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：张翔宇

组员：方世博 陈伟嘉

报告日期：2021.12.18

一、 总体介绍

1.1个人分工

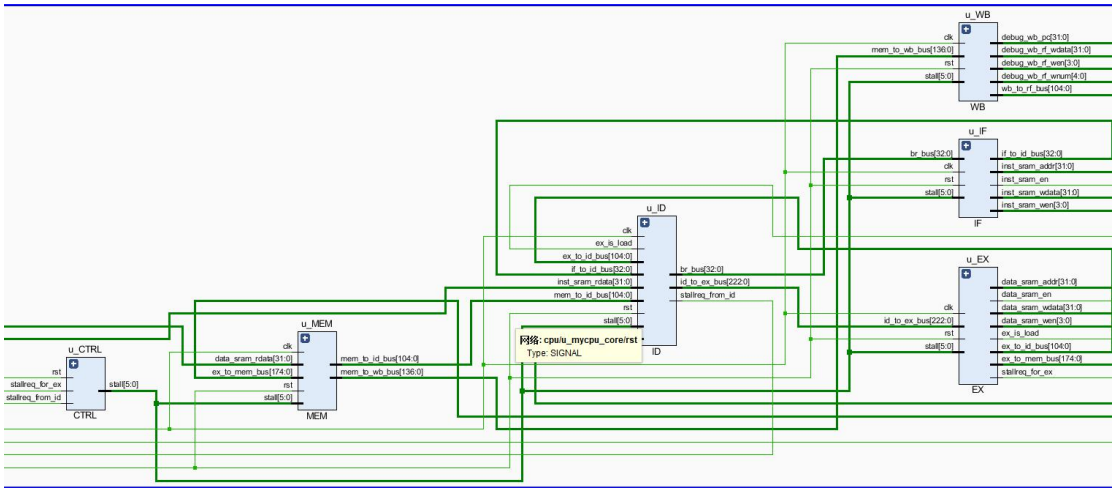
张翔宇：解决数据相关，乘除法器。

方世博：添加指令，load、store 指令的添加。乘除法器。

陈伟嘉：添加指令，跳转。

1.2 总体设计

1.2.1 连线图



不同流水段之间的连线图

共通过 64 点

1.2.2 程序运行环境及工具

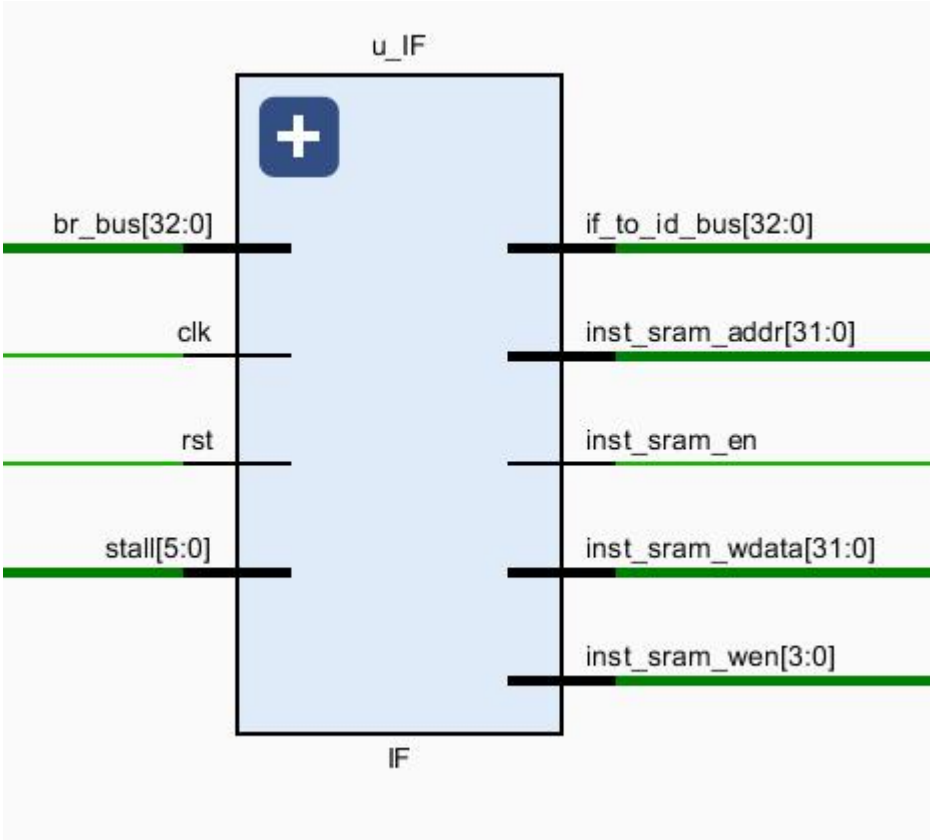
Vivado 设计套件，是 FPGA 厂商赛灵思公司 2012 年发布的集成设计环境。包括高度集成的设计环境和新一代从系统到 IC 级的工具，这些均建立在共享的可扩展数据模型和通用调试环境基础上。这也是一个基于 AMBA AXI4 互联规范、IP-XACT IP 封装元数据、工具命令语言(TCL)、Synopsys 系统约束(SDC) 以及其它有助于根据客户需求量身定制设计流程并符合业界标准的开放式环境。赛灵思构建的的 Vivado 工具把各类可编程技术结合在一起，能够扩展多达 1 亿个等效 ASIC 门的设计。

二、单个流水段说明（IF）

2.1 整体功能说明

IF 段主要完成取指操作，同时完成跳转地址。

2.2 端口介绍



br_bus:从 ID 段传过来的跳转指令的使能信号及跳转地址。

stall：暂停，从 CTRL 段传入插入气泡的信息。

if_to_id_bus:从 if 段传入 id 段的连线。

inst_sram_addr:指令的地址。

inst_sram_en:指令的读信号。

inst_sram_wen:指令的写信号。

inst_sram_wadta:写入指令的数据。

2.3 信号介绍

```
reg [31:0] pc_reg;
reg ce_reg;
wire [31:0] next_pc;
wire br_e;
wire [31:0] br_addr;
```

pc_reg:指令的 pc 地址。

ce_reg:指令的读信号。

br_e:跳转信号。

br_addr:跳转地址

2.4 包含的功能模块说明

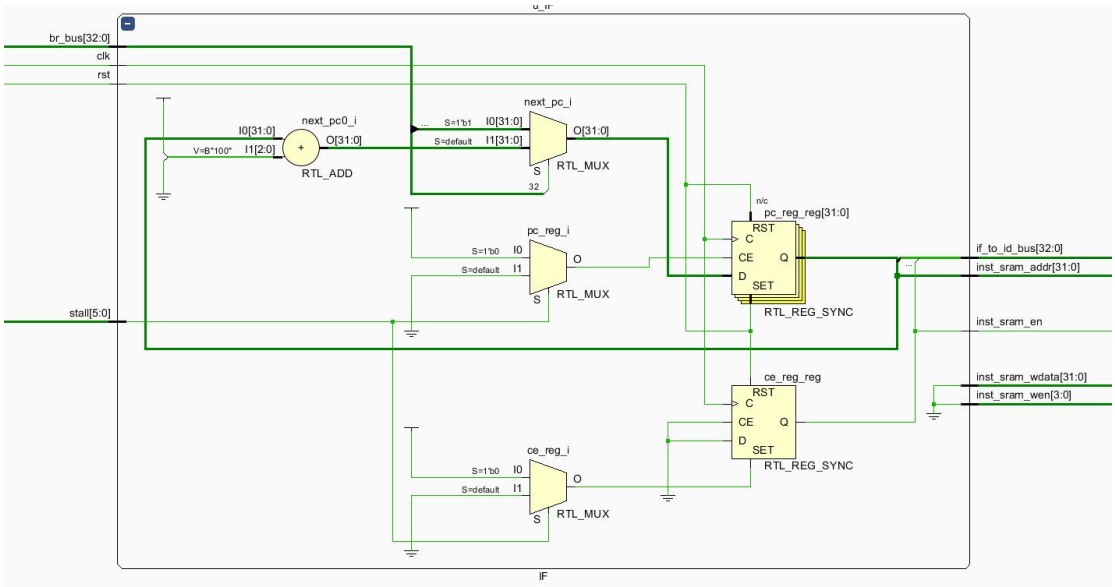
```
assign next_pc = br_e ? br_addr
                 : pc_reg + 32'h4;
```

