实验报告成绩:	成绩评定日期:

2021 ~ 2022 学年秋季学期

A3705060050《计算机系统》必修课

课程实验报告

班级：人工智能 1901

组长：李桉雨

组员：李定霖，邱聖邦

报告日期：2021.12.18

目录

1、 工作量分配：

李桉雨 50%：regfile 数据相关问题处理，除法指令，hilo 寄存器及数据相关处理，mf、mt 指令处理，实现 32 周期乘法器，合作完成乘除法器的合并。
李定霖 25%：添加了 jal，ja，subu，addu 指令，乘法运算，sb、sh 指令，负责部分地址转移指令。
邱聖邦 25%：增加 sll,bne,or 三条指令，添加了 lbu、lh、lhu 指令，负责部分运算指令。
李桉雨-20195251、李定霖-20195183、邱聖邦-20195263

2、 总体设计

2.1、 总体连线图

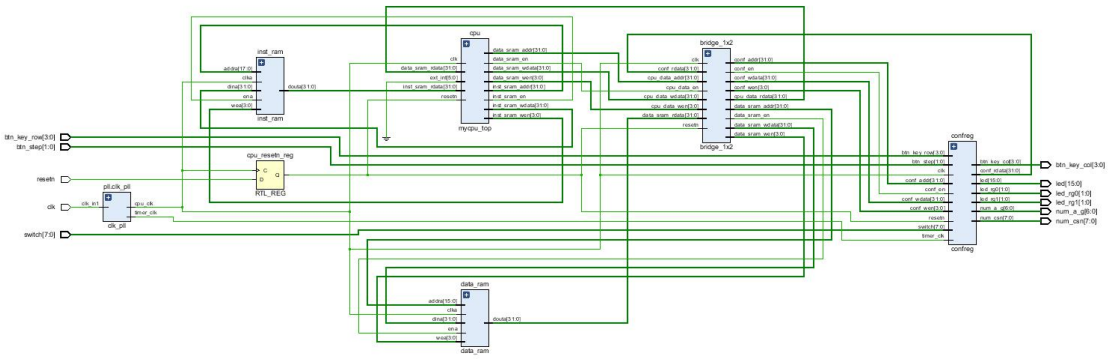


图 2-1

2.2、 各流水段间的连线图

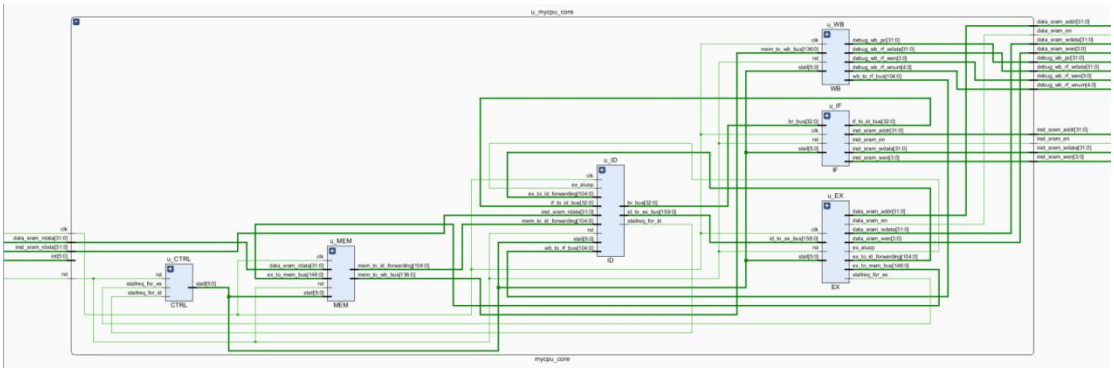


图 2-2

2.2.1、 IF－ID 段流水连线图

图 2-2-1-1：IF－ID 总体设计

图 2-2-1-2：IF 具体设计

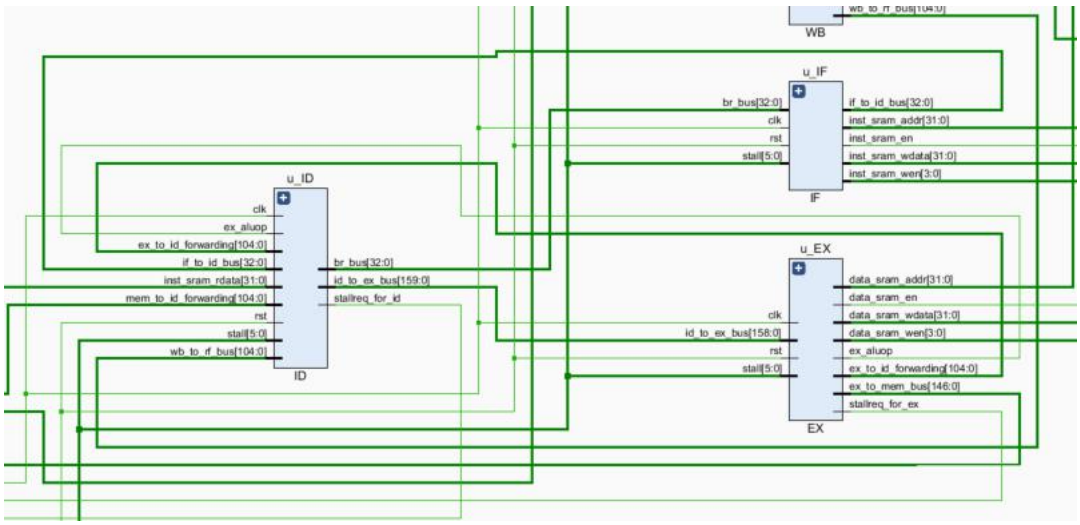


图 2-2-1-1

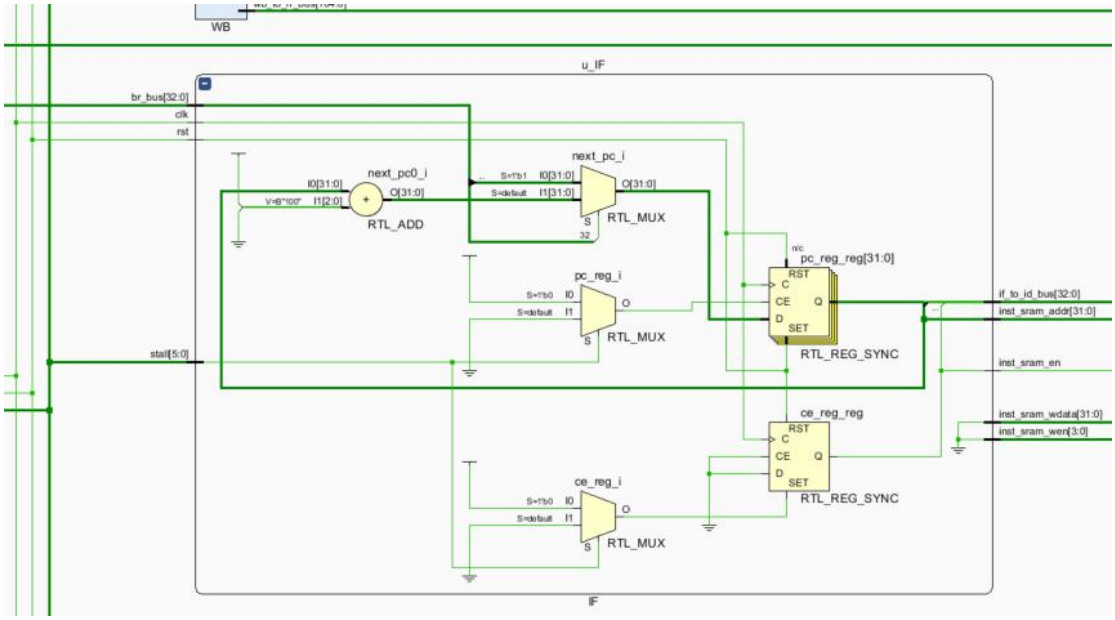


图 2-2-1-2

2.2.2、 ID-EX 段流水连线图

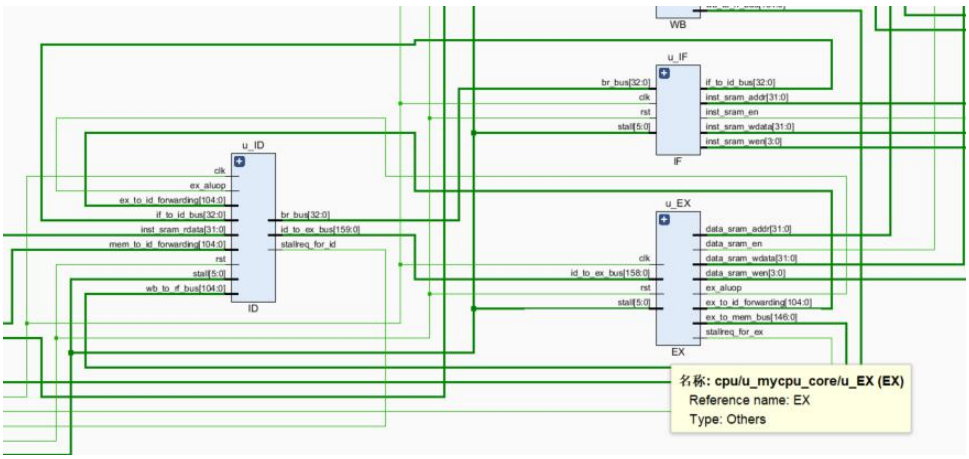


图 2-2-2-1

2.2.3、 EX－MEM 段流水连线图

图 2-2-3-1：EX-MEM 连接示意图

图 2-2-3-2：EX 内部细节

图 2-2-3-1

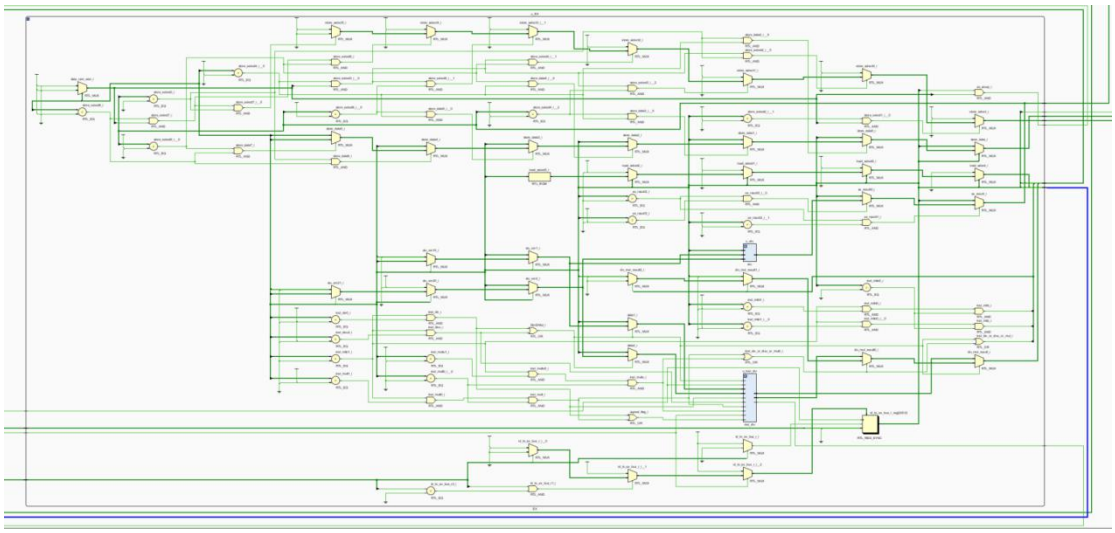


图 2-2-3-2

2.2.4、 MEM－WB 段流水连线图

图 2-2-4-1：MEM-WB 总体设计

图 2-2-4-2：MEM 具体设计

图 2-2-4-3：WB 具体设计

图 2-2-4-4：CTRL 具体设计

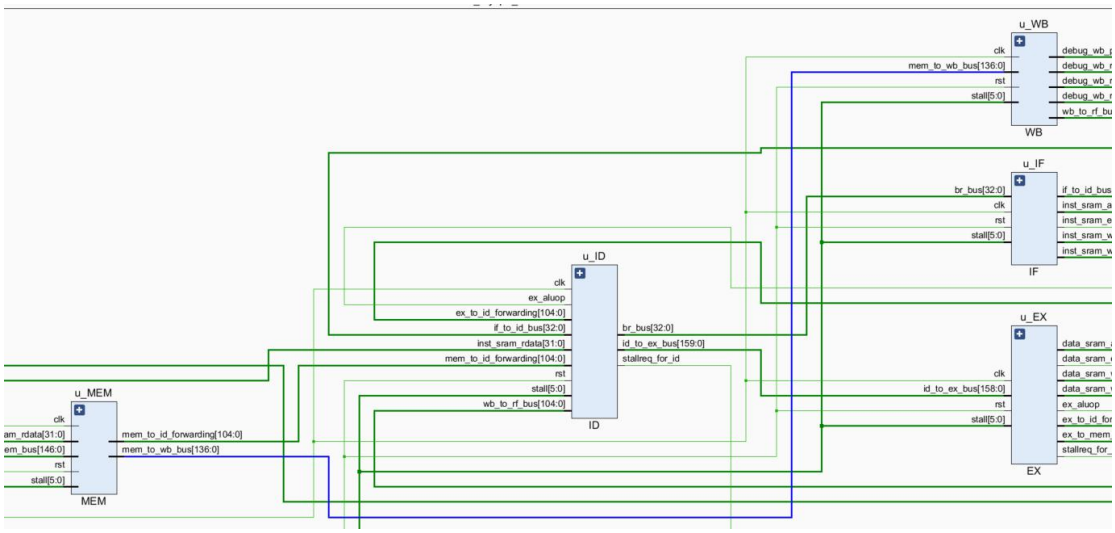


图 2-2-4-1

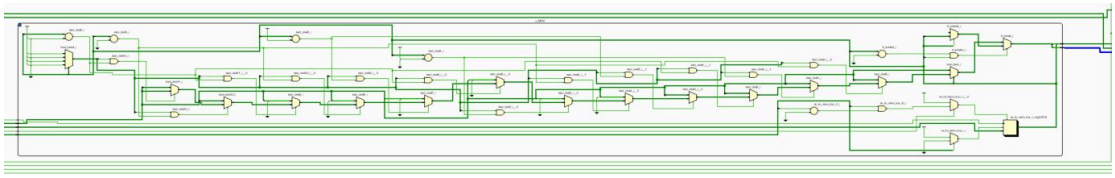


图 2-2-4-2

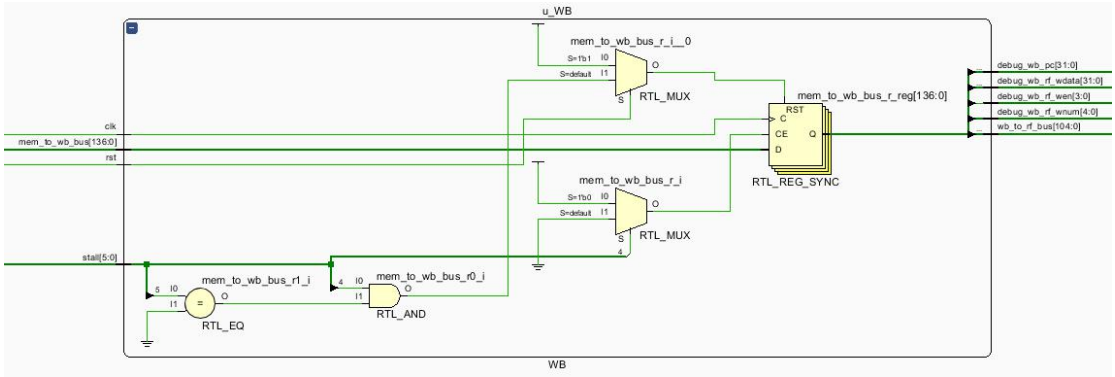


图 2-2-4-3

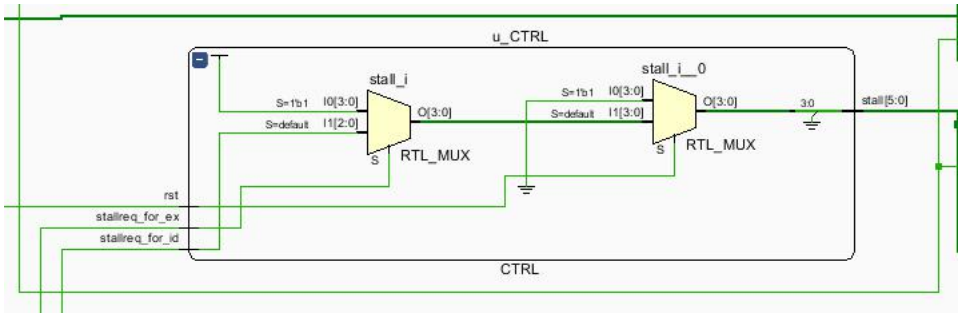


图 2-2-4-4

2.3、 完成的指令数

已完成基本的 64 个点，外加自己实现的 32 周期乘法器、合并乘除法器。

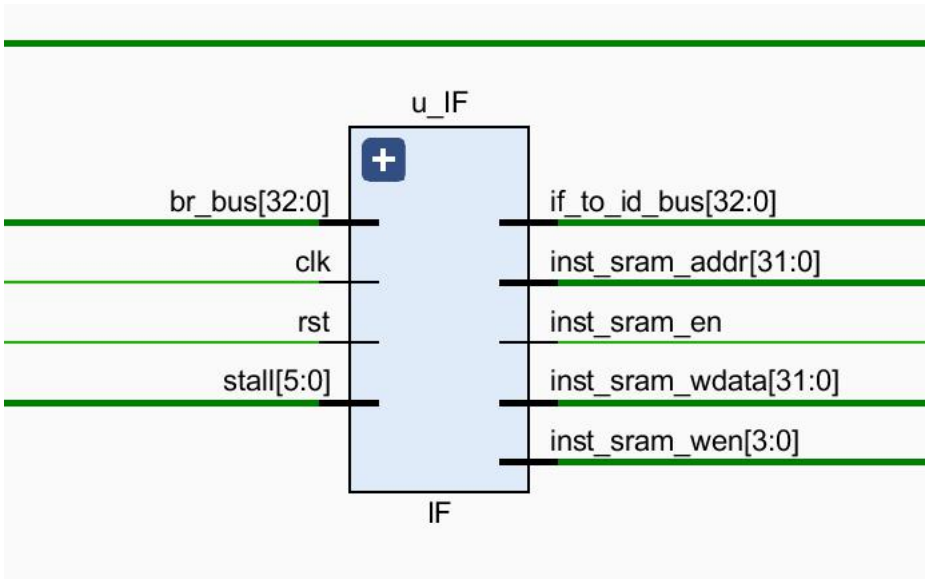
```
----[1462335 ns] Number 8'd62 Functional Test Point PASS!!!
        [1472000 ns] Test is running, debug_wb_pc = 0xbfc44e98
        [1482000 ns] Test is running, debug_wb_pc = 0xbfc45db8
----[1490525 ns] Number 8'd63 Functional Test Point PASS!!!
        [1492000 ns] Test is running, debug_wb_pc = 0xbfc30014
        [1502000 ns] Test is running, debug_wb_pc = 0xbfc30ebc
        [1512000 ns] Test is running, debug_wb_pc = 0xbfc31df4
----[1517065 ns] Number 8'd64 Functional Test Point PASS!!!
--------------------------------------------------------------
```

2.4、 程序运行环境及使用工具

3、 单段流水说明

3.1、 IF 段流水功能说明

3.1.1、功能模块及结构示意图



3.1.2、 整体功能

根据 PC 值从存储器中取出指令，并将指令送入指令寄存器，PC 值加 4 或根据 ID 段传回的地址决定下一条指令的地址。

3.1.3、 端口介绍

1.br_bus:ID 将转移信号以及目的地址发送给 IF 段以用来进行 PC 改变。

2.Stall：从 CTRL 传输的控制信号，用以控制整个流水段的暂停与执行。

3.If_to_id_bus:将 IF 目前所处的 pc 以及使能信号发送给 ID 段，标志着目前各段流水对应的 PC。

4.inst_sram_XXX:访问指令内存端口族，用于从内存中取指，并将通过 inst_sram_rdata 传输给 ID 段。

3.1.4、 信号介绍

```
assign {
    br_e,
    br_addr
} = br_bus;
```

Br_e:转移信号，决定是否进行地址转移
Br_addr:转移地址，在 ID 段计算得出后传回 IF 段。

```
assign next_pc = br_e ? br_addr
               : pc_reg + 32'h4;
```

Next_pc:决定下一个 PC 值是多少，优先是转移地址，否则默认 PC+4.

```
assign inst_sram_en = ce_reg;
assign inst_sram_wen = 4'b0;
assign inst_sram_addr = pc_reg;
assign inst_sram_wdata = 32'b0;
```

Inst 族：en 表示访存许可，wen=4'b0 表示 load，addr 表示指令所在地址。

```
assign if_to_id_bus = {
    ce_reg,
    pc_reg
};
```

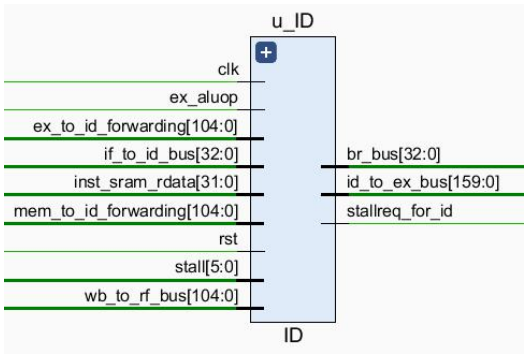
If_to_id_bus:IF 段到 ID 段的常规总线，将当前 PC 传到 ID 段。

3.1.5 总结

IF 段没有做太多改动，功能也与教学内容没太大差别，在这里不做过多赘述。

3.2、 ID 段流水功能说明

3.2.1、 功能模块及结构示意图



3.2.2、 整体功能

对指令进行译码，完成对 ALU 操作数的选择、HILO 寄存器的操作以及转移地址的计算等。

3.2.3、 端口介绍

Input：

1.If_to_id_bus:从 IF 段到 ID 段的总线，与 IF 段功能一样。

2.Inst_sram_rdata:IF 段访存的结果通过该线输入到 ID 段，存有指令。

