目录

一、设计要求

- 设计并实现一个多周期流水 MIPS32 CPU
- 五段流水、可以处理冲突
- 三种类型的指令若干条
- 下载到 FPGA 上进行验证

二、使用工具

Xilinx ISE 14.7

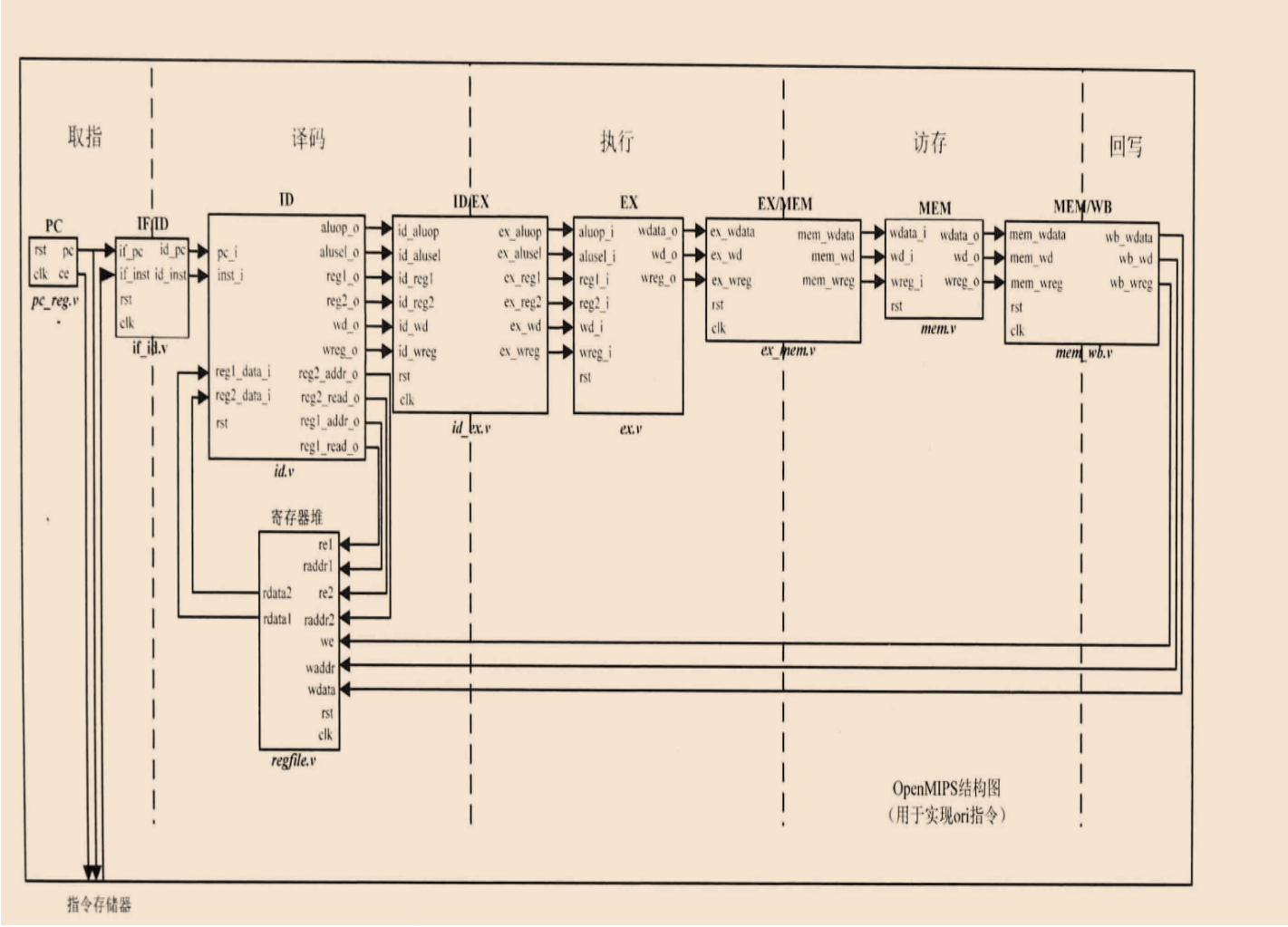
三、设计说明

- 本次课设实现了五段流水线的多周期 CPU
- 实现了 data forwarding 以解决数据相关
- stall 机制使得流水线暂停
- 实现有符号的加减法
- 实现 booth 算法实现有符号乘法
- 共实现 38 条指令

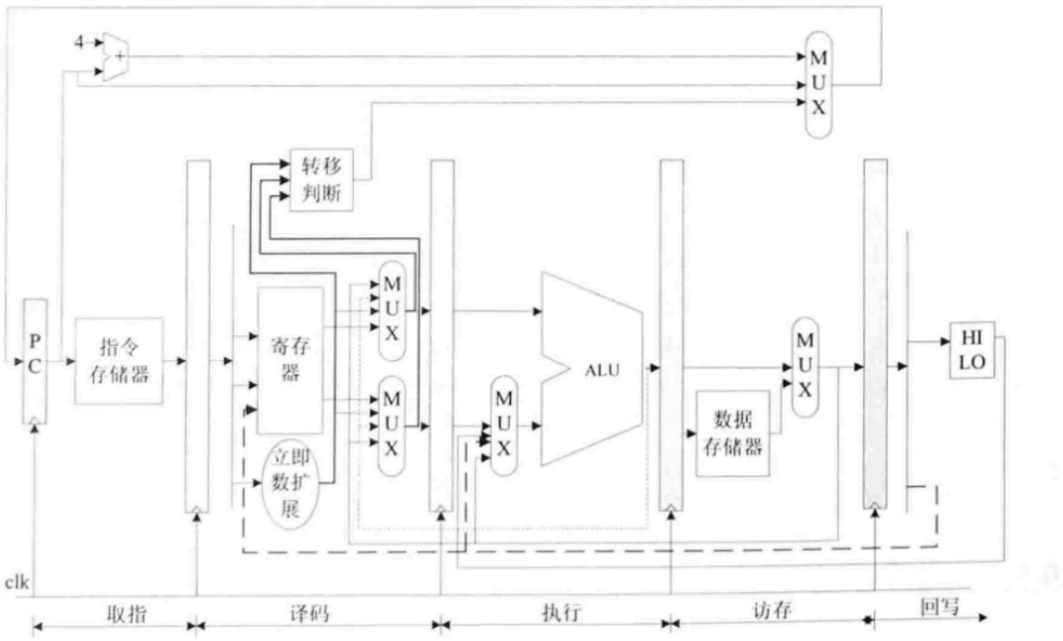
四、设计概览

I. 数据通路图

基本的五段流水数据通路：



基本实现冲突解决以及一些功能后的数据通路图



II. 总体设计

课程设计的要求是实现多周期的 MIPS32 位 CPU，多周期即是将一条指令的执行拆分为多段执行，每段为一个周期的执行时间。

在本次设计中采用的是五段流水线机制，即

- 取指（IF）：从指令存储器中读出指令，同时确定下一条指令的地址
- 译码（ID）：对指令进行译码，从通用寄存器中读出要使用的寄存器的值，如果是转移指令，并且满足转移条件，则需要给出转移目标，作为新的指令地址。
- 执行（EX）：按照译码段给出的操作数、运算类型，进行运算，给出运算结果。如果是 load/store 指令，还需要进行计算 load/store 的目标地址。
- 访存（MEM）：如果是 load/store 指令，则在此阶段需要进行数据存储器的访问。否则，仅仅是中间媒介，将 ex 段的执行结果传送到 wb 阶段去。
- 写回（WB）：将运算结果保存到目标寄存器。

由于时间有限，所以本次设计采用的流水线都是基于同一时钟频率下执行的。为了顺利实现数据和指令的传播，引入流水线寄存器模块即 IF_ID、ID_EX、EX_MEM、MEM_WB 在每一个时钟的上升沿进行数据的传送。未进行异常的实现。全局引入初始化信号 rst，用于初始化整个流水线。

在代码实现方面，为了便于后面调试和写代码，定义了 defines.v 文件，进行一些常量的定义，如数据长度、地址长度、操作符等等。

III. 几个关键的问题：

1. 数据相关

在《计算机体系结构》课程中我学习到，由于一个时钟周期内流水线只会执行一个阶段的操作，而且多个阶段的操作是并行的。那么对于某些指令来说发生一些数据相关的问题。例如：

```
ori $1,$0,0x1100
ori $2,$1,0x0020
```

在这两条指令中，指令 2 的源操作数需要在指令 1 写回后才可以进行读取。而由于流水线的机制，指令 2 在 id 段就需要进行取操作数的操作，但是指令 1 则是在 wb 段将数据写回到目的寄存器。因此指令 2 读出的操作数不是指令想要的那个数值，这就产生了数据相关。

解决数据相关可以让流水线暂停, 等指令 1 完成后再进行指令 2 的 id 操作。但是这样非常影响效率, 因此这里引入 Data Forwarding 机制, 即当数据 available 时就将数据送给需要使用的指令的 id 段。

这里有三种：

- 相邻指令的数据相关：从前一条指令的 ex 段将数据送入 id 即可。
- 相隔一条指令的数据相关：将冲突指令的 mem 段将数据送入 id。
- 相隔两条指令的数据相关：在 wb 阶段根据寄存器的设计解决。

还有一种数据相关是由于 load 指令产生的，但是 load 指令需要在 wb 段进行写会，所以无法通过 Data Forwarding 技术进行消除，这就牵涉到流水线暂停机制，在后面介绍。

2. 转移指令

转移指令即跳转和分支指令。其中跳转指令不需要进行判断，直接将 pc 的值修改为目标地址，而分支指令则需要判断是否满足条件，再进行决定是否需要修改 pc 的值。

这里就产生了新的问题，按照正常顺序执行的指令，pc 值在 if 段送完指令后就自动加 4，但是由于有转移类指令的存在，所以不知道 pc 的正确值，因此

影响了下一条指令的 id 段的执行。可以采用分支预测的方式进行判断，但由于时间有限实现较为困难，所以我采用的是在 id 段就将分支指令执行完毕，根据结果判断是否需要进行修改 pc 的值。

