD0;//SHR

memory[31]=16'h01D0;

memory[32]=16'h0DD0;//SHR

memory[33]=16'h01D0;

memory[34]=16'h07FF;//halt

memory[160]=16'h0000;

memory[161]=302;

memory[162]=16'h0000;

memory[163]=97;

memory[164]=16'h0006;

memory[165]=16'h0002;

memory[166]=16'h0004;

memory[167]=16'h0004;

memory[168]=16'h000E;

end

else begin

if(R\_control==1) begin//读

data\_out=memory[M\_address];

end

else if(W\_control==1) begin//写

memory[M\_address]=data\_in[15:0];

memory[M\_address+1]=data\_in[31:16];

end

else ;

end

end

assign M\_address=address\_in;

assign data\_show={memory[209],memory[208]};

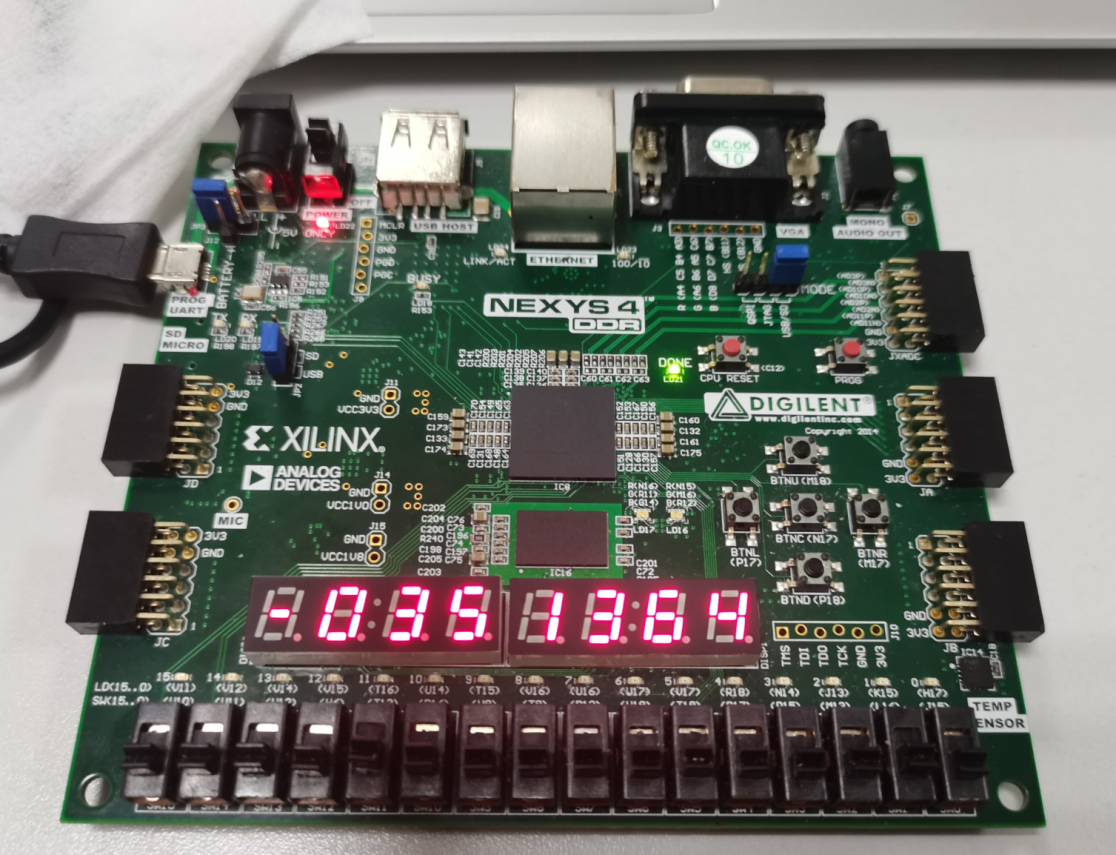
endmodule

# 实物演示

Not（（2+6+...+202）\*5）SHL 2bit\*（-13）

理论结果为：-351364

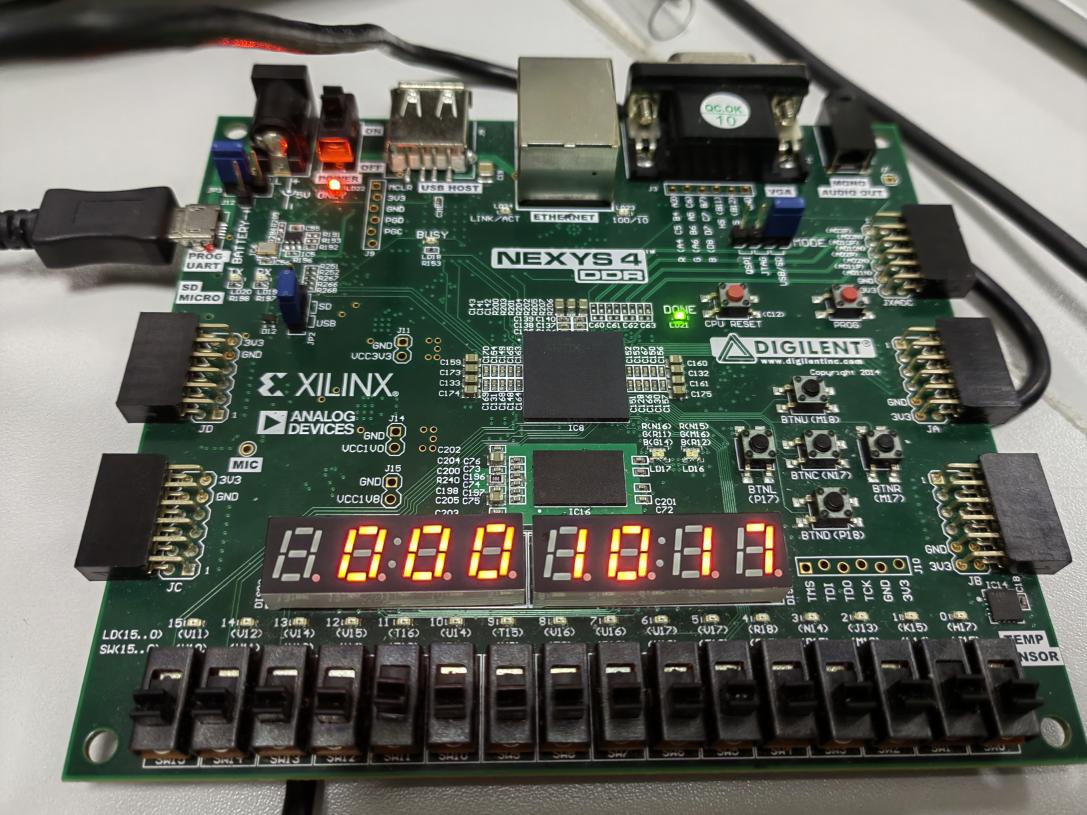
实物结果如图：



（（2+8+14+...+302）OR（3+5+7+...+99））SHR 3bit

理论结果为：1017

实物结果如图：



# 小组成员与分工（待添加）

04020120 柏济舟（50%）：负责...

04020522 凌嘉谦（50%）：负责内部寄存器如MBR、MAR的编写，负责ALU算数部分，完成各部分的整合与撰写实验报告。

# 附录

## Top\_Module

module top\_module(

input clk,

input rst,

output [7:0] seg\_sel,

output [6:0] seg\_data

);

reg [3:0] count=0;

reg clk\_true=0;

//分频

always@(posedge clk or negedge rst) begin

if(rst==0) begin

count<=0;

clk\_true<=0;

end

else begin

if(count==10) begin

count<=0;

clk\_true<=~clk\_true;

end

else count<=count+1;

end

end

wire signed [31:0] data\_show;

reg [31:0] show\_temp;

wire [7:0] IR\_temp;

wire [5:0] control\_address;

wire [7:0] memory\_address;

wire [1:0] RW\_control;

Main my\_Main(

.clk(clk\_true),

.rst(rst),

.C(control\_buffer\_content[31:9]),

.IR\_out(IR\_temp),

.memory\_address(memory\_address),

.data\_show(data\_show)

