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：颜进超 20195279

组员：卢葛威 20195186

王崇骁 20195267

报告日期：2021.12.19

目录

一、工作量说明：

颜进超(40%):解决数据相关的问题，如添加 forwarding 通路、添加 stall 等；添加了乘除法器，HI,LO 寄存器，并添加了乘除法指令及 HI、LO 寄存器有关的指令。

卢葛威(35%):添加其他的指令，协助完成了报告。

王崇骁(25%):协助添加指令，完成报告。

二、总体设计：

本实验在现有五段基础流水线 CPU 框架的基础上进行修改与完善，最终实现一个能够完成 MIPS 指令系统基本指令的 CPU，并在仿真中通过了 64 个基本点。指令集遵循 MIPS 指令集架构，采用流水线技术提高系统稳定以及工作速度与效率，并通过 forwarding 与 stall 技术解决了流水线中可能存在的数据相关。

流水段由取指令(IF)、指令译码(ID)、执行(EX)、存储器访问(MEM)和寄存器回写(WB)五段组成，每条令的执行一般需要五个时钟周期(在没有暂停的情况下)，在每个时钟周期上升沿来临时，此指令所代表的一系列的输入与输出数据和信号将转移到下一级处理。通过将不同的模块与寄存器正确地连接起来并实现相应的逻辑与信号的传递，可以实现流水线正常功能的运行。

初始的 CPU 可以执行 ori、lui、addiu、 beq 这四种指令，在该 CPU 的基础上，需要解决数据相关，并根据比对机制添加所需的指令及乘除法相关部件。需要完成的基本指令分为算术运算指令，逻辑运算指令，移位指令，分支跳转指令，数据移动指令，访存指令。在添加不同类型的指令时，需要对应地修改取指、译码、执行与访存阶段。

三、完成的指令：

以下是添加以及完成的所有指令(包括本身模块中已有四个基础指令)：

1)算术运算指令：

SUBU 指令， ADDU 指令，SLTU 指令，Slt 指令， Add 指令，Addi 指令，Sub 指令，Sltiu 指令,DIV 指令，DIVU 指令，MULTU 指令

2)分支跳转指令

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

3)逻辑运算指令

OR 指令，XOR 指令，AND 指令，NOR 指令，ANDI 指令，SLTI 指令，XORI 指令，ANDI 指令

4)移位指令

SLL 指令，SRL 指令，SRA 指令，SLLV 指令 ，SRAV 指令，SRLV 指令

5)访存指令

LW 指令，SW 指令，LB 指令，LBU 指令，LH 指令，LHU 指令，SB 指令，SH 指令

6)数据移动指令

MFHI 指令，MFLO 指令，MTLO 指令，MTHI 指令

四、程序的运行环境及使用工具：

1.操作系统：Win 10。

2.开发平台：Vivado 2019.2。

3.编程语言：VerilogHDL 硬件描述语言。

五、单个流水段的说明

5.1 IF 段：

1.整体功能说明：

IF 段作为流水线第一个阶段，起着开头的作用，即实现取指与译码阶段之间的寄存器，将取到阶段的指令以及地址，信号等信息在下一时钟传递到译码阶段，PC 值递增，准备取出下一条指令。是将指令从存储器中读取出来的过程。

2. 各个端口以及各个信号的介绍：

接下来将每一部分代码进行解释与分析：

```
module IF(
    input wire clk,
    input wire rst,
    input wire [`StallBus-1:0] stall,
    input wire [`BR_WD-1:0] br_bus,
    output wire [`IF_TO_ID_WD-1:0] if_to_id_bus,
    output wire inst_sram_en,
    output wire [3:0] inst_sram_wen,
    output wire [31:0] inst_sram_addr,
    output wire [31:0] inst_sram_wdata
);
```

1）首先接口 rst 与 clk 分别为宽度为 1 的复位信号与时钟信号，作为复位信号 rst 的作用有两点：1.使 cpu 从初始指令开始运行。2.使系统中的各部分同步运行。时钟信号 clk 的作用为时钟信号通常被用于同步电路当中，扮演计时器的角色，保证相关的电子组件得以同步运作。

2）Stall 是暂停信号，宽度为 6，是用于当一条指令需要用到前面某条指令的结果，从而不能重叠执行时，就发生了数据相关。这时就需要 Stall 充当暂停信号。

3）br_bus 是由 ID 段对分支指令传输的信号组，按位赋值给 br_e,br_addr

4）if_to_id_bus 由 IF 模块发送给 ID 模块的信号组。

5）inst_sram_en 宽度为 1，表示使能信号。

6）inst_sram_wen 宽度为 4，作用是控制对存储器的写入。

7）inst_sram_addr 宽度为 32，表示取指阶段要取的指令对应的地址。

8）inst_sram_wdata 宽度为 32，表示向存储器写的内容。

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

1）pc_reg 宽度为 32 位，表示的是要读取的指令的地址。

2）ce_reg 宽度为 1 位，表示的是指令储存器使能信号。

3）next_pc 宽度 32，表示下一 PC 地址。

4）br_e 宽度为 1，判断是否为跳转指令或分支指令且转移成功。

5）br_addr 宽度为 32，表示跳转指令转移成功时跳转到的地址。

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

1)将 br_bus 按位赋值给 br_e,br_addr

```
always @ (posedge clk) begin
        if (rst) begin
            pc_reg <= 32'hbfbf_fffc;
        end
        else if (stall[0]==`NoStop) begin
            pc_reg <= next_pc;
        end
    end
```

1)在 rst 为高电平时，复位信号有效时，代表开始，将默认指令地址赋值写入指令寄存器 pc_reg，当暂停信号为 NOSTOP 时，即继续运行，将下一 pc 地址写入 pc_reg 指令寄存器。

```
  always @ (posedge clk) begin
        if (rst) begin
            ce_reg <= 1'b0;
        end
```

```
        else if (stall[0]==`NoStop) begin
            ce_reg <= 1'b1;
        end
    end
```

1)此功能块表示正常时钟信号下，当 rst 为高电平，复位信号有效时，代表本周期的开始，将 ce_reg 初始化为 1’ b0，当控制取指阶段的暂停信号为 Nostop，表示寄存器正常执行。

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

1)下一模块是对 next_pc 的赋值，利用三元运算符，br_e 判断是否为跳转指令或分支指令且转移成功，若为 1’ b1，即发生跳转，将跳转到的地址 br_addr 赋给 next_pc,若为 1’ b0，则将指令寄存器 pc_reg 地址指向顺序的下一条指令并赋给 next_pc。

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

1）将指令存储器使能信号 ce_reg，赋值于 inst_sram_en。

2）将 inst_sram_wen 初始化。

3）对 inst_sram_addr 赋值，表示要取的指令地址。

4）对 inst_sram_wdata 的初始化。

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

1)对 ce_reg,pc_reg 的打包成 if_to_id_bus 信号组，转移到 ID 模块。

5.2 ID 段：

1.整体功能说明：

ID 段的作用是对指令进行译码，得到最终运算的类型、子类型、源操作数和要写入的目的寄存器地址等信息。其中运算类型指的是逻辑运算、移位运算、算术运算等，子类型指的是更加详细的运算类型。并进行指令分析从寄存器中读取需要的数据完成数据相关处理和生成发给 EX 段与 MEM 段的控制信号。

ID 段是将存储器中取出的指令进行翻译的过程。经过译码之后得到指令需要的操作数寄存器索引，可以使用此索引从通用寄存器组中将操作数读出。将分支和跳转指令的判断和执行放到 ID 段，可以减小分支和转移的开销，避免控制相关。

2. 各个端口以及各个信号介绍：

```
module ID(
    input wire clk,
    input wire rst,
    input wire [`StallBus-1:0] stall,
    output wire stallreq,
    input wire [`IF_TO_ID_WD-1:0] if_to_id_bus,
    input wire [31:0] inst_sram_rdata,
    input wire [`WB_TO_RF_WD-1:0] wb_to_rf_bus,
    input wire ex_rf_we,
    input wire [4:0] ex_rf_waddr,
    input wire [31:0] ex_ex_result,
    input wire mem_rf_we,
    input wire [4:0] mem_rf_waddr,
    input wire [31:0] mem_rf_wdata,
    output wire [`ID_TO_EX_WD-1:0] id_to_ex_bus,
    output wire [`BR_WD-1:0] br_bus
    input wire is_lw
    input wire [65:0] ex_hilo
);
```