3. 流水线暂停

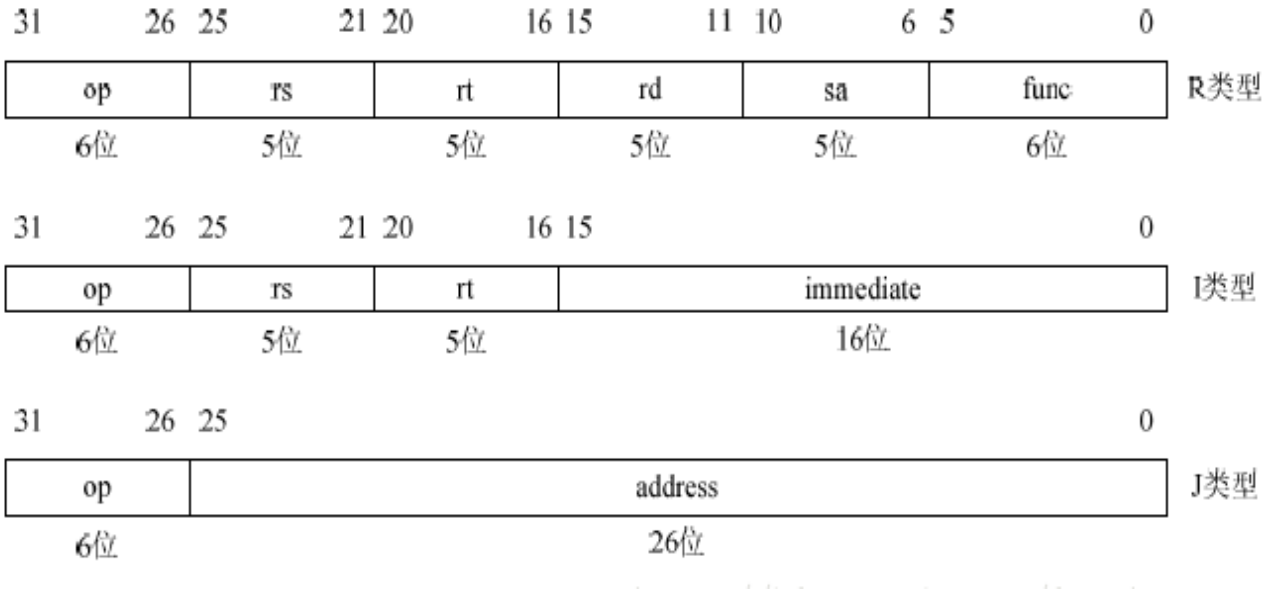
前面提到的由于 load 指令需要在 wb 阶段写回寄存器，所以需要引入流水线暂停机制使得指令能够正确执行。除此之外，在后面实现的 booth 算法由于需要使用 32 个周期进行计算，同样也需要进行流水线的暂停。

在这里引入 stall 机制，根据是否有 stall 信号进行流水线的暂停。如果是 id 段需要暂停，则其之后的 if、pc、id 需要被暂停而其当前的 ex、mem、wb 仍可继续运行。而当 ex 段发出 stall 信号时，其之后的 if、pc、id、ex 需要被暂停，而其当前的 mem、wb 可以继续执行。

五、指令集架构

I. MIPS32 指令集格式

MIPS32 分为三大类指令即 R 类、J 类和 I 类指令，其指令格式如下图：



其中根据 func 的值还可以将 r 类指令进行进一步的划分，这里不多赘述

II. 本次设计中实现的指令的格式

基本的逻辑运算指令

or、xor、and、nor、ori、andi、xori

Op	31-26	25-21	20-16	15-11	10-6	5-0
Or	000000	rs	rt	rd	00000	100101
Ori	001101	rs	rt	Imm		
Xor	000000	rs	rt	rd	00000	100110
And	000000	rs	rt	rd	00000	100100
Nor	000000	rs	rt	rd	00000	100111
Xori	001100	rs	rt	imm		
Andi	001100	rs	rt	imm		

基本的移位指令

sllv、srlv、srav、sll、srl、sra

Op	31-26	25-21	20-16	15-11	10-6	5-0
Sllv	000000	rs	rt	rd	00000	000100
Srlv	000000	rs	rt	rd	00000	000110
Srav	000000	rs	rt	rd	00000	000111
Sll	000000	00000	rt	rd	sa	000000
Srl	000000	00000	rt	rd	sa	000010
Sra	000000	00000	rt	rd	sa	000011

基本的算数指令

add、addu、sub、subu、mult、multu、slt、sltu、addi、addiu、slti、sltiu

Op	31-26	25-21	20-16	15-11	10-6	5-0
Add	000000	rs	rt	rd	00000	100000
Addu	000000	rs	rt	rd	00000	100001
Sub	000000	rs	rt	rd	00000	100010
Subu	000000	rs	rt	rd	00000	100011
Mult	000000	rs	rt	00 0000 0000		011000
Multu	000000	rs	rt	00 0000 0000		011001
Slt	000000	rs	rt	rd	00000	101010
Sltu	000000	rs	rt	rd	00000	101011
Addi	001000	rs	rt	imm		
Addiu	001001	rs	rt	imm		
Slti	001010	rs	rt	imm		
Sltiu	001011	rs	rt	imm		

跳转指令

j、jal、jr、jalr

Op	31-26	25-21	20-16	15-11	10-6	5-0
J	000010	Instr_index				
Jal	000011	Instr_index				
Jr	000000	rs	00 0000 0000		hint	001000
Jalr	000000	rs	00000	rd	hint	001001

分支指令

beq、bgtz、blez、bne、bgez、bgezal、bltz

Op	31-26	25-21	20-16	15-11	10-6	5-0
Beq	000100	rs	rt	Offset		
Bgtz	000111	rs	00000	Offset		
Blez	000110	rs	00000	Offset		
Bne	000101	rs	rt	Offset		
Bgez	000001	rs	00001	Offset		
Bgezal	000001	rs	10001	Offset		
Bltz	000001	rs	00000	Offset		

Lw、sw

Op	31-26	25-21	20-16	15-11	10-6	5-0
Lw	100011	base	rt	Offset		
Sw	101011	base	rt	Offset		

指令的功能表一览

序号	指令	功能
1	Or	Gpr[rd] <- Gpr[rs] or Gpr[rt]
2	Xor	Gpr[rd] <- Gpr[rs] xor Gpr[rt]
3	And	Gpr[rd] <- Gpr[rs] and Gpr[rt]
4	Nor	Gpr[rd] <- Gpr[rs] nor Gpr[rt]
5	Ori	Gpr[rd] <- Gpr[rs] or sign_extended[imm]
6	Andi	Gpr[rd] <- Gpr[rs] and sign_extended[imm]
7	Xori	Gpr[rd] <- Gpr[rs] xor sign_extended[imm]
8	Sll	Gpr[rd] <- Gpr[rt] << sa
9	Srl	Gpr[rd] <- Gpr[rt] >> sa

10	Sra	Gpr[rd] <- Gpr[rt] >> sa (算术)
11	Sllv	Gpr[rd] <- Gpr[rt] << Gpr[rs]
12	Srlv	Gpr[rd] <- Gpr[rt] >> Gpr[rs]
13	Srav	Gpr[rd] <- Gpr[rt] >> Gpr[rs] (算术)
14	Add	Gpr[rd] <- Gpr[rt] + Gpr[rs] （signed）
15	Addu	Gpr[rd] <- Gpr[rt] + Gpr[rs] （unsigned）
16	Sub	Gpr[rd] <- Gpr[rs] - Gpr[rt] （signed）
17	Subu	Gpr[rd] <- Gpr[rs] - Gpr[rt] （unsigned）
18	Mult	Gpr[rd] <- Gpr[rs] * Gpr[rt] （signed）
19	Multu	Gpr[rd] <- Gpr[rs] * Gpr[rt] （unsigned）
20	Slt	Gpr[rd] <- (Gpr[rs] < Gpr[rt]) ? 1 : 0 (overflow)
21	Sltu	Gpr[rd] <- (Gpr[rs] < Gpr[rt]) ? 1 : 0 (no overflow)
22	Addi	Gpr[rd] <- Gpr[rs] + sign_extended(imm) (signed)
23	Addiu	Gpr[rd] <- Gpr[rs] + sign_extended(imm) (unsigned)
24	Slti	Gpr[rd] <- (Gpr[rs] < sign_extend(imm)) ? 1 : 0 (溢出)
25	Sltiu	Gpr[rd] <- (Gpr[rs] < sign_extend(imm)) ? 1 : 0 (无溢出)
26	J	PC <- PC + 4 || instr_index || 00
27	Jal	Gpr[31] <- PC + 8 PC <- PC + 4 || instr_index || 00
28	Jr	PC <- Gpr[rs]
29	Jalr	Gpr[rd] <- PC + 8 PC <- Gpr[rs]
30	Beq	If rs == rt pc <- pc + 4 + offset
31	Bgtz	If rs > 0 pc <- pc + 4 + offset
32	Blez	If rs <= 0 pc <- pc + 4 + offset
33	Bne	If rs != rt pc <- pc + 4 + offset
34	Bgez	If rs >= 0 pc <- pc + 4 + offset
35	Bgezal	Gpr[31] <- PC + 8 if rs >= 0 pc <- pc + 4 + offset
36	Bltz	If rs < 0 if rs >= 0 pc <- pc + 4 + offset
37	lw	Gpr[rt] <- memory[base + offset]
38	sw	memory[base + offset] <- Gpr[rt]