3.Ex_to_id_forwarding:将 EX 段计算得出的结果传回 ID 段，解决流水中的常规数据相关（regfile、hilo 等）。

4.Mem_to_id_forwarding:将 MEM 段计算得出的结果传回 ID 段，解决流水中的常规数据相关（regfile、hilo 等）以及需要暂停的数据相关。

5.wb_to_rf_bus:进行寄存器的常规写回操作（regfile、hilo）。

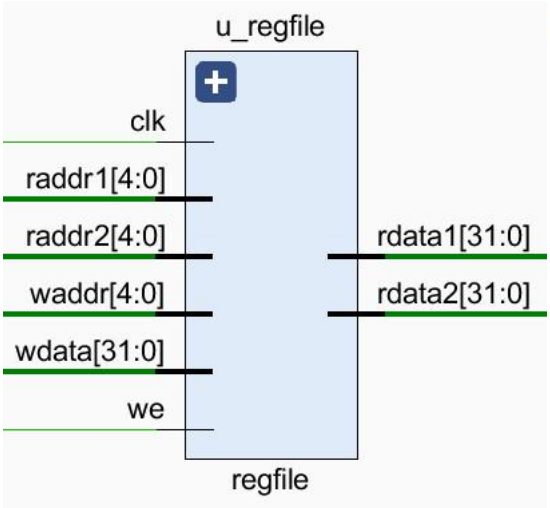
6.ex_aluop:EX 用来提醒 ID 段它的上一条指令是 load 指令，用于进行暂停。

Output：

1. br_bus:将计算得出的转移地址传回 ID 段。
2. Id_to_ex_bus:ID 到 EX 的常规总线，主要信息包括当前 PC、指令、访存信息、操作数类型等。
3. Stallreq_for_id:来自 ID 段向 CTRL 模块发送的暂停请求。

3.2.4、 代码介绍

1）regfile




```
regfile u_regfile(
    .clk    (clk    ),
    .raddr1 (rs ),
    .rdata1 (rdata1 ),
    .raddr2 (rt ),
    .rdata2 (rdata2 ),
    .we     (wb_rf_we     ),
    .waddr  (wb_rf_waddr  ),
    .wdata  (wb_rf_wdata  )
);
```

A)端口介绍：

Input：

1.raddr1/2:五位 wire，输入访存目的寄存器，该实验中对应的是指令中的 rs、rt。

2.we:从 WB 段传回的写回信号，控制寄存器的写入。

3.waddr:从 WB 段传回的写回目的寄存器，决定写回哪个寄存器。

4.wdata:从 WB 段传回的写回数据，写入寄存器的内容。

Output：

1.rdata1/2:从 regfile 目的寄存器取出的数据。

B)说明：未对 regfile 做出改动，hilo 寄存器放在了 ID 段，后续将会介绍。

2)

```
assign opcode = inst[31:26];
assign rs = inst[25:21];
assign rt = inst[20:16];
assign rd = inst[15:11];
assign sa = inst[10:6];
```

```
assign func = inst[5:0];
assign imm = inst[15:0];
assign instr_index = inst[25:0];
assign code = inst[25:6];
assign base = inst[25:21];
assign offset = inst[15:0];
assign sel = inst[2:0];
```

说明：对应指令的不同部分，将其划分，方便后续处理。

3）译码器与 inst_xx

```
decoder_6_64 u0_decoder_6_64(
    .in  (opcode  ),
    .out (op_d )
);

decoder_6_64 u1_decoder_6_64(
    .in  (func  ),
    .out (func_d )
);

decoder_5_32 u0_decoder_5_32(
    .in  (rs  ),
    .out (rs_d )
);

decoder_5_32 u1_decoder_5_32(
    .in  (rt  ),
    .out (rt_d )
);
```

```
assign inst_or     = (op_d[6'b00_0000]&&func_d[6'b10_0101]);
assign inst_sw     =  op_d[6'b10_1011];
assign inst_lw     =  op_d[6'b10_0011];
assign inst_xor    =  (op_d[6'b00_0000]&&func_d[6'b10_0110]);
assign inst_sltu   =  (op_d[6'b00_0000]&&func_d[6'b10_1011]);
assign inst_slt    =  (op_d[6'b00_0000]&&func_d[6'b10_1010]);
assign inst_slti   =  op_d[6'b00_1010];
assign inst_sltiu  =  op_d[6'b00_1011];
assign inst_j      =  op_d[6'b00_0010];
assign inst_add    =  (op_d[6'b00_0000]&&func_d[6'b10_0000]);
assign inst_addi   =  op_d[6'b00_1000];
assign inst_sub    =  (op_d[6'b00_0000]&&func_d[6'b10_0010]);
assign inst_and    =  (op_d[6'b00_0000]&&func_d[6'b10_0100]);
assign inst_andi   =  op_d[6'b00_1100];
assign inst_nor    =  (op_d[6'b00_0000]&&func_d[6'b10_0111]);
assign inst_xori   =  op_d[6'b00_1110];
assign inst_sllv   =  (op_d[6'b00_0000]&&func_d[6'b00_0100]);
assign inst_sra    =  (op_d[6'b00_0000]&&func_d[6'b00_0011]);
assign inst_srav   =  (op_d[6'b00_0000]&&func_d[6'b00_0111]);
assign inst_srl    =  (op_d[6'b00_0000]&&func_d[6'b00_0010]);
assign inst_srlv   =  (op_d[6'b00_0000]&&func_d[6'b00_0110]);
assign inst_bgez   =  (op_d[6'b00_0001]&&rt_d[5'b00_001]);
assign inst_bgtz   =  (op_d[6'b00_0111]&&rt_d[5'b00_000]);
assign inst_blez   =  (op_d[6'b00_0110]&&rt_d[5'b00_000]);
assign inst_bltz   =  (op_d[6'b00_0001]&&rt_d[5'b00_000]);
assign inst_bltzal =  (op_d[6'b00_0001]&&rt_d[5'b10_000]);
assign inst_bgezal =  (op_d[6'b00_0001]&&rt_d[5'b10_001]);
```