1）接口 rst 与 clk 分别为宽度为 1 的复位信号与时钟信号，stall 为宽度 6 位暂停信号这里不再赘述，stallreq 宽度为 1 位，是实现加载，储存指令的时候给该信号赋值，用于产生数据相关时，ID 段暂停操作

2）if_to_id_bus 宽度为 33 位，是由使能信号与指令地址组成的信号组

3）inst_sram_rdata 宽度 32 位，表示指令内容

4）wb_to_rf_bus 宽度为 32 位，由 WB 阶段传输到寄存器的信号组

5）ex_rf_we，ex_rf_waddr,ex_ex_result 宽度分别为 1,5 与 32，表示的是处于执行阶段指令要写入的寄存器信息，在处理数据相关时，由 EX 模块引出的接线。

6）mem_rf_we，mem_rf_waddr，mem_rf_wdata 宽度分别为 1,5 与 32，表示的是处于访存阶段指令要写入的寄存器信息。

7）id_to_ex_bus 宽度为 166，表示由 ID 模块传输到 EX 模块的信号组。

8）br_bus 宽度为 33,是由分支跳转的使能信号和分支跳转的目的地址组成的信号组。

9）is_lw 判断是否为 lw 跳转指令的信号。

10）ex_hilo 宽度为 66 位，表示控制 hi 与 lo 寄存器与数据的信号组

```
reg flag;
reg [`IF_TO_ID_WD-1:0] if_to_id_bus_r;
```

wire [31:0] inst;
wire [31:0] id_pc;
wire ce;
1)flag 宽度为 1 位，判断译码阶段是否暂停的信号。
2)if_to_id_bus_r 宽度为 33 位，表示 ID 模块传输到 EX 模块的信号组。
3)inst,宽度为 32 位,表示处于译码阶段的指令内容。
4)id_pc 宽度为 32 位，表示处于译码阶段的指令地址。
5)ce 宽度为 1 位，表示使能线。
wire wb_rf_we;
wire [4:0] wb_rf_waddr;
wire [31:0] wb_rf_wdata;
1）wb_rf_we，，wb_rf_wdata 宽度分别为 1 位，表示写回到寄存器的使能信号
2）wb_rf_waddr 宽度为 5 位，表示写回到寄存器的地址
3）wb_rf_wdata 宽度为 32 位，表示写回到寄存器的内容

```
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;
```

1）opcode 宽度为 6 位，表示操作码
2）rs,rt,rd,sa 宽度为 5 位，表示 R-R 型指令中源寄存器 rs，rt 和目的寄存器 rd 对应的位，以及移位量 sa
3）func 宽度为 6 位，表示具体的运算操作编码
4）imm 宽度为 16 位，表示立即数(I 类指令)
5）instr_index 宽度为 26 位，表示与 PC 相加的偏移量(J 类指令)
6）code 宽度为 20 位，表示异常处理指令中的 code 段
7）base 宽度为 5 位，表示寄存器储存的地址，即基址
8）offset 宽度为 16 位，表示偏移量
9）sel 宽度为 3 位，是特权指令中的前三位，表示使用协处理器
wire [63:0] op_d, func_d;
wire [31:0] rs_d, rt_d, rd_d, sa_d;
1）op_d, func_d 宽度为 64 位，表示操作的具体内容

2）rs_d, rt_d, rd_d, sa_d 宽度为 32 位，表示寄存器或移位量具体值

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

1）sel_alu_src1， sel_alu_src2 宽度分别为 3 位与 4 位，分别用来选择 alu 两个源操作数的位置

2）alu_op 宽度为 12 位，表示不同子类型的 alu 操作

```
wire data_ram_en;
wire data_ram_wen;
```

1）宽度为 1 位，表示指令是否需要从内存中读写数据

2）宽度为 1 位，表示指令是否有数据写入内存

```
wire rf_we;
wire [4:0] rf_waddr;
wire sel_rf_res;
wire [2:0] sel_rf_dst;
```

1）rf_we 宽度为 1 位，作用是判断指令是否有要写入目的寄存器

2）rf_waddr 宽度为 5 位，表示指令要写入目的寄存器的地址

3)sel_rf_res 宽度为 1 位，访存阶段使用，判断指令是运算类型或 load 类型

4)sel_rf_dst 宽度为 3 位，用来判断目的寄存器是 rd 或 rt 或 31 号寄存器

```
wire [31:0] rdata1, rdata2, uprdata1, uprdata2;
wire hir,lor,hiwen,lowen;
wire [31:0] hilodata,hidata,lodata;
```

1）宽度为 32 位，rdata1, rdata2 表示从 regfile 输入的数据 uprdata1, uprdata2 表示更新后的值，在数据相关时使用

2)hir 表示 hi 寄存器的读信号，lor 表示 lo 寄存器的读信号,hiwen 表示 hi 寄存器的写信号,lodata 表示 lo 寄存器的写信号，宽度全为 1 位

3)hilodata 是复用变量，根据读取 hi 寄存器或 lo 寄存器信号读取相应寄存器的数值，hidata 表示写入 hi 寄存器的数据，lodata 表示写入 lo 寄存器的数据，宽度全为 32 位

```
wire inst_ori, inst_lui, inst_addiu, inst_beq, inst_subu;
wire inst_jal, inst_jr, inst_addu, inst_bne, inst_or ;
.....
wire inst_sb,inst_sh;
```

1)宽度为 1，表示已有的指令，如 ori，lui，addiu 等等指令

```
wire op_add, op_sub, op_slt, op_sltu;
wire op_and, op_nor, op_or, op_xor;
```

wire op_sll, op_srl, op_sra, op_lui;

1）宽度为 1，表示各种 alu 运算，分别是加、减、有符号小于置 1、无符号小于设置 1，以及位与等等

wire [31:0] mfdata;

1)mfdata 宽度为 32 位，用于处理 hi 或 lo 寄存器发生数据相关时的中间变量

wire br_e;

wire [31:0] br_addr;

wire rs_eq_rt;

wire rs_ge_z;

wire rs_gt_z;

wire rs_le_z;

wire rs_lt_z;

wire [31:0] pc_plus_4;

1）br_e 表示分支跳转的使能信号

2）br_addr 宽度 32 位，表示跳转的目的地址

3）rs_eq_rt 判断 rs 寄存器的值与 rt 寄存器的值是否相等

4）rs_ge_z 表示 rs 寄存器的值是否大于等于 0

5）rs_gt_z 表示 rs 寄存器的值是否大于 0

6）rs_le_z 表示 rs 寄存器的值是否小于等于 0

7）rs_lt_z 表示 rs 寄存器的值是否小于 0

8）pc_plus_4 宽度为 32 位，表示 pc 地址的值

assign stallreq = `NoStop;

1)首先是对 stallreq 的初始化，发生数据相关，需要暂停请求时将该信号赋初值

```
    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
```

1）在复位信号有效时，表示本周期的开始，将 if_to_id_bus_r 赋予初值，是

宽度为`IF_TO_ID_WD'位的 0，对于暂停信号的处理，在这里先给出 stall 的含义：

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。

2）当 stall[1]为 Stop，stall[2]为 Nostop 时，表示取指阶段暂停而译码阶段继续，表示无 if_to_id_bus，将 if_to_id_bus_r 赋为 0

3）stall[1]为 Nostop 时，取指的指令进入译码阶段，将 if_to_id_bus 信号赋值给 if_to_id_bus_r

```
reg flag;
        always @ (posedge clk) begin
        if (stall[1]==`Stop) begin
            flag <= 1'b1;
        end
        else begin
            flag <= 1'b0;
        end
    end
assign inst = flag?instreg:inst_sram_rdata;
```

1)flag 表示是否发生数据相关

2)当复位信号为高电平，即本周期的开始，当 stall[1]==`Stop 表示 IF 段需要暂停，将 flag 赋值为 1’b1，需要处理数据相关

3)若 IF 段不发生暂停将 flag 置位 1’b0

4)三元运算符判断是否需要处理数据相关，若需要，则将上一周期 instreg 寄存的指令赋值给 inst，若不需要直接本周期的指令赋给 inst

```
assign inst = inst_sram_rdata;
```

