同濟大學
TONGJI UNIVERSITY

计算机系统结构课程实验
总结报告

实验题目：动态流水线设计与性能定量分析

学号：2151769

姓名：吕博文

指导教师：陆有军

日期：2023.12.23

一、 实验环境部署与硬件配置说明

本次实验使用的实验环境是 Vivado 软件，Modelsim 软件，具体配置以及建立项目编写文件的格式和上学期完成 CPU 时的部署配置基本一致，具体实验编写代码采用 Vivado 作为主项目编译器，利用 Vivado 自带的仿真功能和 ModelSim 的仿真功能进行前仿真，在下板方面，主板使用学校统一配置的 NEXYS-4 进行下板实验，汇编代码生成 COE 文件采用 MARS 模拟器进行实现，整个动态流水线项目涉及基本 Verilog 文件（.v 后缀）的编写，IP 核（coe 文件的导入）,前仿真以及后仿真的实现。

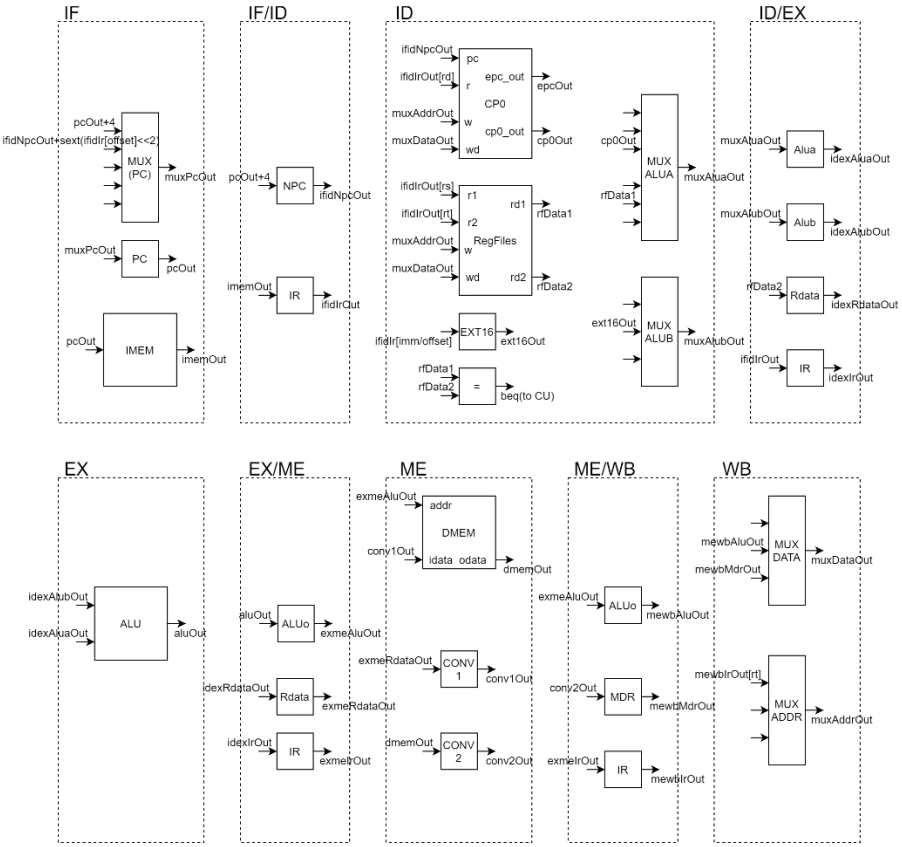
二、 实验的总体结构

实验要求：完成至少 31 条 MIPS 指令的动态流水线 CPU 设计，并支持中断。在 CPU 运行验证程序的过程中，由按键或拨动开关产生一个暂停的中断，再次按键或拨动开关结束中断，继续运行后续的运算，并在数码管上动态显示运算值。

根据实验的要求，选择进行 MIPS 指令的动态流水线 CPU 设计，根据给定验证程序的要求，设计最终的输出值存储到第 28 号寄存器中，具体下板过程中时钟信号经过分频处理之后输出，使得七段数码管规律动态显示实时计算的值，通过 J15 号开关控制清零结果，通过 L16 号开关作为外部中断信号控制 CPU 运算的中断，再次恢复 L16 号开关 CPU 继续运算。

1、动态流水线的总体结构

动态流水线指在同一时间，多功能流水线之间各段按照不同方式连接，允许同时执行多种功能。主要可以分为五个阶段 IF-ID-EX-ME-WB，还有各个阶段之间的流水寄存器，整个动态流水线的数据通路如下：



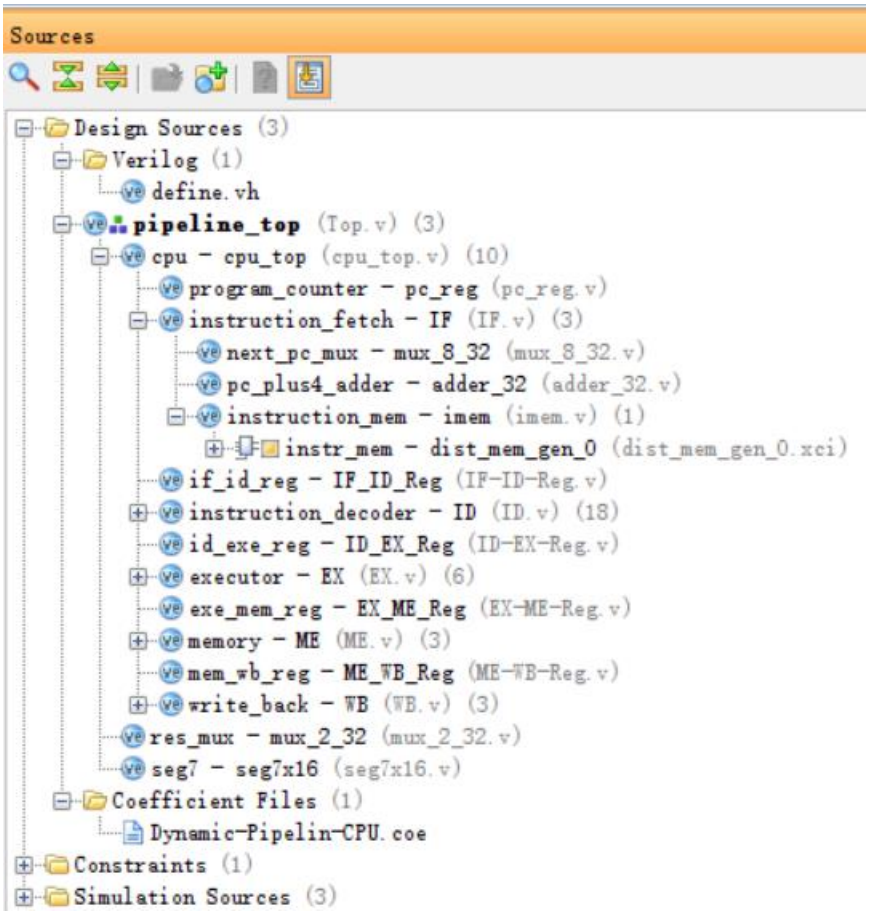
三、 总体架构部件的解释说明

动态流水线主要分为五个阶段，各个 1 所做的主要工作如下：

取指阶段（IF）：从指令存储器读出指令，同时确定下一条指令地址。
译码阶段（ID）：对指令进行译码，确定指令的操作类型以及操作数的值，根据指令的类型生成控制信号。
执行阶段（EX）：执行指令，例如算术运算，逻辑运算或者数据传输
访存阶段（ME）：执行需要访存的指令，例如 load 或者 store 指令。
写回阶段（WB）：将指令的结果写回寄存器文件或者内存。

动态流水线总体结构部件的解释说明

在设计动态流水线时，我们需要考虑可能出现的数据相关和冲突，本次实验项目中，对于数据相关，采用数据前推和必要时暂停的解决方法，对于控制相关，采用提前分支判断和延迟槽的解决办法，整个模块层次图如下：



下面分别对几个重要模块进行详细解释说明：

（1） 项目顶层模块

项目顶层模块，负责调用 CPU 模块进行 CPU 的运行，负责选择结果输出，分频以及调用七段数码管模块显示运算值。

```
//整个动态流水线的顶层程序
module pipeline_top(
    input clk,
```

```
    input rst,
    input switch_res,// 选择输出结果信号
    input stop,//外部中断信号
    //output [31:0]res,//输出的结果
    output [7:0] o_seg,//七段数码管参数
    output [7:0] o_sel //七段数码管参
数


    );
```

（2） CPU 模块

负责整个动态流水线 CPU 的运行，输入时钟信号，复位信号，以及外部中断信号，输出 pc 值，指令以及计算结果。

```
module cpu_top(
    input clk,
    input rst,
    input stop, //外部中断信号，stop = 1 表示外部中断
    output [31:0] instruction,
    output [31:0] pc,
    output [31:0] reg28,    //存储 c[i]+d[i]的结果
    output [31:0] reg29     //未使用到的寄存器，值为 0
    );
```

（3） IF 阶段

取址指令，确定下一条执行的指令内容。

```
module IF(
    input [31:0] pc,
    input [31:0] cp0_pc,  // 中断跳转地址
    input [31:0] b_pc,   // beq bne bgez 跳转地址
    input [31:0] r_pc,  // jr jalr 跳转地址
    input [31:0] j_pc,  // j jal 跳转地址
    input [2:0] pc_mux_sel,
    output [31:0] npc,  // next pc
    output [31:0] pc4,  // pc + 4
    output [31:0] instruction
    );
```

（4） IF-ID 阶段

IF-ID 中间的流水寄存器，用于存放取出的指令和 NPC

```
//根据 IF-ID 中间的流水寄存器的一些情况决定是否要对 ID 阶段使用的 PC 或指令做出改变
module IF_ID_Reg(
```

```
    input clk,
    input rst,
    input [31:0] if_pc4,     // IF 阶段的 PC + 4
    input [31:0] if_instruction,  // IF 阶段的指令
    input stall,             // 是否暂停流水线的信号
    input is_branch,         // 是否是分支指令的信号
    output reg [31:0] id_pc4,     // ID 阶段使用的 PC + 4
    output reg [31:0] id_instruction  // ID 阶段使用的指令
    );
```

（5） ID 阶段

指令译码，该阶段模块包括寄存器堆，控制单元，Hi 寄存器，Lo 寄存器以及各种控制分支的多路选择器，输入包括 WB 阶段传回的数据信号，IF 前推的值，EX 和 ME 产生的值，输出各类控制信号。

```
module ID(
    input clk,
    input rst,
    input [31:0] pc4,
    input [31:0] rf_wdata,    //Regfile 写数据
    input [31:0] hi_wdata,
    input [31:0] lo_wdata,
    input [31:0] instruction,
    input rf_wena,
    input hi_wena,
    input [4:0] rf_waddr,  // Regfile 写地址

    input lo_wena,

    // Data from EXE
    input [4:0] exe_rf_waddr,
    input exe_rf_wena,

    input [31:0] exe_mul_hi,  // 乘法结果高位
    input [31:0] exe_mul_lo,  // 乘法结果低位
    input [31:0] exe_div_r,  // 余数
    input [31:0] exe_div_q,  // 商
    input [31:0] exe_rs_data_out,
    input [31:0] exe_lo_out,
    input [31:0] exe_pc4,
    input [31:0] exe_clz_out,
    input exe_hi_wena,
    input exe_lo_wena,
```

```
    input [31:0] exe_alu_out,
    input [31:0] exe_hi_out,
    input [2:0] exe_rf_mux_sel,
    input [5:0] exe_op,
    input [1:0] exe_hi_mux_sel,
    input [1:0] exe_lo_mux_sel,

    input [5:0] exe_func,

    // Data from MEM
    input [4:0] mem_rf_waddr,
    input mem_rf_wena,
    input mem_hi_wena,
    input [31:0] mem_mul_hi,
    input [31:0] mem_mul_lo,
    input [31:0] mem_div_r,
    input mem_lo_wena,
    input [31:0] mem_div_q,
    input [31:0] mem_rs_data_out,
    input [31:0] mem_dmem_out,
    input [31:0] mem_lo_out,
    input [31:0] mem_pc4,
    input [31:0] mem_alu_out,
    input [31:0] mem_hi_out,
    input [1:0] mem_hi_mux_sel,
    input [31:0] mem_clz_out,
    input [1:0] mem_lo_mux_sel,
    input [2:0] mem_rf_mux_sel,

    output [31:0] id_cp0_pc,  // 中断地址

    output [31:0] id_imm,  // 立即数
    output [31:0] id_shamt,
    output [31:0] id_pc4,
    output [31:0] id_rs_data_out,
    output [31:0] id_rt_data_out,
    output [31:0] id_r_pc,
    output [31:0] id_b_pc,  // 跳转地址
    output [31:0] id_j_pc,  // Beq and Bne address

    output [4:0] id_rf_waddr,
    output [3:0] id_aluc,
    output [31:0] id_cp0_out,    //cp0 输出结果
    output [31:0] id_hi_out,     //output data of hi register
```

```
    output [31:0] id_lo_out,     //output data of lo register

    output id_mul_sign,
    output id_div_sign,
    output id_cutter_sign,
    output id_clz_ena,
    output id_dmem_ena,
    output id_hi_wena,
    output id_lo_wena,
    output id_rf_wena,
    output id_mul_ena,
    output id_div_ena,

    output id_dmem_wena,
    output id_cutter_mux_sel,
    output id_alu_mux1_sel,
    output [1:0] id_dmem_w_cs,
    output [1:0] id_dmem_r_cs,

    output [2:0] id_cutter_sel,
    output [2:0] id_rf_mux_sel,
    output [1:0] id_alu_mux2_sel,
    output [1:0] id_hi_mux_sel,
    output [1:0] id_lo_mux_sel,
    output [2:0] id_pc_mux_sel,
    output [5:0] id_op,
    output [5:0] id_func,
    output stall,
    output is_branch,
    output [31:0] reg28,
    output [31:0] reg29
    );
```

（6） ID-EX 阶段

ID 阶段和 EX 阶段之间的流水寄存器。

```
module ID_EX_Reg(
    input clk,
    input rst,
    input wena,                          // 输入写使能信号
    input [31:0] id_pc4,                // 输入来自 ID 阶段的下一条 PC 值
    input [31:0] id_rs_data_out,         // 输入来自 ID 阶段的 rs 寄存器数据
    input [31:0] id_rt_data_out,         // 输入来自 ID 阶段的 rt 寄存器数据
    input [31:0] id_imm,
```

```
input [31:0] id_shamt,
input [31:0] id_cp0_out,            // 输入来自 ID 阶段的 CP0 输出
input [31:0] id_hi_out,             // 输入来自 ID 阶段的 hi 寄存器输出
input [31:0] id_lo_out,             // 输入来自 ID 阶段的 lo 寄存器输出
input [4:0] id_rf_waddr,            // 输入来自 ID 阶段的寄存器写地址
input id_clz_ena,                   // 输入来自 ID 阶段的 clz 使能信号
input id_mul_ena,
input id_div_ena,
input id_dmem_ena,
input id_mul_sign,
input id_div_sign,
input id_cutter_sign,
input [3:0] id_aluc,                // 输入来自 ID 阶段的 ALU 控制信号
input id_rf_wena,                   // 输入来自 ID 阶段的寄存器文件写使能信号
input id_hi_wena,
input id_lo_wena,
input id_dmem_wena,                 // 输入来自 ID 阶段的数据存储器写使能信号
input [1:0] id_dmem_w_cs,           // 输入来自 ID 阶段的数据存储器写控制信号
input [1:0] id_dmem_r_cs,           // 输入来自 ID 阶段的数据存储器读控制信号
input stall,                        // 输入流水线暂停信号
input id_cutter_mux_sel,            // 输入来自 ID 阶段的截断器选择信号
input id_alu_mux1_sel,
input [1:0] id_alu_mux2_sel,
input [1:0] id_hi_mux_sel,          // 输入来自 ID 阶段的 hi 寄存器写入选择信号
input [1:0] id_lo_mux_sel,          // 输入来自 ID 阶段的 lo 寄存器写入选择信号
input [2:0] id_cutter_sel,          // 输入来自 ID 阶段的截断器选择信号
input [2:0] id_rf_mux_sel,
input [5:0] id_op,                  // 输入来自 ID 阶段的操作码
input [5:0] id_func,                // 输入来自 ID 阶段的功能码
output reg [31:0] exe_pc4,
output reg [31:0] exe_rs_data_out,
output reg [31:0] exe_rt_data_out,
output reg [31:0] exe_imm,          // 输出到 EX 阶段的立即数
output reg [31:0] exe_shamt,
output reg [31:0] exe_cp0_out,      // 输出到 EX 阶段的 CP0 输出
output reg [31:0] exe_hi_out,
output reg [31:0] exe_lo_out,
output reg [4:0] exe_rf_waddr,
output reg exe_clz_ena,             // 输出到 EX 阶段的 clz 使能信号
output reg exe_mul_ena,             // 输出到 EX 阶段的乘法器使能信号
output reg exe_div_ena,             // 输出到 EX 阶段的除法器使能信号
output reg exe_dmem_ena,
output reg exe_mul_sign,            // 输出到 EX 阶段的乘法器符号扩展信号
output reg exe_div_sign,            // 输出到 EX 阶段的除法器符号扩展信号
```

```
    output reg exe_cutter_sign,
    output reg [3:0] exe_aluc,          // 输出到 EX 阶段的 ALU 控制信号
    output reg exe_rf_wena,             // 输出到 EX 阶段的寄存器文件写使能信号
    output reg exe_hi_wena,             // 输出到 EX 阶段的 hi 寄存器写使能信号
    output reg exe_lo_wena,
    output reg exe_dmem_wena,           // 输出到 EX 阶段的数据存储器写使能信号
    output reg [1:0] exe_dmem_w_cs,
    output reg [1:0] exe_dmem_r_cs,     // 输出到 EX 阶段的数据存储器读控制信号
    output reg exe_alu_mux1_sel,
    output reg [1:0] exe_alu_mux2_sel,
    output reg exe_cutter_mux_sel,      // 输出到 EX 阶段的截断器选择信号
    output reg [1:0] exe_hi_mux_sel,
    output reg [1:0] exe_lo_mux_sel,    // 输出到 EX 阶段的 lo 寄存器写入选择信号
    output reg [2:0] exe_cutter_sel,    // 输出到 EX 阶段的截断器选择信号
    output reg [2:0] exe_rf_mux_sel,
    output reg [5:0] exe_op,            // 输出到 EX 阶段的操作码
    output reg [5:0] exe_func           // 输出到 EX 阶段的功能码
    );
```

（7） EX 阶段

EX 阶段，该模块包含了具体的计算部件例如：ALU 模块，乘法器，除法器，多路选择器等模块，接受 ID 传入的控制信号和操作数值，传出计算结果和控制信号。

```
module EX(
    input rst,
    input [31:0] pc4,
    input [31:0] rs_data_out,
    input [31:0] rt_data_out,
    input [31:0] imm,
    input [31:0] shamt,
    input [31:0] cp0_out,
    input [31:0] hi_out,
    input [31:0] lo_out,
    input [4:0] rf_waddr,
    input [3:0] aluc,
    input mul_ena,
    input div_ena,
    input clz_ena,
    input dmem_ena,
    input mul_sign,
    input div_sign,
    input cutter_sign,
    input rf_wena,
```

```
    input hi_wena,
    input lo_wena,
    input dmem_wena,
    input [1:0] dmem_w_cs,
    input [1:0] dmem_r_cs,
    input cutter_mux_sel,
    input alu_mux1_sel,
    input [1:0] alu_mux2_sel,
    input [1:0] hi_mux_sel,
    input [1:0] lo_mux_sel,
    input [2:0] cutter_sel,
    input [2:0] rf_mux_sel,
    output [31:0] exe_mul_hi,  // Higher bits of the multiplication result
    output [31:0] exe_mul_lo,  // Lower bits of the multiplication result
    output [31:0] exe_div_r,  // Remainder of the division result
    output [31:0] exe_div_q,  // Quotient of the division result
    output [31:0] exe_clz_out,
    output [31:0] exe_alu_out,  // Result of ALU
    output [31:0] exe_pc4,
    output [31:0] exe_rs_data_out,
    output [31:0] exe_rt_data_out,
    output [31:0] exe_cp0_out,
    output [31:0] exe_hi_out,
    output [31:0] exe_lo_out,
    output [4:0] exe_rf_waddr,
    output exe_dmem_ena,
    output exe_rf_wena,
    output exe_hi_wena,
    output exe_lo_wena,
    output exe_dmem_wena,
    output [1:0] exe_dmem_w_cs,
    output [1:0] exe_dmem_r_cs,
    output exe_cutter_sign,
    output exe_cutter_mux_sel,
    output [2:0] exe_cutter_sel,
    output [1:0] exe_hi_mux_sel,
    output [1:0] exe_lo_mux_sel,
    output [2:0] exe_rf_mux_sel
    );
```