A) 译码器介绍：

译码器的工作是将 6 位或者 5 位的二进制数转换成只有 1 位是 1 的 64/32 位输出，方便信号的赋值。

B) inst_xx 介绍：

op_d[6'bXX_XXXX]意思是 64 位数据中二进制数 XXXXXX 对应的位置的数据：0/1，其他 XX_d 以此类推，如果每个_d 对应数据都为 1，则相关指令会被点亮，意味着该指令的功能已经确定。

C) 说明：这部分指令参考“A03_“系统能力培养大赛”MIPS 指令系统规范_v1.01”，

具体内容在此不做描述。

4）ALU 操作数选择

```
// rs to reg1 操作数一有三种可能
assign sel_alu_src1[0] = inst_ori
                         |inst_addiu

// pc to reg1
assign sel_alu_src1[1] = inst_jal|inst_bltzal|inst_bgezal|inst_jalr;

// sa_zero_extend to reg1
assign sel_alu_src1[2] = inst_sll|inst_sra|inst_srl;


// rt to reg2 操作数二有四种可能
assign sel_alu_src2[0] = inst_subu
                         |inst_addu
                         |inst_bne

// imm_sign_extend to reg2
assign sel_alu_src2[1] = inst_

// 32'b8 to reg2
assign sel_alu_src2[2] = inst_

// imm_zero_extend to reg2
assign sel_alu_src2[3] = inst_
```

说明：操作数一可选项：rs 寄存器、PC 值、sa 零拓展；

操作数二可选项：rt 寄存器、立即数符号拓展、（PC+8）专用、立即数零拓展；只需要在对应操作数后用指令信号赋值即可。

5）ALU 操作选择

```
assign op_add = inst_addiu|inst_jal|inst_addu|inst_sw
assign op_sub = inst_subu|inst_sub;
assign op_slt = inst_slt|inst_slti ;
assign op_sltu = inst_sltu|inst_sltiu;
assign op_and = inst_and|inst_andi;
assign op_nor = inst_nor;
assign op_or = inst_ori|inst_or;
assign op_xor = inst_xor|inst_xori;
assign op_sll = inst_sll|inst_sllv;
assign op_srl = inst_srl|inst_srlv;
assign op_sra = inst_sra|inst_srav;
assign op_lui = inst_lui;

assign alu_op = {op_add, op_sub, op_slt, op_sltu,
                 op_and, op_nor, op_or, op_xor,
                 op_sll, op_srl, op_sra, op_lui};
```

说明：用法同上，只需要在对应操作后用信号赋值即可。

6)

```
// store in [rd] 根据字段进行地址的写入，看写入的是rt还是rd
assign sel_rf_dst[0] = inst_subu|inst_addu|inst_sll|inst_or|inst_xor
// store in [rt]
assign sel_rf_dst[1] = inst_ori | inst_lui | inst_addiu|inst_lw|inst
// store in [31]，31号寄存器固定用法，某些跳转指令会将地址传入这里
assign sel_rf_dst[2] = inst_jal|inst_bltzal|inst_bgezal;

// sel for regfile address
assign rf_waddr = {5{sel_rf_dst[0]}} & rd
                | {5{sel_rf_dst[1]}} & rt
                | {5{sel_rf_dst[2]}} & 32'd31;
```

说明：写入目的地，根据指令不同去处不同。

7）地址跳转操作

```
assign pc_plus_4 = id_pc + 32'h4;
    assign rs_eq_rt = (r1 == r2);//beq 判断，两个寄存器内容是否相同
    assign rs_neq_rt= (r1!= r2);
    assign br_e = (inst_beq & rs_eq_rt)
                  ||inst_jal
                  ||inst_jr
                  ||(inst_bne & rs_neq_rt)
                  ||inst_j
                  ||(inst_bgez&&(r1[31]==1'b0))
```

```
                ||(inst_bgtz&&(r1!=32'b0)&&(r1[31]==1'b0))
                ||(inst_blez&&((r1[31]!=1'b0)||r1==32'b0))
                ||(inst_bltz&&(r1[31]!=1'b0))
                ||(inst_bltzal&&(r1[31]!=1'b0))
                ||(inst_bgezal&&(r1[31]==1'b0))
                ||inst_jalr;
assign br_addr = inst_beq ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b00}) :
                 inst_jal?({pc_plus_4 [31:28],inst[25:0],2'b00}):
                 inst_jr?(r1):
                 inst_bne?(pc_plus_4+{{14{inst[15]}},{inst[15:0],2'b00}}):
                 inst_j?({pc_plus_4 [31:28],inst[25:0],2'b00}):
                 inst_bgez?(pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b00}):
                 inst_bgtz?(pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b00}):
                 inst_blez?(pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b00}):
                 inst_bltz?(pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b00}):
                 inst_bltzal?(pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b00}):
                 inst_bgezal?(pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b00}):
                 inst_jalr?(r1):
                 32'b0;
```

说明：与具体指令有关，除了判断条件有些许不同，大致一样。

8）访存相关

```
// load and store enable
assign data_ram_en = inst_sw|inst_lw|inst_lb|inst_lbu|inst_lh|inst_lhu|inst_sb|inst_sh ;

// write enable全1为写，全0为读
assign data_ram_wen = (inst_sw||inst_sb||inst_sh)?4'b1111:
                      (inst_lw||inst_lb||inst_lbu||inst_lh||inst_lhu)?4'b0000:4'b0000;
```

说明：1111 表示 store，0000 表示 load，但在后续会有一定改变，我会在 EX 篇讲到，其他操作很简单，只需要把相关信号赋值即可。

9）<u>常规数据相关</u>

这里是 ID 段的重头戏之一，接下来我将会重点讲解：

A）产生原因：处于 ID 段的指令的源操作数需要上一条指令在 EX 段的计算结果，但常规的流水在 WB 段才会将新值写入寄存器，此时 ID 段的指令就会因得不到正确数值导致后续执行错误，因此需要在 EX、MEM 与 ID 之间加入一条数据通路 forwarding，如果 ID 有需要的数就直接用来替换。

B) 数据通路

```
assign{
      ex_lo_wen,
      ex_hi_wen,
      ex_div_mul_flag,
      ex_div_mul_result,
      ex_forwarding_we,
      ex_forwarding_waddr,
      ex_forwarding_wdata
      }=ex_to_id_forwarding;
 assign{
      mem_lo_wen,
      mem_hi_wen,
      mem_div_mul_flag,
      mem_div_mul_result,
      mem_forwarding_we,
      mem_forwarding_waddr,
      mem_forwarding_wdata
      }=mem_to_id_forwarding;
```

注：forwarding 定向路径变量赋值情况。

```
  assign r1 = (ex_forwarding_we &&(ex_forwarding_waddr==rs))?ex_forwarding_wdata:
          ((mem_forwarding_we&&(mem_forwarding_waddr==rs))?mem_forwarding_wdata:
           ((wb_rf_we &&(wb_rf_waddr==rs))?wb_rf_wdata : rdata1));
  assign r2 = (ex_forwarding_we &&(ex_forwarding_waddr==rt))?ex_forwarding_wdata:
          ((mem_forwarding_we&&(mem_forwarding_waddr==rt))?mem_forwarding_wdata:
           ((wb_rf_we &&(wb_rf_waddr==rt))?wb_rf_wdata : rdata2));
```

说明：在 forwarding 总线里，后三个变量与常规数据相关有关系。首先判断各个段是否要写回寄存器且目的寄存器与当前指令的 rs、rt 是否相同，如果相同就将传回来的值存入 r1、r2，后续传给 EX 段。需要注意的是，判断优先级是 EX>MEM>WB>regfile，因为 EX 段距离 ID 段最近，如果发生数据相关，那么 EX 传回的数据一定是最“着急”使用的，所以 EX 的优先级高于其他，以此类推。

10）<u>需要暂停的数据相关</u>

除了常规的数据相关，还有一类更重要的数据相关，这类数据相关需要通过 CTRL 请求暂停来完成。

A) 产生原因：

对于 load 指令，本实验中在 EX 段向内存发出请求，在 MEM 段才能得到内存里的数据，如果按照常规数据相关方式来处理，那么从 EX 段得到的数据只是内存里的目的地址，没有任何意义，所以无论如何都需要等一个周期，因此需要暂停一个周期，等 MEM 段从内存中得到数据后再通过已有的路径传回 ID 段。

B) 解决思路：

从 EX 段传达一个信号给 ID 段，告知 ID 段上一条指令是 load 指令，让 ID 段向 CTRL 申请暂停。

```
assign
stallreq_for_id=(ex_aluop&&((ex_forwarding_waddr==rs)||(ex_forwarding_waddr
==rt)))?1'b1:1'b0;
```

注：ID.v

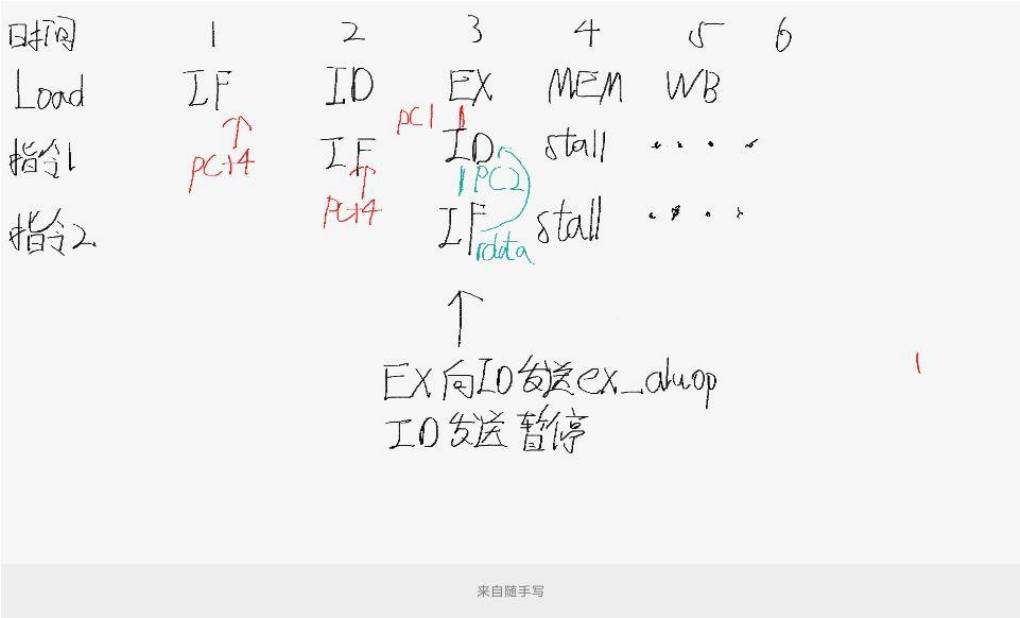
```
assign ex_aluop=(data_ram_en&&(data_ram_wen==4'b0000))?1'b1:1'b0;
```

注：EX.v

说明：EX 段传过来的信号 ex_aluop 表明是否是 load 指令，ID 段通过判断 rs、rt 是否与 waddr 相同且上一条是否是 load 指令，决定是否发送暂停请求。

C) 其它问题：

如果只是简单的进行信号判断，虽然能正常的暂停与从 MEM 段传回正确数值，但也在另一方面出现了错误：



在做实验的时候发现，虽然指令 1 成功暂停了，传回的数值、PC 值也没有错，但如图中展示一样指令 2 的取指访存工作没有暂停，在下一周期，取到的指令将取代在 ID 中暂停的指令 1。这样，原指令被覆盖，不仅 rs、rt 将会被篡改，MEM 段传回的值也将毫无意义：没有解决指令 1 的问题，指令 1 还被跳了过去。

解决方案：因为只暂停一个周期，所以可以将 inst_sram_rdata 暂时存入一个寄存器，需要的时候从该寄存器取值，等暂停结束后再恢复正常，这样就解决了暂停后下一指令覆盖原指令的问题

代码如下：

```
always @ (posedge clk) begin
        if (stall[2]==`Stop && stall[3]==`NoStop) begin
            inst_reg <= inst_sram_rdata;
            stall_flag <= 1'b1;
        end
        else begin
            inst_reg <= 32'b0;
            stall_flag <= 1'b0;
        end
    End
assign inst = stall_flag1 ? inst_reg1 :
        stall_flag ? inst_reg :
          inst_sram_rdata ;
```

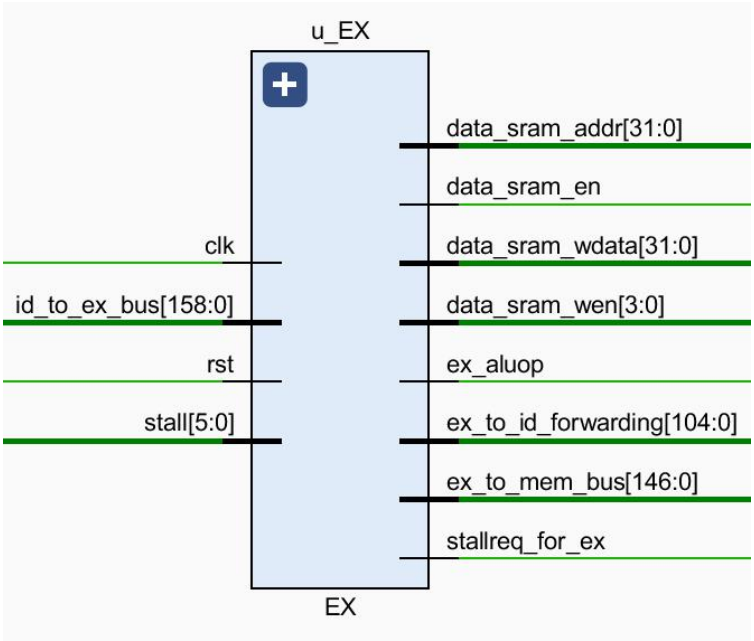
说明：当暂停时，inst_sram_rdata 会存入 inst_reg，在暂停结束后，inst 会恢复正常取值。

