| 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 1902

组长：韩世龙

组员：刘璐　张有成

报告日期：2021.12.19

# 目录

# 一、概述

## （1）工作量分配：

共通过了 64 个测试点，并且额外完成一个 32 周期的移位乘法器

（1）刘璐—— 乘除法指令、HI 和 LO 寄存器的构建、访存指令以及基本指令的加，32 周期移位乘法器的编写

（2）韩世龙——数据相关、流水线暂停指令、跳转指令以及基本的添加

（3）张有成——跳转指令、访存指令、基本指令添加

## （2）总体设计：

CPU 主体结构分为六大部分，其中五部分为流水线的基本组成单元：IF（取值段），ID（译码段），EX（执行段），MEM（访存段），WB（写回段），另外一部分为 Ctrl（控制段）。

### IF 段

IF 段只负责根据是否有跳转给下一个 PC 赋值，将取指令信号变为真，并将 PC 的值作为地址发送给存储器，使存储器根据 PC 地址向 ID 段返回 32 位 inst 指令。同时将 PC 值传给 ID 段方便后面指令的使用；

### ID 段

ID 段的主要工作为指令的解析，寄存器的访存以及跳转指令的地址计算。ID 段从内存接收到指令数据流之后，将指令按位拆分并判断指令的具体类别。不同类别的指令会有读 rt 寄存器，写 rt 寄存器，立即数拓展等不同的操作。将不同操作的信号赋值给对应的线，线将信号进一步传入到 ID 内部的 regfile 寄存器模块中，这样便可以对放置在 refile 模块中的 rs，rt，rd，hi，lo 等不同的寄存器进行读和取操作，最后将读取的操作数 1 和操作数 2 再通过线传回 ID 段，完成寄存器的访存操作。若解析到指令为跳转指令，则计算跳转地址，并将结

果通过 br_bus 总线传回 IF 段，以便 IF 段在最短时间拿到下一条指令的地址。另外将各种使能信号以及操作数 1 和操作数 2 通过 ID_TO_EX 总线传递给下一阶段的 EX。

## EX 段

EX 段的主要工作是进行各种运算和向存储器发出存取信号。EX 段在接收到 ID 段的信号之后，将基本操作组合的 alu_op 信号、操作数 1 和操作数 2 传入 EX 段内部的 alu 模块进行具体的运算，运算结果会由线 alu_result 返回到 EX 段中；访问存储器的地址也由 alu 模块计算得到，而写入存储器的内容取决于指令的类别，最后将访问存储器的信号由 EX 段传出 CPU 模块到达存储器模块；另外将存储器读取使能信号，EX 段计算的结果等数据通过 EX_TO_MEM 总线传到 MEM 段，方便 MEM 段后续操作。

## MEM 段

MEM 段的主要工作为获取从存储器读取到的数据，并将数据进行处理得到需要存回寄存器的值。MEM 段接收到从存储器返回的数据 data_sram_rdata，根据通过总线传来的数据 data_ram_readen、data_ram_en、ex_result 来判断截取 data_sram_rdata 的长度和位置，将结果存入 rf_wdata（存入寄存器的值）中。最后将各类寄存器的读写使能信号、地址和写入数据合并为 MEM_TO_WB 总线传入 WB 段。

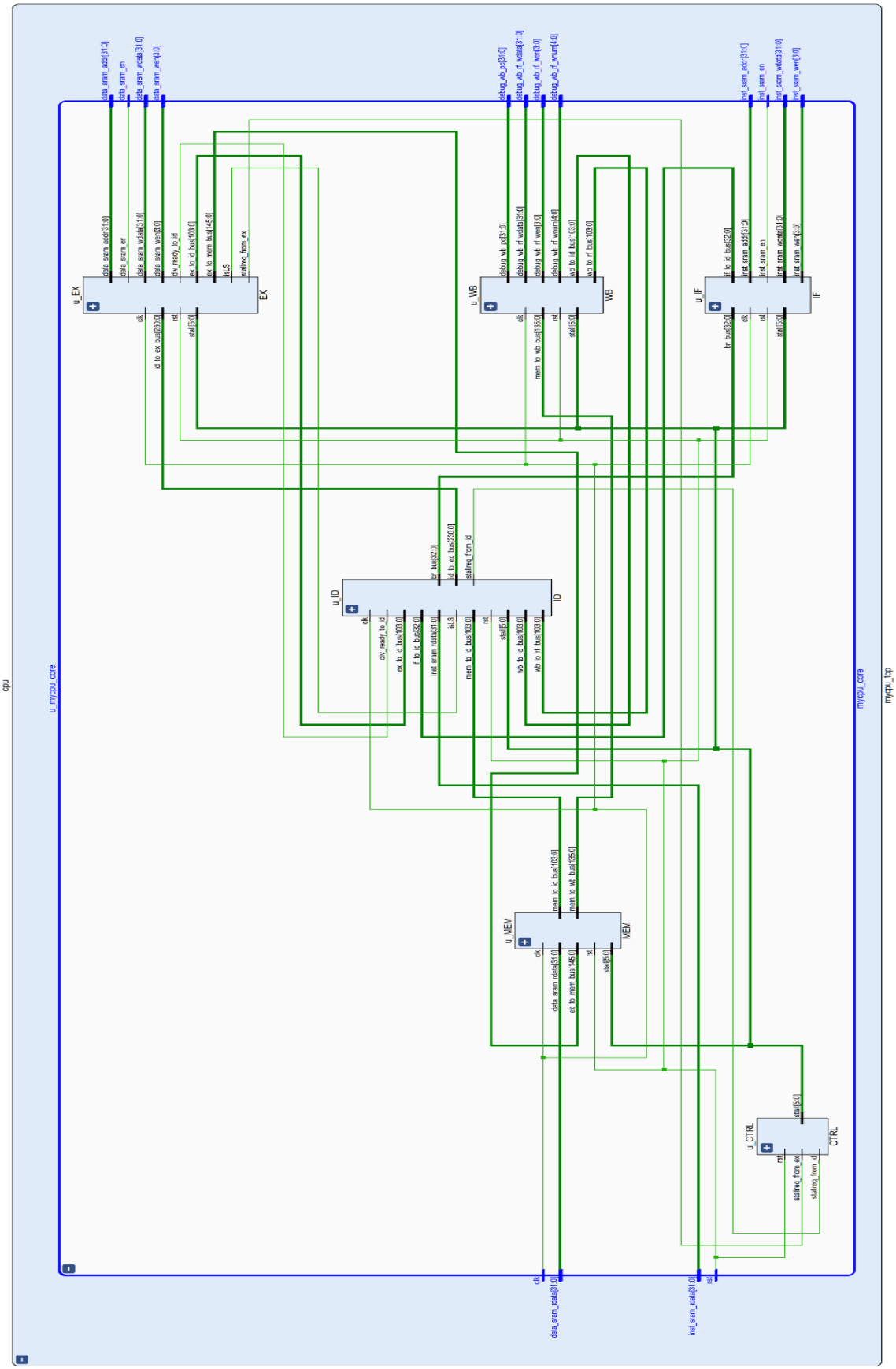
## WB 段

WB 段的工作最为简单，只起到一个数据传递的作用。

## Ctrl 段

Ctrl 段控制流水线的暂停处理。它接收来自 ID 段和 EX 段的暂停请求，对于来自不同段的暂停请求给与不同的暂停信号 stall，并传递给流水线的各个部分，使其按照 stall 信号进行暂停操作。

(3) 不同流水段的连接图:


cpu
u_mycpu_core
mycpu_core
mycpu_top
u_EX
EX
u_WB
WB
u_IF
IF
u_ID
ID
u_MEM
MEM
u_CTRL
CTRL

