计算机组织与结构 CPU 实验报告

一、实验目的

本实验的目的是设计并验证一个简单的 CPU（中央处理器）。这个 CPU 有基本的指令集，并且我们将利用它的指令集来生成一个非常简单的程序来验证它的性能。简单起见，我们只会考虑 CPU、寄存器、主存储器和指令集之间的关系也就是说，我们只需要考虑以下部分：读/写寄存器、读/写记忆以及执行指令。一个简单的 CPU 至少有四个部分组成：控制单元、内部寄存器、ALU（算术逻辑单元）和指令集，这是我们项目设计的主要方面。

二、实验任务

CPU 设计中使用单地址指令格式。指令字包括两部分：操作码(OPCODE)，用来定义指令的功能；地址段(Address Part)，用来存放要被操作的指令的地址。称之为直接寻址(Direct Addressing)。在一些少量的指令中，地址段就是操作数，这是立即数寻址(Immediate Addressing)。

简化起见，主存储器的大小为 256×16bits。其中每一条指令的大小为 16bits，指令中操作码部分 8bits，地址段 8bits，指令的格式如下图所示：

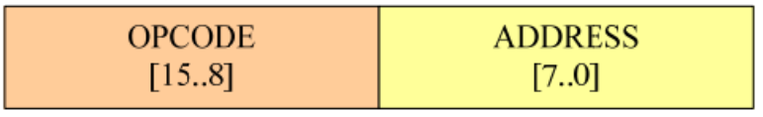


图 1 指令格式

相关指令的操作码如下图所示：（为了简化，我将功能一样的逻辑左移和算术左移用一个操作码，都是 0DH）

INSTRUCTION	OPCODE	COMMENTS
STORE X	01H	ACC→[X]
LOAD X	02H	[X]→ACC
ADD X	03H	ACC+[X]→ACC
SUB X	04H	ACC-[X]→ACC
JMPGEZ X	05H	If ACC≥0 then X→PC else PC+1→PC
JMP X	06H	X→PC
HALT	07H	Halt a program
MPY X	08H	ACC×[X]→ACC, MR
DIV X	09H	ACC/[X]→ACC, DR
AND X	0AH	ACC and [X]→ACC
OR X	0BH	ACC or [X]→ACC
NOT X	0CH	NOT [X]→ACC
SHIFTLL/SHIFTLA	0DH	SHIFT ACC to Left 1 bit, Logic/Arithmetic Shift
SHIFTRL	0EH	SHIFT ACC to Right 1 bit, Logic Shift
SHIFTRA	0FH	SHIFT ACC to Right 1 bit, Arithmetic Shift

三、实验分析

参考《计算机组织和结构》以及计算机的冯·诺依曼架构，CPU 与外部主存储器的通信方式主要通过一根数据线（MBR 负责）和一根地址线（MAR 负责）来实现，还有一个来自 CPU 内部的信号线控制读/写；PC 寄存器中存储了程序的地址；IR 寄存器中存放操作码；BR、ALU、ACC 模块共同负责程序中的逻辑运算；MCU（微程序控制单元）作为 CPU 内部控制核心单元，发出控制信号来操控 CPU 中各个模块的运行，MCU 内部工作方式也是通过“地址线+数据线”来实现，内部主要模块为 CAR、CBR 和 ROM，ROM 中存储了 CPU 的指令控制信号集。主要的工程文件架构如下图所示：

图 3 程序架构

运行 RTL analysis 中的 Schematic 得到下面的 RTL 图表结构：

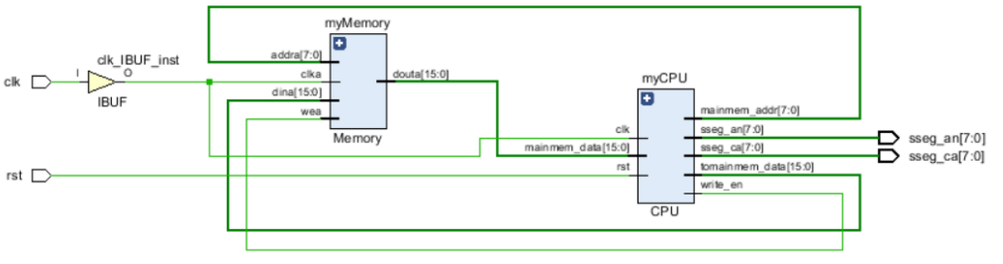


图 4 整体系统 RTL 图表

下面我们将分析各个模块实现的功能：

1. 主存储器调用现成的 IP 核 Block Memory 实现的，存储空间为 256*16bits，一共可以存储 256 条数据（包括指令和数据），具体结构如下图所示：

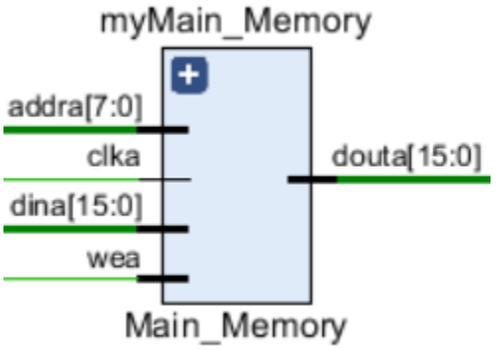


图 5 主存储器模型图

2. CPU 及其内部模块

CPU 由很多内部模块集成而成，功能复杂，下面是 CPU 的外部结构图：

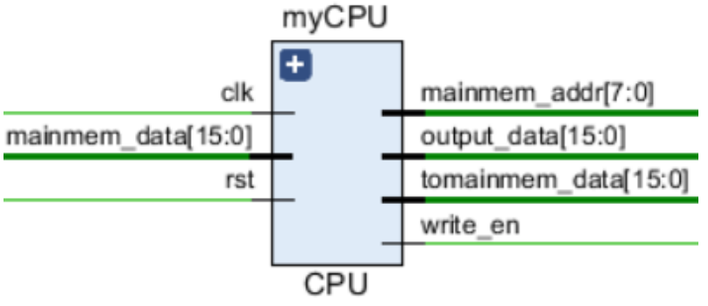


图 6 CPU 模型图

下面我们介绍 CPU 每个引脚的功能：

Name	Character	Explanation
clk	Input	Time signal for CPU
mainmem_data	Input[15:0]	Receive data from main memory
rst	Input	Reset signal for CPU
mainmem_addr	Output [7:0]	Address of main memory to be executed
output_data	Output [15:0]	Data from ACC
tomainmem_data	Output [15:0]	Data written into main memory by CPU
write_en	Output	Write enable signal of main memory and if disable read