判断下一步地址是 next_pc 还是跳转地址。如果 br_e 值为 0，说明不是跳转指令。那么 next_pc 的值就为 pc+4。如果 br_e 值为 1，说明是跳转指令。那么 next_pc 的值就为跳转的地址 br_addr。

```
always @ (posedge clk) begin
    if (rst) begin
        ce_reg <= 1'b0;
    end
    else if (stall[0]==`NoStop) begin
        ce_reg <= 1'b1;
    end
end
```

通过 stall 来实现 IF 段的暂停。

2.5 结构示意图

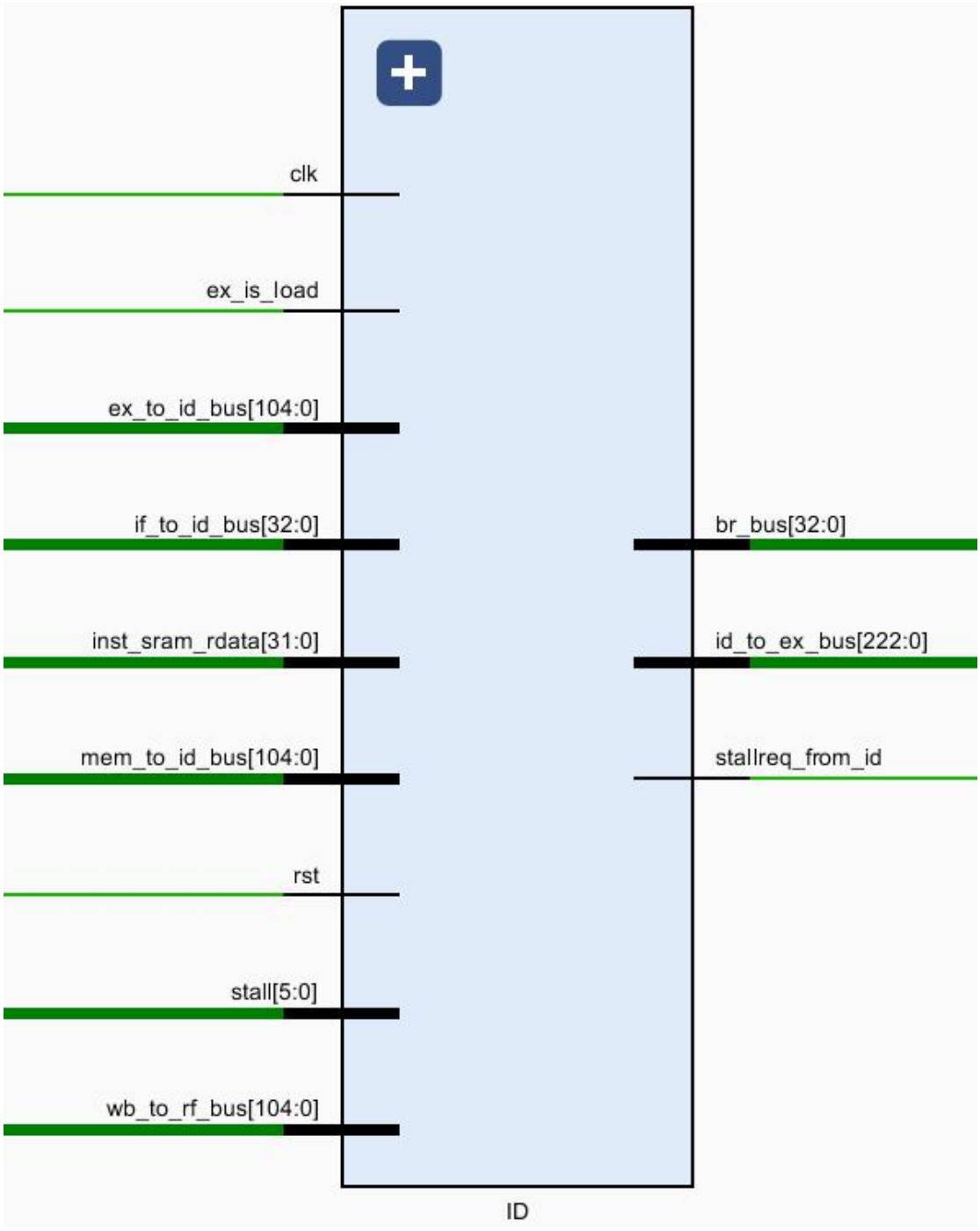


三、 单段流水说明（ID）

3.1 整体功能说明

完成对指令的判断及相应信号的赋值。或跳转地址的计算。

3.2 端口介绍



Ex_is_load:判断 ex 段执行的指令是否是 load 类指令。

If_ti_id_bus:if 段传过来的连线。

Ex_to_id_bus：ex 传过来的连线，用来解决数据相关。

Mem_to_id_bus:mem 段传过来的连线，用来解决数据相关。

Wb_to_rf_bus:wb 段传过来的连线，往 reg_file 中写入数据。

Stall：暂停信息。

stallreq_from_id：传入 ctrl 段的暂停信息。

Br_bus:跳转信息。

3.3 信号介绍

```
wire [31:0] inst;
wire [31:0] id_pc;
```

Inst：32 位指令。

id_pc：id 段目前的 pc 值。

```
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];
```

```
wire [5:0] opcode;
wire [4:0] rs,rt,rd,sa;
wire [5:0] func;
wire [15:0] imm;
wire [25:0] instr_index;
wire [19:0] code;
wire [4:0] base;
wire [15:0] offset;
wire [2:0] sel;
```

32 位 inst 指令的各个分段。

```
reg[31:0] HI;
reg[31:0] LO;
wire hi_en,hi_wen,lo_en,lo_wen;
```

HI 寄存器，LO 寄存器。以及 Hi，LO 读写使能信号。

```
assign inst_ori     = op_d[6'b00_1101];
assign inst_lui     = op_d[6'b00_1111];
assign inst_addiu   = op_d[6'b00_1001];
assign inst_beq     = op_d[6'b00_0100];
```

指令集

```
wire [2:0] sel_alu_src1;
wire [3:0] sel_alu_src2;
wire [11:0] alu_op;
```

Sel_alu_src1： rs to reg1， pc to reg1， sa_zero_extend to reg1.

Sel_alu_src2: rt to reg2, mm sign extend to reg2, 2'b8 to reg2, imm_zero_extend to reg2.

alu_op ： 指令集

```
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  )
);
```

Regfile 寄存器的连线接口。

```
wire br_e;
wire [31:0] br_addr;
wire rs_eq_rt;
wire rs_not_eq_rt;
```

跳转指令的信号。Br_e:是否跳转。 Br_addr:跳转地址。Rs_eq_rt:rs 寄存器的值是否等于 rt 寄存器的值。

3.4 包含的功能模块说明

```
always @ (posedge clk) begin
    if (rst) begin
        if_to_id_bus_r <= `IF_TO_ID_WD'b0;
    end
    else if (stall[1]==`Stop && stall[2]==`NoStop) begin
        if_to_id_bus_r <= `IF_TO_ID_WD'b0;
    end
    else if (stall[1]==`NoStop) begin
        if_to_id_bus_r <= if_to_id_bus;
    end
end
```

暂停的实现。

```
assign rdata11 = (ex_id_we & (ex_id_waddr==rs))? ex_id_wdata: ((mem_id_we &(mem_id_waddr==rs)) ? mem_id_wdata:
assign rdata22 = (ex_id_we & (ex_id_waddr==rt))? ex_id_wdata: ((mem_id_we &(mem_id_waddr==rt)) ? mem_id_wdata:
```