);

wire[31:0] control\_buffer\_content;

control\_memory my\_control\_memory(

.rst(rst),

.control\_address(control\_address),

.control\_buffer\_content(control\_buffer\_content)

);

wire indirect\_flag,interrupt\_flag;

wire [5:0] opcode;

Instruction\_register my\_Instruction\_register(

.IR(IR\_temp),

.indirect\_flag(indirect\_flag),

.interrupt\_flag(interrupt\_flag),

.opcode(opcode)

);

wire [2:0] address\_select;

branch\_logic my\_branch\_logic(

.conditions(control\_buffer\_content[8:6]),

.indirect\_flag(indirect\_flag),

.interrupt\_flag(interrupt\_flag),

.address\_select(address\_select)

);

wire [5:0] IR\_decoder\_address;

Instruction\_decoder my\_Instruction\_decoder(

.opcode(opcode),

.IR\_decoder\_address(IR\_decoder\_address)

);

wire [5:0] output\_address,CAR\_OUT;

control\_address\_register my\_control\_address\_register(

.rst(rst),

.address\_in(output\_address),

.CAR\_OUT(CAR\_OUT),

.control\_address(control\_address)

);

multiplexer my\_multiplexer(

.clk(clk\_true),

.rst(rst),

.address\_select(address\_select),

.address\_field(control\_buffer\_content[5:0]),

.CAR(CAR\_OUT),

.IR\_decoder\_address(IR\_decoder\_address),

.output\_address(output\_address)

);

wire [6:0] seg\_data\_0;

wire [6:0] seg\_data\_1;

wire [6:0] seg\_data\_2;

wire [6:0] seg\_data\_3;

wire [6:0] seg\_data\_4;

wire [6:0] seg\_data\_5;

wire [6:0] seg\_data\_6;

wire [6:0] seg\_data\_7;

wire [3:0] bin\_data\_0;

wire [3:0] bin\_data\_1;

wire [3:0] bin\_data\_2;

wire [3:0] bin\_data\_3;

wire [3:0] bin\_data\_4;

wire [3:0] bin\_data\_5;

wire [3:0] bin\_data\_6;

reg [3:0] bin\_data\_7;

//判断数据的正负

always@(posedge clk) begin

if(data\_show<0) begin

show\_temp<=~(data\_show-1);

bin\_data\_7<=4'ha;

end

else begin

show\_temp<=data\_show;

bin\_data\_7<=4'hb;

end

End

//显示输出

assign bin\_data\_0=show\_temp%10;

assign bin\_data\_1=(show\_temp/10)%10;

assign bin\_data\_2=(show\_temp/100)%10;

assign bin\_data\_3=(show\_temp/1000)%10;

assign bin\_data\_4=(show\_temp/10000)%10;

assign bin\_data\_5=(show\_temp/100000)%10;

assign bin\_data\_6=(show\_temp/1000000)%10;

seg\_decoder seg\_decoder\_0

(

.bin\_data(bin\_data\_0), // bin data input

.seg\_data(seg\_data\_0) // seven segments LED output

);

seg\_decoder seg\_decoder\_1

(

.bin\_data(bin\_data\_1), // bin data input

.seg\_data(seg\_data\_1) // seven segments LED output

);

seg\_decoder seg\_decoder\_2

(

.bin\_data(bin\_data\_2), // bin data input

.seg\_data(seg\_data\_2) // seven segments LED output

);

seg\_decoder seg\_decoder\_3

(

.bin\_data(bin\_data\_3), // bin data input

.seg\_data(seg\_data\_3) // seven segments LED output

);

seg\_decoder seg\_decoder\_4

(

.bin\_data(bin\_data\_4), // bin data input

.seg\_data(seg\_data\_4) // seven segments LED output

);

seg\_decoder seg\_decoder\_5

(

.bin\_data(bin\_data\_5), // bin data input

.seg\_data(seg\_data\_5) // seven segments LED output

);

seg\_decoder seg\_decoder\_6

(

.bin\_data(bin\_data\_6), // bin data input

.seg\_data(seg\_data\_6) // seven segments LED output

);

seg\_decoder seg\_decoder\_7

(

.bin\_data(bin\_data\_7), // bin data input

.seg\_data(seg\_data\_7) // seven segments LED output

);

seg\_scan scan\_0(

.clk(clk\_true),

.rst\_n(rst),

.seg\_sel(seg\_sel), //digital led chip select

.seg\_data(seg\_data), //eight segment digital tube output,MSB is the decimal point

