实验报告成绩:	成绩评定日期:

2022 ~ 2023 学年秋季学期

《计算机系统》必修课

课程实验报告

班级：人工智能 2202

组长：段雨洁 20226426

组员：石晓瑞 20226503

报告日期：2025.1.4

目录

一 实验概括

1.1 工作量

姓名	完成任务点	工作量占比
段雨洁	暂停的部分实现、乘除法器的连接、一部分指令的添加，参与乘法器设计	50%
石晓瑞	实现一部分数据通路，添加部分指令定义，添加暂停部分，参与乘法器设计	50%

1.2 总体设计

该代码展示了一个基于 RISC-V 指令集的处理器核心 — SampleCPU 的设计与实现。该处理器核心采用典型的五级流水线架构，包括取指(IF)、译码(ID)、执行(EX)、访存(MEM)、写回(WB)五个阶段，旨在优化指令执行过程并提升系统的整体性能。

我们的流水线 CPU 设计由七个主要模块构成，包括指令取指(IF)模块、指令译码(ID)模块、执行(EX)模块、访存(MEM)模块、写回(WB)模块、控制(CTRL)模块以及寄存器文件(regfile)模块。整个设计包含 32 个 32 位的整数寄存器，

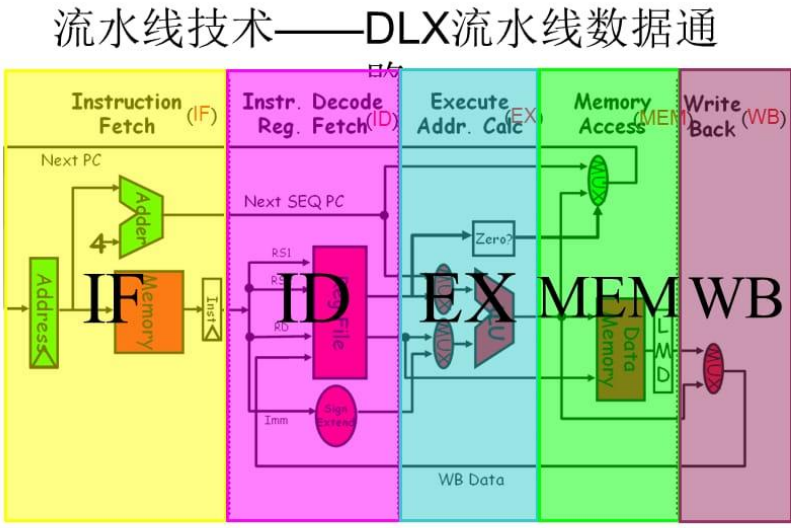


图 1 DLX 的流水线

并采用大端模式存储数据，具有 32 位的数据总线和地址总线宽度。我们的 CPU 能够满足实验课程中规定的 64 个测试点的要求。

实验环境配置如下：

1. 仿真与综合工具：采用龙芯杯指定的 XILINX 公司的 Vivado 2019.2 版本作为仿真运行工具和 FPGA 综合工具。
2. 代码编辑器：使用 VS Code 作为 Vivado 2019.2 的默认代码编辑器，便于代码编写、调试和管理。
3. 版本控制与协同开发：通过 git 来完成小组内部的协同开发和代码同步，确保团队成员能够高效协作，及时更新和整合代码.

1.2.1 取指令周期（IF）

取指令周期是处理器执行流程的第一步，它负责从指令存储器中获取即将执行的指令。在这个阶段，处理器会增加程序计数器（PC）的值，通常是加上一个固定步长，如 4 个字节，或者在遇到分支指令时，根据目标地址进行更新。然后，新的 PC 值会被送到指令存储器的地址线，以指定要读取的指令地址。接着，处理器激活指令存储器并从中读取指令。最后，根据读取的指令和控制信号，如分支或跳转指令，更新 PC 的值，以确保下一周期能够获取正确的指令。

1.2.2 指令译码/读寄存器周期（ID）

在指令译码/读寄存器周期，处理器对在取指令周期中获取的指令进行解码，以确定指令的类型、操作码以及源和目标寄存器。根据解码结果，处理器从寄存器文件中读取所需的源寄存器值，并生成控制信号，这些信号将指导后续周期如何处理当前指令。此外，处理器还会检查是否存在数据冒险，并在必要时从后续阶段转发数据。解码信息和寄存器值随后被发送到执行阶段。

1.2.3 执行/有效地址计算周期（EX）

执行/有效地址计算周期是处理器执行实际运算的地方。在这里，算术或逻辑操作被执行，如加法、减法、与、或等。对于需要访问内存的指令，这个阶段还会计算内存地址。同时，处理器会计算分支条件并确定是否满足。准备就绪后，结果或计算后的地址会被发送到下一阶段。

1.2.4 存储器访问 / 分支完成周期（MEM）

在存储器访问 / 分支完成周期，处理器执行与数据存储器的交互操作。这包括根据执行阶段的地址和控制信号进行读写操作。对于分支指令，这个阶段会完成跳转，更新 PC 至新的分支地址。对于加载指令，数据从存储器读取到寄存器；而对于存储指令，数据则被写入到指定的内存地址。

1.2.5 写回周期（WB）

写回周期是处理器执行流程的最后阶段，它负责将执行或内存访问阶段的结果写回到目标寄存器。这个阶段还会根据指令执行的结果更新程序状态，如条件标志等。最后，处理器会清理用于当前指令的任何临时或过程状态，为下一指令的处理做好准备。。

1.2.6 在本方案的实现中

分支指令需要 4 个时钟周期（移到 ID 段，只需 2 个周期）， store 指令需要 4 个时钟周期，其它指令需要 5 个时钟周期。

1.3 MIPS 指令格式

1.3.1MIPS 指令的基本组成

MIPS 指令包括操作码（Opcode）、寄存器标识符（Register Identifiers）、立即数和目标地址。一般来说，MIPS 指令的基本格式为：

```
+---------+---------+---------+---------+---------+---------+
| Opcode  | rs      | rt      | rd      | shamt   | funct   |
+---------+---------+---------+---------+---------+---------+
```

- Opcode：操作码，用于识别指令类型
- rs、rt、rd：源操作数寄存器、目标操作数寄存器和目的寄存器
- shamt：位移量，表示移位操作时的位移量
- funct：功能码，用于确定具体的操作

1.3.2 MIPS 指令的类型分类

根据指令的不同功能和操作类型，MIPS 指令可以分为三种类型：R 型指令、I 型指令和 J 型指令。

- R 型指令：主要用于寄存器-寄存器的操作，如加法、逻辑运算等
- I 型指令：主要用于寄存器-立即数的操作，如加载数据、存储数据等
- J 型指令：用于跳转指令，如无条件跳转、调用子程序等