这部分是解决数据相关的部分，通过 ex，mem，wb 段传回来的数据来判断是否产生了数据相关。若要写回的寄存器与 id 段所要用到的寄存器相同，则产生了数据相关。此时应该先判断是否可以用 ex 段传回的数据，若不可以则需要检查是否可以用 mem 段传回的数据，最后再用 wb 段传回的数据。

```
assign HI2= ex_id_div_flag ? ex_id_div_result[63:32] : mem_id_div_flag ? mem_id_div_result[63:32] : wb_id_div_flag ? wb_id_div_result[63:32]
: ex_id_mt_flag[1] ? ex_id_wdata : mem_id_mt_flag[1] ? mem_id_wdata : HI;
assign LO2= ex_id_div_flag ? ex_id_div_result[31:0] : mem_id_div_flag ? mem_id_div_result[31:0] : wb_id_div_flag ? wb_id_div_result[31:0]
: ex_id_mt_flag[0] ? ex_id_wdata : mem_id_mt_flag[0] ? mem_id_wdata : LO;
```

这部分解决的是与 hi, lo 寄存器相关的指令的数据冲突。基本原理与上面的一样。

```
assign stallreq_from_id = (ex_is_load & ex_id_waddr==rs ) | (ex_is_load & ex_id_waddr==rt ) ;
```

判断上一段指令是否为 load 类指令，若上一条指令是 load 类指令，则 id 段需要暂定，此时将 stallreq_from_id 赋值为一并传向 ctrl 段。

```
always @ (posedge clk) begin
    if (stall[2]==`Stop & stall[3]==`NoStop) begin
        flag <= 1'b1;
        inst_reg <= inst_sram_rdata;
    end
    else begin
        flag <= 1'b0;
        inst_reg <= 32'b0;
    end
end
always @ (posedge clk) begin
    if(stall[3]==1'b1 & stall[4]==1'b0 & inst_reg2 == 32'b0) begin
        flag2 <= 1'b1;
        inst_reg2 <= inst_sram_rdata;
    end
    else if(stall[3]==1'b0) begin
        flag2 <=1'b0;
        inst_reg2 <= 32'b0;
    end
end
assign inst = flag ? inst_reg : flag2 ? inst_reg2 : inst_sram_rdata ;
```

对暂停时的 pc 和指令进行保存，在暂停结束时重新赋值回去。若是 id 段的暂停，则使 if，id 段暂停一个周期即可。若是 ex 段的暂停，则 if，id，ex 段暂停 32 个周期。

```
always @ (posedge clk) begin
    if(wb_id_div_flag == 1'b1) begin
        LO <= wb_id_div_result[31:0];
        HI <= wb_id_div_result[63:32];
    end
    else if (wb_id_mt_flag[0] == 1'b1) begin
        LO <=  wb_rf_wdata;
    end
    else if (wb_id_mt_flag[1] == 1'b1) begin
        HI <=  wb_rf_wdata;
    end
```

通过 wb 段传回的信息对 HI,LO 寄存器进行赋值。

```
assign br_addr = inst_beq ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :
inst_jal ? ({pc_plus_4[31:28], inst[25:0],2'b0}):
inst_jr ? rdata11:
inst_bne ? (pc_plus_4+{{14{inst[15]}},{inst[15:0],2'b00}}):
inst_j ? ( {pc_plus_4[31:28], instr_index, 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?(rdata11):
 32'b0;
```

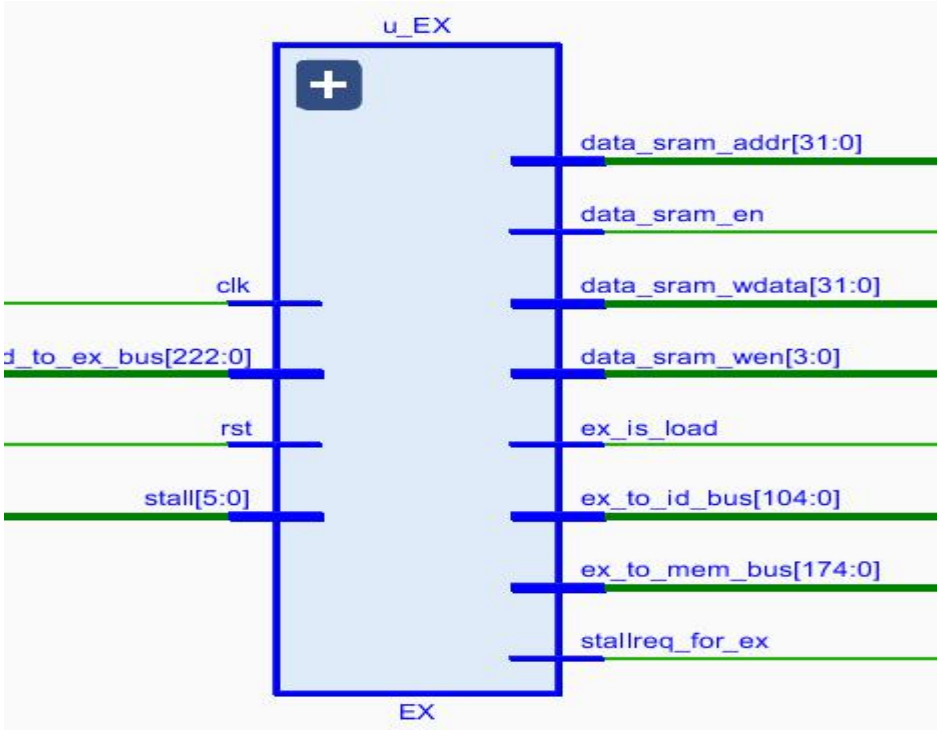
跳转地址的赋值。

四、 单段流水说明（EX）

4.1 整体功能说明

进行值的计算，或内存地址的计算。
乘除法器的实现。

4.2 端口说明



id_to_ex_bus:id 段传入 ex 段的连线。
stall:插入的气泡
data_sram_addr:内存地址。
data_sram_en:内存读信号。
data_sram_wdata:写入内存数据。
data_sram_wen:内存写信号。
ex_is_load:判断当前 ex 段是否为 load 类指令。
ex_to_id_bus:ex 传向 id 段的连线。
ex_to_mem_bus:ex 段传向 mem 段的连线。
stallreq_for_ex:ex 段传向 ctrl 段的气泡信号。

4.3 信号说明

```
assign ex_is_load = (inst[31:26] == 6'b10_0011) ? 1'b1 : 1'b0;
```

判断 inst 的第 26 到 31 位是否为 100011, 若是 100011 则说明该指令是 load 指令。则将 ex_is_load 赋值为 1。然后传到 id 段后，id 会向 ctrl 段发出暂停信号。之后 ctrl 段会将暂停信息传向 if，id，ex 段。这些段将会暂停一周期。

```
wire [31:0] HI;
wire [31:0] LO;
assign {
    HI,
    LO,
```

在 ex 段定义 hi，lo 信号储存从 id 段传入的 hi，lo 数据。如果 ex 段需要这两个数据，那么就可以直接从这两个信号中取得从 id 段传向 ex 段的 hi 和 lo 寄存器中的值。

```
alu u_alu(
    .alu_control (alu_op ),
    .alu_src1    (alu_src1    ),
    .alu_src2    (alu_src2    ),
    .alu_result  (alu_result  )
);
```

Alu 的接口。alu_op 是该操作数的操作类型，src1 是操作数 1，src2 是操作数 2。Aluresult 是经过 alu 计算出来得到的值。

```
assign ex_result = (inst[31:26]==6'b000000 & inst[5:0]==6'b010010) ? LO
: (inst[31:26]==6'b000000 & inst[5:0]==6'b010000) ? HI : alu_result;
```