六、具体模块设计

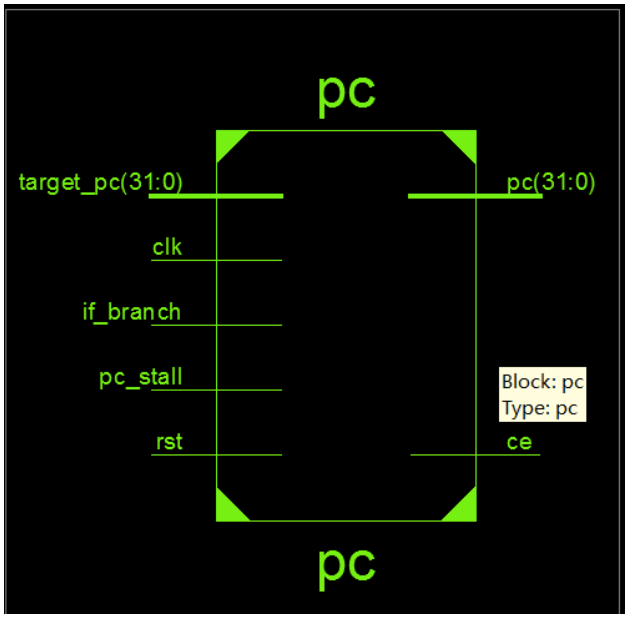
为了方便预先在 defines.v 的文件中定义，如图：

```
`define EXE_OR   6'b100101
`define EXE_XOR 6'b100110
`define EXE_NOR 6'b100111
`define EXE_ANDI 6'b001100
`define EXE_ORI  6'b001101
`define EXE_XORI 6'b001110
`define EXE_LUI 6'b001111

`define EXE_SLL  6'b000000
`define EXE_SLLV  6'b000100
`define EXE_SRL  6'b000010
`define EXE_SRLV  6'b000110
`define EXE_SRA  6'b000011
`define EXE_SRAV  6'b000111
`define EXE_SYNC  6'b001111
`define EXE_PREF  6'b110011

`define EXE_MOVZ  6'b001010
`define EXE_MOVN  6'b001011
`define EXE_MFHI  6'b010000
`define EXE_MTHI  6'b010001
`define EXE_MFLO  6'b010010
`define EXE_MTLO  6'b010011

`define EXE_SLT  6'b101010
```

I. IF

1.pc 模块

模块图如下：



PC 段进行的就是指令计数器的增加以及形成指令地址。根据指令地址从 inst_rom 中取出送到 if_id 段进行后续的操作。由于是按字节寻址，所以在获得指令的地址后需要进行右移两位将指令从指令存储器取出。关键代码如下：

```
module pc(
      input wire clk,
      input wire rst,
      input wire pc_stall, //输入信号表示pc阶段是否需要暂停
      input wire if_branch, //判断是否是分支指令的结果，pc需要
      input wire[`RegBus] target_pc, //表示如果实现的是分支预

      output reg[`InstAddrBus] pc,
      output reg ce     //片选信号

   );
   always @(posedge clk) begin
    if(rst == `RstEnable) begin
      ce <= `ChipDisable;
    end else begin
      ce <= `ChipEnable;
    end
   end

   always@(posedge clk) begin
    if(ce == `ChipDisable) begin
      pc <= 32'h00000000;  //指令寄存器禁用时，pc的值为0
    end else if(pc_stall == 1'b0) begin //如果没有stall
      if(if_branch == 1'b1) begin       //如果是分支指令，则
         pc <= target_pc;
```

在 pc 模块中，重要的输入是 clk、pc_stall、if_branch。clk 即时钟信号，pc 在每一个时钟的上升沿进行加 4 操作。Rst 是复位信号，若为 1 则将 pc 的值还原为 0。pc_stall 即表示 pc 段是否被暂停，如果被暂停则不进行任何操作。If_branch 则表示当前指令是否是转移指令，若在没有被暂停的情况下，如果是转移指令，则将 id 段译码的目标地址赋给 pc，否则则是正常情况下的 pc 的值加 4。在设计中，我未添加 if 模块，而是将 pc 和 inst_rom 组合成为 if 段，将指令送往 if_id 寄存器中实现指令的传输。

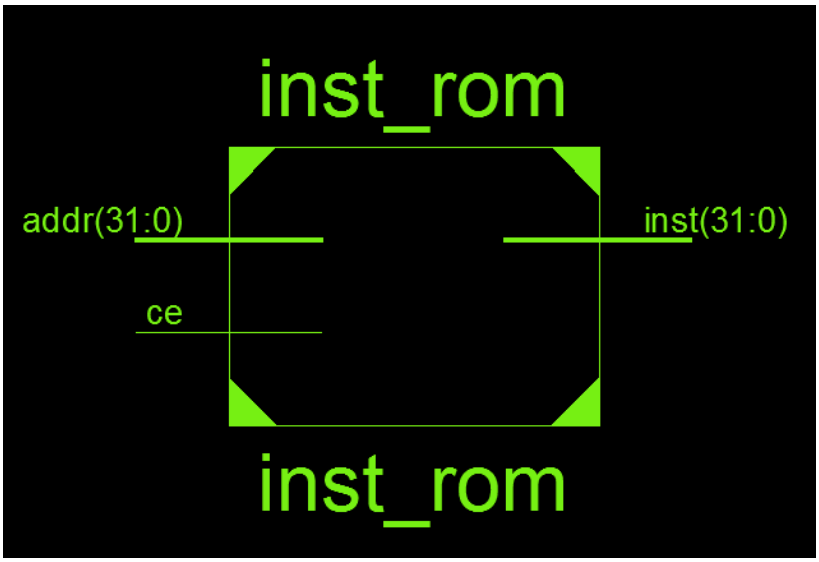
在测试过程中遇到了一个问题，即 pc 的初值。由于开始的程序设计是没有

引入 rst 信号，所以 pc 的初值不定，也就使得无法读取第一条指令，整个测试程序无法运行，为了让程序可以正常运行，将 rst 引入，即初始化。在 testbench 文件中手动调节测试时间和改变 rst 的值，对 pc 的值进行初始化，最终使得 pc 可以正常工作。

除了使用 rst 进行初始化，也可以用 initial 语句对 pc 的值进行初始化。

2. instruction memory

模块如图：



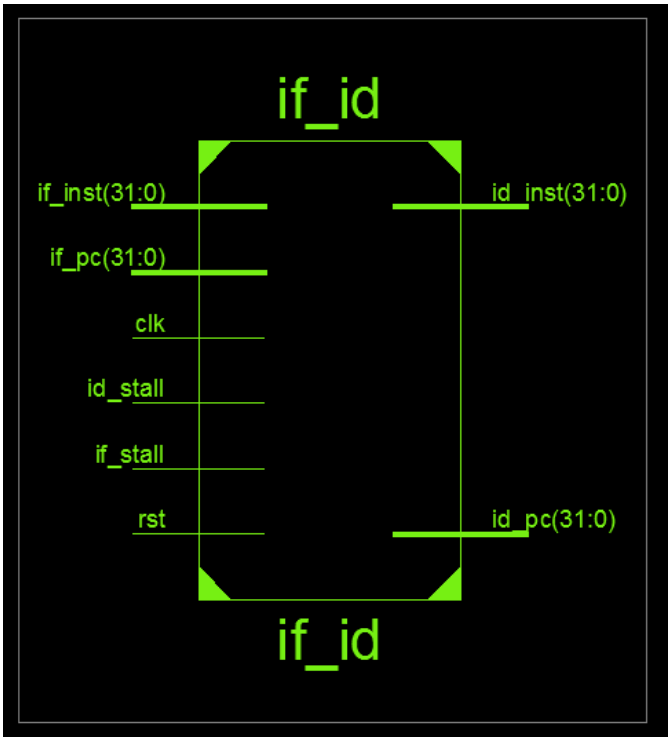
即指令存储器，由$readmemh 函数将预先写在文件的指令读取到存储器中代码如下：

```
);

reg[`InstBus] inst_mem[0:`InstMemNum-1]; // 定义的寄存器，用
initial $readmemh("testmemory", inst_mem); //使用文件存放指
always@(*)begin
 if(ce == `ChipDisable) begin
  inst <= `ZeroWord;
 end else begin
```

输入式 pc，即指令的地址，以及使能信号 ce，由于按照字节寻址，需要将输入的 pc 的值右移 2 位才能找到正确的目标指令。而后将得到的指令通过 inst 端口输出到 if_id 流水寄存器中。

II. IF_ID

模块如图：



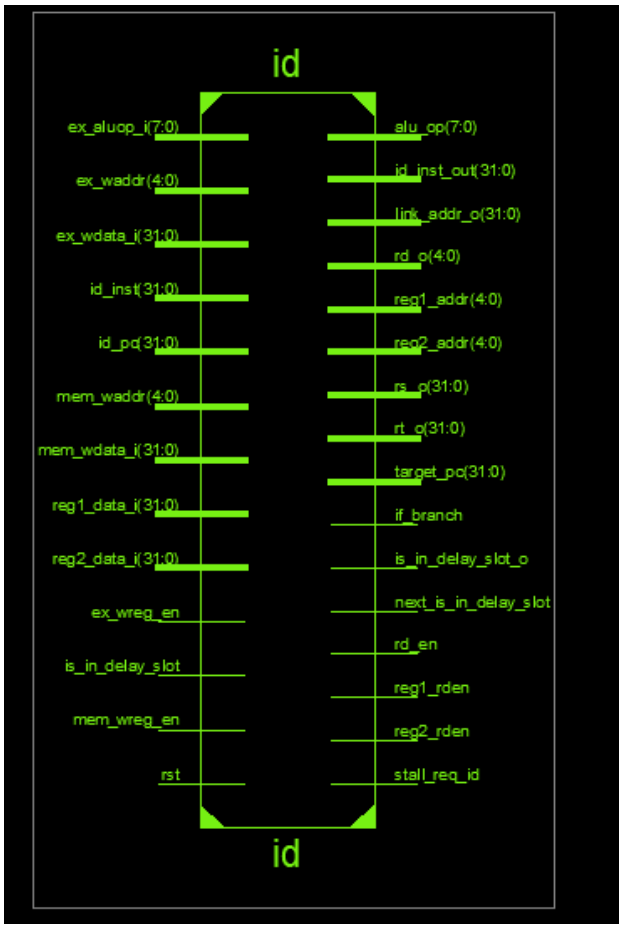
if_id 为 if 段和 id 段之间的流水线寄存器，用于接受来自 pc 段传来的当前地址，以及从指令存储器输出的指令，作为媒介将需要的 pc、指令在 clk 的上升沿送往 id 段。为了实现流水线暂停的机制，还引入了两个 stall 信号即 id、if 的 stall 信号，如果 if 需要暂停但是 id 不需要，则为了让流水线暂停将 if 的输出全部赋为 0，即让流水线空运行，若 if 不需要暂停，则将指令和地址正常输出到 id 段。