1.4 不同流水段之间的连线图

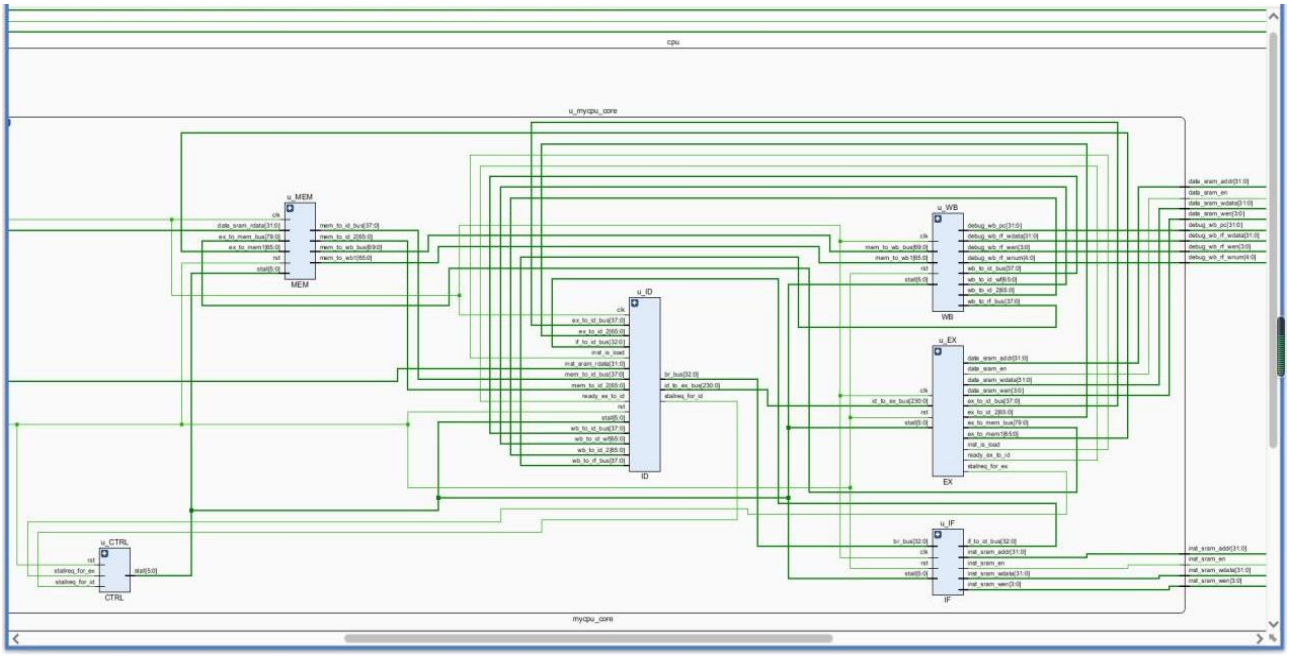


图 2 流水线 CPU 总体结构图

1.5 指令完成度

我们已经实现了所有必要的指令定义，以满足当前测试点的要求。这些指令涵盖了运算逻辑、位移、分支、加载和存储等多种操作，确保了处理器的基本功能和灵活性。以下是具体实现的指令列表：

1. **算术逻辑指令：**

- inst_add, inst_addu, inst_sub, inst_subu: 有符号和无符号的加法和减法。
- inst_addi, inst_addiu: 立即数加法，有符号和无符号。
- inst_and, inst_andi: 逻辑与操作，包括立即数。
- inst_or, inst_ori: 逻辑或操作，包括立即数。
- inst_xor, inst_xori: 逻辑异或操作，包括立即数。
- inst_nor: 逻辑或非操作。
- inst_slt, inst_sltu: 有符号和无符号的小于比较。
- inst_slti, inst_sltiu: 立即数小于比较，有符号和无符号。

2. **移位指令：**

- inst_sll, inst_sllv: 逻辑左移，包括变量移位。
- inst_srl, inst_srlv: 逻辑右移，包括变量移位。
- inst_sra, inst_srav: 算术右移，包括变量移位。

3. **分支指令：**

- inst_beq, inst_bne: 如果两个寄存器相等或不相等，则分支。
- inst_bgez, inst_bgtz, inst_blez, inst_bltz: 基于符号位的分支指令。
- inst_bltzal, inst_bgezal: 链接的分支指令，用于子程序调用。
- inst_j, inst_jal, inst_jalr: 跳转指令，包括链接跳转和寄存器间接跳转。

4. **加载和存储指令：**

- inst_lw, inst_lh, inst_lhu, inst_lb, inst_lbu: 从内存加载数据到寄存器。
- inst_sw, inst_sh, inst_sb: 从寄存器存储数据到内存。

5. **寄存器传输指令：**

- inst_mfhi, inst_mflo: 将 HI 和 LO 寄存器的值移动到通用寄存器。
- inst_mthi, inst_mtlo: 将通用寄存器的值移动到 HI 和 LO 寄存器。

6. **乘法和除法指令：**

- inst_mult, inst_multu: 有符号和无符号乘法。
- inst_div, inst_divu: 有符号和无符号除法。

7. **其他指令：**

- inst_jr: 寄存器间接跳转。
- inst_lui: 加载上位立即数。
- inst_lsa: 带移位的加法。

二 单个流水段说明

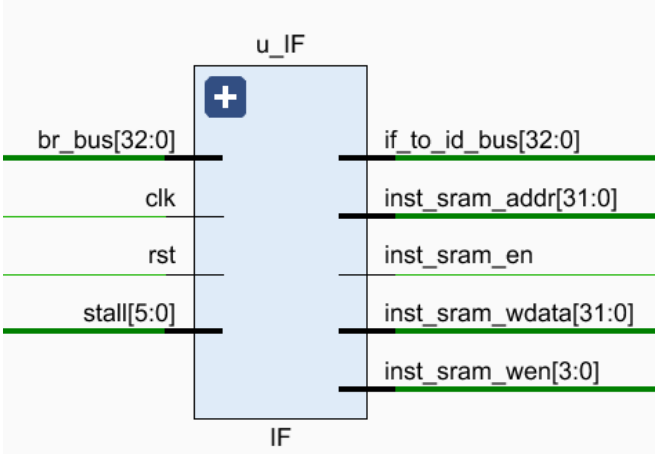
2.1 IF 段

2.1.1 整体功能说明

取指令周期（IF 段）定义一个指令获取（IF）模块，它是 CPU 中负责从内存读取指令的部分。模块通过程序计数器（PC）跟踪当前指令的位置，并在每

个时钟周期检查是否需要根据外部信号暂停或跳转。如果 rst 信号被激活，PC 会重置到一个预设的地址，通常是程序的起始点。在正常操作时，如果没有暂停信号（由 stall 线控制），PC 会更新到下一个连续的地址或由分支指令指定的新地址（由 br_bus 提供）。这保证了处理器能连续地获取指令，为执行单元提供持续的指令流。此模块还生成对应的信号来控制指令存储器的读操作，确保每获取到的指令都能被传递到指令解码（ID）阶段。

2.1.2 端口介绍



输入：

clk、rst、stall、br_bus

输出：

if_to_id_bus[32:0],inst_sram_en,inst_sram_wen[3:0],

inst_sram_addr[31:0],inst_sram_wdata[31:0].

2.1.3 信号介绍

在这个指令获取（IF）模块中，主要信号包括程序计数器（PC），条件使能（CE）寄存器，以及来自分支总线（br_bus）的分支执行（br_e）和分支地址（br_addr）。PC 寄存器用于跟踪当前的指令地址，而 CE 寄存器控制指令

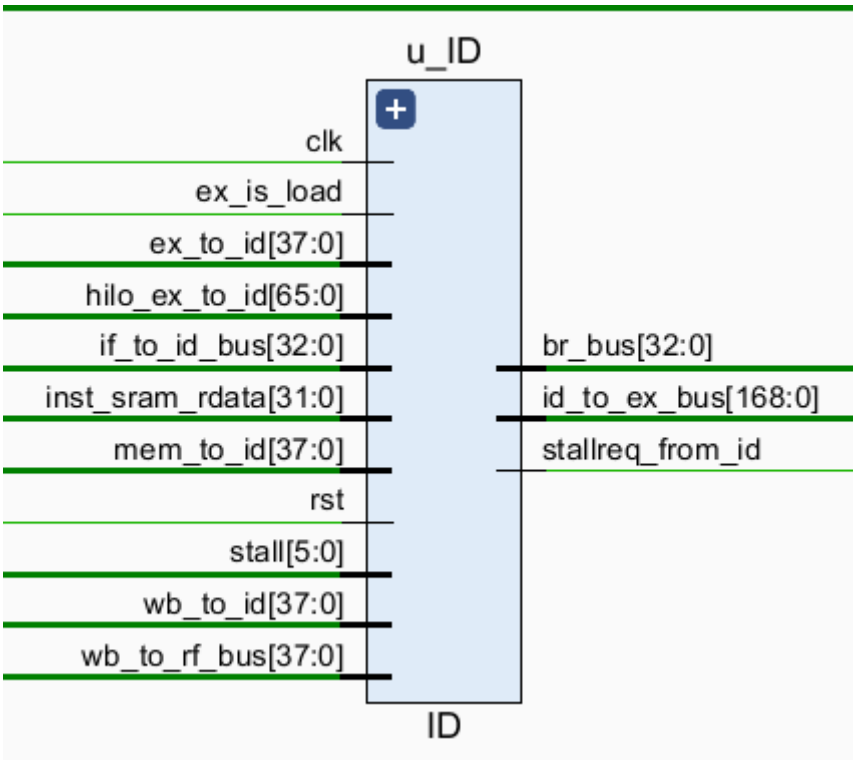
存储器的访问。在每个时钟周期，如果复位（rst）激活，则 PC 被重置为预设的起始地址 0xbfbf_fffc，并且 CE 被置为 0，停止内存访问。若无暂停（由 stall 的第一个比特控制），PC 将更新为下一个地址，该地址根据 br_e 信号决定：如果 br_e 为真，表示需要执行分支，PC 跳转到 br_addr 指定的新地址；否则，PC 递增 4，指向下一条顺序指令的地址。这些操作通过 always @ (posedge clk)块实现，确保了在不同控制信号的影响下，处理器可以有效地获取每条指令，维持指令流的连续性。同时，CE 信号确保在适当的时刻使能或禁用对指令存储器的访问，配合 PC 的更新，为指令解码阶段提供正确的指令数据。