## （4）完成的指令：

完成指令共 52 条，包括：

（1）算术运算：

ADD，ADDI，ADDU，ADDIU；

SUB，SUBU；

SLT，SLTI，SLTU，SLTIU；

DIV，DIVU，MULT，MULTU；

（2）逻辑运算：

AND，ANDI，LUI，NOR，OR，ORI，XOR，XORI

（3）移位指令：

SLLV，SLL，SRAV，SRA，SRLV，SRL

（4）分支跳转：

BEQ，BNE，BGEZ，BGTZ，BLEZ，BLTZ，BGEZAL，BLTZAL，J，JAL，JR，JALR

（5）数据移动：

MFHI，MFLO，MTHI，MTLO

（6）访存指令：

LB，LBU，LH，LHU，LW，SB，SH，SW

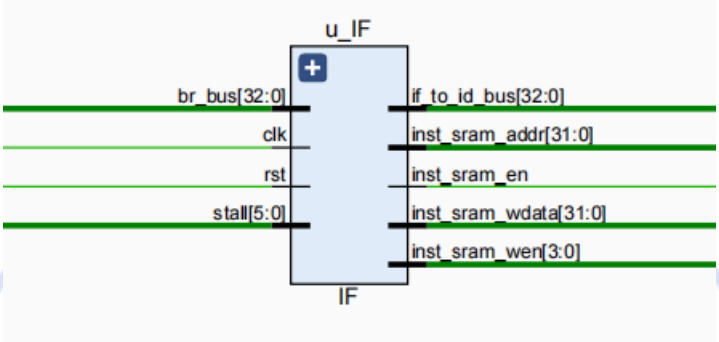
## （5）程序运行环境及使用工具：

win10 + vivado19.2

# 二、单个流水段说明:

## (1) IF 段:

### 整体功能:




### 1、跳转指令:

首先 IF 段接收从 ID 段传来的跳转指令数据 br_bus。在信号中，建立一个 pc_reg 用于存放当前的 pc 指令地址。next_pc 用于存放经过运算后进行跳转的下一条指令地址。Br_bus 作为从 ID 接收的信号，其中 br_e 为跳转使能信号，判断如果 br_e 为 1，那么将 next_pc 中的值赋为跳转后的指令。否则，进行正常取指令操作，例如，判断是跳转语句还是正常取指指令的代码如下:

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

### 2、传输指令地址:

IF 段的输出是将 PC 的值作为地址发送给存储器，使存储器根据 PC 地址向 ID 段返回 32 位 inst 指令。同时将 PC 值传给 ID 段方便后面指令的使用。如输出中包含 inst_sram_en、inst_sram_wen、inst_sram_addr、inst_sram_wdata。其中，如果 ce_reg 在的值根据是否需要插入气泡来判断，具体代码如下:

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

```
            ce_reg <= 1'b1;
        end
    End
```

如果需要插入气泡，那么 ce_reg 就为 0，即在之后不需要进行存储器取指令。if_to_id_bus 由 IF 段传到 ID 段，内部包含：

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

Ce_reg 为存储器取指令使能信号；Pc_reg 为 ID 段当前指令对应的地址；

最后的输出：

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

分别对输出到存储器中的信号进行赋值。如 inst_sram_en 中存储着 ce_reg，如果 ce_reg 为零，那么说明进行了插入气泡的操作，则不需要对存储器进行处理，否则使能信号为 1，传出。存储器中的指令地址为当前 pc 指令地址。数据都默认设为 0。

# 具体信号讲解：

## 输入信号讲解：

clk 和 rst 分别表示时钟信号和复位信号；

stall 表示 CTRL 控制的暂停信号；

br_bus 为 ID 传输而来的用于判断是否需要进行跳转的指令。内部包含：

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

Br_bus 作为从 ID 接收的信号，其中 br_e 为跳转使能信号，判断如果 br_e 为 1，那么将 next_pc 中的值赋为跳转后的指令。否则，进行正常取指令操作，br_addr 中存储的是跳转指令操作之后所存放的跳转地址。

### 输出信号讲解

```
1. if_to_id_bus，内部包含：
2.  {
3.          ce_reg,
4.          pc_reg
5.      };
```

其中 ce_reg 为存储器取指令使能信号，在经过是否需要插入气泡的判断之后进行赋值。pc_reg 为存储器取值指令值。

next_pc 是用于计算的下一条 pc 指令的值。Next_pc 的值在经过 br_e 的判断之后，由跳转指令地址或者是当前指令值+4 构成。

## （2）ID 段：

### 整体功能：

### 1、译码

首先对于 32 位的指令进行拆解方便后续指令操作，对于大部分算术运算指令其中第 26-31 位是 opcode 编码区，第 21-25 位是 rs 寄存器的编号（同时也是 base），第 16-20 位是 rt 寄存器的编号，11-15 位是 rd 寄存器的编号，6-10 位是 sa 编码区，0-5 位是 func 编码区。另外，对于一些需要使用立即数 imm 或者 offset 的指令，0-15 位便是 imm/offset。

之后将指令的使能信号设置为其唯一的编码，例如 ORI 指令只要求 opcode 段为 00_1101 即可，而 AND 指令需要 opcode 段为 00_0000 的同时还要求 sa 为 00000 和 func 为 10_0100。这样当一条指令被 ID 段接收到之后，就可以只使对应的指令使能信号变为 1，完成译码操作。

## 2、操作数内容、指令基本操作信号、寄存器存储信号

第一个操作数 sel_alu_src1 有三种来源：rs 寄存器的值，pc 值，sa 无符号拓展；第二个操作数 sel_alu_src2 有四种来源：rt 寄存器的值，imm 有符号拓展，数字 8 的拓展，imm 的无符号拓展。在每个可能的信号来源处添加使能信号，不同类别的指令点亮不同的使能信号，通过 sel_alu_src1 和 sel_alu_src2 两条线路传入 EX 段进行下一步运算操作。

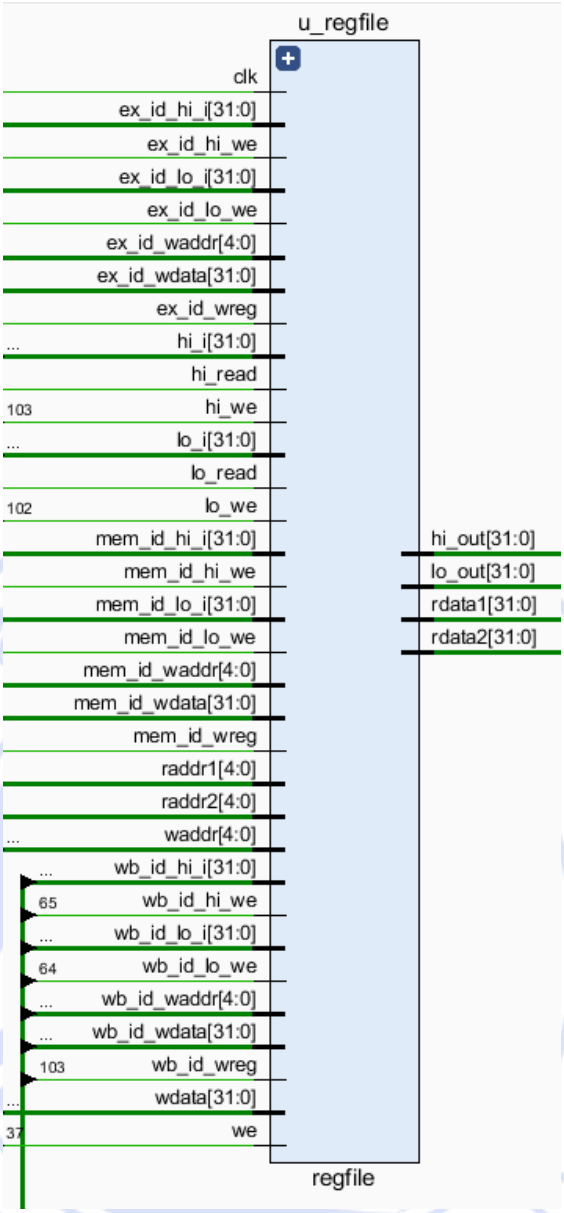
指令的基本操作类型包括：ADD，SUB，SLT，SLTU，AND，NOR，OR，XOR，SLL，SRL，SRA，LUI。将他们作为使能信号并绑定在 alu_op 的总线中，传入 EX 段中进行运算类型的判断。