接下来介绍 CPU 内部模块，CPU 内部连线示意图如下：

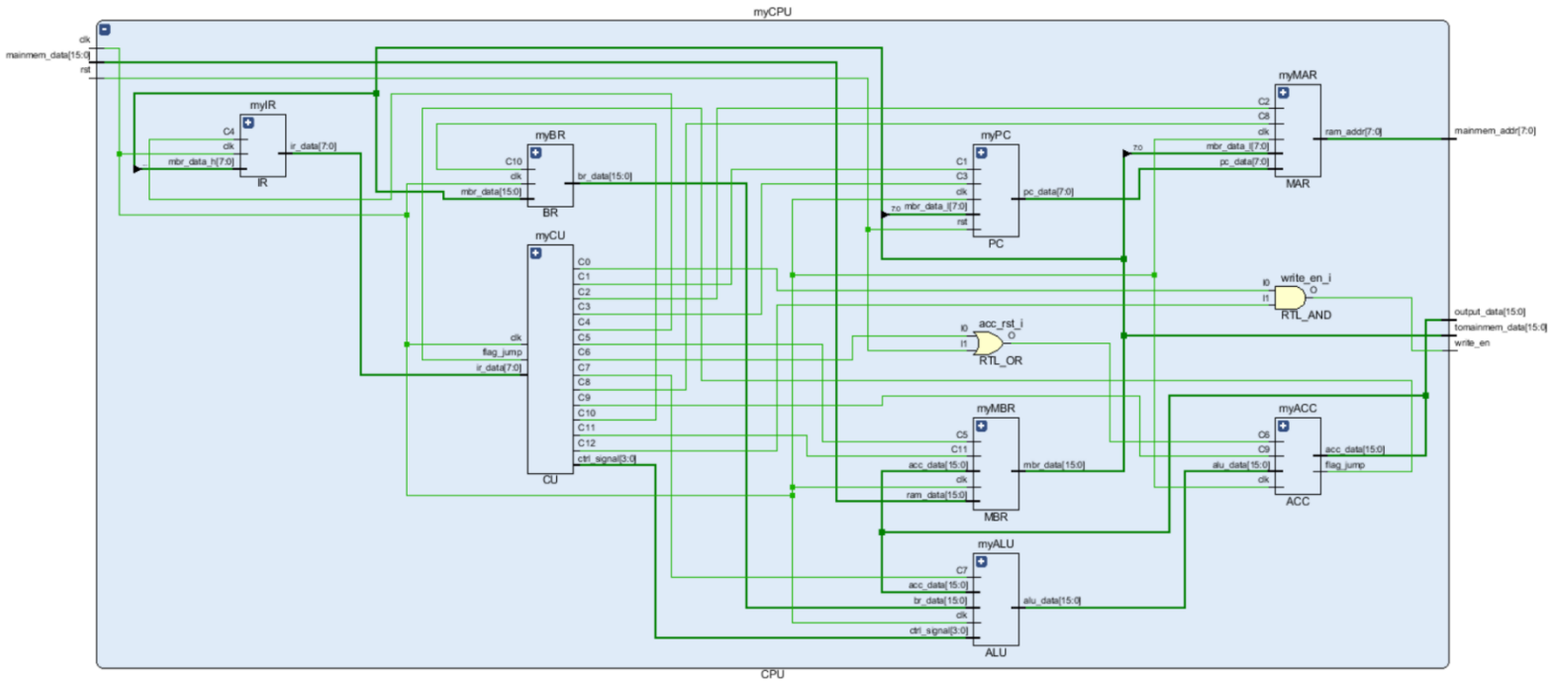


图 7 CPU 内部连线示意图

2.1. MAR 存储器地址寄存器(Memory Address Register)

MAR 包含要从主存储器中读取或要写入主存储器的数据的主存储器的地址。在这里，“读”操作表示为 CPU 从主存储器中读取数据，“写”操作表示为 CPU 写入数据到主存储器。在我们的设计中，MAR 大小为 8bits 为了完全指示主存储器的 256 个地址位，具体结构如下图所示：