传输数据的关键代码如下：

```
;
 always@(posedge clk)begin
  if(rst == `RstEnable) begin
    id_pc <= `ZeroWord;
    id_inst <= `ZeroWord;
  end else if((if_stall == 1'b1) && (id_stall == 1'b0)) begin
    id_pc <= `ZeroWord;
    id_inst <= `ZeroWord; //如果需要暂停，且id段可以接着进行则将本次的
  end else if (if_stall == 1'b0) begin
    id_pc <= if_pc;
    id_inst <= if_inst;
  end
 end
```

III. ID

模块如图：



本次设计的最为核心的模块就是 id 段。因为在这一段需要进行指令的译码、取数，如果是转移指令还需要在 id 段进行分支判断以及修改 pc 的值。

基本的译码操作：

输入端口 id_inst、rst，即从流水线寄存器传来的指令和初始化信号。根据传入的指令，将指令进行分割，基本分为[31:26] [25:21] [20:16] [15:11] [10:6] [5:0]。其中根据 31-26 位的指令值进行判断该指令属于哪一种即 op，而 25-21、20-16、15-11 则分别代表着 rs、rt、rd 的地址。5-0 表示 funct 即对 op 一样的指令进行进一步的划分。而对于一些立即数的指令，则 15-0 作为立即数，并根据所得到的指令类型对其进行零位或符号位扩展至 32 位操作数。

根据指令的类型确定是否需要对寄存器进行读写，reg1_data、reg2_data 表

示从寄存器中读出的值，reg1_rden、reg2_rden 分别表示是否需要对 rs、rt 寄存器进行读操作。Reg1_addr、reg2_addr 表示要读的寄存器的地址。输出信号 rs_o、rt_o 表示两个操作数，送往 id_ex 流水线寄存器，供 ex 段进行指令的执行操作。

数据相关：

在前文已经提到由于某些指令，可能会产生数据相关，所以在这里使用 Data Forwarding 技术来消除数据相关。将上一条指令的 ex 段的写地址、写数据、写使能以及上上一条指令的 mem 段的写地址、写数据、写使能作为输入输入到 id 段。根据指令译码后判断，要读的寄存器的地址是否和输入的写地址相同且是否是需要进行读写，如果满足条件，则把输入的写数据直接赋给输出端口。这样就实现了在数据 available 时将数据送达。消除了数据相关。

Data Forwarding 相关代码：

```
        stall_reg1_load <= 1'b1;
end else if((reg1_rden == 1'b1) && (ex_wreg_en == 1'b1) //解决相邻指令的
            && ex_waddr == reg1_addr)begin
        rs_o <= ex_wdata_i;
end else if((reg1_rden == 1'b1) && (mem_wreg_en == 1'b1) //解决相隔一条指
            && mem_waddr == reg1_addr) begin
        rs_o <= mem_wdata_i;
end else if(reg1_rden == 1'b0)begin              //如果不需要访问寄存器则用立
        rs_o <= imm;
end else if(reg1_rden == 1'b1)begin
        rs_o <= reg1_data_i;
end else begin
        rs_o <= `ZeroWord;
```

Load 型数据相关：

由于 load 型数据相关无法通过 Data Forwarding 技术消除，所以需要引入 stall 使流水线暂停，从而使得指令正确执行。将 ex_aluop 作为输入获取上一条指令的操作符，若是 load 则发出 stall 信号，将其通过端口 stall_req_id 送往 stall control 部件，进行 stall 的控制。再根据之前在流水线寄存器设定的 stall 机制将流水线暂停，暂停一个周期后再恢复流水线，进行正常运转。

Load 型数据相关的代码：

```
always@(*)begin
 stall_reg1_load <= 1'b0;
 if(rst == `RstEnable) begin
      rs_o <= `ZeroWord;
 end else if(pre_inst_if_load == 1'b1 && ex_waddr == reg1_addr && reg1_rden
      stall_reg1_load <= 1'b1;
```

其中 stall_reg1_load 表示操作数需要从 load 型指令执行完后的寄存器中读取值。

控制相关：

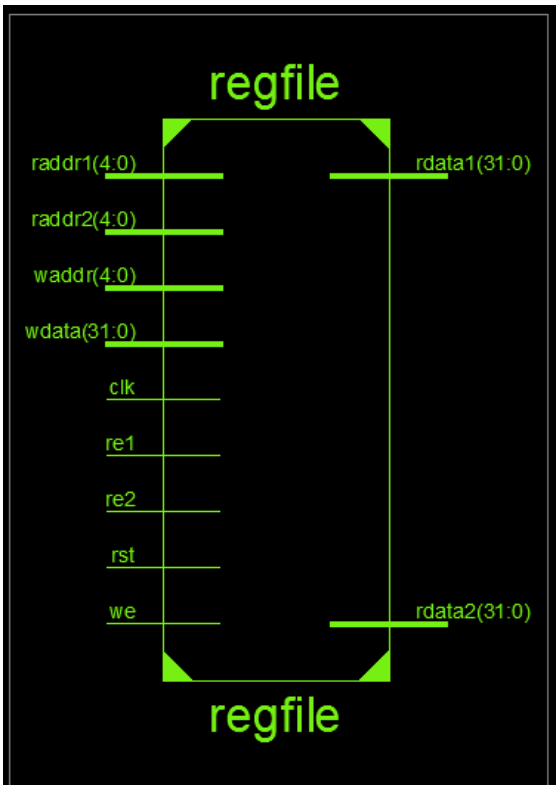
由于转移指令的存在使得下一条指令的 pc 的至无法确定，从而会影响整个流水线的效率。因此，采用的解决方法是在 id 段就将转移指令执行完毕。根据指令对取出的两个操作数进行运算，判断是否符合条件再将正确的 pc 的值传回 pc 段。同时发出 if_branch 的信号表明当前指令是分支指令，而使得 pc 段进入相应的程序分支，得到正确的 pc 的值。接下来的操作则是正常运行。

控制相关部分代码：

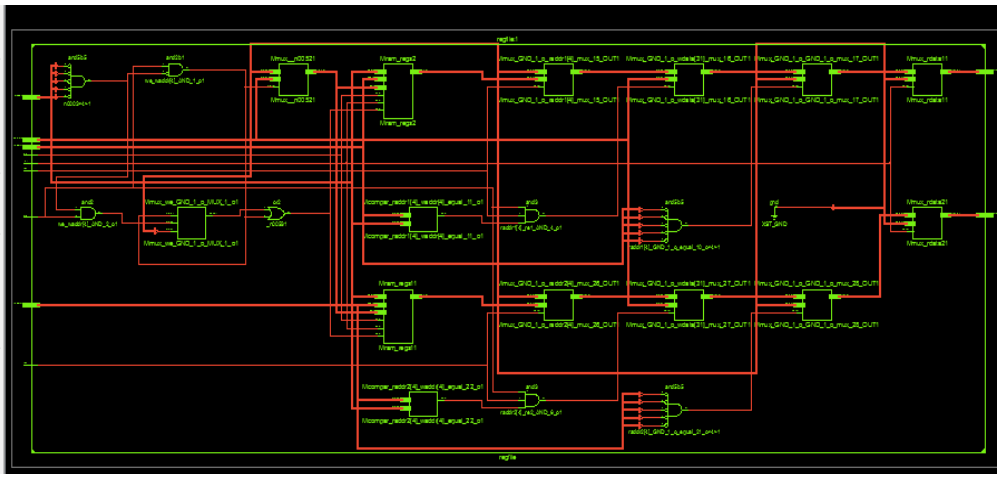
```
`EXE_BNE: begin        //分支指令，如果两个寄存器的数不一样，则跳转
   rd_en <= `WriteDisable;
   alu_op <= `EXE_BNE_OP;
   reg1_rden <= 1'b1;
   reg2_rden <= 1'b1;
   instvalid <= `InstValid;
   if(rs_o != rt_o) begin
      target_pc <= pc_4 + offset_extended;
      if_branch <= 1'b1;
```