寄存器存储信号包括 rf_we（寄存器存储使能信号），sel_rf_dst[0]（存入 rd 寄存器使能），sel_rf_dst[1]（存入 rt 寄存器使能），sel_rf_dst[2]（存入 31 号寄存器使能），最后利用或运算得到最终的存入寄存器的地址：

```
1. assign rf_waddr = {5{sel_rf_dst[0]}} & rd
2.                 | {5{sel_rf_dst[1]}} & rt
3.                 | {5{sel_rf_dst[2]}} & 32'd31;
```

## 3、寄存器访存与数据相关

寄存器模块被设置在 ID 段内部，其中包括 32 个 32 位寄存器以及一个 32 位 HI 寄存器和一个 32 位 LO 寄存器。寄存器模块与 ID 连接的线路如下图所示：




寄存器模块主要解决的问题为寄存器数据的取存和数据相关问题。其中有关寄存器数据存入的线路有 we（写使能信号），wdata（写入数据），waddr（写入地址），hi_we(HI 寄存器写使能)，hi_i（HI 寄存器写入数据），lo_we（LO 寄存器写使能），lo_i（LO 寄存器写入数据），该部分只需正常给对应地址赋值即可，例如 reg_array[waddr] <= wdata。然而在寄存器数据读取时会遇到数据相关问题，ex_id_wreg（EX 段结果是否要存入寄存器的信号）、mem_id_wreg（MEM 段结果是否要存入寄存器的信号）、wb_id_wreg（WB 段结果是否要存入寄存器的信号）如果为 1，则他们需要在 WB 段之后才能存入寄存器内，若在此之前需要取他们的结果就会导致 RAW 数据相关，此处采用的 forwarding 技术可以部分解决该问题。当读取数据时，若读取的地址恰好与 EX，MEM 或者 WB 段要写回的寄存器地址（ex_id_waddr、mem_id_waddr、

wb_id_waddr）相同，同时他们的写回寄存器的使能信号为真，这时就可以不从寄存器中取值，直接将 EX，MEM 或者 WB 段要存回的值（ex_id_wdata、mem_id_wdata、wb_id_wdata）赋给取值信号。例如取 rs 寄存器数据操作代码如下：

```
assign rdata1 = (raddr1 == 5'b0) ? 32'b0 :
  ((ex_id_wreg==1'b1)&&(ex_id_waddr==raddr1))?ex_id_wdata:
  ((mem_id_wreg==1'b1)&&(mem_id_waddr==raddr1))?mem_id_wdata:
  ((wb_id_wreg==1'b1)&&(wb_id_waddr==raddr1))?wb_id_wdata:
  reg_array[raddr1];
```

## 4、跳转指令

与跳转指令有关的主要信号有：br_e（跳转使能信号），br_addr（跳转地址），pc_plus_4（非跳转下一条指令地址）。当有指令是跳转指令且条件符合时，跳转使能信号就会变为 1，并根据指令的跳转规则给 br_addr 赋值。

例如 beq 跳转指令描述：如果寄存器 rs 的值等于寄存器 rt 的值则转移，否则顺序执行。转移目标由立即数 offset 左移 2 位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到。

实际操作：

```
assign inst_beq     = op_d[6'b00_0100];
assign rs_eq_rt = (rdata1 == rdata2);
assign br_e = (inst_beq & rs_eq_rt)|...
assign br_addr = inst_beq ? (pc_plus_4 + {{14{inst[15]}},
  inst[15: 0], 2'b0}) :
```

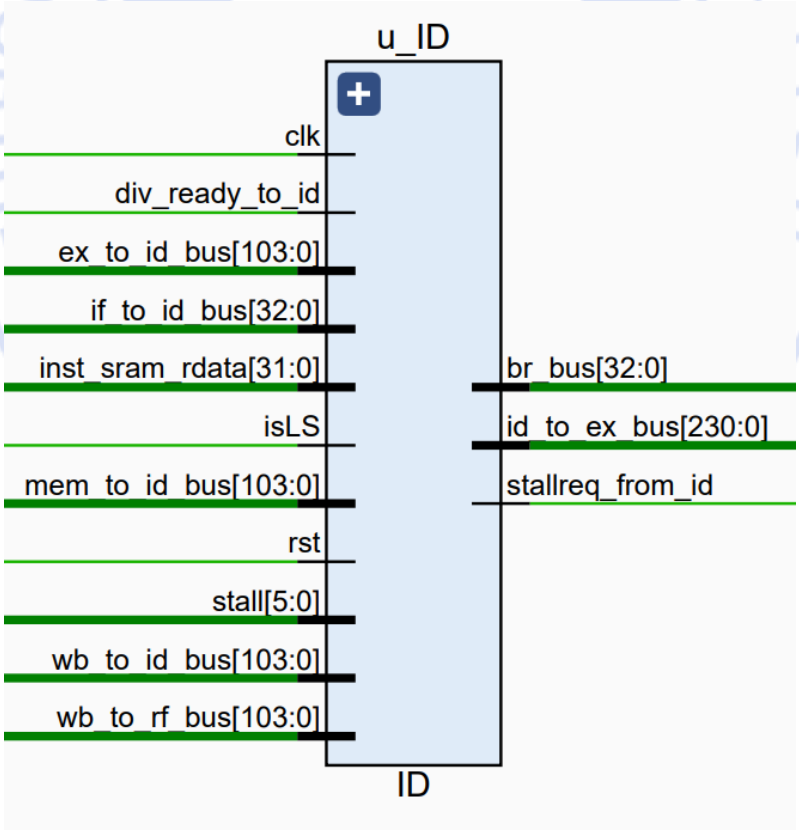
当指令的 opcode 段与 000100 匹配，则 beq 指令被点亮。若此时从 rs 取出的值 rdata1 和 rt 取出的 rdata2 也相等，则 br_e 跳转使能信号将会被点亮。对 offset 左移两位也就相当于在其后面拼接两个 0，有符号扩展就是按照最高位进行填充，最后将三部分拼接得到{{14{inst[15]}}，inst[15：0]，2'b0}，再与 pc_plus_4 相加得到跳转后的指令地址 br_addr。

## 5、请求暂停

当涉及到 LW 指令时，由于访问存储器需要在 EX 段发出访问地址，在 MEM 段才能返回数据，这就导致数据无法及时的送回到 ID 段，此时就需要加入暂停信号。首先在 EX 段判断当前的指令是否为 LW，若指令为 LW，让 isLS 信号值为 1 并传回 ID 段。ID 段接收的该值之后，再判断当前指令读取的寄存器地址是否与 LW 执行完之后需要存入的寄存器地址相同。如果两个地址相同，由于此时 LW 取储存器的值还未存入寄存器，会导致读取数据错误，所以要将请求暂停信号（stallreq_from_id）变为 1 并传给 CTRL 段。关键代码：

```
1. assign stallreq_from_id=(isLS&((rs==ex_id_waddr)|(rt==ex_id_waddr
   )))? `Stop: `NoStop;
```