这部分是进行判断 ex_result 是取 lo 或 hi 或 aluresult 的值。

```
wire mul_flag;
wire [63:0] mul_result;
reg signed_mul_o; // 有符号乘法标记
reg [31:0]mul_opdata1_o;
reg [31:0]mul_opdata2_o;
reg mul_start_o;
wire mul_ready_i;
reg stallreq_for_mul;
assign inst_mult = (inst[31:26]==6'b000000 & inst[5:0]==6'b011000) ? 1'b1 : 1'b0;
assign inst_multu = (inst[31:26]==6'b000000 & inst[5:0]==6'b011001) ? 1'b1 : 1'b0;
assign mul_flag = inst_mult | inst_multu ;
```

这一部分是实现乘法的信号，mul_flag 是乘法信号，如果 inst_mult 或 inst_multu 的值为 1 那么 mul_flag 的值也为一。

Mul_result 是乘法器计算出的值。

Signed_mul_o 是有符号乘法标记。

Mul_opdata1_o 是乘法的操作数一。

Mul_opdata2_o 是乘法的操作数二。

Mul_start_o 是乘法器开始信号。

Mul_ready_i 是乘法器完成计算信号。

Stallreq_for_mul 是乘法器的暂停信号。因为完成一个乘法工作需要 32 个周期，所以需要在进行乘法操作 if，id，ex 段暂停 32 个周期。

```
mymul u_mul(
    .clk          (clk           ),
    .rst          (rst           ),
    .signed_mul_i (signed_mul_o     ),
    .opdata1_i        (mul_opdata1_o      ), // 乘法源操作数1
    .opdata2_i        (mul_opdata2_o     ), // 乘法源操作数2
    .result_o     (mul_result     ), // 乘法结果 64bit
    .start_i     (mul_start_o),
    .annul_i        (1'b0),
    .ready_o        (mul_ready_i)
```

这是自制乘法器的接口

```
wire div_flag;
wire [63:0] div_result;
wire inst_div, inst_divu;
wire div_ready_i;
reg stallreq_for_div;
assign stallreq_for_ex = stallreq_for_div | stallreq_for_mul;

reg [31:0] div_opdata1_o;
reg [31:0] div_opdata2_o;
reg div_start_o;
reg signed_div_o;
```

这是除法器的接口。
Div_flag 是除法指令的信号。
Div_ready_i 是除法器完成除法计算工作的信号。
Stallreq_for_div 是除法器的暂停信号，因为完成一个除法工作需要 32 个周期，所以需要在进行除法操作 if，id，ex 段暂停 32 个周期。

```
wire flag;
wire[1:0] mt_flag;
wire [63:0] result;
wire [31:0] ex_result2;
assign mt_flag[0] = (inst[31:26] == 6'b000000 & inst[5:0] == 6'b010011) ? 1'b1 : 1'b0; //lo
assign mt_flag[1] = (inst[31:26] == 6'b000000 & inst[5:0] == 6'b010001) ? 1'b1 : 1'b0;  //hi
assign flag = mul_flag | div_flag ;
assign result = mul_flag ? mul_result : div_flag ? div_result : 64'b0 ;
assign ex_result2 = ( mt_flag[0] | mt_flag[1] ) ? rf_rdata1 : ex_result;
```

这一部分是完成乘法和除法指令的传递信号的工作，以及 mt 类指令的信号的赋值。如果是乘法运算或除法运算那么 flag 将被赋值为 1。如果是乘法运算，那么 result 中将被赋值 mul_result。如果是除法运算那么 result 将被赋值为 div_result。然后一起传向 mem 段。

4.4 主要功能

```
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;
```

在 ex 段通过选择从 id 段传入 ex 段的信号来选择操作和操作数。其中 sel_alu_src1[0]==1 就将 ex 段当前的 pc 值传给操作数一。sel_alu_src1[0]==0，并且 sel_alu_src1[1]==1 就将 sa 零扩展的值传给操作数一。如果 sel_alu_src1[0]==0 并且 sel_alu_src1[1]==0，就把从 regfile 中取出的数赋值给操作数一。

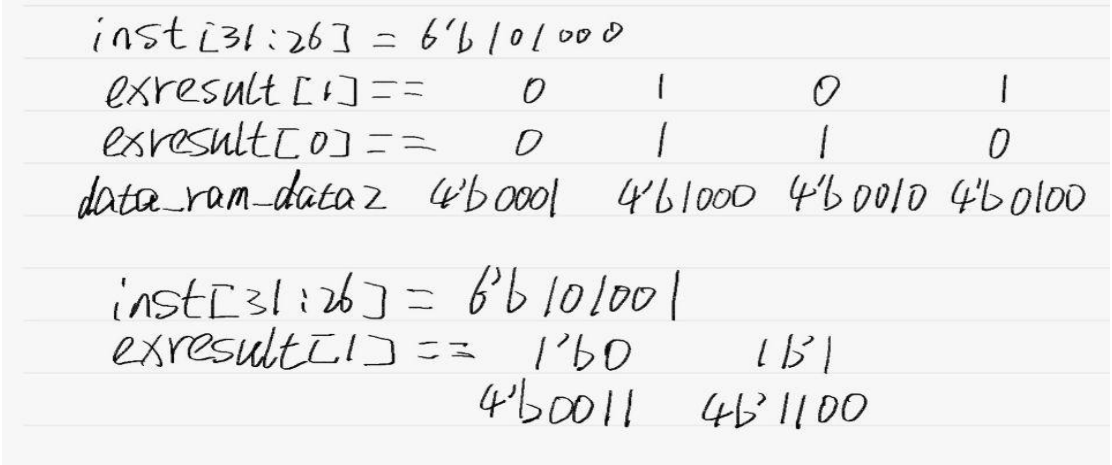
sel_alu_src2[0]==1 就将立即数符号扩展的值传给操作数二，如果 sel_alu_src2[0]==0 并且 sel_alu_src2[1]==1 就将 32’b8 传给操作数二。如果 sel_alu_src2[0]==0，sel_alu_src2[1]==0，sel_alu_src2[2]==1 就将立即数零扩展的值传给操作数二。如果 sel_alu_src2[0]==0，sel_alu_src2[1]==0，sel_alu_src2[2]==0 就将从 regfile 中取出的值传给操作数二。

```
wire [31:0] rf_rdata22;
assign rf_rdata22 = inst[31:26]==6'b101000 ? {4{rf_rdata2[7:0]}}
: inst[31:26]==6'b101001 ? {2{rf_rdata2[15:0]}}
: rf_rdata2;
```

这里实现的是如果 inst 的 op 为 101000 那么就将 rf_rdata2[7:0]的值扩展为 32 位赋值给 rf_rdata22。如果 inst 的 op 为 101001，那么就将 rf_rdata2[15:0]的值扩展为 32 位赋值给 rf_rdata22。如果 inst 的 op 不是 101000 或 101001，那么就赋值给它原来的值。

```
assign data_ram_wen2 = (inst[31:26]==6'b101000) ? ( (ex_result[1]==1'b0 & ex_result[0]==1'b0) ? 4'b0001
: (ex_result[0]==1'b1 & ex_result[1]==1'b0) ? 4'b0010
: (ex_result[0]==1'b0 & ex_result[1]==1'b1) ? 4'b0100
: (ex_result[0]==1'b1 & ex_result[1]==1'b1) ? 4'b1000
: data_ram_wen )
: (inst[31:26]==6'b101001) ? ( ex_result[1]==1'b0 ? 4'b0011
: ex_result[1]==1'b1 ? 4'b1100
: data_ram_wen )
: data_ram_wen;
```

这一部分的的功能是根据 ex_result 的值来给 data_ram_wen2 赋值。



```
assign data_sram_en = data_ram_en;
assign data_sram_wen = data_ram_wen2;
assign data_sram_addr = ex_result;
assign data_sram_wdata = rf_rdata22;
```