（8） EX-ME 阶段

EX-ME 阶段的流水寄存器

```
module EX_ME_Reg(
```

```
    input clk,
    input rst,
    input wena,
    input [31:0] exe_mul_hi,
    input [31:0] exe_mul_lo,
    input [31:0] exe_div_r,
    input [31:0] exe_div_q,
    input [31:0] exe_clz_out,
    input [31:0] exe_alu_out,
    input [31:0] exe_pc4,
    input [31:0] exe_rs_data_out,
    input [31:0] exe_rt_data_out,
    input [31:0] exe_cp0_out,
    input [31:0] exe_hi_out,
    input [31:0] exe_lo_out,
    input [4:0] exe_rf_waddr,
    input exe_dmem_ena,
    input exe_cutter_sign,
    input exe_rf_wena,
    input exe_hi_wena,
    input exe_lo_wena,
    input exe_dmem_wena,
    input [1:0] exe_dmem_w_cs,
    input [1:0] exe_dmem_r_cs,
    input exe_cutter_mux_sel,
    input [1:0] exe_hi_mux_sel,
    input [1:0] exe_lo_mux_sel,
    input [2:0] exe_cutter_sel,
    input [2:0] exe_rf_mux_sel,
    output reg [31:0] mem_mul_hi,
    output reg [31:0] mem_mul_lo,
    output reg [31:0] mem_div_r,
    output reg [31:0] mem_div_q,
    output reg [31:0] mem_clz_out,
    output reg [31:0] mem_alu_out,
    output reg [31:0] mem_pc4,
    output reg [31:0] mem_rs_data_out,
    output reg [31:0] mem_rt_data_out,
    output reg [31:0] mem_cp0_out,
    output reg [31:0] mem_hi_out,
    output reg [31:0] mem_lo_out,
    output reg [4:0] mem_rf_waddr,
    output reg mem_dmem_ena,
    output reg mem_cutter_sign,
```

```
    output reg mem_rf_wena,
    output reg mem_hi_wena,
    output reg mem_lo_wena,
    output reg mem_dmem_wena,
    output reg [1:0] mem_dmem_w_cs,
    output reg [1:0] mem_dmem_r_cs,
    output reg mem_cutter_mux_sel,
    output reg [1:0] mem_hi_mux_sel,
    output reg [1:0] mem_lo_mux_sel,
    output reg [2:0] mem_cutter_sel,
    output reg [2:0] mem_rf_mux_sel
    );
```

（9） ME 阶段

ME 阶段，包含存储器 DMEM 模块，选择数据模块，符号扩展模块。

```
module ME(
    input clk,
    input [31:0] mul_hi,
    input [31:0] mul_lo,
    input [31:0] div_r,
    input [31:0] div_q,
    input [31:0] clz_out,
    input [31:0] alu_out,
    input [31:0] pc4,
    input [31:0] rs_data_out,
    input [31:0] rt_data_out,
    input [31:0] cp0_out,
    input [31:0] hi_out,
    input [31:0] lo_out,
    input [4:0] rf_waddr,
    input dmem_ena,
    input rf_wena,
    input hi_wena,
    input lo_wena,
    input dmem_wena,
    input cutter_sign,
    input [1:0] dmem_w_cs,
    input [1:0] dmem_r_cs,
    input cutter_mux_sel,
    input [2:0] cutter_sel,
    input [1:0] hi_mux_sel,
    input [1:0] lo_mux_sel,
    input [2:0] rf_mux_sel,
    output [31:0] mem_mul_hi,  // Higher bits of the multiplication result
```

```
    output [31:0] mem_mul_lo,  // Lower bits of the multiplication result
    output [31:0] mem_div_r,  // Remainder of the division result
    output [31:0] mem_div_q,  // Quotient of the division result
    output [31:0] mem_clz_out,
    output [31:0] mem_alu_out,  // Result of ALU
    output [31:0] mem_dmem_out,
    output [31:0] mem_pc4,
    output [31:0] mem_rs_data_out,
    output [31:0] mem_cp0_out,
    output [31:0] mem_hi_out,
    output [31:0] mem_lo_out,
    output [4:0] mem_rf_waddr,
    output mem_rf_wena,
    output mem_hi_wena,
    output mem_lo_wena,
    output [1:0] mem_hi_mux_sel,
    output [1:0] mem_lo_mux_sel,
    output [2:0] mem_rf_mux_sel
    );
```

（10） ME-WB 阶段

ME-WB 阶段之间的流水寄存器

```
module ME_WB_Reg(
    input clk,
    input rst,
    input wena,
    input [31:0] mem_mul_hi,
    input [31:0] mem_mul_lo,
    input [31:0] mem_div_r,
    input [31:0] mem_div_q,
    input [31:0] mem_clz_out,
    input [31:0] mem_alu_out,
    input [31:0] mem_dmem_out,
    input [31:0] mem_pc4,
    input [31:0] mem_rs_data_out,
    input [31:0] mem_cp0_out,
    input [31:0] mem_hi_out,
    input [31:0] mem_lo_out,
    input [4:0] mem_rf_waddr,
    input mem_rf_wena,
    input mem_hi_wena,
    input mem_lo_wena,
    input [1:0] mem_hi_mux_sel,
    input [1:0] mem_lo_mux_sel,
```

```
    input [2:0] mem_rf_mux_sel,
    output reg [31:0] wb_mul_hi,
    output reg [31:0] wb_mul_lo,
    output reg [31:0] wb_div_r,
    output reg [31:0] wb_div_q,
    output reg [31:0] wb_clz_out,
    output reg [31:0] wb_alu_out,
    output reg [31:0] wb_dmem_out,
    output reg [31:0] wb_pc4,
    output reg [31:0] wb_rs_data_out,
    output reg [31:0] wb_cp0_out,
    output reg [31:0] wb_hi_out,
    output reg [31:0] wb_lo_out,
    output reg [4:0] wb_rf_waddr,
    output reg wb_rf_wena,
    output reg wb_hi_wena,
    output reg wb_lo_wena,
    output reg [1:0] wb_hi_mux_sel,
    output reg [1:0] wb_lo_mux_sel,
    output reg [2:0] wb_rf_mux_sel
    );
```

（11） WB 阶段

WB 阶段，数据写回阶段，包含了 Hi，Lo，RegFiles 输入数据的多路选择器，输入各类计算结果和控制信号，输出写回 ID 的信号和数据。

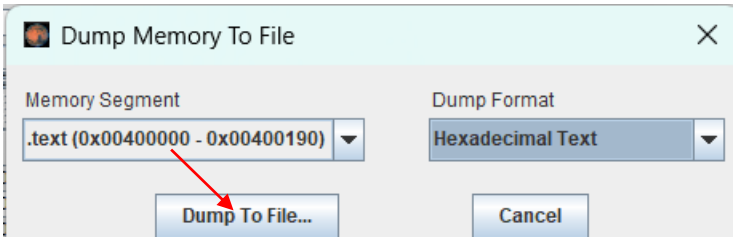
```
module WB(
    input [31:0] mul_hi,
    input [31:0] mul_lo,
    input [31:0] div_r,
    input [31:0] div_q,
    input [31:0] clz_out,
    input [31:0] alu_out,
    input [31:0] dmem_out,
    input [31:0] pc4,
    input [31:0] rs_data_out,
    input [31:0] cp0_out,
    input [31:0] hi_out,
    input [31:0] lo_out,
    input [4:0] rf_waddr,
    input rf_wena,
    input hi_wena,
    input lo_wena,
    input [1:0] hi_mux_sel,
```

```
    input [1:0] lo_mux_sel,
    input [2:0] rf_mux_sel,
    output [31:0] hi_wdata,
    output [31:0] lo_wdata,
    output [31:0] rf_wdata,
    output [4:0] wb_rf_waddr,
    output wb_rf_wena,
    output wb_hi_wena,
    output wb_lo_wena
    );
```

四、 实验仿真过程

进行前仿真测试，首先我们需要根据对应的测试要求完成汇编语言程序的编写。

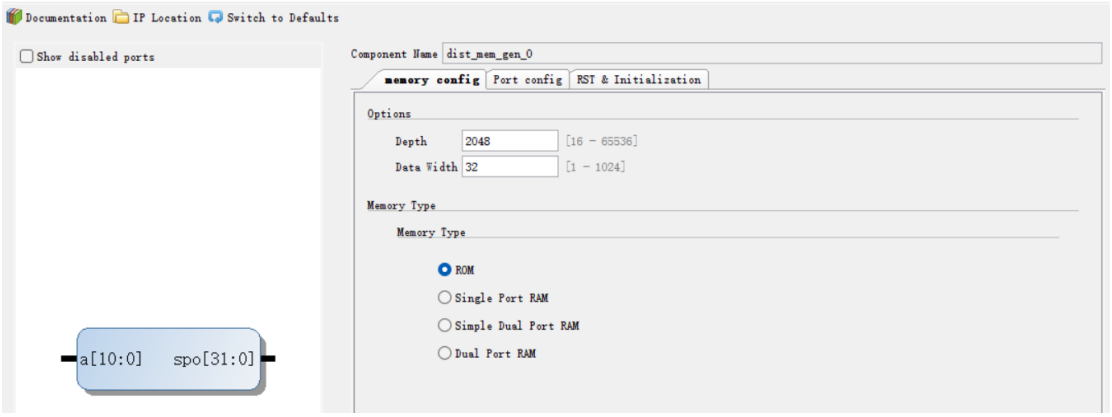
之后我们利用 MARS 进行 COE 文件的导出



导出的 COE 文件加入前两行之后即可作为 IP 核的内容导入：

```
memory_initialization_radix = 16;
memory_initialization_vector =
08100003
00000000
08100001
20020000
20030001
```

之后，我们按要求配置好 Vivado 的 IP 核



我们写出 testbench 测试程序如下：

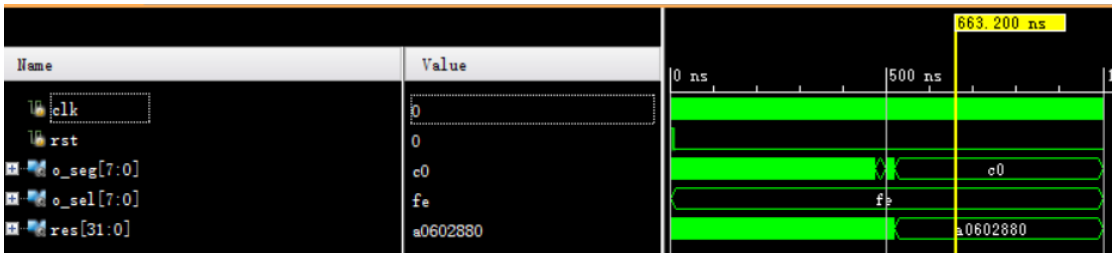
```
`timescale 1ns / 1ps
```

```
module test_bench(
    );
    reg clk,rst;
    wire [7:0] o_seg,o_sel;
    wire [31:0] res;
    initial begin
        clk = 1'b0;
        rst = 1'b1;
        #10 rst = ~rst;
    end
    always
    begin
        #0.1 clk = ~clk;
    end
    pipeline_top uut(
        .clk(clk),
        .rst(rst),
        .switch_res(0),
        .stop(0),
        .res(res),
        .o_seg(o_seg),
        .o_sel(o_sel)
    );
endmodule
```

动态流水线的仿真过程

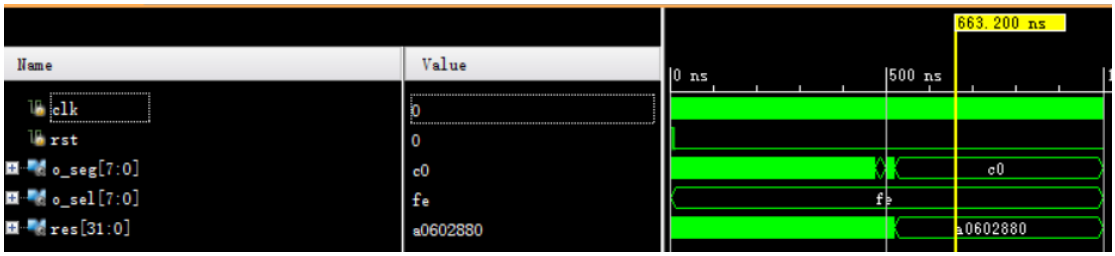
注意我们进行前仿真时需要首先把 CPU 模块传入的 clk 变为未分频之前的 clk，否则会因为时间太短而无法看到最后的计算结果。

得到如下波形图：

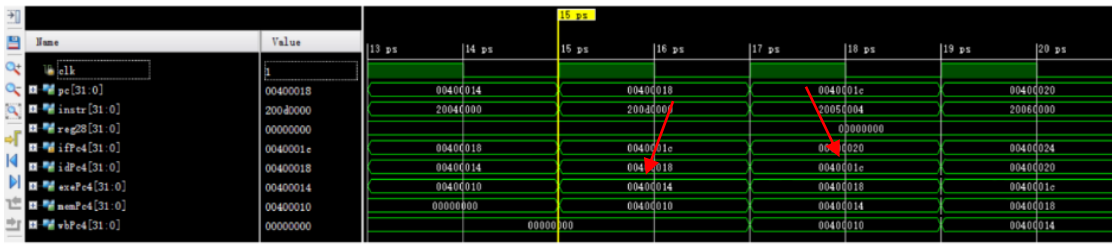


五、 实验仿真的波形图及某时刻寄存器值的物理意义

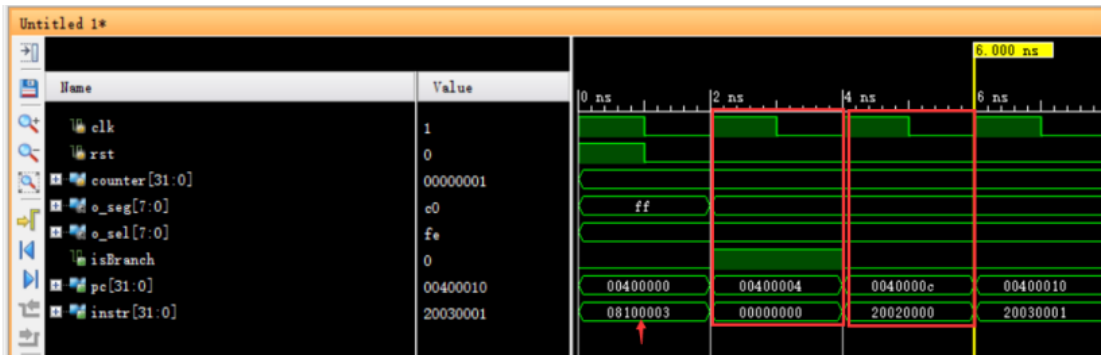
1、动态流水线的波形图及某时刻寄存器值的物理意义



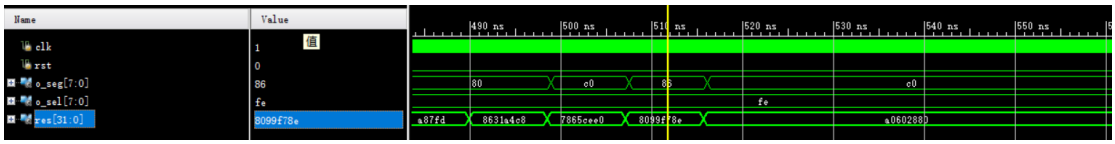
首先我们根据波形图可以看出来，res 代表着 c[i] + d[i]的实时结果，我们可以看到 res 在不断变化，知道最后的 0xA0602880,之后保持不变，根据高级程序语言计算结果，最后结果相符，可以表示前仿真最后结果正确。



我们将波形放大了看，横向来说，分别是每一级的信号输出结果，纵向来说，是每个时钟周期当前的输出，我们可以看出，每一级的 NPC 信号都成功传递到下一级。



上述波形分析的是汇编语句中的第一句 j main,属于跳转语句，分支成功转移，位于延迟槽内的指令失效，封锁其写信号，PC 值变为跳转之后的 0x0040000C。



我们通过观察寄存器中最后四次的值分别是 0x8631a4c8,0x7865cee0,0x8099f78e,0xa0602880 与最后计算出来的值相吻合。

六、 实验验算数学模型及算法程序

```
int a[m],b[m],c[m],d[m];
a[0]=0;
b[0]=1;
a[i]=a[i-1]+i;
b[i]=b[i-1]+3i;
```

$$c[i]=\begin{cases} a[i], & 0\le i\le 19 \\ a[i]+b[i], & 20\le i\le 39 \\ a[i]*b[i], & 40\le i\le 59 \end{cases}$$

$$d[i]=\begin{cases} b[i], & 0\le i\le 19 \\ a[i]*c[i], & 20\le i\le 39 \\ c[i]*b[i], & 40\le i\le 59 \end{cases}$$

我们根据实验验算数学模型编写汇编程序如下：

```
.data
A:.space 240
B:.space 240
C:.space 240
D:.space 240
E:.space 240
.text
j main
exc:
nop
j exc

main:
addi $2,$0,0    #a[i]
addi $3,$0,1    #b[i]
addi $4,$0,0    #c[i]
addi $13,$0,0   #d[i]
addi $5,$0,4    #counter
addi $6,$0,0    #a[i-1]
addi $7,$0,1    #b[i-1]
addi $10,$0,0   #flag for i<20 || i<40
addi $11,$0,240 #sum counts
addi $14,$0,3
addi $30,$0,0

# 把 0 1 0 0 ($2,...,$13) 分别存入 A B C D
lui $27,0x0000
```

```
addu $27,$27,$0
sw $2,A($27)
lui $27,0x0000
addu $27,$27,$0
sw $3,B($27)
lui $27,0x0000
addu $27,$27,$0
sw $2,C($27)
lui $27,0x0000
addu $27,$27,$0
sw $3,D($27)

# 循环
loop:
## $5(4) 除以 4 (=i) 存入 $12
srl $12,$5,2
# $6 加 i. 自此, $6 就是 a[i] 而不是 a[i-1] 了
add $6,$6,$12
# 把 a[i] 的内容存入 A[i] 中
lui $27,0x0000
addu $27,$27,$5
sw $6,A($27)
# $14 (3) 乘以 $5/4 (i) ( = 3i )
mul $15,$14,$12
# 把 $7 (b[i-1]) 的内容加上 3i, 存入 B[i]. 自此, $7 就是 b[i] 而不是 b[i-1]
add $7,$7,$15
lui $27,0x0000
addu $27,$27,$5
sw $7,B($27)
# $5 是否小于 80 (i 是否小于 20)? 记入 $10
slti $10,$5,80
# 若不是, 跳转
bne $10,1,c1

# (0<=i<=19)
# 把 $6 的内容存入 C[i] 中 (c[i] = a[i])
lui $27,0x0000
addu $27,$27,$5
sw $6,C($27)
# 把 $7 的内容存入 D[i] 中 (d[i] = b[i])
lui $27,0x0000
addu $27,$27,$5
sw $7,D($27)
addi $15,$6,0 # $15 $16 分别赋值为 c[i] d[i]
```

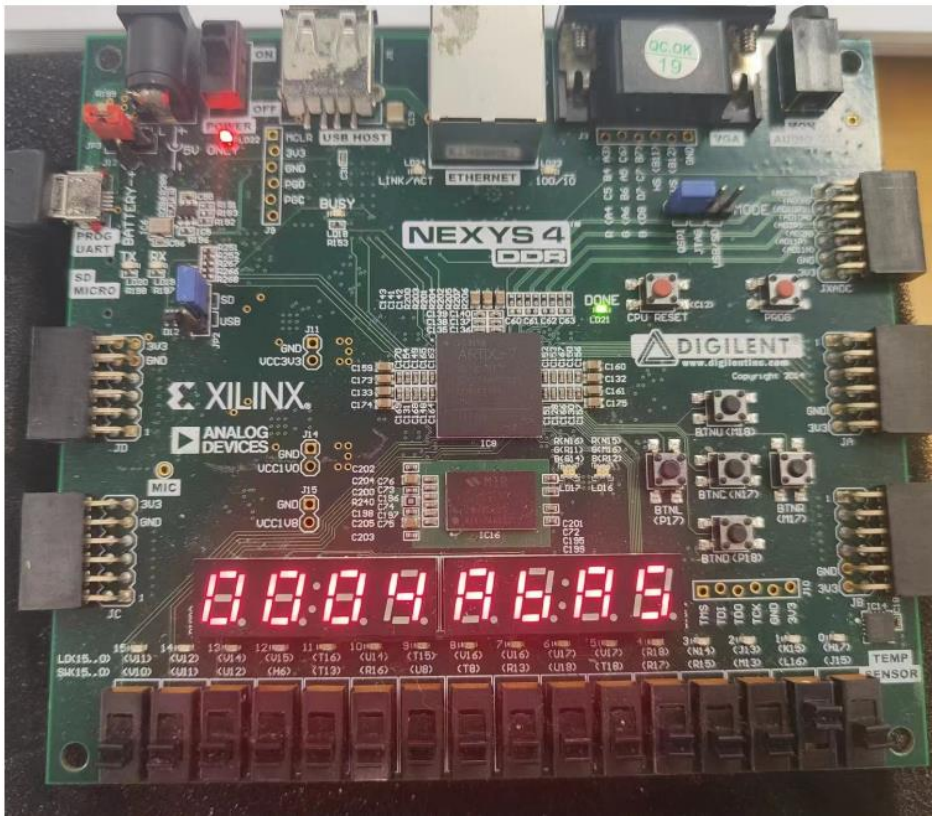
```
addi $16,$7,0
j endc
c1: # (20<=i<=39)
# i 是否小于 40 ？ 若不是，跳转到 c2
slti $10,$5,160
addi $27,$0,1
bne $10,$27,c2
# C[i] = a[i] + b[i]
add $15,$6,$7
lui $27,0x0000
addu $27,$27,$5
sw $15,C($27)
# D[i] = a[i] * b[i]
mul $16, $15,$6
lui $27,0x0000
addu $27,$27,$5
sw $16,D($27)
j endc
c2: # (i>=40)
# C[i] = a[i] * b[i]
mul $15,$6,$7
lui $27,0x0000
addu $27,$27,$5
sw $15,C($27)
# D[i] = c[i] * b[i]
mul $16,$15,$7
lui $27,0x0000
addu $27,$27,$5
sw $16,D($27)

endc:
add $28,$15,$16 #$28 = c[i] + d[i]
lui $27,0x0000
addu $27,$27,$5
sw $28,E($27) # 将 c[i] + d[i] 存入 E[i]
addi $5,$5,4 # i = i + 1
bne $5,$11,loop # i = 60 不跳转
break
# 最后 E[i] = c[i] + d[i]，可通过验证 E[59] 的正确性来验证 c[i] 和 d[i] 正确性
```