ID 段接口和结构示意图：



## 具体信号讲解：

### 输入信号：

clk 和 rst 分别表示时钟信号和复位信号；

stall 表示 CTRL 控制的暂停信号；

div_ready_to_id 由 EX 段传到 ID 段，用来判断当前是否需要因除法运算而暂停周期；

isLS 由 EX 段传到 ID 段，用来判定 ID 当前周期是否需要申请暂停。

if_to_id_bus 由 IF 段传到 ID 段，内部包含：

```
1.      assign {
2.          ce,
3.          id_pc
4.  } = if_to_id_bus_r;
```

ce 为存储器取指令使能信号；

id_pc 为 ID 段当前指令对应的地址；

inst_sram_rdata 由存储器传入 ID 段。它是存储器根据 IF 段提供的 PC 地址取出的 32 位指令；

ex_to_id_bus，mem_to_id_bus，wb_to_id_bus 都是为了解决数据相关问题。（forwarding）

ex_to_id_bus 由 EX 段传到 ID 段，内部包含：

```
1.      assign {
2.          ex_id_wreg,
3.          ex_id_waddr,
4.          ex_id_wdata,
5.          ex_id_hi_we,
6.          ex_id_lo_we,
7.          ex_id_hi,
8.          ex_id_lo
9.  }=ex_to_id_bus;
```

ex_id_wreg 表示 EX 段计算出的结果是否需要存入寄存器，

ex_id_waddr 表示 EX 段计算的结果需要存放在寄存器中的地址，
ex_id_wdata 表示 EX 段计算结果的值。

ex_id_hi_we 表示 EX 段计算出的结果是否要存入 HI 寄存器，

ex_id_lo_we 表示 EX 段计算出的结果是否要存入 LO 寄存器，

ex_id_hi 表示 EX 段计算出结果的数值，

ex_id_lo 表示 EX 段计算出结果的数值。

（由于 HI，LO 寄存器都只有一个，所以在数据相关判断时不需要地址的判断，只需要使能信号和计算结果即可）。

mem_to_id_bus 由 MEM 段传到 ID 段，其中包括：

```
1.      assign {
2.          mem_id_wreg,
3.          mem_id_waddr,
4.          mem_id_wdata,
5.          mem_id_hi_we,
6.          mem_id_lo_we,
7.          mem_id_hi,
8.          mem_id_lo
9.      }=mem_to_id_bus;
```

mem_id_wreg 表示 MEM 段计算出的结果是否需要存入寄存器，
mem_id_waddr 表示 MEM 段计算的结果需要存放在寄存器中的地址，
mem_id_wdata 表示 MEM 段计算结果的值。

mem_id_hi_we 表示 MEM 段计算出的结果是否要存入 HI 寄存器，
mem_id_lo_we 表示 MEM 段计算出的结果是否要存入 LO 寄存器，

mem_id_hi 表示 MEM 段计算出结果的数值，

mem_id_lo 表示 MEM 段计算出结果的数值。

（由于 HI，LO 寄存器都只有一个，所以在数据相关判断时不需要地址的判断，只需要使能信号和计算结果即可）。

wb_to_id_bus 由 WB 段传到 ID 段，其中包括：

```
1.      assign {
```

```
2.          wb_id_wreg,
3.          wb_id_waddr,
4.          wb_id_wdata,
5.          wb_id_hi_we,
6.          wb_id_lo_we,
7.          wb_id_hi,
8.          wb_id_lo
9.      }=wb_to_id_bus;
```

wb_id_wreg 表示 WB 段计算出的结果是否需要存入寄存器，

wb_id_waddr 表示 WB 段计算的结果需要存放在寄存器中的地址，
wb_id_wdata 表示 WB 段计算结果的值。

wb_id_hi_we 表示 WB 段计算出的结果是否要存入 HI 寄存器，

wb_id_lo_we 表示 WB 段计算出的结果是否要存入 LO 寄存器，

wb_id_hi 表示 WB 段计算出结果的数值，

wb_id_lo 表示 WB 段结果的数值。

（由于 HI，LO 寄存器都只有一个，所以在数据相关判断时不需要地址的判断，只需要使能信号和计算结果即可）。

wb_to_rf_bus 由 WB 段传入 ID 段，起到将之前计算的结果存入对应寄存器的作用：

```
1.  assign {
2.          ex_rf_hi_we,
3.          ex_rf_lo_we,
4.          ex_rf_hi,
5.          ex_rf_lo,
6.          wb_rf_we,
7.          wb_rf_waddr,
8.          wb_rf_wdata
9.      } = wb_to_rf_bus;
```

ex_rf_hi_we 表示 HI 寄存器存入使能，

ex_rf_lo_we 表示 LO 寄存器存入使能，

ex_rf_hi 表示 HI 寄存器存入的数据，

ex_rf_lo 表示 LO 寄存器存入的数据;

wb_rf_we 表示普通寄存器存入使能,

wb_rf_waddr 表示普通寄存器存入地址,

wb_rf_wdata 表示存入数据。

## 输出信号:

br_bus 由 ID 传出到 IF 段，起到下一条 PC 值跳转的作用:

```
1.      assign br_bus = {
2.          br_e,
3.          br_addr
4.  };
```

br_e 表示 PC 跳转使能信号，br_addr 表示 PC 跳转地址;

stallreq_from_id 由 ID 段传出到 CTRL 段，表示 ID 段的请求暂停信号;

id_to_ex_bus 由 ID 段传出到 EX 段，包括多种信号:

```
1. assign id_to_ex_bus = {
2.           data_ram_readen, //3
3.           hi_write,
4.           lo_write,
5.           hi_read,
6.           lo_read,
7.           hi_out_file,
8.           lo_out_file,
9.           id_pc,          // 158: 127
10.          inst,           // 126: 95
11.          alu_op,         // 94: 83
12.          sel_alu_src1,   // 82: 80
13.          sel_alu_src2,   // 79: 76
14.          data_ram_en,    // 75
15.          data_ram_wen,   // 74: 71
16.          rf_we,          // 70
17.          rf_waddr,        // 69: 65
```

```
18.         sel_rf_res,       // 64
19.         rdata1,          // 63: 32
20.         rdata2           // 31: 0
21. };
```

data_ram_readen 长度 4 位，用来区分不同类别的访存指令；

hi_write 表示 HI 寄存器写入信号，lo_write 表示 LO 寄存器写入信号；

hi_read 表示 HI 寄存器读取信号，lo_read 表示 LO 寄存器读取信号；

hi_out_file 表示 HI 寄存器读取的数据，

lo_out_file 表示 LO 寄存器读取的数据；

id_pc 与 IF 传入的值相同，表示当前指令对应的地址；

inst 表示当前 32 位指令；

alu_op 表示基本运算类型的综合，长度 12 位，其中最多一个信号被点亮；

sel_alu_src1 表示操作数 1 的类型：rs、pc、sa 无符号拓展；

sel_alu_src2 表示操作数 2 的类型：rt、imm 有符号拓展、imm 无符号拓展、数字 8 的 32 位表示

data_ram_en 表示存储器读取使能；

data_ram_wen 表示存储器写使能；

rf_we 表示寄存器存储使能；

rf_waddr 表示寄存器写入地址；

sel_rf_res 表示寄存器写入类别：rt、rd、31 号寄存器；

rdata1 表示从寄存器取出的 rs 的值；

rdata2 表示从寄存器取出的 rt 的值；

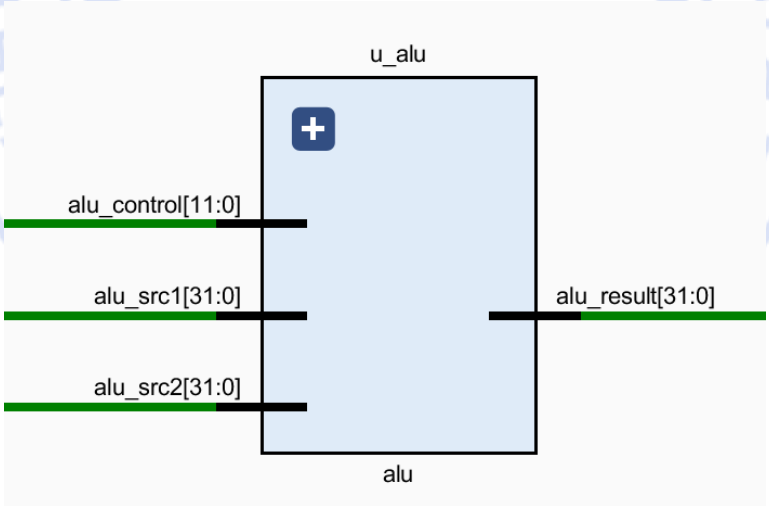
## （3）EX 段：

### 整体功能：

在 EX 段中，主要的进行运算操作以及向存储器发出存取信号，同时还会向下一层 MEM 段传输 HILO 相关信号。首先 EX 的 input 为 id_to_ex_bus 和 stall 信号。EX 段接受从 ID 段传输过来的解析过后的指令以及是否 stall 的使能信号。id_to_ex_bus 信号中包含了 ID 解析完指令后的所有数据，以及与 HILO 寄存器之间的信号。当 ID 段中判断指令是否需要进行添加气泡操作，然后将此使能信号传送给 EX 段。EX 的 output 接口主要为 EX 段向 MEM 段发送的数据、向 ID 段传送的数据、存储器返回数据 data_sram 的相关数据。以及一些是否进行乘除法和添加气泡的使能信号传输。