2.2 ID 段

2.2.1 整体功能说明

指令解码（ID）模块是 MIPS 处理器的核心组件之一，负责解析从指令获取（IF）阶段传递来的指令，并根据指令的操作码和功能码提取出关键操作信息。该模块将指令解析为具体的执行信号，这些信号指导执行单元（EX）进行相应的操作，如算术和逻辑运算、数据存储器访问等。ID 模块同时处理来自执行、内存和写回阶段的反馈，动态管理流水线的暂停和数据前递，以优雅地处理数据冒险和控制冒险。此外，它还负责处理程序的分支跳转，生成分支信号，确保程序逻辑的正确执行。通过这种方式，ID 模块保证了即使在复杂的指令流和多变的执行环境中，处理器也能高效稳定地运行。

2.2.2 端口介绍



输入端口：clk , rst , stall, ex_is_load, if_to_id_bus[32:0] inst_sram_rdata[32:0], wb_to_rf_bus[37:0], ex_to_id[38:0], mem_to_id [38:0], wb_to_id[38:0], hilo_ex_to_id [66:0];

输出端口：stallreq , id_to_ex_bus [168:0], br_bus [32:0], stallreq_from_id

2.2.3 信号介绍

1. 指令信号

- ✓ **功能：**指令信号是解码的核心，模块从 inst_sram_rdata 中获取当前指令并解析它的各个字段（操作码、功能码、寄存器编号等），以生成相应的控制信号。
- ✓ **对应代码：**

```
assign inst = (q) ? inst : inst_sram_rdata;
```

q 用来判断流水线是否暂停。如果暂停，则保持当前指令不变，否则从 inst_sram_rdata 获取新指令。

2. 暂停控制信号

- ✓ **功能：**在 EX 阶段存在加载操作且与当前指令存在数据依赖时，ID 模块会生成暂停请求 stallreq_from_id，防止流水线错误地执行。
- ✓ **对应代码：**

```
assign stallreq_from_id = (ex_is_load == 1'b1) &&
                          ((rs == ex_to_id[36:32]) || (rt == ex_to_id[36:32]));
```

ex_is_load：是判断是否为 load 指令。

检查当前指令的 rs 和 rt 是否与 EX 阶段的目标寄存器相同。如果相同，说明存在数据冒险，需要暂停流水线。

3. 分支跳转控制信号

- ✓ **功能**：分支控制信号 br_e 决定是否执行分支跳转，而 br_addr 表示跳转目标地址。它们通过当前指令的类型和条件计算得出。
- ✓ **对应代码**：

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


assign br_addr = inst_beq ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0], 2'b0}) :
                 (inst_jr | inst_jalr) ? (rdata11) :
                 inst_jal ? ({pc_plus_4[31:28], inst[25:0], 2'b0}) :
                 inst_j ? ({pc_plus_4[31:28], inst[25:0], 2'b0}) :
                 (inst_bgezal | inst_bltzal | inst_blez | inst_bltz | inst_bgez | inst_bgtz) ?
                 (pc_plus_4 + {{14{inst[15]}}, inst[15:0], 2'b00}) :
                 inst_bne ? (pc_plus_4 + {{14{inst[15]}}, {inst[15:0], 2'b00}}) : 32'b0;
```

br_e：通过指令类型（如 inst_j, inst_beq, inst_bne 等）和比较结果（rs_ge_z、rs_eq_rt 等）判断是否跳转。

br_addr：根据具体指令类型计算跳转目标地址，例如：

- ✓ 条件跳转（inst_beq）：pc_plus_4 加偏移量。
- ✓ 无条件跳转（inst_j、inst_jal）：目标地址直接来自指令。

4. ALU 操作控制信号

- ✓ **功能**：ALU 操作类型由 alu_op 决定，指示执行单元需要进行的具体计算（如加法、减法、逻辑运算等）。
- ✓ **对应代码**：

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

- ✓ **具体指令操作标志**（如 op_add，op_sub）：

```
assign op_add = inst_lsa | inst_sh | inst_sb | inst_lhu | inst_lh | inst_lbu | inst_lb |
                inst_addi | inst_add | inst_addiu | inst_lw | inst_addu | inst_jal |
                inst_sw | inst_bltzal | inst_bgezal | inst_jalr;
assign op_sub = inst_sub | inst_subu;
```

5. 数据存储器控制信号

- ✓ **功能**：这些信号控制数据存储器的读写，包括是否启用存储器（data_ram_en）、是否写入数据（data_ram_wen）、以及具体的读写字节使能信号（data_ram_readen）。
- ✓ **对应代码**：

```
assign data_ram_en = inst_sh | inst_sb | inst_lhu | inst_lh | inst_lbu | inst_lw | inst_sw | inst_lb;

assign data_ram_wen = inst_sw ? 4'b1111 : 4'b0000;

assign data_ram_readen = inst_lw  ? 4'b1111 :
                         inst_lb  ? 4'b0001 :
                         inst_lbu ? 4'b0010 :
                         inst_lh  ? 4'b0011 :
                         inst_lhu ? 4'b0100 :
                         inst_sb  ? 4'b0101 :
                         inst_sh  ? 4'b0111 :
                         4'b0000;
```

6. 寄存器文件控制信号

- ✓ **功能**：这些信号控制寄存器文件的读写，包括是否写入寄存器（rf_we）、目标写地址（rf_waddr）、以及从哪个数据来源写入（sel_rf_res）。
- ✓ **对应代码**：

```
assign rf_we = inst_lsa | inst_lhu | inst_lh | inst_lbu | inst_lb | inst_mfhi | inst_mflo |
               inst_jalr | inst_bgezal | inst_bltzal | inst_srl | inst_srlv | inst_srav |
               inst_sra | inst_sllv | inst_andi | inst_and | inst_sub | inst_addi | inst_add |
               inst_sltiu | inst_slti | inst_slt | inst_sltu | inst_nor | inst_xori |
               inst_xor | inst_sll | inst_ori | inst_lui | inst_addiu | inst_subu |
               inst_jal | inst_lw | inst_addu | inst_or;

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

7. **前递数据信号**

- ✓ **功能**：在流水线操作中，直接从 EX、MEM、WB 阶段获取最新结果（即前递机制），避免数据冒险。
- ✓ **对应代码**：

```
assign rdata11 = (inst_mfhi | inst_mflo) ? mf_data : rdata1;
assign rdata22 = (inst_mfhi | inst_mflo) ? mf_data : rdata2;
```

- ✓ **HI/LO 寄存器的前递数据**：

```
assign mf_data = (inst_mfhi & hi_wen) ? hi_data :
                 (inst_mfhi) ? hilo_data :
                 (inst_mflo & lo_wen) ? lo_data :
                 (inst_mflo) ? hilo_data :
                 (32'b0);
```

8. **分支和跳转比较信号**

- ✓ **功能**：这些信号根据当前指令的寄存器值和条件生成分支判断结果，用于控制程序跳转逻辑。
- ✓ **对应代码**：

```
assign rs_eq_rt = (rdata11 == rdata22); // rs 和 rt 相等
assign rs_ge_z = (rdata11[31] == 1'b0); // rs >= 0
assign rs_gt_z = (rdata11[31] == 1'b0 & rdata11 != 32'b0); // rs > 0
assign rs_le_z = (rdata11[31] == 1'b1 | rdata11 == 32'b0); // rs <= 0
assign rs_lt_z = (rdata11[31] == 1'b1); // rs < 0
```