.seg\_data\_7(seg\_data\_7),

.seg\_data\_6(seg\_data\_6),

.seg\_data\_5(seg\_data\_5),

.seg\_data\_4(seg\_data\_4),

.seg\_data\_3(seg\_data\_3),

.seg\_data\_2(seg\_data\_2),

.seg\_data\_1(seg\_data\_1),

.seg\_data\_0(seg\_data\_0)

);

Endmodule

## Main\_Memory

module main\_memory(

input rst,

input [7:0] address\_in,

input signed [15:0] data\_in,

input R\_control,

input W\_control,

output reg signed [15:0] data\_out,

output signed [15:0] data\_show

);

wire [7:0] M\_address;

reg signed [15:0] memory [255:0];

always@ (\*) begin

//always@ (posedge clk or negedge rst) begin

if(rst==0)

Begin

//内存

memory[0]=16'h02A0;

memory[1]=16'h01A4;

memory[2]=16'h02A2;

memory[3]=16'h01A3;

memory[4]=16'h02A4;

memory[5]=16'h03A3;

memory[6]=16'h01A4;

memory[7]=16'h02A3;

memory[8]=16'h04A1;

memory[9]=16'h01A3;

memory[10]=16'h05A5;

memory[11]=16'h07FF;//halt

memory[160]=16'h0000;

memory[161]=16'h0001;

memory[162]=16'h0064;

memory[163]=16'h0000;

memory[164]=16'h0000;

memory[165]=16'h0004;

end

else begin

if(R\_control==1) begin//

data\_out=memory[M\_address];

end

else if(W\_control==1) begin//

memory[M\_address]=data\_in;

end

else ;

end

end

assign M\_address=address\_in;

assign data\_show=memory[164]; //数据显示

Endmodule

## ALU

module ALU(

input rst,

input [15:0] ACCin,BRin,

input [22:0] C,

output reg signed [15:0] ALU = 0, MRout = 0

);

reg signed [31:0] MPY = 0;

always @ (\*)

begin

if(rst == 0)

begin

ALU = 0;

MRout = 0;

end

else begin

if(C[22]==1 && C[10]==1 && C[5]==1)

begin

ALU = BRin; //Store

end

else if(C[12]==1)

begin

ALU = ACCin; //Load

end

else if(C[15]==1 && C[13]==1 && C[7]==1 && C[4:1]==4'b0011)

begin

ALU = ACCin + BRin; //ADD

end

else if(C[15]==1 && C[13]==1 && C[7]==1 && C[4:1]==4'b0100)

begin

ALU = ACCin - BRin; //SUB

end

else if(C[15]==1 && C[13]==1 && C[7]==1 && C[4:1]==4'b0101)

begin

MPY = $signed(ACCin) \* $signed(BRin); //MPY

ALU = MPY[15:0];

MRout = MPY[31:16];

end

else if(C[15]==1 && C[13]==1 && C[7]==1 && C[4:1]==4'b0110)

begin

ALU = ACCin & BRin; //AND

end

else if(C[15]==1 && C[13]==1 && C[7]==1 && C[4:1]==4'b0111)

begin

ALU = ACCin | BRin; //OR

end

else if(C[15]==1 && C[13]==1 && C[7]==1 && C[4:1]==4'b1000)

begin

ALU = ~ACCin; //NOT

end

else if(C[15]==1 && C[13]==1 && C[7]==1 && C[4:1]==4'b1001)

begin

ALU = {1'b0 , ACCin[15:1]}; //SHIFTR

end

else if(C[15]==1 && C[13]==1 && C[7]==1 && C[4:1]==4'b1010)

begin

ALU = {ACCin[14:0] , 1'b0}; //SHIFTL

end

else ;

end

end

Endmodule

## BR

module BR(

input clk,

input rst,

input [22:0] C,

input signed [15:0] MBRin,

output signed [15:0] ALUout

);

reg signed [15:0] BR\_temp;

always @ (posedge(clk))

begin

if(rst == 0)