### 1、alu 模块

运算部分主要依赖于 EX 段内嵌的 alu 模块，模块接口如下图：




alu_src1 和 alu_src2 分别为两个操作数，alu_control 由 ID 段的 alu_op 而来，存储了 12 种基本指令的使能信号（只有其中一个为 1）。由于从 ID 段传来的只有 sel_alu_src1 和 sel_alu_src2 两个使能信号组，所以在 EX 段需要进行判断 alu_src1 和 alu_src2 应当存储的内容，可以通过选择器进行选择：

```
1. assign alu_src1 = sel_alu_src1[1] ? ex_pc :
2.                   sel_alu_src1[2] ? sa_zero_extend : rf_rdata1;
```

```
3. assign alu_src2 = sel_alu_src2[1] ? imm_sign_extend :
4.                  sel_alu_src2[2] ? 32'd8 :
5.                  sel_alu_src2[3] ? imm_zero_extend : rf_rdata2;
```

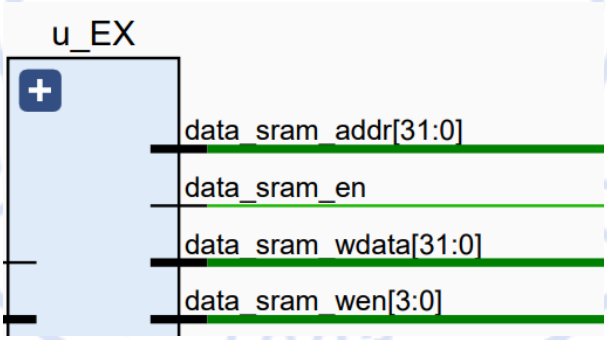
将操作数 alu_src1，操作数 alu_src2 以及 alu_control 传入 alu 模块之后，首先对 alu_control 进行拆解，拆解为原来的 12 种基本运算类型，并且对于每种操作类型给与具体的操作流程。

例如 LUI 指令，指令描述：将 16 位立即数 imm 写入寄存器 rt 的高 16 位，寄存器 rt 的低 16 位置 0。

由于 imm 属于操作数二 alu_src2 的内容，所以我只要取 alu_src2 的低 16 位与 16 个 0 拼接即可。代码如下：assign lui_result = {alu_src2[15：0]，16'b0};

在算完结果之后，用拆解出来所有的使能信号与其对应结果进行与运算，这样就只有当使能信号为真的时候计算结果才有效，再将这些与运算之后的结果进行或运算得到最后的 alu_result 由 alu 模块传回 EX 段。

## 2、内存访问信号传递




EX 段向内存发送的信号包括 data_sram_en（内存读取使能），data_sram_wen（内存写入使能），data_sram_addr（内存访问地址），data_sram_wdata（内存写入数值）。

内存读取使能信号由 ID 段的 data_ram_en 传来，直接赋值即可。内存写使能信号为 4 位，其值与具体指令和 EX 段计算结果的最后两位有关。内存访问地址便是 alu 模块计算出的结果。内存写入数值与内存写入信号的类型有关，例如当 data_sram_wen 等于 0001 时，写入内存内容为进行无符号扩展的操作数二后八位。

## 3、请求暂停

EX 段的暂停请求来源于乘除法操作，由于该操作执行时间多于一个周期，所以当前指令为乘除法且乘除法准备信号为 0 时就需要请求暂停。

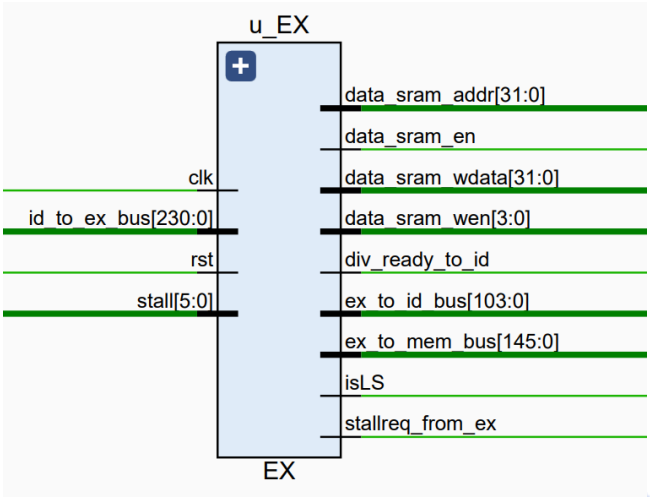
## 4、数据相关

该段有最新得到的 ex_result，hi_ex 和 lo_ex 的运算结果，为了尽快将数据提供给后续正好要取该寄存器内容指令使用，需要添加 ex_to_id_bus 总线。由于需要知道该数据最后属于哪一个寄存器以及是否有效，所以在总线中必须要有寄存器地址和使能信号。

ex_to_id_bus 内容包括如下：

```
1. assign ex_to_id_bus={
2.          rf_we，（写寄存器使能信号）
3.          rf_waddr，（写入寄存器地址）
4.          ex_result，（写入寄存器内容）
5.          hi_we，（HI 寄存器使能）
6.          lo_we，（LO 寄存器使能）
7.          hi_ex，（HI 寄存器存入内容）
8.          lo_ex（LO 寄存器存入内容）
9.      };
```

具体信号讲解：




输出信号：

首先在 EX 段中主要是起运算功能，alu_op 用于接收从 ID 段传输而来的操作信号。Sel_alu_src1 和 Sel_alu_src2 为接收的操作数选择信号。从 ID 段接收之后，进行判断，alu_src1 操作数是 ex_pc 还是 sel_alu_src[1]，alu_src1[2]操作数是判断 sa 零拓展还是 rf_rdata1。同理，Sel_alu_src1[1]的值是判断是否为立即数符号拓展，若是则 alu_src2 变为该值；Sel_alu_src1[2]判断是否为，Sel_alu_src1[3]则判断是立即数零拓展还是寄存器数据 rf_rdata2。最终将操作信号和操作数传入 alu 处理。

气泡信号为 isLS 信号，是根据接收的 inst[31：26]判断是否为 6’b100011，如果是，则说明需要添加气泡操作，否则不需要，然后将 isLS 信号传入 MEM 段。气泡的添加使能信号 STALL 从 ID 段接收之后，在 always 执行周期时会进行判断，如果 stall[2]==`Stop && stall[3]==`NoStop 这个判断语句成立，那么会将 id_to_ex_bus_r 赋值为 0，否则就会继续执行操作。

第三部分为乘除法器。首先添加乘除法指令。乘法分别为 mult 和 multu。乘除法的详细指令运算会在 mul 和 div 模块详细讲解。新建 mul_result 和 mul_signed 变量来存储乘法运算的符号以及最终运算得到的值。if_mul 用来存储指令。如果 if_mul 为不为零，那么就将寄存器里取到的值 rf_rdata1 传入 rf_rdata_mul1，rf_rdata2 传入 rf_rdata_mul2。之后进行乘法运算。除法同理。