2.2.4 regfile 功能模块介绍

实现了 MIPS 处理器的通用寄存器文件，同时扩展支持 HI 和 LO 寄存器的读写操作。该模块为 32 个通用寄存器（$0 至$31）提供读写功能，支持三条数据链路（ex_to_id、mem_to_id 和 wb_to_id）的数据冒险处理，用于实现流水线的前递功能。此外，模块中包含 HI 和 LO 寄存器，支持常见的乘除法和特殊寄存器操作。

1. **寄存器文件结构**：

- ✓ 包含 32 个 32 位宽的通用寄存器，用 reg_array[31:0]表示。

- ✓ 寄存器$0 固定为常量值 0，即无论读取还是写入，$0 始终为 32'b0。

2. **读操作：**

- ✓ **两路读端口：**

1. raddr1：用于指定第一路的读取寄存器地址。
2. raddr2：用于指定第二路的读取寄存器地址。

- ✓ **冒险处理：**支持从执行阶段（EX）、存储器阶段（MEM）和写回阶段（WB）的数据前递，优先返回最新数据，以解决流水线数据冒险问题。
- ✓ **实现代码：**

```
assign rdata1 = (raddr1 == 5'b0) ? 32'b0 :
                ((raddr1 == ex_rf_waddr) && ex_rf_we) ? ex_result :
                ((raddr1 == mem_rf_waddr) && mem_rf_we) ? mem_result :
                ((raddr1 == wb_rf_waddr) && wb_rf_we) ? wb_result :
                reg_array[raddr1];

assign rdata2 = (raddr2 == 5'b0) ? 32'b0 :
                ((raddr2 == ex_rf_waddr) && ex_rf_we) ? ex_result :
                ((raddr2 == mem_rf_waddr) && mem_rf_we) ? mem_result :
                ((raddr2 == wb_rf_waddr) && wb_rf_we) ? wb_result :
                reg_array[raddr2];
```

3. **写操作：**
   - ✓ **写端口：**
     1. we：写使能信号，只有在 we 为高电平且目标地址不为$0 时，才执行写入操作。
     2. waddr：指定目标寄存器地址。
     3. wdata：指定写入的数据。
   - ✓ **实现代码：**

```
always @ (posedge clk) begin
    if (we && waddr != 5'b0) begin
        reg_array[waddr] <= wdata;
    end
end
```

4. **数据前递（冒险处理）：**
   - ✓ 支持流水线数据前递功能，处理从 EX、MEM、WB 阶段的数据冒险。
   - ✓ 每个阶段传递的数据结构：

1. ex_to_id、mem_to_id、wb_to_id 分别包含寄存器写使能信号（rf_we）、写地址（rf_waddr）、写数据（rf_wdata）。

- ✓ **实现代码：**

```
assign {
    ex_rf_we,
    ex_rf_waddr,
    ex_result
} = ex_to_id;

assign {
    mem_rf_we,
    mem_rf_waddr,
    mem_result
} = mem_to_id;

assign {
    wb_rf_we,
    wb_rf_waddr,
    wb_result
} = wb_to_id;
```

5. **HI 和 LO 寄存器的支持：**
   - ✓ 模块中引入了 HI 和 LO 两个特殊寄存器，主要用于乘法和除法指令的结果存储，或者通过 MTHI、MTLO 指令直接写入。
   - ✓ **HI 寄存器：**
     1. 写入条件：hi_we 为高电平时，将 hi_data 写入 hi_o。
     2. 读取条件：hi_r 为高电平时，将 hi_o 的值通过 hilo_data 输出。
   - ✓ **LO 寄存器：**
     1. 写入条件：lo_we 为高电平时，将 lo_data 写入 lo_o。
     2. 读取条件：lo_r 为高电平时，将 lo_o 的值通过 hilo_data 输出。
   - ✓ **实现代码：**

```
always @ (posedge clk) begin
    if (hi_we) begin
        hi_o <= hi_data;
    end
end
always @ (posedge clk) begin
    if (lo_we) begin
        lo_o <= lo_data;
    end
```

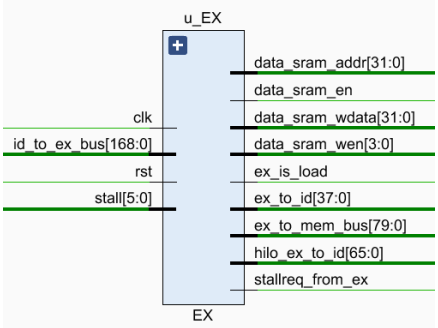
```
end
assign hilo_data = (hi_r) ? hi_o :
                   (lo_r) ? lo_o :
                   32'b0;
```

2.3 EX 段

2.3.1 整体功能说明

实现了 MIPS 处理器的执行阶段（EX）模块，该模块负责根据解码阶段（ID）传递过来的指令和操作数，执行具体的算术逻辑操作（ALU 操作）、乘法、除法，以及数据存储器访问操作等。模块通过选择不同的操作源（alu_src1 和 alu_src2），结合 ALU 运算类型信号（alu_op），生成 ALU 计算结果（alu_result）。此外，模块还实现了乘法和除法的操作，分别调用 mymul 和 div 模块处理有符号或无符号的乘除运算，结果存储在 HI 和 LO 寄存器中。模块同时通过 ex_to_mem_bus 将执行结果传递到存储器阶段（MEM），并通过 ex_to_id 向解码阶段反馈写回信息，保证流水线的正确性。此外，模块支持暂停控制，通过 stallreq_from_ex 指示当前阶段是否需要暂停流水线以等待乘法或除法的完成。结合代码实现，EX 模块通过 data_sram 接口生成内存访问的地址、写数据、写使能等信号，实现对数据存储器的读写操作，确保数据流和控制流的正确性。

2.3.2 端口介绍



输入端口：clk, rst, stall[5:0],id_to_ex_bus[168:0]

输出端口：ex_to_mem_bus[79:0], data_sram_en, data_sram_wen[3:0], data_sram_addr[31:0], ex_to_id[37:0], data_sram_wdata[31:0], stallreq_from_ex, ex_is_load, hilo_ex_to_id[65:0]

2.3.3 信号介绍

1. **流**水线寄存器信号

- ✓ **功能**: id_to_ex_bus_r 是从解码阶段（ID）传递到执行阶段（EX）的流水线寄存器，用于存储解码阶段的控制信号和操作数。
- ✓ **代码**:

```
reg [`ID_TO_EX_WD-1:0] id_to_ex_bus_r;
always @ (posedge clk) begin
    if (rst) begin
        id_to_ex_bus_r <= `ID_TO_EX_WD'b0;
    end
    else if (stall[2] == `Stop && stall[3] == `NoStop) begin
        id_to_ex_bus_r <= `ID_TO_EX_WD'b0;
    end
    else if (stall[2] == `NoStop) begin
        id_to_ex_bus_r <= id_to_ex_bus;
    end
end
```

2. 指令信号

- ✓ **功能**: 从 id_to_ex_bus_r 中解析当前指令 inst，并提取其控制信息（如 alu_op）和操作数（如 rf_rdata1,rf_rdata2）。
- ✓ **代码**:

```
assign {
    data_ram_readen,  //168:165
    inst_mthi,        //164
    inst_mtlo,        //163
    inst_multu,       //162
    inst_mult,        //161
    inst_divu,        //160
    inst_div,         //159
    ex_pc,            //148:117
    inst,             //116:85
    alu_op,           //84:83
    sel_alu_src1,     //82:80
    sel_alu_src2,     //79:76
    data_ram_en,      //75
```

```
    data_ram_wen,      //74:71
    rf_we,             //70
    rf_waddr,          //69:65
    sel_rf_res,        //64
    rf_rdata1,         //63:32
    rf_rdata2          //31:0
} = id_to_ex_bus_r;
```

3. ALU 操作信号

- ✓ **功能**: alu_op 表示 ALU 的具体操作类型, alu_src1 和 alu_src2 是 ALU 的两个输入操作数。alu_result 是 ALU 的计算结果。
- ✓ **代码**:

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

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

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

assign ex_result = alu_result;
```

4. 数据存储器控制信号

- ✓ **功能**: 控制对数据存储器的访问, 包括使能信号 (data_sram_en) 、写使能信号 (data_sram_wen) 、访问地址 (data_sram_addr) 和写入数据 (data_sram_wdata) 。
- ✓ **代码**:

```
assign data_sram_en = data_ram_en;

assign data_sram_wen =   (data_ram_readen == 4'b0101 &&
ex_result[1:0] == 2'b00) ? 4'b0001 :
```

```
                            (data_ram_readen == 4'b0101 &&
ex_result[1:0] == 2'b01) ? 4'b0010 :
                            (data_ram_readen == 4'b0101 &&
ex_result[1:0] == 2'b10) ? 4'b0100 :
                            (data_ram_readen == 4'b0101 &&
ex_result[1:0] == 2'b11) ? 4'b1000 :
                            (data_ram_readen == 4'b0111 &&
ex_result[1:0] == 2'b00) ? 4'b0011 :
                            (data_ram_readen == 4'b0111 &&
ex_result[1:0] == 2'b10) ? 4'b1100 :
                            data_ram_wen;

    assign data_sram_addr = ex_result;

    assign data_sram_wdata = data_sram_wen == 4'b1111 ? rf_rdata2 :
                            data_sram_wen == 4'b0001 ? {24'b0,
rf_rdata2[7:0]} :
                            data_sram_wen == 4'b0010 ? {16'b0,
rf_rdata2[7:0], 8'b0} :
                            data_sram_wen == 4'b0100 ? {8'b0,
rf_rdata2[7:0], 16'b0} :
                            data_sram_wen == 4'b1000 ?
{rf_rdata2[7:0], 24'b0} :
                            data_sram_wen == 4'b0011 ? {16'b0,
rf_rdata2[15:0]} :
                            data_sram_wen == 4'b1100 ?
{rf_rdata2[15:0], 16'b0} :
                            32'b0;
```

5. 流水线数据传递信号

- ✓ **功能**: ex_to_mem_bus 将执行结果和控制信号传递到存储器阶段（MEM）。

  ex_to_id 将当前阶段的写回信号反馈到解码阶段（ID）。

- ✓ **代码**:

```
    assign ex_to_mem_bus = {
        data_ram_readen, //79:76
        ex_pc,           //75:44
        data_ram_en,     //43
        data_ram_wen,    //42:39
        sel_rf_res,      //38
        rf_we,           //37
        rf_waddr,        //36:32
```

```
    ex_result        //31:0
};

assign ex_to_id = {
    rf_we,           //37
    rf_waddr,        //36:32
    ex_result        //31:0
};
```

6. 乘法控制信号

- ✓ 功能：支持有符号和无符号的乘法操作，调用 mymul 模块完成具体计算。mul_start_o 控制乘法开始，mul_result 保存乘法结果。
- ✓ 代码:

```
mymul my_mul(
    .rst            (rst           ),
    .clk            (clk           ),
    .signed_mul_i   (signed_mul_o  ),
    .a_o            (mul_opdata1_o ),
    .b_o            (mul_opdata2_o ),
    .start_i        (mul_start_o   ),
    .result_o       (mul_result    ),
    .ready_o        (mul_ready_i   )
);

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

```
                2'b01: begin
                    if (mul_ready_i == `MulResultNotReady) begin
                        stallreq_for_mul = `Stop;
                        mul_start_o = `MulStart;
                        signed_mul_o = 1'b0;
                    end
                end
            endcase
        end
    end
```

7. 除法控制信号

- ✓ **功能**: 支持有符号和无符号的除法操作, 调用 div 模块完成具体计算。

  div_start_o 控制除法开始, div_result 保存除法结果。

- ✓ **代码**:

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

always @ (*) begin
    if (rst) begin
        stallreq_for_div = `NoStop;
        div_start_o = `DivStop;
        signed_div_o = 1'b0;
    end else begin
        case ({inst_div, inst_divu})
            2'b10: begin
                if (div_ready_i == `DivResultNotReady) begin
                    stallreq_for_div = `Stop;
                    div_start_o = `DivStart;
                    signed_div_o = 1'b1;
                end
```

```
                end
                2'b01: begin
                    if (div_ready_i == `DivResultNotReady) begin
                        stallreq_for_div = `Stop;
                        div_start_o = `DivStart;
                        signed_div_o = 1'b0;
                    end
                end
            endcase
        end
    end
```

8. HI/LO 寄存器信号

- ✓ **功能**：控制 HI/LO 寄存器的写入，用于存储乘法和除法结果。
- ✓ **代码**：

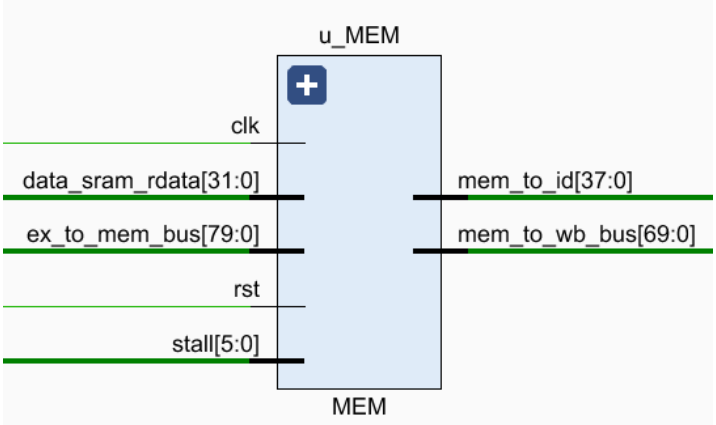
```
assign hi_wen = inst_divu | inst_div | inst_mult | inst_multu |
inst_mthi;
assign lo_wen = inst_divu | inst_div | inst_mult | inst_multu |
inst_mtlo;

assign hi_data =  (inst_div | inst_divu)   ?
div_result[63:32] :
                  (inst_mult | inst_multu) ?
mul_result[63:32] :
                  (inst_mthi)              ? rf_rdata1        :
                  32'b0;

assign lo_data =  (inst_div | inst_divu)   ? div_result[31:0] :
                  (inst_mult | inst_multu) ? mul_result[31:0] :
                  (inst_mtlo)              ? rf_rdata1        :
                  32'b0;

assign hilo_ex_to_id = {
    hi_wen,          // 65
    lo_wen,          // 64
    hi_data,         // 63:32
    lo_data          // 31:0
};
```

2.4 MEM 段

2.4.1 整体功能说明

这段代码实现了 MIPS 处理器流水线中的存储器访问阶段（MEM）模块，该模块负责处理数据存储器的读写操作，并将执行阶段（EX）传递的结果数据加工后，传递到写回阶段（WB）以及反馈给解码阶段（ID）。具体而言，当启用数据存储器访问时（data_ram_en 为 1），模块根据 data_ram_readen 信号解析出数据存储器的读取模式（如字节、半字、字等），并根据存储器返回的数据 data_sram_rdata 以及访问地址 ex_result，对数据进行扩展（如符号扩展或零扩展）。如果未启用存储器访问，则将 EX 阶段的计算结果直接传递到写回阶段（rf_wdata）。此外，模块通过 mem_to_id 将当前写回相关信号反馈到 ID 阶段，解决潜在的数据依赖问题。整个过程受流水线控制信号 stall 影响，支持暂停信号的传播与寄存器清空逻辑，确保流水线运行的正确性和数据的完整性。

2.4.2 端口介绍



输入端口：clk, rst, stall[5:0], ex_to_mem_bus[79:0], data_sram_rdata[31:0]

输出端口：mem_to_id[37:0], mem_to_wb_bus[69:0]

2.4.3 信号介绍

1. 流水线寄存器信号

- ✓ **功能**：ex_to_mem_bus_r 是从执行阶段（EX）传递到存储器阶段（MEM）的流水线寄存器，用于存储 EX 阶段生成的控制信号和数据。
- ✓ **代码**：

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

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

- ✓ 在复位信号 rst 拉高时，清空寄存器内容。
- ✓ 当流水线暂停（stall[3]为 Stop）时，保持当前寄存器内容不变。
- ✓ 当流水线继续（stall[3]为 NoStop）时，更新寄存器内容为 ex_to_mem_bus。

2. 从 EX 阶段传递的信号解析

- ✓ **功能**：从 ex_to_mem_bus_r 中解析出 EX 阶段传递的信号，包括内存控制信号、执行结果和寄存器写回信号。
- ✓ **代码**：