这一部分是与内存交互的提地方，如果 data_ram_wen 的值为一，那么就往 ex 段算出来的值作为地址传入 rf_rdata22 的值。

```
always @ (*) begin
        if (rst) begin
            stallreq_for_mul = `NoStop;
            mul_opdata1_o = `ZeroWord;
            mul_opdata2_o = `ZeroWord;
            mul_start_o = `MulStop;
            signed_mul_o = 1'b0;
        end
        else begin
            stallreq_for_mul = `NoStop;
            mul_opdata1_o = `ZeroWord;
            mul_opdata2_o = `ZeroWord;
            mul_start_o = `MulStop;
            signed_mul_o = 1'b0;
            case ({inst_mult,inst_multu})
                2'b10:begin
                    if (mul_ready_i == `MulResultNotReady) begin
                        mul_opdata1_o = rf_rdata1;
                        mul_opdata2_o = rf_rdata2;
                        mul_start_o = `MulStart;
                        signed_mul_o = 1'b1;
                        stallreq_for_mul = `Stop;
                    end
                    else if (mul_ready_i == `MulResultReady) begin
                        mul_opdata1_o = rf_rdata1;
                        mul_opdata2_o = rf_rdata2;
                        mul_start_o = `MulStop;
                        signed_mul_o = 1'b1;
                        stallreq_for_mul = `NoStop;
                    end
                    else begin
                        mul_opdata1_o = `ZeroWord;
                        mul_opdata2_o = `ZeroWord;
```

```
                    mul_start_o = `MulStop;
                    signed_mul_o = 1'b0;
                    stallreq_for_mul = `NoStop;
                end
            end
            2'b01:begin
                if (mul_ready_i == `MulResultNotReady) begin
                    mul_opdata1_o = rf_rdata1;
                    mul_opdata2_o = rf_rdata2;
                    mul_start_o = `MulStart;
                    signed_mul_o = 1'b0;
                    stallreq_for_mul = `Stop;
                end
                else if (mul_ready_i == `MulResultReady) begin
                    mul_opdata1_o = rf_rdata1;
                    mul_opdata2_o = rf_rdata2;
                    mul_start_o = `MulStop;
                    signed_mul_o = 1'b0;
                    stallreq_for_mul = `NoStop;
                end
                else begin
                    mul_opdata1_o = `ZeroWord;
                    mul_opdata2_o = `ZeroWord;
                    mul_start_o = `MulStop;
                    signed_mul_o = 1'b0;
                    stallreq_for_mul = `NoStop;
                end
            end
            default:begin
            end
        endcase
    end
end
```

这是自制乘法器的工作原理，根据不同的状态值来给各个信号附上初始值。

```
`MulFree: begin
    if (start_i == `MulStart && annul_i == 1'b0) begin
        if (opdata1_i == `ZeroWord || opdata2_i == `ZeroWord) begin   //任何操作数为0，都进入MUL_BY_ZERO状态
            state <= `MulByZero;
        end
        else begin
            state <= `MulOn;
            cnt <= 6'b000000;
            if (signed_mul_i == 1'b1 && opdata1_i[31] == 1'b1) begin    //op1为负数，取补码
                temp_op1 = ~opdata1_i + 1;
            end
            else begin
                temp_op1 = opdata1_i;
            end
            if (signed_mul_i == 1'b1 && opdata2_i[31] == 1'b1) begin    //op2为负数，取补码
                temp_op2 = ~opdata2_i + 1;
            end
            else begin
                temp_op2 = opdata2_i;
            end
            multiplicand <= {32'b0, temp_op1};
            multiplier <= temp_op2;
            product_temp <= {`ZeroWord, `ZeroWord};
        end
```

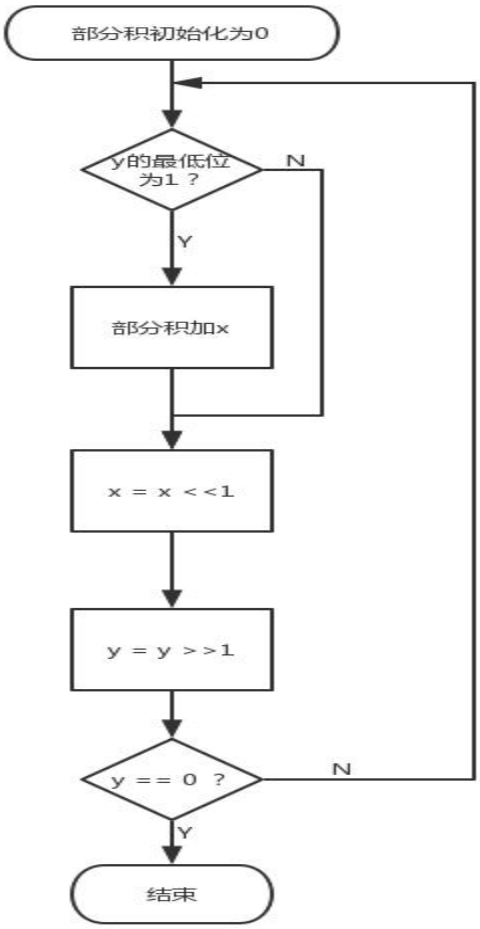
这是其中 mulfree 状态的代码。

z = x * y 中，x 是被乘数，在 Verilog 代码中 multiplicand 表示，y 是乘数，在代码中用 multiplier 表示。因为 x 和 y 都是带符号数，所以应该是用补码乘法，但是如果对 x 和 y 求绝对值，让两个绝对值相乘，然后再判断正负，效果和补码乘法是相同。后面给出的 Verilog 代码就是基于这种思路编写的。两个 32 位整数相乘，实际上是进行了 32 次加法操作。下面以两个 4 位二进制数相乘来说明乘法实现的过程。

```
Multiplicand      1000
Multiplier      x 1001
                  1000
                 0000
                0000
               1000
Product(积)  0.1001000
```

从上图中可以看到，被乘数 x 为 1000，乘数 y 为 1001，上面的乘法过程是手工运算的一个步骤，而计算机在做乘法时就是模拟上述手工运算的执行过程。因为是两个 4 位数相乘，所以结果应该是四个数加和得到的。先判断 y 的最低位是 0 还是 1，如果是 1，则需要把 x 加到部分积上，若为 0，则需要把 0 加到部分积上 (实际上加 0 的这个过程计算机并不执行，因为加 0 对部分积没有任何影响)，x 左移一位，之后再让 y 右移一位，若 y 为 0，则循环结束，否则继续此循环过

程。流程图如下。



流程图中，x 因为需要左移，所以 32 位长度的 x 应该用一个 64 位寄存器来存储，这样才能保证 x 左移后不会发生高位丧失。

程序仿真开始时，文件会对输入信号进行初始化。使得 mult_begin 为 1，并且给出两个操作数 mult_op1 和 mult_op2 分别作为乘数和被乘数。对 mult_op1 和 mult_op2 进行分解，分解出他们的符号和绝对值，后面的运算是让 mult_op1 和 mult_op2 的绝对值进行运算，相当于是两个无符号数的乘法。当乘法信号有效后，也就是说乘法开始之后，把 x 的绝对值赋值给一个 64 位的 reg 型变量 multiplicand，把 y 的绝对值赋值给一个 32 位 reg 型变量 multiplier，根绝 multiplier 最低位是 0 还是 1，决定着 64 位 wire 型变量 partial_product 赋值 0 还还是赋值 multiplicand。临时结果 product_temp 加上部分积之后再把加的结果赋值给自己，根据 mult_op1 和 mult_op2 的符号计算乘积结果的符号。最终的乘积结果（product）是 wire 型变量，用 assign 赋值，每当临时结果（product_temp）发生改变时，product 也立即发生变化。

```
div u_div(
    .rst          (rst          ),
    .clk          (clk          ),
    .signed_div_i (signed_div_o ),
    .opdata1_i    (div_opdata1_o    ),
    .opdata2_i    (div_opdata2_o    ),
    .start_i      (div_start_o      ),
    .annul_i      (1'b0       ),
    .result_o     (div_result     ),
    .ready_o      (div_ready_i      )
```

这是除法器的接口。

除法器的工作原理与乘法器相像

```
`DivFree: begin            //除法器空闲
    if (start_i == `DivStart && annul_i == 1'b0) begin
        if(opdata2_i == `ZeroWord) 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
                temp_op1 = ~opdata1_i + 1;
            end else begin
                temp_op1 = opdata1_i;
            end
            if (signed_div_i == 1'b1 && opdata2_i[31] == 1'b1 ) begin
                temp_op2 = ~opdata2_i + 1;
            end else begin
                temp_op2 = opdata2_i;
            end
            dividend <= {`ZeroWord, `ZeroWord};
            dividend[32: 1] <= temp_op1;
            divisor <= temp_op2;
        end
```

