实验报告成绩:	成绩评定日期:

2024 ~ 2025 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：张腾飞

组员：余敏波

报告日期：2025.1.4

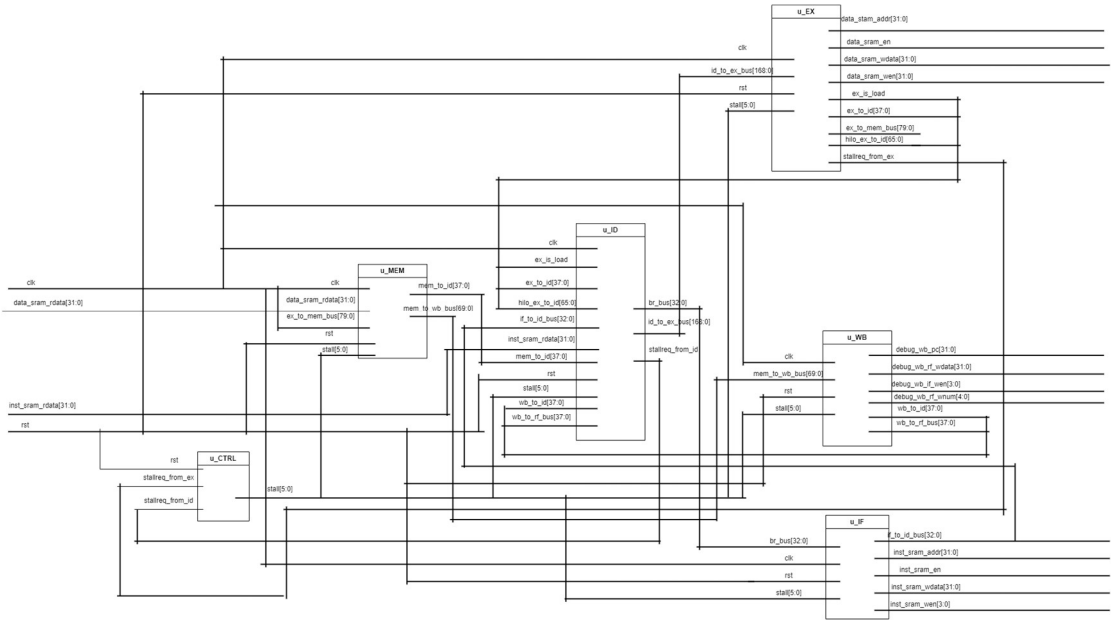
一、工作量划分

张腾飞：regfile.v、IF.v、ID.v　　50%

余敏波：EX.v、MEM.v、WB.v　　50%

二、 总体设计

该流水线设计包括七个板块：IF.v, ID.v, EX.v, MEM.v, WB.v, CTRL.v, regfile.v。总共 32 个 32 位整数寄存器，采用大端模式存储，具有 32bit 数据地址总线宽度。



三、 各文件说明

1. regfile.v

该文件是寄存器文件模块，用于处理寄存器的读写操作。主要功能和组成部分如下：

1. **输入信号**：

- clk：时钟信号，用于同步电路操作。

- raddr1 和 raddr2：读取地址 1 和读取地址 2，分别用于指定要读取的寄存器 地址。
- we：写使能信号，用于指定是否写入寄存器。
- waddr：写入地址，用于指定要写入的寄存器地址。
- wdata：写入数据，用于指定要写入的寄存器数据。
- hi_r 和 lo_r：读取高位和低位寄存器的使能信号。
- hi_we 和 lo_we：写入高位和低位寄存器的使能信号。
- hi_data 和 lo_data：写入高位和低位寄存器的数据。

2. **输出信号**：

- rdata1 和 rdata2：读取数据 1 和读取数据 2，分别用于输出读取的寄存器数据。
- hilo_data：从高位或低位寄存器读取的数据。

3. **内部寄存器**：

- hi_o 和 lo_o：高位和低位寄存器，用于存储特殊寄存器的数据。
- reg_array：寄存器数组，用于存储 32 个通用寄存器的数据。

4. **主要逻辑**：

写操作:

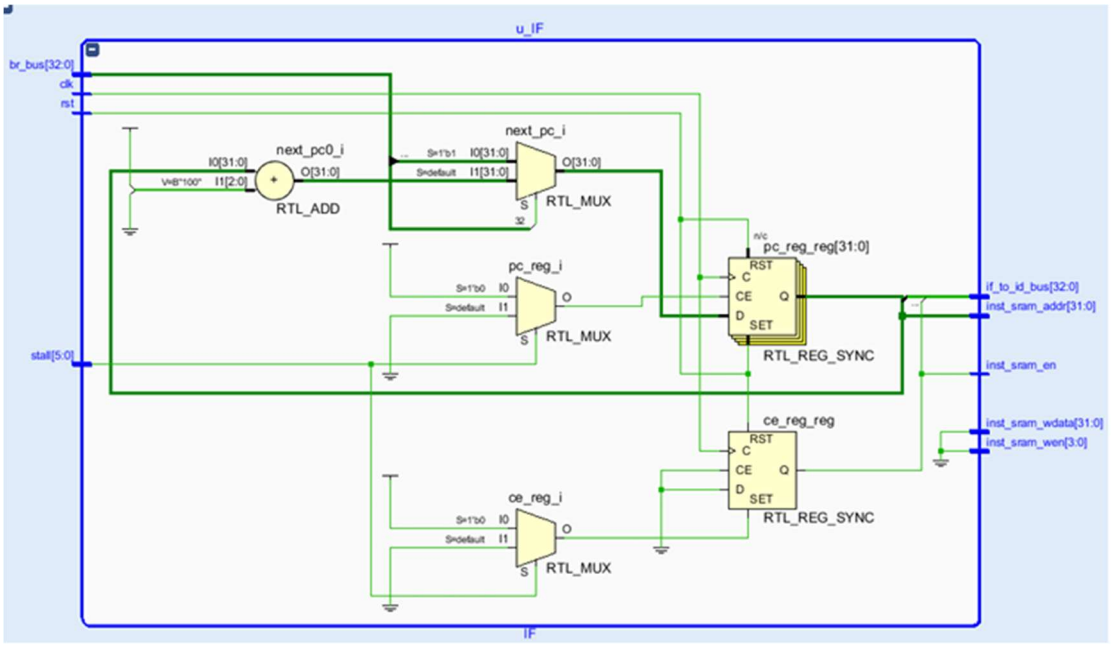
- 在时钟上升沿，如果 hi_we 有效，更新高位寄存器 hi_o。
- 在时钟上升沿，如果 lo_we 有效，更新低位寄存器 lo_o。
- 在时钟上升沿，如果 we 有效且写入地址不为 0，更新寄存器数组中对应地址的值。

读操作:

- 根据 hi_r 和 lo_r 信号，选择输出高位或低位寄存器的数据到 hilo_data。

根据 raddr1 和 raddr2 信号，读取寄存器数组中对应地址的值到 rdata1 和 rdata2，如果地址为 0，则输出 0。

2. IF.v



该文件定义了一个名为 IF 的指令获取模块，用于从指令存储器中获取指令，并将其传递给指令解码阶段（ID），以便进行进一步的指令解码和执行。该模块的主要功能和组成部分如下：

1. **输入信号**：

- clk：时钟信号，用于同步电路操作。
- rst：复位信号，用于初始化模块状态。
- stall：暂停信号，用于控制模块的暂停和继续。
- br_bus：分支总线信号，包含分支使能信号和分支地址。

2. **输出信号**：

- if_to_id_bus：传递给 ID 阶段的数据总线，包含指令存储器使能信号和程序计数器值。
- inst_sram_en：指令 SRAM 使能信号。
- inst_sram_wen：指令 SRAM 写使能信号，固定为 0，因为这是一个读取操作。
- inst_sram_addr：指令 SRAM 地址信号，等于当前程序计数器值。
- inst_sram_wdata：指令 SRAM 写数据信号，固定为 0，因为这是一个读取操作。