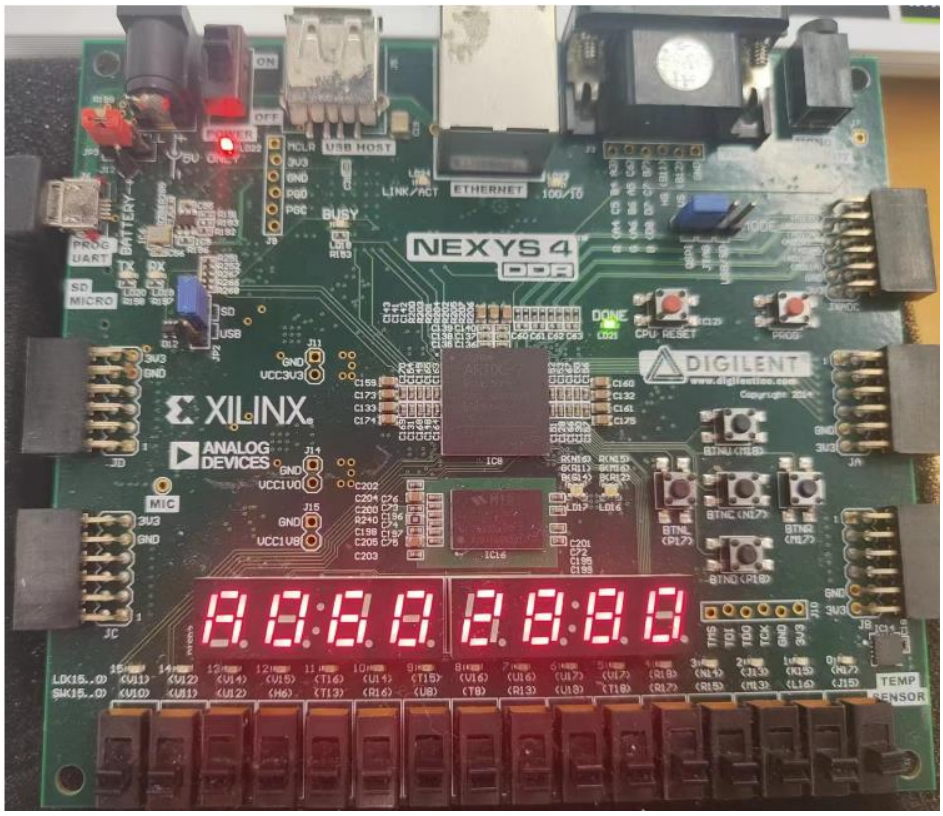
七、 实验验算程序下板测试过程与实现

下板测试时需要注意的是首先要编写好对应的 xdc 文件，要对 CPU 模块传入的时钟信号进行分频方便人眼识别。

这是运算过程中加入外部中断信号（L16 开关）时 CPU 暂停的界面。



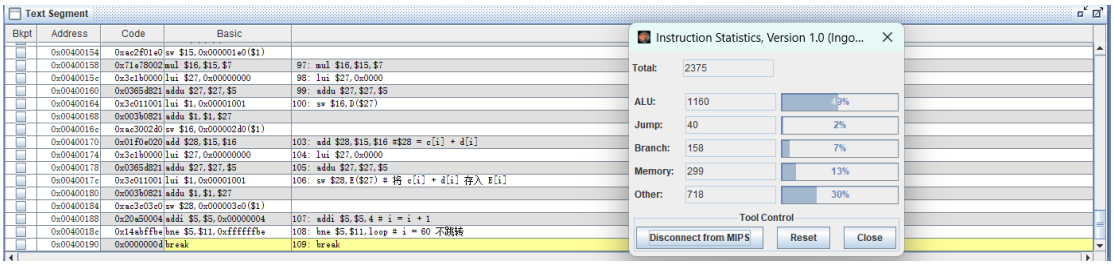
这是最终运算结果



八、 流水线的性能指标定性分析（包括：吞吐率、加速比、效率及相关与冲突分析）

1、 动态流水线的性能指标定性分析

如下图，易知汇编程序总的执行指令条数为 2375 条，根据波形图计算得出执行周期为 2530 个时钟周期。



吞吐率

吞吐率指在单位时间内流水线所完成的指令条数。本动态流水线 CPU 完成任务执行了 2375 条指令，一 共使用了 2530 个时钟周期，因此吞吐率计算为：

吞吐率 = 2375/2530 = 0.9387/时钟周期

加速比

不使用流水线和使用流水线所用的时间比称为流水线的加速比。根据 MARS 模拟器的分析结果，本次程序一共使用了 299 条访存指令，158 条分支指令 1160 条 ALU 指令，40 条跳转指令，718 条其他指令，加速比计算：

加速比 = 299*4 + 158*3 + 1160*4 + 40*2 + 718*3 / 2530 = 3.345

效率

流水线的效率指流水线中各功能段的利用率，即流水线各段处于工作时间的时空区与流水线各段总的 时空区的比值。对于本动态流水线 CPU，效率的计算如下：

效率 = 299*4 + 158*3 + 1160*4 + 40*2 + 718*3 / （2530*5）= 66.91%

冲突分析

（1）对于写后读冲突，我们通过定向技术来解决，即将结果数据从其产生的部分直接传送到所有需要它的功能部件。

（2）对于不能通过数据前向传播解决的冲突，我们进行暂停流水线的方法解决。

（3）控制相关，利用延迟槽和提前判断分支技术。在 ID 阶段进行转移条件的判断，如果满足转移条件，修改 PC 为转 移目标地址，令处于延迟槽内的指令失效，即封锁处于 ID 阶段指令的任何写有效信号，如果延迟槽内也是 分支指令，则禁止其对 PC 的修改。如果不满足转移条件，延迟槽内的指令正常执行。

九、 总结与体会

在本次实验中，我成功地完成了一个动态流水线 CPU 的设计。这是一次非常有挑战性的任务，同时也是一个充实而有趣的学习过程。以下

是我在这个实验中的一些总结和个人体会：

这个实验让我深刻理解了流水线架构的工作原理，以及它是如何通过分阶段执行指令来提高 CPU 性能的。我不仅理论上了解了流水线的优势，还亲身体验了如何在实际设计中应用这些概念。

在实验过程中，我面临了一些技术挑战，如数据冒险和流水线停顿。通过深入研究和学习，我成功地引入了一些解决方案，例如数据前推和指令重排序，以最大程度地减小冒险对性能的影响。

我深刻体会到了测试与验证在硬件设计中的关键性。通过创建多样化的测试用例，我能够验证 CPU 的正确性和性能，并及时发现和解决一些潜在的问题。

这个实验是我硬件设计领域的一次重要经验。在完成项目的过程中，我积累了大量的知识，并对计算机体系结构有了更深层次的理解。未来，我期待进一步学习和改进我的设计，也希望能够应用这些知识到更为复杂和先进的 CPU 设计中。