3.2.5、 总结

总体来说，ID 段的操作是最复杂的，包括了上述很多的更新。特别是在数据相关的处理上，既考验对组合逻辑的连接也考验对时序逻辑的把控。而与 hilo 寄存器的相关操作，我会在 EX 段给予讲解。

3.3、 EX 段流水功能说明

3.3.1、 功能模块及结构示意图



3.3.2、 整体功能

个人认为整个流水段中最关键的部分之一，进行操作数的计算（alu、mul_div 等），解决常规数据相关，与 MEM 段共同进行访存操作（load/store）等。

3.3.3、 端口介绍

Input：

1.id_to_ex_bus:从 ID 段到 EX 段的总线，与 ID 段功能一样。

2.stall:来自 CTRL 段的暂停命令。

Output：

1.ex_to_mem_bus:从 EX 段到 MEM 段的总线，包括 hilo 使能，EX 段计算结

果，乘除法计算结果等。

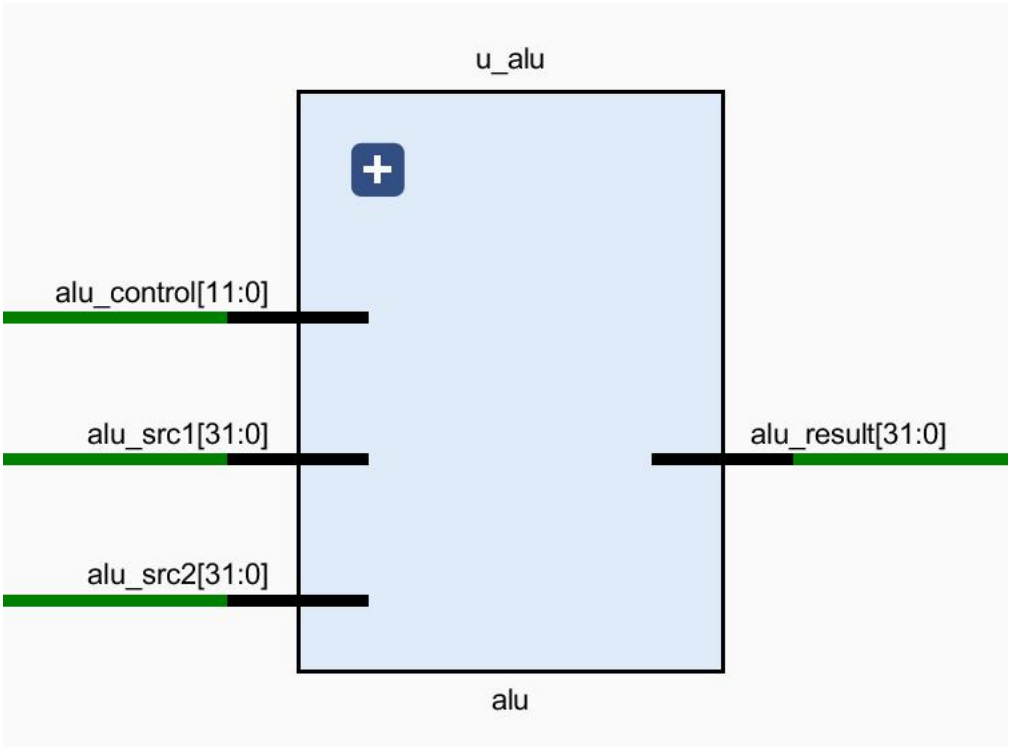
2.data_sram_XXXX:访存功能族，用于 load/store 指令与内存的交互。

3.Ex_to_id_forwarding:将 EX 段计算得出的结果传回 ID 段，解决流水中的常规数据相关（regfile、hilo 等）。

4.ex_aluop:EX 用来提醒 ID 段它的上一条指令是 load 指令，用于进行暂停。

3.3.4、 代码介绍

1）ALU



说明：ALU 内部没有做任何改变，模式也是简单的以两个操作数、运算种类为输入，以运算结果为输出。

```
assign alu_src1 = sel_alu_src1[1] ? ex_pc :
                  sel_alu_src1[2] ? sa_zero_extend : rf_rdata1;
assign alu_src2 = sel_alu_src2[1] ? imm_sign_extend :
                  sel_alu_src2[2] ? 32'd8 :
                  sel_alu_src2[3] ? imm_zero_extend : rf_rdata2;
```

两个操作数与 ID 段的选择完全匹配，具体意思在 ID 段已有讲解。

2）load/store 指令

```
wire[3:0] load_select;
assign load_select=(inst[31:26]==6'b100011)?4'b0000://LW
                   (inst[31:26]==6'b100000)?4'b1001://有符号 LB
                   (inst[31:26]==6'b100100)?4'b0001://无符号 LBU
                   (inst[31:26]==6'b100001)?4'b1011://LH
                   (inst[31:26]==6'b100101)?4'b0011://LHU
                   4'b0000;
wire [31:0] store_select;
assign store_select=(inst[31:26]==6'b101011&&data_ram_wen==4'b1111)?4'b1111://SW
(inst[31:26]==6'b101000&&data_ram_wen==4'b1111&&ex_result[1:0]==2'b00)?4'b0001://SB
(inst[31:26]==6'b101000&&data_ram_wen==4'b1111&&ex_result[1:0]==2'b01)?4'b0010://SB
(inst[31:26]==6'b101000&&data_ram_wen==4'b1111&&ex_result[1:0]==2'b10)?4'b0100://SB
(inst[31:26]==6'b101000&&data_ram_wen==4'b1111&&ex_result[1:0]==2'b11)?4'b1000://SB
(inst[31:26]==6'b101001&&data_ram_wen==4'b1111&&ex_result[1:0]==2'b10)?4'b1100://SH
(inst[31:26]==6'b101001&&data_ram_wen==4'b1111&&ex_result[1:0]==2'b00)?4'b0011://SH 4'b0000;
wire [31:0] store_data;
assign store_data=(inst[31:26]==6'b101011)?rf_rdata2:
                  (ex_result[1:0]==2'b00&&inst[31:26]==6'b101000)?{4{rf_rdata2[7:0]}}:
                  (ex_result[1:0]==2'b01&&inst[31:26]==6'b101000)?{4{rf_rdata2[7:0]}}:
                  (ex_result[1:0]==2'b10&&inst[31:26]==6'b101000)?{4{rf_rdata2[7:0]}}:
                  (ex_result[1:0]==2'b11&&inst[31:26]==6'b101000)?{4{rf_rdata2[7:0]}}:
                  (ex_result[1:0]==2'b00&&inst[31:26]==6'b101001)?{2{rf_rdata2[15:0]}}:
                  (ex_result[1:0]==2'b10&&inst[31:26]==6'b101001)?{2{rf_rdata2[15:0]}}:
                  rf_rdata2;
assign data_sram_addr =ex_result ;
assign data_sram_en =data_ram_en ;
assign data_sram_wen =store_select ;
assign data_sram_wdata =store_data;
```

说明：在加入 load/store 后，原有的访存格式被改变：

首先是 load 指令，因为 load 都是要先从内存中取值，所以 wen 依然为 0000，但我们依然需要将 load 类型的信息提供给 MEM 段，也就是 load_select，例如 LW 类型仍然不变，还是 0000，LB 被分到了 1001，以此类推。load_select 将会在后续 MEM 段处理数据时提供选择的依据。

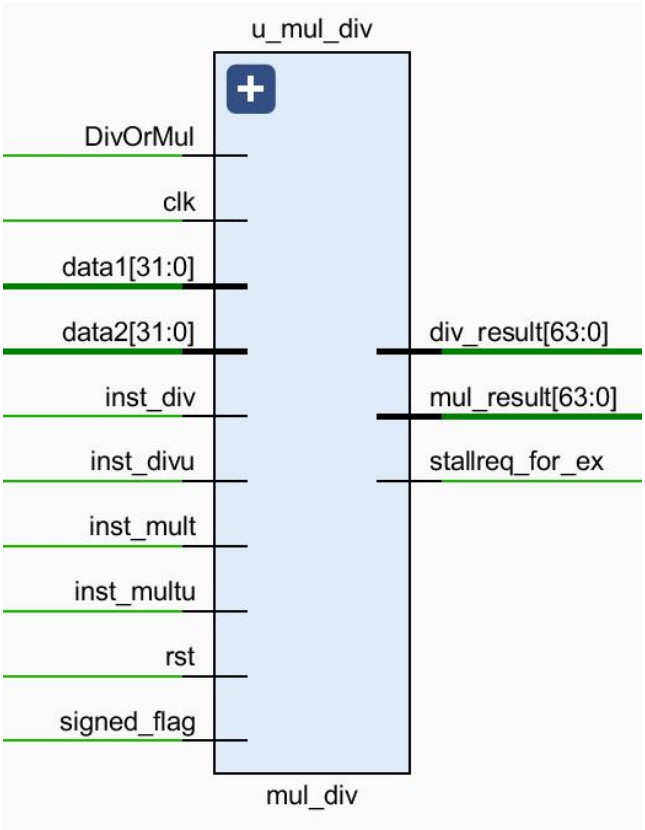
其次是 store 指令，与 load 很大不同的是，该指令在 EX 段就运行结束，因此需要在该段就进行细分，

如代码所示，根据地址的后两位以及 6 位 opcode 的不同，wen 将会细分为

0001,0010,0100,1000,0011,1100,1111，对应 32 位（4 字节）中的不同字节。

3）乘除法器

说明：这里只做简单的展示，具体实现请见后面的可选模块部分。



//输入端口

assign inst_mult = ((inst[31:26] == 6'b000000) && (inst[5:0] == 6'b011000)&&(inst[15:6]==10'b0)) ? 1'b1 : 1'b0 ;

assign inst_multu = (inst[31:26] == 6'b000000) && (inst[5:0] == 6'b011001)&&(inst[15:6]==10'b0) ? 1'b1 : 1'b0 ;

assign inst_div = (inst[31:26] == 6'b000000) && (inst[5:0] == 6'b011010) ? 1'b1 : 1'b0 ;

assign inst_divu = (inst[31:26] == 6'b000000) && (inst[5:0] == 6'b011011) ? 1'b1 : 1'b0 ;

//以各个指令的信号作为输入

assign DivOrMul=(inst_div||inst_divu)?1'b1:(inst_mult||inst_multu)?1'b0:1'b0;//乘除法模式转换，目前未使用

assign data1=DivOrMul?rf_rdata1:alu_src1;//操作数选择，如果是乘法就从 alu 操作数中选取，否则从 ID 段

assign data2=DivOrMul?rf_rdata2:alu_src2;//的 regfile 中选取

assign signed_flag=(inst_div||inst_mult)?1'b1:(inst_divu||inst_multu)?1'b0:1'b0; //是否是有符号的标志

//输出端口

Stallreq_for_ex:从乘除法器内部发送的暂停请求。

如果只是与之前一样处理暂停看似没有问题，但是采用一样的逻辑后，取指只能延迟一个周期，过了一个周期后还是会恢复原样，可乘除法都要暂停 32 个周期，

这显然是不行的，因此，在原有的暂停基础上，加上一些限定条件就可以解决，代码如下：

ID.v:

```
always @ (posedge clk) begin
        if (stall[3]==`Stop && stall[4]==`NoStop && inst_reg1==32'b0) begin
            inst_reg1 <= inst_sram_rdata;
            stall_flag1 <= 1'b1;
        end
        else if(stall[3]==1'b0)begin
            inst_reg1 <= 32'b0;
            stall_flag1 <= 1'b0;
        end
    end
assign inst = stall_flag1 ? inst_reg1 :
    stall_flag ? inst_reg :
     inst_sram_rdata ;
```

与之前 id 发送的暂停不同的是，ex 段的暂停请求多加入了判断寄存器是否为空的条件，如果寄存器里有指令了就不再改变内容了，直到暂停结束，对 inst 的赋值也相对的改变，先判断 EX 暂停再判断 ID 暂停，否则默认从内存取指。

4）HILO 寄存器

对于 HILO 的更新是在处理完 regfile 数据相关后遇到的另一个难题，但 HILO 本质和 regfile 并无区别，对于数据相关的处理也十分相近，接下来是我的处理过程：

问题来源：如果在乘除法指令后有 mf 操作，因为要用到 HI/LO,所以需要将 EX 段对 HI/LO 的更新传回 ID 段，MEM、WB 以此类推。

A) 对于乘除法指令，因为乘除法的结果要同时存入 HILO 寄存器，所以要解决一对的数据相关，代码如下：

```
wire inst_div_or_divu_or_mul;
assign inst_div_or_divu_or_mul =(inst_div||inst_divu)||(inst_mult||inst_multu);
wire [63:0]div_mul_result;
assign div_mul_result=(inst_div||inst_divu)?div_result:
```

```
                        (inst_mult||inst_multu)?mul_result:
                        (inst_mtlo)?{32'b0, rf_rdata1}:
                        (inst_mthi)?{ rf_rdata1,32'b0}:64'b0;
assign ex_to_id_forwarding = {
            lo_wen,
            hi_wen,
            inst_div_or_divu_or_mul,
            div_mul_result,
            rf_we,              // 37
            rf_waddr,           // 36:32
            ex_result           // 31:0
    };
assign ex_to_mem_bus = {
            load_select,
            lo_wen,
            hi_wen,
            inst_div_or_divu_or_mul,
            div_mul_result,
            ex_pc,              // 75:44
            data_ram_en,        // 43
            data_ram_wen,       // 42:39
            sel_rf_res,         // 38
            rf_we,              // 37
            rf_waddr,           // 36:32
            ex_result           // 31:0
    };
```