1）对取指阶段取得的指令赋值给 inst，以继续流水线的译码工作

```
assign {
        hiwen,
        lowen,
        hidata,
        lodata
       } = ex_hilo;
```

1)ex_hilo 宽度为 66 位，表示控制 hi 与 lo 寄存器与数据的信号组

```
regfile u_regfile(....)
```

1)对 regfile 的例化。

```
assign opcode = inst[31:26];
.....
assign sel = inst[2:0];
```

1)根据不同指令对指令内容进行切片处理。

```
decoder_6_64 u0_decoder_6_64(...)
decoder_6_64 u1_decoder_6_64(...)
decoder_5_32 u0_decoder_5_32(...)
decoder_5_32 u1_decoder_5_32(...)
```

1)分别对应四个译码器，前两个为 6 位译码器，后两个为 5 位译码器，作用是用来根据指令特定部分的编码区别不同的指令。原理是独热码变一进制。

```
assign inst_ori     = op_d[6'b00_1101];
assign inst_lui     = op_d[6'b00_1111];
  .....
assign inst_break   = op_d[6'b00_0000]&func_d[6'b001_101];
assign inst_eret    = op_d[6'b01_0000]&func_d[6'b01_1000];
```

1)根据指令集中不同的操作码，以此来激活信号，区分不同的指令。

```
assign sel_alu_src1[0] = inst_ori | inst_addiu |inst_subu ...;
assign sel_alu_src1[1] = inst_jal...|inst_jalr;
assign sel_alu_src1[2] = inst_sll|inst_sra;
assign sel_alu_src2[0] = inst_subu|inst_addu|inst_sll|inst_or.....;
assign sel_alu_src2[1] = inst_lui | inst_addiu|inst_lw....;
assign sel_alu_src2[2] = inst_jal....;
assign sel_alu_src2[3] = inst_ori....;
```

1)sel_alu_src1 用来判断 alu 的第一个操作数位置，分别是 rs 中的值、pc 值或 sa 移位量

2) sel_alu_src2 用来判断 alu 的第二个操作数位置,分别是 rt 中的值、立即数符号扩展的值、32’b8 或立即数 0 扩展的值

```
assign op_add = inst_addiu |inst_jal|inst_addu|inst_lw.....;
......
assign op_lui = inst_lui;
```

1）根据有关指令采用的运算种类及操作方式分类

```
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};
```

1) 确定 alu 操作类型,也是独热码。

```
assign data_ram_en = inst_lw......| inst_sb |inst_sh;
assign data_ram_wen = inst_sw| inst_sb |inst_sh;
```

1）表示 lw，sw...sb,sh 指令需要从储存器中读写数据

2）表示 sw,sb,sh 指令会将数据写入储存器

```
assign sl = inst_lw  ? 4'b0001
              :inst_sw  ? 4'b0010
              :inst_lb  ? 4'b0011
              :inst_lbu ? 4'b0100
              :inst_lh  ? 4'b0101
              :inst_lhu ? 4'b0110
              :inst_sb  ? 4'b0111
              :inst_sh  ? 4'b1000
               :4'b0000;
```

1)用来区分不同的存取指令。

```
assign rf_we = inst_ori....|inst_lbu|inst_lh|inst_lhu;;
```

1）表示该指令执行后会将结果写入到寄存器

```
assign sel_rf_dst[0] = inst_subu|inst_addu.....|inst_mflo;
assign sel_rf_dst[1] = inst_ori | inst_lui.....|inst_lhu;
assign sel_rf_dst[2] = inst_jal|inst_bgezal|inst_bltzal;
```

1）表示结果写入目的 rd 寄存器

2）表示结果写入 rt 寄存器中

3）表示结果写入第 31 号寄存器

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

1）根据 sel_rf_dst 的值确定写入寄存器的地址

```
assign sel_rf_res =inst_lw|inst_lb|inst_lbu|inst_lh|inst_lhu;
assign stallreq = (is_lw &&((rs == ex_rf_waddr)||(rt == ex_rf_waddr)))
```

1）访存指令，在 MEM 阶段确定写入寄存器的结果是 alu 运算的结果还是从内存中取到的数据。

2）stallreq 表示暂停的请求信号，若当前指令为 lw 跳转指令，而且若当前 rs 寄存器的地址与上一条指令在执行阶段写入寄存器的地址相等或者当前 rt 寄存器的地址与上一条指令在执行阶段写入寄存器的地址相等,则说明需要进行暂停

```
assign hir = inst_mfhi;
```

```
assign lor = inst_mflo;
```
1) 当指令为 mfhi 时，表示需要读取 hi 寄存器进行数据移动
2) 当指令为 mflo 时，表示需要读取 ho 寄存器进行数据移动
```
assign mfdata = (inst_mfhi & hiwen)? hidata:
                    inst_mfhi? hilodata:
                     (inst_mflo & lowen)? lodata:
                     inst_mflo ? hilodata:
                     32'b0;
```
1) 表示发生数据相关时对 mfdata 中间变量的赋值，三元运算符，若指令为 mfhi 表示需要读 hi 寄存器的数据且同时需要写入 hi 寄存器时，则将写入 hi 寄存器的 hidata 赋值给 mfdata
2) 若指令为 mfhi 但不需要写入 hi 寄存器时，则将从 hi 寄存器读取的复位变量 hilodata 赋值给 mfdata
3) 同理不再赘述，当不发生数据相关时，则不需要中间变量 mfdata 赋值为 0
```
assign uprdata1 = (inst_mfhi|inst_mflo)?mfdata:
((ex_rf_we == 1'b1) && (ex_rf_waddr == rs))  ?  ex_ex_result :
((mem_rf_we == 1'b1) && (mem_rf_waddr == rs))  ? mem_rf_wdata :
((wb_rf_we == 1'b1) && (wb_rf_waddr == rs))?wb_rf_wdata:rdata1;
assign uprdata2 = (inst_mfhi|inst_mflo)?mfdata:((ex_rf_we == 1'b1) &&
(ex_rf_waddr == rt))  ?  ex_ex_result :((mem_rf_we == 1'b1) &&
(mem_rf_waddr == rt))  ?  mem_rf_wdata : ((wb_rf_we == 1'b1) &&
(wb_rf_waddr == rt))? wb_rf_wdata:  rdata2   ;
```
1）数据相关时进行处理，共两个寄存器，首先判断是否需要数据相关处理，如果需要则判断是 EX 阶段或 MEM 阶段或 WB 阶段，若是 EX 阶段，且执行阶段要将数据写入到寄存器内并且当前 rs 寄存器的地址与执行阶段写入寄存器的地址相等说明发生数据相关，则将执行阶段后的结果 ex_ex_result 赋值给 uprdata1，将数据前推，其余阶段同理，若不发生数据相关，则将数据 rdata 赋值给 uprdata 即可，另外当指令为 mfhi 或者 mflo 时，则将 mfdata 赋值给 uprdata
```
assign id_to_ex_bus = {
        inst_div,
 ......
        uprdata1,          // 63:32
        uprdata2           // 31:0
    };
```
1)将由 ID 段传到 EX 段的信号组进行打包
```
assign pc_plus_4 = id_pc + 32'h4;
```

```
assign rs_eq_rt = (uprdata1 == uprdata2);
assign rs_ge_z  = (uprdata1[31] == 1'b0);
assign rs_gt_z  = (uprdata1[31] == 1'b0 &&uprdata1!= 32'h00000000);
assign rs_le_z  = (uprdata1[31] == 1'b1 ||uprdata1== 32'h00000000);
assign rs_lt_z  = (uprdata1[31] == 1'b1);
```

1）对 pc 地址加 4.对于 PC 计算问题，取下一 PC 地址，为了能够在每个时钟周期启动一条新的指令，流水线必须在 IF 段获得下一条指令的地址，并将其保存在 PC 中。但是，分支指令会改变 PC 的值，而且只有在 Mem 段结束时，这个新值才会被写入 PC，出现矛盾。因此对于分支指令需要在 ID 模块计算 PC 值