通过这个实验，我不仅提升了硬件设计技能，还培养了解决问题的能力和持续学习的态度。这将对我的学术和职业生涯产生积极的影响，让我更有信心面对未来的挑战。

十、 附件（所有程序）

1、 动态流水线的设计程序

```
`timescale 1ns / 1ps

module test_bench(
    );
    reg clk,rst;
    wire [7:0] o_seg,o_sel;
    wire [31:0] res;
    initial begin
        clk = 1'b0;
        rst = 1'b1;
        #10 rst = ~rst;
    end
    always
    begin
        #0.1 clk = ~clk;
    end
    pipeline_top uut(
        .clk(clk),
        .rst(rst),
        .switch_res(0),
        .stop(0),
        .res(res),
        .o_seg(o_seg),
```

```
        .o_sel(o_sel)
    );
endmodule
```

```
`timescale 1ns / 1ps

module adder_32(
    input [31:0] a,
    input [31:0] b,
    output [31:0] result
    );

    assign result = a + b;

endmodule
```

```
`timescale 1ns / 1ns

module alu(
    input [31:0] a,
    input [31:0] b,
    input [3:0] aluc,
    output [31:0] r,
    output zero,
    output carry,
    output negative,
    output overflow
    );

    parameter Addu = 4'b0000;
    parameter Add = 4'b0010;
    parameter Subu = 4'b0001;
    parameter Sub = 4'b0011;
    parameter And = 4'b0100;
    parameter Or = 4'b0101;
    parameter Xor = 4'b0110;
    parameter Nor = 4'b0111;
    parameter Lui1 = 4'b1000;
    parameter Lui2 = 4'b1001;
    parameter Slt = 4'b1011;
    parameter Sltu = 4'b1010;
    parameter Sra = 4'b1100;
```

```
    parameter Sll = 4'b1110;
    parameter Srl = 4'b1101;
    parameter Slr = 4'b1111;

    reg [32:0] result;
    reg if_same_signal;
    reg flag;

    always @ (*) begin
        case(aluc)
            Addu: result = a + b;
            Add: begin
                result = $signed(a) + $signed(b);
                if_same_signal = ~(a[31] ^ b[31]);
                flag = (if_same_signal && result[31] != a[31]) ? 1 : 0;
            end
            Subu: result = a - b;
            Sub: begin
                result = $signed(a) - $signed(b);
                if_same_signal = ~(a[31] ^ b[31]);
                flag = (~if_same_signal && result[31] != a[31]) ? 1 : 0;
            end

            And: result = a & b;
            Or: result = a | b;
            Xor: result = a ^ b;
            Nor: result = ~(a | b);

            Lui1: result = {b[15:0], 16'b0};
            Lui2: result = {b[15:0], 16'b0};

            Slt: result = ($signed(a) < $signed(b)) ? 1 : 0;
            Sltu: result =  (a < b) ? 1 : 0;

            Sra: result = $signed(b) >>> a;
            Sll: result = b << a;
            Slr: result = b << a;
            Srl: result = b >> a;
        endcase
    end

    assign r = result[31:0];
    assign carry = (aluc == Addu | aluc == Subu | aluc == Sltu | aluc == Sra | aluc == Srl
| aluc == Sll) ? result[32] : 1'bz;
```

```
    assign zero = (r == 32'b0) ? 1 : 0;
    assign negative = result[31];
    assign overflow = (aluc == Add | aluc == Sub) ? flag : 1'bz;

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"

module branch_predict(
    input clk,
    input rst,
    input [31:0] data_in1,
    input [31:0] data_in2,
    input [5:0] op,           // 操作码
    input [5:0] func,         // 功能码
    input exception,          // 异常信号
    output reg is_branch      // 输出的分支信号
    );

    always @ (*) begin
        if(rst == `RST_ENABLED)
            is_branch <= 1'b0; // 复位时，分支信号为0
        else if(op == `BEQ_OP)
            is_branch <= (data_in1 == data_in2) ? 1'b1 : 1'b0; // 如果是BEQ指令，根据数据比较结果设置分支信号
        else if(op == `BNE_OP)
            is_branch <= (data_in1 != data_in2) ? 1'b1 : 1'b0; // 如果是BNE指令，根据数据比较结果设置分支信号
        else if(op == `BGEZ_OP)
            is_branch <= (data_in1 >= 0) ? 1'b1 : 1'b0; // 如果是BGEZ指令，根据数据比较结果设置分支信号
        else if(op == `TEQ_OP && func == `TEQ_FUNC)
            is_branch <= (data_in1 == data_in2) ? 1'b1 : 1'b0; // 如果是TEQ指令，根据数据比较结果设置分支信号
        else if(op == `J_OP)
            is_branch <= 1'b1; // 如果是J指令，设置分支信号为1
        else if(op == `JAL_OP)
            is_branch <= 1'b1; // 如果是JAL指令，设置分支信号为1
        else if(op == `JR_OP && func == `JR_FUNC)
            is_branch <= 1'b1; // 如果是JR指令，设置分支信号为1
        else if(op == `JALR_OP && func == `JALR_FUNC)
```

```
            is_branch <= 1'b1; // 如果是 JALR 指令，设置分支信号为 1
        else if(exception)
            is_branch <= 1'b1; // 如果有异常信号，设置分支信号为 1
        else
            is_branch <= 1'b0; // 其他情况下，分支信号为 0
    end
endmodule
```

```
`timescale 1ns / 1ps

module control_unit(
    input is_branch,              // 输入：分支信号
    input [5:0] op,               // 输入：操作码
    input [5:0] func,             // 输入：功能码
    input [31:0] status,          // 输入：处理器状态寄存器
    input [31:0] instruction,     // 输入：指令寄存器

    output rf_wena,               // 输出：寄存器文件写使能信号
    output clz_ena,               // 输出：clz 计算使能信号
    output mul_ena,               // 输出：乘法器使能信号
    output div_ena,               // 输出：除法器使能信号
    output dmem_ena,              // 输出：数据存储器使能信号
    output hi_wena,               // 输出：hi 寄存器写使能信号
    output lo_wena,               // 输出：lo 寄存器写使能信号
    output rf_rena1,              // 输出：寄存器文件读使能信号 1
    output rf_rena2,              // 输出：寄存器文件读使能信号 2
    output dmem_wena,             // 输出：数据存储器写使能信号

    output ext16_sign,            // 输出：16 位符号扩展信号
    output cutter_sign,           // 输出：截断信号
    output [1:0] dmem_w_cs,       // 输出：数据存储器写控制信号
    output [1:0] dmem_r_cs,       // 输出：数据存储器读控制信号
    output mul_sign,              // 输出：乘法器符号扩展信号
    output div_sign,              // 输出：除法器符号扩展信号
    output [3:0] aluc,            // 输出：ALU 控制信号
    output [4:0] rd,              // 输出：写入寄存器文件的目标寄存器地址

    output [4:0] cp0_addr,        // 输出：CP0 寄存器地址
    output [4:0] cause,           // 输出：异常原因代码
    output mfc0,                  // 输出：MFC0 指令信号
    output mtc0,                  // 输出：MTC0 指令信号
    output eret,                  // 输出：ERET 指令信号
```

```
output exception,            // 输出：异常信号

output ext5_mux_sel,         // 输出：extend5 模块的选择信号
output cutter_mux_sel,       // 输出：截断器的选择信号
output alu_mux1_sel,         // 输出：ALU 的第一个输入选择信号
output [2:0] cutter_sel,     // 输出：截断器的选择信号
output [2:0] rf_mux_sel,     // 输出：寄存器文件写入选择信号
output [2:0] pc_mux_sel,     // 输出：PC 输入选择信号
output [1:0] alu_mux2_sel,   // 输出：ALU 的第二个输入选择信号
output [1:0] hi_mux_sel,     // 输出：hi 寄存器写入选择信号
output [1:0] lo_mux_sel      // 输出：lo 寄存器写入选择信号
);

//下面是不同指令对应的操作码
wire Addi = (op == 6'b001000);
wire Andi = (op == 6'b001100);
wire Ori = (op == 6'b001101);
wire Sltiu = (op == 6'b001011);
wire Addiu = (op == 6'b001001);

wire Xori = (op == 6'b001110);
wire Slti = (op == 6'b001010);
wire Addu = (op == 6'b000000 && func==6'b100001);
wire And = (op == 6'b000000 && func == 6'b100100);
wire Lui = (op == 6'b001111);

wire Beq = (op == 6'b000100);
wire Bne = (op == 6'b000101);
wire Jal = (op == 6'b000011);
wire Jr = (op == 6'b000000 && func == 6'b001000);
wire J = (op == 6'b000010);


wire Lw = (op == 6'b100011);
wire Sll = (op == 6'b000000 && func == 6'b000000);


wire Xor = (op == 6'b000000 && func == 6'b100110);
wire Nor = (op == 6'b000000 && func == 6'b100111);
wire Or = (op == 6'b000000 && func == 6'b100101);

wire Subu = (op == 6'b000000 && func == 6'b100011);
wire Sllv = (op == 6'b000000 && func == 6'b000100);
wire Slt = (op == 6'b000000 && func == 6'b101010);
```

```
wire Sra = (op == 6'b000000 && func == 6'b000011);
wire Srl = (op == 6'b000000 && func == 6'b000010);
wire Srlv = (op == 6'b000000 && func == 6'b000110);
wire Srav = (op == 6'b000000 && func == 6'b000111);
wire Sltu = (op == 6'b000000 && func == 6'b101011);

wire Lbu = (op == 6'b100100);
wire Lhu = (op == 6'b100101);
wire Sb = (op == 6'b101000);
wire Clz = (op == 6'b011100 && func == 6'b100000);
wire Divu = (op == 6'b000000 && func == 6'b011011);
wire Eret = (op == 6'b010000 && func == 6'b011000);
wire Jalr = (op == 6'b000000 && func == 6'b001001);
 wire Sw = (op == 6'b101011);
wire Add = (op == 6'b000000 && func == 6'b100000);
wire Sub = (op == 6'b000000 && func == 6'b100010);
wire Lb = (op == 6'b100000);


wire Mfc0 = (instruction[31:21] == 11'b01000000000 && instruction[10:3]==8'b00000000);
wire Mfhi = (op == 6'b000000 && func == 6'b010000);
wire Mflo = (op == 6'b000000 && func == 6'b010010);
wire Sh = (op == 6'b101001);
wire Lh = (op == 6'b100001);
wire Mtc0 = (instruction[31:21] == 11'b01000000100 && instruction[10:3]==8'b00000000);
wire Teq = (op == 6'b000000 && func == 6'b110100);
wire Bgez = (op == 6'b000001);
wire Mthi = (op == 6'b000000 && func == 6'b010001);

wire Syscall = (op == 6'b000000 && func== 6'b001100);

wire Break = (op == 6'b000000 && func == 6'b001101);
wire Div = (op == 6'b000000 && func == 6'b011010);
wire Mtlo = (op == 6'b000000 && func == 6'b010011);
wire Mul = (op == 6'b011100 && func == 6'b000010);
wire Multu = (op == 6'b000000 && func == 6'b011001);



assign hi_wena = Div+Divu+Multu+Mthi+Mul;
assign lo_wena = Div+Divu+Multu+Mtlo+Mul;
assign clz_ena = Clz;
assign mul_ena = Mul+Multu;
assign div_ena = Div+Divu;
```

```
    assign rf_wena = Addi+Addiu+Andi+Ori+Sltiu+Lui+Xori+Slti+Addu+
    And+Xor+Nor+Or+Sll+Sllv+Sltu+Sra+Srl+Subu+Add+Sub+Slt+Srlv+Srav+
    Lb+Lbu+Lh+Lhu+Lw+Mfc0+Clz+Jal+Jalr+Mfhi+Mflo+Mul;

    assign rf_rena1 = Addi+Addiu+Andi+Ori+Sltiu+Xori+Slti+Addu+And+
    Beq+Bne+Jr+Lw+Xor+Nor+Or+Sllv+Sltu+Subu+Sw+Add+Sub+Slt+Srlv+Srav+
    Clz+Divu+Jalr+Lb+Lbu+Lhu+Sb+Sh+Lh+Mul+Multu+Teq+Div;

    assign rf_rena2 = Addu+And+Beq+Bne+Xor+Nor+Or+Sll+Sllv+Sltu+Sra+
    Srl+Subu+Sw+Add+Sub+Slt+Srlv+Srav+Divu+Sb+Sh+Mtc0+Mul+Multu+Teq+Div;


    assign dmem_wena = Sb+Sh+Sw;
    assign dmem_w_cs[1] = Sh+Sb;
    assign dmem_w_cs[0] = Sw+Sb;
    assign dmem_r_cs[1] = Lh+Lb+Lhu+Lbu;
    assign dmem_r_cs[0] = Lw+Lb+Lbu;
    assign dmem_ena = Lw+Sw+Sb+Sh+Lb+Lh+Lhu+Lbu;


    assign cutter_sign = Lb+Lh;


    assign ext5_mux_sel = Sllv+Srav+Srlv;
    assign mul_sign = Mul;
    assign div_sign = Div;
    assign ext16_sign = Addi+Addiu+Sltiu+Slti;

    assign alu_mux2_sel[1] = Bgez;
    assign alu_mux1_sel =
~(Sll+Srl+Sra+Div+Divu+Mul+Multu+J+Jr+Jal+Jalr+Mfc0+Mtc0+Mfhi+Mflo+Mthi+Mtlo+Clz+Eret+Sy
scall+Break);

    assign alu_mux2_sel[0] =
Slti+Sltiu+Addi+Addiu+Andi+Ori+Xori+Lb+Lbu+Lh+Lhu+Lw+Sb+Sh+Sw+Lui;

    assign aluc[0] = Subu+Sub+Or+Nor+Slt+Sllv+Srlv+Sll+Srl+Slti+Ori+Beq+Bne+Bgez+Teq;
     assign aluc[1] =
Add+Sub+Xor+Nor+Slt+Sltu+Sll+Sllv+Addi+Xori+Beq+Bne+Slti+Sltiu+Bgez+Teq;
     assign aluc[2] = And+Or+Xor+Nor+Sll+Srl+Sra+Sllv+Srlv+Srav+Andi+Ori+Xori;
    assign aluc[3] = Slt+Sltu+Sllv+Srlv+Srav+Lui+Srl+Sra+Slti+Sltiu+Sll;


    assign cutter_sel[0] = Lh+Lhu+Sb;
```

```
    assign cutter_sel[1] = Lb+Lbu+Sb;
    assign cutter_sel[2] = Sh;
    assign cutter_mux_sel = ~(Sb+Sh+Sw);

    assign rf_mux_sel[2] =
~(Beq+Bne+Bgez+Div+Divu+Sb+Multu+Sh+Sw+J+Jr+Jal+Jalr+Mfc0+Mtc0+Mflo+Mthi+Mtlo+Clz+Eret+S
yscall+Teq+Break);
    assign rf_mux_sel[1] = Mul+Mfc0+Mtc0+Clz+Mfhi;
    assign rf_mux_sel[0] =
~(Beq+Bne+Bgez+Div+Divu+Multu+Lb+Lbu+Lh+Lhu+Lw+Sb+Sh+Sw+J+Mtc0+Mfhi+Mflo+Mthi+Mtlo+Clz+E
ret+Syscall+Teq+Break);

    assign hi_mux_sel[1] = Mthi;
    assign hi_mux_sel[0] = Mul+Multu;

    assign pc_mux_sel[2] = Eret+(Beq&&is_branch)+(Bne&&is_branch)+(Bgez&&is_branch);
    assign pc_mux_sel[1] = ~(J+Jr+Jal+Jalr+pc_mux_sel[2]);
    assign pc_mux_sel[0] = Eret+exception+Jr+Jalr;

    assign rd =
(Add+Addu+Sub+Subu+And+Or+Xor+Nor+Slt+Sltu+Sll+Srl+Sra+Sllv+Srlv+Srav+Clz+Jalr+Mfhi+Mflo
+Mul) ?
                  instruction[15:11] :
(( Addi+Addiu+Andi+Ori+Xori+Lb+Lbu+Lh+Lhu+Lw+Slti+Sltiu+Lui+Mfc0) ?
                  instruction[20:16] : (Jal?5'd31:5'b0));

    assign lo_mux_sel[1] = Mtlo;
    assign lo_mux_sel[0] = Mul+Multu;

    //CP0 相关
    assign cp0_addr = instruction[15:11];
    assign mfc0 = Mfc0;
    assign mtc0 = Mtc0;
    assign exception = status[0] && ((Syscall && status[1]) || (Break && status[2]) || (Teq
&& status[3]));
    assign eret = Eret;
    assign cause = Break ? 5'b01001 : (Syscall ? 5'b01000 : (Teq ? 5'b01101 : 5'b00000));

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"
```

```
module cp0(
    input clk,
    input rst,
    input mfc0,            // MFC0（Move From CP0）指令信号
    input mtc0,            // MTC0（Move To CP0）指令信号
    input [31:0] pc,
    input [4:0] addr,
    input [31:0] wdata,
    input exception,       // 异常信号
    input eret,            // ERET（Exception Return）指令信号
    input [4:0] cause,     // 异常原因码
    output [31:0] rdata,   // 从 CP0 寄存器读取的数据
    output [31:0] status,  // 状态寄存器的值
    output [31:0] exc_addr // 异常地址
    );

    reg [31:0] register [31:0]; // CP0 寄存器数组
    reg [31:0] temp_status;     // 临时存储状态寄存器的值
    integer i;

    // 寄存器写入和状态更新逻辑
    always @ (posedge clk or posedge rst)
    begin
        if (rst == `RST_ENABLED) begin
            for(i = 0; i < 32; i = i + 1) begin
                if(i == `STATUS)
                    register[i] <= 32'h0000000f; // 在复位时，将状态寄存器初始化为默认值
                else
                    register[i] <= 0;
            end
            temp_status <= 0;
        end

        else begin
            if(mtc0)
                register[addr] <= wdata; // 如果是 MTC0 指令，则将指定的 CP0 寄存器写入新的数据

            else if(exception) begin
                register[`EPC] = pc;  // 如果有异常，将程序计数器（PC）的值写入 EPC 寄存器
                temp_status = register[`STATUS]; // 保存状态寄存器的值
                register[`STATUS] = register[`STATUS] << 5; // 修改状态寄存器的值
                register[`CAUSE] = {25'b0, cause, 2'b0}; // 设置异常原因码
            end
```

```
            else if(eret)
                register[`STATUS] = temp_status; // 如果是 ERET 指令，则恢复状态寄存器的值
        end
    end

    // 根据不同的指令信号，决定从 CP0 寄存器读取的数据
    assign exc_addr = eret ? register[`EPC] : `EXCEPTION_ADDR;
    assign rdata = mfc0 ? register[addr] : 32'h0; // 根据 MFC0 指令信号决定读取的数据
    assign status = register[`STATUS]; // 状态寄存器的值

endmodule
```

```
`timescale 1ns / 1ps

module cpu_top(
    input clk,
    input rst,
    input stop, //外部中断信号，stop = 1 表示外部中断
    output [31:0] instruction,
    output [31:0] pc,
    output [31:0] reg28,    //存储 c[i]+d[i]的结果
    output [31:0] reg29     //未使用到的寄存器，值为 0
    );

    // Wires for stall
    wire stall;

    // Wires for branch prediction
    wire is_branch;

    // Wires for pc
    wire pc_wena;

    // Wires for IF
    // IF Input
    wire [31:0] if_pc_i;  // PC + 4
    wire [31:0] if_cp0_pc_i;  // CP0 PC
    wire [31:0] if_b_pc_i;  // BEQ and BNE PC
    wire [31:0] if_r_pc_i;  // Jr and Jalr PC
    wire [31:0] if_j_pc_i;  // j instruction PC
    wire [2:0] if_pc_mux_sel_i;
```

```
// IF Output
wire [31:0] if_npc_o;
wire [31:0] if_pc4_o;
wire [31:0] if_instruction_o;

// Wires for ID
// ID Input
wire [31:0] id_pc4_i;  // PC + 4
wire [31:0] id_instruction_i;
wire [31:0] id_rf_wdata_i;    //data for writing in regfile
wire [31:0] id_hi_wdata_i;
wire [31:0] id_lo_wdata_i;
wire [4:0] id_rf_waddr_i;  // Regfile write address
wire id_rf_wena_i;        //writing enable for regfile
wire id_hi_wena_i;        //writing enable for hi register
wire id_lo_wena_i;        //writing enable for lo register

// ID Output
wire [31:0] id_cp0_pc_o;  // Interrupt address
wire [31:0] id_r_pc_o;
wire [31:0] id_b_pc_o;  // Jump address
wire [31:0] id_j_pc_o;  // Beq and Bne address
wire [31:0] id_rs_data_out_o;
wire [31:0] id_rt_data_out_o;
wire [31:0] id_imm_o;  // Immediate value
wire [31:0] id_shamt_o;
wire [31:0] id_pc4_o;
wire [31:0] id_cp0_out_o;    //output data of cp0
wire [31:0] id_hi_out_o;     //output data of hi register
wire [31:0] id_lo_out_o;      //output data of lo register
wire [4:0] id_rf_waddr_o;
wire [3:0] id_aluc_o;
wire id_mul_sign_o;
wire id_div_sign_o;
wire id_cutter_sign_o;
wire id_clz_ena_o;
wire id_mul_ena_o;
wire id_div_ena_o;
wire id_dmem_ena_o;
wire id_hi_wena_o;
wire id_lo_wena_o;
wire id_rf_wena_o;
wire id_dmem_wena_o;
wire [1:0] id_dmem_w_cs_o;
```

```
    wire [1:0] id_dmem_r_cs_o;
    wire id_cutter_mux_sel_o;
    wire id_alu_mux1_sel_o;
    wire [1:0] id_alu_mux2_sel_o;
    wire [1:0] id_hi_mux_sel_o;
    wire [1:0] id_lo_mux_sel_o;
    wire [2:0] id_cutter_sel_o;
    wire [2:0] id_rf_mux_sel_o;
    wire [2:0] id_pc_mux_sel_o;
    wire [5:0] id_op_o;
    wire [5:0] id_func_o;

    // Wires for EXE
    // EXE Input
    wire [31:0] exe_pc4_i;
    wire [31:0] exe_imm_i;
    wire [31:0] exe_shamt_i;
    wire [31:0] exe_rs_data_out_i;
    wire [31:0] exe_rt_data_out_i;
    wire [31:0] exe_cp0_out_i;
    wire [31:0] exe_hi_out_i;
    wire [31:0] exe_lo_out_i;
    wire [4:0] exe_rf_waddr_i;
    wire [3:0] exe_aluc_i;
    wire exe_mul_ena_i;
    wire exe_div_ena_i;
    wire exe_clz_ena_i;
    wire exe_dmem_ena_i;
    wire [1:0] exe_dmem_w_cs_i;
    wire [1:0] exe_dmem_r_cs_i;
    wire exe_mul_sign_i;
    wire exe_div_sign_i;
    wire exe_cutter_sign_i;
    wire exe_rf_wena_i;
    wire exe_hi_wena_i;
    wire exe_lo_wena_i;
    wire exe_dmem_wena_i;
    wire exe_cutter_mux_sel_i;
    wire exe_alu_mux1_sel_i;
    wire [1:0] exe_alu_mux2_sel_i;
    wire [2:0] exe_cutter_sel_i;
    wire [1:0] exe_hi_mux_sel_i;
    wire [1:0] exe_lo_mux_sel_i;
    wire [2:0] exe_rf_mux_sel_i;
```

```
wire [5:0] exe_op_i;
wire [5:0] exe_func_i;

// EXE Output
wire [31:0] exe_mul_hi_o;  // Higher bits of the multiplication result
wire [31:0] exe_mul_lo_o;  // Lower bits of the multiplication result
wire [31:0] exe_div_r_o;  // Remainder of the division result
wire [31:0] exe_div_q_o;  // Quotient of the division result
wire [31:0] exe_clz_out_o;
wire [31:0] exe_alu_out_o;  // Result of ALU
wire [31:0] exe_pc4_o;
wire [31:0] exe_rs_data_out_o;
wire [31:0] exe_rt_data_out_o;
wire [31:0] exe_cp0_out_o;
wire [31:0] exe_hi_out_o;
wire [31:0] exe_lo_out_o;
wire [4:0] exe_rf_waddr_o;
wire exe_dmem_ena_o;
wire exe_cutter_sign_o;
wire exe_rf_wena_o;
wire exe_hi_wena_o;
wire exe_lo_wena_o;
wire exe_dmem_wena_o;
wire [1:0] exe_dmem_w_cs_o;
wire [1:0] exe_dmem_r_cs_o;
wire exe_cutter_mux_sel_o;
wire [2:0] exe_cutter_sel_o;
wire [1:0] exe_hi_mux_sel_o;
wire [1:0] exe_lo_mux_sel_o;
wire [2:0] exe_rf_mux_sel_o;

// Wires for MEM
// MEM Input
wire [31:0] mem_mul_hi_i;
wire [31:0] mem_mul_lo_i;
wire [31:0] mem_div_r_i;
wire [31:0] mem_div_q_i;
wire [31:0] mem_clz_out_i;
wire [31:0] mem_alu_out_i;
wire [31:0] mem_pc4_i;
wire [31:0] mem_rs_data_out_i;
wire [31:0] mem_rt_data_out_i;
wire [31:0] mem_cp0_out_i;
wire [31:0] mem_hi_out_i;
```

```
    wire [31:0] mem_lo_out_i;
    wire [4:0] mem_rf_waddr_i;
    wire mem_dmem_ena_i;
    wire mem_rf_wena_i;
    wire mem_hi_wena_i;
    wire mem_lo_wena_i;
    wire mem_dmem_wena_i;
    wire mem_cutter_sign_i;
    wire [1:0] mem_dmem_w_cs_i;
    wire [1:0] mem_dmem_r_cs_i;
    wire mem_cutter_mux_sel_i;
    wire [2:0] mem_cutter_sel_i;
    wire [1:0] mem_hi_mux_sel_i;