第四部分为 HILO 的 EX 模块阶段，译码阶段的结果会传递到 EX 段并据此进行计算。考虑到 EX 段需要读写 HI、LO 寄存器，还要解决 HI、LO 寄存器带来的相关问题。新建相关变量来接收存储 HI、LO 寄存器的值以及是否要写入 HILO 寄存器，同时打包向后阶段的 HILO 信号。hi_read、lo_read 为读取的当前 hi、io 寄存器内的值。HI_write 和 LO_write 为向 HILO 寄存器输出的值。如：

```
1. assign ex_result =  hi_read ? hi_out_id:
2.                         lo_read ? lo_out_id:
3.                         alu_result;
```

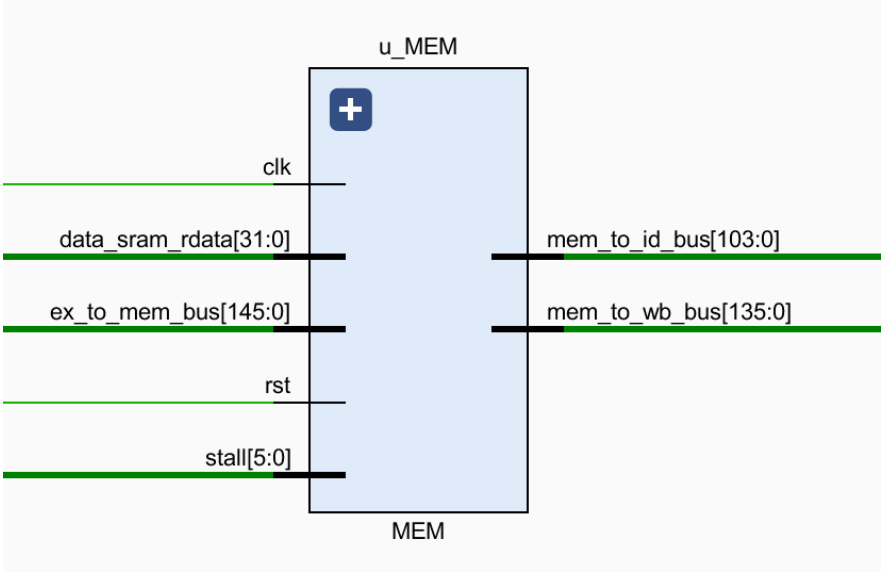
判断如果从 ID 段接收到需要向 HILO 寄存器读写的指令，那么则会自动判断，然后将 ex_result 的值写为 hi_read 或者 lo_read。如果没有接收到 HILO 寄存器的处理指令，那么就为 alu_result。

## （4）MEM 段：

### 整体功能：

MEM 段功能较为单一，主要是接收并处理内存返回的查询结果。

接口示意图如下：



## 1、内存返回结果处理

data_sram_rdata 是内存返回的查询结果，长度为 32 位。但由于不同的指令需要的查询结果内容不同，所以需要根据 ex_to_mem_bus 总线里的 data_ram_readen，data_ram_en， ex_result 信号的综合判断，才能截取到对应指令想要的查询结果。ex_to_mem_bus 包括如下：

```
1.      assign {
2.          data_ram_readen,
3.          hi_we,
4.          lo_we,
5.          hi_ex,
6.          lo_ex,
7.          mem_pc,          // 75: 44
8.          data_ram_en,     // 43
9.          data_ram_wen,    // 42: 39
10.         sel_rf_res,      // 38
11.         rf_we,           // 37
12.         rf_waddr,        // 36: 32
13.         ex_result        // 31: 0
14.     } =  ex_to_mem_bus;
```

例如 LB 指令需要的是读取内存中一个字节的有符号拓展数据。LB 指令对应的信号 data_ram_readen（4 位）为 0001，data_ram_en 为 1，且 ex_result 的最后两位 00，当这些条件都满足时就将 data_sram_rdata 的后 8 位取出并有符号扩展，结果存储到 rf_wdata 中等待存回寄存器中：

```
1. assign rf_wdata = (data_ram_readen==4'b0001 && data_ram_en==1'b1
   && ex_result[1: 0]==2'b00) ? ({{24{data_sram_rdata[7]}},
   data_sram_rdata[7: 0]}):
```

## 2、数据相关

MEM 段产生了最新的计算结果 rf_wdata（访问内存返回值），为了尽快将数据提供给后续正好要取该寄存器内容指令使用，需要添加 mem_to_id_bus 总

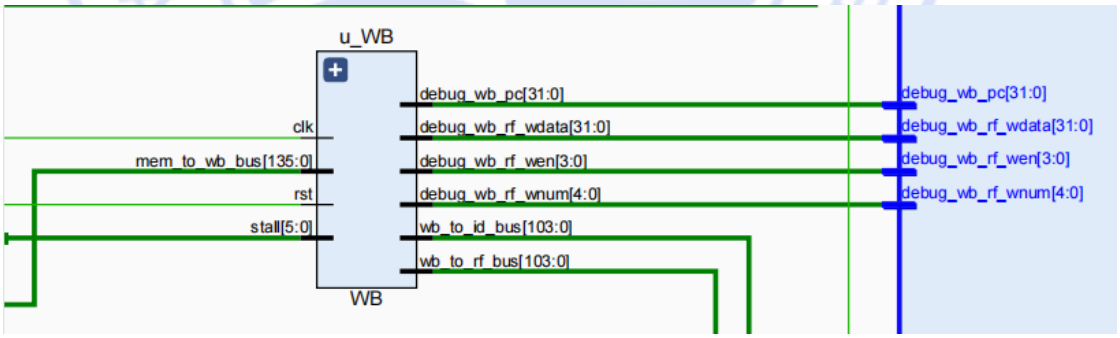
线。由于需要知道该数据最后属于哪一个寄存器以及是否有效，所以在总线中必须要有寄存器地址和使能信号。mem_to_id_bus 内容如下：

```
1.      assign mem_to_id_bus={
2.          rf_we，（写寄存器使能信号）
3.          rf_waddr，（写入寄存器地址）
4.          rf_wdata，（写入寄存器内容）
5.          hi_we，（HI 寄存器使能）
6.          lo_we，（LO 寄存器使能）
7.          hi_ex，（HI 寄存器存入内容）
8.          lo_ex（LO 寄存器存入内容）
9.  };
```

## （5）WB 段：

### 整体功能：

首先 WB 段接收 MEM 段传输过来的 HILO 寄存器相关的存储指令。在这里判断是否需要进行 HILO 写入。HILO 指令相关信息通过 hi_we，lo_we，hi_ex，lo_ex，从 EX 段一直传输到 WB 段，之后再直接送到 HILO 模块。并且根据这个修改 HILO 寄存器的值。




另一个传输操作为正常接收 MEM 段传输到到 WB 段的信号并且向寄存器传输和 ID 段传输。

```
1. always @ (posedge clk) begin
2.          if (rst) begin
3.              mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
4.          End
5.          else if (stall[4]==`Stop && stall[5]==`NoStop) begin
```

```
            mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
        end
        else if (stall[4]==`NoStop) begin
            mem_to_wb_bus_r <= mem_to_wb_bus;
        end
    end
```

如果复位信号为 1，将 mem_to_wb_bus_r 赋值为 0。如果需要进行插入气泡，那么也将 mem_to_wb_bus_r 赋值为 0。经过判断，不需要插入气泡。则将 mem_to_wb_bus 的值给 mem_to_wb_bus_r

## 具体信号讲解:

### 输入信号:

clk 和 rst 分别表示时钟信号和复位信号;

stall 表示 CTRL 控制的暂停信号;

mem_to_wb_bus 为 MEM 段向 WB 段传输的信号，其中内部结构为:

```
    assign mem_to_wb_bus = {
        hi_we,
        lo_we,
        hi_ex,
        lo_ex,
        mem_pc,     // 69: 38
        rf_we,      // 37
        rf_waddr,   // 36: 32
        rf_wdata    // 31: 0
    };
```