BR\_temp <= 0;

else

begin

if(C[16] == 1)

BR\_temp <= MBRin;

end

end

assign ALUout = BR\_temp;

Endmodule

## ACC

module ACC(

input clk,

input rst,

input [22:0] C,

input signed [15:0] BERin,

input signed [15:0] ALUin,

output signed [15:0] MBRout

);

reg signed [15:0] ACC\_temp ;

always @ (posedge(clk))

begin

if(rst == 0)

begin

ACC\_temp <= 0;

end

else

begin

if(C[12]==1)

ACC\_temp <= BERin;

else if(C[13] == 1)

ACC\_temp <= ALUin;

else ;

end

end

assign MBRout = ACC\_temp;

Endmodule

## IR

module IR(

//input clk,

input rst,

input [22:0] C,

input [15:0] MBRin,

output [7:0] CUout,

output [7:0] IRaddress\_out

);

reg [7:0] IR\_temp = 8'b0,IRaddress\_temp=8'b0;

always @ (\*)

begin

if(rst == 0) begin

IR\_temp = 0;

IRaddress\_temp = 0;

end

else

begin

if(C[9] == 1) begin

IR\_temp = MBRin[15:8];

IRaddress\_temp = MBRin[7:0];

end

end

end

assign CUout = IR\_temp;

assign IRaddress\_out = IRaddress\_temp;

Endmodule

**MAR**

module MAR(

input clk,

input rst,

input [22:0] C,

input [15:0] MBRin,

input [7:0] PCin,IRin,

output [7:0] Memoryout

);

reg [7:0] MAR\_temp = 8'b0;

always @ (posedge(clk) or negedge rst)

begin

if(rst == 0)

MAR\_temp <= 0;

else

begin

if(C[20] == 1)

MAR\_temp <= PCin;

else if(C[8] == 1)

MAR\_temp <= IRin;

end

end

assign Memoryout = MAR\_temp;

Endmodule

## MBR

module MBR(

input clk,

input rst,

input [22:0] C,

input signed [15:0] ACCin,Memoryin,

input signed [7:0] PCin,

output [15:0] MBRout

);

reg signed [15:0] MBR\_temp;

always @ (posedge clk)

begin

if(rst == 0)

MBR\_temp <= 0;

else

Begin

if(C[6] == 1)

MBR\_temp <= Memoryin;

else if(C[11] == 1)

MBR\_temp <= ACCin;

else if(C[21] == 1)

MBR\_temp <= PCin[7:0];

end

end

assign MBRout = MBR\_temp;

Endmodule

## PC

module PC(

input clk,

input rst,

input signed [15:0] ACCin,

input [22:0] C,

input [15:0] MBRin,

output [7:0] MARout

);

reg [7:0] PC\_temp;

always @ (posedge clk or negedge rst)

begin

if(rst == 0)

PC\_temp <= 0;

else begin

if(C[0] == 1)

PC\_temp <= PC\_temp + 1;

else if(C[19] == 1) begin

if(ACCin>=0)PC\_temp <= MBRin[7:0];

else ;

end

end

end

assign MARout = PC\_temp;

Endmodule

## Control\_Memory

module control\_memory(

input rst,

input [5:0] control\_address,

output reg [31:0] control\_buffer\_content

);

wire [31:0] control\_memory\_content [255:0];

//fetch

assign control\_memory\_content[0]={23'b00100000000000000000000,3'b000,6'b000001};//21

assign control\_memory\_content[1]={23'b10000100000000001000001,3'b000,6'b000010};

assign control\_memory\_content[2]={23'b00001000000000000000000,3'b000,6'b000011};//000

assign control\_memory\_content[3]={23'b00000000000000000000000,3'b001,6'b000101};//001

assign control\_memory\_content[4]={23'b00000000000000000000000,3'b100,6'b001100};//100