3. **内部寄存器**：

- pc_reg：程序计数器寄存器，存储当前指令地址。
- ce_reg：指令存储器使能寄存器，控制指令存储器的使能状态。

4. **内部信号**：

- next_pc：下一个程序计数器值，根据当前程序计数器值和分支信号计算得出。
- br_e 和 br_addr：从分支总线信号中提取的分支使能信号和分支地址。

5. **主要逻辑**：

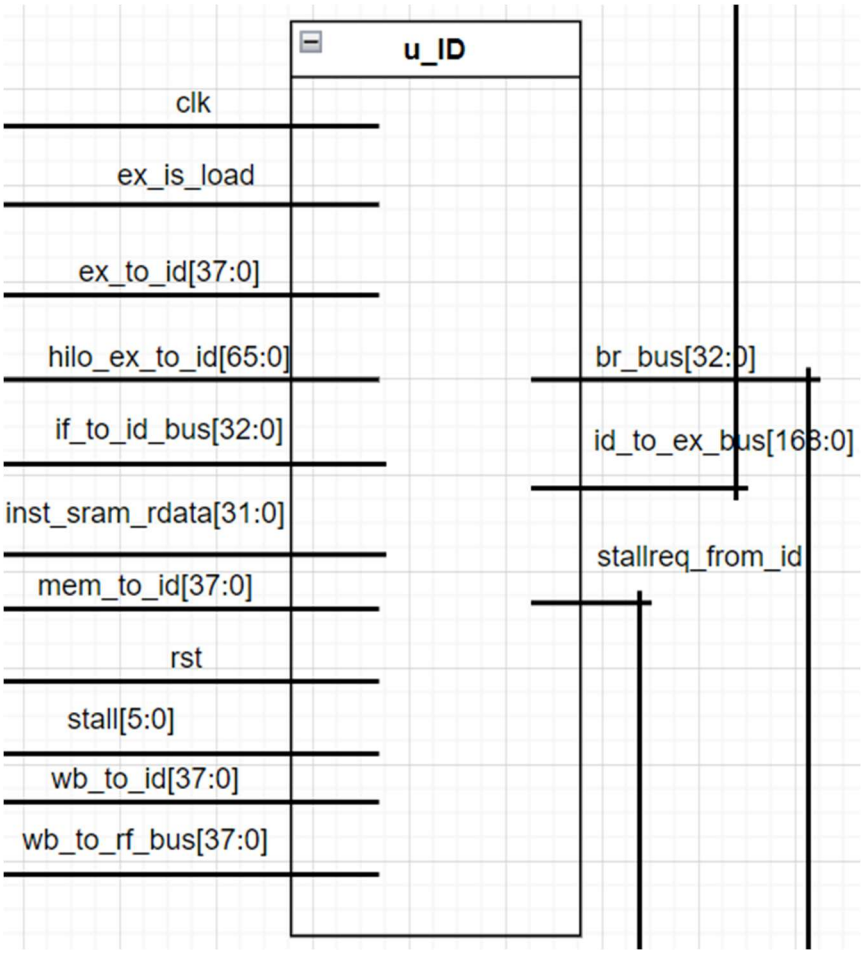
在时钟上升沿，如果复位信号有效，程序计数器初始化为特定值；否则，如 果没有暂停信号，更新程序计数器为下一个值。

在时钟上升沿，如果复位信号有效，指令存储器使能信号为 0；否则，如果没有暂停信号，使能指令存储器。

计算下一个程序计数器值，如果有分支信号，使用分支地址；否则，程序计数器加 4。

分配指令 SRAM 信号和传递给 ID 阶段的数据总线信号。

3. ID.v



该文件实现了 CPU 流水线中的指令解码阶段。ID 段的主要任务是从指令获取阶段（IF 段）接收指令，并对指令进行解码，生成相应的控制信号和数据，传递给执行阶段（EX 段）。该模块的主要功能和组成部分如下：