*Register

模块如图：



内部结构：

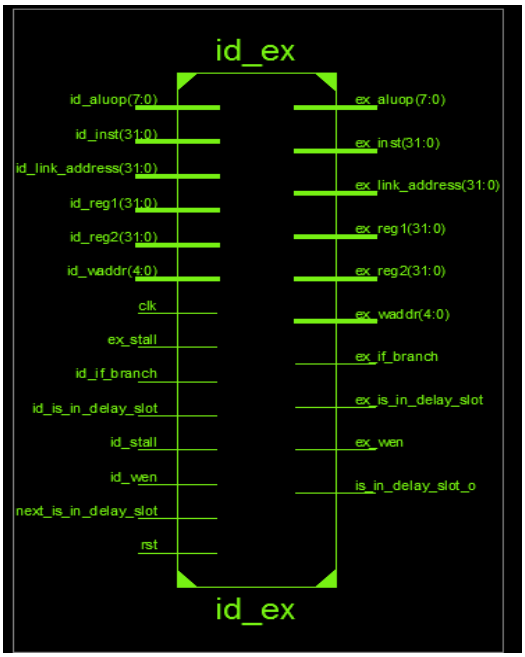


在 register 模块定义了 32 个 32 位的通用寄存器，用于暂存数据。有输入端口 raddr、waddr、re1、re2、wdata 分别表示读的地址、写的地址、两个读使能、写入的数据。在写数据时由于是 wb 段写入，需要在 clk 的上升沿写入，而读数据时为了消除相隔两条指令的数据相关，立即赋值，从而实现寄存器的设计。由于采用字节寻址，同样需要将输入的地址右移 2 位得到正确的寄存器地址。reg 相关代码如下：

```
always@(*)begin
 if (rst == `RstEnable) begin
   rdata1 <= `ZeroWord;
 end else if (raddr1 == 5'b0) begin
   rdata1 <= `ZeroWord;
end else if((raddr1 == waddr) && (we == `WriteEnable)
            && (re1 == `ReadEnable))begin  //这里解决了相隔两条指令的数据相
   rdata1 <= wdata;
end else if(re1 == `ReadEnable)begin
   rdata1 <= regs[raddr1];
end else begin
   rdata1 <= `ZeroWord;
```

IV. ID_EX

模块如图



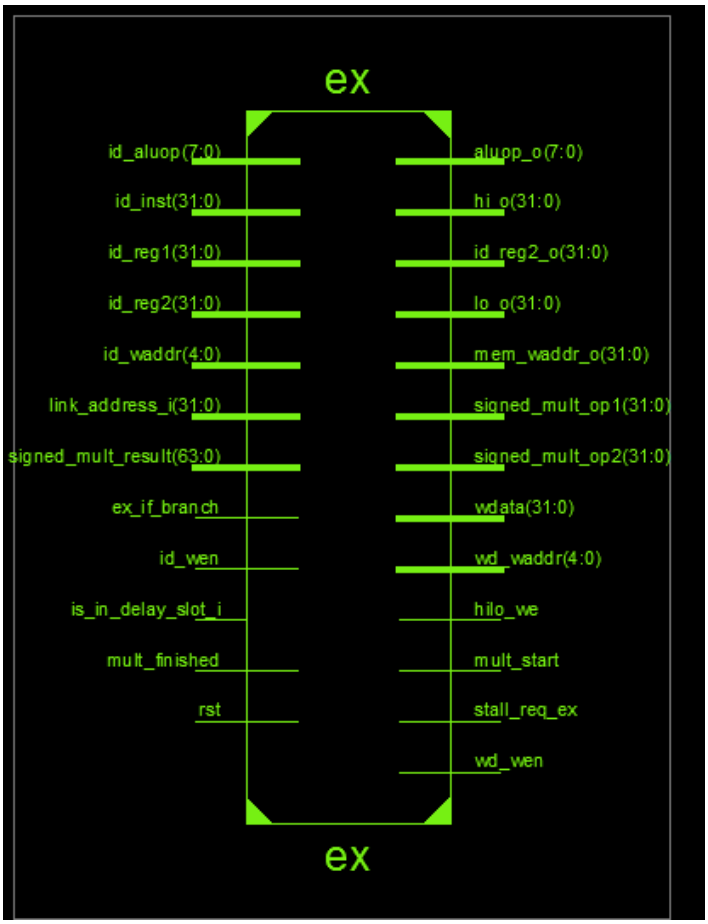
ID_EX 是 id 段和 ex 段的流水线寄存器，用于对指令译码得到的操作数、操作符以及读写使能。传输到 ex 段进行指令的执行。

为了符合流水线的暂停机制，增加输入 ex_stall 和 id_stall，根据 stall 请求情况对 id_ex 段进行暂停。实现功能和 if_id 一样，也是传输空操作数和空指令（类似于 rst=1 时进行的初始化操作）。部分代码如下:

```
    ex_aluop <= `EXE_NOP_OP;
    ex_reg1 <= `ZeroWord;
    ex_reg2 <= `ZeroWord;
    ex_waddr <= `NOPRegAddr;
    ex_wen <= `WriteDisable;        //当id段需要暂停而ex段不需要时，则将
    ex_link_address <= `ZeroWord;
    ex_is_in_delay_slot <= 1'b0;
    ex_if_branch <= 1'b0;
  end else if(id_stall == 1'b0) begin     //如果译码段没有被暂停
    ex_aluop <= id_aluop;
    ex_reg1 <= id_reg1;
    ex_reg2 <= id_reg2;
    ex_waddr <= id_waddr;
    ex_wen <= id_wen;
    ex_link_address <= id_link_address;
```

V. EX

模块如图：



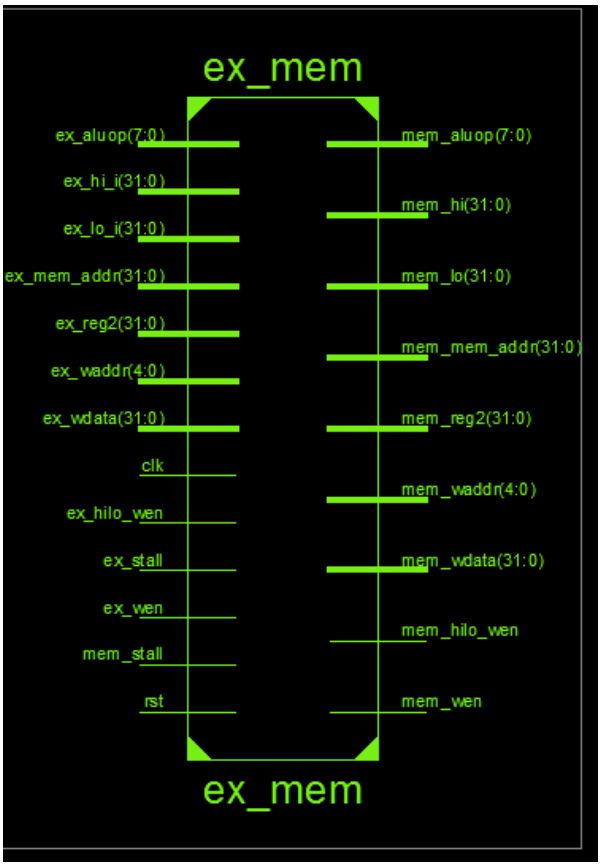
ex 段即是执行阶段。在本次设计中我并没有单独写 alu 模块，而是将其整合进 ex 段，这就使得 ex 的代码量非常大，调试过程比较麻烦，所以在以后的设计中可以考虑进行拆分。

输入端口主要是从 id 段传来的操作数、操作符以及读写使能信号和读写地址。根据操作符进行相应的运算。用中间变量 logicout 存储结果，然后根据写地址以及写使能决定将数据是否输出及其输出的地址。对于一些基本的逻辑、算数运算按照 verilog 自带的运算符即可完成计算。而对于比较指令、有符号的加减乘以及其输出将需要特别的处理，这些将在后面的关键设计原理模块中展开叙述。

同时根据是否是有符号乘法需要设定 ex_stall_req 的信号，因为采用的是 booth 算法，需要多周期执行因此需要将流水线暂停。

VI. EX_MEM

模块如图：



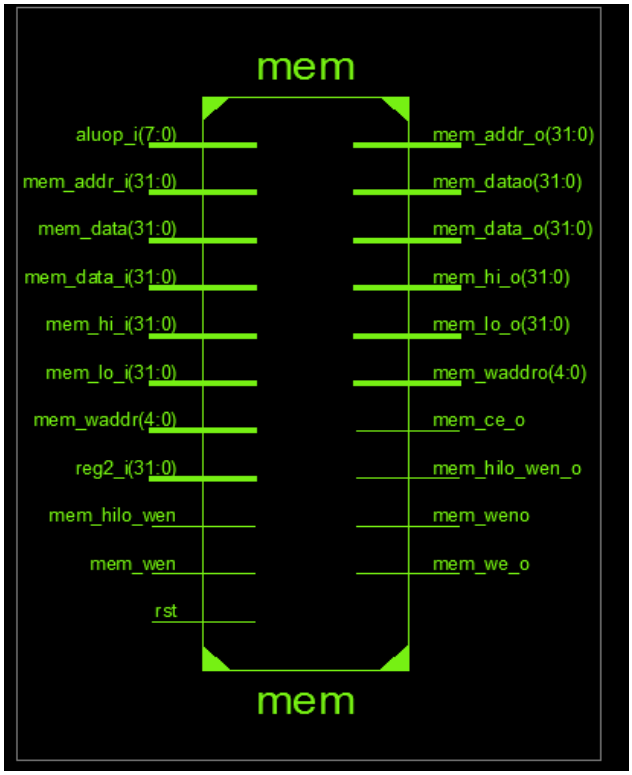
ex_mem 流水线寄存器的功能很简单就是根据 clk 的上升沿将 ex 产生的结果以及传递来的写地址、写使能信号送往 mem 段。要注意，为了方便实现 load、store 指令，需要将 ex 段的 alu 操作符传入 mem 段，进行判断是否要写到 memory 中或者是从 memory 读取数据。乘法的寄存器也是通过 ex_mem 流水线寄存器传输并存放数据。若 stall 的信号为 1 则需要将输出数据置空。

相关代码如下：

```
end else if((ex_stall == 1'b1) && (mem_stall == 1'b0)) begin
  mem_waddr <= `NOPRegAddr;
  mem_wen <= `WriteDisable;
  mem_wdata <= `ZeroWord;          //如果ex段需要暂停而mem段不需要，则将空值输
end else if(ex_hilo_wen == 1'b1) begin
  mem_waddr <= `NOPRegAddr;
  mem_wen <= `WriteDisable;
  mem_wdata <= `ZeroWord;
  mem_hilo_wen <= `WriteEnable;
  mem_hi <= ex_hi_i;
  mem_lo <= ex_lo_i;
end else if(ex_stall == 1'b0) begin
  mem_waddr <= ex_waddr;
```