其中 hi_we，lo_we 为是否需要向 HILO 寄存器中写入的使能信号。hi_ex 和 lo_ex 为访存阶段要写入 HILO 寄存器的值。最后输出 debug 用的信号。将 HILO 相关操作信号转给 HILO 模块。相关寄存器操作转给 ID 段。

### 输出信号:

wb_to_rf_bus 表示 WB 段向寄存器段传递的信号。其内部结构为:

```
1. assign wb_to_rf_bus = {hi_we, lo_we, hi_ex, lo_ex, rf_we,
   rf_waddr, rf_wdata};
```

其中 hi，lo 都为 HILO 寄存器相关指令。rf_we 为向寄存器传输信号的使能信号。rf_waddr 为传输信号的地址。rf_wdata 则为所传输的数据。

debug_wb_pc，debug_wb_rf_wen，debug_wb_rf_wnum，debug_wb_rf_wdata 分别为输出的测试指令，使能信号，以及指令的输出值。

wb_to_id_bus，表示 WB 段向 ID 段回写的信号。内部结构为：

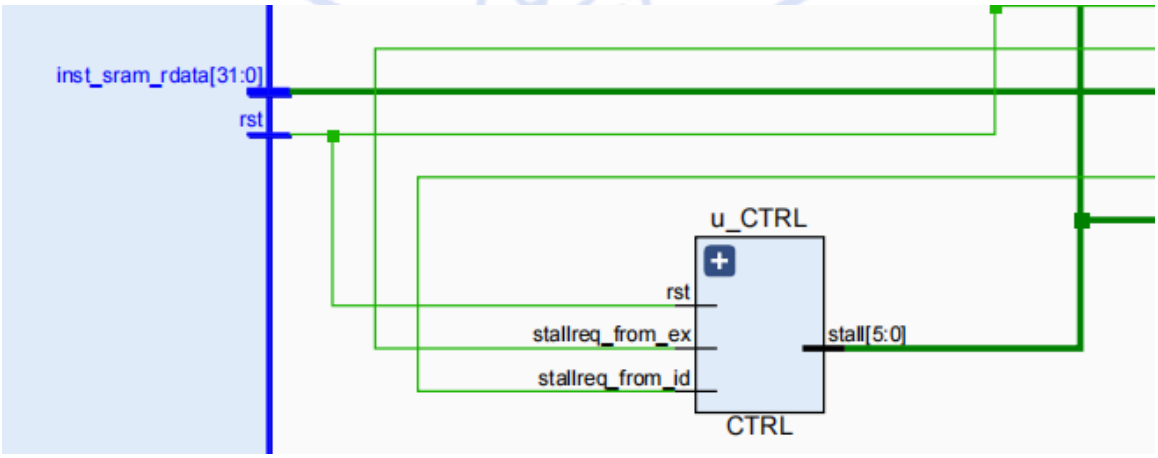
```
1. assign wb_to_id_bus={rf_we, rf_waddr, rf_wdata, hi_we, lo_we,
   hi_ex, lo_ex};
```

rf_we 表示 WB 段向寄存器取的使能信号，rf_waddr，rf_wdata 分别代表取的地址和数据存放。主要是通过 ID 段来传输。其余信号为 HILO 相关操作信号。

## （6）Ctrl 段：

## 整体功能：

控制流水线的暂停处理。它接收来自 ID 段和 EX 段的暂停请求，对于来自不同段的暂停请求给与不同的暂停信号 stall，并传递给流水线的各个部分，使其按照 stall 信号进行暂停操作。



```
1.  always @ (*) begin
2.          if (rst) begin
3.              stall = `StallBus'b0;
4.          end else if(stallreq_from_id==`Stop) begin
5.              stall= 6'b00_0111;
6.          end else if(stallreq_from_ex==`Stop) begin
7.              stall=6'b00_1111;
8.          end
9.          else begin
10.             stall = `StallBus'b0;
11.         end
12.     end
```

一开始暂停信号设定为 0，如果从 ID 段接收到的 stallreq_from_id 信号为暂停信号，那么就给 stall 信号赋值为 6'b00_0111，后三位为 1，用于在其他段的暂停操作判断。如果从 EX 段接收到的 stallreq_from_id 信号为暂停信号，那么就给 stall 信号赋值为 6'b00_1111，后四位为 1，用于在其他段的暂停操作判断。

## 具体信号讲解:

### 输入信号:

rst 复位信号

stallreq_from_ex，从 EX 段传输过来的暂停信号。

stallreq_from_id，从 ID 段传输过来的暂停信号。

### 输出信号:

Stall 暂停信号

# 三、32 周期的移位乘法器说明:

## 增加 my_mul 模块:

在原来的基础上增加了 my_mul.v 文件，my_mul 模块接口含义如下:

```
1. module my_mul(
2.  input wire rst,                  //复位
3.  input wire clk,                  //时钟
4.  input wire signed_mul_i,         //是否为有符号乘法运算，1 位有符号
5.  input wire[31:0] muldata1_i,     //被乘数
6.  input wire[31:0] muldata2_i,     //乘数
7.  input wire start_i,              //是否开始乘法运算
8.  input wire annul_i,              //是否取消乘法运算，1 位取消
9.  output reg[63:0] result_o,      //乘法运算结果
10. output reg ready_o               //乘法运算是否结束
11.);
```

My_mul 模块的主要部分是一个状态机，共有三个状态:

MulFree:乘法模块空闲。

MulOn： 乘法运算进行中。

MulEnd： 乘法运算结束。

## MulFree:

开始乘法运算，若被乘数或乘数为有符号运算，要进行取补码再保存

没有开始乘法运算，保持 ready_o 为 `MulResultNotReady；result_o 为 0

```
1. `MulFree: begin    //乘法器空闲
2.      if (start_i == `MulStart && annul_i == 1'b0) begin
3.        state <= `MulOn;
4.        cnt_mul <= 6'b000000;
5.        if(signed_mul_i == 1'b1 && muldata1_i[31] == 1'b1) begin
    //被乘数为负数
6.         temp_op1 = ~muldata1_i + 1;
7.        end else begin
```

```
        temp_op1 = muldata1_i;
      end
      if (signed_mul_i == 1'b1 && muldata2_i[31] == 1'b1 ) begin
    //乘数为负数
       temp_op2 = ~muldata2_i + 1;
      end else begin
       temp_op2 = muldata2_i;
      end
      multiplicand <= {32'b0,temp_op1};//被乘数
      multiplier <= temp_op2;          //乘数
      product_temp <= {`ZeroWord, `ZeroWord};
    end else begin
     ready_o <= `MulResultNotReady;
     result_o <= {`ZeroWord, `ZeroWord};
    End
```

MulOn：

（1）如果输入信号 annul_i 为 1，表示处理器取消乘法运算，那么 my_mul 模块直接回到 MulFree 状态

（2）如果 annul_i 为 0，且 cnt _mul 不为 32，进行值得存储，同时保持 MulOn 状态，cnt _mul+1

（3）如果 annul_i 为 0，且 cnt 为 32，如果是有符号乘法，且被乘数、乘数一正一负，结果取补码，得到最终的结果。同时进入 DivEnd 状态，cnt_mul 归零。

```
`MulOn: begin    //乘法运算
    if(annul_i == 1'b0) begin   //进行乘法运算
     if(cnt_mul != 6'b100000) begin
      multiplicand <= {multiplicand[62:0],1'b0};  //被乘数x 每次左移一位。
      multiplier <= {1'b0,multiplier[31:1]};      //相当于乘数y右移一位
      product_temp <= product_temp + partial_product;
      cnt_mul <= cnt_mul + 1;                  //乘法运算次数
     end else begin
      if ((signed_mul_i == 1'b1) && ((muldata1_i[31] ^ muldata2_i[31]) == 1'b1)) begin
       product_temp <= (~product_temp + 1);
      end
      state <= `MulEnd;
```

```
13.         cnt_mul <= 6'b000000;
14.       end
15.      end else begin
16.       state <= `MulFree;
17.      end
```