1. 输入信号：

   clk：时钟信号，用于同步电路操作。

   rst：复位信号，用于初始化模块状态。

   stall：暂停信号，用于控制模块的暂停和继续。

   ex_is_load：指示 EX 段是否是加载指令的信号。

if_to_id_bus：从 IF 段传来的数据总线。

inst_sram_rdata：从 SRAM 读取的指令。

wb_to_rf_bus：从写回阶段（WB 段）传来的数据总线。

ex_to_id、mem_to_id、wb_to_id：分别从 EX 段、MEM 段和 WB 段传来的数 据。

hilo_ex_to_id：从 EX 段传来的高位和低位寄存器数据。

2. 输出信号：

id_to_ex_bus：传递给 EX 段的数据总线。

br_bus：传递给分支预测单元的数据总线。

stallreq_from_id：从 ID 段传来的暂停请求信号。

3. 内部寄存器和信号：

if_to_id_bus_r：存储从 IF 段传来的数据。

inst：当前指令。

id_pc：当前指令的程序计数器值。

ce：控制信号。

wb_rf_we、wb_rf_waddr、wb_rf_wdata：写寄存器文件的使能信号、地址和数据。

wb_id_we、wb_id_waddr、wb_id_wdata：写 ID 段的使能信号、地址和数据。

mem_id_we、mem_id_waddr、mem_id_wdata：写 MEM 段的使能信号、地址和数据。

ex_id_we、ex_id_waddr、ex_id_wdata：写 EX 段的使能信号、地址和数据。

opcode、rs、rt、rd、sa、func、imm、instr_index、code、base、offset、sel：指令的各个字段。

op_d、func_d：操作码和功能码的解码结果。

rs_d、rt_d、rd_d、sa_d：寄存器地址的解码结果。

sel_alu_src1、sel_alu_src2：ALU 操作数选择信号。

alu_op：ALU 操作码。

data_ram_en、data_ram_wen、data_ram_readen：数据存储器的使能信号、写使能信号和读使能信号。

rf_we、rf_waddr、sel_rf_res、sel_rf_dst：寄存器文件的写使能信号、写地址、结果选择信号和目标寄存器选择信号。

rdata1、rdata2、rdata11、rdata22、rdata111：寄存器数据。

hi_r、hi_wen、lo_r、lo_wen、hi_data、lo_data、hilo_data：高位和低位寄存器的读写信号和数据。

br_e、br_addr、rs_eq_rt、rs_ge_z、rs_gt_z、rs_le_z、rs_lt_z、pc_plus_4：分支相关信号

4. 主要逻辑：

数据寄存：在时钟上升沿，如果复位信号有效，清空从 IF 段传来的数据；否则，根据暂停信号更新从 IF 段传来的数据。

指令选择：根据暂停信号选择当前指令。

指令解码：从指令中提取操作码、寄存器地址、功能码等字段。