    wire [1:0] mem_lo_mux_sel_i;
    wire [2:0] mem_rf_mux_sel_i;

    // MEM Output
    wire [31:0] mem_mul_hi_o;  // Higher bits of the multiplication result
    wire [31:0] mem_mul_lo_o;  // Lower bits of the multiplication result
    wire [31:0] mem_div_r_o;  // Remainder of the division result
    wire [31:0] mem_div_q_o;  // Quotient of the division result
    wire [31:0] mem_clz_out_o;
    wire [31:0] mem_alu_out_o;  // Result of ALU
    wire [31:0] mem_dmem_out_o;
    wire [31:0] mem_pc4_o;
    wire [31:0] mem_rs_data_out_o;
    wire [31:0] mem_cp0_out_o;
    wire [31:0] mem_hi_out_o;
    wire [31:0] mem_lo_out_o;
    wire [4:0] mem_rf_waddr_o;
    wire mem_rf_wena_o;
    wire mem_hi_wena_o;
    wire mem_lo_wena_o;
    wire [1:0] mem_hi_mux_sel_o;
    wire [1:0] mem_lo_mux_sel_o;
    wire [2:0] mem_rf_mux_sel_o;

    // Wires for WB
    // WB Input
    wire [31:0] wb_mul_hi_i;
    wire [31:0] wb_mul_lo_i;
    wire [31:0] wb_div_r_i;
    wire [31:0] wb_div_q_i;
    wire [31:0] wb_clz_out_i;
```

```
    wire [31:0] wb_alu_out_i;
    wire [31:0] wb_dmem_out_i;
    wire [31:0] wb_pc4_i;
    wire [31:0] wb_rs_data_out_i;
    wire [31:0] wb_cp0_out_i;
    wire [31:0] wb_hi_out_i;
    wire [31:0] wb_lo_out_i;
    wire [4:0] wb_rf_waddr_i;
    wire wb_rf_wena_i;
    wire wb_hi_wena_i;
    wire wb_lo_wena_i;
    wire [1:0] wb_hi_mux_sel_i;
    wire [1:0] wb_lo_mux_sel_i;
    wire [2:0] wb_rf_mux_sel_i;

    // WB Output
    wire [31:0] wb_hi_wdata_o;
    wire [31:0] wb_lo_wdata_o;
    wire [31:0] wb_rf_wdata_o;
    wire [4:0] wb_rf_waddr_o;
    wire wb_rf_wena_o;
    wire wb_hi_wena_o;
    wire wb_lo_wena_o;

    wire id_exe_wena;
    wire exe_mem_wena;
    wire mem_wb_wena;

    assign instruction = if_instruction_o;
    assign pc = if_pc_i;
    assign pc_wena = 1'b1;
    assign id_exe_wena = 1'b1;
    assign exe_mem_wena = 1'b1;
    assign mem_wb_wena = 1'b1;

    assign if_pc_mux_sel_i = id_pc_mux_sel_o;
    assign if_cp0_pc_i = id_cp0_pc_o;
    assign if_b_pc_i = id_b_pc_o;
    assign if_r_pc_i = id_r_pc_o;
    assign if_j_pc_i = id_j_pc_o;

    assign id_rf_wdata_i = wb_rf_wdata_o;
    assign id_hi_wdata_i = wb_hi_wdata_o;
    assign id_lo_wdata_i = wb_lo_wdata_o;
```

```
assign id_rf_waddr_i = wb_rf_waddr_o;
assign id_rf_wena_i = wb_rf_wena_o;
assign id_hi_wena_i = wb_hi_wena_o;
assign id_lo_wena_i = wb_lo_wena_o;

//PC 寄存器
pc_reg program_counter(
    .clk(clk),
    .rst(rst),
    .wena(pc_wena),
    .stall(stall | stop),
    .data_in(if_npc_o),
    .data_out(if_pc_i)
);

//IF 阶段
IF instruction_fetch(
    .pc(if_pc_i),
    .cp0_pc(if_cp0_pc_i),
    .b_pc(if_b_pc_i),
    .r_pc(if_r_pc_i),
    .j_pc(if_j_pc_i),
    .pc_mux_sel(if_pc_mux_sel_i),
    .npc(if_npc_o),
    .pc4(if_pc4_o),
    .instruction(if_instruction_o)
);

//IF-ID 流水寄存器
IF_ID_Reg if_id_reg(
    .clk(clk),
    .rst(rst),
    .if_pc4(if_pc4_o),
    .if_instruction(if_instruction_o),
    .stall(stall),
    .is_branch(is_branch),
    .id_pc4(id_pc4_i),
    .id_instruction(id_instruction_i)
);

//ID 阶段
ID instruction_decoder(
    .clk(clk),
    .rst(rst),
```

```
        .pc4(id_pc4_i),
        .instruction(id_instruction_i),
        .rf_wdata(id_rf_wdata_i),
        .hi_wdata(id_hi_wdata_i),
        .lo_wdata(id_lo_wdata_i),
        .rf_waddr(id_rf_waddr_i),
        .rf_wena(id_rf_wena_i),
        .hi_wena(id_hi_wena_i),
        .lo_wena(id_lo_wena_i),

        //EX 数据
        .exe_rf_waddr(exe_rf_waddr_o),
        .exe_rf_wena(exe_rf_wena_o),
        .exe_hi_wena(exe_hi_wena_o),
        .exe_lo_wena(exe_lo_wena_o),
        .exe_mul_hi(exe_mul_hi_o),
        .exe_mul_lo(exe_mul_lo_o),
        .exe_div_r(exe_div_r_o),
        .exe_div_q(exe_div_q_o),
        .exe_rs_data_out(exe_rs_data_out_o),
        .exe_lo_out(exe_lo_out_o),
        .exe_pc4(exe_pc4_o),
        .exe_clz_out(exe_clz_out_o),
        .exe_alu_out(exe_alu_out_o),
        .exe_hi_out(exe_hi_out_o),
        .exe_hi_mux_sel(exe_hi_mux_sel_o),
        .exe_lo_mux_sel(exe_lo_mux_sel_o),
        .exe_rf_mux_sel(exe_rf_mux_sel_o),
        .exe_op(exe_op_i),
        .exe_func(exe_func_i),

        // ME 数据
        .mem_rf_waddr(mem_rf_waddr_o),
        .mem_rf_wena(mem_rf_wena_o),
        .mem_hi_wena(mem_hi_wena_o),
        .mem_lo_wena(mem_lo_wena_o),
        .mem_mul_hi(mem_mul_hi_o),
        .mem_mul_lo(mem_mul_lo_o),
        .mem_div_r(mem_div_r_o),
        .mem_div_q(mem_div_q_o),
        .mem_rs_data_out(mem_rs_data_out_o),
        .mem_dmem_out(mem_dmem_out_o),
        .mem_lo_out(mem_lo_out_o),
        .mem_pc4(mem_pc4_o),
```

```
        .mem_clz_out(mem_clz_out_o),
        .mem_alu_out(mem_alu_out_o),
        .mem_hi_out(mem_hi_out_o),
        .mem_hi_mux_sel(mem_hi_mux_sel_o),
        .mem_lo_mux_sel(mem_lo_mux_sel_o),
        .mem_rf_mux_sel(mem_rf_mux_sel_o),

        //ID 数据
        .id_cp0_pc(id_cp0_pc_o),
        .id_r_pc(id_r_pc_o),
        .id_b_pc(id_b_pc_o),
        .id_j_pc(id_j_pc_o),
        .id_rs_data_out(id_rs_data_out_o),
        .id_rt_data_out(id_rt_data_out_o),
        .id_imm(id_imm_o),
        .id_shamt(id_shamt_o),
        .id_pc4(id_pc4_o),
        .id_cp0_out(id_cp0_out_o),
        .id_hi_out(id_hi_out_o),
        .id_lo_out(id_lo_out_o),
        .id_rf_waddr(id_rf_waddr_o),
        .id_aluc(id_aluc_o),
        .id_mul_sign(id_mul_sign_o),
        .id_div_sign(id_div_sign_o),
        .id_cutter_sign(id_cutter_sign_o),
        .id_clz_ena(id_clz_ena_o),
        .id_mul_ena(id_mul_ena_o),
        .id_div_ena(id_div_ena_o),
        .id_dmem_ena(id_dmem_ena_o),
        .id_hi_wena(id_hi_wena_o),
        .id_lo_wena(id_lo_wena_o),
        .id_rf_wena(id_rf_wena_o),
        .id_dmem_wena(id_dmem_wena_o),
        .id_dmem_w_cs(id_dmem_w_cs_o),
        .id_dmem_r_cs(id_dmem_r_cs_o),
        .id_cutter_mux_sel(id_cutter_mux_sel_o),
        .id_alu_mux1_sel(id_alu_mux1_sel_o),
        .id_alu_mux2_sel(id_alu_mux2_sel_o),
        .id_hi_mux_sel(id_hi_mux_sel_o),
        .id_lo_mux_sel(id_lo_mux_sel_o),
        .id_cutter_sel(id_cutter_sel_o),
        .id_rf_mux_sel(id_rf_mux_sel_o),
        .id_pc_mux_sel(id_pc_mux_sel_o),
        .id_op(id_op_o),
```

```
    .id_func(id_func_o),
    .stall(stall),
    .is_branch(is_branch),
    .reg28(reg28),
    .reg29(reg29)
);

//ID-EX 流水寄存器
ID_EX_Reg id_exe_reg(
    .clk(clk),
    .rst(rst),
    .wena(id_exe_wena),
    .id_pc4(id_pc4_o),
    .id_rs_data_out(id_rs_data_out_o),
    .id_rt_data_out(id_rt_data_out_o),
    .id_imm(id_imm_o),
    .id_shamt(id_shamt_o),
    .id_cp0_out(id_cp0_out_o),
    .id_hi_out(id_hi_out_o),
    .id_lo_out(id_lo_out_o),
    .id_rf_waddr(id_rf_waddr_o),
    .id_clz_ena(id_clz_ena_o),
    .id_mul_ena(id_mul_ena_o),
    .id_div_ena(id_div_ena_o),
    .id_dmem_ena(id_dmem_ena_o),
    .id_mul_sign(id_mul_sign_o),
    .id_div_sign(id_div_sign_o),
    .id_cutter_sign(id_cutter_sign_o),
    .id_aluc(id_aluc_o),
    .id_rf_wena(id_rf_wena_o),
    .id_hi_wena(id_hi_wena_o),
    .id_lo_wena(id_lo_wena_o),
    .id_dmem_wena(id_dmem_wena_o),
    .id_dmem_w_cs(id_dmem_w_cs_o),
    .id_dmem_r_cs(id_dmem_r_cs_o),
    .stall(stall),
    .id_cutter_mux_sel(id_cutter_mux_sel_o),
    .id_alu_mux1_sel(id_alu_mux1_sel_o),
    .id_alu_mux2_sel(id_alu_mux2_sel_o),
    .id_hi_mux_sel(id_hi_mux_sel_o),
    .id_lo_mux_sel(id_lo_mux_sel_o),
    .id_cutter_sel(id_cutter_sel_o),
    .id_rf_mux_sel(id_rf_mux_sel_o),
    .id_op(id_op_o),
```

```
    .id_func(id_func_o),
    .exe_pc4(exe_pc4_i),
    .exe_rs_data_out(exe_rs_data_out_i),
    .exe_rt_data_out(exe_rt_data_out_i),
    .exe_imm(exe_imm_i),
    .exe_shamt(exe_shamt_i),
    .exe_cp0_out(exe_cp0_out_i),
    .exe_hi_out(exe_hi_out_i),
    .exe_lo_out(exe_lo_out_i),
    .exe_rf_waddr(exe_rf_waddr_i),
    .exe_clz_ena(exe_clz_ena_i),
    .exe_mul_ena(exe_mul_ena_i),
    .exe_div_ena(exe_div_ena_i),
    .exe_dmem_ena(exe_dmem_ena_i),
    .exe_mul_sign(exe_mul_sign_i),
    .exe_div_sign(exe_div_sign_i),
    .exe_cutter_sign(exe_cutter_sign_i),
    .exe_aluc(exe_aluc_i),
    .exe_rf_wena(exe_rf_wena_i),
    .exe_hi_wena(exe_hi_wena_i),
    .exe_lo_wena(exe_lo_wena_i),
    .exe_dmem_wena(exe_dmem_wena_i),
    .exe_dmem_w_cs(exe_dmem_w_cs_i),
    .exe_dmem_r_cs(exe_dmem_r_cs_i),
    .exe_alu_mux1_sel(exe_alu_mux1_sel_i),
    .exe_alu_mux2_sel(exe_alu_mux2_sel_i),
    .exe_cutter_mux_sel(exe_cutter_mux_sel_i),
    .exe_hi_mux_sel(exe_hi_mux_sel_i),
    .exe_lo_mux_sel(exe_lo_mux_sel_i),
    .exe_cutter_sel(exe_cutter_sel_i),
    .exe_rf_mux_sel(exe_rf_mux_sel_i),
    .exe_op(exe_op_i),
    .exe_func(exe_func_i)
);

//EX 阶段
EX executor(
    .rst(rst),
    .pc4(exe_pc4_i),
    .rs_data_out(exe_rs_data_out_i),
    .rt_data_out(exe_rt_data_out_i),
    .imm(exe_imm_i),
    .shamt(exe_shamt_i),
    .cp0_out(exe_cp0_out_i),
```

```
        .hi_out(exe_hi_out_i),
        .lo_out(exe_lo_out_i),
        .rf_waddr(exe_rf_waddr_i),
        .aluc(exe_aluc_i),
        .mul_ena(exe_mul_ena_i),
        .div_ena(exe_div_ena_i),
        .clz_ena(exe_clz_ena_i),
        .dmem_ena(exe_dmem_ena_i),
        .mul_sign(exe_mul_sign_i),
        .div_sign(exe_div_sign_i),
        .cutter_sign(exe_cutter_sign_i),
        .rf_wena(exe_rf_wena_i),
        .hi_wena(exe_hi_wena_i),
        .lo_wena(exe_lo_wena_i),
        .dmem_wena(exe_dmem_wena_i),
        .dmem_w_cs(exe_dmem_w_cs_i),
        .dmem_r_cs(exe_dmem_r_cs_i),
        .cutter_mux_sel(exe_cutter_mux_sel_i),
        .alu_mux1_sel(exe_alu_mux1_sel_i),
        .alu_mux2_sel(exe_alu_mux2_sel_i),
        .hi_mux_sel(exe_hi_mux_sel_i),
        .lo_mux_sel(exe_lo_mux_sel_i),
        .cutter_sel(exe_cutter_sel_i),
        .rf_mux_sel(exe_rf_mux_sel_i),
        .exe_mul_hi(exe_mul_hi_o),
        .exe_mul_lo(exe_mul_lo_o),
        .exe_div_r(exe_div_r_o),
        .exe_div_q(exe_div_q_o),
        .exe_clz_out(exe_clz_out_o),
        .exe_alu_out(exe_alu_out_o),
        .exe_pc4(exe_pc4_o),
        .exe_rs_data_out(exe_rs_data_out_o),
        .exe_rt_data_out(exe_rt_data_out_o),
        .exe_cp0_out(exe_cp0_out_o),
        .exe_hi_out(exe_hi_out_o),
        .exe_lo_out(exe_lo_out_o),
        .exe_rf_waddr(exe_rf_waddr_o),
        .exe_dmem_ena(exe_dmem_ena_o),
        .exe_rf_wena(exe_rf_wena_o),
        .exe_hi_wena(exe_hi_wena_o),
        .exe_lo_wena(exe_lo_wena_o),
        .exe_dmem_wena(exe_dmem_wena_o),
        .exe_dmem_w_cs(exe_dmem_w_cs_o),
        .exe_dmem_r_cs(exe_dmem_r_cs_o),
```

```
    .exe_cutter_sign(exe_cutter_sign_o),
    .exe_cutter_mux_sel(exe_cutter_mux_sel_o),
    .exe_cutter_sel(exe_cutter_sel_o),
    .exe_hi_mux_sel(exe_hi_mux_sel_o),
    .exe_lo_mux_sel(exe_lo_mux_sel_o),
    .exe_rf_mux_sel(exe_rf_mux_sel_o)
);

//EX-ME 流水寄存器
EX_ME_Reg exe_mem_reg(
    .clk(clk),
    .rst(rst),
    .wena(exe_mem_wena),
    .exe_mul_hi(exe_mul_hi_o),
    .exe_mul_lo(exe_mul_lo_o),
    .exe_div_r(exe_div_r_o),
    .exe_div_q(exe_div_q_o),
    .exe_clz_out(exe_clz_out_o),
    .exe_alu_out(exe_alu_out_o),
    .exe_pc4(exe_pc4_o),
    .exe_rs_data_out(exe_rs_data_out_o),
    .exe_rt_data_out(exe_rt_data_out_o),
    .exe_cp0_out(exe_cp0_out_o),
    .exe_hi_out(exe_hi_out_o),
    .exe_lo_out(exe_lo_out_o),
    .exe_rf_waddr(exe_rf_waddr_o),
    .exe_dmem_ena(exe_dmem_ena_o),
    .exe_cutter_sign(exe_cutter_sign_o),
    .exe_rf_wena(exe_rf_wena_o),
    .exe_hi_wena(exe_hi_wena_o),
    .exe_lo_wena(exe_lo_wena_o),
    .exe_dmem_wena(exe_dmem_wena_o),
    .exe_dmem_w_cs(exe_dmem_w_cs_o),
    .exe_dmem_r_cs(exe_dmem_r_cs_o),
    .exe_cutter_mux_sel(exe_cutter_mux_sel_o),
    .exe_hi_mux_sel(exe_hi_mux_sel_o),
    .exe_lo_mux_sel(exe_lo_mux_sel_o),
    .exe_cutter_sel(exe_cutter_sel_o),
    .exe_rf_mux_sel(exe_rf_mux_sel_o),
    .mem_mul_hi(mem_mul_hi_i),
    .mem_mul_lo(mem_mul_lo_i),
    .mem_div_r(mem_div_r_i),
    .mem_div_q(mem_div_q_i),
    .mem_clz_out(mem_clz_out_i),
```

```
    .mem_alu_out(mem_alu_out_i),
    .mem_pc4(mem_pc4_i),
    .mem_rs_data_out(mem_rs_data_out_i),
    .mem_rt_data_out(mem_rt_data_out_i),
    .mem_cp0_out(mem_cp0_out_i),
    .mem_hi_out(mem_hi_out_i),
    .mem_lo_out(mem_lo_out_i),
    .mem_rf_waddr(mem_rf_waddr_i),
    .mem_dmem_ena(mem_dmem_ena_i),
    .mem_cutter_sign(mem_cutter_sign_i),
    .mem_rf_wena(mem_rf_wena_i),
    .mem_hi_wena(mem_hi_wena_i),
    .mem_lo_wena(mem_lo_wena_i),
    .mem_dmem_wena(mem_dmem_wena_i),
    .mem_dmem_w_cs(mem_dmem_w_cs_i),
    .mem_dmem_r_cs(mem_dmem_r_cs_i),
    .mem_cutter_mux_sel(mem_cutter_mux_sel_i),
    .mem_hi_mux_sel(mem_hi_mux_sel_i),
    .mem_lo_mux_sel(mem_lo_mux_sel_i),
    .mem_cutter_sel(mem_cutter_sel_i),
    .mem_rf_mux_sel(mem_rf_mux_sel_i)
);

//ME 阶段
ME memory(
    .clk(clk),
    .mul_hi(mem_mul_hi_i),
    .mul_lo(mem_mul_lo_i),
    .div_r(mem_div_r_i),
    .div_q(mem_div_q_i),
    .clz_out(mem_clz_out_i),
    .alu_out(mem_alu_out_i),
    .pc4(mem_pc4_i),
    .rs_data_out(mem_rs_data_out_i),
    .rt_data_out(mem_rt_data_out_i),
    .cp0_out(mem_cp0_out_i),
    .hi_out(mem_hi_out_i),
    .lo_out(mem_lo_out_i),
    .rf_waddr(mem_rf_waddr_i),
    .dmem_ena(mem_dmem_ena_i),
    .rf_wena(mem_rf_wena_i),
    .hi_wena(mem_hi_wena_i),
    .lo_wena(mem_lo_wena_i),
    .dmem_wena(mem_dmem_wena_i),
```

```
    .cutter_sign(mem_cutter_sign_i),
    .dmem_w_cs(mem_dmem_w_cs_i),
    .dmem_r_cs(mem_dmem_r_cs_i),
    .cutter_mux_sel(mem_cutter_mux_sel_i),
    .cutter_sel(mem_cutter_sel_i),
    .hi_mux_sel(mem_hi_mux_sel_i),
    .lo_mux_sel(mem_lo_mux_sel_i),
    .rf_mux_sel(mem_rf_mux_sel_i),

    .mem_mul_hi(mem_mul_hi_o),
    .mem_mul_lo(mem_mul_lo_o),
    .mem_div_r(mem_div_r_o),
    .mem_div_q(mem_div_q_o),
    .mem_clz_out(mem_clz_out_o),
    .mem_alu_out(mem_alu_out_o),
    .mem_dmem_out(mem_dmem_out_o),
    .mem_pc4(mem_pc4_o),
    .mem_rs_data_out(mem_rs_data_out_o),
    .mem_cp0_out(mem_cp0_out_o),
    .mem_hi_out(mem_hi_out_o),
    .mem_lo_out(mem_lo_out_o),
    .mem_rf_waddr(mem_rf_waddr_o),
    .mem_rf_wena(mem_rf_wena_o),
    .mem_hi_wena(mem_hi_wena_o),
    .mem_lo_wena(mem_lo_wena_o),
    .mem_hi_mux_sel(mem_hi_mux_sel_o),
    .mem_lo_mux_sel(mem_lo_mux_sel_o),
    .mem_rf_mux_sel(mem_rf_mux_sel_o)
);

//ME-WB 流水寄存器
ME_WB_Reg mem_wb_reg(
    .clk(clk),
    .rst(rst),
    .wena(mem_wb_wena),
    .mem_mul_hi(mem_mul_hi_o),
    .mem_mul_lo(mem_mul_lo_o),
    .mem_div_r(mem_div_r_o),
    .mem_div_q(mem_div_q_o),
    .mem_clz_out(mem_clz_out_o),
    .mem_alu_out(mem_alu_out_o),
    .mem_dmem_out(mem_dmem_out_o),
    .mem_pc4(mem_pc4_o),
    .mem_rs_data_out(mem_rs_data_out_o),
```

```
    .mem_cp0_out(mem_cp0_out_o),
    .mem_hi_out(mem_hi_out_o),
    .mem_lo_out(mem_lo_out_o),
    .mem_rf_waddr(mem_rf_waddr_o),
    .mem_rf_wena(mem_rf_wena_o),
    .mem_hi_wena(mem_hi_wena_o),
    .mem_lo_wena(mem_lo_wena_o),
    .mem_hi_mux_sel(mem_hi_mux_sel_o),
    .mem_lo_mux_sel(mem_lo_mux_sel_o),
    .mem_rf_mux_sel(mem_rf_mux_sel_o),
    .wb_mul_hi(wb_mul_hi_i),
    .wb_mul_lo(wb_mul_lo_i),
    .wb_div_r(wb_div_r_i),
    .wb_div_q(wb_div_q_i),
    .wb_clz_out(wb_clz_out_i),
    .wb_alu_out(wb_alu_out_i),
    .wb_dmem_out(wb_dmem_out_i),
    .wb_pc4(wb_pc4_i),
    .wb_rs_data_out(wb_rs_data_out_i),
    .wb_cp0_out(wb_cp0_out_i),
    .wb_hi_out(wb_hi_out_i),
    .wb_lo_out(wb_lo_out_i),
    .wb_rf_waddr(wb_rf_waddr_i),
    .wb_rf_wena(wb_rf_wena_i),
    .wb_hi_wena(wb_hi_wena_i),
    .wb_lo_wena(wb_lo_wena_i),
    .wb_hi_mux_sel(wb_hi_mux_sel_i),
    .wb_lo_mux_sel(wb_lo_mux_sel_i),
    .wb_rf_mux_sel(wb_rf_mux_sel_i)
);

//WB 阶段
WB write_back(
    .mul_hi(wb_mul_hi_i),
    .mul_lo(wb_mul_lo_i),
    .div_r(wb_div_q_i),
    .div_q(wb_div_q_i),
    .clz_out(wb_clz_out_i),
    .alu_out(wb_alu_out_i),
    .dmem_out(wb_dmem_out_i),
    .pc4(wb_pc4_i),
    .rs_data_out(wb_rs_data_out_i),
    .cp0_out(wb_cp0_out_i),
    .hi_out(wb_hi_out_i),
```

```
        .lo_out(wb_lo_out_i),
        .rf_waddr(wb_rf_waddr_i),
        .rf_wena(wb_rf_wena_i),
        .hi_wena(wb_hi_wena_i),
        .lo_wena(wb_lo_wena_i),
        .hi_mux_sel(wb_hi_mux_sel_i),
        .lo_mux_sel(wb_lo_mux_sel_i),
        .rf_mux_sel(wb_rf_mux_sel_i),

        .hi_wdata(wb_hi_wdata_o),
        .lo_wdata(wb_lo_wdata_o),
        .rf_wdata(wb_rf_wdata_o),
        .wb_rf_waddr(wb_rf_waddr_o),
        .wb_rf_wena(wb_rf_wena_o),
        .wb_hi_wena(wb_hi_wena_o),
        .wb_lo_wena(wb_lo_wena_o)
    );

endmodule
```

```
`timescale 1ns / 1ps

module cutter(
    input [31:0] data_in,
    input [2:0] sel,
    input sign,
    output [31:0] data_out
    );

    reg [31:0] temp;

    always @ (*) begin
        case(sel)
            3'b001: temp <= {(sign && data_in[15]) ? 16'hffff : 16'h0000, data_in[15:0]};
            3'b010: temp <= {(sign && data_in[7]) ? 24'hffffff : 24'h000000, data_in[7:0]};
            3'b011: temp <= {24'h000000, data_in[7:0]};
            3'b100: temp <= {16'h0000, data_in[15:0]};
            default: temp <= data_in;
        endcase
    end

    assign data_out = temp;
```

```
endmodule
```

```
`define CHIP_ENABLED         1'b1
`define CHIP_DISABLED        1'b0
`define READ_ENABLED         1'b1
`define READ_DISABLED        1'b0
`define UNSIGNED             1'b0
`define SIGNED               1'b1
`define ENABLED              1'b1
`define WRITE_ENABLED        1'b1
`define WRITE_DISABLED       1'b0
`define DISABLED             1'b0
`define RST_ENABLED          1'b1
`define RST_DISABLED         1'b0

`define ZERO_32BIT           32'h00000000

`define EXCEPTION_ADDR       32'h00400004



// CP0 Register Number
`define STATUS              12
`define CAUSE               13
`define EPC                 14
// CP0 Exception Cause
`define BREAK               5'b01001
`define TEQ                 5'b01101
`define SYSCALL             5'b01000

// Stall
`define RUN                 1'b0
`define STOP                1'b1

// Instructions Operand
`define ADDI_OP      6'b001000

`define LUI_OP       6'b001111

`define ANDI_OP      6'b001100
`define ORI_OP       6'b001101
`define BNE_OP       6'b000101
`define J_OP         6'b000010
```

```
`define JAL_OP      6'b000011
`define JR_OP       6'b000000
`define LW_OP       6'b100011
`define SLTIU_OP    6'b001011
`define ADDU_OP     6'b000000
`define AND_OP      6'b000000
`define BEQ_OP      6'b000100

`define XORI_OP     6'b001110
`define SLTI_OP     6'b001010
`define ADDIU_OP    6'b001001
`define XOR_OP      6'b000000
`define NOR_OP      6'b000000
`define OR_OP       6'b000000

`define SW_OP       6'b101011
`define ADD_OP      6'b000000
`define SUB_OP      6'b000000
`define SLT_OP      6'b000000
`define SRLV_OP     6'b000000
`define SRAV_OP     6'b000000
`define CLZ_OP      6'b011100