这是除法器在 divfree 状态的代码。

```
assign ex_to_mem_bus = {
    mt_flag,      //2
    flag,
    result,
    inst,
    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_result2       // 31:0
```

这是 ex 段传向 mem 段的总线。

```
assign ex_to_id_bus={
    mt_flag,
    flag,
    result,
    rf_we,
    rf_waddr,
    ex_result2
};
```

这是 ex 段传向 id 段的解决数据相关的定向路径。

五、 单段流水说明 (MEM)

5.1 整体功能说明

访存的实现

5.2 端口说明

```
input wire clk,
input wire rst,
input wire [`StallBus-1:0] stall,
input wire [`EX_TO_MEM_WD-1+64+1+2+32:0] ex_to_mem_bus,
input wire [31:0] data_sram_rdata,
output wire [`MEM_TO_WB_WD-1+64+1+2:0] mem_to_wb_bus,
output wire [37+64+1+2:0] mem_to_id_bus
```

Stall 是暂停气泡。

Ex_to_mem_bus 是 ex 段传向 mem 段的数据通路。

Data_sram_radata 是从内存中读取的数据。

Mem_to_wb_bus 是从 mem 段传向 wb 段的数据通路。

Mem_to_id_bus 是从 mem 段传向 id 段的定向路径。

5.3 信号说明

```
always @ (posedge clk) begin
    if (rst) begin
        ex_to_mem_bus_r <= `EX_TO_MEM_WD+64+1+2+32'b0;
    end
    else if (stall[3]==`Stop && stall[4]==`NoStop) begin
        ex_to_mem_bus_r <= `EX_TO_MEM_WD+64+1+2+32'b0;
    end
    else if (stall[3]==`NoStop) begin
        ex_to_mem_bus_r <= ex_to_mem_bus;
    end
end
```

这是在 mem 段实现的暂停功能。

```
assign {
    mt_flag,
    div_flag,          // 107
    div_result,        //106:76
    inst,
    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;
```

这是 ex 段传向 mem 段的数据通路。

```
assign mem_to_wb_bus = {
    mt_flag,
    div_flag,
    div_result,
    mem_pc,       // 69:38
    rf_we,        // 37
    rf_waddr,     // 36:32
    rf_wdata      // 31:0
};
```

这是 mem 段传向 wb 段的数据通路

```
assign mem_to_id_bus={
    mt_flag,
    div_flag,
    div_result,
    rf_we,
    rf_waddr,
    rf_wdata
};
```

这是 mem 段传向 id 段的定向路径。

5.4 主要功能说明

assign data_sram_rdata2 = (inst[31:26]==6'b100000) ? ((ex_result[0] == 1'b0 & ex_result[1] == 1'b0) ? {{24{data_sram_rdata[7]}},data_sram_rdata[7:0]}

: (ex_result[0] == 1'b1 & ex_result[1] == 1'b0) ? {{24{data_sram_rdata[15]}},data_sram_rdata[15:8]}

: (ex_result[0] == 1'b0 & ex_result[1] == 1'b1) ? {{24{data_sram_rdata[23]}},data_sram_rdata[23:16]}

: (ex_result[0] == 1'b1 & ex_result[1] == 1'b1) ? {{24{data_sram_rdata[31]}},data_sram_rdata[31:24]}

: data_sram_rdata)

: (inst[31:26]==6'b100100) ? ((ex_result[0] == 1'b0 & ex_result[1] == 1'b0) ? {24'b0,data_sram_rdata[7:0]}

: (ex_result[0] == 1'b1 & ex_result[1] == 1'b0) ? {24'b0,data_sram_rdata[15:8]}

: (ex_result[0] == 1'b0 & ex_result[1] == 1'b1) ? {24'b0,data_sram_rdata[23:16]}

: (ex_result[0] == 1'b1 & ex_result[1] == 1'b1) ? {24'b0,data_sram_rdata[31:24]}

: data_sram_rdata)

: (inst[31:26]==6'b100001) ? (ex_result[1] == 1'b0 ? {{16{data_sram_rdata[15]}},data_sram_rdata[15:0]}

: (ex_result[1] == 1'b1) ? {{16{data_sram_rdata[31]}},data_sram_rdata[31:16]}

: data_sram_rdata)

: (inst[31:26]==6'b100101) ? (ex_result[1] == 1'b0 ? {16'b0,data_sram_rdata[15:0]}

: (ex_result[1] == 1'b1) ? {16'b0,data_sram_rdata[31:16]}

: data_sram_rdata)

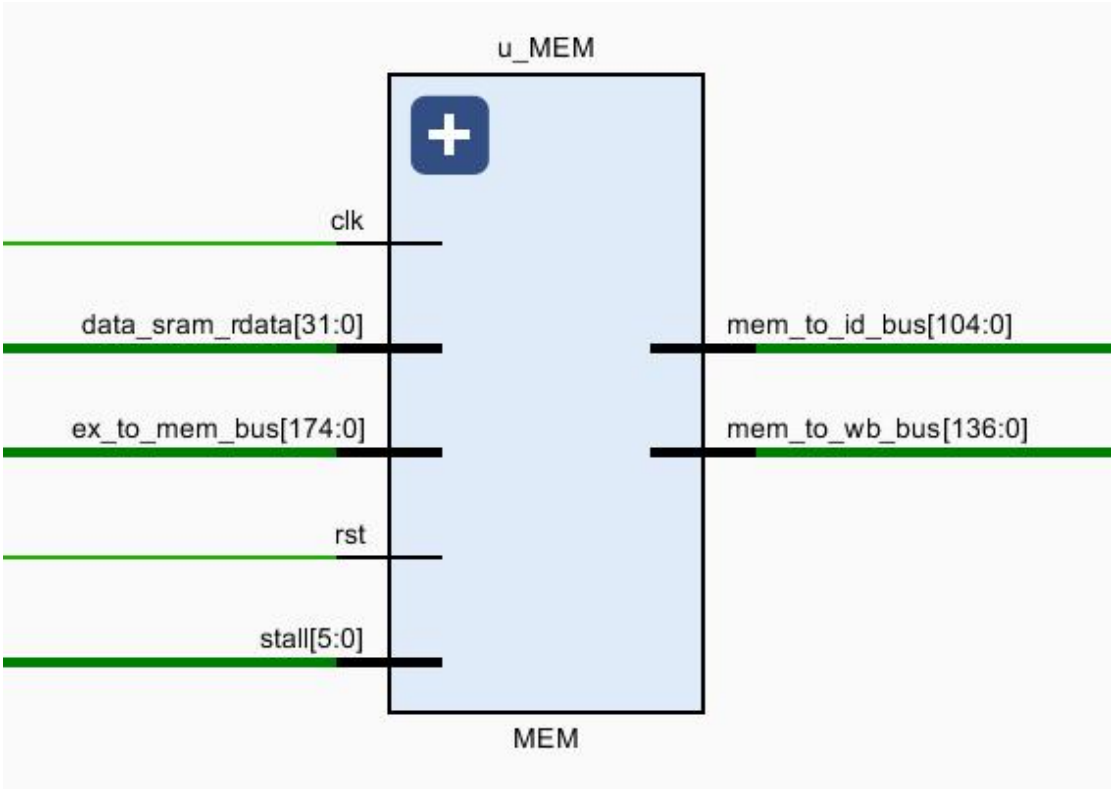
:data_sram_rdata;

这是实现一些指令的特殊要求，如根据 ex 段计算得到的结果 ex_result 的最后一位来对从内存中取到的数据进行处理。

assign rf_wdata = (data_ram_wen==4'b0000 & data_ram_en==1'b1) ? data_sram_rdata2 : sel_rf_res ? mem_result : ex_result;