VII. MEM

模块如图：



mem 段即进行访存的阶段，将 ex 段产生的结果写入数据存储器中，根据 ex 段传来的写地址和写使能决定写数据的位置。输入端口 aluop_i 用以表明是否是 sw 指令，即是否需要对数据存储器进行访问。

为了适应 32 位的乘法，mem 还增加了输入 mem_hi_i、mem_lo_i 表示 32 为乘法结果的高位和低位结果分别送到 hilo 寄存器中。

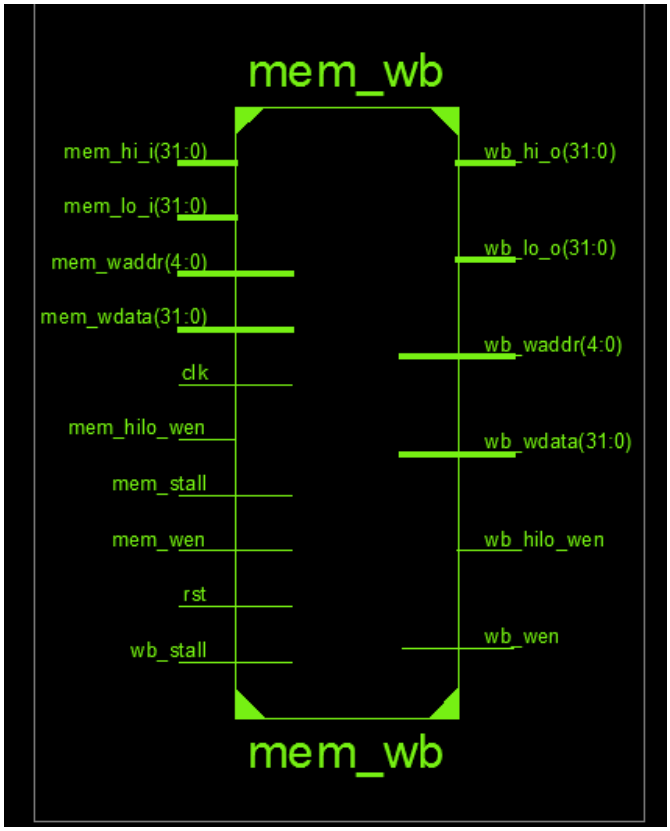
除了对于数据 memory 的访问外，mem 还要将数据传到 wb 阶段，进行寄存器值得修改。通过输出端口将数据和写地址、写使能信号传入 mem_wb 流水线寄存器中。

相关代码如下：

```
end else begin
  mem_waddro <= mem_waddr;
  mem_weno <= mem_wen;
  mem_datao <= mem_data;
  mem_hilo_wen_o <= mem_hilo_wen;
  mem_hi_o <= mem_hi_i;
  mem_lo_o <= mem_lo_i;
  mem_we_o <= `WriteDisable;
  mem_addr_o <= `ZeroWord;
  mem_ce_o <= `ChipDisable;
  case(aluop_i)
      `EXE_LW_OP: begin
         mem_addr_o <= mem_addr_i;
         mem_we_o <= `WriteDisable;
         mem_datao <= mem_data_i;
         mem_ce_o <= `ChipEnable;
      end
      `EXE_SW_OP: begin
         mem_addr_o <= mem_addr_i;
         mem_we_o <= `WriteEnable;
         mem_data_o <= reg2_i;
         mem_ce_o <= `ChipEnable;
      end
```

VIII. MEM_WB

模块如图：



mem_wb 即 mem 段和 wb 段之间的流水线寄存器。由于 wb 段主要的功能是将结果写入目的寄存器对寄存器的值进行更新。所以在本次设计中，我没有单独写出 wb 段，而是之间通过 mem_wb 流水线寄存器将值送出。

同样，为了实现 5 段流水线，在 clk 的上升沿将输入数据送出到目的寄存器，做到一个周期一次访存。

为了契合流水线暂停机制，引入的 mem_stall、wb_stall 信号则控制着该流水线寄存器的值的输出。

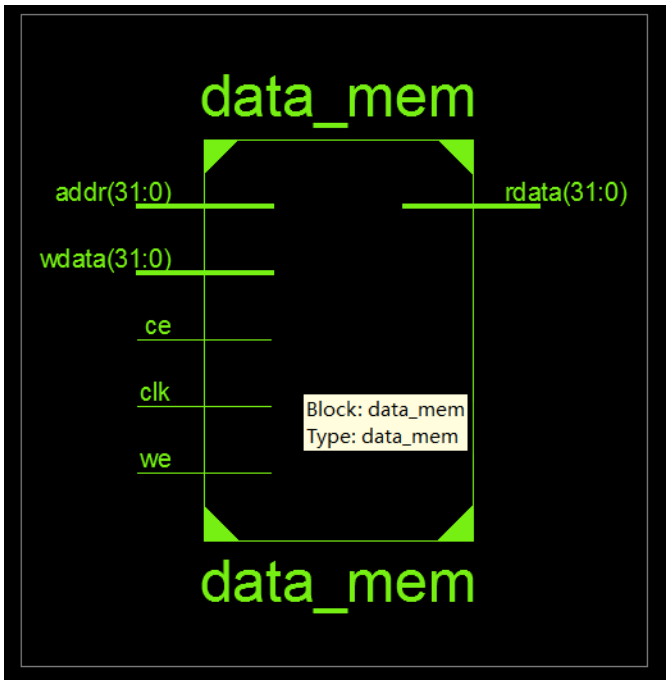
部分代码如下：

```
always@(posedge clk) begin
 if(rst == `RstEnable) begin
   wb_waddr <= `NOPRegAddr;
   wb_wen <= `WriteDisable;
   wb_wdata <= `ZeroWord;
 end else if((mem_stall == 1'b1) && (wb_stall == 1'b0)) begin
   wb_waddr <= `NOPRegAddr;
   wb_wen <= `WriteDisable;
   wb_wdata <= `ZeroWord;    //如果mem段需要暂停而wb段不需要则将空值写给wb
 end else if(mem_stall == 1'b0) begin
   wb_waddr <= mem_waddr;
   wb_wen <= mem_wen;
   wb_wdata <= mem_wdata;
 end
end
```

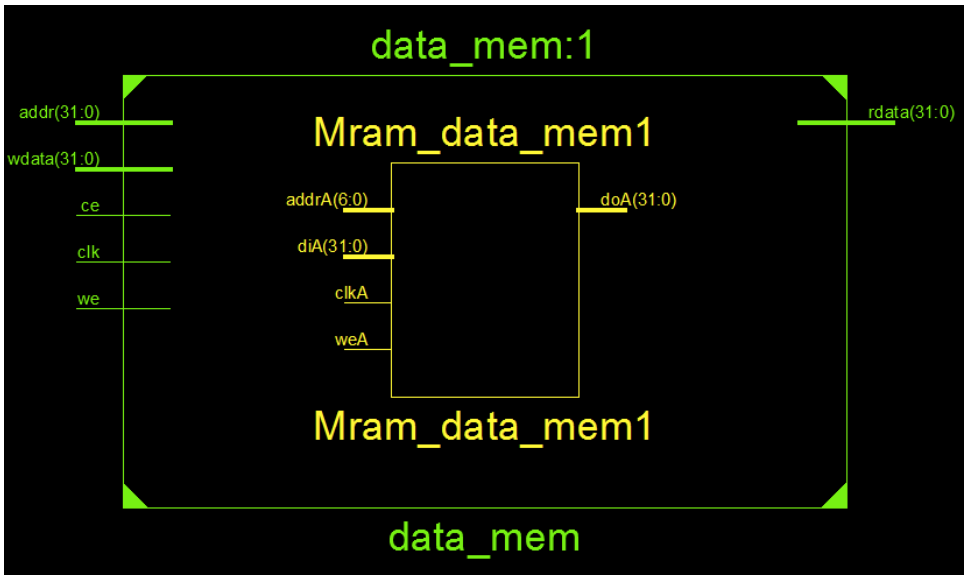
IX. Memory

Data Memory

模块如图：



内部结构：



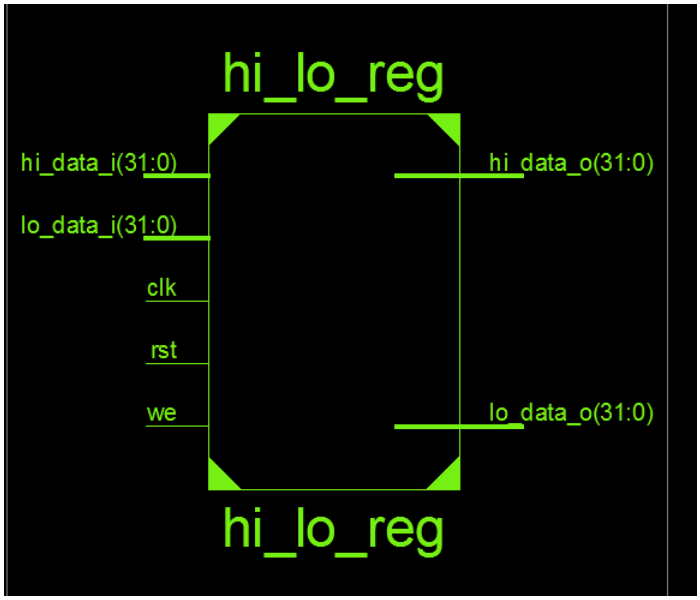
data_mem 即数据存储器，用以存放指令所需的一些数据。在本次设计中，涉及到数据存储器的指令不多，所以将 data memory 的大小定义为 32 个 32 位的。Data memory 的逻辑相对简单。在 clk 的上升沿读取数据，而在任意时刻可以将数据写入。只是在访存的时候需要注意是按字节寻找，所以需要将输入的地址数据右移 2 位。