2）判断 rs 寄存器的值 uprdata1 与 rt 寄存器 uprdata2 的值是否相等

3）rs_ge_z 与 assign rs_gt_z 根据数据符号位及是否等于 0 分别代表 rs 中的值大于等于 0 和大于 0

4）rs_le_z 与 rs_lt_z 根据数据符号位及是否等于 0 分别代表 rs 中的值小于等于 0 和小于 0

```
assign br_e =  inst_j|inst_jal |inst_jr |(inst_beq & rs_eq_rt)|(inst_bne
&(!rs_eq_rt))|(inst_bgez&rs_ge_z)|(inst_bgtz&rs_gt_z)|(inst_blez&rs_l
e_z)|(inst_bltz&rs_lt_z)|(inst_bgezal&rs_ge_z)|(inst_bltzal&rs_lt_z)|
inst_jalr;
```

1）br_e 用来判断是否为跳转指令或分支指令且转移成功，等式右边是该指令能够进行跳转的条件，若满足条件则 br_e 为 1，进行跳转

```
assign br_addr =  (inst_jal|inst_j)  ?  {pc_plus_4
[31:28],inst[25:0],2'b0}:
(inst_jr|inst_jalr) ? uprdata1 :
(inst_beq|inst_bne|inst_bgez|inst_bgtz|inst_blez|inst_bltz|
inst_bgezal|inst_bltzal)
? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :32'b0;
```

1）利用三元运算符，根据指令要求跳转的地址进行赋值

3. 功能模块说明：

一共有两部分功能模块：寄存器与译码器

一，寄存器

MIPS32 架构定义了 32 个通用寄存器，用$0、$1....$31 表示，都是 32 位，且在使用中，这些寄存器的用法都遵循一系列约定，另外还有两个特殊寄存器，HI(乘除结果高位寄存器)和 LO(乘除结果低位寄存器)。进行乘法运算是，HI 和 LO 保存乘法运算的结果，其中 HI 存储高 32 位，LO 存储低 32 位，进行除法运算时，HI 和 LO 保存除法运算的结果，其中 HI 存储余数，LO 存储商。

在寄存器模块中，通过给定两个要读取得寄存器的地址和 HILO 寄存器有关

信号可以得到寄存器中保存的数据；通过写信号和地址或 HILO 寄存器有关信号可以将给定的数据写入对应的寄存器。

二，译码器

译码是将具有特定含义的二进制代码变换(翻译)成一定的输出信号，以表示二进制代码的原意。因此能实现译码功能的组合电路称为译码器，译码器是一个多输入、多输出的组合逻辑电路。它的作用是把给定的代码进行“翻译”，变成相应的状态，使输出通道中相应的一路有信号输出。

译码器能够翻译地址指令它的功能是将具有特定含义的二进制码进行辨别，并转换成控制信号。译码器在 ID 段可以用来根据指令特定部分的编码区别不同的指令。

5.3 EX 段：

1.整体功能说明：

依据译码阶段送入的源操作数，操作码，进行运算，根据当前是什么指令执行对应的操作，比如 add 指令，则将寄存器 1 的值和寄存器 2 的值相加，如果是内存加载指令，则读取对应地址的内存数据。指令译码之后所需要进行的计算类型都已得知，并且已经从通用寄存器组中读取出了所需的操作数，那么接下来便进行指令执行。指令执行是指对指令进行真正运算的过程。

2. 各个端口以及各个信号介绍：

```
module EX(
    input wire clk,
    input wire rst,
    input wire [`StallBus-1:0] stall,
    input wire [`ID_TO_EX_WD-1:0] id_to_ex_bus,
    output wire [`EX_TO_MEM_WD-1:0] ex_to_mem_bus,
    output wire data_sram_en,
    output wire [3:0] data_sram_wen,
    output wire [31:0] data_sram_addr,
    output wire [31:0] data_sram_wdata,
    output wire is_lw,
    output wire ex_id_we
    output wire stallreq_for_ex,
    output wire [65:0] ex_hilo
);
```

1）首先接口 rst 与 clk 分别为宽度为 1 的复位信号与时钟信号，stall 为宽度 6 位暂停信号这里不再赘述，

2）id_to_ex_bus 宽度为 166 位，表示 ID 阶段到 EX 阶段的信号的集合

3）ex_to_mem_bus 宽度为 75 位，表示 EX 阶段到 MEM 阶段的信号的集合

4)data_sram_en 宽度为 1 位，表示指令是否需要从内存中读取数据

5)data_sram_wen 宽度为 4 位，控制数据写入储存器的字节

6)data_sram_addr 宽度为 32 位，表示内存地址

7)data_sram_wdata 宽度为 32 位，表示写入内存的数据值

8)is_lw 宽度为 1 位，表示该指令是否为 lw 指令

9)ex_id_we 宽度为 1 位，与 is_lw 指令相关，表明判断指令写入寄存器的信号

10)stallreq_for_ex 宽度为 1 位，用来判断执行阶段是否需要暂停请求

11)ex_hilo 宽度为 66 位，表示控制 hi 与 lo 寄存器与数据的信号组

```
reg [`ID_TO_EX_WD-1:0] id_to_ex_bus_r;
```

1）id_to_ex_bus_r 宽度为 166 位，表示 ID 阶段到 EX 阶段的信号的集合

```
wire [31:0] ex_pc, inst;
wire [11:0] alu_op;
wire [2:0] sel_alu_src1;
wire [3:0] sel_alu_src2;
wire data_ram_en;
wire data_ram_wen;
wire [3:0] sl;
wire rf_we;
wire [4:0] rf_waddr;
wire sel_rf_res;
wire [31:0] rf_rdata1, rf_rdata2;
reg is_in_delayslot;
```

1）ex_pc,表示 ex 阶段的 pc 地址，inst 宽度为 32 位表示指令

2）alu_op 宽度为 12 位，表示不同子类型的操作

3)sel_alu_src1、sel_alu_src2 宽度分别为 3 位与 4 位，sel_alu_src1 用来判断 alu 的第一个操作数位置,分别是 rs 中的值、pc 值或 sa 移位量,sel_alu_src2 用来判断 alu 的第二个操作数位置,分别是 rt 中的值、立即数符号扩展的值、32'b8 或立即数 0 扩展的值

3)data_ram_en 宽度为 1 位，表示指令是否需要从内存中读取数据

4)data_ram_wen 宽度为 1 位，表示指令是否有数据写入储存器

5)sl 宽度为 4 位，用来区分不同的存取指令

6)rf_we 宽度为 1 位，作用是判断指令是否有要写入寄存器

7)rf_waddr 宽度为 5 位，表示指令写入的目的寄存器地址

8)sel_rf_res 宽度为 1 位，访存阶段使用，判断指令是运算类型或 load 类型

9)rf_rdata1, rf_rdata2 宽度为 32 位，表示从 regfile 读出的数据

10)is_in_delayslot 为 reg 类型，表示指令是否在延迟槽中

wire inst_div,inst_divu,inst_mult,inst_multu,inst_mthi,inst_mtlo

1)宽度为 1 位，表示指令，inst_div 为有符号除，inst_divu 无符号除，inst_mult,inst_multu 表示有符号乘和无符号乘，inst_mthi 为 mthi 指令,inst_mtlo 为 mtlo 指令

wire [31:0] imm_sign_extend, imm_zero_extend, sa_zero_extend;

1)宽度都为 32 位，表示立即数符号扩展，立即数 0 扩展以及 sa 移位量

wire [31:0] alu_src1, alu_src2;

wire [31:0] alu_result, ex_result;

1)alu_src1, alu_src2 宽度为 32 位，表示 alu 运算的两个操作数

2)alu_result, ex_result 宽度为 32 位，alu_result 表示 alu 计算后的结果，ex_result 表示执行阶段得到的结果

wire [63:0] mul_result;

wire mul_signed;

wire [31:0] muldata1;

wire [31:0] muldata2;

1)mul_result 宽度为 64 位，表示乘法的结果

2)mul_signed 宽度为 1 位，判断是否为有符号的乘法运算

3)muldata1 宽度为 32 位，表示乘法源操作数 1

4)muldata2 宽度为 32 位，表示乘法源操作时 2

wire [63:0] div_result;

wire div_ready_i, mul_ready_i;

reg stallreq_for_div, stallreq_for_mul;