使用解码器对操作码和功能码进行解码，生成相应的控制信号。

寄存器文件操作：实例化寄存器文件模块，处理寄存器的读写操作。

根据指令类型和数据依赖关系，选择正确的寄存器数据。

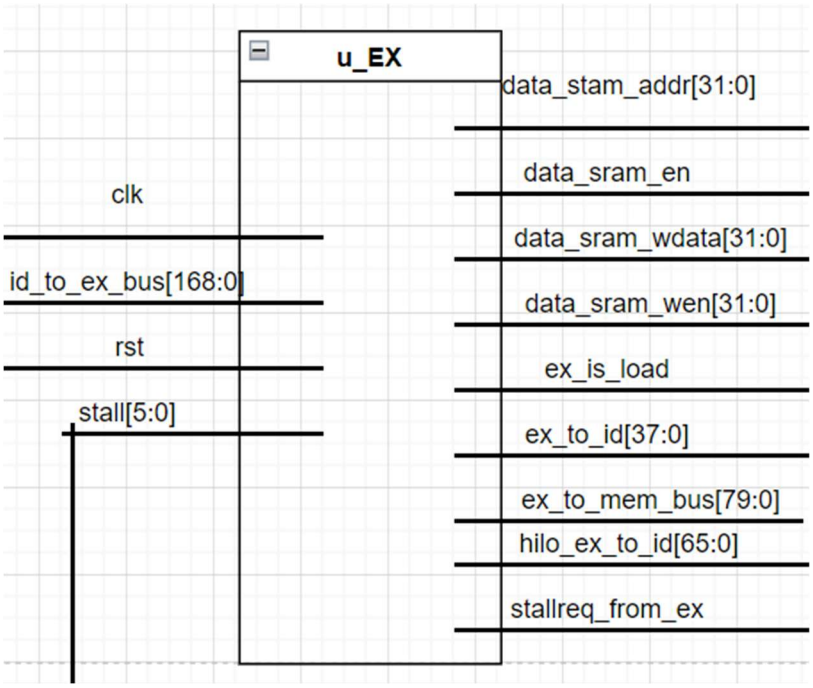
ALU 操作选择：根据指令类型，生成 ALU 操作码和操作数选择信号。

数据存储器操作：根据指令类型，生成数据存储器的使能信号、写使能信号和读使能信号。

分支预测：根据指令类型和寄存器数据，生成分支使能信号和分支地址。

暂停请求：根据数据依赖关系，生成暂停请求信号。

3.EX.v



（Execute，执行）阶段是负责指令实际操作的阶段。这一阶段的主要任务是执行算术和逻辑运算、地址计算或其他需要处理的数据操作。在原有的基础上添加了 ex_to_id、stallreq_from_ex、ex_is_load、hilo_ex_to_id 接口。以下为各个接口作用：

1.输入接口：

1.clk，宽度为 1，时钟信号。

2.rst，宽度为 1，复位信号

3. stall，宽度为 6，控制暂停信号

4.id_to_ex_bus，宽度为 169，ID 段传给 EX 段接口。

2.输出接口：

1.ex_to_mem_bus,宽度为 80，用于 EX 段传给 MEM 段接口

2. data_sram_en，宽度为 1，输出内存数据的读写使能信号

3.data_sram_wen，宽度为 4，输出内存数据的写使能信号

4.data_sram_addr，宽度为 32，输出内存数据存放地址

5.ex_to_id ，宽度为 4，用于 EX 段传给 ID 段接口

6.data_sram_wdata，宽度为 38，用于输出写入内存的数据

7.stallreq_from_ex ，宽度为 1，EX 用于发出是否暂停的接口

8.ex_is_load ，宽度为 1，用于 EX 段发给 ID 段的用来判断上一条指令是否是储存指令的接口

9.hilo_ex_to_id ，宽度为 66，用于将 EX 段将乘除法器结果发给 ID 段的 regfile 模块

在原有变量的基础上添加定义 data_ram_readen、inst_mthi、inst_mtlo、inst_multu、inst_mult、inst_divu、 inst_div 变量，将 ID 段传来的 id_to_ex_bus_r 中的值相应传给各个变量。

针对 inst[31:26]的值来判断当前指令：如果值为 6'b10_0011，则将 ex_is_load 赋值为 1；如果不是，就将 ex_is_load 赋值为 0。之后为 alu 模块准备输入信号，定义了有符号、无符号、sa 零扩展的信号，接下来使用条件选择语句 (sel_alu_src1, sel_alu_src2) 来选择 ALU 的输入源，然后便可以传入相应参数

调用 alu 模块并把结果赋值给 ex_rusult。在原本 ex_to_mem_bus 接口中添加 data_ram_readen 数据 RAM 的读使能信号。在传递给 ID 段的接口中定义寄存器文件的写使能信号、写地址以及 alu 计算结果三个信号。