部分代码如下：

```
always@(posedge clk) begin
    if(ce == `ChipDisable) begin
    end else if(we == `WriteEnable) begin
        data_mem[addr >> 2] = wdata;
    end
end

//读操作
always@(*) begin
    if(ce == `ChipDisable) begin
        rdata <= `ZeroWord;
    end else if(we == `WriteDisable) begin
        rdata <= data_mem[addr >> 2];
    end else begin
        rdata <= `ZeroWord;
    end
end
```

Hi_Lo Register

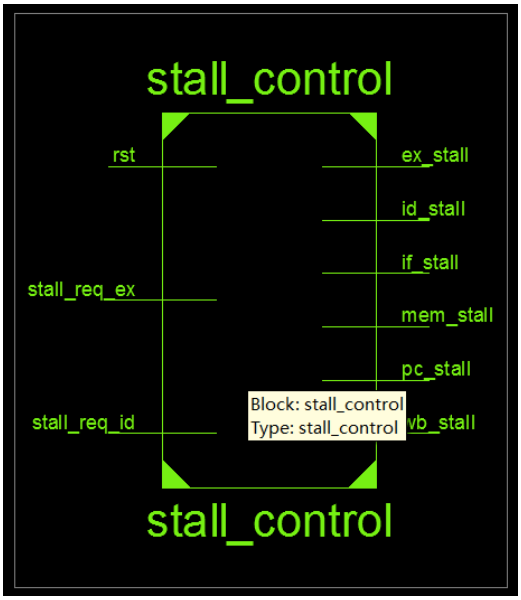
模块如图：



hilo reg 也是设计得非常简单，只是进行数据的写入和读出（由于本次设计没有实现需要用该寄存器来读取的指令，所以输出端口未实例化）。Hi 表示 64 位结果的高 32 位，lo 则是其低 32 位。同样是在使能信号有效且为 clk 的上升沿是将数据写入。

X. Stall Control Unity

模块如图：



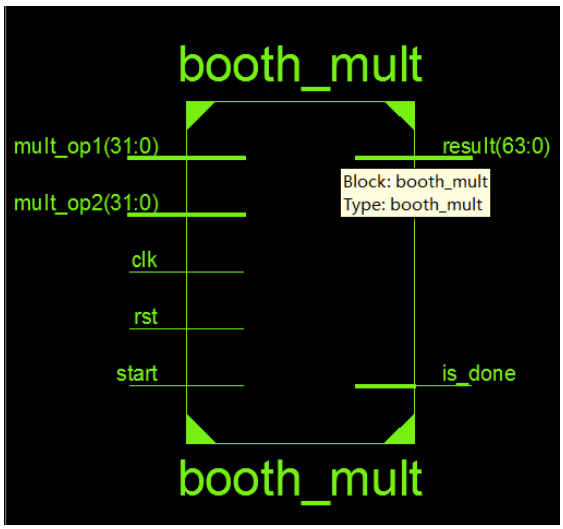
该部件是为了实现流水暂停而添加的。用于发出各段的 stall 信号。有两个输入端口 stall_req_id 以及 stall_req_ex 表示 id 和 ex 是否发来 stall 请求信号。根据输入的 stall 请求信号对应发出 stall 信号。如果是 id 段发来的则将 pc、if、id 的 stall 信号置为 1 表示需要暂停，而其他的则是表示可以正常执行。若是 ex 发来的则将 pc、if、id、ex 的置为 1 其他可以正常执行。

部分代码如下：

```
always@(*)begin
 if(rst == `RstEnable) begin
   pc_stall <= 1'b0;
   if_stall <= 1'b0;
   id_stall <= 1'b0;
   ex_stall <= 1'b0;
   mem_stall <= 1'b0;
   wb_stall <= 1'b0;
 end else if(stall_req_id == 1'b1) begin
   pc_stall <= 1'b1;
   if_stall <= 1'b1;
   id_stall <= 1'b1;
   ex_stall <= 1'b0;
   mem_stall <= 1'b0;
   wb_stall <= 1'b0;
 end else if(stall_req_ex == 1'b1) begin
   pc_stall <= 1'b1;
   if_stall <= 1'b1;
   id_stall <= 1'b1;
   ex_stall <= 1'b1;
   mem_stall <= 1'b0;
   wb_stall <= 1'b0;
 end else begin
   pc_stall <= 1'b0;
   if_stall <= 1'b0;
   id_stall <= 1'b0;
```

XI. Booth Multiplier

模块如图：



boot_mult 用于实现多周期有符号数乘法的部件。关键的三个输入端口 mult_op1、mult_op2、start 分别表示乘法的两个操作数已经 booth 算法开始信号。而后将 64 位的结果输出，并且输出 is_done 信号表示已计算完毕恢复流水线的正常运行。具体的 booth 算法实现在下面的关键设计原理展开叙述。

七、关键设计原理

这里主要讲的是两个关键的算法问题，即实现有符号加减和 booth 算法。
还有就是将上述的所有模块集合起来，形成最终的 cpu。

● 有符号加减

有符号的加减法的难点在于需要对操作数进行转换，以及判断结果的溢出。
有符号的加法规则：$[x]_{补} + [y]_{补} = [x + y]_{补}$
有符号的减法： $[x]_{补} + [-y]_{补} = [x - y]_{补}$

由于在我的程序中，数据的输入都是采用原码的形式，所以在这里需要将操作数转换成补码。实现方法如下：

```
assign id_reg1_comp = (id_reg1[31] == 1'b1) ?
                      {id_reg1[31], ~id_reg1[30:0]} + 1 : id_reg1;    //操作数1的补码
assign id_reg2_comp = (id_reg2[31] == 1'b1) ?
                      {id_reg2[31], ~id_reg2[30:0]} + 1 : id_reg2;    //操作数2的补码
assign id_reg2_not = {~id_reg2[31], id_reg2[30:0]}; //操作数2的相反数
assign id_reg2_not_comp = (id_reg2_not[31] == 1'b1) ?
                          {id_reg2_not[31], ~id_reg2_not[30:0]} + 1 : id_reg2_not;  //操作数
```

取完补码后就直接进行加减的计算得到中间结果。但由于该结果也是由补码表示的所以还需要将其取补作为最终的结果赋给 logicout 输出。该部分代码如下：

```
assign signed_sum_result = (temp[31] == 1'b1) ?
                           {temp[31], ~temp[30:0]} + 1 : temp;
```

最后是对于有符号加法的溢出判断。通俗来讲，溢出就是当两个操作数同号但是结果是与操作数异号的。因此溢出的判断比较简单，只需要对数据的最高位即符号位判断即可。具体代码如下：

```
assign overflow = ((!id_reg1[31] && !id_reg2[31]) && signed_sum_result[31]) ||
                  ((id_reg1[31] && id_reg2[31]) && !signed_sum_result[31]) ||
                  ((id_reg1[31] && !id_reg2[31]) && !signed_sub_result[31]) ||
                  ((!id_reg1[31] && id_reg2[31]) && signed_sub_result[31]);
```

● booth 算法

booth 算法用于实现有符号乘法。具体算法为：

Booth 算法就是对乘数从低位开始判断，根据两个数据位的情况决定进行加法、减法还是仅仅移位操作。判断的两个数据位位当前位及其右边的位（初始

化需要增加一个辅助位 0），移位操作是向右移动。当二进制数第一次遇到 1 时，可以用减法取代一串的 1，而当遇到最后一个 1 后面的 0 时，再加上被乘数。

假设 X、Y 都是用补码形式表示的机器数，[X]补和[Y]补=Ys. Y1Y2…Yn，都是任意符号表示的数。比较法求新的部分积，取决于两个比较位的数位，即 Yi+1Yi 的状态。