1)div_result 宽度为 64 位，表示除法的结果

2)div_ready_i, mul_ready_i 宽度为 1 位，表明除法与乘法是否结束

3)stallreq_for_div, stallreq_for_mul 宽度为 1 位，表明除法与乘法的暂停信号

reg [31:0] div_opdata1_o;

reg [31:0] div_opdata2_o;

reg div_start_o;

reg signed_div_o;

1)div_opdata1_o 宽度为 32 位，表示被除数

2)div_opdata2_o 宽度为 32 位，表示除数

3)div_start_o 宽度为 1 位，表示是否进行除法运算

4)signed_div_o 宽度为 1，表示是否是有符号除法

reg [31:0] mul_opdata1_o;

```
reg [31:0] mul_opdata2_o;
reg mul_start_o;
reg signed_mul_o;
wire hiwe,lowe;
wire [31:0]hidata;
wire [31:0]lodata;
```

1) mul_opdata1_o 宽度为 32 位，表示被乘数

2) mul_opdata2_o 宽度为 32 位，表示乘数

3)mul_start_o 宽度为 1 位，表示是否进行乘法运算

4)signed_mul_o 宽度为 1 位，表示是否有符号乘法

5)hiwe,lowe 宽度为 1 位，表示指令是否要写入 hi 与 ho 寄存器

6)hidata 宽度为 32 位，表示写入 hi 寄存器的数据值

7)lodata 宽度为 32 位，表示写入 lo 寄存器的数据值

```
always @ (posedge clk) begin
     if (rst) begin
          id_to_ex_bus_r <= `ID_TO_EX_WD'b0;
     end
     else if (stall[2]==`Stop && stall[3]==`NoStop) begin
          id_to_ex_bus_r <= `ID_TO_EX_WD'b0;
     end
     else if (stall[2]==`NoStop) begin
          id_to_ex_bus_r <= id_to_ex_bus;
     end
 end
```

1)此功能块是对变量 id_to_ex_bus_r 的赋值，意思为在 rst 为高电平时，表示复位信号有效时,将 id_to_ex_bus_r 赋予初值,是宽度为`ID_TO_EX_WD'位的 0，对于暂停信号的处理，在 ID 模块的说明时已经给出含义，因此这里当 stall[2]为 Stop，stall[3]为 Nostop 时，表示译码阶段暂停而执行阶段继续，使用空指令作为下一个周期进入执行阶段的指令，stall[2]为 Nostop 时，译码后进入执行阶段 id_to_ex_bus_r <= id_to_ex_bus

```
assign is_lw = (inst[31:26] == 6'b100011);
assign ex_id_we =(is_lw?1'b0:rf_we);
```

1)判断指令是否为 lw，指令第 26 位至 31 位若为 6'b100011 则指令为 lw 指令，

2)三元运算符，若指令为 lw 指令，说明需要暂停则赋值为 0，若不为 lw 指令，则将 rf_we 赋值给 ex_id_we,表明有指令要写入寄存器

```
assign imm_sign_extend = {{16{inst[15]}},inst[15:0]};
```

```
assign imm_zero_extend = {16'b0, inst[15:0]};
assign sa_zero_extend = {27'b0,inst[10:6]};
```

1)imm_sign_extend 表示符号拓展，在指令的 0-15 位立即数基础上符号位拓展至 32 位

2)imm_zero_extend 表示 0 拓展，在指令的 0-15 位立即数基础上进行 0 的填充至 32 位

3)sa_zero_extend 表示对 sa 的 0 拓展，在指令的 6-10 位的 sa 立即数基础上进行 0 填充至 32 位

```
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;
```

1）利用三元运算符，分别给 alu_src1 与 alu_src2 进行赋值，判断 alu 的第一个操作数与第二个操作数的位置

2）比如对 alu_src1 的三元运算，判断 sel_alu_src1[1]是否为 1’ b1，若是，则为 pc 值，因此将 ex_pc 赋值给 alu_src1，否则继续判断并进行赋值

```
assign ex_result = alu_result ;
assign data_sram_addr = ex_result;
assign data_sram_en = data_ram_en;
assign  data_sram_wen  =  (sl==4'b0111  &&  ex_result[1:0]  ==  2'b00
&&data_ram_wen == 1'b1)? 4'b0001 .................
   :(sl==4'b0010&&data_ram_wen == 1'b1)  ? 4'b1111
   : 4'b0000;
assign data_sram_wdata =(data_sram_wen==4'b1111) ? rf_rdata2
   :(data_sram_wen==4'b0001) ? .................
   :32'b0;
```

1）将 alu 处理后的结果即执行阶段后得到的结果赋给 ex_result，可能是运算型指令得到的运算结果，也可能是存取型指令的访存地址

2）即 EX 段结果赋给内存地址 data_sram_addr

3)将 ID 模块指令判断是否需要从内存中读取数据的信号赋值给 data_sram_en，输出给内存模块作为读信号

4)根据是否是 store 型指令及具体是哪一条 store 指令和访存地址最低两位的内容确定对内存的写信号，控制要写入的字节

5)根据对内存的写信号确定要写入的字节的位数，比如 4’ b0001 表示要写入最低位的一个字节，从而确定最终写入内存的数据(其余位补 0)

```
assign ex_to_mem_bus = {
        sl,
        ex_pc,
        sel_rf_res,
        rf_we,
        rf_waddr,
        ex_result
    };
```

1)将 EX 段信号组打包发送给 MEM 段

```
assign mul_signed =inst_mult?1:0;
assign muldata1 =(inst_mult|inst_multu)?rf_rdata1:32'b0;
assign muldata2 =(inst_mult|inst_multu)?rf_rdata2:32'b0;
```

1)表示判断是否有符号乘法标记,若有则将 mul_signed 赋值为 1’ b1,否则为 1’ b0

2)若为乘法(有符号或无符号)则将寄存器 1 中的数据赋值给源操作数 1

3)若为乘法(有符号或无符号)则将寄存器 2 中的数据赋值给源操作数 2

```
assign stallreq_for_ex = stallreq_for_div| stallreq_for_mul;
```

1)给出信号 stallreq_for_ex 的值，若有除法与乘法会导致流水线暂停

```
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
.........
    end
```

1)对除法器进行除法前的初始化工作，判断是否需要暂停，以及除法的源操作数的赋值，是否开始以及有无符号的除法

```
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
.........
end
```

1)对除法器进行乘法前的初始化工作，判断是否需要暂停，以及乘法的源操作数的赋值，是否开始以及有无符号的乘法

```
assign hiwe =inst_mthi|inst_mult|inst_multu|inst_div|inst_divu;
assign lowe =inst_mtlo|inst_mult|inst_multu|inst_div|inst_divu;
assign hidata =(inst_div|inst_divu)?div_result[63:32]:
               (inst_mult|inst_multu)?mul_result[63:32]:
                inst_mthi?rf_rdata1:
                 32'b0;
assign lodata =(inst_div|inst_divu)?div_result[31:0]:
               (inst_mult|inst_multu)?mul_result[31:0]:
                inst_mtlo?rf_rdata1:
                 32'b0;
```

1) 判断是否要写入 hi 寄存器，若指令为 mthi 或 mult 或 multu 或 div 或是 divu 则将写入 hi 寄存器的信号赋值为 1

2) 判断是否要写入 lo 寄存器，若指令为 mtlo 或 mult 或 multu 或 div 或是 divu 则将写入 lo 寄存器的信号赋值为 1

3) 对 hi 寄存器的数据进行赋值，判断是否为除法，若为除法则将除法结果赋值给 hidata，若不为除法继续判断是否为乘法以此类推，若不是这些指令，则赋值为 32b’ 0

4) 与 3)类似不再赘述

```
assign ex_hilo ={
        hiwe,
        lowe,
        hidata,
        lodata
        };
```

1)将 ex_hilo 信号组按位赋值给各变量

3. 功能模块说明：

Alu

算术逻辑单元是由与门和或门构成的算术逻辑单元。给定 alu 运算的两个操作数以及操作类型，可以返回操作的结果。该 alu 模块内部实现了加、减、与等十二种算术、逻辑与移位运算。