//indirect

assign control\_memory\_content[5]={23'b00000000000000100000000,3'b000,6'b000110};

assign control\_memory\_content[6]={23'b10000100000000001000000,3'b000,6'b000111};

assign control\_memory\_content[7]={23'b00001000000000000000000,3'b000,6'b001000};

assign control\_memory\_content[8]={23'b00000000000000000000000,3'b100,6'b001100};//100

//interrupt

assign control\_memory\_content[9]={23'b01000000000000000000000,3'b000,6'b001010};

assign control\_memory\_content[10]={23'b10000000001000000100000,3'b000,6'b001011};

assign control\_memory\_content[11]={23'b00000000000000000000000,3'b100,6'b000000};//

//op-map

assign control\_memory\_content[12]={23'b00000000000001000000000,3'b011,6'b000000};//011

//interrupt detect

assign control\_memory\_content[13]={23'b00000000000000000000000,3'b010,6'b001001};

assign control\_memory\_content[14]={23'b00000000000000000000000,3'b100,6'b000000};

//add

assign control\_memory\_content[15]={23'b00000000000000100000000,3'b000,6'b010000};

assign control\_memory\_content[16]={23'b10000100000000001000000,3'b000,6'b010001};

assign control\_memory\_content[17]={23'b00000010000000000000000,3'b000,6'b010010};

assign control\_memory\_content[18]={23'b00000001010000010000110,3'b100,6'b001101};//

//sub

assign control\_memory\_content[19]={23'b00000000000000100000000,3'b000,6'b010100};

assign control\_memory\_content[20]={23'b10000100000000001000000,3'b000,6'b010101};

assign control\_memory\_content[21]={23'b00000010000000000000000,3'b000,6'b010110};

assign control\_memory\_content[22]={23'b00000001010000010001000,3'b100,6'b001101};

//mul

assign control\_memory\_content[23]={23'b00000000000000100000000,3'b000,6'b010100};

assign control\_memory\_content[24]={23'b10000100000000001000000,3'b000,6'b010101};

assign control\_memory\_content[25]={23'b00000010000000000000000,3'b000,6'b010110};

assign control\_memory\_content[26]={23'b00000001010000010001010,3'b100,6'b001101};

//and

assign control\_memory\_content[27]={23'b00000000000000100000000,3'b000,6'b010100};

assign control\_memory\_content[28]={23'b10000100000000001000000,3'b000,6'b010101};

assign control\_memory\_content[29]={23'b00000010000000000000000,3'b000,6'b010110};

assign control\_memory\_content[30]={23'b00000001010000010001100,3'b100,6'b001101};

//or

assign control\_memory\_content[31]={23'b00000000000000100000000,3'b000,6'b010100};

assign control\_memory\_content[32]={23'b10000100000000001000000,3'b000,6'b010101};

assign control\_memory\_content[33]={23'b00000010000000000000000,3'b000,6'b010110};

assign control\_memory\_content[34]={23'b00000001010000010001110,3'b100,6'b001101};

//not

assign control\_memory\_content[35]={23'b00000000000000100000000,3'b000,6'b010100};

assign control\_memory\_content[36]={23'b10000100000000001000000,3'b000,6'b010101};

assign control\_memory\_content[37]={23'b00000010000000000000000,3'b000,6'b010110};

assign control\_memory\_content[38]={23'b00000000010000010010000,3'b100,6'b001101};

//shiftr

assign control\_memory\_content[39]={23'b00000000000000100000000,3'b000,6'b010100};

assign control\_memory\_content[40]={23'b10000100000000001000000,3'b000,6'b010101};

assign control\_memory\_content[41]={23'b00000010000000000000000,3'b000,6'b010110};

assign control\_memory\_content[42]={23'b00000000010000010010010,3'b100,6'b001101};

//shiftl

assign control\_memory\_content[43]={23'b00000000000000100000000,3'b000,6'b010100};

assign control\_memory\_content[44]={23'b10000100000000001000000,3'b000,6'b010101};

assign control\_memory\_content[45]={23'b00000010000000000000000,3'b000,6'b010110};

assign control\_memory\_content[46]={23'b00000000010000010010100,3'b100,6'b001101};