布斯乘法规则归纳如下：

首先设置附加位 Yn+1=0，部分积初值[Z0]补=0。

当 n≠0 时，判 YnYn+1，

若 YnYn+1=00 或 11，即相邻位相同时，上次部分积右移一位，直接得部分积。

若 YnYn+1=01，上次部分积加[X]补，然后右移一位得新部分积。

若 YnYn+1=10，上次部分积加[-X]补，然后右移一位得新部分积。

当 n=0 时，判 YnYn+1(对应于 Y0Y1)，运算规则同(1)只是不移位。即在运算的最后一步，乘积不再右移。

注意：

（A）比较法中不管乘数为正为负，符号位都参加运算，克服了校正法的缺点

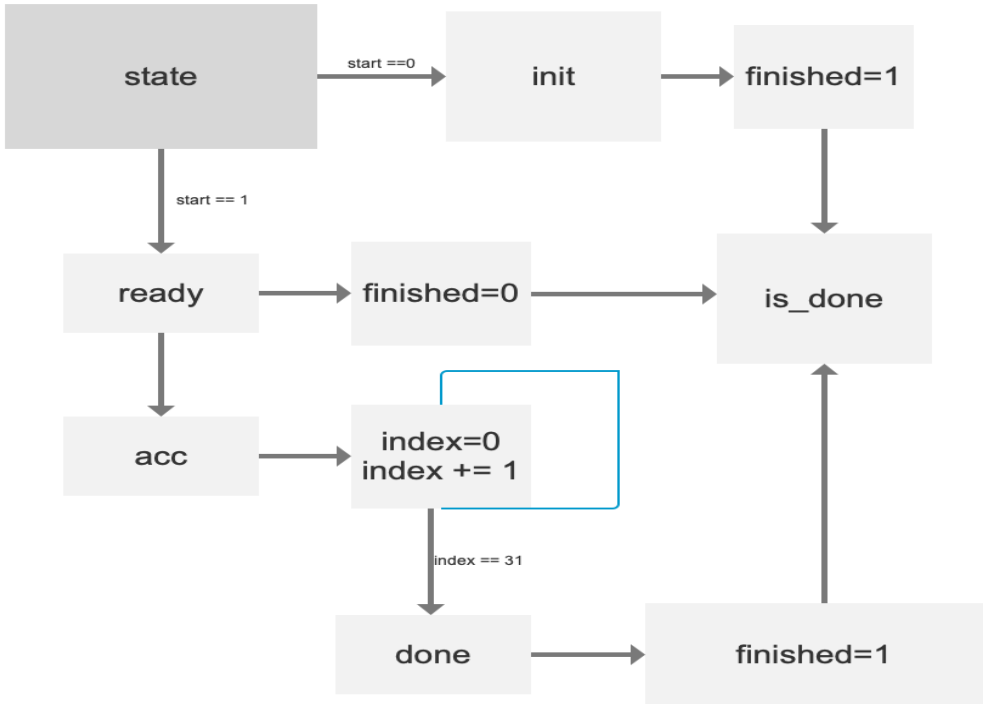
（B）运算过程中采用变形补码运算（双符号位）

（C）算法运算时的关键是 YnYn+1 的状态：后者（Yn+1）减前者（Yn），判断是加减（+/-X）

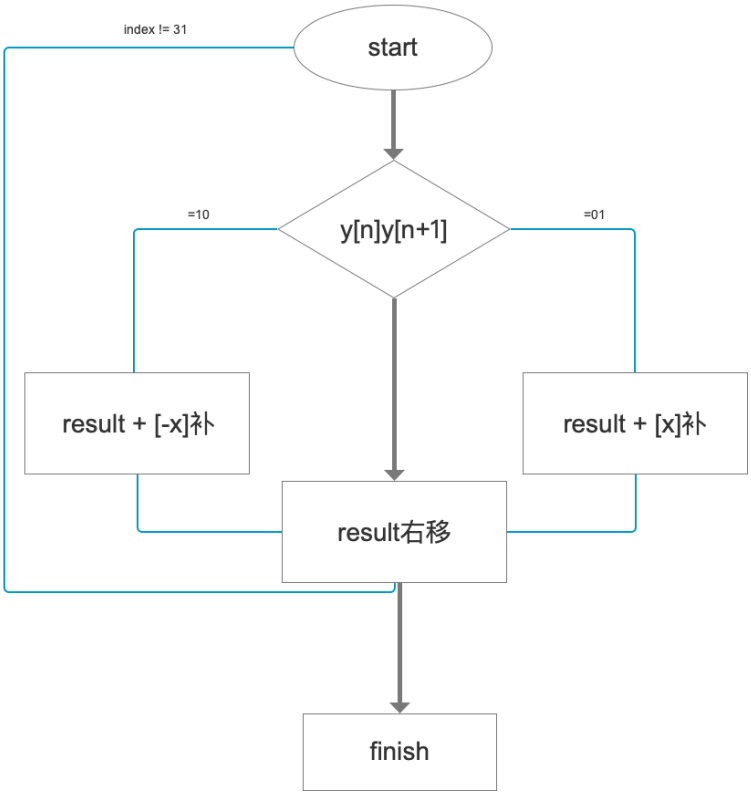
在 CPU 的设计中，为了实现 booth 算法需要将其设计成多周期的乘法器。由于是 32 位的乘法，所以需要 32 个周期。

为此设置 4 个状态即 init——初始化状态，即 booth 乘法器未启动；ready——准备状态，即收到了乘法运算使能信号，准备开始进行乘法的运算；acc——运算状态，即乘法器正式进入计算阶段，根据设定的算法进行加法、移位计算；done——根据设置的计数标志而设定的状态，一旦达到指定的数值，就转为 done 状态表示运算完毕。

状态转换图：


state
start ==0
init
finished=1
start == 1
ready
finished=0
is_done
acc
index=0
index += 1
index == 31
done
finished=1

计算流程图：


index != 31
start
=10
y[n]y[n+1]
=01
result + [-x]补
result + [x]补
result右移
finish

booth 算法采用的是用加法和移位代替乘法的计算。采用的是双符号位计算。开始时会讲 is_done 置为 0 表示未完成，同时凭借这个信号输出 stall 的请求，让流水线暂停。

将乘数的末两位作为标志若为 01 则加上被乘数的补码，后右移一位；若是 10 则加上被乘数的相反数的补码，再右移一位；若是 00 或 11 则直接右移一位。全局有计数值 index_i，进行计数共运行了多少次。当计算到第 31 次表示已经完成计算，将状态置为 done。输出信号 is_done 表示已经完成计算，将暂停的流水线重新恢复运行。

最后将结果返回到 ex 段，根据高低位写入到 hilo 寄存器中。

关键代码如下：

```
always@(posedge clk or negedge start) begin
   if(rst == `RstEnable) begin
      current_state <= init;
      is_done <= 1'b0;
   end else if(start == 1'b0) begin
      current_state <= init;            //初始阶段
   end else begin
      current_state <= next_state;
   end
end

always@(current_state or index_i) begin
   case(current_state)
      init: begin
         next_state = ready;            //准备阶段
      end
      ready: begin
         next_state = acc;              //计算阶段
      end
      acc: begin
         if(index_i == 6'h1f) begin       //表示已经完成31次移位操作
            next_state = done;
         end
      end
    endcase
end
```

```
always@(current_state or index_i) begin
   case(current_state)
      init: begin
         finished = 0;
      end
      ready: begin
         x = (mult_op1[31] == 1'b1) ?
             {mult_op1[31],mult_op1[31], ~mult_op1[30:0]} + 1 : {mult_op1[31], mult_op1[31:0]};
         op1_opposite = {~mult_op1[31], ~mult_op1[31], mult_op1[30:0]};//op1的相反数
         x_comp = (op1_opposite[32] == 1'b1) ?                  //op1的相反数的补码
                  {op1_opposite[32], op1_opposite[32], ~op1_opposite[30:0]} + 1 : op1_opposite;
         y[32:0] = (mult_op2[31] == 1'b1) ?
             {{{mult_op2[31], ~mult_op2[30:0]} + 1}, 1'b0 } : {mult_op2[31:0], 1'b0};
         z = 0;
      end
      acc: begin
         case(y[1:0])
            2'b01: begin
               z = z + x;
               {z[32:0], y[32:0]} = {z[32], z[32:0], y[32:1]}; //结果和剩余乘数左移1位
            end
            2'b10: begin
               z = z + x_comp;
               {z[32:0], y[32:0]} = {z[32], z[32:0], y[32:1]};
            end
            default: begin     //如果是00或11
```

```
                {z[32:0], y[32:0]} = {z[32], z[32:0], y[32:1]}; //直接左移
            end
          endcase
       end
       default: begin
          finished = 1;
       end
    endcase
end
always@(posedge clk)begin
    if(current_state == acc)begin
       index_i <= index_i + 1'b1;
    end else begin
       index_i <= 0;
    end
    is_done <= finished;
end
assign temp = {z[31:0], y[32:1]};
always@(posedge is_done) begin
    result <= (temp[63] == 1'b1) ?
              {temp[63], ~temp[62:0]} + 1 : temp;
end
```

- 模块级联

在将所有的模块都写完后，需要做的就是将模块进行级联。级联的方法比较简单，只需要定义一些中间变量，将输入输出口连接起来即可。再添加全局变量 clk 和 rst 作为核心的信号进行 cpu 的运行。

相关代码如下：

```
if_id if_id0(
          .clk(clk), .if_pc(pc), .rst(rst),
          .if_stall(if_stall), .id_stall(id_stall),
          .if_inst(rom_data_inst), .id_pc(id_pc_inst),
          .id_inst(id_inst_i)
             );

//进行id段的模块化
id id0(
      .id_pc(id_pc_inst), .id_inst(id_inst_i), .rst(rst),

      //来自regfile的输入
      .reg1_data_i(reg1_data), .reg2_data_i(reg2_data),

      //送到regfile模块的信息
      .reg1_rden(reg1_read), .reg2_rden(reg2_read),
      .reg1_addr(reg1_addr), .reg2_addr(reg2_addr),

      //送到id段的信息
      .alu_op(id_aluop_o), .rs_o(id_reg1_o),
      .rt_o(id_reg2_o), .rd_en(id_wreg_en), .rd_o(id_waddr),
```

```
//例化处理器top_cpu
top_cpu top_cpu0(
            .clk(clk), .rom_data_inst(inst), .rst(rst),
            .rom_addr_out(inst_addr), .rom_ce_o(rom_ce)
                );

//例化指令寄存器inst_rom
inst_rom inst_rom0(
            .ce(rom_ce),
            .addr(inst_addr), .inst(inst)
                  );
```

八、测试&仿真

最后就是进行测试程序的编写以及进行仿真验证。由于时间比较仓促，所以未进行上板验证。测试程序主要是验证一些指令的正确性，测试程序如下：

左边是汇编程序，右边是翻译成的 16 进制机器语言程序，用于程序读取

ori $1,$0,#x8001	34010810
ori $2,$0,#x8002	34022180
and $3,$1,$2	00221824
add $3,$1,$2	00221820
or $4,$2,$0	00402025
bne $1,$2,#9(跳到 pc=36)	14220003
nop	00000000
lw $5,$1,#1	8ca10001
sw $6,$2,#1	acc20001
ori $1,$0,#0ff	340100ff
ori $2,$0,#0ff	340200ff
mult $1,$2	00220018

最后形成的仿真图像如下：

验证 ori 指令：

此时$1 和$2 的值已经修改为目标值，指令执行成功。

验证 and 指令

$3 的值为 00000000，结果正确。

验证 add 指令

$3 的值变为 2990，符合有符号的加法运算结果，指令执行正确。

验证 or 指令

$4 的值已变为 2180，符合 or 指令的运算，指令执行正确。

验证 bne 指令

相隔一个周期，pc 的值发生了跳变，指令执行正确。

验证 mult 和 stall

可以看出在执行 mult 指令时，pc 的值已经被暂停下来，表示流水线已经暂停。

表示 booth 乘法器的运算结果为 fe01，和预期结果相同，指令执行正确

九、参考资料

[1] 雷思磊.自己动手写 CPU[M] 电子工业出版社 2014 年 9 月

[2] 夏宇闻.Verilog 经典教程[M]