说明：inst_div_or_divu_or_mul 标定当前指令是否是乘除法；

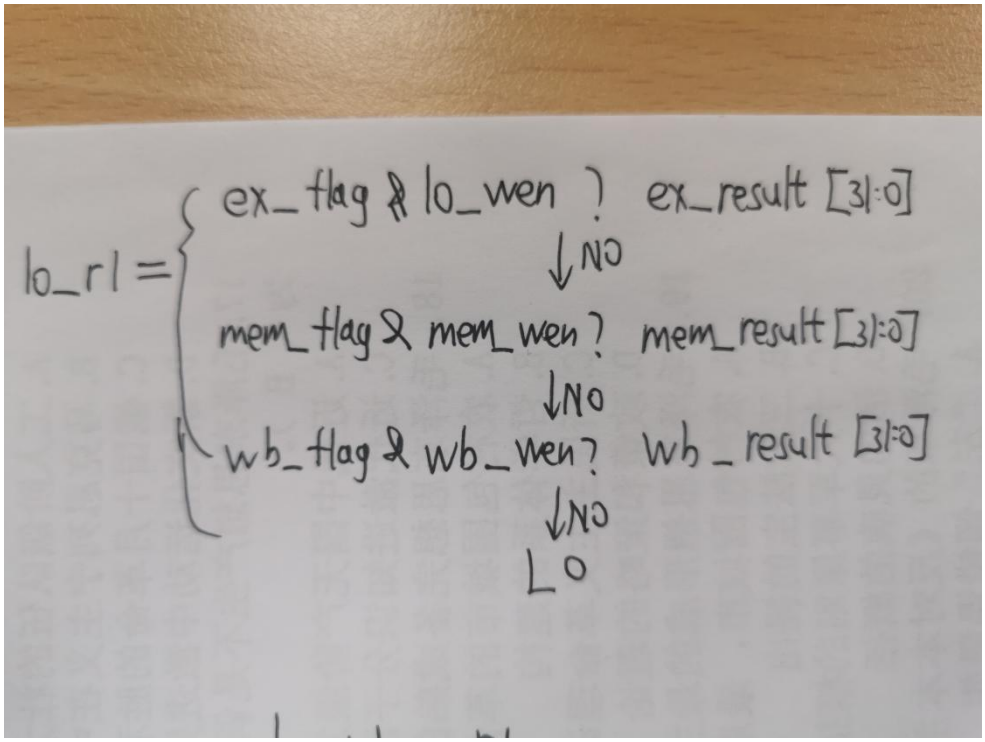
div_mul_result 对传到 ID 段的数据进行选择：除法结果/乘法结果/mtlo/mthi；

在 ID 段也要相应做出改变：

```
reg [31:0] HI;
reg [31:0] LO;
always @ (posedge clk) begin
    if (wb_div_mul_flag!=1'b0||wb_lo_wen==1'b1) begin
        LO<= wb_div_mul_result[31:0];
    end
    if (wb_div_mul_flag!=1'b0||wb_hi_wen==1'b1) begin
        HI<= wb_div_mul_result[63:32];
```

```
            end
        End
assign lo_r1 = ((ex_div_mul_flag==1'b1||ex_lo_wen==1'b1)&&inst_mflo)?ex_div_mul_result[31:0]:
                    ((mem_div_mul_flag==1'b1||mem_lo_wen==1'b1)&&inst_mflo)?mem_div_mul_result[31:0]:
                    ((wb_div_mul_flag==1'b1||wb_lo_wen==1'b1)&&inst_mflo)?wb_div_mul_result[31:0] : LO;
assign hi_r2 =    ((ex_div_mul_flag==1'b1||ex_hi_wen==1'b1)&&inst_mfhi)?ex_div_mul_result[63:32]:
                    ((mem_div_mul_flag==1'b1||mem_hi_wen==1'b1)&&inst_mfhi)?mem_div_mul_result[63:32]:
                    ((wb_div_mul_flag==1'b1||wb_hi_wen==1'b1)&&inst_mfhi)?wb_div_mul_result[63:32] :HI;
assign rr1=(inst_mflo||inst_mfhi)?lo_r1:r1;
assign rr2=(inst_mflo||inst_mfhi)?hi_r2:r2;
```

示意图：



代码说明：

时序逻辑：与 regfile 中寄存器的赋值原理相同，当 WB 段的 HILO 写回信号为 1 或乘除法写回信号为 1 就写入；

组合逻辑：lo_r1/hi_r2 变量是为了解决数据相关问题（r1、r2 是我处理 mtlo、mthi 的一种方式，稍后我会讲解），原理与 ID 段解决常规数据相关问题相近：首先判断 EX 段是否是乘除法操作（ex_div_mul_flag）或者 LO 写入操作（ex_lo_wen），再判断当前指令是否是 mf 指令（inst_mflo/inst_mfhi），如果满足，就将 EX 段的结果传给 lo_r1/hi_r2（ex_div_mul_result），否则按照相同流程

判断 MEM、WB，默认赋值为 HI/LO 寄存器的数据。

B) MFLO、MFHI 操作

Mf 指令的目的是将 hilo 寄存器中的数值存入 regfile 寄存器，因此中间一定要有从 hilo 模式到常规回写的转换，下面是我的思路：

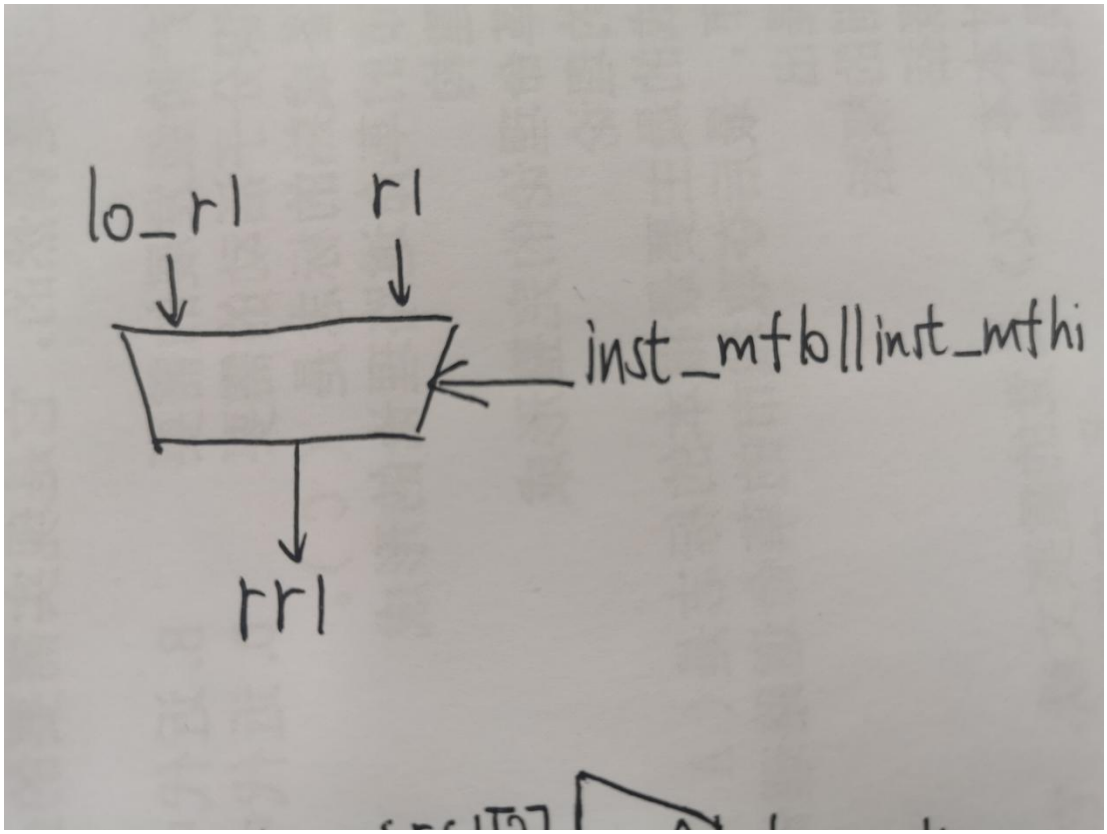
ID.v:

```
assign sel_alu_src1[0] =inst_mflo|.....
assign sel_alu_src2[0] = inst_mfhi |.......
assign sel_rf_dst[0] = inst_mfhi|inst_mflo....
assign rf_we = inst_mfhi|inst_mflo....
//////////////////////////////
assign id_to_ex_bus = {
            id_pc,                  // 158:127
            inst,                   // 126:95
            alu_op,                 // 94:83
            sel_alu_src1,       // 82:80
            sel_alu_src2,       // 79:76
            data_ram_en,        // 75
            data_ram_wen,       // 74:71
            rf_we,                  // 70
            rf_waddr,               // 69:65
            sel_rf_res,         // 64
            rr1,                // 63:32 //EX 段的 rf_rdata1
            rr2                 // 31:0 //EX 段的 rf_rdata2
    };
assign rr1=(inst_mflo||inst_mfhi)?lo_r1:r1;
assign rr2=(inst_mflo||inst_mfhi)?hi_r2:r2;
```

EX.v:

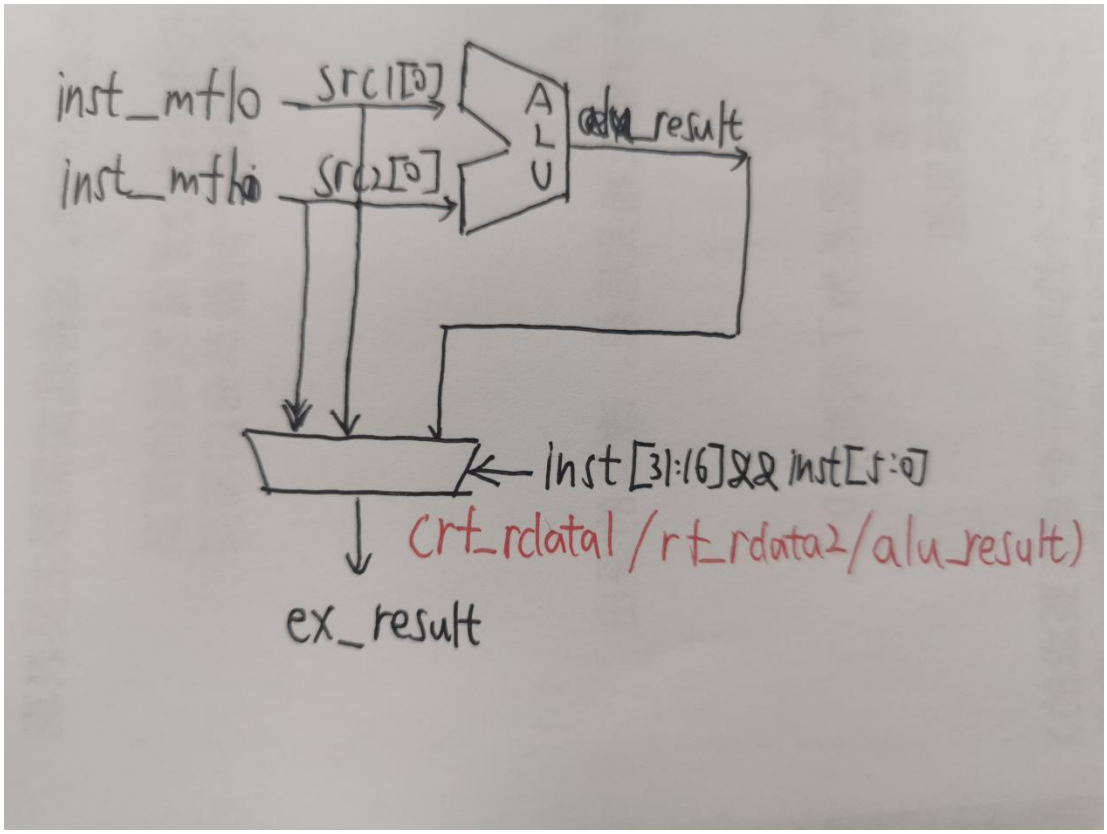
```
assign ex_result =((inst[31:16]==16'b0)&&(inst[5:0]==6'b010010))?rf_rdata1:
                  ((inst[31:16]==16'b0)&&(inst[5:0]==6'b010000))?rf_rdata2:alu_result;
```

说明:既然要进行模式转换，那么我将 lo 寄存器的数据与 rdata1（r1）进行合并，将 hi 寄存器与 rdata2（r2）进行合并（rr1、rr2），rr1、rr2 将会随着总线传给 EX 段，如图所示：



按照寄存器赋值操作，将该赋值的信号都给赋值（sel_rf_dst[0] 、rf_we）。

紧接着在 EX 段，通过判断 inst 是否是 mf 指令，决定 ex_result 的类型，再通过 ex_to_mem 总线和 forwarding 定向路径输出，如图所示：



这样，mf 指令就从 hilo 模式回到了 rf 变量，进行 regfile 的赋值，这样就减少了线宽，增加了通用性。

C）MTLO、MTHI 操作

MTLO、MTHI 的操作是将 regfile 寄存器的值传给 hilo 寄存器，是从 regfile 模式转换到 hilo 模式，我的解决方案是：为 HI、LO 单独创建一个写使能，充分利用解决乘除法时的变量 div_mul_result——高位储存 hi 值，低位储存 lo 值，这样既将数据传回了 ID 段，也没有扩大太多线宽，且可以直接利用现有的解决数据相关的方法，写使能信号会一直传到 WB 段，并在每个流水段传回 ID 段以解决数据相关问题，代码如下：

EX.v

```
assign inst_mtlo=((inst[31:26] == 6'b000000) && (inst[5:0] == 6'b010011)&&(inst[20:6]==15'b0)) ? 1'b1 : 1'b0 ;
assign inst_mthi=((inst[31:26] == 6'b000000) && (inst[5:0] == 6'b010001)&&(inst[20:6]==15'b0)) ? 1'b1 : 1'b0 ;
assign div_mul_result=(inst_div||inst_divu)?div_result:
```

```
                        (inst_mult||inst_multu)?mul_result:
                        (inst_mtlo)?{32'b0, rf_rdata1}:
                        (inst_mthi)?{ rf_rdata1,32'b0}: 64'b0;
assign lo_wen=inst_mtlo?1'b1:1'b0;
assign hi_wen=inst_mthi?1'b1:1'b0;
```