Mymul（乘法器是自己实现的）

```
module mymul (
  input wire clk,
  input wire rst,
  input wire mul_signed,
  input wire [31:0] ina,
  input wire [31:0] inb,
  input wire start_i,
  output reg [63:0] result_o,
  output reg ready_o
);
```

1）clk 时钟信号，rst 复位信号，宽度为 1 位

2）mul_signed 宽度为 1 位，判断是否为有符号乘法

3）ina,inb 宽度为 32 位，表示乘数与被乘数

4）start_i 宽度为 1 位，判断乘法是否开始

5）result_o 宽度为 64 位，表示乘法器运算结果

6）ready_o 宽度为 1 位，表示乘法是否结束

```
reg [5:0] cnt;
reg[63:0] middle;
reg[63:0] wyq;
reg [1:0] state;
reg[31:0] temp_op1;
reg[31:0] temp_op2;
```

1）cnt 宽度为 6 位，记录进行了几轮

2）middle 宽度为 64 位，作为中间变量，记录乘法过程及结果

3）wyq 宽度为 64 位，表示位移器

4）state 宽度为 1 位，表示乘法器处于的状态

5）temp_op1 宽度为 32 位，表示运算过程中的被乘数

6）temp_op2 宽度为 32 位，表示运算过程中的乘数

```
always @ (posedge clk) begin
    if (rst) begin
        state <= `MulFree;
        result_o <= {`ZeroWord,`ZeroWord};
        ready_o <= `MulResultNotReady;
```

1）复位信号开始时，将乘法器状态 state 与乘法结果和判断乘法是否结束的信号初始化

```
end else begin
          case(state)

              `MulFree: begin
                 if (start_i == `MulStart) begin
                        state <= `MulOn;
                        cnt <= 6'b000000;
                        if(mul_signed == 1'b1 && ina[31] == 1'b1) begin
                            temp_op1 = ~ina + 1;
                        end else begin
                            temp_op1 = ina;
                        end
                        if (mul_signed == 1'b1 && inb[31] == 1'b1 ) begin
                            temp_op2 = ~inb + 1;
                        end else begin
                            temp_op2 = inb;
                        end

                    wyq <= {`ZeroWord,temp_op1};
                    middle <= {`ZeroWord,`ZeroWord};

                    end else begin
                        ready_o <= `MulResultNotReady;
                        result_o <= {`ZeroWord, `ZeroWord};
                    end
                end
```

1）首先判断乘法器的状态，若乘法器空闲，则表明此时乘法器可以进行乘法运算，开始乘法运算，将乘法器状态 state 设为 ON，并将乘法轮数设为 0，接下来判断是否为有符号乘法和被乘数与乘数的正负，若为有符号乘法且乘数或被乘数为负数，则对其进行取反码加 1 赋值给被乘数 temp_op1 或 temp_op2，否则直接将 ina 与 inb 赋值即可

2）将 wyq 与 middle 初始化，middle 记录被乘数移位后的结果，wyq 记录被乘数移位的过程，将被乘数扩展为 64 位

3）若乘法器不开始进行乘法运算，乘法器未结束，乘法器结果 result_o 初始化为 64 位的 0

```
`MulOn: begin
       if(cnt != 6'b100000) begin
              if (temp_op2[0] == 1'b1) begin
                  middle <= wyq + middle;
                  wyq = wyq<<1;
                  temp_op2 = temp_op2>>1;
              end else begin
                  wyq = wyq<<1;
                  temp_op2 = temp_op2>>1;
              end
                  cnt <= cnt +1;
       end else begin
                  if ((mul_signed == 1'b1) && ((ina[31] ^ inb[31]) ==
1'b1)) begin
                      middle <= ~middle + 1;
                  end
                  state <= `MulEnd;
                  cnt <= 6'b000000;
                     end
              End
```

1）此段表示乘法器状态为 ON 下，已经开始做乘法运算，当运算次数不大于等于 6’b100000 次则继续运行

2）进行判断，乘数的最低位若为 1 则将被乘数加到中间状态变量 middle 中，移位器向左移动一位，乘数向右移动一位保证取到乘数的最低位，若乘数的最低位不为 1 则只进行移动，不进行相加处理，最后将乘法次数加 1

3）如果为符号乘法且最后乘法结果为负数，则将中间状态变量 middle 取反码并加 1，最后将状态设为乘法结束，乘法次数设为 0

```
`MulEnd: begin
    result_o <= middle;
    ready_o <= `MulResultReady;
    if (start_i == `MulStop) begin
    state <= `MulFree;
    ready_o <= `MulResultNotReady;
    result_o <= {`ZeroWord, `ZeroWord};
```

1) 表示当乘法器状态为结束时，将中间状态变量 middle 赋给最后的结果 result_o，将 ready_o 赋值为结束，如果乘法器结束，则将状态，乘法器是否结

束信号以及乘法器结果初始化，以进行下一次的乘法运算。
(除法模块已给且实现思路类似，不再赘述)

5.4 MEM 段：

1. 整体功能说明：

访存阶段需要通过执行阶段 EX 传递过来的信息，根据具体的加载、存储指令来对数据存储器 RAM 进行读/写操作(本 CPU 写入内存操作在 EX 段完成)。访存阶段需要通过执行阶段 EX 传递过来的信息，根据具体的加载、存储指令来对数据存储器 RAM 进行读/写操作并准备将访存阶段的运算结果在下一个时钟传递到回写阶段。MEM 段是一个存储器访问指令将数据从存储器中读出，或者写入存储器的过程。

2. 各个端口以及各个信号介绍：

```
module MEM(
    input wire clk,
    input wire rst,
    input wire [`StallBus-1:0] stall,
    input wire [`EX_TO_MEM_WD-1:0] ex_to_mem_bus,
    input wire [31:0] data_sram_rdata,
    output wire [`MEM_TO_WB_WD-1:0] mem_to_wb_bus
);
```

1）首先接口 rst 与 clk 分别为宽度为 1 的复位信号与时钟信号，stall 为宽度 6 位暂停信号不再赘述，ex_to_mem_bus 宽度为 75，是由 EX 模块传递给 MEM 模块的信号及数据的集合

2）data_sram_rdata 宽度为 32 位，表示写入储存器的数据值

3）mem_to_wb_bus 宽度为 70 位，表示由 MEM 模块传输到 WB 段的信号组

```
reg [`EX_TO_MEM_WD-1:0] ex_to_mem_bus_r;
```

1）宽度为 75，是由 EX 模块传递给 MEM 模块信号组

```
wire [31:0] mem_pc;
wire [3:0]sl;
wire sel_rf_res;
wire rf_we;
wire [4:0] rf_waddr;
wire [31:0] rf_wdata;
wire [31:0] ex_result;
wire [31:0] mem_result;
```

1）mem_pc 表示 mem 阶段的 pc 地址

2)sl 宽度为 4 位，区分不同的存取指令

3)sel_rf_res 宽度为 1 位，访存阶段使用，判断指令是运算类型或 load 类型
4)rf_we 宽度为 1 位，作用是判断指令是否有要写入寄存器
5)rf_waddr 宽度为 5 位，表示指令执行写入的目的寄存器地址
6)rf_wdata 宽度为 32 位，表示写回到寄存器的内容
7)ex_result 宽度为 32 位，表示执行阶段得到的结果
8)mem_result 宽度为 32 位，mem 阶段的结果

```
assign mem_result = data_sram_rdata;
assign rf_wdata = (sl==4'b0001 && sel_rf_res==1'b1) ? mem_result
...................
: ex_result;
```

1）从内存中读取的数据值作为 MEM 阶段的结果
2）三元运算符，若指令是运算型指令，则写到寄存器中的数据为 EX 阶段得到的结果；若指令是 load 型指令，则根据具体的指令和访存地址的最低两位选择从内存中读取数据的有效字节和扩展方式(访存的时候先统一读再细分)。比如对 lb 指令，当地址后两位为 2’ b00 时，选择数据的最低一个字节的数据并进行符号扩展得到最终的结果。

```
assign mem_to_wb_bus = {
    mem_pc,
    rf_we,
    rf_waddr,
    rf_wdata
    };
```