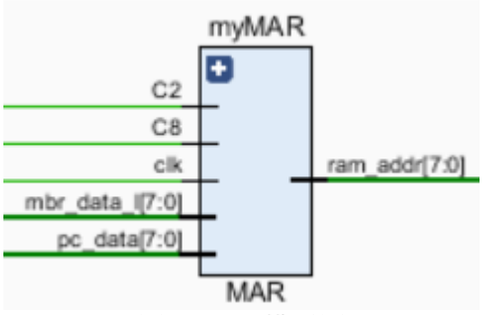


图 8 MAR 模型图

2.2. MBR 存储器缓冲寄存器(Memory Buffer Register)

MBR 包含要存储到主存储器中的值或从主存储器中读取的最后一个值，MBR 连接到系统总线的数据线。在我们的设计中，MBR 有 16 位，下面是具体结构：

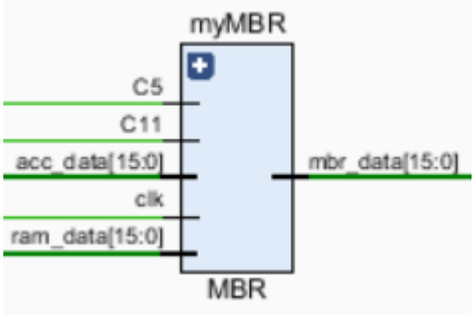


图 9 MBR 模型图

2.3. PC 程序计数器(Program Counter)

PC 监控着在程序中使用过的指令的地址，在我们的设计中，PC 有 8 位，下面是具体结

构：

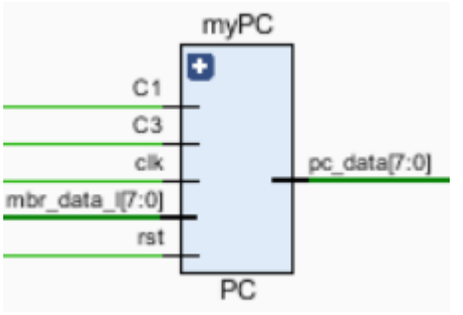


图 10 PC 模型图

2.4. IR 指令寄存器(Instruction Register)

IR 包含指令的操作码部分，在我们的设计中，IR 有 8 位，下面是具体结构：

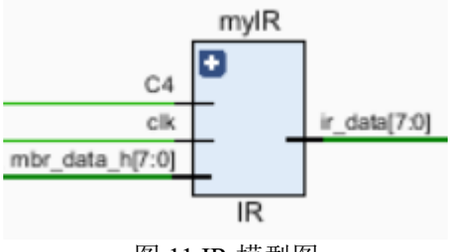


图 11 IR 模型图

2.5. BR 缓冲寄存器 (Buffer Register)

BR 被用来作为 ALU 的一个输入，存储着 ALU 的一个的操作数。在我们的设计中，BR 有 16 位，下面是具体结构：

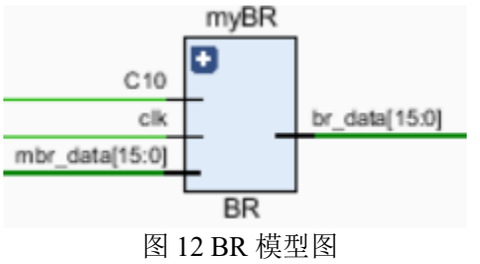


图 12 BR 模型图

2.6. ACC 累加器(Accumulator)

ACC 存储着 ALU 的另一个操作数，并且 ACC 常用来存储 ALU 的计算结果。在我们的设计中，ACC 有 16 位，下面是具体结构：

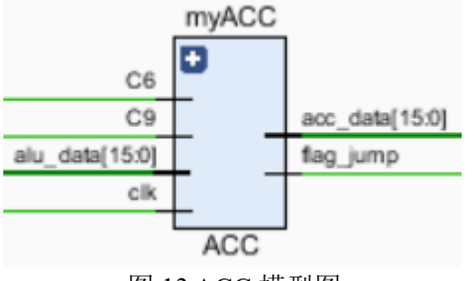


图 13 ACC 模型图

2.7. 微控制器(MCU)

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(ROM)中，产生正确执行指令集所需的所有控制信号，这里我们的控制信号总共有 20 位，微指令包含一些同时执行的微操作。

下面是 MCU 的具体结构：

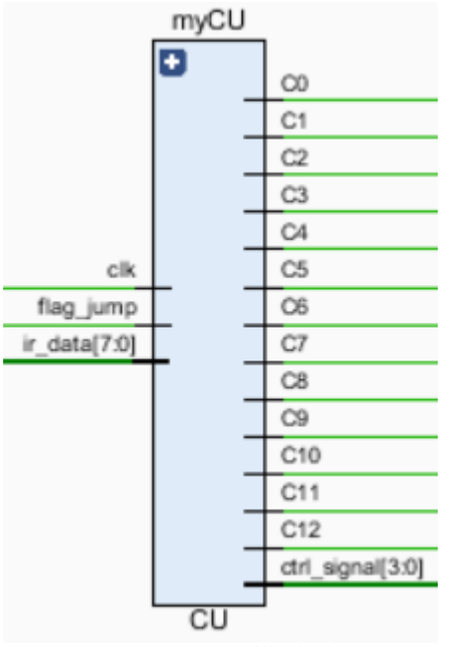


图 14 MCU 模型图

MCU 其中包含 CAR，CBR 和 ROM。MCU 的整体连接图如下：

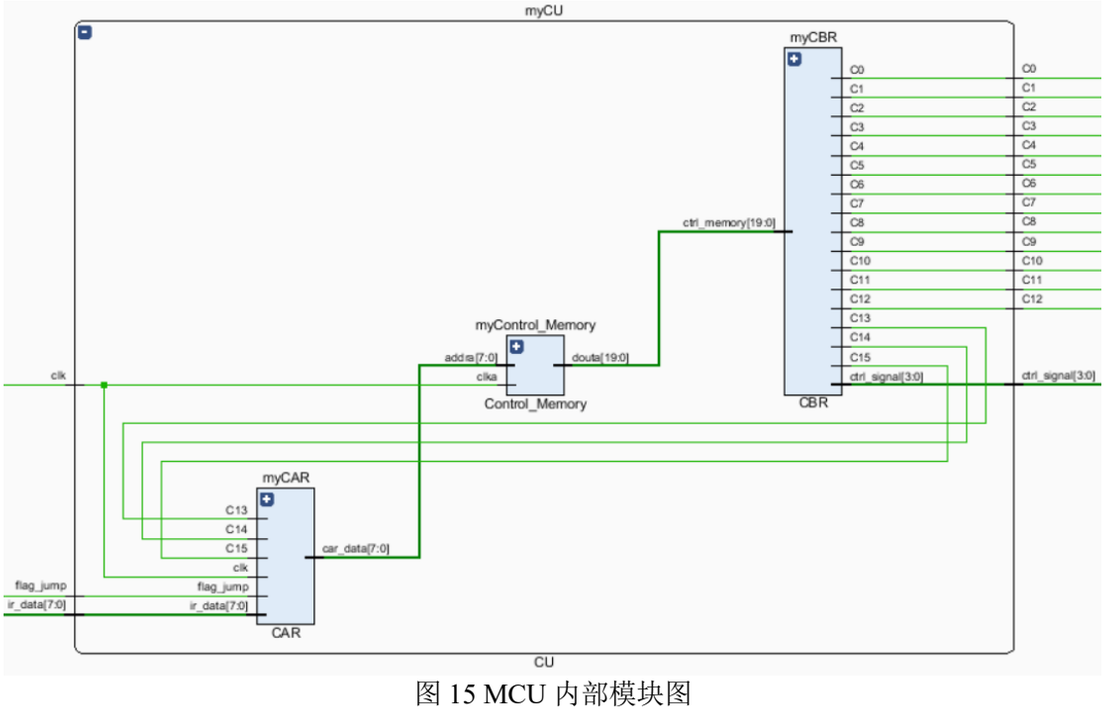


图 15 MCU 内部模块图

一组微指令存储在控制存储器中。控制地址寄存器(CAR)包含下一个要读取的微指令的地址。当从控制存储器中读取一条微指令时，它被传输到控制缓冲寄存器(CBR)中。然后由 CBR 输出的一系列控制信号去控制 CPU 内其他寄存器模块的运行，从而实现程序的执行。

2.8. 显示模块 (Display)