ID.v

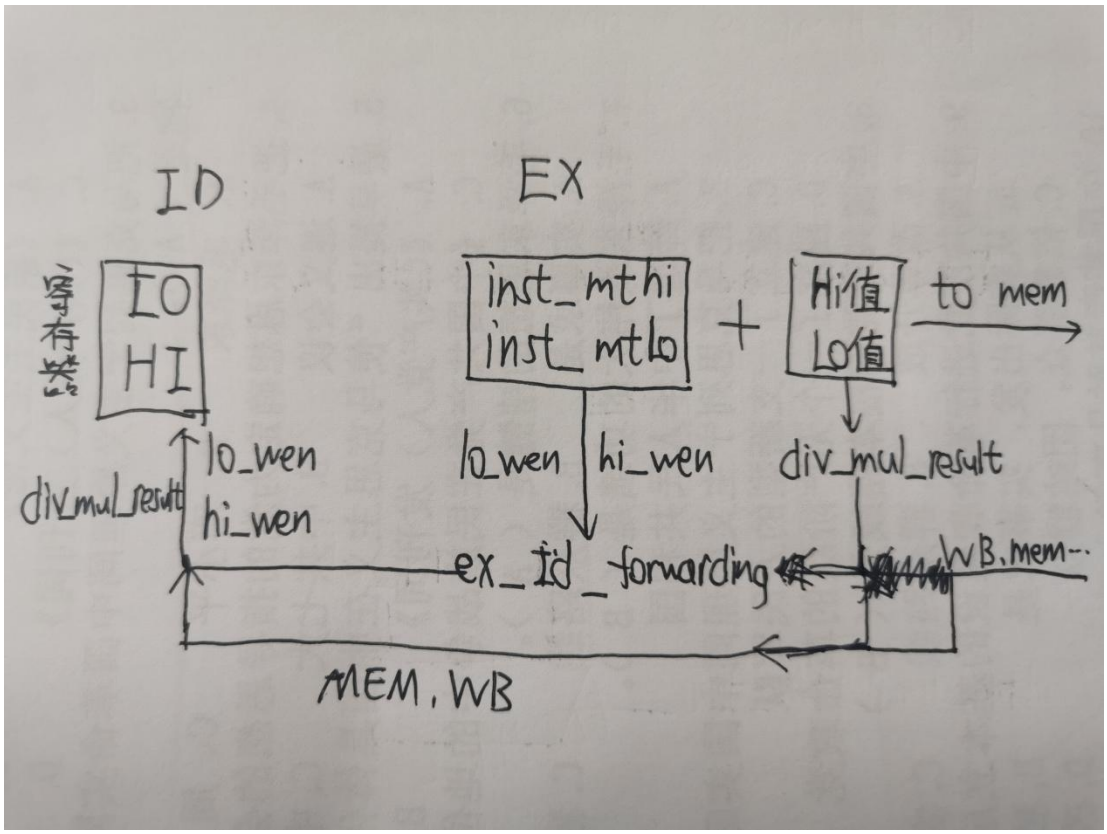
```
assign{
      ex_lo_wen,
      ex_hi_wen,
      ex_div_mul_flag,
      ex_div_mul_result,
      ex_forwarding_we,
      ex_forwarding_waddr,
      ex_forwarding_wdata
      }=ex_to_id_forwarding;
 assign{
      mem_lo_wen,
      mem_hi_wen,
      mem_div_mul_flag,
      mem_div_mul_result,
      mem_forwarding_we,
      mem_forwarding_waddr,
      mem_forwarding_wdata
      }=mem_to_id_forwarding;
 always @ (posedge clk) begin
          if (wb_div_mul_flag!=1'b0||wb_lo_wen==1'b1) begin
              LO<= wb_div_mul_result[31:0];
          end
          if (wb_div_mul_flag!=1'b0||wb_hi_wen==1'b1) begin
              HI<= wb_div_mul_result[63:32];
          end
    End

Assign_lo_r1=
((ex_div_mul_flag==1'b1||ex_lo_wen==1'b1)&&inst_mflo)?ex_div_mul_result[31:0]:
((mem_div_mul_flag==1'b1||mem_lo_wen==1'b1)&&inst_mflo)?mem_div_mul_result[31:0]:
((wb_div_mul_flag==1'b1||wb_lo_wen==1'b1)&&inst_mflo)?wb_div_mul_result[31:0] : LO;
Assign_hi_r2=
((ex_div_mul_flag==1'b1||ex_hi_wen==1'b1)&&inst_mfhi)?ex_div_mul_result[63:32]:
((mem_div_mul_flag==1'b1||mem_hi_wen==1'b1)&&inst_mfhi)?mem_div_mul_result[63:32]:
((wb_div_mul_flag==1'b1||wb_hi_wen==1'b1)&&inst_mfhi)?wb_div_mul_result[63:32] :HI;
```

说明：EX 段代码中，创建了两个 MT 信号（inst_mtlo/inst_mthi）用于识别 mt 指令，对于乘除法变量赋值（div_mul_result）更新为：判断完是否是乘除法后再

判断是否是 mt 指令。

ID 段代码中，依次判断 EX、MEM、WB 中的 hi/lo_wen 写使能信号，如果为 1 且下一条是 mf 指令，就和乘除法一样，用 forwarding 中的数据直接替换，值得注意的是，LO 寄存器与 r1 是几乎绑定的关系，同样的 HI 寄存器与 r2 也是几乎绑定的，一方面这样做可以有效利用已有的线宽，另一方面也体现了模式转换的思路。

过程如图所示：

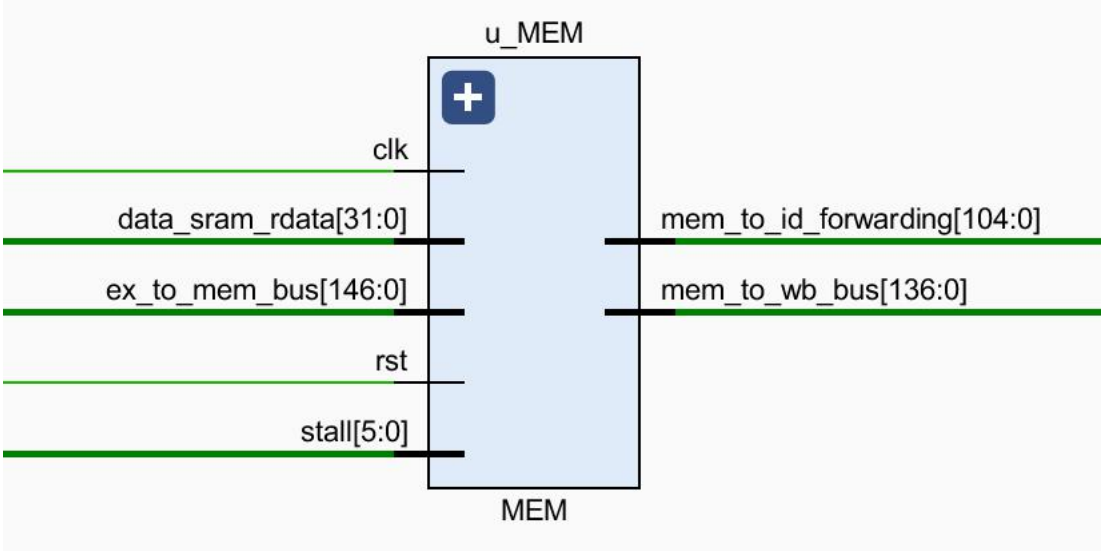


3.3.5、 总结

ID 与 EX 段应该说是最有挑战的两个流水段了，在 EX 段，我们解决了 load/store 指令，乘除法问题、MT 指令、MF 指令以及一系列数据相关问题。关于乘除法我会在可选模块做出详细阐述。

3.4、 MEM 段流水功能说明

3.4.1、 功能模块及结构示意图



3.4.2、 整体功能

进行与寄存器有关的操作，与 EX 共同完成 load/store 指令的执行，解决必须暂停的数据相关。

3.4.3、 端口介绍

Input：

1.ex_to_mem_bus:从 EX 段到 MEM 段的总线。

2.data_sram_rdata:EX 段访存(主 load)的结果通过该线输入到 MEM 段，存有数据。

3.stall:收到 CTRL 模块发送的暂停指令。

Output：

1.mem_to_id_forwarding:将从内存得到的数据或之前传输的数据传回 ID 段，优先级低于 EX 段 forwarding。

2.mem_to_wb_bus:MEM 到 WB 的常规总线，主要信息包括当前 hilo 使能、乘除法信息、回写寄存器信息等。

3.4.4、 代码介绍

1)

```
assign {
    load_select,
    lo_wen,
    hi_wen,
    inst_div_or_divu_or_mul,
    div_mul_result,
    mem_pc,          // 75:44
    data_ram_en,     // 43
    data_ram_wen,    // 42:39
    sel_rf_res,      // 38
    rf_we,           // 37
    rf_waddr,        // 36:32
    ex_result        // 31:0
} =  ex_to_mem_bus_r;
```

从上到下依次为：load 选择指令、LO 写使能、HI 写使能、乘除法指令信号、乘除法结果、MEM 段 PC，内存访问信号、内存写使能线，选择，回写信息（rf_XXX）

说明：从 EX 段传输的总线赋值情况。

2)

```
wire [31:0]load_deal;
assign load_deal= (load_select==4'b0000)?data_sram_rdata:
                  (load_select==4'b1001&&ex_result[1:0]==2'b00)?{{24{data_sram_rdata[7]}},data_sram_rdata[7:0]}:
                  (load_select==4'b1001&&ex_result[1:0]==2'b01)?{{24{data_sram_rdata[15]}},data_sram_rdata[15:8]}:
                  (load_select==4'b1001&&ex_result[1:0]==2'b10)?{{24{data_sram_rdata[23]}},data_sram_rdata[23:16]}:
                  (load_select==4'b1001&&ex_result[1:0]==2'b11)?{{24{data_sram_rdata[31]}},data_sram_rdata[31:24]}:
                    (load_select==4'b0001&&ex_result[1:0]==2'b00)?{24'b0,data_sram_rdata[7:0]}:
                    (load_select==4'b0001&&ex_result[1:0]==2'b01)?{24'b0,data_sram_rdata[15:8]}:
                    (load_select==4'b0001&&ex_result[1:0]==2'b10)?{24'b0,data_sram_rdata[23:16]}:
                    (load_select==4'b0001&&ex_result[1:0]==2'b11)?{24'b0,data_sram_rdata[31:24]}:
                      (load_select==4'b0011&&ex_result[1:0]==2'b00)?{16'b0,data_sram_rdata[15:0]}:
                      (load_select==4'b0011&&ex_result[1:0]==2'b10)?{16'b0,data_sram_rdata[31:16]}:
                        (load_select==4'b1011&&ex_result[1:0]==2'b00)?{{16{data_sram_rdata[15]}},data_sram_rdata[15:0]}:
                        (load_select==4'b1011&&ex_result[1:0]==2'b10)?{{16{data_sram_rdata[31]}},data_sram_rdata[31:16]}:
                          data_sram_rdata;
assign rf_wdata = (data_ram_wen==4'b0000&&data_ram_en==1'b1)?load_deal:sel_rf_res ? mem_result : ex_result;
```

附：EX.v

```
assign load_select=(inst[31:26]==6'b100011)?4'b0000://LW
                   (inst[31:26]==6'b100000)?4'b1001://有符号LB
                   (inst[31:26]==6'b100100)?4'b0001://无符号LBU
                   (inst[31:26]==6'b100001)?4'b1011://LH
                   (inst[31:26]==6'b100101)?4'b0011://LHU
                   4'b0000;
```

功能：

A) load_deal:结合 EX 段的 load_select 可以得知，在对 load 进行选择以后，根据 ex_result 的结果，即 EX.ALU 的地址计算结果的后两位决定写入寄存器的数据内容。比如对于 LB 指令（load_select==2'b1001），地址后两位 00，表示将内存中数据的 0-7 位作符号拓展存入 load_deal 中，以此类推。

B) rf_wdata:(data_ram_wen==4'b0000&&data_ram_en==1'b1)用来判断是否是 load 指令，如果是 load 指令，则回写的数据是 load_deal 的结果，即经过处理的内存数据，否则选择 EX 段传过来的计算结果，这一步的选择目的是解决需要暂停的数据相关。

3)

```
assign mem_to_wb_bus = {
     lo_wen,
     hi_wen,
     inst_div_or_divu_or_mul,
     div_mul_result,
    mem_pc,       // 41:38
    rf_we,        // 37
    rf_waddr,     // 36:32
    rf_wdata      // 31:0
};
```

从上到下依次为：LO 写使能、HI 写使能、乘除法指令信号、乘除法结果、MEM 段 PC，回写信息（rf_XXX）

说明：要传输到 WB 段的总线赋值情况。

4)

```
assign mem_to_id_forwarding = {
        lo_wen,
         hi_wen,
        inst_div_or_divu_or_mul,
        div_mul_result,
        rf_we,        // 37
        rf_waddr,     // 36:32
        rf_wdata      // 31:0
    };
```

从上到下依次为：LO 写使能、HI 写使能、乘除法指令信号、乘除法结果、MEM

段 PC，回写信息（rf_XXX）

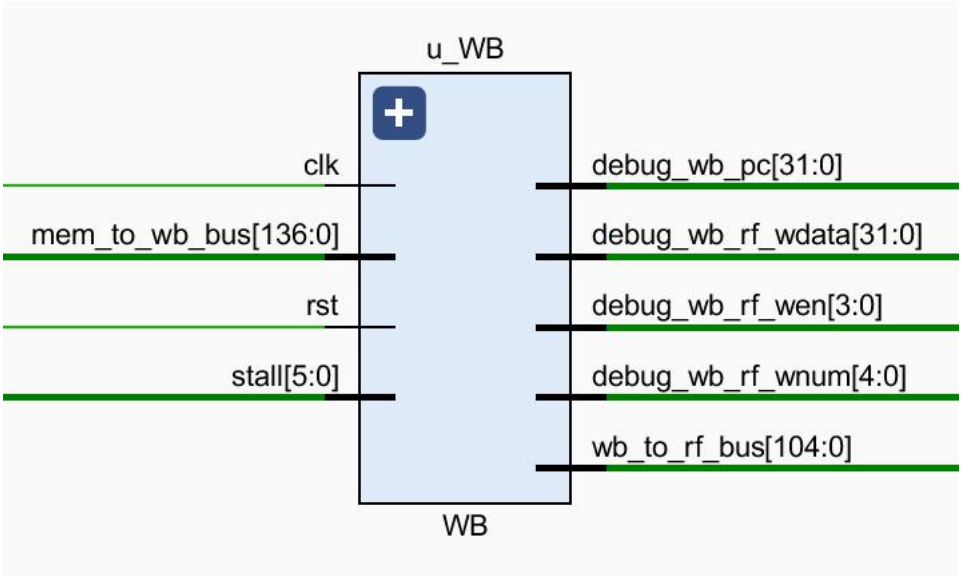
说明：要传输到 ID 段的数据通路，用于解决流水的数据相关问题，优先级低于 EX_forwarding。