1）将 MEM 段的信号组打包，传给下一阶段 WB 段

5.5 WB 段：

1.整体功能说明：

WB 段作为写回阶段，将运算结果保存到目的寄存器，写回(Write-Back)是指将指令执行的结果写回通用寄存器组的过程。如果是普通运算指令，该结果值来自于“执行”阶段计算的结果；如果是存储器读指令，该结果来自于“访存”阶段从存储器中读取出来的数据。

2. 各个端口及信号的介绍：

```
module WB(
    input wire clk,
    input wire rst,
    input wire [`StallBus-1:0] stall,
    input wire [`MEM_TO_WB_WD-1:0] mem_to_wb_bus,
    output wire [`WB_TO_RF_WD-1:0] 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
);
```

1）首先接口 rst 与 clk 分别为宽度为 1 的复位信号与时钟信号，stall 为宽度 6 位暂停信号不再赘述。

2）mem_to_wb_bus 即是 MEM 模块传输到 WB 段信号组

3）wb_to_rf_bus 是由 WB 阶段写回到寄存器信号组

4）debug_wb_pc 宽度为 32 位，表示 WB 段的 pc 地址

5）debug_wb_rf_wen 宽度为 4 位，表示是否写入寄存器

6）debug_wb_rf_wnum 宽度为 5 位，表示写入寄存器的地址

7）debug_wb_rf_wdata 宽度为 32 位，表示写入寄存器的数据值

```
reg [`MEM_TO_WB_WD-1:0] mem_to_wb_bus_r;
```

1)宽度为 70，是由 MEM 模块传递给 WB 模块的信号组

```
wire [31:0] wb_pc;
wire rf_we;
wire [4:0] rf_waddr;
wire [31:0] rf_wdata;
```

1)wb_pc 宽度为 32 位，表示 WB 段的 pc 地址

2)rf_we 宽度为 1 位，表示判断是否写入寄存器的信号

3)rf_waddr 宽度为 5 位，表示写入寄存器的地址

4)rf_wdata 宽度为 32 位，表示写入寄存器的数据值

```
assign debug_wb_pc = wb_pc;
assign debug_wb_rf_wen = {4{rf_we}};
assign debug_wb_rf_wnum = rf_waddr;
assign debug_wb_rf_wdata = rf_wdata;
```

1)将 pc 地址赋值给输出端口 debug_wb_pc

2)将判断是否写入寄存器的信号赋值给输出端口 debug_wb_rf_wen

3)将写入寄存器的地址赋值给输出端口 debug_wb_rf_wnum

4)将写入寄存器的内容赋值给输出端口 debug_wb_rf_wdata，以上均为调试用

```
always @ (posedge clk) begin
        if (rst) begin
            mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
        end
        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
```

1)此功能块是对变量 mem_to_wb_bus_r 的赋值，意思为在 rst 为高电平时，表示复位信号有效时，将 mem_to_wb_bus_r 赋予初值，宽度为`MEM_TO_WB_WD'位的 0，对于暂停信号的处理，在 ID 模块已经给出含义，当 stall[4]为 Stop，stall[5]为 Nostop 时，表示访存阶段暂停而回写阶段继续，使用空指令作为下一个周期进入执行阶段的指令，stall[4]为 Nostop 时，访存后进入回写阶段

```
mem_to_wb_bus_r <= mem_to_wb_bus
```

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

1)表示 MEM 阶段到 WB 阶段的信号及数据的集合，按位赋值给各信号

```
assign wb_to_rf_bus = {
        rf_we,
        rf_waddr,
        rf_wdata
    };
```

1)表示 WB 阶段到寄存器的信号及数据的集合

5.6 其他主要模块说明：

CTRL 模块

CTRL 模块作用是接受各阶段传递过来的流水线的暂停请求信号，从而控制流水线各阶段的运行，CTRL 模块的输入来自请求暂停信号 stallreq，并对暂停请求信号进行判断，然后输出流水线暂停信号 stall，从而控制 PC 的值，以及流水线各个阶段的寄存器。

MMU 模块

CPU 发出虚拟地址(VA)到达 MMU，MMU 转换成物理地址(PA)发给硬件。

mycpu_core 模块

将内部所有的模块实例化并完成接线。

六、组员的实验感受：

颜进超：

这次实验我解决数据相关的问题，添加 forwarding 通路、添加 stall，对于课上的知识融会贯通，学习并理解了乘法器并进行添加了，对于 HI、LO 寄存器有关的指令刚开始有些不理解，通过小组讨论和询问学长最后成功解决，很满足很有收获。

卢葛威：

这次实验让我受益匪浅。通过这次实验，我了解到了 MIPS 指令系统的基本指令，了解了这些指令不同的类型和它们的译码和执行方式。经过这次实验的实践，我对课上第三章流水线部分的内容有了更深的了解，包括流水线各段的功能和联系、各个功能模块以及如何利用 forwarding 通路或插入 stall 解决数据相关。实践出真知，将理论知识付诸实践才能真正地融会贯通，掌握的知识更完整。

王崇骁：

通过这次实验，对于书本的内容应用到现实，对于流水线的五个阶段以及指令有了进一步的认知，也清楚了数据暂停与数据相关的知识，学习并掌握了一个简单 CPU 的工作原理，收获颇多

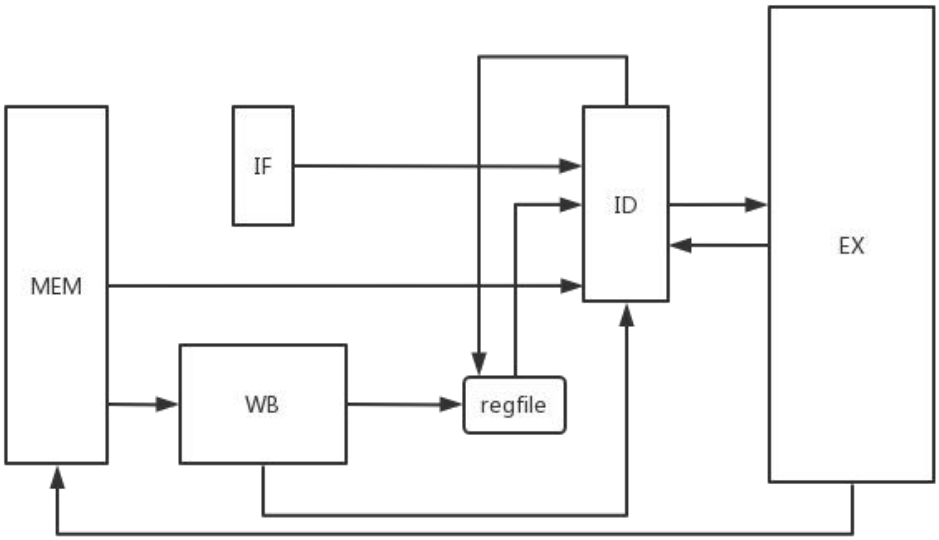
改进意见：

实验可以结合课程的进度先进行一些基础部分的搭建，比如添加一些基本的指令和逻辑。早一点开始可以早点入门和熟悉 verilog 语言，使后面深入的开发更易于开展，时间也更加充裕。

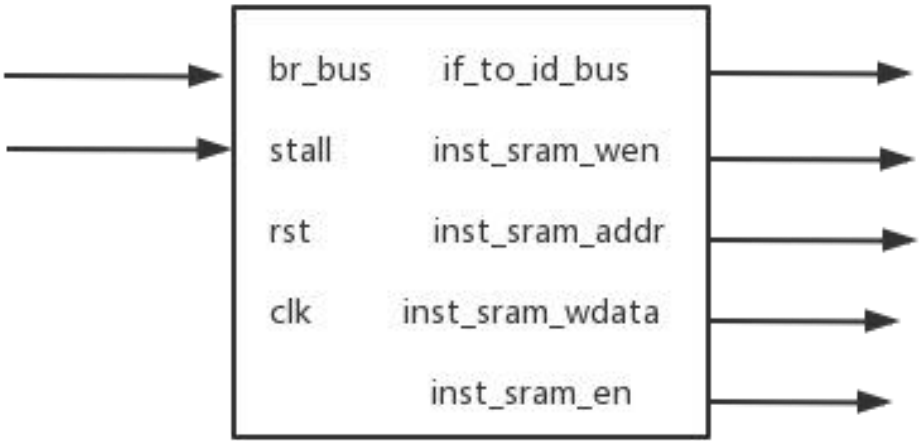
七、参考资料：

1.《自己动手写 CPU》----------雷思磊
2.《A03_“系统能力培养大赛”MIPS 指令系统规范_v1.01》
3.《计算机体系结构》
4.《计算机系统概论》