```
assign {
    data_ram_readen, // 79:76
    mem_pc,          // 75:44
    data_ram_en,     // 43
    data_ram_wen,    // 42:39
    sel_rf_res,      // 38
    rf_we,           // 37
    rf_waddr,        // 36:32
    ex_result        // 31:0
} = ex_to_mem_bus_r;
```

- ✓ **data_ram_readen**：存储器读取模式信号（字、半字、字节等）。
- ✓ **mem_pc**：指令对应的程序计数器（PC）值。

- ✓ data_ram_en：存储器访问使能信号，指示是否需要访问数据存储器。
- ✓ data_ram_wen：存储器写使能信号，指示写操作的字节选择。
- ✓ sel_rf_res：寄存器写回数据选择信号（从存储器或 ALU 结果）。
- ✓ rf_we：寄存器写使能信号，指示是否需要写回寄存器。
- ✓ rf_waddr：寄存器写地址。
- ✓ ex_result：EX 阶段的 ALU 计算结果或内存访问地址。

3. 寄存器写回数据选择

- ✓ **功能**：根据存储器读取模式（data_ram_readen）和存储器数据（data_sram_rdata），生成最终的写回数据（rf_wdata）。
- ✓ **代码**：

```
assign rf_wdata = (data_ram_readen == 4'b1111 && data_ram_en == 1'b1) ? data_sram_rdata :
                  (data_ram_readen == 4'b0001 && data_ram_en == 1'b1 && ex_result[1:0] == 2'b00) ? ({{24{data_sram_rdata[7]}}, data_sram_rdata[7:0]}) :
                  (data_ram_readen == 4'b0001 && data_ram_en == 1'b1 && ex_result[1:0] == 2'b01) ? ({{24{data_sram_rdata[15]}}, data_sram_rdata[15:8]}) :
                  (data_ram_readen == 4'b0001 && data_ram_en == 1'b1 && ex_result[1:0] == 2'b10) ? ({{24{data_sram_rdata[23]}}, data_sram_rdata[23:16]}) :
                  (data_ram_readen == 4'b0001 && data_ram_en == 1'b1 && ex_result[1:0] == 2'b11) ? ({{24{data_sram_rdata[31]}}, data_sram_rdata[31:24]}) :
                  (data_ram_readen == 4'b0010 && data_ram_en == 1'b1 && ex_result[1:0] == 2'b00) ? ({24'b0, data_sram_rdata[7:0]}) :
                  (data_ram_readen == 4'b0010 && data_ram_en == 1'b1 && ex_result[1:0] == 2'b01) ? ({24'b0, data_sram_rdata[15:8]}) :
                  (data_ram_readen == 4'b0010 && data_ram_en == 1'b1 && ex_result[1:0] == 2'b10) ? ({24'b0, data_sram_rdata[23:16]}) :
```

```
                    (data_ram_readen == 4'b0010 && data_ram_en ==
1'b1 && ex_result[1:0] == 2'b11) ? ({24'b0,
data_sram_rdata[31:24]}) :
                    (data_ram_readen == 4'b0011 && data_ram_en ==
1'b1 && ex_result[1:0] == 2'b00) ? ({{16{data_sram_rdata[15]}},
data_sram_rdata[15:0]}) :
                    (data_ram_readen == 4'b0011 && data_ram_en ==
1'b1 && ex_result[1:0] == 2'b10) ? ({{16{data_sram_rdata[31]}},
data_sram_rdata[31:16]}) :
                    (data_ram_readen == 4'b0100 && data_ram_en ==
1'b1 && ex_result[1:0] == 2'b00) ? ({16'b0,
data_sram_rdata[15:0]}) :
                    (data_ram_readen == 4'b0100 && data_ram_en ==
1'b1 && ex_result[1:0] == 2'b10) ? ({16'b0,
data_sram_rdata[31:16]}) :
                    ex_result;
```

- ✓ 根据 data_ram_readen 和地址低两位 ex_result[1:0]，选择读取的字节或半字并进行符号扩展或零扩展。
- ✓ 如果不需要读取存储器（data_ram_en 为 0），直接返回 ex_result 作为写回数据。

4. **数据通路输出信号**

- ✓ **功能**:

  mem_to_wb_bus: 将当前阶段的结果传递给写回阶段（WB）。

  mem_to_id: 将写回信号反馈给解码阶段（ID）以解决数据冒险。

- ✓ **代码**:

```
assign mem_to_wb_bus = {
    mem_pc,     // 41:38 - 程序计数器
    rf_we,      // 37    - 寄存器写使能信号
    rf_waddr,   // 36:32 - 寄存器写地址
    rf_wdata    // 31:0  - 寄存器写回数据
};

assign mem_to_id = {
    rf_we,      // 37    - 寄存器写使能信号
    rf_waddr,   // 36:32 - 寄存器写地址
```

```
    rf_wdata     // 31:0  - 寄存器写回数据
};
```

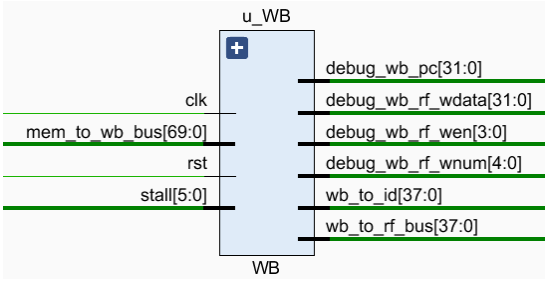
- ✓ **mem_to_wb_bus**：打包存储器访问结果和相关控制信号，传递给 WB 阶段。
- ✓ **mem_to_id**：将当前写回信息（写地址、写数据）反馈给 ID 阶段，用于解决潜在的数据依赖问题。

2.5 WB 段

2.5.1 整体功能说明

这段代码实现了 MIPS 处理器流水线的写回阶段（WB），该模块负责将存储器阶段（MEM）传递过来的结果数据写回到寄存器文件，同时向解码阶段（ID）反馈写回信息以解决数据冒险问题。模块通过 mem_to_wb_bus_r 流水线寄存器存储 MEM 阶段传递的控制信号和数据（包括程序计数器 wb_pc、寄存器写使能信号 rf_we、寄存器写地址 rf_waddr 和写数据 rf_wdata），并将这些信号加工后生成两条输出路径：一是通过 wb_to_rf_bus 将写回信号发送到寄存器文件，二是通过 wb_to_id 将写回相关信息反馈到 ID 阶段。同时，该模块提供了调试信号（debug_wb_pc、debug_wb_rf_wen 等），用于监控写回阶段的关键操作。整个过程支持暂停信号（stall）的控制逻辑，确保流水线的正确性和数据完整性。

2.5.2 端口介绍



输入端口：clk, rst, stall[5:0], mem_to_wb_bus[79:0]

输出端口：wb_to_rf_bus[37:0], wb_to_id[37:0], debug_wb_pc[31:0], debug_wb_rf_wen[3:0], debug_wb_rf_wnum[4:0], debug_wb_rf_wdata[31:0]

2.5.3 信号介绍

1. 流水线寄存器信号

- ✓ **功能**：mem_to_wb_bus_r 是从存储器阶段（MEM）传递到写回阶段（WB）的流水线寄存器，用于存储 MEM 阶段生成的控制信号和数据。
- ✓ **代码**：

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

always @ (posedge clk) begin
    if (rst) begin
        mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
    end
    else if (stall[4] == `Stop && stall[5] == `NoStop) begin
        mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
    end
    else if (stall[4] == `NoStop) begin
        mem_to_wb_bus_r <= mem_to_wb_bus;
    end
end
```

- ✓ **复位信号（rst）**：当复位信号拉高时，清空寄存器内容。
- ✓ **暂停信号（stall）**：

  如果流水线暂停，保持当前寄存器内容不变。

  如果流水线继续运行，更新寄存器内容为 mem_to_wb_bus。

- ✓ **作用**：通过寄存器保持从 MEM 阶段传递过来的写回信息，确保数据在流水线中的正确传播。

2. 从 MEM 阶段传递的信号解析

- ✓ **功能**：从 mem_to_wb_bus_r 中解析出存储器阶段传递的控制信号和写回数据。
- ✓ **代码**：

```
assign {
    wb_pc,          // 程序计数器（PC）值
    rf_we,          // 寄存器写使能信号
    rf_waddr,       // 寄存器写地址
```

```
    rf_wdata      // 寄存器写回数据
} = mem_to_wb_bus_r;
```