我们将运算结果转化为相应的段码，并指定相应的位码以实现十六进制数的数码管显示。下面是具体结构：

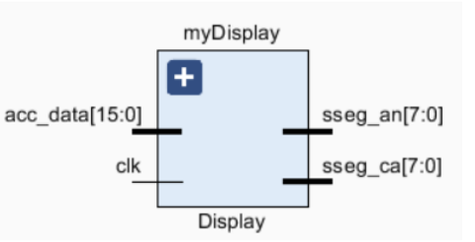


图 16 Display 模型图

下面以 LOAD 指令为例来说明这个过程。LOAD 指令控制流程如下图。

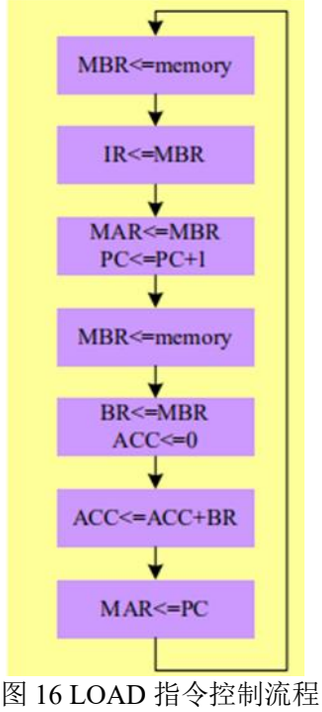


图 16 LOAD 指令控制流程

然后我们需要按照下表，根据相应的微操作确定相应的控制信号。

Control Signal	Microoperation
C0	Memory←MAR
C1	PC←PC+1
C2	MAR←PC
C3	PC←MBL_L
C4	IR←MBR_H
C5	MBR←Memory
C6	ACC←0
C7	ALU←ACC
C8	MAR←MBR_L
C9	ACC←ALU
C10	BR←MBR
C11	MBR←ACC

C12	Memory←MBR
C13	CAR←IR
C14	CAR←CAR+1
C15	CAR←0
ctrl_signal[3:0](C19:C16)	
01H	ADD
02H	SUB
03H	MPY
04H	DIV
05H	AND
06H	OR
07H	HNOT
08H	SLL/SLA
09H	SRL
0AH	SRA

然后我们参照 LOAD 指令并根据我们自己设计的结构以及控制流程，得到 LOAD 的微操作及对应的控制信号如下：

Instruction	**Opcode**	**CAR 即 Control Memory 中的地址**	**每个时钟周期内**	
			控制信号	**微指令**
LOAD	02H	07H	04060H	MBR←Memory ACC←0 CAR←CAR+1
		08H	04480H	BR←MBR ALU←ACC CAR←CAR+1
		09H	14000H	ALU←ALU+BR CAR←CAR+1
		0AH	08204H	ACC←ALU CAR←0 MAR←PC

同理我们参照 LOAD 指令得到其他所有指令的微程序，如下表。因为执行每条指令都需要 FETCH 的操作，所以这里我们将所有指令的 FETCH 操作都提取出来放在第一条指令里面，以后无论什么指令都从第一条指令开始执行，执行完以后在根据具体的指令执行对应的微操作即可。

Instruction	**Opcode**	**CAR 即 ROM 中的地址**	**每个时钟周期内**	
			控制信号	**微指令**
FETCH		00H	04020H	MBR←Memory

				CAR←CAR+1
		01H	04010H	IR←MBR_H CAR←CAR+1
		02H	04102H	PC←PC+1 MAR←MBR_L CAR←CAR+1
		03H	02000H	CAR jump to certain address of Control Memory That is CAR←IR
STORE	01H	04H	04800H	MBR←ACC CAR←CAR+1
		05H	05001H	Memory←MBR Memory←MAR CAR←CAR+1
		06H	08004H	CAR←0 MAR←PC
LOAD	02H	07H	04060H	MBR←Memory ACC←0 CAR←CAR+1
		08H	04480H	BR←MBR ALU←ACC CAR←CAR+1
		09H	14000H	ALU←ALU+BR CAR←CAR+1
		0AH	08204H	ACC←ALU CAR←0 MAR←PC
ADD	03H	0BH	04020H	MBR←Memory CAR←CAR+1
		0CH	04480H	BR←MBR ALU←ACC CAR←CAR+1
		0DH	14000H	ALU←ALU+BR CAR←CAR+1
		0EH	08204H	ACC←ALU CAR←0 MAR←PC
SUB	04H	0FH	04020H	MBR←Memory CAR←CAR+1
		10H	04480H	BR←MBR ALU←ACC CAR←CAR+1
		11H	24000H	ALU←ALU-BR CAR←CAR+1
		12H	08204H	ACC←ALU CAR←0 MAR←PC

JMPGEZ	05H (发生跳转 flag=1)	13H	04008H	PC←MBR_L CAR←CAR+1
		14H	08004H	CAR←0 MAR←PC
JUMP	06H	13H	04008H	PC←MBR_L CAR←CAR+1
		14H	08004H	CAR←0 MAR←PC
HALT	07H	15H	00000H	NONE
MPY	08H	16H	04020H	MBR←Memory CAR←CAR+1
		17H	04480H	BR←MBR ALU←ACC CAR←CAR+1
		18H	34000H	ALU← (ALU*BR)_L MR← (ALU*BR)_H CAR←CAR+1
		19H	08204H	ACC←ALU CAR←0 MAR←PC
DIV	09H	1AH	04020H	MBR←Memory CAR←CAR+1
		1BH	04480H	BR←MBR ALU←ACC CAR←CAR+1
		1CH	44000H	ALU←ALU/BR DR←ALU%BR CAR←CAR+1
		1DH	08204H	ACC←ALU CAR←0 MAR←PC
AND	0AH	1EH	04020H	MBR←Memory CAR←CAR+1
		1FH	04480H	BR←MBR ALU←ACC CAR←CAR+1
		20H	54000H	ALU←ALU and BR CAR←CAR+1
		21H	08204H	ACC←ALU CAR←0 MAR←PC
OR	0BH	22H	04020H	MBR←Memory CAR←CAR+1
		23H	04480H	BR←MBR ALU←ACC

				CAR←CAR+1
		24H	64000H	ALU←ALU or BR CAR←CAR+1
		25H	08204H	ACC←ALU CAR←0 MAR←PC
NOT	0CH	26H	04020H	MBR←Memory CAR←CAR+1
		27H	04400H	BR←MBR CAR←CAR+1
		28H	74000H	ALU←not(BR) CAR←CAR+1
		29H	08204H	ACC←ALU CAR←0 MAR←PC
SLL/SLA	0DH	2AH	04080H	ALU←ACC CAR←CAR+1
		2BH	84000H	SLL/SLA ALU CAR←CAR+1
		2CH	08204H	ACC←ALU CAR←0 MAR←PC
SRL	0EH	2DH	04080H	ALU←ACC CAR←CAR+1
		2EH	94000H	SRL ALU CAR←CAR+1
		2FH	08204H	ACC←ALU CAR←0 MAR←PC
SRA	0FH	30H	04080H	ALU←ACC CAR←CAR+1
		31H	A4000H	SRA ALU CAR←CAR+1
		32H	08204H	ACC←ALU CAR←0 MAR←PC