`define DIVU_OP     6'b000000
`define ERET_OP     6'b010000
`define JALR_OP     6'b000000
`define LB_OP       6'b100000

`define SLL_OP      6'b000000
`define SLLV_OP     6'b000000
`define SLTU_OP     6'b000000
`define SRA_OP      6'b000000
`define SRL_OP      6'b000000
`define SUBU_OP     6'b000000
`define LHU_OP      6'b100101
`define SB_OP       6'b101000
`define SH_OP       6'b101001
`define LH_OP       6'b100001
`define MFHI_OP     6'b000000
`define MFLO_OP     6'b000000
`define BGEZ_OP     6'b000001
`define MTHI_OP     6'b000000
`define MTLO_OP     6'b000000
`define MUL_OP      6'b011100
`define MULTU_OP    6'b000000
```

```
`define SYSCALL_OP   6'b000000
`define TEQ_OP       6'b000000
`define LBU_OP       6'b100100
`define BREAK_OP     6'b000000
`define DIV_OP       6'b000000

// Instruction Func
`define ADDU_FUNC      6'b100001
`define AND_FUNC       6'b100100
`define SRLV_FUNC      6'b000110
`define SRAV_FUNC      6'b000111
`define ERET_FUNC      6'b011000
`define TEQ_FUNC       6'b110100
`define BREAK_FUNC     6'b001101
`define JR_FUNC        6'b001000
`define XOR_FUNC        6'b100110
`define OR_FUNC        6'b100101
`define SLL_FUNC       6'b000000
`define SLLV_FUNC      6'b000100
`define SLTU_FUNC      6'b101011
`define SRA_FUNC       6'b000011
`define NOR_FUNC       6'b100111

`define SRL_FUNC       6'b000010
`define SUBU_FUNC      6'b100011
`define ADD_FUNC       6'b100000

`define CLZ_FUNC       6'b100000
`define DIVU_FUNC      6'b011011
`define SUB_FUNC       6'b100010
`define SLT_FUNC       6'b101010

`define JALR_FUNC      6'b001001
`define MFHI_FUNC      6'b010000
`define MFLO_FUNC      6'b010010
`define MTHI_FUNC      6'b010001
`define MTLO_FUNC      6'b010011
`define MUL_FUNC       6'b000010
`define MULTU_FUNC     6'b011001
`define SYSCALL_FUNC   6'b001100

`define DIV_FUNC       6'b011010
```

```
`timescale 1ns / 1ps
`include "define.vh"
module divider(
    //input clk,
    input reset,
    input ena,
    input sign,
    input [31:0] dividend,
    input [31:0] divisor,
    output [31:0] q,  // Quotient
    output [31:0] r   // Remainder
    );

    reg [63:0] temp_dividend;
    reg [63:0] temp_divisor;
    reg is_minus;
    reg is_div_minus;
    integer counter;

    always @ (*)
    begin
        if (reset == `RST_ENABLED) begin
            temp_dividend <= 0;
            temp_divisor <= 0;
            is_minus <= 0;
            is_div_minus <= 0;
        end

        else if(ena == `ENABLED) begin
            // Unsigned
            if(sign == `UNSIGNED) begin
                temp_dividend = dividend;
                temp_divisor = {divisor, 32'b0};

                for (counter = 0; counter < 32; counter = counter + 1) begin
                    temp_dividend = temp_dividend << 1;

                    if (temp_dividend >= temp_divisor) begin
                        temp_dividend = temp_dividend - temp_divisor;
                        temp_dividend = temp_dividend + 1;
                    end
                end

                counter = 0;
```

```
        end

        // Signed
        else begin
            temp_dividend = dividend;
            temp_divisor = {divisor, 32'b0};
            is_minus = dividend[31] ^ divisor[31];   //judge
            is_div_minus = dividend[31];

            if (dividend[31] == 1) begin
            temp_dividend = dividend ^ 32'hffffffff;
            temp_dividend = temp_dividend + 1;
            end

            if (divisor[31] == 1) begin
                temp_divisor = {divisor ^ 32'hffffffff, 32'b0};
                temp_divisor = temp_divisor + 64'h0000000100000000;
            end

            for (counter = 0; counter < 32; counter = counter + 1) begin
                temp_dividend = temp_dividend << 1;

                if (temp_dividend >= temp_divisor) begin
                    temp_dividend = temp_dividend - temp_divisor;
                    temp_dividend = temp_dividend + 1;
                end
            end

            if (is_div_minus == 1) begin
                temp_dividend = temp_dividend ^ 64'hffffffff00000000;
                temp_dividend = temp_dividend + 64'h0000000100000000;
            end

            if (is_minus == 1) begin
                temp_dividend = temp_dividend ^ 64'h00000000ffffffff;
                temp_dividend = temp_dividend + 64'h0000000000000001;

                if (temp_dividend[31:0] == 32'b0) begin
                    temp_dividend = temp_dividend - 64'h0000000100000000;
                end
            end
        end
    end
end
```

```
    assign q = (ena == `CHIP_ENABLED) ? temp_dividend[31:0] : 32'b0;
    assign r = (ena == `CHIP_ENABLED) ? temp_dividend[63:32]: 32'b0;

endmodule
```

```
`timescale 1ns / 1ps

`include "define.vh"

module dmem (
    input clk,
    input ena,
    input wena,
    input [1:0] w_cs,
    input [1:0] r_cs,
    input [31:0] data_in,
    input [31:0] addr,
    output reg [31:0] data_out
    );

    parameter SW = 2'b01;
    parameter SH = 2'b10;
    parameter SB = 2'b11;
    parameter LW = 2'b01;
    parameter LH = 2'b10;
    parameter LB = 2'b11;

    reg [31:0] temp [1023:0];

    // Write
    always @ (posedge clk)
    begin
        if(ena == `ENABLED) begin
            if(wena == `WRITE_ENABLED) begin
                if(w_cs == SW) begin
                    temp[(addr - 32'h10010000) / 4] <= data_in;
                end

                else if(w_cs == SH) begin
                    case((addr - 32'h10010000) % 4)
                        32'h0: temp[(addr - 32'h10010000) / 4][15:0] <= data_in[15:0];
```

```
                    32'h2: temp[(addr - 32'h10010000) / 4][31:16] <= data_in[15:0];
                endcase
            end

            else begin
                case((addr - 32'h10010000) % 4)
                    32'h0: temp[(addr - 32'h10010000) / 4][7:0] <= data_in[7:0];
                    32'h1: temp[(addr - 32'h10010000) / 4][15:8] <= data_in[7:0];
                    32'h2: temp[(addr - 32'h10010000) / 4][23:16] <= data_in[7:0];
                    32'h3: temp[(addr - 32'h10010000) / 4][31:24] <= data_in[7:0];
                endcase
            end
        end
    end

    // Read
    always @ (*)
    begin
        if(ena == `ENABLED && wena == `WRITE_DISABLED) begin
            if(r_cs == LW) begin
                data_out <= temp[(addr - 32'h10010000) / 4];
            end

            else if(r_cs == LH) begin
                case((addr - 32'h10010000) % 4)
                    32'h0: data_out <= temp[(addr - 32'h10010000) / 4][15:0];
                    32'h2: data_out <= temp[(addr - 32'h10010000) / 4][31:16];
                endcase
            end

            else begin
                case((addr - 32'h10010000) % 4)
                    32'h0: data_out <= temp[(addr - 32'h10010000) / 4][7:0];
                    32'h1: data_out <= temp[(addr - 32'h10010000) / 4][15:8];
                    32'h2: data_out <= temp[(addr - 32'h10010000) / 4][23:16];
                    32'h3: data_out <= temp[(addr - 32'h10010000) / 4][31:24];
                endcase
            end
        end
    end

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"

module EX_ME_Reg(
    input clk,
    input rst,
    input wena,
    input [31:0] exe_mul_hi,
    input [31:0] exe_mul_lo,
    input [31:0] exe_div_r,
    input [31:0] exe_div_q,
    input [31:0] exe_clz_out,
    input [31:0] exe_alu_out,
    input [31:0] exe_pc4,
    input [31:0] exe_rs_data_out,
    input [31:0] exe_rt_data_out,
    input [31:0] exe_cp0_out,
    input [31:0] exe_hi_out,
    input [31:0] exe_lo_out,
    input [4:0] exe_rf_waddr,
    input exe_dmem_ena,
    input exe_cutter_sign,
    input exe_rf_wena,
    input exe_hi_wena,
    input exe_lo_wena,
    input exe_dmem_wena,
    input [1:0] exe_dmem_w_cs,
    input [1:0] exe_dmem_r_cs,
    input exe_cutter_mux_sel,
    input [1:0] exe_hi_mux_sel,
    input [1:0] exe_lo_mux_sel,
    input [2:0] exe_cutter_sel,
    input [2:0] exe_rf_mux_sel,
    output reg [31:0] mem_mul_hi,
    output reg [31:0] mem_mul_lo,
    output reg [31:0] mem_div_r,
    output reg [31:0] mem_div_q,
    output reg [31:0] mem_clz_out,
    output reg [31:0] mem_alu_out,
    output reg [31:0] mem_pc4,
    output reg [31:0] mem_rs_data_out,
    output reg [31:0] mem_rt_data_out,
    output reg [31:0] mem_cp0_out,
```

```
    output reg [31:0] mem_hi_out,
    output reg [31:0] mem_lo_out,
    output reg [4:0] mem_rf_waddr,
    output reg mem_dmem_ena,
    output reg mem_cutter_sign,
    output reg mem_rf_wena,
    output reg mem_hi_wena,
    output reg mem_lo_wena,
    output reg mem_dmem_wena,
    output reg [1:0] mem_dmem_w_cs,
    output reg [1:0] mem_dmem_r_cs,
    output reg mem_cutter_mux_sel,
    output reg [1:0] mem_hi_mux_sel,
    output reg [1:0] mem_lo_mux_sel,
    output reg [2:0] mem_cutter_sel,
    output reg [2:0] mem_rf_mux_sel
    );

    always @ (posedge clk or posedge rst)
    begin
        if(rst == `RST_ENABLED) begin
            mem_mul_hi <= 0;
            mem_mul_lo <= 0;
            mem_div_r <= 0;
            mem_div_q <= 0;
            mem_clz_out <= 0;
            mem_alu_out <= 0;
            mem_pc4 <= 0;
            mem_rs_data_out <= 0;
            mem_rt_data_out <= 0;
            mem_cp0_out <= 0;
            mem_hi_out <= 0;
            mem_lo_out <= 0;
            mem_rf_waddr <= 0;
            mem_dmem_ena <= 0;
            mem_cutter_sign <= 0;
            mem_rf_wena <= 0;
            mem_hi_wena <= 0;
            mem_lo_wena <= 0;
            mem_dmem_wena <= 0;
            mem_dmem_w_cs <= 0;
            mem_dmem_r_cs <= 0;
            mem_cutter_mux_sel <= 0;
            mem_hi_mux_sel <= 0;
```

```
            mem_lo_mux_sel <= 0;
            mem_cutter_sel <= 0;
            mem_rf_mux_sel <= 0;
        end

        else if(wena == `WRITE_ENABLED) begin
            mem_mul_hi <= exe_mul_hi;
            mem_mul_lo <= exe_mul_lo;
            mem_div_r <= exe_div_r;
            mem_div_q <= exe_div_q;
            mem_clz_out <= exe_clz_out;
            mem_alu_out <= exe_alu_out;
            mem_pc4 <= exe_pc4;
            mem_rs_data_out <= exe_rs_data_out;
            mem_rt_data_out <= exe_rt_data_out;
            mem_cp0_out <= exe_cp0_out;
            mem_hi_out <= exe_hi_out;
            mem_lo_out <= exe_lo_out;
            mem_rf_waddr <= exe_rf_waddr;
            mem_dmem_ena <= exe_dmem_ena;
            mem_cutter_sign <= exe_cutter_sign;
            mem_rf_wena <= exe_rf_wena;
            mem_hi_wena <= exe_hi_wena;
            mem_lo_wena <= exe_lo_wena;
            mem_dmem_wena <= exe_dmem_wena;
            mem_dmem_w_cs <= exe_dmem_w_cs;
            mem_dmem_r_cs <= exe_dmem_r_cs;
            mem_cutter_mux_sel <= exe_cutter_mux_sel;
            mem_hi_mux_sel <= exe_hi_mux_sel;
            mem_lo_mux_sel <= exe_lo_mux_sel;
            mem_cutter_sel <= exe_cutter_sel;
            mem_rf_mux_sel <= exe_rf_mux_sel;
        end
    end

endmodule
```

```
`timescale 1ns / 1ps

module EX(
    input rst,
    input [31:0] pc4,
    input [31:0] rs_data_out,
```

```
    input [31:0] rt_data_out,
    input [31:0] imm,
    input [31:0] shamt,
    input [31:0] cp0_out,
    input [31:0] hi_out,
    input [31:0] lo_out,
    input [4:0] rf_waddr,
    input [3:0] aluc,
    input mul_ena,
    input div_ena,
    input clz_ena,
    input dmem_ena,
    input mul_sign,
    input div_sign,
    input cutter_sign,
    input rf_wena,
    input hi_wena,
    input lo_wena,
    input dmem_wena,
    input [1:0] dmem_w_cs,
    input [1:0] dmem_r_cs,
    input cutter_mux_sel,
    input alu_mux1_sel,
    input [1:0] alu_mux2_sel,
    input [1:0] hi_mux_sel,
    input [1:0] lo_mux_sel,
    input [2:0] cutter_sel,
    input [2:0] rf_mux_sel,
    output [31:0] exe_mul_hi,  // Higher bits of the multiplication result
    output [31:0] exe_mul_lo,  // Lower bits of the multiplication result
    output [31:0] exe_div_r,  // Remainder of the division result
    output [31:0] exe_div_q,  // Quotient of the division result
    output [31:0] exe_clz_out,
    output [31:0] exe_alu_out,  // Result of ALU
    output [31:0] exe_pc4,
    output [31:0] exe_rs_data_out,
    output [31:0] exe_rt_data_out,
    output [31:0] exe_cp0_out,
    output [31:0] exe_hi_out,
    output [31:0] exe_lo_out,
    output [4:0] exe_rf_waddr,
    output exe_dmem_ena,
    output exe_rf_wena,
    output exe_hi_wena,
```

```
output exe_lo_wena,
output exe_dmem_wena,
output [1:0] exe_dmem_w_cs,
output [1:0] exe_dmem_r_cs,
output exe_cutter_sign,
output exe_cutter_mux_sel,
output [2:0] exe_cutter_sel,
output [1:0] exe_hi_mux_sel,
output [1:0] exe_lo_mux_sel,
output [2:0] exe_rf_mux_sel
);

wire zero;
wire carry;
wire negative;
wire overflow;
wire [31:0] alu_in1;
wire [31:0] alu_in2;

assign exe_pc4 = pc4;
assign exe_rs_data_out = rs_data_out;
assign exe_rt_data_out = rt_data_out;
assign exe_hi_out = hi_out;
assign exe_lo_out = lo_out;
assign exe_rf_waddr = rf_waddr;
assign exe_cp0_out = cp0_out;
assign exe_dmem_ena = dmem_ena;
assign exe_rf_wena = rf_wena;
assign exe_hi_wena = hi_wena;
assign exe_lo_wena = lo_wena;
assign exe_dmem_wena = dmem_wena;
assign exe_dmem_r_cs = dmem_r_cs;
assign exe_dmem_w_cs = dmem_w_cs;
assign exe_cutter_sign = cutter_sign;
assign exe_cutter_mux_sel = cutter_mux_sel;
assign exe_cutter_sel = cutter_sel;
assign exe_hi_mux_sel = hi_mux_sel;
assign exe_lo_mux_sel = lo_mux_sel;
assign exe_rf_mux_sel = rf_mux_sel;

multiplier cpu_multiplier(
    .reset(rst),
    .ena(mul_ena),
    .sign(mul_sign),
```

```
    .a(rs_data_out),
    .b(rt_data_out),
    .mul_hi(exe_mul_hi),
    .mul_lo(exe_mul_lo)
);

divider cpu_divider(
    .reset(rst),
    .ena(div_ena),
    .sign(div_sign),
    .dividend(rs_data_out),
    .divisor(rt_data_out),
    .q(exe_div_q),
    .r(exe_div_r)
);

lead0_counter cpu_lead0_coutner(
    .data_in(rs_data_out),
    .ena(clz_ena),
    .data_out(exe_clz_out)
);

mux_2_32 alu_mux1(
    .C0(shamt),
    .C1(rs_data_out),
    .S0(alu_mux1_sel),
    .oZ(alu_in1)
);

mux_4_32 alu_mux2(
    .C0(rt_data_out),
    .C1(imm),
    .C2(32'b0),
    .C3(32'b0),
    .S0(alu_mux2_sel),
    .oZ(alu_in2)
);

alu cpu_alu(
    .a(alu_in1),
    .b(alu_in2),
    .aluc(aluc),
    .r(exe_alu_out),
    .zero(zero),
```

```
        .carry(carry),
        .negative(negative),
        .overflow(overflow)
    );

endmodule
```

```
`timescale 1ns / 1ps

module extend_5_32 (
    input [4:0] a,
    output [31:0] b
    );

    assign b = {27'b0, a};

endmodule
```

```
`timescale 1ns / 1ps

module extend_16_32 (
    input [15:0] data_in,
    input sign,
    output [31:0] data_out
    );

    assign data_out = (sign == 0 || data_in[15] == 0) ? {16'b0, data_in} : {16'hffff, data_in};

endmodule
```

```
`timescale 1ns / 1ns

module extend_sign_18_32 (
    input [17:0] data_in,
    output [31:0] data_out
    );

    assign data_out = (data_in[17] == 0) ? {14'b0, data_in} : {14'b11111111111111, data_in};

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"
module id_df(
    input clk,
    input rst,
    input [5:0] op,
    input [5:0] func,
    input [4:0] rs,
    input [4:0] rt,
    input rf_rena1,
    input rf_rena2,

    // Data from EXE
    input [4:0] exe_rf_waddr,
    input exe_rf_wena,
    input exe_hi_wena,
    input exe_lo_wena,
    input [31:0] exe_df_hi_out,
    input [31:0] exe_df_lo_out,
    input [31:0] exe_df_rf_wdata,
    input [5:0] exe_op,
    input [5:0] exe_func,

    // Data from MEM
    input [4:0] mem_rf_waddr,
    input mem_rf_wena,
    input mem_hi_wena,
    input mem_lo_wena,
    input [31:0] mem_df_hi_out,
    input [31:0] mem_df_lo_out,
    input [31:0] mem_df_rf_wdata,

    output reg [31:0] rs_data_out,
    output reg [31:0] rt_data_out,
    output reg [31:0] hi_out,     //output data of hi register
    output reg [31:0] lo_out,     //output data of lo register
    output reg stall,
    output reg if_df,
    output reg is_rs,
    output reg is_rt
    );
```

```
// Data hazard judge for ID and EXE and MEM
always @ (negedge clk or posedge rst)
begin
    if(rst == `RST_ENABLED) begin
        stall <= `RUN;
        rs_data_out <= 0;
        rt_data_out <= 0;
        hi_out <= 0;
        lo_out <= 0;
        if_df <= 0;
        is_rs <= 0;
        is_rt <= 0;
    end

    else if(stall == `STOP) begin
        stall <= `RUN;

        if(is_rs == 1'b1) begin
            rs_data_out <= mem_df_rf_wdata;
        end
        else begin
            rt_data_out <= mem_df_rf_wdata;
        end
    end

    else if(stall == `RUN) begin
        if_df = 0;
        is_rs = 0;
        is_rt = 0;
        // MFHI
        if(op == `MFHI_OP && func == `MFHI_FUNC) begin
            // EXE HI
            if(exe_hi_wena == `WRITE_ENABLED) begin
                hi_out <= exe_df_hi_out;
                if_df <= 1;
            end
            // MEM HI
            else if(mem_hi_wena == `WRITE_ENABLED) begin
                hi_out <= mem_df_hi_out;
                if_df <= 1;
            end
        end
```

```
            // MFLO
            else if(op == `MFLO_OP && func == `MFLO_FUNC) begin
                // EXE LO
                if(exe_lo_wena == `WRITE_ENABLED) begin
                    lo_out <= exe_df_lo_out;
                    if_df <= 1;
                end
                // MEM LO
                else if(mem_lo_wena == `WRITE_ENABLED) begin
                    lo_out <= mem_df_lo_out;
                    if_df <= 1;
                end
            end

            // Rs and Rt
            else begin
                // EXE Rs
                if(exe_rf_wena == `WRITE_ENABLED && rf_rena1 == `READ_ENABLED && exe_rf_waddr
== rs) begin
                    // Lw, Lh, Lb, Lbu, Lhu
                    if(exe_op == `LW_OP || exe_op == `LH_OP || exe_op == `LHU_OP || exe_op
== `LB_OP || exe_op == `LBU_OP) begin
                        stall <= `STOP;
                        is_rs <= 1;
                        if_df <= 1;
                    end
                    else begin
                        rs_data_out <= exe_df_rf_wdata;
                        if_df <= 1;
                        is_rs <= 1;
                    end
                end
                // MEM Rs
                else if(mem_rf_wena == `WRITE_ENABLED && rf_rena1 == `READ_ENABLED &&
mem_rf_waddr == rs) begin
                    rs_data_out <= mem_df_rf_wdata;
                    if_df <= 1;
                    is_rs <= 1;
                end

                // EXE Rt
                if(exe_rf_wena == `WRITE_ENABLED && rf_rena2 == `READ_ENABLED && exe_rf_waddr