//store

assign control\_memory\_content[47]={23'b00000000000100100000000,3'b000,6'b010110};

assign control\_memory\_content[48]={23'b10000000000010000100000,3'b100,6'b001101};

//load

assign control\_memory\_content[49]={23'b00000000000000100000000,3'b000,6'b010101};

assign control\_memory\_content[50]={23'b10000100000000001000000,3'b000,6'b010110};

assign control\_memory\_content[51]={23'b00000000001000000000000,3'b100,6'b001101};

//jmp

assign control\_memory\_content[52]={23'b00000000000000100000000,3'b000,6'b010101};

assign control\_memory\_content[53]={23'b10000100000000001000000,3'b000,6'b010110};

assign control\_memory\_content[54]={23'b00010000000000000000000,3'b100,6'b000000};

//jmpgez

assign control\_memory\_content[55]={23'b00000000000000100000000,3'b000,6'b010101};

assign control\_memory\_content[56]={23'b10000100000000001000000,3'b000,6'b010110};

assign control\_memory\_content[57]={23'b00010000000000000000000,3'b100,6'b000000};

//halt

assign control\_memory\_content[58]={23'b00000000000000000000000,3'b100,6'b111010};//

always@(\*) begin

if(rst==0) control\_buffer\_content=control\_memory\_content[0];

else control\_buffer\_content=control\_memory\_content[control\_address];

end

Endmodule

## Instruction\_register

module Instruction\_register(

input [7:0] IR,

output indirect\_flag,

output interrupt\_flag,

output [5:0] opcode

);

assign indirect\_flag=IR[7];

assign interrupt\_flag=IR[6];

assign opcode=IR[5:0];

Endmodule

## Branch\_logic

module branch\_logic(

input [2:0] conditions,

input indirect\_flag,

input interrupt\_flag,

output reg [2:0] address\_select

);

always@(\*) begin

if(conditions==3'b100) address\_select=3'b010;//

else if(conditions==3'b011) address\_select=3'b001;//

else if(conditions==3'b001) begin

if(indirect\_flag==1) address\_select=3'b010;//

else address\_select=3'b100;

end

else if(conditions==3'b010) begin

if(interrupt\_flag==1) address\_select=3'b010;//

else address\_select=3'b100;

end

else if(conditions==3'b000) address\_select=3'b100;//

else ;

end

Endmodule

## Instruction\_Decoder

module Instruction\_decoder(

input [5:0] opcode,

output reg [5:0] IR\_decoder\_address

);

always @(\*) begin

case(opcode)

6'b000001:IR\_decoder\_address=47;

6'b000010:IR\_decoder\_address=49;

6'b000011:IR\_decoder\_address=15;

6'b000100:IR\_decoder\_address=19;

6'b000101:IR\_decoder\_address=52;

6'b000110:IR\_decoder\_address=55;

6'b000111:IR\_decoder\_address=58;

6'b001000:IR\_decoder\_address=23;

6'b001010:IR\_decoder\_address=27;

6'b001011:IR\_decoder\_address=31;

6'b001100:IR\_decoder\_address=35;

6'b001101:IR\_decoder\_address=39;

6'b001110:IR\_decoder\_address=43;

default: ;

endcase

end

Endmodule

## Control\_Address\_Register

module control\_address\_register(

input rst,

input [5:0] address\_in,

output reg [5:0] CAR\_OUT,

output reg [5:0] control\_address

);

always@(\*)

if(rst==0) begin

CAR\_OUT=0;

control\_address=0;

end

else begin

CAR\_OUT=address\_in+1;

control\_address=address\_in;

end

Endmodule

## Multiplexer

module multiplexer(

input clk,

input rst,

input [2:0] address\_select,

input [5:0] address\_field,

input [5:0] CAR,

input [5:0] IR\_decoder\_address,

output reg [5:0] output\_address

);

always@(posedge clk or negedge rst) begin

if(rst==0) output\_address=1;

else begin

case (address\_select)

3'b001: output\_address = IR\_decoder\_address;

3'b010: output\_address = address\_field;