2.9. ALU 算术逻辑单元

算术逻辑单元(ALU)是完成基本算术和逻辑运算的计算单元。几乎所有的操作都是将相应的数据带到 ALU 来进行处理，然后把结果取出。在我们的设计中，ALU 支持以下操作。

Operations	Explanations
ADD	(ACC)←(ACC)+(BR)
SUB	(ACC)←(ACC)-(BR)
AND	(ACC)←(ACC)and(BR)
OR	(ACC)←(ACC)or(BR)
NOT	(ACC)←Not(ACC)
SLL/SLA	(ACC)←Shift (ACC) to Left 1 bit, Logic Shift/Arithmetic Shift
SRL	(ACC)←Shift (ACC) to Right 1 bit, Logic Shift
SRA	(ACC)←Shift (ACC) to Right1 bit, Arithmetic Shift

ALU 模块的结构如下：

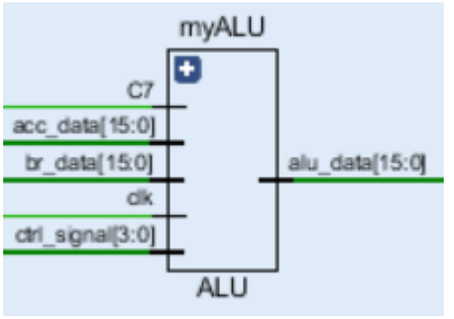
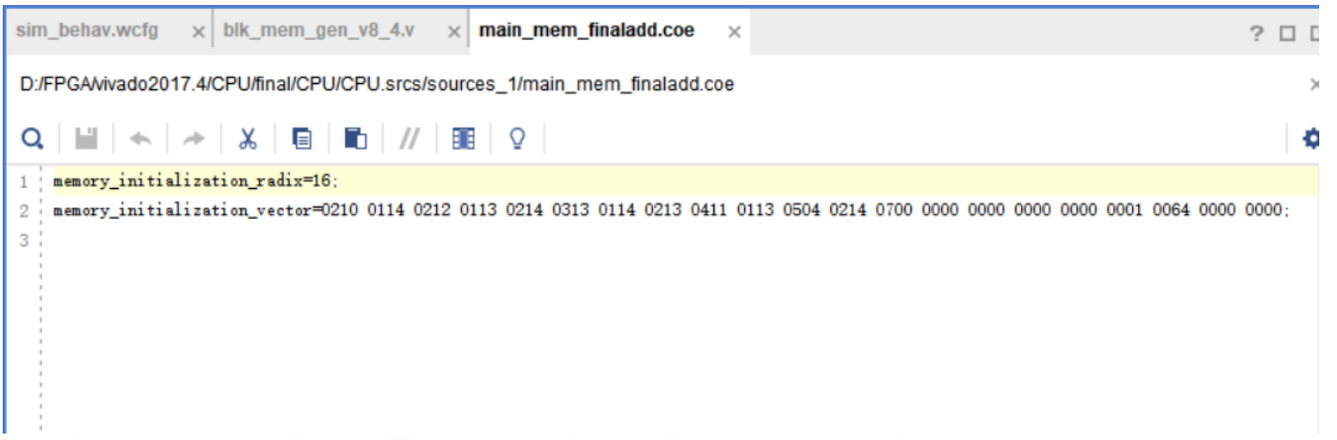


图 17 ALU 模型图

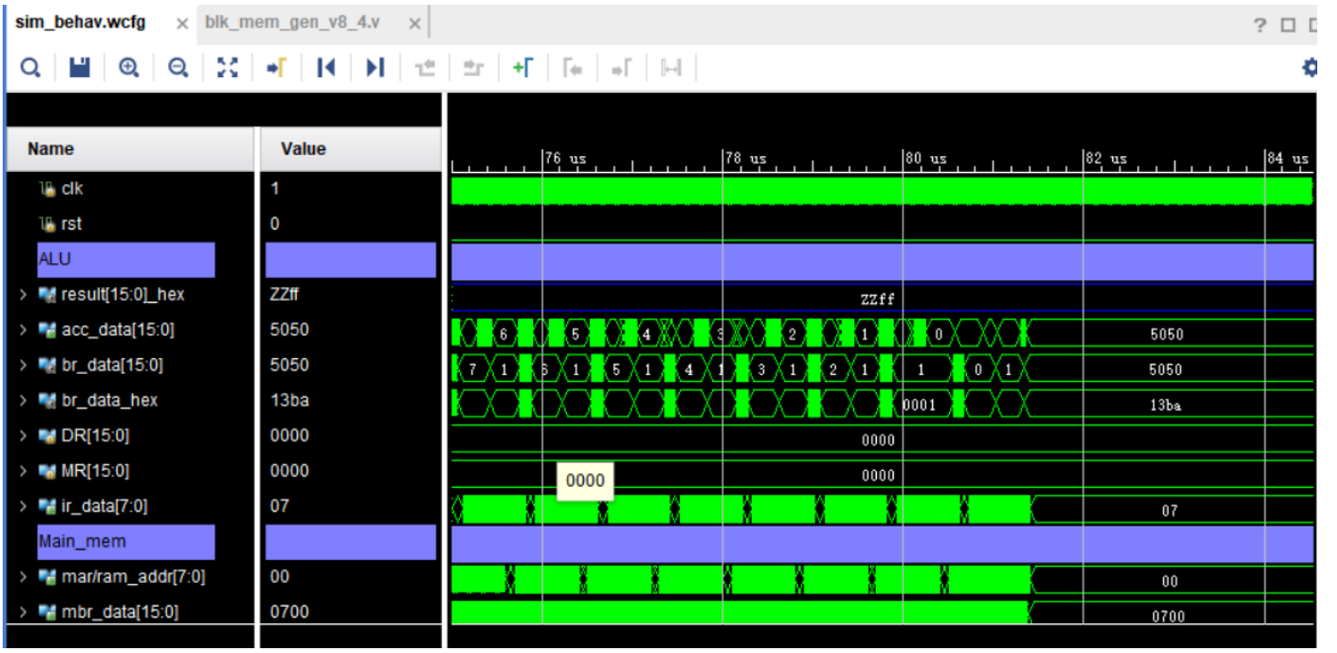
综上，在完成了 CPU 内部各个模块的设计之后，将其外部接口互相连接起来，进行封装，设置好 CPU 的输入输出引脚，一个 CPU 模块就设计好了。