== rt) begin
                    // Lw, Lh, Lb, Lbu, Lhu
```

```
                    if(exe_op == `LW_OP || exe_op == `LH_OP || exe_op == `LHU_OP || exe_op
== `LB_OP || exe_op == `LBU_OP) begin
                        stall <= `STOP;
                        is_rt <= 1;
                        if_df <= 1;
                    end
                    else begin
                        rt_data_out <= exe_df_rf_wdata;
                        if_df <= 1;
                        is_rt <= 1;
                    end
                end
                // MEM Rt
                else if(mem_rf_wena == `WRITE_ENABLED && rf_rena2 == `READ_ENABLED &&
mem_rf_waddr == rt) begin
                    rt_data_out <= mem_df_rf_wdata;
                    if_df <= 1;
                    is_rt <= 1;
                end
            end
        end
    end

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"
module ID_EX_Reg(
    input clk,
    input rst,
    input wena,                         // 输入写使能信号
    input [31:0] id_pc4,                // 输入来自 ID 阶段的下一条 PC 值
    input [31:0] id_rs_data_out,         // 输入来自 ID 阶段的 rs 寄存器数据
    input [31:0] id_rt_data_out,         // 输入来自 ID 阶段的 rt 寄存器数据
    input [31:0] id_imm,
    input [31:0] id_shamt,
    input [31:0] id_cp0_out,            // 输入来自 ID 阶段的 CP0 输出
    input [31:0] id_hi_out,             // 输入来自 ID 阶段的 hi 寄存器输出
    input [31:0] id_lo_out,             // 输入来自 ID 阶段的 lo 寄存器输出
    input [4:0] id_rf_waddr,            // 输入来自 ID 阶段的寄存器写地址
    input id_clz_ena,                   // 输入来自 ID 阶段的 clz 使能信号
    input id_mul_ena,
```

```
input id_div_ena,
input id_dmem_ena,
input id_mul_sign,
input id_div_sign,
input id_cutter_sign,
input [3:0] id_aluc,                // 输入来自 ID 阶段的 ALU 控制信号
input id_rf_wena,                   // 输入来自 ID 阶段的寄存器文件写使能信号
input id_hi_wena,
input id_lo_wena,
input id_dmem_wena,                 // 输入来自 ID 阶段的数据存储器写使能信号
input [1:0] id_dmem_w_cs,           // 输入来自 ID 阶段的数据存储器写控制信号
input [1:0] id_dmem_r_cs,           // 输入来自 ID 阶段的数据存储器读控制信号
input stall,                        // 输入流水线暂停信号
input id_cutter_mux_sel,            // 输入来自 ID 阶段的截断器选择信号
input id_alu_mux1_sel,
input [1:0] id_alu_mux2_sel,
input [1:0] id_hi_mux_sel,          // 输入来自 ID 阶段的 hi 寄存器写入选择信号
input [1:0] id_lo_mux_sel,          // 输入来自 ID 阶段的 lo 寄存器写入选择信号
input [2:0] id_cutter_sel,          // 输入来自 ID 阶段的截断器选择信号
input [2:0] id_rf_mux_sel,
input [5:0] id_op,                  // 输入来自 ID 阶段的操作码
input [5:0] id_func,                // 输入来自 ID 阶段的功能码
output reg [31:0] exe_pc4,
output reg [31:0] exe_rs_data_out,
output reg [31:0] exe_rt_data_out,
output reg [31:0] exe_imm,          // 输出到 EX 阶段的立即数
output reg [31:0] exe_shamt,
output reg [31:0] exe_cp0_out,      // 输出到 EX 阶段的 CP0 输出
output reg [31:0] exe_hi_out,
output reg [31:0] exe_lo_out,
output reg [4:0] exe_rf_waddr,
output reg exe_clz_ena,             // 输出到 EX 阶段的 clz 使能信号
output reg exe_mul_ena,             // 输出到 EX 阶段的乘法器使能信号
output reg exe_div_ena,             // 输出到 EX 阶段的除法器使能信号
output reg exe_dmem_ena,
output reg exe_mul_sign,            // 输出到 EX 阶段的乘法器符号扩展信号
output reg exe_div_sign,            // 输出到 EX 阶段的除法器符号扩展信号
output reg exe_cutter_sign,
output reg [3:0] exe_aluc,          // 输出到 EX 阶段的 ALU 控制信号
output reg exe_rf_wena,             // 输出到 EX 阶段的寄存器文件写使能信号
output reg exe_hi_wena,             // 输出到 EX 阶段的 hi 寄存器写使能信号
output reg exe_lo_wena,
output reg exe_dmem_wena,           // 输出到 EX 阶段的数据存储器写使能信号
output reg [1:0] exe_dmem_w_cs,
```

```
output reg [1:0] exe_dmem_r_cs,     // 输出到 EX 阶段的数据存储器读控制信号
output reg exe_alu_mux1_sel,
output reg [1:0] exe_alu_mux2_sel,
output reg exe_cutter_mux_sel,      // 输出到 EX 阶段的截断器选择信号
output reg [1:0] exe_hi_mux_sel,
output reg [1:0] exe_lo_mux_sel,    // 输出到 EX 阶段的 lo 寄存器写入选择信号
output reg [2:0] exe_cutter_sel,    // 输出到 EX 阶段的截断器选择信号
output reg [2:0] exe_rf_mux_sel,
output reg [5:0] exe_op,            // 输出到 EX 阶段的操作码
output reg [5:0] exe_func           // 输出到 EX 阶段的功能码
);

always @ (posedge clk or posedge rst) begin
    if(rst == `RST_ENABLED || stall == `STOP) begin
        exe_imm <= 32'b0;
        exe_shamt <= 32'b0;
        exe_cp0_out <= 32'b0;
        exe_mul_sign <= 1'b0;
        exe_div_sign <= 1'b0;
        exe_cutter_sign <= 1'b0;
        exe_aluc <= 4'b0;
        exe_dmem_r_cs <= 2'b0;
        exe_alu_mux1_sel <= 1'b0;
        exe_alu_mux2_sel <= 1'b0;
        exe_rf_wena <= 1'b0;
        exe_pc4 <= 32'b0;
        exe_rs_data_out <= 32'b0;
        exe_rt_data_out <= 32'b0;
        exe_hi_wena <= 1'b0;
        exe_lo_wena <= 1'b0;
        exe_hi_out <= 32'b0;
        exe_lo_out <= 32'b0;
        exe_rf_waddr <= 5'b0;
        exe_clz_ena <= 1'b0;
        exe_mul_ena <= 1'b0;
        exe_div_ena <= 1'b0;
        exe_dmem_ena <= 1'b0;

        exe_dmem_wena <= 1'b0;
        exe_dmem_w_cs <= 2'b0;

        exe_cutter_mux_sel <= 1'b0;
        exe_hi_mux_sel <= 2'b0;
        exe_lo_mux_sel <= 2'b0;
```

```
            exe_cutter_sel <= 3'b0;
            exe_rf_mux_sel <= 3'b0;
            exe_op <= 6'b0;
            exe_func <= 6'b0;
        end

        else if(wena == `WRITE_ENABLED) begin

            exe_alu_mux1_sel <= id_alu_mux1_sel;
            exe_mul_sign <= id_mul_sign;
            exe_div_sign <= id_div_sign;
            exe_cutter_sign <= id_cutter_sign;
            exe_aluc <= id_aluc;
            exe_rf_wena <= id_rf_wena;
            exe_hi_wena <= id_hi_wena;

            exe_lo_mux_sel <= id_lo_mux_sel;
            exe_cutter_sel <= id_cutter_sel;
            exe_rf_mux_sel <= id_rf_mux_sel;
            exe_alu_mux2_sel <= id_alu_mux2_sel;
            exe_imm <= id_imm;
            exe_shamt <= id_shamt;
            exe_cp0_out <= id_cp0_out;
            exe_lo_wena <= id_lo_wena;
            exe_dmem_wena <= id_dmem_wena;
            exe_dmem_w_cs <= id_dmem_w_cs;
            exe_dmem_r_cs <= id_dmem_r_cs;
            exe_cutter_mux_sel <= id_cutter_mux_sel;
            exe_hi_mux_sel <= id_hi_mux_sel;
            exe_hi_out <= id_hi_out;
            exe_lo_out <= id_lo_out;
            exe_rf_waddr <= id_rf_waddr;
            exe_clz_ena <= id_clz_ena;
            exe_mul_ena <= id_mul_ena;
            exe_div_ena <= id_div_ena;
            exe_pc4 <= id_pc4;
            exe_rs_data_out <= id_rs_data_out;
            exe_rt_data_out <= id_rt_data_out;
            exe_dmem_ena <= id_dmem_ena;

            exe_op <= id_op;
            exe_func <= id_func;
        end
    end
```

```
endmodule
```

```
`timescale 1ns / 1ps

module ID(
    input clk,
    input rst,
    input [31:0] pc4,
    input [31:0] rf_wdata,    //Regfile 写数据
    input [31:0] hi_wdata,
    input [31:0] lo_wdata,
    input [31:0] instruction,
    input rf_wena,
    input hi_wena,
    input [4:0] rf_waddr,  // Regfile 写地址

    input lo_wena,

    // Data from EXE
    input [4:0] exe_rf_waddr,
    input exe_rf_wena,

    input [31:0] exe_mul_hi,  // 乘法结果高位
    input [31:0] exe_mul_lo,  // 乘法结果低位
    input [31:0] exe_div_r,  // 余数
    input [31:0] exe_div_q,  // 商
    input [31:0] exe_rs_data_out,
    input [31:0] exe_lo_out,
    input [31:0] exe_pc4,
    input [31:0] exe_clz_out,
    input exe_hi_wena,
    input exe_lo_wena,

    input [31:0] exe_alu_out,
    input [31:0] exe_hi_out,
    input [2:0] exe_rf_mux_sel,
    input [5:0] exe_op,
    input [1:0] exe_hi_mux_sel,
    input [1:0] exe_lo_mux_sel,

    input [5:0] exe_func,
```

```
    // Data from MEM
    input [4:0] mem_rf_waddr,
    input mem_rf_wena,
    input mem_hi_wena,
    input [31:0] mem_mul_hi,
    input [31:0] mem_mul_lo,
    input [31:0] mem_div_r,
    input mem_lo_wena,
    input [31:0] mem_div_q,
    input [31:0] mem_rs_data_out,
    input [31:0] mem_dmem_out,
    input [31:0] mem_lo_out,
    input [31:0] mem_pc4,
    input [31:0] mem_alu_out,
    input [31:0] mem_hi_out,
    input [1:0] mem_hi_mux_sel,
    input [31:0] mem_clz_out,
    input [1:0] mem_lo_mux_sel,
    input [2:0] mem_rf_mux_sel,

    output [31:0] id_cp0_pc,  // 中断地址

    output [31:0] id_imm,  // 立即数
    output [31:0] id_shamt,
    output [31:0] id_pc4,
    output [31:0] id_rs_data_out,
    output [31:0] id_rt_data_out,
    output [31:0] id_r_pc,
    output [31:0] id_b_pc,  // 跳转地址
    output [31:0] id_j_pc,  // Beq and Bne address

    output [4:0] id_rf_waddr,
    output [3:0] id_aluc,
    output [31:0] id_cp0_out,    //cp0 输出结果
    output [31:0] id_hi_out,     //output data of hi register
    output [31:0] id_lo_out,     //output data of lo register

    output id_mul_sign,
    output id_div_sign,
    output id_cutter_sign,
    output id_clz_ena,
    output id_dmem_ena,
    output id_hi_wena,
    output id_lo_wena,
```

```
output id_rf_wena,
output id_mul_ena,
output id_div_ena,

output id_dmem_wena,
output id_cutter_mux_sel,
output id_alu_mux1_sel,
output [1:0] id_dmem_w_cs,
output [1:0] id_dmem_r_cs,

output [2:0] id_cutter_sel,
output [2:0] id_rf_mux_sel,
output [1:0] id_alu_mux2_sel,
output [1:0] id_hi_mux_sel,
output [1:0] id_lo_mux_sel,
output [2:0] id_pc_mux_sel,
output [5:0] id_op,
output [5:0] id_func,
output stall,
output is_branch,
output [31:0] reg28,
output [31:0] reg29
);

// Regfile
wire [4:0] rs;          // 源寄存器 rs
wire [4:0] rt;          // 源寄存器 rt
wire [4:0] rd;          // 目标寄存器 rd（写入寄存器文件的目标地址）
wire [5:0] op;          // 操作码 op
wire [5:0] func;        // 功能码 func
wire rf_rena1;
wire rf_rena2;

wire [15:0] ext16_data_in;    // extend16 模块的输入数据
wire [17:0] ext18_data_in;    // extend18 模块的输入数据
wire [31:0] ext16_data_out;   // extend16 模块的输出数据
wire [31:0] ext18_data_out;   // extend18 模块的输出数据
wire ext16_sign;             // extend16 模块的符号位
wire mfc0;                   // MFC0 指令信号
wire mtc0;                   // MTC0 指令信号
wire eret;                   // ERET 指令信号


// 数据转发
```

```
 wire if_df;                // 数据转发使能信号
 wire is_rs;                // 是否为源寄存器 rs 数据
 wire is_rt;                // 是否为源寄存器 rt 数据
 wire [31:0] exe_df_hi_out;
 wire [31:0] exe_df_lo_out;
 wire [31:0] df_hi_temp;
 wire [31:0] df_lo_temp;
 wire [31:0] df_rs_temp;
 wire [31:0] df_rt_temp;
 wire [31:0] exe_df_rf_wdata; // 执行阶段数据转发的寄存器文件写入数据
 wire [31:0] mem_df_hi_out; // 存储阶段数据转发的 hi 寄存器输出
 wire [31:0] mem_df_lo_out; // 存储阶段数据转发的 lo 寄存器输出
 wire [31:0] mem_df_rf_wdata; // 存储阶段数据转发的寄存器文件写入数据


 // CP0 ports
 wire cp0_exception;
 wire [4:0] cp0_addr;
 wire [31:0] cp0_status;
 wire [4:0] cp0_cause;

// Ext5
 wire [31:0] hi_temp;    // hi 寄存器临时数据
 wire [31:0] lo_temp;    // lo 寄存器临时数据
 wire [31:0] rs_temp;    // 源寄存器 rs 临时数据
 wire [31:0] rt_temp;    // 源寄存器 rt 临时数据
 wire ext5_mux_sel;     // 用于选择 Ext5 的多路选择器的控制信号
 wire [4:0] ext5_mux_out;  // Ext5 多路选择器的输出

 assign rs = instruction[25:21];          // 源寄存器 rs 地址字段
 assign rt = instruction[20:16];          // 源寄存器 rt 地址字段
 assign op = instruction[31:26];          // 操作码字段

 assign func = instruction[5:0];          // 功能码字段
 assign ext16_data_in = instruction[15:0]; // 16 位扩展的输入数据
 assign ext18_data_in = {instruction[15:0], 2'b00}; // 18 位扩展的输入数据

 assign id_pc4 = pc4;                      // ID 阶段的 PC + 4
 assign id_rf_waddr = rd;                  // ID 阶段的寄存器文件写入地址
 assign id_imm = ext16_data_out;           // ID 阶段的立即数
 assign id_j_pc = {pc4[31:28], instruction[25:0], 2'b00}; // ID 阶段的跳转地址
 assign id_r_pc = id_rs_data_out;          // ID 阶段的寄存器读地址
```

```
    assign id_rs_data_out = (if_df && is_rs) ? df_rs_temp : rs_temp; // ID 阶段的源寄存器 rs 数据输出
    assign id_rt_data_out = (if_df && is_rt) ? df_rt_temp : rt_temp; // ID 阶段的源寄存器 rt 数据输出
    assign id_op = op;                      // ID 阶段的操作码
    assign id_func = func;                  // ID 阶段的功能码
    assign id_hi_out = if_df ? df_hi_temp : hi_temp;  // ID 阶段的 hi 寄存器数据输出
    assign id_lo_out = if_df ? df_lo_temp : lo_temp;  // ID 阶段的 lo 寄存器数据输出


   id_df data_forwarding(
    .clk(clk),
    .rst(rst),
    .op(op),                        // 操作码信号
    .func(func),                    // 功能码信号
    .rs(rs),
    .rt(rt),
    .rf_rena1(rf_rena1),            // 寄存器文件读使能信号 1
    .rf_rena2(rf_rena2),            // 寄存器文件读使能信号 2

    //EXE 阶段信号
    .exe_rf_waddr(exe_rf_waddr),  // EXE 阶段的寄存器文件写入地址信号
    .exe_rf_wena(exe_rf_wena),    // EXE 阶段的寄存器文件写使能信号
    .exe_hi_wena(exe_hi_wena),    // EXE 阶段的 HI 寄存器写使能信号
    .exe_lo_wena(exe_lo_wena),    // EXE 阶段的 LO 寄存器写使能信号
    .exe_df_hi_out(exe_df_hi_out),// EXE 阶段的数据转发给 ID 阶段 HI 寄存器的数据
    .exe_df_lo_out(exe_df_lo_out),// EXE 阶段的数据转发给 ID 阶段 LO 寄存器的数据
    .exe_df_rf_wdata(exe_df_rf_wdata), // EXE 阶段的数据转发给 ID 阶段寄存器文件的数据
    .exe_op(exe_op),                // EXE 阶段的操作码信号
    .exe_func(exe_func),            // EXE 阶段的功能码信号

    //MEM 阶段信号
    .mem_rf_waddr(mem_rf_waddr),  // MEM 阶段的寄存器文件写入地址信号
    .mem_rf_wena(mem_rf_wena),    // MEM 阶段的寄存器文件写使能信号
    .mem_hi_wena(mem_hi_wena),    // MEM 阶段的 HI 寄存器写使能信号
    .mem_lo_wena(mem_lo_wena),    // MEM 阶段的 LO 寄存器写使能信号
    .mem_df_hi_out(mem_df_hi_out),// MEM 阶段的数据转发给 ID 阶段 HI 寄存器的数据
    .mem_df_lo_out(mem_df_lo_out),// MEM 阶段的数据转发给 ID 阶段 LO 寄存器的数据
    .mem_df_rf_wdata(mem_df_rf_wdata), // MEM 阶段的数据转发给 ID 阶段寄存器文件的数据

    //数据
    .rs_data_out(df_rs_temp),
    .rt_data_out(df_rt_temp),
    .hi_out(df_hi_temp),
```

```
    .lo_out(df_lo_temp),
    .stall(stall),                // 流水线暂停信号
    .if_df(if_df),                // 数据转发使能信号
    .is_rs(is_rs),                // 是否需要转发给 rs 寄存器
    .is_rt(is_rt)                 // 是否需要转发给 rt 寄存器
);

    // 四路选择器选择结果
    mux_4_32 exe_df_mux_hi(
        .C0(exe_div_r),
        .C1(exe_mul_hi),
        .C2(exe_rs_data_out),
        .C3(32'h0),
        .S0(exe_hi_mux_sel),
        .oZ(exe_df_hi_out)
    );


     /// 四路选择器选择结果
    mux_4_32 mem_df_mux_lo(
        .C0(mem_div_q),
        .C1(mem_mul_lo),
        .C2(mem_rs_data_out),
        .C3(32'b0),
        .S0(mem_lo_mux_sel),
        .oZ(mem_df_lo_out)
    );

    // 四路选择器选择结果
    mux_4_32 exe_df_mux_lo(
        .C0(exe_div_q),
        .C1(exe_mul_lo),
        .C2(exe_rs_data_out),
        .C3(32'b0),
        .S0(exe_lo_mux_sel),
        .oZ(exe_df_lo_out)
    );
    // 四路选择器选择结果
    mux_4_32 mem_df_mux_hi(
        .C0(mem_div_r),
        .C1(mem_mul_hi),
        .C2(mem_rs_data_out),
        .C3(32'h0),
        .S0(mem_hi_mux_sel),
```

```
    .oZ(mem_df_hi_out)
);

// 八路选择器选择结果
mux_8_32 mem_df_mux_rf(
    .C0(mem_lo_out),
    .C1(mem_pc4),
    .C2(mem_clz_out),
    .C3(32'b0),
    .C4(mem_dmem_out),
    .C5(mem_alu_out),
    .C6(mem_hi_out),
    .C7(mem_mul_lo),
    .S0(mem_rf_mux_sel),
    .oZ(mem_df_rf_wdata)
);

// 八路选择器选择结果
mux_8_32 exe_df_mux_rf(
    .C0(exe_lo_out),
    .C1(exe_pc4),
    .C2(exe_clz_out),
    .C3(32'b0),
    .C4(32'b0),
    .C5(exe_alu_out),
    .C6(exe_hi_out),
    .C7(exe_mul_lo),
    .S0(exe_rf_mux_sel),
    .oZ(exe_df_rf_wdata)
);



// 5 位二路选择器
mux_2_5 extend5_mux(
    .C0(instruction[10:6]),
    .C1(id_rs_data_out[4:0]),
    .S0(ext5_mux_sel),
    .oZ(ext5_mux_out)
);

// Adder for beq and bne
adder_32 b_pc_adder(
    .a(pc4),
```

```
    .b(ext18_data_out),
    .result(id_b_pc)
);

// 无符号高位补齐
extend_5_32 sa_ext(
    .a(ext5_mux_out),
    .b(id_shamt)
);

// 有符号高位补齐
extend_16_32 imm_ext(
    .data_in(ext16_data_in),
    .sign(ext16_sign),
    .data_out(ext16_data_out)
);

// 高位扩充
extend_sign_18_32 ext18_b_pc(
    .data_in(ext18_data_in),
    .data_out(ext18_data_out)
);

//寄存器模块
regfile cpu_regfile(
    .clk(clk),
    .rst(rst),
    .wena(rf_wena),
    .raddr1(rs),
    .raddr2(rt),
    .rena1(rf_rena1),
    .rena2(rf_rena2),
    .waddr(rf_waddr),
    .wdata(rf_wdata),
    .rdata1(rs_temp),
    .rdata2(rt_temp),
    .reg28(reg28),
    .reg29(reg29)
);

//CP0 模块
cp0 cpu_cp0(
    .clk(clk),
    .rst(rst),
```

```
    .mfc0(mfc0),
    .mtc0(mtc0),
    .pc(pc4 - 4),
    .addr(cp0_addr),
    .wdata(id_rt_data_out),
    .exception(cp0_exception),
    .eret(eret),
    .cause(cp0_cause),
    .rdata(id_cp0_out),
    .status(cp0_status),
    .exc_addr(id_cp0_pc)
);

// 分支预测模块
branch_predict b_p(
    .clk(clk),
    .rst(rst),
    .data_in1(id_rs_data_out),
    .data_in2(id_rt_data_out),
    .op(op),
    .func(func),
    .exception(cp0_exception),
    .is_branch(is_branch)
);

// HI Register
register hi_reg(
    .clk(clk),
    .rst(rst),
    .wena(hi_wena),
    .data_in(hi_wdata),
    .data_out(hi_temp)
);

// LO Register
register lo_reg(
    .clk(clk),
    .rst(rst),
    .wena(lo_wena),
    .data_in(lo_wdata),
    .data_out(lo_temp)
);

//核心控制器，组合逻辑产生控制信号并传出
```

```
    control_unit control_unit(
        .is_branch(is_branch),
        .instruction(instruction),
        .op(op),
        .func(func),
        .status(cp0_status),
        .rf_wena(id_rf_wena),
        .hi_wena(id_hi_wena),
        .lo_wena(id_lo_wena),
        .dmem_wena(id_dmem_wena),
        .rf_rena1(rf_rena1),
        .rf_rena2(rf_rena2),
        .clz_ena(id_clz_ena),
        .mul_ena(id_mul_ena),
        .div_ena(id_div_ena),
        .dmem_ena(id_dmem_ena),
        .dmem_w_cs(id_dmem_w_cs),
        .dmem_r_cs(id_dmem_r_cs),
        .ext16_sign(ext16_sign),
        .cutter_sign(id_cutter_sign),
        .mul_sign(id_mul_sign),
        .div_sign(id_div_sign),
        .aluc(id_aluc),
        .rd(rd),
        .mfc0(mfc0),
        .mtc0(mtc0),
        .eret(eret),
        .exception(cp0_exception),
        .cp0_addr(cp0_addr),
        .cause(cp0_cause),
        .ext5_mux_sel(ext5_mux_sel),
        .cutter_mux_sel(id_cutter_mux_sel),
        .alu_mux1_sel(id_alu_mux1_sel),
        .alu_mux2_sel(id_alu_mux2_sel),
        .hi_mux_sel(id_hi_mux_sel),
        .lo_mux_sel(id_lo_mux_sel),
        .cutter_sel(id_cutter_sel),
        .rf_mux_sel(id_rf_mux_sel),
        .pc_mux_sel(id_pc_mux_sel)
    );

endmodule
```

```
`timescale 1ns / 1ps

`include "define.vh"

//根据 IF-ID 中间的流水寄存器的一些情况决定是否要对 ID 阶段使用的 PC 或指令做出改变
module IF_ID_Reg(
    input clk,
    input rst,
    input [31:0] if_pc4,     // IF 阶段的 PC + 4
    input [31:0] if_instruction,  // IF 阶段的指令
    input stall,             // 是否暂停流水线的信号
    input is_branch,         // 是否是分支指令的信号
    output reg [31:0] id_pc4,     // ID 阶段使用的 PC + 4
    output reg [31:0] id_instruction  // ID 阶段使用的指令
    );

    always @ (posedge clk or posedge rst)
    begin
        if (rst == `RST_ENABLED) begin
            id_pc4 <= `ZERO_32BIT;      // 将 ID 阶段的 PC + 4 重置为 0
            id_instruction <= `ZERO_32BIT; // 将 ID 阶段的指令重置为 0
        end

        else if(is_branch == 1'b1) begin
            id_pc4 <= 32'h0;              // 将 ID 阶段的 PC + 4 重置为 0
            id_instruction <= 32'h0;      // 将 ID 阶段的指令重置为 0
        end

        else if(stall == `RUN) begin