MulEnd:

除法运算结束，result_o 的宽度是 64 位用来存储临时结果。设置输出信号 ready_o 为 MulResultReady，表示乘法结束，然后等待 EX 模块送来 MulStop 信号，当 Ex 模块送来 MulStop 信号时，my_mul 模块回到 MulFree 状态

```
1. `MulEnd: begin    //乘法结束
2.       result_o <= product_temp;
3.       ready_o <= `MulResultReady;
4.       if (start_i == `MulStop) begin
5.        state <= `MulFree;
6.        ready_o <= `MulResultNotReady;
7.        result_o <= {`ZeroWord, `ZeroWord};
8.       end
9.     end
```

# 修改 EX 模块:

接口如下:

```
1. my_mul mymul(
2.       .rst            (rst            ),
3.       .clk            (clk            ),
4.       .signed_mul_i   (signed_mul_o   ),
5.       .muldata1_i     (mul_opdata1_o  ),//被乘数
6.       .muldata2_i     (mul_opdata2_o  ),//乘数
7.       .start_i        (mul_start_o    ),
8.       .annul_i        (1'b0           ),
9.       .result_o       (mul_result     ),
10.      .ready_o        (mul_ready_i    )
11.    );
```

如果是 mult 指令，并且 my_mul 模块没有声明除法结束（即 mul_ready_i 等于 MulResultNotReady），那么输出被乘数、乘数、除法开始信号、有符号乘法等信息 my_mul 块，设置 mul_start_o 为 MulStart，以指示 my_mul 模块开始除法运算；同时，设置 stallreq_for_mul 为 Stop,表示由于乘法运算请求流水线暂停。

反之,如果 my_mul 模块声明除法结束(即 mul_ready_i 等于 MulResultReady），那么设置 mul_start_o 为 MulStop，以指示 my_mul 模块停止除法运算；同时，设置 stallreq_for_mul 为 NoStop，表示不是由于乘法运算请求流水线暂停。

multu 指令的执行过程与 mult 指令类似。

```
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) begi
n
                        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
```

记得 EX 段传入 ID 段得暂停也要修改：

```
assign stallreq_from_ex = ((if_div) & div_ready_i==1'b0)|((if_mul
) & mul_ready_i==1'b0);
```

# 四、组员感受和改进意见:

## 韩世龙:

在本次实验中，我主要负责处理数据相关、流水线暂停处理、跳转指令和一些基本指令的添加。在这次实验中，我一开始心里有些抵触，因为之前没有这方面的知识，实验也是无从下手。在那时我总想有个人能够给我讲明白实验的大概流程，但是在找了多个人询问之后，才发现有些东西并不是通过询问就可以明白的，只有靠自己一点点看参考资料才能真正明白实验的流程和原理。就这样，我在不断的摸索和询问中逐渐明白了实验的各个模块的功能。但由于在开始时另外两个队友有没能腾出时间来搞本次实验，所以我的进度较为缓慢，但几天之后两名队员也加入了进来，我们分工合作，工作进度也突飞猛进。这让我真切地感受到一个人的力量是有限的，只有一个配合协调的团队才能更好的完成任务。万事开头难，我建议老师以后可以考虑在实验的前期多做一些演示，让同学尽快的融入实验中。

## 张有成:

在这次实验中，我主要负责添加 1 至 36 条指令的任务。在实验的一开始，我是十分的迷茫和焦躁的。因为第一次接触 CPU 的设计，但是我们专业之前并没有学习过计算机内部结构或者是组成原理的相关知识。在这次计算机系统实验开始之初，我还是有些迷茫的。不过随着自主看书学习，同时复习老师上课讲的知识，我最终了解了具体的流程和做法。也让我对 vivado 工具以及 Verilog 这门语言有了初步的认知。

在实验的过程中，我主要负责 1 到 36 条指令的添加以及 HILO 寄存器指令的部分添加。在实验一开始，我不是很了解原先代码中各个参数的作用，而且很多参数在多个模块中都有涉及，于是我就感到十分迷茫。后来我大概明白使能信号、传输的地址、传输数据的作用。我通过仔细查看 IF、ID、EX、MEM、WB 的代码，对比其中相似的点和数据之后，学会了如何添加指令。就拿 lw 指令来举例，lw 指令是稍难的一条，需要将 BASE 寄存器中的值加上符号拓展立即数得到访存的虚地址，如果地址不是 4 的整数倍，则触发地址错例，否则则根据虚地址从存储器中连续读取 4 个字节的值，写入 rt 寄存器。那么就要将 rs、符号拓展的值取到寄存器中。然后进行运算。给使能信号赋值，最终存入

rt 段中。还需要补加数据相关的线路。在实验的最后，我们完成了基本的指令添加部分。也成功在电路板上模拟仿真成功。

刘璐：

在这次实验中我主要完成 36-64 指令的添加、HILO 寄存器部分的设计、以及 32 周期的移位乘法器的编写。

由于本学期选的课太多，而且最开始的时候对于 cpu 并不熟悉写起来比较头疼，刚开始我们组只有龙哥一个人在做。直到第一节课的时候老师统计进度，意外地发现我们组的进度已经算是最慢的了，所以我开始投入绝大部分精力在这个实验上。第一天直接从下午写到第二天凌晨五点，通过了 35-51 指令。其中包括 HILO 指令的设计，后来又很快的解决了后面乘除法指令的部分，成功通过 64 个点。

在这其中我也遇到了很多问题，比如 HILO 寄存器那里思路不够清楚，后面 51-58 个点也莫名其妙的卡住，上板跑的时候文件不知道为什么生成不出来等。在这里首先我要特别感谢我的队友们，他们是我最坚强的后盾，从开始着手做到最后圆满完成，我们的小群一直很活跃。我们分工明确、互相给对方加油打气、互帮互助共同解决遇到的问题。通过这次实验，我真真切切的感觉到什么叫合作共赢。一个人的力量是有限的，只有三个人齐心协力，才能以最快的速度完成，得到最好的结果。同时我也要感谢给予我帮助的同学们，最开始上手太慢，向很多同学寻求帮助，大家都耐心的帮我答疑解惑、捋顺思路、解决问题。是他们让我觉得，2021 年的这个冬天十分温暖。

然后我要感谢老师和助教学长给予我们的帮助。在于老师的课上总能收获到很多，于老师真的算是大学阶段难得遇见的老师。助教学长更是从一开课就很照顾我们。实验过程中也在群里耐心为我们解答问题，在课上也细心为我们讲解。由衷地说一句老师和助教学长辛苦啦！

最后我想把我实验过程中的经验分享给以后的学弟学妹们。首先一定要弄清楚原理，明白原理后上手会很快。记得不要为了完成任务盲目的去做，要弄明白每一步是在干什么，捋清思路，会收获很多。看波形图 debug 也是实验中很重要的部分，因为他只会告诉你过了多少点，而且有的报错也并不是问题所在，学会看波形图会节省很多找 bug 的时间。写代码的过程中要细心，有时可能只是因为一点点小问题就卡住很久，比如要生成上板的文件时我们总生成不

出来，后来在助教学长的帮助下发现代码中有一个环的存在，这种低级的错误真的不该犯。

计算机系统的实验已经告一段落了，但是我对这一科目的研究并没有结束。这次由于时间紧迫我们只能在基础的前提上完成一点拓展，希望以后我可以通过自己的努力过后面的点。

# 五、参考文献:

[1] A03_“系统能力培养大赛”MIPS 指令系统规范_v1.01

[2] 雷思磊.自己动手做 cpu [M]．北京:科学出版社