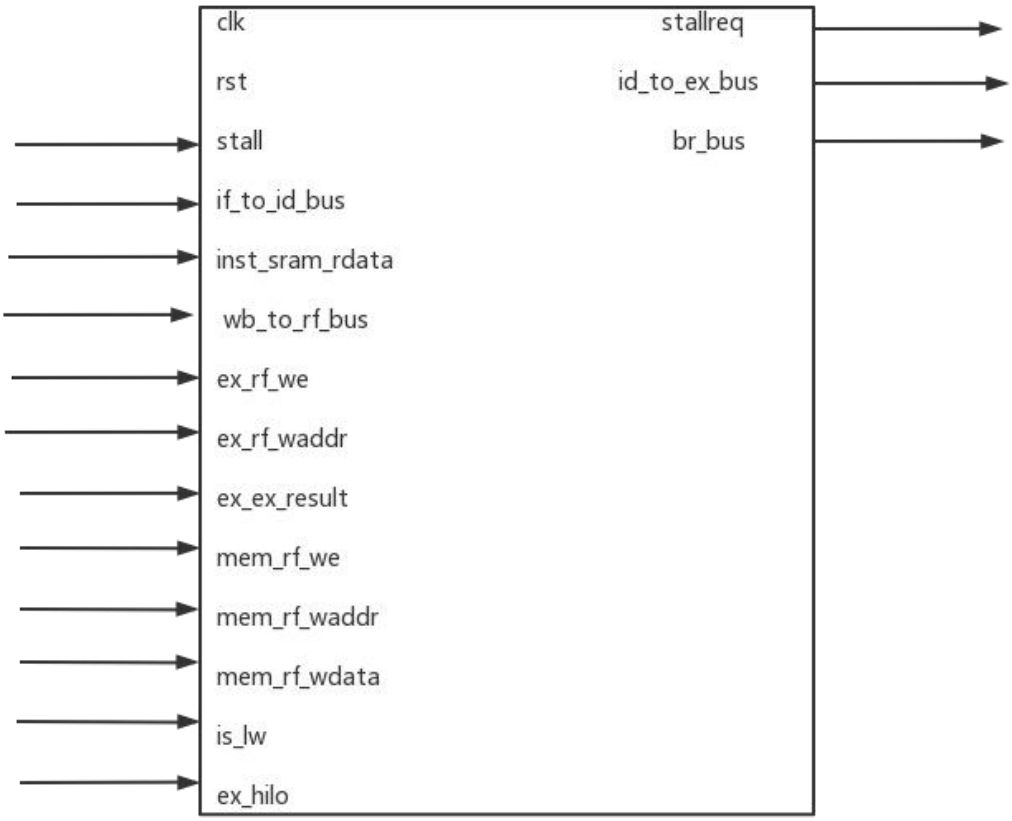
八、总流水线结构图与各阶段结构图



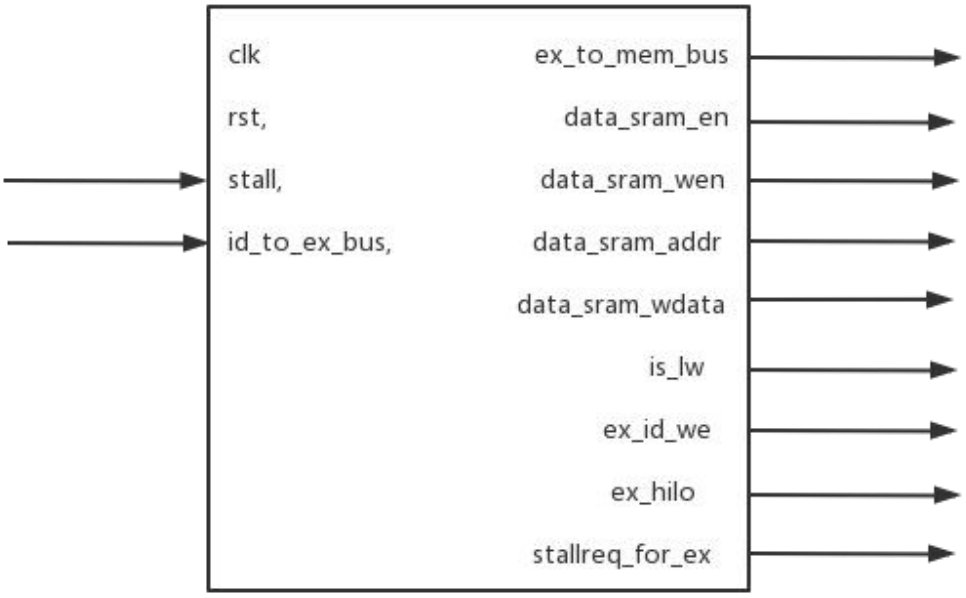
IF 段



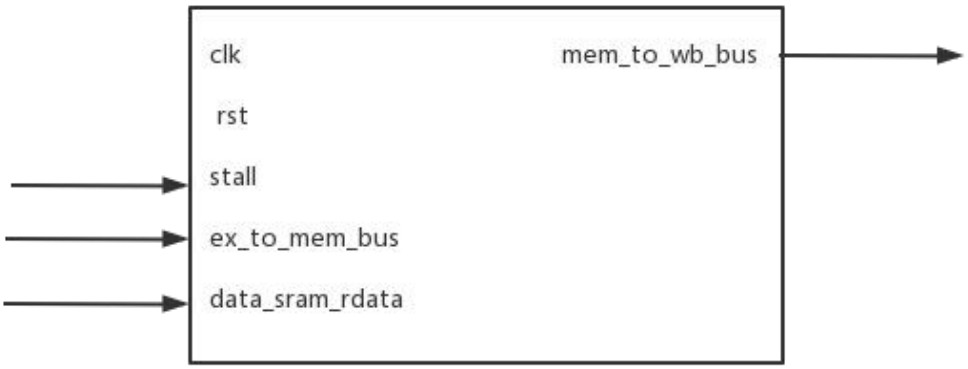
ID 段


clk
rst
stall
if_to_id_bus
inst_sram_rdata
wb_to_rf_bus
ex_rf_we
ex_rf_waddr
ex_ex_result
mem_rf_we
mem_rf_waddr
mem_rf_wdata
is_lw
ex_hilo
stallreq
id_to_ex_bus
br_bus

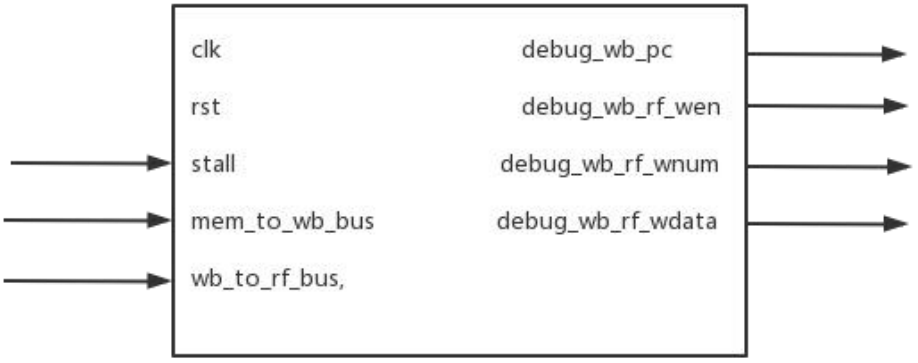
EX 段


clk
rst,
stall,
id_to_ex_bus,
ex_to_mem_bus
data_sram_en
data_sram_wen
data_sram_addr
data_sram_wdata
is_lw
ex_id_we
ex_hilo
stallreq_for_ex

MEM 段


clk
rst
stall
ex_to_mem_bus
data_sram_rdata
mem_to_wb_bus

WB 段


clk
rst
stall
mem_to_wb_bus
wb_to_rf_bus,
debug_wb_pc
debug_wb_rf_wen
debug_wb_rf_wnum
debug_wb_rf_wdata