            id_pc4 <= if_pc4;         // 将 IF 阶段的 PC + 4 传递给 ID 阶段
            id_instruction <= if_instruction; // 将 IF 阶段的指令传递给 ID 阶段
        end
    end

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"

module IF(
    input [31:0] pc,
```

```
    input [31:0] cp0_pc,  // 中断跳转地址
    input [31:0] b_pc,   // beq bne bgez 跳转地址
    input [31:0] r_pc,  // jr jalr 跳转地址
    input [31:0] j_pc,  // j jal 跳转地址
    input [2:0] pc_mux_sel,
    output [31:0] npc,  // next pc
    output [31:0] pc4,  // pc + 4
    output [31:0] instruction
    );

    //根据 pc_mux_sel 来选择具体的下一条指令 pc
    mux_8_32 next_pc_mux(
        .C0(j_pc),
        .C1(r_pc),
        .C2(pc4),
        .C3(`EXCEPTION_ADDR),
        .C4(b_pc),
        .C5(cp0_pc),
        .C6(32'b0),
        .C7(32'b0),
        .S0(pc_mux_sel),
        .oZ(npc)
    );

    //实现 NPC 的功能
    adder_32 pc_plus4_adder(
        .a(pc),
        .b(32'h4),
        .result(pc4)
    );

    //从 IP 核中取出具体的指令 instruction
    imem instruction_mem(pc[12:2], instruction);

endmodule
```

```
`timescale 1ns / 1ps

module imem(
    input [10:0] addr,
    output [31:0] instr
    );
```

```
    dist_mem_gen_0 instr_mem(
        .a(addr),
        .spo(instr)
    );
endmodule
```

```
`timescale 1ns / 1ps

`include "define.vh"

module lead0_counter(
    input [31:0] data_in,
    input ena,
    output [31:0] data_out
    );

    reg [31:0] count;

    always @(*)
    begin
        if (ena == `ENABLED) begin
            for (count = 0; count < 32 && data_in[31 - count] != 1; count = count)
                count = count + 1;
        end
    end

    assign data_out = count;

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"

module ME_WB_Reg(
    input clk,
    input rst,
    input wena,
    input [31:0] mem_mul_hi,
    input [31:0] mem_mul_lo,
    input [31:0] mem_div_r,
```

```
    input [31:0] mem_div_q,
    input [31:0] mem_clz_out,
    input [31:0] mem_alu_out,
    input [31:0] mem_dmem_out,
    input [31:0] mem_pc4,
    input [31:0] mem_rs_data_out,
    input [31:0] mem_cp0_out,
    input [31:0] mem_hi_out,
    input [31:0] mem_lo_out,
    input [4:0] mem_rf_waddr,
    input mem_rf_wena,
    input mem_hi_wena,
    input mem_lo_wena,
    input [1:0] mem_hi_mux_sel,
    input [1:0] mem_lo_mux_sel,
    input [2:0] mem_rf_mux_sel,
    output reg [31:0] wb_mul_hi,
    output reg [31:0] wb_mul_lo,
    output reg [31:0] wb_div_r,
    output reg [31:0] wb_div_q,
    output reg [31:0] wb_clz_out,
    output reg [31:0] wb_alu_out,
    output reg [31:0] wb_dmem_out,
    output reg [31:0] wb_pc4,
    output reg [31:0] wb_rs_data_out,
    output reg [31:0] wb_cp0_out,
    output reg [31:0] wb_hi_out,
    output reg [31:0] wb_lo_out,
    output reg [4:0] wb_rf_waddr,
    output reg wb_rf_wena,
    output reg wb_hi_wena,
    output reg wb_lo_wena,
    output reg [1:0] wb_hi_mux_sel,
    output reg [1:0] wb_lo_mux_sel,
    output reg [2:0] wb_rf_mux_sel
    );

    always @ (posedge clk or posedge rst)
    begin
        if(rst == `RST_ENABLED) begin
            wb_mul_hi <= 0;
            wb_mul_lo <= 0;
            wb_div_r <= 0;
            wb_div_q <= 0;
```

```
            wb_clz_out <= 0;
            wb_alu_out <= 0;
            wb_dmem_out <= 0;
            wb_pc4 <= 0;
            wb_rs_data_out <= 0;
            wb_cp0_out <= 0;
            wb_hi_out <= 0;
            wb_lo_out <= 0;
            wb_rf_waddr <= 0;
            wb_rf_wena <= 0;
            wb_hi_wena <= 0;
            wb_lo_wena <= 0;
            wb_hi_mux_sel <= 0;
            wb_lo_mux_sel <= 0;
            wb_rf_mux_sel <= 0;
        end

        else if(wena == `WRITE_ENABLED) begin
            wb_mul_hi <= mem_mul_hi;
            wb_mul_lo <= mem_mul_lo;
            wb_div_r <= mem_div_r;
            wb_div_q <= mem_div_q;
            wb_clz_out <= mem_clz_out;
            wb_alu_out <= mem_alu_out;
            wb_dmem_out <= mem_dmem_out;
            wb_pc4 <= mem_pc4;
            wb_rs_data_out <= mem_rs_data_out;
            wb_cp0_out <= mem_cp0_out;
            wb_hi_out <= mem_hi_out;
            wb_lo_out <= mem_lo_out;
            wb_rf_waddr <= mem_rf_waddr;
            wb_rf_wena <= mem_rf_wena;
            wb_hi_wena <= mem_hi_wena;
            wb_lo_wena <= mem_lo_wena;
            wb_hi_mux_sel <= mem_hi_mux_sel;
            wb_lo_mux_sel <= mem_lo_mux_sel;
            wb_rf_mux_sel <= mem_rf_mux_sel;
        end
    end

endmodule
```

```
`timescale 1ns / 1ps

module ME(
    input clk,
    input [31:0] mul_hi,
    input [31:0] mul_lo,
    input [31:0] div_r,
    input [31:0] div_q,
    input [31:0] clz_out,
    input [31:0] alu_out,
    input [31:0] pc4,
    input [31:0] rs_data_out,
    input [31:0] rt_data_out,
    input [31:0] cp0_out,
    input [31:0] hi_out,
    input [31:0] lo_out,
    input [4:0] rf_waddr,
    input dmem_ena,
    input rf_wena,
    input hi_wena,
    input lo_wena,
    input dmem_wena,
    input cutter_sign,
    input [1:0] dmem_w_cs,
    input [1:0] dmem_r_cs,
    input cutter_mux_sel,
    input [2:0] cutter_sel,
    input [1:0] hi_mux_sel,
    input [1:0] lo_mux_sel,
    input [2:0] rf_mux_sel,
    output [31:0] mem_mul_hi,  // Higher bits of the multiplication result
    output [31:0] mem_mul_lo,  // Lower bits of the multiplication result
    output [31:0] mem_div_r,  // Remainder of the division result
    output [31:0] mem_div_q,  // Quotient of the division result
    output [31:0] mem_clz_out,
    output [31:0] mem_alu_out,  // Result of ALU
    output [31:0] mem_dmem_out,
    output [31:0] mem_pc4,
    output [31:0] mem_rs_data_out,
    output [31:0] mem_cp0_out,
    output [31:0] mem_hi_out,
    output [31:0] mem_lo_out,
    output [4:0] mem_rf_waddr,
    output mem_rf_wena,
```

```
output mem_hi_wena,
output mem_lo_wena,
output [1:0] mem_hi_mux_sel,
output [1:0] mem_lo_mux_sel,
output [2:0] mem_rf_mux_sel
);

wire [31:0] dmem_out;
wire [31:0] cutter_mux_out;

assign mem_pc4 = pc4;
assign mem_rs_data_out = rs_data_out;
assign mem_hi_out = hi_out;
assign mem_lo_out = lo_out;
assign mem_cp0_out = cp0_out;
assign mem_mul_hi = mul_hi;
assign mem_mul_lo = mul_lo;
assign mem_div_q = div_q;
assign mem_div_r = div_r;
assign mem_clz_out = clz_out;
assign mem_alu_out = alu_out;
assign mem_rf_waddr = rf_waddr;
assign mem_rf_wena = rf_wena;
assign mem_hi_wena = hi_wena;
assign mem_lo_wena = lo_wena;
assign mem_hi_mux_sel = hi_mux_sel;
assign mem_lo_mux_sel = lo_mux_sel;
assign mem_rf_mux_sel = rf_mux_sel;

mux_2_32 cutter_mux(
    .C0(rt_data_out),
    .C1(dmem_out),
    .S0(cutter_mux_sel),
    .oZ(cutter_mux_out)
);

cutter cpu_cutter(
    .data_in(cutter_mux_out),
    .sel(cutter_sel),
    .sign(cutter_sign),
    .data_out(mem_dmem_out)
);

dmem cpu_dmem(
```

```
        .clk(clk),
        .ena(dmem_ena),
        .wena(dmem_wena),
        .w_cs(dmem_w_cs),
        .r_cs(dmem_r_cs),
        .data_in(mem_dmem_out),
        .addr(alu_out),
        .data_out(dmem_out)
    );

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"
module multiplier(
    input reset,
    input ena,
    input sign,
    input [31:0] a,
    input [31:0] b,
    output [31:0] mul_hi,
    output [31:0] mul_lo
    );

    parameter BIT_NUM = 32;

    reg [31:0] temp_a;
    reg [31:0] temp_b;
    reg [63:0] stored;
    reg [63:0] temp;
    reg is_minus;
    integer i;

    always @(*)
    begin
        if(reset == `RST_ENABLED) begin
            stored <= 0;
            is_minus <= 0;
            temp_a <= 0;
            temp_b <= 0;
        end

        else if(ena == `ENABLED) begin
```

```
        // If multiplicator or multiplicand is 0, the result will be 0
        if (a == 0 || b == 0) begin
            stored <= 0;
        end

        // Unsigned
        else if(sign == `UNSIGNED) begin
            stored = 0;

            for (i = 0; i < 32; i = i + 1) begin
                temp = b[i] ? ({32'b0, a} << i) : 64'b0;
                stored = stored + temp;
            end
        end

        // Signed
        else begin
            stored = 0;
            is_minus = a[31] ^ b[31];   //judge
            temp_a = a;
            temp_b = b;

            if (a[31] == 1) begin
                temp_a = a ^ 32'hffffffff;
                temp_a = temp_a + 1;
            end

            if (b[31] == 1) begin
                temp_b = b ^ 32'hffffffff;
                temp_b = temp_b + 1;
            end

            for (i = 0; i < 32; i = i + 1) begin
                temp = temp_b[i] ? ({32'b0, temp_a} << i) : 64'b0;
                stored = stored + temp;
            end

            if (is_minus == 1) begin
                stored = stored ^ 64'hffffffffffffffff;
                stored = stored + 1;
            end
        end
    end
end
```

```
    assign mul_hi = (ena == `ENABLED) ? stored[63:32] : 32'b0;
    assign mul_lo = (ena == `ENABLED) ? stored[31:0] : 32'b0;

endmodule
```

```
`timescale 1ns / 1ns

module mux_2_5(
    input [4:0] C0,
    input [4:0] C1,
    input S0,
    output reg [4:0] oZ
    );

    always @ (C0 or C1 or S0) begin
        case(S0)
            1'b0: oZ <= C0;
            1'b1: oZ <= C1;
        endcase
   end

endmodule
```

```
`timescale 1ns / 1ps

module mux_2_32(
    input [31:0] C0,
    input [31:0] C1,
    input S0,
    output reg [31:0] oZ
    );

    always @ (C0 or C1 or S0) begin
        case(S0)
            1'b0: oZ <= C0;
            1'b1: oZ <= C1;
        endcase
   end

endmodule
```

```
`timescale 1ns / 1ps

module mux_4_32(
    input [31:0] C0,
    input [31:0] C1,
    input [31:0] C2,
    input [31:0] C3,
    input [1:0] S0,
    output [31:0] oZ
    );

    reg [31:0] tmp_res;

    always @(*) begin
        case(S0)
            2'b00: tmp_res <= C0;
            2'b01: tmp_res <= C1;
            2'b10: tmp_res <= C2;
            2'b11: tmp_res <= C3;
            default: tmp_res <= 31'bz;
        endcase
   end

   assign oZ = tmp_res;

endmodule
```

```
`timescale 1ns / 1ps

module mux_8_32(
    input [31:0] C0,
    input [31:0] C1,
    input [31:0] C2,
    input [31:0] C3,
    input [31:0] C4,
    input [31:0] C5,
    input [31:0] C6,
    input [31:0] C7,
    input [2:0] S0,
    output [31:0] oZ
    );
```

```
    reg [31:0] tmp_res;

    always @(*) begin
        case(S0)
            3'b000: tmp_res <= C0;
            3'b001: tmp_res <= C1;
            3'b010: tmp_res <= C2;
            3'b011: tmp_res <= C3;
            3'b100: tmp_res <= C4;
            3'b101: tmp_res <= C5;
            3'b110: tmp_res <= C6;
            3'b111: tmp_res <= C7;
            default: tmp_res <= 32'bz;
        endcase
   end

   assign oZ = tmp_res;

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"

module pc_reg(
    input clk,
    input rst,
    input wena,
    input stall,  // 暂停
    input [31:0] data_in,   // 下一个 pc
    output [31:0] data_out   // 当前 pc
    );

    reg [31:0] pc;

    always @ (posedge clk or posedge rst)
    begin
        if(rst == `RST_ENABLED) begin
            pc <= 32'h00400000;//MIPS 标准下的初始 pc 值
        end

        else if(stall == `RUN) begin
            if(wena == `WRITE_ENABLED) begin
```

```
                pc <= data_in;
            end
        end
    end

    assign data_out = pc;

endmodule
```

```
`timescale 1ns / 1ps
`include "define.vh"
module regfile(
    input clk,
    input rst,
    input wena,           // 写使能信号
    input [4:0] raddr1,   // 读地址 1
    input [4:0] raddr2,   // 读地址 2
    input rena1,          // 读使能 1
    input rena2,          // 读使能 2
    input [4:0] waddr,    // 写地址
    input [31:0] wdata,   // 写数据
    output reg [31:0] rdata1, // 读数据 1
    output reg [31:0] rdata2, // 读数据 2
    output [31:0] reg28,     // 寄存器 28，存储计算结果
    output [31:0] reg29      // 寄存器 29，通常为零
    );

    reg [31:0] Regs[0:31]; // 寄存器文件，共 32 个寄存器
    integer i;

    // 写
    always @ (posedge clk or posedge rst)
    begin
        if(rst == `RST_ENABLED) begin
            for(i = 0; i < 32; i = i + 1)
                Regs[i] = 0; // 复位时将所有寄存器清零
        end
        else begin
            if((wena == `WRITE_ENABLED) && (waddr != 0))
                Regs[waddr] = wdata; // 如果写使能信号有效且写地址不为 0，则写入数据到指定寄存器
        end
    end
```

```
    // 读
    always @ (*)
    begin
        if(rst == `RST_ENABLED)
              rdata1 <= `ZERO_32BIT;

        else if(raddr1 == 5'b0)
            rdata1 <= `ZERO_32BIT;

        else if((raddr1 == waddr) && (wena == `WRITE_ENABLED) && (rena1 == `READ_ENABLED))
            rdata1 <= wdata; // 如果读地址 1 等于写地址且写使能信号有效且读使能信号有效，则读取写入的数据
        else if(rena1 == `READ_ENABLED)
            rdata1 <= Regs[raddr1]; // 从指定寄存器读取数据
        else
            rdata1 <= `ZERO_32BIT;
    end

    // 读
    always @ (*)
    begin
        if(rst == `RST_ENABLED)
              rdata2 <= `ZERO_32BIT;

        else if(raddr2 == 5'b0)
            rdata2 <= `ZERO_32BIT;

        else if((raddr2 == waddr) && (wena == `WRITE_ENABLED) && (rena2 == `READ_ENABLED))
            rdata2 <= wdata; // 如果读地址 2 等于写地址且写使能信号有效且读使能信号有效，则读取写入的数据
        else if(rena2 == `READ_ENABLED)
            rdata2 <= Regs[raddr2]; // 从指定寄存器读取数据
        else
            rdata2 <= `ZERO_32BIT;
    end

    assign reg28 = Regs[28]; // 计算结果存储在寄存器 28
    assign reg29 = Regs[29]; // 寄存器 29 通常为零

endmodule
```

```
`timescale 1ns / 1ns

module register(
    input clk,
    input rst,
    input wena,
    input [31:0] data_in,
    output [31:0] data_out
    );

    reg [31:0] data;

    always @ (posedge clk or posedge rst)
    begin
        if(rst == 1) begin
            data <= 0;
        end

        else if(wena == 1) begin
            data <= data_in;
        end
    end

    assign data_out = data;

endmodule
```

```
`timescale 1ns / 1ns

module seg7x16(
    input clk,
    input reset,
    input cs,
    input [31:0] i_data,
    output [7:0] o_seg,
    output [7:0] o_sel
    );

    reg [14:0] cnt;
     always @ (posedge clk, posedge reset)
      if (reset)
        cnt <= 0;
      else
        cnt <= cnt + 1'b1;
```

```
wire seg7_clk = cnt[14];

 reg [2:0] seg7_addr;

 always @ (posedge seg7_clk, posedge reset)
   if(reset)
      seg7_addr <= 0;
    else
      seg7_addr <= seg7_addr + 1'b1;

 reg [7:0] o_sel_r;

 always @ (*)
   case(seg7_addr)
      7 : o_sel_r = 8'b01111111;
      6 : o_sel_r = 8'b10111111;
      5 : o_sel_r = 8'b11011111;
      4 : o_sel_r = 8'b11101111;
      3 : o_sel_r = 8'b11110111;
      2 : o_sel_r = 8'b11111011;
      1 : o_sel_r = 8'b11111101;
      0 : o_sel_r = 8'b11111110;
    endcase

 reg [31:0] i_data_store;
 always @ (posedge clk, posedge reset)
   if(reset)
      i_data_store <= 0;
    else if(cs)
      i_data_store <= i_data;

 reg [7:0] seg_data_r;
 always @ (*)
   case(seg7_addr)
      0 : seg_data_r = i_data_store[3:0];
      1 : seg_data_r = i_data_store[7:4];
      2 : seg_data_r = i_data_store[11:8];
      3 : seg_data_r = i_data_store[15:12];
      4 : seg_data_r = i_data_store[19:16];
      5 : seg_data_r = i_data_store[23:20];
      6 : seg_data_r = i_data_store[27:24];
      7 : seg_data_r = i_data_store[31:28];
    endcase
```

```
    reg [7:0] o_seg_r;
    always @ (posedge clk, posedge reset)
      if(reset)
         o_seg_r <= 8'hff;
       else
         case(seg_data_r)
           4'h0 : o_seg_r <= 8'hC0;
         4'h1 : o_seg_r <= 8'hF9;
         4'h2 : o_seg_r <= 8'hA4;
         4'h3 : o_seg_r <= 8'hB0;
         4'h4 : o_seg_r <= 8'h99;
         4'h5 : o_seg_r <= 8'h92;
         4'h6 : o_seg_r <= 8'h82;
         4'h7 : o_seg_r <= 8'hF8;
         4'h8 : o_seg_r <= 8'h80;
         4'h9 : o_seg_r <= 8'h90;
         4'hA : o_seg_r <= 8'h88;
         4'hB : o_seg_r <= 8'h83;
         4'hC : o_seg_r <= 8'hC6;
         4'hD : o_seg_r <= 8'hA1;
         4'hE : o_seg_r <= 8'h86;
         4'hF : o_seg_r <= 8'h8E;
         endcase

    assign o_sel = o_sel_r;
    assign o_seg = o_seg_r;

endmodule
```

```
`timescale 1ns / 1ps

//整个动态流水线的顶层程序
module pipeline_top(
    input clk,
    input rst,
    input switch_res,// 选择输出结果信号
    input stop,//外部中断信号
    //output [31:0]res,//输出的结果
    output [7:0] o_seg,//七段数码管参数
    output [7:0] o_sel //七段数码管参
数

    );
```

```
parameter K = 99_999;

reg [31:0] cnt;
reg clk_new;

wire [31:0] output_res;//计算结果
wire [31:0] clear_res;// 全零值，清空结果
wire [31:0] seg7_in;//七段数码管输入值
wire [31:0] instruction;
wire [31:0] pc;


//时钟分频程序，方便下板演示
always @ (posedge clk or posedge rst)
begin
    if(rst) begin
        clk_new <= 0;
        cnt<=0;
    end
    else if(cnt == K) begin
        cnt <= 0;
        clk_new <= ~clk_new;
    end
    else
        cnt<=cnt+1;
end

//cpu 顶层程序，负责五个阶段 IF-ID-EX-ME-WE
cpu_top cpu(
    .clk(clk),   //分频之后的时钟信号
    .rst(rst),
    .instruction(instruction),
    .pc(pc),
    .reg28(output_res),//输出结果存储在 28 号寄存器
    .reg29(clear_res),//29 号寄存器没有使用过，所以值为 0
    .stop(stop) //外部中断信号
);

//选择器，选择一个结果输出
mux_2_32 res_mux(
    .C0(output_res),
    .C1(clear_res),
    .S0(switch_res),
```

```
        .oZ(seg7_in)
    );
    assign res = seg7_in;
    //七段数码管输出最终结果
    seg7x16 seg7(clk, rst, 1, seg7_in, o_seg, o_sel);

endmodule
```

```
`timescale 1ns / 1ps

module WB(
    input [31:0] mul_hi,
    input [31:0] mul_lo,
    input [31:0] div_r,
    input [31:0] div_q,
    input [31:0] clz_out,
    input [31:0] alu_out,
    input [31:0] dmem_out,
    input [31:0] pc4,
    input [31:0] rs_data_out,
    input [31:0] cp0_out,
    input [31:0] hi_out,
    input [31:0] lo_out,
    input [4:0] rf_waddr,
    input rf_wena,
    input hi_wena,
    input lo_wena,
    input [1:0] hi_mux_sel,
    input [1:0] lo_mux_sel,
    input [2:0] rf_mux_sel,
    output [31:0] hi_wdata,
    output [31:0] lo_wdata,
    output [31:0] rf_wdata,
    output [4:0] wb_rf_waddr,
    output wb_rf_wena,
    output wb_hi_wena,
    output wb_lo_wena
    );

    // MUX for LO register
    mux_4_32 mux_lo(
        .C0(div_q),
        .C1(mul_lo),
```

```
        .C2(rs_data_out),
        .C3(32'b0),
        .S0(lo_mux_sel),
        .oZ(lo_wdata)
    );

    // MUX for regfile
    mux_8_32 mux_rf(
        .C0(lo_out),
        .C1(pc4),
        .C2(clz_out),
        .C3(cp0_out),
        .C4(dmem_out),
        .C5(alu_out),
        .C6(hi_out),
        .C7(mul_lo),
        .S0(rf_mux_sel),
        .oZ(rf_wdata)
    );
    // MUX for HI register
    mux_4_32 mux_hi(
        .C0(div_r),
        .C1(mul_hi),
        .C2(rs_data_out),
        .C3(32'h0),
        .S0(hi_mux_sel),
        .oZ(hi_wdata)
    );
    assign wb_rf_wena = rf_wena;
    assign wb_hi_wena = hi_wena;
    assign wb_rf_waddr = rf_waddr;
    assign wb_lo_wena = lo_wena;

endmodule
```