四、仿真与测试

1. 加法：实现 100 到 1 的加法.
下图为指令截图



首先保存 100，然后实现加法，再减一，判断循环，直到加完 0，不满足循环，停止。
下图为仿真结果图，可以看见，最终结果为 5050，符合预期。



2. 乘法：实现带负数的乘法（6*-5）.
下图为指令截图

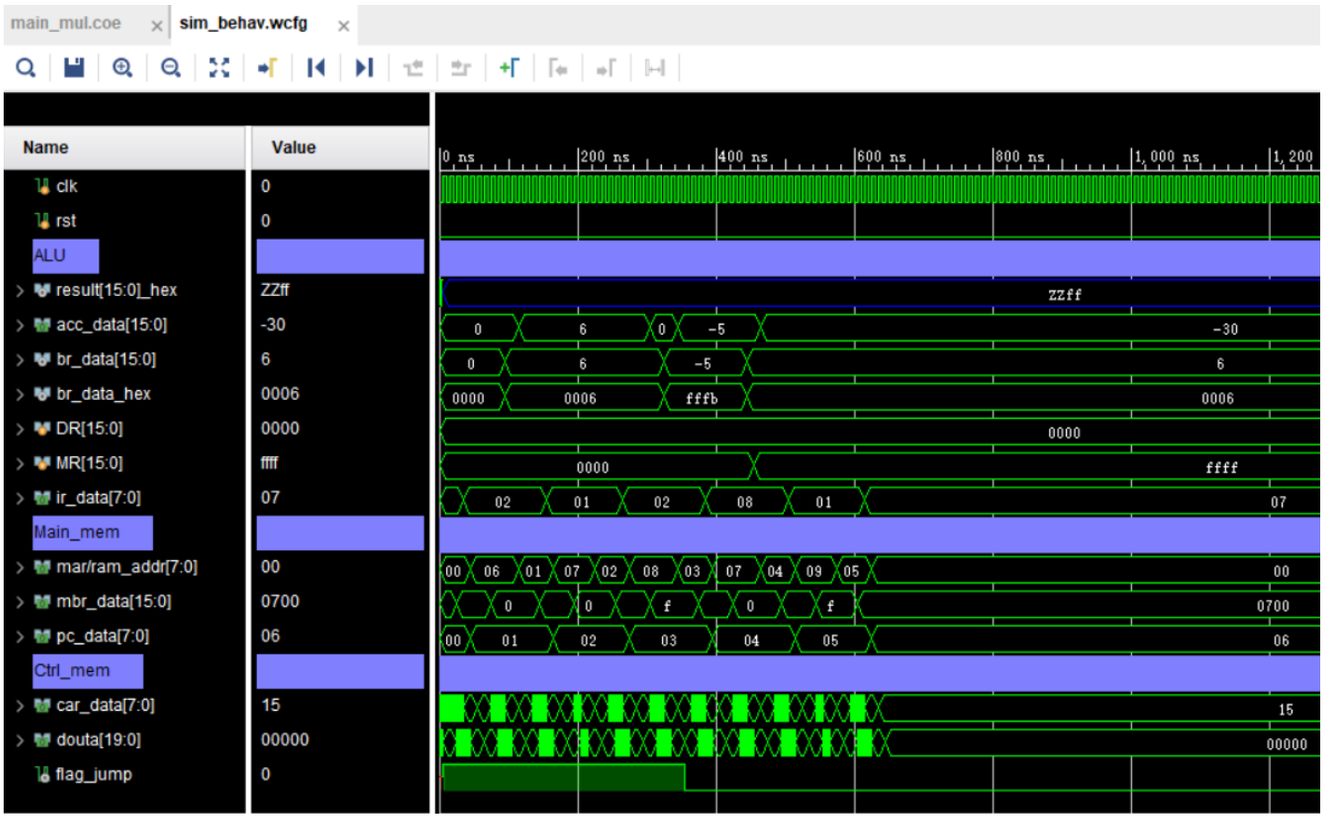
main_mul.coe × sim_behav.wcfg ×

D:/final/CPU/CPU.srcs/sources_1/main_mul.coe

```
1  memory_initialization_radix=16;
2  memory_initialization_vector=0206 0107 0208 0807 0109 0700 0006 0000 fffb 0000;
3
```

首先取出 06 位置处的正数 6，转存到 07 位置处；接着取出 08 位置处的负数-5，并与 07 位置处的正数 6 相乘，存储到 09 位置处，遇到 HALT 停止。

下图为仿真结果图，可以看见，最终结果为-30，符合预期。



五、实验总结

1. 遇到的问题和解决方法

1.1. 因为数字系统在运行的过程中是并行执行，所以有的时候我们需要对指令的微操作顺序执行加一些限制（例如我们通过判断是不是对应的操作码最后的一个微操作来限制微操作地址的增加），不然容易出现一些微操作多执行的情况，这个就是我们在实验中遇到的一个困难。

1.2. 因为引用现成 IP 核进行 Block Memory 设计的时候勾选了输出缓存寄存器的设置，导致主存储器输出数据有一定的延时，导致 CPU 执行的时序发生错误，解决方法就是去掉勾选的设置，就能使主存储器及时响应数据读写操作。

1.3. 在仿真实验中，发现 ALU 模块的 OR 运算得到的最终结果和真实计算器得到的结果不同，通过查验发现代码写错了，将按位或'|'写成了逻辑或'||'导致最后计算结果的错误。

六、实验代码

1.顶层模块

```
module myComputer(
    input clk,
    input rst,
//    output [15:0] result,
    output [7:0]sseg_ca,
    output [7:0]sseg_an
    );

    wire [15:0] write_bus;
    wire [15:0] read_bus;
    wire [7:0] mainmem_addr;
    wire write_en;

    CPU myCPU(
        .clk(clk),
        .rst(rst),
//        .output_data(result),
        .mainmem_data(read_bus),
        .mainmem_addr(mainmem_addr),
        .tomainmem_data(write_bus),
        .write_en(write_en),
        .sseg_ca(sseg_ca),
        .sseg_an(sseg_an)
        );

    Memory myMemory (
          .clka(clk),    // input wire clka
          .wea(write_en),      // input wire [0 : 0] wea
          .addra(mainmem_addr),  // input wire [7 : 0] addra
          .dina(write_bus),    // input wire [15 : 0] dina
          .douta(read_bus)  // output wire [15 : 0] douta
        );

endmodule
```