- ✓ **wb_pc**：存储器阶段对应指令的程序计数器值，用于调试和追踪写回阶段的操作。
- ✓ **rf_we**：寄存器写使能信号，指示是否需要写回数据到寄存器文件。
- ✓ **rf_waddr**：寄存器写地址，指定写回的目标寄存器编号。
- ✓ **rf_wdata**：寄存器写回的数据，可能是 ALU 结果或存储器读取数据。

3. 寄存器写回数据传递

- ✓ **功能**：将写回信号和数据通过 wb_to_rf_bus 传递给寄存器文件。
- ✓ **代码**：

```
assign wb_to_rf_bus = {
    rf_we,          // 写使能信号
    rf_waddr,       // 写地址
    rf_wdata        // 写数据
};
```

  - ○ **wb_to_rf_bus**：用于寄存器文件的写入信号和数据总线。
    - ▪ **rf_we**：控制寄存器文件是否写入。
    - ▪ **rf_waddr**：目标寄存器编号。
    - ▪ **rf_wdata**：需要写入的数据。

4. 写回信息反馈给解码阶段

- ✓ **功能**：通过 wb_to_id 将当前写回相关信息（写使能信号、写地址、写数据）反馈给解码阶段（ID），用于解决数据冒险问题。
- ✓ **代码**：

```
assign wb_to_id = {
    rf_we,          // 写使能信号
    rf_waddr,       // 写地址
    rf_wdata        // 写数据
```

```
};
```

- ✓ **wb_to_id**：写回阶段生成的信息反馈到解码阶段，提供写地址和数据，确保在下一指令需要该数据时能够直接获取。

5. **调试信号**

- ✓ **功能**：生成调试信息，用于监控写回阶段的执行状态，包括当前指令的 PC 值、寄存器写使能信号、写入的寄存器编号和写入的数据。
- ✓ **代码**：

```
assign debug_wb_pc = wb_pc;                    // 当前指令的 PC 值
assign debug_wb_rf_wen = {4{rf_we}};           // 写使能信号（扩展为 4 位以适配调试总线）
assign debug_wb_rf_wnum = rf_waddr;            // 写入寄存器编号
assign debug_wb_rf_wdata = rf_wdata;           // 写入寄存器的数据
```

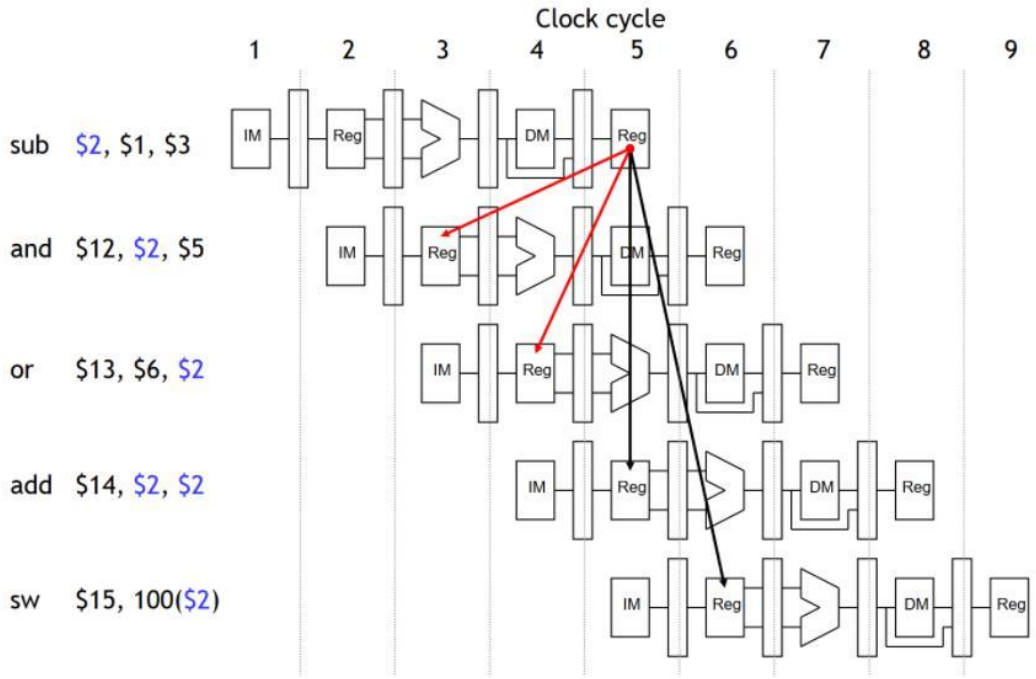
- ✓ **debug_wb_pc**：写回阶段对应的指令 PC 值，用于调试时定位具体指令。
- ✓ **debug_wb_rf_wen**：写使能信号（4 位扩展）。
- ✓ **debug_wb_rf_wnum**：写入寄存器编号。
- ✓ **debug_wb_rf_wdata**：写入寄存器的数据。

三 实验过程遇见的问题

数据相关：

问题描述:

在流水线中运行指令的过程中，难免会出现以下的问题。



可以看到，SUB 指令的结果是 AND 指令的输入，也是 OR 指令的输入；而 ADD 指令与 SW 指令的输入同样依赖 SUB 指令。其中，红线由前指向后，因此从时间流动的角度，红线就代表一处「数据冲突」，即 Data Hazard.

解决办法：数据前递 Data Forwarding

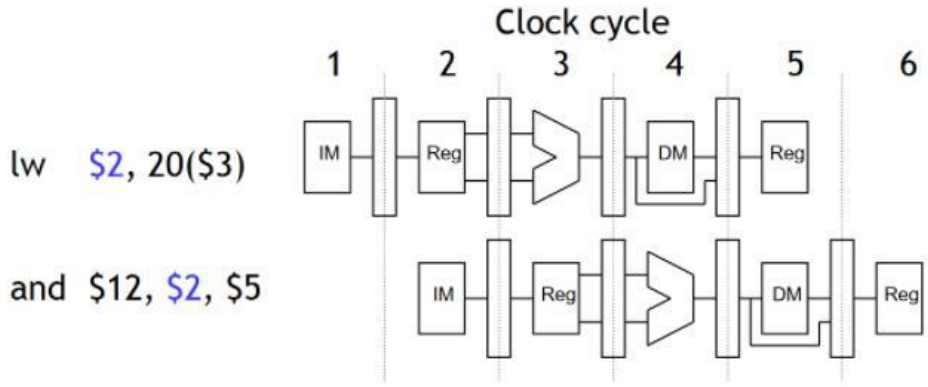
事实上，对 SUB 指令来说，其结果的产生是在其第三流水阶段：EX，也就是整个流水线 CPU 的第三个时钟周期。而对 SUB 之后的 AND 以及 OR 来说，寄存器 $2 的值是在它们的第三阶段 EX 需要的，也就是流水线 CPU 的第 4-5 个时钟周期。纵观整个流水过程，事实上 $2 的值是在第 3 个时钟周期产生，并在第 4-5 个时钟周期需要，因此我们只需要越过流水线五个阶段中 WB（WriteBack）的过程，让第 3 个时钟周期里面计算产生的 $2 寄存器值直接赋值给第 4-5 个时钟周期指令 ADD 以及 OR 指令即可。
实现：在 EX、MEM、WB（其实 WB 段可以没有）中将当前时钟周期内得到的要写入寄存器的地址以及要写入的数据，发给 ID 段，在 ID 判断当前操作数的地址是否与其将要写入寄存器的地址相等，如果相等的话，就直接将要写入寄存器的值拿过来用即可。
另外，在 hilo 寄存器中也会有数据相关的问题。不过和通用的寄存器不太相同的地方是，我们的 hilo 寄存器在 EX 段得到要写入的值后就会直接发送给 ID 段，在下一个周期的 ID 段完成写入，因此在 hilo 寄存器中的数据相关只存在于两条相邻的指令中，而 MEM 和 WB 段不会存在 hilo 寄存器的数据相关的问题。第一条指令需要完成 hilo 寄存器的写入，而第二条指令请求读取 hilo 寄存器的值。