这是根据 data_ram_wen 和 data_ram_en 的值来判断 rf_wdata 是取什么值。

5.5 总体架构



六、 单段流水说明（wb)

6.1 整体功能说明

数据的写回

6.2 端口说明

```
input wire clk,
input wire rst,
input wire [`StallBus-1:0] stall,
input wire [`MEM_TO_WB_WD-1+64+1+2:0] mem_to_wb_bus,
output wire [`WB_TO_RF_WD-1+64+1+2:0] wb_to_rf_bus,
output wire [31:0] debug_wb_pc,
output wire [3:0] debug_wb_rf_wen,
output wire [4:0] debug_wb_rf_wnum,
output wire [31:0] debug_wb_rf_wdata
```

Stall:wb 段的气泡暂停信号。
Mem_to_wb_bus:mem 段到 wb 段的数据通路。
Wb_to_rf_bus:wb 到 id 段的数据通路。

6.3 信号说明

```
wire [31:0] wb_pc;
wire rf_we;
wire [4:0] rf_waddr;
wire [31:0] rf_wdata;
wire div_flag;
wire [63:0] div_result;
wire [1:0] mt_flag;
```

Wb_pc:是当前 wb 段 pc 的值。
Rf_we:是是否存入寄存器的信号。
Rf_waddr:是写入寄存器的地址。
Rf_wdata:是写入寄存器的数据。
Div_flag:是除法器信号。
Div_result:是除法或乘法结果。
Mt_flag:是 mt 类指令的信号。

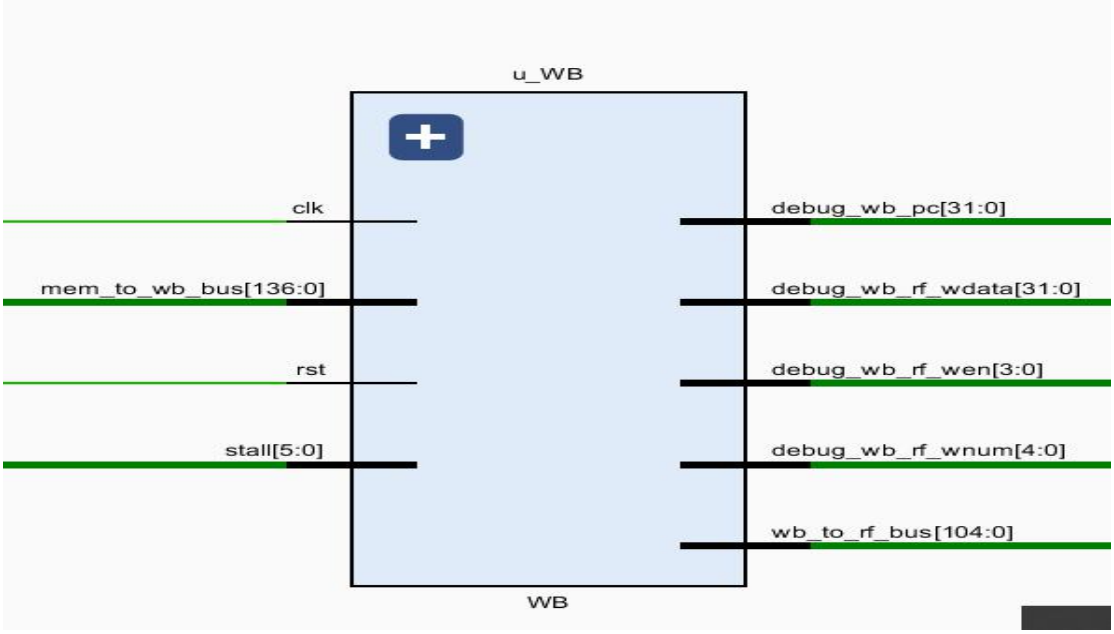
```
assign {
    mt_flag,
    div_flag,
    div_result,
    wb_pc,
    rf_we,
    rf_waddr,
    rf_wdata
} = mem_to_wb_bus_r;
```

这是 mem 段传入 wb 段的数据通路。

```
assign wb_to_rf_bus = {
    mt_flag,
    div_flag,
    div_result,
    rf_we,          //37
    rf_waddr,       //36:32
    rf_wdata        //31:0
};
```

这是 wb 段传入 id 段的数据通路。

6.4 总体架构



七、 单段流水说明（ctrl）

6.1 整体功能说明

暂停气泡的控制和实现

6.2 端口说明

```
input wire rst,
input wire stallreq_from_id,
input wire stallreq_for_ex,
output reg [`StallBus-1:0] stall
```

Stall:ctrl 段传出的暂停信号。

Stallreq_from_id:从 id 段传入的暂停信号。

Stallreq_from_ex:从 ex 段传入的暂停信号。

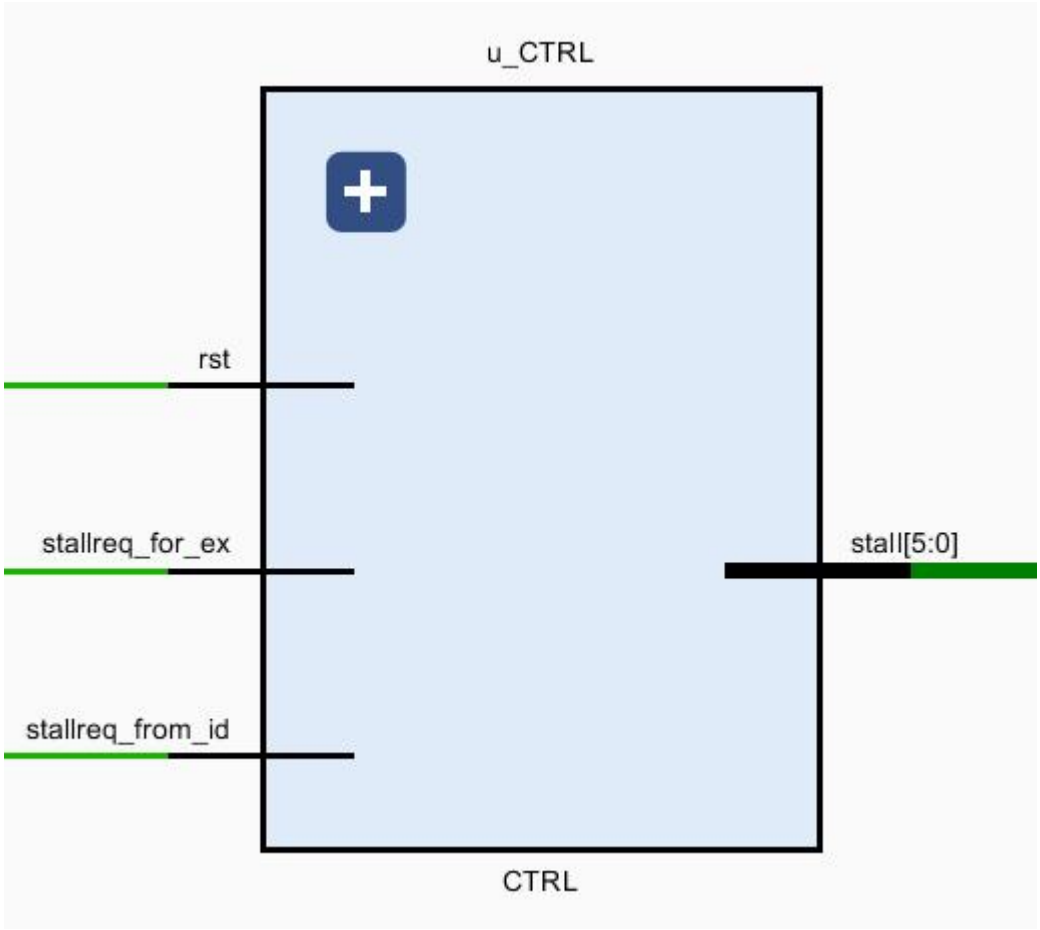
6.3 主要实现功能

```
always @ (*) begin
    if (rst) begin
        stall <= `StallBus'b0;
    end
    else if(stallreq_from_id == 1'b1) begin
        stall <=6'b000111;
    end
    else if(stallreq_for_ex == 1'b1) begin
        stall <=6'b001111 ;
    end
    else begin
        stall <=6'b000000;
    end
end
```