3.4.5、 总结

MEM 更新的主要内容：load 指令的更新伴随着存入寄存器内容的改变，通过 load_deal 解决这一问题；另外，通过对 rf_wdata 的选择也解决了需要暂停的数据相关。

3.5、 WB 段流水功能说明

3.5.1、 功能模块及结构示意图



3.5.2、 整体功能

负责写入 regfile、hilo 寄存器。

3.5.3、 端口介绍

Input：

1.mem_to_wb_bus:从 EX 段到 MEM 段的总线。

2.stall:收到 CTRL 模块发送的暂停指令。

Output：

1.wb_to_rf_bus:将数据传回 ID 段的 regfile、hilo 中。

3.5.4、 代码介绍

1)

```
assign {
    lo_wen,
    hi_wen,
    inst_div_or_divu_or_mul,
    div_mul_result,
    wb_pc,
    rf_we,
    rf_waddr,
    rf_wdata
} = mem_to_wb_bus_r;
```

从上到下依次为：LO 写使能、HI 写使能、乘除法指令信号、乘除法运算结果、回写段 PC 值、回写寄存器信息（rf_XX）

说明：由 MEM 段传入的总线赋值情况。

2)

```
assign wb_to_rf_bus = {
    lo_wen,
    hi_wen,
    inst_div_or_divu_or_mul,
    div_mul_result,
    rf_we,
    rf_waddr,
    rf_wdata
};
```

从上到下依次为：LO 写使能、HI 写使能、乘除法指令信号、乘除法运算结果、回写寄存器信息（rf_XX）。

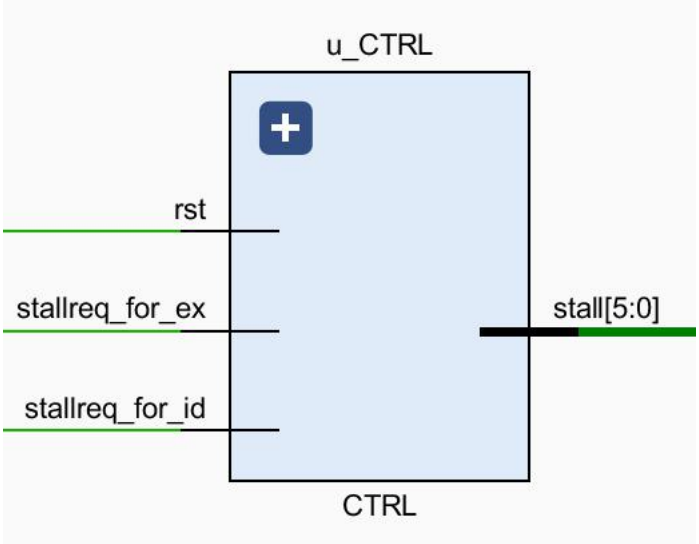
说明：传入 ID 段的总线，用来写入 regfile 和 hilo 寄存器。

3.5.5、 总结

WB 段的改变只有线宽的变化，与最初相比多传了 hilo 寄存器和乘除法的相关信息，没有太多需要赘述的。

3.6、 CTRL 段流水功能说明

3.6.1、 功能模块及结构示意图



3.6.2、 整体功能

进行整个流水过程的暂停控制。

3.6.3、 端口介绍

Input：

1.stallreq_for_ex:收到来自 EX 段的暂停请求。

2.stallreq_for_id:收到来自 ID 段的暂停请求。

Output：

1.stall:将暂停命令发送给各个流水段。

3.6.4、 代码介绍

```
always @ (*) begin
    if (rst) begin
        stall = `StallBus'b0;
    end else if(stallreq_for_ex==1'b1) begin
       stall=6'b001111;
    end else if(stallreq_for_id==1'b1) begin
       stall=6'b000111;
    end else begin
        stall = `StallBus'b0;
    end
end
```

通过 EX、ID 段发送的请求调整暂停。

stall[0]表示取指地址 PC 是否保持不变，为 1 表示保持不变。

stall[1]表示流水线取指阶段是否暂停，为 1 表示暂停。

stall[2]表示流水线译码阶段是否暂停，为 1 表示暂停。

stall[3]表示流水线执行阶段是否暂停，为 1 表示暂停。

stall[4]表示流水线访存阶段是否暂停，为 1 表示暂停。

stall[5]表示流水线回写阶段是否暂停，为 1 表示暂停。

3.6.5、 总结

CTRL 段整体没有太多可讲内容，与最初的版本最大的不同在于加入了两个输入，改进了内部判定条件，使其可以处理来自 EX、ID 段的暂停请求。

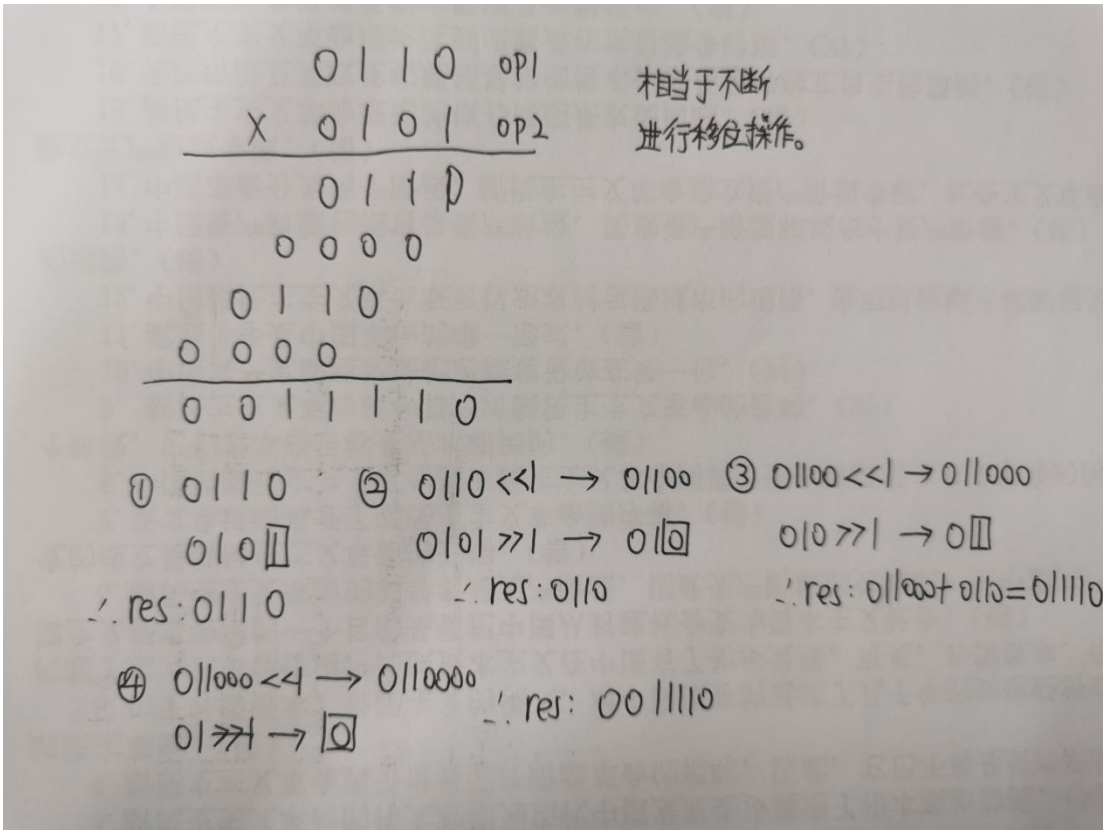
4、 乘法器以及合并乘除法器说明（可选模块）

在 EX 段，我只是简单讲解了乘除法器的外部端口，这里我着重讲述两个方面，一个是乘法器原理、一个是合并乘除法器的原理。

A)乘法器设计

1）乘法器文件：

2）乘法器原理



乘法原理：op2 从最低位开始依次判断是否为 1，若为 1 就将 op1 最低位与 op2 当前位对齐，最后加到一起。因此，在流水中可以让 op2 与 op1 相反方向进行移位，从最低位开始，如果是 1 就将移位后的 op1 加入到结果中，否则保持原值不动，所以 op2 有几位，这个过程就进行几次，本实验中均是 32 位运算，所以循环要进行 32 次，这是判断乘法结束的重要条件。

3）代码解读：

```
module mul_32_1(
    input wire rst,
    input wire clk,
    input wire signed_mul_i,
    input wire[31:0] opdata1_i,
    input wire[31:0] opdata2_i,
    input wire start_i,                          ///是否开始乘法运算
    input wire annul_i,                          //是否取消乘法运算，1 位取消
    output reg signed [63:0] result_o,                  //乘法运算结果
    output reg ready_o
    );
```

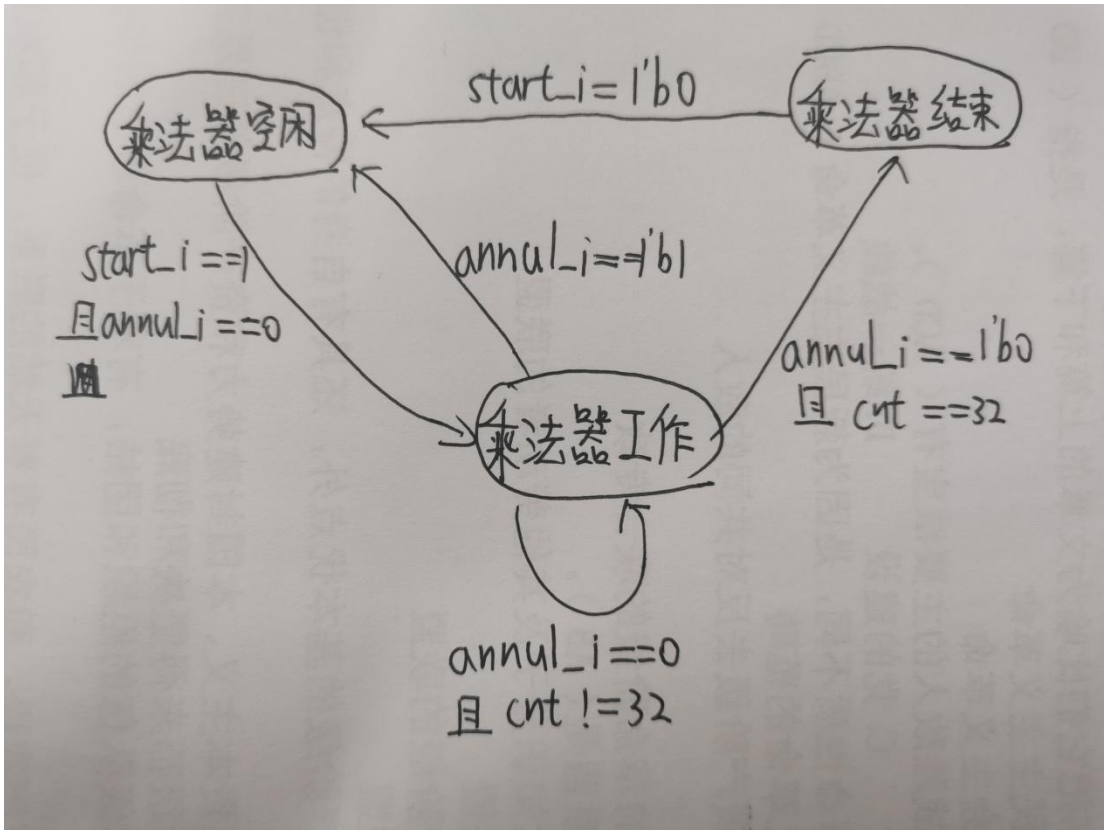
说明：因为受到除法器的启发，因此在输入输出端口与除法器没有太大区别，无

非是改变了变量名，变量含义已在注释中。

```
reg [5:0] cnt;                                  //记录乘法进行了几轮
reg [1:0] state;                                //乘法器处于的状态
reg[63:0] mult1_shift;
reg[31:0] mult2_shift;
reg[63:0] mult1_acc;
reg[31:0] neg1;
reg[31:0] neg2;
always @ (posedge clk) begin
    if (rst) begin
        state <= 2'b00;
        result_o <= {32'b0,32'b0};
        ready_o <= 1'b0;
    end else begin
        case(state)
            2'b00: begin                //空闲
                if (start_i == 1'b1 && annul_i == 1'b0) begin
                    state <= 2'b10;
                    cnt <= 6'b000000;
                    if(signed_mul_i == 1'b1 && opdata1_i[31] == 1'b1) begin
        ///负数
                        mult1_shift <= {32'b0,(~opdata1_i + 1)};
                    end else begin
                        mult1_shift <= {32'b0,opdata1_i};
                    end
                    if (signed_mul_i == 1'b1 && opdata2_i[31] == 1'b1 ) begin
        //负数
                        mult2_shift <= (~opdata2_i + 1);
                    end else begin
                        mult2_shift <= opdata2_i;
                    end
                    begin
                        mult1_acc<=64'b0;
                     end
                end else begin
                  ready_o <= 1'b0;
                  result_o <= {32'b0, 32'b0};
                end
            end
            2'b10: begin
                if(annul_i == 1'b0) begin               //进行乘法运算
                    if(cnt != 6'b100000) begin
                    mult1_acc
```

```
<=(mult2_shift[0]==1'b1)?(mult1_acc+mult1_shift):mult1_acc;
                                mult1_shift<=mult1_shift<<1;
                                mult2_shift<=mult2_shift>>1;

                                cnt <= cnt +1;          //乘法运算次数
                            end else begin
                                                state <= 2'b11;
                                                cnt <= 6'b000000;
                                                end
                        end else begin
                            state <= 2'b00;
                        end
                    end
                    2'b11: begin                //乘法结束
                        result_o    <=    (((opdata1_i[31]    ==    1'b1&&opdata2_i[31]    ==
1'b1)||(opdata1_i[31]    ==    1'b0&&opdata2_i[31]    ==    1'b0)&&signed_mul_i    ==
1'b1)||signed_mul_i == 1'b0)?mult1_acc:(~mult1_acc+1);
                        ready_o <= 1'b1;
                        if (start_i == 1'b0) begin
                            state <= 2'b00;
                            ready_o <= 1'b0;
                            result_o <= {32'b0, 32'b0};
                        end
                    end
                endcase
            end
        end
endmodule
```