2.CPU

```
module CPU(
    input clk,
    input rst,
//    output [15:0]output_data,
    input [15:0] mainmem_data,
    output [7:0] mainmem_addr,
    output [15:0] tomainmem_data,
    output write_en,
    output [7:0]sseg_ca,
    output [7:0]sseg_an
    );
    wire [15:0] mbr_data;
    //wire [15:0] memory_data;
    wire [7:0] mbr_data_h=mbr_data[15:8];
    wire [7:0] mbr_data_l=mbr_data[7:0];
    //wire [7:0] mar_data;
    wire [7:0] pc_data;
    wire [7:0] ir_data;
    wire [15:0] acc_data;
    wire [15:0] alu_data;
    wire [15:0] br_data;
    wire [3:0] ctrl_signal;
    wire C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,flag_jump,acc_rst;

//    assign output_data=acc_data;//ACC 作为输出
    assign tomainmem_data=mbr_data;
```

```
    assign write_en=C0&C12;
    assign acc_rst=C6|rst;

    //例化
    MBR myMBR(
    .clk(clk),
    .C5(C5),
    .C11(C11),
    .ram_data(mainmem_data),
    .acc_data(acc_data),
    .mbr_data(mbr_data)
    );

    MAR myMAR(
    .clk(clk),
    .C2(C2),
    .C8(C8),
    .mbr_data_l(mbr_data_l),
    .pc_data(pc_data),
    .ram_addr(mainmem_addr)
    );

   PC myPC(
       .clk(clk),
       .C3(C3),
       .rst(rst),
       .C1(C1),
       .mbr_data_l(mbr_data_l),
       .pc_data(pc_data)
       );

    IR myIR(
        .clk(clk),
        .C4(C4),
        .mbr_data_h(mbr_data_h),
        .ir_data(ir_data)
        );

     CU myCU(
           .flag_jump(flag_jump),
           .clk(clk),
           .ir_data(ir_data),
           .ctrl_signal(ctrl_signal),
           .C0(C0), .C1(C1), .C2(C2), .C3(C3), .C4(C4), .C5(C5), .C6(C6), .C7(C7), .
C8(C8), .C9(C9),
           .C10(C10), .C11(C11), .C12(C12)
            );

     BR myBR(
           .clk(clk),
           .C10(C10),
           .mbr_data(mbr_data),
           .br_data(br_data)
           );

      ALU myALU(
              .clk(clk),
              .br_data(br_data),
              .C7(C7),
              .acc_data(acc_data),
              .ctrl_signal(ctrl_signal),
              .alu_data(alu_data)
              );

      ACC myACC(
              .clk(clk),
              .C9(C9),
```

```
            .C6(acc_rst),//reset
            .alu_data(alu_data),
            .flag_jump(flag_jump),
            .acc_data(acc_data)
            );
     Display myDisplay(
        .clk(clk),
        .acc_data(acc_data),
        .sseg_ca(sseg_ca),
        .sseg_an(sseg_an)
     );
endmodule
```

3.MBR

```
module MBR(
    input clk,
    input C5,
    input C11,
    input [15:0] ram_data,
    input [15:0] acc_data,
    output reg [15:0] mbr_data=16'h0000
    //output [7:0] mbr_data_h,
    //output [7:0] mbr_data_l
    );

    always@(posedge clk)
    begin
        if(C5) mbr_data<=ram_data;
        else if(C11) mbr_data<=acc_data;
    end

  //  assign mbr_data_h=mbr_data[15:8];
   // assign mbr_data_l=mbr_data[7:0];

endmodule
```

4.MAR

```
module MAR(
    input clk,
    input C2,
    input C8,
    input [7:0] mbr_data_l,
    input [7:0] pc_data,
    output reg [7:0] ram_addr=8'h00
    );
    always@(posedge clk)
    begin
        if(C2) ram_addr<=pc_data;
        else if(C8) ram_addr<=mbr_data_l;
    end
endmodule
```

5.PC

```
module PC(
    input clk,
    input C3,
    input rst,
    input C1,
    input [7:0] mbr_data_l,
    output reg [7:0] pc_data=8'h00
    );
    always@(posedge clk)
    begin
        if(rst) pc_data<=8'h00;
        else if(C1) pc_data<=pc_data+1;
```

```
        else if(C3) pc_data<=mbr_data_l;
    end

endmodule
```

6.IR

```
module IR(
    input clk,
    input C4,
    input [7:0] mbr_data_h,
    output reg [7:0] ir_data=8'h00
    );
    always@(posedge clk)
    begin
        if(C4) ir_data<=mbr_data_h;
    end
endmodule
```

7.CU

```
module CU(
    input flag_jump,clk,
    input [7:0] ir_data,
    output  [3:0] ctrl_signal,
    output  C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11, C12
    );
    wire C13, C14, C15;
    wire [7:0] ctrl_addr;
    wire [19:0] ctrl_data;
    CAR myCAR(.clk(clk), .C13(C13), .C14(C14), .C15(C15), .ir_data(ir_data),
    .flag_jump(flag_jump),
    .car_data(ctrl_addr)
    );

    CBR myCBR(.ctrl_memory(ctrl_data),
    .C0(C0), .C1(C1), .C2(C2), .C3(C3), .C4(C4), .C5(C5), .C6(C6), .C7(C7), .C8(C8)
, .C9(C9),
    .C10(C10), .C11(C11), .C12(C12), .C13(C13), .C14(C14), .C15(C15),
    .ctrl_signal(ctrl_signal)
     );

     Control_Memory myControl_Memory (
        .clka(clk),    // input wire clka
        .addra(ctrl_addr),  // input wire [7 : 0] addra
        .douta(ctrl_data)  // output wire [19 : 0] douta
      );

Endmodule
```

8.CAR

```
module CAR(
    input clk,
    input C13,
    input C14,
    input C15,
    input [7:0] ir_data,
    input flag_jump,
    output reg [7:0] car_data=8'h00
    );
    always@(posedge clk)
    begin
        if(C15) car_data<=8'h00;//'Reset
        else
if(C14&&car_data!=8'h03&&car_data!=8'h06&&car_data!=8'h0A&&car_data!=8'h0E&&car_dat
a!=8'h12&&car_data!=8'h14&&car_data!=8'h15&&car_data!=8'h19&&car_data!=8'h1D&&car_d
ata!=8'h21&&car_data!=8'h25&&car_data!=8'h29&&car_data!=8'h2C&&car_data!=8'h2F&&car
_data!=8'h32)
```

```
        car_data<=car_data+1;//Increment
        else if(C13)
        begin
            case(ir_data)
                8'h01: car_data<=8'h04;//STORE X
                8'h02: car_data<=8'h07;//LOAD X
                8'h03: car_data<=8'h0B;//ADD X
                8'h04: car_data<=8'h0F;//SUB X
                8'h05:
                begin
                    if(flag_jump) car_data<=8'h13;//JMPGEZ X
                    else car_data<=8'h14;//Not Jump
                end
                8'h06: car_data<=8'h13;//JUMP X
                8'h07: car_data<=8'h15;//HALT
                8'h08: car_data<=8'h16;//MPY X
                8'h09: car_data<=8'h1A;//DIV X
                8'h0A: car_data<=8'h1E;//AND X
                8'h0B: car_data<=8'h22;//OR X
                8'h0C: car_data<=8'h26;//NOT X
                8'h0D: car_data<=8'h2A;//SLL/SLA
                8'h0E: car_data<=8'h2D;//SRL
                8'h0F: car_data<=8'h30;//SRA
                8'h00: car_data<=8'h14;//FETCH OVER
                default:;
            endcase
        end
    end
endmodule
```