如果是来自 id 段的暂停，那么 if，id 段对应的 stall 为 1。

如果是来自 ex 段的暂停，那么 if，id，ex 段对应的 stall 为 1。

6.4 整体结构：



八、 实验感受和改进意见

8.1 实验感受

张翔宇：在我们的大学生涯中，有很多上理论知识课程的机会，但是实训的次数却是很少，因此这次的实训中提供了一个好的机会来实践自己的知识水平，我能够达到什么样的层次，也是看看自己这么久的学习生活到底有没有进步。在实训中助教讲了关于 vivado 的操作方法，还有如何安装软件，利用软件来达到我们的目的。在我们平时上课的过程中总是看到老师在讲台上面用多媒体讲解理论知识，当时感觉这样的操作其实挺简单的，尤其是对着书进行操作，感觉我能够做到很好，但是真的等我上手操作时，才感觉到自己在这方面的浅薄无知。在开始还是老师先讲述实验的内容是什么，我们应该在这样的实验中明白一些什么，以及应该要操作到什么程度，完成了之后会出现什么现象，都是我们在这样的实训中懂得的。当我打开电脑根本不知道要干些什么的时候，同学在我旁边翻书的身影映入眼帘，我们现在还是要先看书弄明白步骤才能知道要怎么操作才行，第一步是打开什么软件，第二部是在软件上面做些什么，当我磕磕绊绊的做完一系列的流程之后才发现这都是机械的操作，根本没有自己的想法在内，什么都是规规矩矩的，我还需要做点其他的东西。实训的时间过得很快，在平时我有很多的

想法，在老师讲课的过程中涌现出来，但是都没有自己操作的机会，在这次实训中我首先熟悉了 vivado 的常规化操作，随后明白实验的流程是什么，最后了解怎么让自己的创新性想法在电脑上实现，尤其是在完成一个试验之后，发现自己成功了之后，这种兴奋难以将其诉说出来。完成实训之后，我对于自己在理论知识水平有了一个大致的了解，虽然上课时觉得老师讲的很容易，但是真正轮到自己时才发现难度有多高，我在 vivado 上面的学习还是半瓶子水晃荡，没有实质的深入的了解，完成的任务和其他同学相比也就是中规中矩，没有什么太过出色的地方。经过这样的实训，我明白自己的修行还是不够，对书本的了解程度也只仅限于老师上课的内容，没有认真地研读和课后的训练导致自己基础不够扎实，在今后的学习时光中，我会在这上面下更多的功夫，不辜负自己的学生生活。

方世博：刚才开始接触逻辑设计很多人会觉得很简单：因为 verilog 的语法不多，半天就可以把书看完了。但是很快许多人就发现这个想法是错误的，他们经常埋怨综合器怎么和自己的想法差别这么大：它竟然连用 for 循环写的一个计数器都不认识！原因是做逻辑设计的思维和做软件的很不相同，我们需要从电路的角度去考虑问题。

在这个过程中首先要明白的是软件设计和逻辑设计的不同，并理解什么是硬件意识。

软件代码的执行是一个顺序的过程，编绎以后的机器码放在存储器里，等着 CPU 一条一条的取指并执行；因此软件设计中经常会带有顺序处理的思维。而逻辑设计则不同，我们设计的是数字电路，它是由很多很多的与非门及 D 触发器构成的，上电之后所有与非门和 D 触发器都同时工作，不会因为 A 触发器的代码描述在 B 触发器之前 A 触发器就是先工作，事实上，RTL 级代码的代码先后顺序在综合成网表文件后这种顺序就消失了，取代的是基本逻辑电路之间的互联关系描述；因此逻辑设计需要的是一种并发的思维，我们也需要用并发的思维去考虑电路的设计。

当然，我们设计的电路功能一般都有先后顺序的关系，如果这种顺序不能通过代码的先后顺序来实现，那么要怎么完成这一功能呢？在逻辑设计中，我们所说的先后顺序都是基于时间轴来实现：它的承载体就是时序逻辑，也就是那些触发器。其次就是要熟悉基本电路的设计。

基本的电路不是很多，也就是 D 触发器、计数器、移位寄存器、状态机、多路选择器、译码器等几种，所有复杂的电路都可由这些基本的电路构成。

我认为，在入门的时候，对于基本电路的设计应该固定化、标准化，每种电路该用什么样的代码描述，应该要固定、统一，尽量少一些花哨的东西。

代码规范主要是代码书写、命名等规范。比如不能用 TAB 键空格、低电平有效信号命名时加_n(如 rst_n 等)、每行只能写一行代码等。
设计时序是进行逻辑设计的基本要求：时序是设计出来的，不是仿出来的，更不是凑出来的。我们要先对整个设计有一些规划--时时刻刻都要有设计时序的思想。设计时序最重要的是做好方案，这里说的方案绝不是只是摆几个框图在那里。我们在做设计的时候需要做总体设计方案、逻辑详细设计方案。这两种方案包括了很多东西，逻辑总体方案主要是一级模块的划分及接口时序的定义，而逻辑详细方案就是代码的文字及图形描述!
对于入门来说，接触的比较多的是逻辑详细设计方案。在这一级别的方案中，我们是要求的是至少要做到模块内部所有关键信号的时序都要先设计好，这里讲的设计时序主要就是画波形图，在一个操作周期内每个信号在每一个时钟周期该是什么样子就画成什么样子。
时序设计好之后，模块内部各个信号之间的关系就理得差不多了，之后就是将它翻译成代码了，这个过程以体力劳动为主，在加约束之前，我们首先要定义一些术语好告诉 EDA 软件我们想干什么，这些术语便是 Fmax、Tsu、Tco 等等这些东西。
有了术语，还要有一种通信方式与 EDA 软件通信，脚本语言充当了这一角色。。在加了约束之后，EDA 工具就可以更好地按照我们的意愿去干活了。如果约束加的过高，就相当于让 EDA 工具去做一件不可能完成的事，找更短的路径的时候说不定找着找着就掉下悬崖了，效果反而更差。
加约束只能做一些锦上添花的事情。所以，我们在做方案的时候就需要对关键路径进行预估，要通过设计而不是约束解决这些问题。

陈伟嘉：计算机系统是一门知识内容很多的课程，除了书本知识的讲解，老师在我们这届开始了体量较大的实验内容，通过阅读参考资料，结合课内所学，我们需要实现 CPU 的流水。
在一段时间的研究过后，我们发现这是一项较为艰难的任务，除了任务本身的难度，要小组成员共同完成一项整体性较强的实验需要每个人对实验整体内容的掌握和其他人的沟通、配合以及理解。我们需要学习掌握一个新的软件——vivado 来实现 CPU 的构造，软件的安装、使用、调试都是入门的难题。相较于纸上谈兵，真正上手实践才会发现在实际问题中各种细节就会产生各种影响，一点小小的疏漏就会困扰小组几天，当我们几个星期都没能通过第一个点时，我们都对这个实验产生了些许畏惧。但通过与其他同学的学习交流以及对助教的咨询，我们一点一点实现了零的突破。当通过第一个点后，后续的内容叫稍显流畅一些，但不间断的仍有难点出现。经历了对参考资料的反复理解和书上所没有内容的自行

实践摸索，我们逐渐将实验内容完成，在真正完成任务后，我才惊讶地发现原来任何问题的实际操作和书上的理论相差甚远，同时，即使完成了任务，我仍感到自己水平有所欠缺。

就像老师所讲，实验的开展是为了从硬件角度让我们更好的理解软件知识，软硬件的结合会让我们对所学内容有更深刻的理解。通过小组的配合一起攻克难关，虽然过程艰辛，但是成功后的喜悦和收获还是令人振奋的。

8.2 改进意见

希望助教给的初始代码多加一些注释。