解决方法同样是使用了数据前递，因为上一个周期需要写入 hilo 寄存器的值已经发送给了 ID 段，只不过还没有写入到 hilo 寄存器中。因此如果发生了 hilo 寄存器的数据相关，在该 ID 段就不需要从 hilo 寄存器中读取，直接将上一个时钟周期的 EX 段发过来的要写入 hilo 寄存器的值拿过来用就可以了。

访存冲突

问题描述:

虽然前面介绍的方法规避了 ALU 算术运算的数据冲突，但是对于需要访问数据存储器的指令（比如 LW）来说，我们并不能规避类似下面的指令带来的数据冲突：



也就是要用的寄存器的值在用的时候并不是最新的值，最新的值还没有写进去。

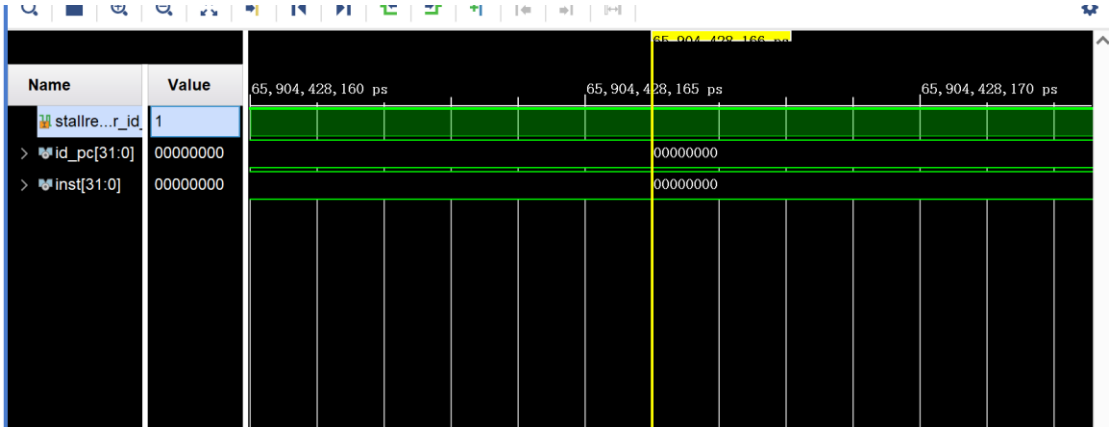
解决方法:

在 LW 指令的 MEM 阶段，我们就获得了相应的数据，那么，对于下一条 AND 指令，我们只需要在其 EX 阶段前将流水线 Stall 住一个时钟周期，即可将 Data Memory 的数据前递至正确的地方。这种解决方法也叫向流水线中引入一个 bubble。

实现:

在 EX 段将是否是 load 指令、和将要写入的寄存器的地址发送给 ID 段，在

ID 段进行判断，如果 ID 段请求读取数据的寄存器的地址和 EX 段将要写入的寄存器的地址想同，并且 EX 段当前是 load 类指令，则可以判断发生了访存冲突。需要在 ID 段发送给 CTRL 段请求说明 ID 段有访存冲突，然后 CTRL 段就会发送给流水线各个部位 stall 值，流水线的各个部位再根据 stall 的值进行相对应的暂停处理。等到 EX 段获得了 ID 段需要读取的数据后，再将该数据直接发送到 ID 段，然后流水线恢复正常运行。如果没有恢复指令，则会出现下面的问题。



观察波形图可知发现一直加气泡，导致 pc 始终为 0，也就是说并没有恢复操作或者说因为某个细节没处理到位导致暂停指令始终为 1。为解决这个问题，我们检查了 stallreq_from_id 的赋值情况发现括号位置有误，导致一直为 1，一直处于暂停状态。

四 实验感受及改进意见

4.1 段雨洁

这次实验让我对 CPU 的设计有了更加深刻的认识，特别是流水线设计的复杂性和其中的关键问题。通过实现五级流水线（IF、ID、EX、MEM、WB）模块，我理解了如何在每个阶段传递数据，并有效处理控制冒险和数据冒险问题。尤其是在实现寄存器文件（regfile）模块时，我终于明白了前递机制的重要性，以及如何通过解决数据依赖来避免流水线停顿。此外，在实现跳转指令（如 jal 和 jr）时，对指令地址计算有了更清晰的理解。这让我感受到从理论到实践的巨大差异，也真正理解了计算机系统课本中“分支预测”和“数据前递”的实际含义。

在实验过程中，我遇到了不少问题。比如，在实现 lw 指令时，发现 EX 阶段无法获取正确的加载数据，这导致了之后的指令使用了错误的数据。为了解决这个问题，我们引入了“气泡”（暂停一个时钟周期），这虽然解决了问题，但也导致了一些新的挑战，例如气泡的实现导致流水线中的数据停滞。这使我深刻体会到流水线控制的复杂性以及对设计精准性的高要求。我们通过 Git 来管理代码，每个人负责不同的模块，这种分工合作让我们能够更高效地完成工作。

因为我们没怎么学过相关数电、计组的知识，希望学校可以给课程安排的合理一点，使用 vivado 以及它的语法 verilog 都是没接触过的。

4.2 石晓瑞

这次实验的过程让我深刻体会到了计算机体系结构和硬件描述语言（如 Verilog）的复杂性和挑战性。在实验初期，我感到非常迷茫，尤其是对实验内容和方向的不确定。通过向老师和助教的询问，我逐渐了解到我们需要通过分析波形图来定位错误，这一理论上的指导为我的实践过程奠定了基础。然而，真正开始动手进行代码调试及修改时，我意识到困难远超想象。

实验使用了一门新的编程语言，Verilog。为了适应这门语言，我认真阅读了老师提供的《自己动手做 CPU——雷思磊》，试图从中学习它的语法和特点。经过阅读第一、第二章，我对 Verilog 有了一定的理解，尤其是由于我之前有过 C++ 等编程语言的基础，使我能够较快地上手。后续章节主要介绍了流水线的实现逻辑，我将其与 SimpleCPU 进行对照，虽然实现方式各有不同，但我发现它们在底层逻辑上的一致性为我理解整体结构提供了帮助。

在掌握了代码的大致情境后，我开始着手修改代码。最初遇到的难题是数据相关问题，即 Read After Write (RAW) 的数据冲突。我查阅了相关资料，了解到需要针对三条数据通路进行处理：ex_to_id、mem_to_id 和 wb_to_id。在 ID、EX、MEM 和 WB 阶段添加数据通路后，问题依然存在。经过进一步调研，我发现是因为在 mycpu_core 中未正确实例化这些通路，导致它们并未真正连接。解决这一问题后，数据相关问题得到缓解，但新的挑战接踵而至。

在指令的实现上，一开始的几条指令的实现相对简单，但在实现 lw (load word) 指令时，我感到困惑，甚至对 lw 的功能和流程的理解都存在障碍。经过查阅大量资料，我意识到 lw 和 sw (store word) 指令与存储器的关系十分密切，这是数据存储的核心部分。了解 lw 指令后又遇到一个新的问题：由于 lw 指令的结果必须在下一个时钟周期才能计算出来，而如果下条指令依赖于这个结果，就必须在执行前暂停一个时钟周期，这就是所谓的“加气泡”解决方案。

实验过程中我还遇到了一些其他问题，尤其是在实现乘除指令时，为此我们尝试设计了一个乘法器，并通过高低位分开存储来解决乘法后位数成倍增加的问题。这一过程让我更加熟悉了 Verilog 的特性和设计思路。

通过本次实验，我不仅提高了对 Verilog 编程语言的理解，还对计算机体系结构有了更为深刻的认识。实践中遇到的每一个问题都促进了我的学习，我意识到理论知识和实践能力的重要性。今后在学习和工作中，我会更加注重查阅资料与深入思考，努力提高自己的解决问题的能力。

五 参考文献

[1] 雷思磊.《自己动手做 cpu_雷思磊》[M/CD].

[2] 《“系统能力培养大赛”MIPS指令系统规范_v1.01》[M/CD].