9.CBR

```
module CBR(
    input [19:0]ctrl_memory,
    output  C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,
    output  [3:0] ctrl_signal
    );
   assign C0=ctrl_memory[0];
   assign C1=ctrl_memory[1];
   assign C2=ctrl_memory[2];
   assign C3=ctrl_memory[3];
   assign C4=ctrl_memory[4];
   assign C5=ctrl_memory[5];
   assign C6=ctrl_memory[6];
   assign C7=ctrl_memory[7];
   assign C8=ctrl_memory[8];
   assign C9=ctrl_memory[9];
   assign C10=ctrl_memory[10];
   assign C11=ctrl_memory[11];
   assign C12=ctrl_memory[12];
   assign C13=ctrl_memory[13];
   assign C14=ctrl_memory[14];
   assign C15=ctrl_memory[15];
   assign ctrl_signal=ctrl_memory[19:16];
endmodule
```

10.BR

```
module BR(
    input clk,
    input C10,
    input [15:0] mbr_data,
    output reg [15:0] br_data=16'h0000
    );
    always@(posedge clk)
    begin
        if(C10) br_data<=mbr_data;
    end
```

```
endmodule

11.ALU
module ALU(
input clk,
input signed [15:0] br_data,
input C7,
input signed [15:0] acc_data,
input [3:0] ctrl_signal,
output reg signed [15:0] alu_data=16'h0000
    );

    reg signed [15:0] DR=16'h0000;
    reg signed [15:0] MR=16'h0000;

    reg signed [15:0] acc_temp=16'h0000;
    reg signed [31:0] result_temp=32'h00000000;
    always@(posedge clk)
    begin
        if(C7) acc_temp<=acc_data;
        case(ctrl_signal)

            4'h1:    //ADD
            begin
                acc_temp=acc_temp+br_data;
                alu_data=acc_temp;
            end

            4'h2:   //SUB
            begin
                acc_temp=acc_temp-br_data;
                alu_data=acc_temp;
            end

            4'h3:   //MPY
            begin
                result_temp=acc_temp*br_data;
                acc_temp=result_temp[15:0];
                alu_data=acc_temp;
                MR=result_temp[31:16];
            end

            4'h4:   //DIV
            begin
                if(br_data)
                begin
                    DR=acc_temp%br_data;//取余
                    acc_temp=(acc_temp-DR)/br_data;
                    alu_data=acc_temp;
                end
            end

            4'h5:   //AND
            begin
                acc_temp=acc_temp&br_data;
                alu_data=acc_temp;
            end

            4'h6:   //OR
            begin
                acc_temp=acc_temp|br_data;
                alu_data=acc_temp;
            end

            4'h7:   //NOT
            begin
                acc_temp=~br_data;
```

```
            alu_data=acc_temp;
        end

        4'h8:   //SLL or SLA shift left logic/arithmetic
        begin
            acc_temp={acc_temp[14:0],1'b0};
            alu_data=acc_temp;
        end

        4'h9:  //SRL shift right logic
        begin
            acc_temp={1'b0,acc_temp[15:1]};
            alu_data=acc_temp;
        end

        4'hA://SRA shift right arithmetic
        begin
            acc_temp={acc_temp[15],acc_temp[15:1]};
            alu_data=acc_temp;
        end

        /*4'hf://reset
        begin
            acc_temp=16'h0000;
            alu_data=acc_temp;
        end*/

        default:;
      endcase
    end

endmodule
```

12.ACC

```
module ACC(
    input clk,
    input C9,
    input C6,//reset
    input signed [15:0] alu_data,
    output reg flag_jump,
    output reg signed [15:0] acc_data=16'h0000
    );
    always@(posedge clk)
    begin
        if(C6) acc_data<=0;
        else if(C9) acc_data<=alu_data;
        if(acc_data>=0) flag_jump<=1;
        else flag_jump<=0;
    end
endmodule
```

13.Display

```
module Display(
    input clk,
    input [15:0] acc_data,
    output reg [7:0] sseg_ca,
    output reg [7:0] sseg_an
    );
    reg [3:0]disp_dat;
    reg [2:0]disp_bit;
    //150ms 刷新时钟
    reg [14:0] sx;
    reg clk_sx;
    always @ (posedge clk)
    begin
    //150ms 刷新时钟
```

```
        if(sx==1500)
        begin
            clk_sx=~clk_sx;
            sx<=0;
        end
        else
            sx=sx+1;
    end
    always@(posedge clk)
    begin
        case(disp_dat)
        4'h0:sseg_ca=8'hc0;
        4'h1:sseg_ca=8'hf9;
        4'h2:sseg_ca=8'ha4;
        4'h3:sseg_ca=8'hb0;
        4'h4:sseg_ca=8'h99;
        4'h5:sseg_ca=8'h92;
        4'h6:sseg_ca=8'h82;
        4'h7:sseg_ca=8'hf8;
        4'h8:sseg_ca=8'h80;
        4'h9:sseg_ca=8'h90;
        4'ha:sseg_ca=8'h88;
        4'hb:sseg_ca=8'h83;
        4'hc:sseg_ca=8'hc6;
        4'hd:sseg_ca=8'ha1;
        4'he:sseg_ca=8'h86;
        4'hf:sseg_ca=8'h8e;
        default:sseg_ca=8'hff;
        endcase
    end
    always @ (posedge clk_sx)
    begin
        if(disp_bit>=3)
            disp_bit=0;
        else
            disp_bit=disp_bit+1;
        case(disp_bit)
            3'h0:
        begin
            sseg_an=8'b11111110;
            disp_dat=acc_data[3:0];
        end
        3'h1:
        begin
        sseg_an=8'b11111101;
        disp_dat=acc_data[7:4];
        end
        3'h2:
        begin
        sseg_an=8'b11111011;
        disp_dat=acc_data[11:8];
        end
        3'h3:
        begin
        sseg_an=8'b11110111;
        disp_dat=acc_data[15:12];
        end
        default:
            begin
                sseg_an=8'b11111111;
                disp_dat=0;
            end
        endcase

    end
endmodule
```