3'b100: output\_address = CAR;

default: ;

endcase

end

end

Endmodule

## Seg\_Decoder

module seg\_decoder

(

input[3:0] bin\_data, // bin data input

output reg[6:0] seg\_data // seven segments LED output

);

always@(\*)

begin

case(bin\_data)

4'd0:seg\_data <= 7'b000\_0001;//{a,b,c,d,e,f,g}

4'd1:seg\_data <= 7'b100\_1111;

4'd2:seg\_data <= 7'b001\_0010;

4'd3:seg\_data <= 7'b000\_0110;

4'd4:seg\_data <= 7'b100\_1100;

4'd5:seg\_data <= 7'b010\_0100;

4'd6:seg\_data <= 7'b010\_0000;

4'd7:seg\_data <= 7'b000\_1111;

4'd8:seg\_data <= 7'b000\_0000;

4'd9:seg\_data <= 7'b000\_0100;

4'ha:seg\_data <= 7'b111\_1110;

//4'hb:seg\_data <= 7'b000\_0011;

//4'hc:seg\_data <= 7'b100\_0110;

//4'hd:seg\_data <= 7'b010\_0001;

//4'he:seg\_data <= 7'b000\_0110;

//4'hf:seg\_data <= 7'b000\_1110;

default:seg\_data <= 7'b111\_1111;

endcase

end

Endmodule

## Seg\_Scan

module seg\_scan(

input clk,

input rst\_n,

output reg[7:0] seg\_sel, //digital led chip select

output reg[6:0] seg\_data, //eight segment digital tube output,MSB is the decimal point

input[6:0] seg\_data\_7,

input[6:0] seg\_data\_6,

input[6:0] seg\_data\_5,

input[6:0] seg\_data\_4,

input[6:0] seg\_data\_3,

input[6:0] seg\_data\_2,

input[6:0] seg\_data\_1,

input[6:0] seg\_data\_0

);

parameter SCAN\_FREQ = 2000; //scan frequency

parameter CLK\_FREQ = 50000000; //clock frequency

parameter SCAN\_COUNT = CLK\_FREQ /(SCAN\_FREQ \* 6) - 1;

reg[31:0] scan\_timer; //scan time counter

reg[3:0] scan\_sel; //Scan select counter

always@(posedge clk or negedge rst\_n)

begin

if(rst\_n == 1'b0)

begin

scan\_timer <= 32'd0;

scan\_sel <= 4'd0;

end

else if(scan\_timer >= SCAN\_COUNT)

begin

scan\_timer <= 32'd0;

if(scan\_sel == 4'd7)

scan\_sel <= 4'd0;

else

scan\_sel <= scan\_sel + 4'd1;

end

else

begin

scan\_timer <= scan\_timer + 32'd1;

end

end

always@(posedge clk or negedge rst\_n)

begin

if(rst\_n == 1'b0)

begin

seg\_sel <= 8'b11111111;

seg\_data <= 8'hff;

end

else

begin

case(scan\_sel)

//first digital led

4'd0:

begin

seg\_sel <= 8'b1111\_1110;

seg\_data <= seg\_data\_0;

end

//second digital led

4'd1:

begin

seg\_sel <= 8'b1111\_1101;

seg\_data <= seg\_data\_1;

end

4'd2:

begin

seg\_sel <= 8'b1111\_1011;

seg\_data <= seg\_data\_2;

end

4'd3:

begin

seg\_sel <= 8'b1111\_0111;

seg\_data <= seg\_data\_3;

end

4'd4:

begin

seg\_sel <= 8'b1110\_1111;

seg\_data <= seg\_data\_4;

end

4'd5:

begin

seg\_sel <= 8'b1101\_1111;

seg\_data <= seg\_data\_5;

end

4'd6:

begin

seg\_sel <= 8'b1011\_1111;

seg\_data <= seg\_data\_6;

end

4'd7:

begin

seg\_sel <= 8'b0111\_1111;

seg\_data <= seg\_data\_7;

end

default:

begin

seg\_sel <= 8'b1111\_1111;

seg\_data <= 7'hff;

end

endcase

end

end

Endmodule