说明：乘法器与除法器最大的不同在于少了除数为零的状态，状态图如下：、



有了状态图我们了解到了最重要的信息：主要的运算在乘法器工作状态完成，其他部分只负责控制，因此我们只需要在工作状态进行修改，其他的不需要改变即可：

```
2'b00: begin                //空闲
                    if (start_i == 1'b1 && annul_i == 1'b0) begin
                                state <= 2'b10;
                                cnt <= 6'b000000;
                                if(signed_mul_i == 1'b1 && opdata1_i[31] == 1'b1) begin
            ///负数
                                    mult1_shift <= {32'b0,(~opdata1_i + 1)};
                                end else begin
                                    mult1_shift <= {32'b0,opdata1_i};
                                end
                                if (signed_mul_i == 1'b1 && opdata2_i[31] == 1'b1 ) begin
            //负数
                                    mult2_shift <= (~opdata2_i + 1);
                                end else begin
                                    mult2_shift <= opdata2_i;
                                end
                                begin
                                    mult1_acc<=64'b0;
```

```
                    end
                end else begin
                    ready_o <= 1'b0;
                    result_o <= {32'b0, 32'b0};
                end
            End
```

当乘法器处于空闲时，会先进行操作数处理，如果是有符号运算，那么先将操作数转换成补码，并将累加结果变量 acc 初始化，将乘法次数清零，将状态变为 10，即将开始乘法计算。

```
2'b10: begin
        if(annul_i == 1'b0) begin                    //进行乘法运算
            if(cnt != 6'b100000) begin
        mult1_acc<=(mult2_shift[0]==1'b1)?(mult1_acc+mult1_shift):mult1_acc;
                    mult1_shift<={mult1_shift[62:0],1'b0};
                    mult2_shift<={1'b0,mult2_shift[31:1]};
                    cnt <= cnt +1;              //乘法运算次数
                end else begin
                                state <= 2'b11;
                                cnt <= 6'b000000;
                                end
                end else begin
                    state <= 2'b00;
                end
            end
```

当 cnt 不为 32 时，执行乘法运算——mult1_shift 是储存操作数 1 的移位结果，mult2_shift 储存操作数 2 的移位结果，mult1_acc 储存累加结果，当 acc==32 后，state 会转变为 11，意思是进入乘法结束状态，并将 cnt 重新清零。

```
2'b11: begin                //乘法结束
    result_o <= (((opdata1_i[31] == 1'b1&&opdata2_i[31] == 1'b1)||(opdata1_i[31] == 1'b0&&opdata2_i[31] == 1'b0)&&signed_mul_i == 1'b1)||signed_mul_i == 1'b0)?mult1_acc:(~mult1_acc+1);
                ready_o <= 1'b1;
                if (start_i == 1'b0) begin
                    state <= 2'b00;
                    ready_o <= 1'b0;
                    result_o <= {32'b0, 32'b0};
                end
            end
```

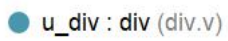
当状态达到结束的时候，若是有符号数，则根据正负数选择 acc 或 acc 补码赋值，若是无符号数，则直接将 acc 赋给 result，输出至乘法器外部。

乘法外部的暂停与除法的暂停相同，因此会在乘除法合并时阐述。

B)合并乘除法器

思路：正如乘法器中讲的一样，除了工作状态有改变之外，其余状态没有区别（乘法只少了除数为零这一状态），因此主要的改变在 DIVON：

1）文件



2）代码：

```
module div(
    input wire rst,
    input wire clk,
    input wire signed_div_i,
    input wire[31:0] opdata1_i,
    input wire[31:0] opdata2_i,
    input wire start_i,                 ///是否开始除法运算
    input wire annul_i,                 //是否取消除法运算，1 位取消
    output reg[63:0] result_o,          //除法运算结果
    output reg ready_o,                     //除法运算是否结束
    input wire [1:0]sel
);
```

除法器的模块，只多了 sel 输入，目的是标志运算类型：01 为乘法、10 为除法、默认 00。

```
wire [32:0] div_temp;
    reg [5:0] cnt;                  //记录试商法进行了几轮
    reg[64:0] dividend;                 //低 32 位保存除数、中间结果，第 k 次迭代结束的时候 dividend[k:0]保存的就是当前得到的中间结果，
//dividend[31:k+1]保存的是被除数没有参与运算的部分，dividend[63:32]是每次迭代时的被减数
    reg [1:0] state;                //除法器处于的状态
    reg[31:0] divisor;
    reg[63:0] emp_op1;
    reg[31:0] temp_op1;//shift1
    reg[31:0] temp_op2;//shift2
    reg[63:0] mult1_acc;
```

```
assign div_temp = {1'b0, dividend[63: 32]} - {1'b0, divisor};
```

变量声明里我们只加入了 emp_op1、mult1_acc,分别用于储存 64 位零扩展的操作数 1、乘法累加结果。

```
`DivFree: begin                    //除法器空闲
    if (start_i == `DivStart && annul_i == 1'b0) begin
    if(opdata2_i == `ZeroWord&&sel==2'b10) begin                ///如果除数为 0
                    state <= `DivByZero;
                end else begin
                    state <= `DivOn;                            //除数不为 0
                    cnt <= 6'b000000;
                    if(signed_div_i == 1'b1 && opdata1_i[31] == 1'b1) begin
    ///被除数为负数
                        temp_op1 = ~opdata1_i + 1;
                        emp_op1 = {32'b0,~opdata1_i + 1};
                    end else begin
                        emp_op1 = {32'b0,opdata1_i}
                        temp_op1 = opdata1_i;
                    end
                    if (signed_div_i == 1'b1 && opdata2_i[31] == 1'b1 ) begin
    //除数为负数
                        temp_op2 = ~opdata2_i + 1;
                    end else begin
                        temp_op2 = opdata2_i;
                    end
                    mult1_acc <=64'b0;
                    if(sel==2'b10)begin
                    dividend <= {`ZeroWord, `ZeroWord};
                    dividend[32: 1] <= temp_op1;
                    divisor <= temp_op2;
                    end
                end
            end else begin
                ready_o <= `DivResultNotReady;
                result_o <= {`ZeroWord, `ZeroWord};
            end
        End
```

乘除法器空闲时，先做判断，如果是除法操作且除数为零，进入“除数为零”状态，否则进行数据处理，这个部分乘除法没有区别，只是乘法需要扩充 32 个 0 至 emp_op1（64 位，给 op1 移位预留空间）,且 acc 需要清零。如果是除法（10），

则按照 if 语句正常进行。

```
`DivByZero: begin                    //除数为 0
                dividend <= {`ZeroWord, `ZeroWord};
                state <= `DivEnd;
            end

`DivOn: begin                        //除数不为 0
        if(annul_i == 1'b0) begin                    //进行除法运算
                if(cnt != 6'b100000&&sel==2'b10) begin
                        if (div_temp[32] == 1'b1) begin
                            dividend <= {dividend[63:0],1'b0};
                        end else begin
                            dividend <= {div_temp[31:0],dividend[31:0], 1'b1};
                        end
                        cnt <= cnt +1;              //除法运算次数
                    end else if(sel==2'b10) begin
         if ((signed_div_i == 1'b1) && ((opdata1_i[31] ^ opdata2_i[31]) == 1'b1)) begin
                            dividend[31:0] <= (~dividend[31:0] + 1);
                        end
          if ((signed_div_i == 1'b1) && ((opdata1_i[31] ^ dividend[64]) == 1'b1)) begin
                            dividend[64:33] <= (~dividend[64:33] + 1);
                        end
                        state <= `DivEnd;
                        cnt <= 6'b000000;
                    end
                    if(cnt != 6'b100000&&sel==2'b01) begin
        mult1_acc <=(temp_op2[0]==1'b1)?(mult1_acc+emp_op1):mult1_acc;
                        emp_op1<={emp_op1[62:0],1'b0};
                        temp_op2<={1'b0,temp_op2[31:1]};
                        cnt <= cnt +1;              //乘法运算次数
                    end else if(sel==2'b01) begin
                        state <= 2'b11;
                        cnt <= 6'b000000;
                    end

                end else begin
                    state <= `DivFree;
                end
            End
```

如果除数为零，进入相应状态。当乘除法器启动后，进入 ON 状态，根据 sel 进行选择，如果是 10，则进入除法的 if 语句，如果是 01，则进入乘法的 if 语句，

乘法计算过程与 32 周期乘法器 mul_32_1 完全相同。无论是哪种计算，运算时 cnt 都会累加，结束后都会进入 END 状态。

```
`DivEnd: begin                    //除法结束
            result_o <= (sel==2'b10)?{dividend[64:33], dividend[31:0]}:
              (sel==2'b01)?(((opdata1_i[31] == 1'b1&&opdata2_i[31] == 1'b1)||(opdata1_i[31] == 1'b0&&opdata2_i[31] == 1'b0)&&signed_div_i == 1'b1)||signed_div_i == 1'b0)?mult1_acc:(~mult1_acc+1):64'b0;
                              ready_o <= `DivResultReady;
                              if (start_i == `DivStop) begin
                                    state <= `DivFree;
                                    ready_o <= `DivResultNotReady;
                                    result_o <= {`ZeroWord, `ZeroWord};
                              end
                        End
```

运算结束时，根据运算类型进行输出，如果为除法，按照第一行输出，如果是乘法，按照第二行输出，赋值规则与乘法器相同。

3）乘除法器外部（mul_div.v）

```
assign sel=(inst_div||inst_divu)?2'b10:
             (inst_mult||inst_multu)?2'b01:
                2'b00;
 assign stallreq_for_ex=(stallreq_for_div||stallreq_for_mul)?1'b1:1'b0;
always @ (*) begin
            if (rst) begin
                  stallreq_for_div = `NoStop;
                  div_opdata1_o = `ZeroWord;
                  div_opdata2_o = `ZeroWord;
                  div_start_o = `DivStop;
                  signed_div_o = 1'b0;
            end
            else begin
                  stallreq_for_div = `NoStop;
                  div_opdata1_o = `ZeroWord;
                  div_opdata2_o = `ZeroWord;
                  div_start_o = `DivStop;
                  signed_div_o = 1'b0;
                  case ({(inst_div||inst_mult),(inst_divu||inst_multu)})
                        2'b10:begin
                              if (div_ready_i == `DivResultNotReady) begin
                                    div_opdata1_o = data1;
                                    div_opdata2_o = data2;
```

```
                div_start_o = `DivStart;
                signed_div_o = 1'b1;
                stallreq_for_div = `Stop;
            end
            else if (div_ready_i == `DivResultReady) begin
                div_opdata1_o = data1;
                div_opdata2_o = data2;
                div_start_o = `DivStop;
                signed_div_o = 1'b1;
                stallreq_for_div = `NoStop;
            end
            else begin
                div_opdata1_o = `ZeroWord;
                div_opdata2_o = `ZeroWord;
                div_start_o = `DivStop;
                signed_div_o = 1'b0;
                stallreq_for_div = `NoStop;
            end
        end
        2'b01:begin
            if (div_ready_i == `DivResultNotReady) begin
                div_opdata1_o = data1;
                div_opdata2_o = data2;
                div_start_o = `DivStart;
                signed_div_o = 1'b0;
                stallreq_for_div = `Stop;
            end
            else if (div_ready_i == `DivResultReady) begin
                div_opdata1_o = data1;
                div_opdata2_o = data2;
                div_start_o = `DivStop;
                signed_div_o = 1'b0;
                stallreq_for_div = `NoStop;
            end
            else begin
                div_opdata1_o = `ZeroWord;
                div_opdata2_o = `ZeroWord;
                div_start_o = `DivStop;
                signed_div_o = 1'b0;
                stallreq_for_div = `NoStop;
            end
        end
        default:begin
        end
```

```
            endcase
        end
    End
```

Sel 的赋值细节如上，ex 段的暂停由乘除法器控制（mul 在乘除法器合并后没有使用，但是可以起到标示作用，就没删除），时序逻辑未做太大改变，只是将判断条件改为 ：

case ({(inst_div||inst_mult),(inst_divu||inst_multu)})

5、 实验感受及改进意见

感受：整个实验过程的感受，可以用“过山车”来形容。在过 pass point1 前，应该说是最艰难的时期，面对刚接手的实验和海量的代码，实在无从下手，很多变量的意义不清楚、线路的连接思路很乱，但是在助教的耐心指导以及和同学们合作探讨下，总算是克服了困难。point1 后，在 point43 之前，都很简单，只是加入了一些常规指令，只用了一个下午的时间就完成了。但在 43 之后，加入了乘除法器和 hilo 寄存器，又面临了一定的困难：遇到了新的数据相关和数据选择、接线难题，而在 43 点到 64 点之间，难度又降低很多，不久就完成了。综上，我们经历了“**难-易-难-易**”的实验过程，虽然艰辛，但在最后完成的时候心中有着无限的成就感与喜悦感。最后我们还挑战了乘法器和乘除法合并，这里要特别感谢助教老师，在周末给予我们耐心的指导，才让我们做得很顺利，最终实现了乘法器的制作、乘除法器的合并。

改进意见:我们建议在初期多加入一些注释和引导，多数同学没有使用 vivado 的经历，初期一些简单的了解也不足以理解大段的代码，而且结合做完之后的感受，万事开头难，如果前面引导得好，学生们的感受能好很多，也能学到更多知识。

6、 参考材料

1）A03_“系统能力培养大赛”MIPS 指令系统规范_v1.01

2）自己动手做 cpu_雷思磊

3）手把手教你学 FPGA 设计：基于大道至简的至简设计法

4）https://github.com/fluclight001/SampleCPU