设置内存读写使能 data_sram_wen：先判断当前指令是什么指令，若 data_ram_readen 为 4'b0101，则为 sb 指令，若为 4'b0111，则是 sh 指令。对于 sb 指令，进一步根据 ex_result 值进行判断。如果是 2'b00，就将内存写使能赋值为 4'b0001，表示要写入的是第一个字节，如果是 2'b01，就赋值为 4'b0010，表示要写入的是第二个字节，以此类推。对于 sh 指令，若 ex_result[1:0] 为 2'b00，将内存写使能赋值为 4'b0011，表示要写入 的是第一、第二字节，若为 2‘b10，则将内存写使能 赋值为 4'b1100，表示要写入的是第三、第四字节。将输出内存数据赋值为 alu 计算结果。

根据 data_sram_wen 的值决定要写入到内存的数据：若为 4'b1111，表明要将四个字节全部写入内存，即将 data_sram_wdata 赋值为 rf_rdata2。除此之外分别还有对 sb 指令写入最后一个字节、第三个字节、第二个字节、第一个字节，对 sh 指令写入后半 2 个字节、前半 2 个字节的情况。上述操作完成后，EX 段可以获取到想要获取的内存地址的数据。之后便可以将该段结果传递给其他段：将内存的读使能信号，当前的 Pc 值，内 存的读写使能，以及内存写使能信号，以及寄存器的写使能信号，以及寄存器 要写的地址与数据传给 MEM 段；将寄存器的写使能信号，以及寄存器要写的地址 与数据传给 ID 段，用来让 ID 段判断是否会出现相关的情况发生，还有乘除法器高位寄存 器以及低位寄存器的写使能信号，以及乘除法器高位和低位要写入的数据也传给 ID 段，让 ID 段在调用 regfile 的同时，将乘除法器高位和低位的值也一并写入寄存器中，

提高了 CPU 效率。之后的代码便是对乘除法高低位寄存器写使能和写入数据的定义操作。

在 EX 段我们加入了 MUL 和 DIV 模块，借此完成乘法和除法的指令。

需要特别说明， 的是，我们重写了 mul 模块，在 EX 段将原有乘法器替换为 32 周期移位乘法器。下面是对其的解释。

- **stallreq_for_mul**：请求流水线暂停的信号。当乘法单元忙碌时，可能需要暂停流水线以等待乘法结果。

- **mul_ready_i**：乘法单元是否准备好输出结果的信号。1 表示准备好，0 表示尚未准备好。

- **signed_mul_o**：一个标志，表示是否执行有符号乘法。1 表示有符号乘法，0 表示无符号乘法。

- **mul_opdata1_o** 和 **mul_opdata2_o**：待乘的数据，分别对应乘法的两个操作数。

- **mul_start_o**：指示是否启动乘法操作的信号。1 表示启动，0 表示停止。

- **mymul**：这是一个乘法单元模块，执行乘法操作。它有输入输出信号，接受两个操作数并计算结果。

- **mul_result**：乘法结果的输出信号。

- **mul_ready_i**：乘法结果是否已准备好的信号。

在状态机中：

如果 rst 信号为高（即系统复位），那么：

- stallreq_for_mul = NoStop;`：不会请求流水线暂停。
- 所有相关的寄存器（mul_opdata1_o、mul_opdata2_o、mul_start_o 和 signed_mul_o）都被清零或设置为默认值。

在系统不复位的情况下，首先为所有信号设定默认值，然后利用 {inst_mult,inst_multu}判断是否为有符号乘法，若为 2'b10，则是有符号乘法，

- 如果 mul_ready_i == MulResultNotReady`（即乘法单元的结果尚未准备好），那么：
  - 设置 mul_opdata1_o 和 mul_opdata2_o 为寄存器的值 rf_rdata1 和 rf_rdata2。
  - 设置 mul_start_o 为启动状态（MulStart）。
  - 设置 signed_mul_o 为 1，表示有符号乘法。
  - 请求流水线暂停（stallreq_for_mul = Stop;`），等待乘法计算完成。
- 如果 mul_ready_i == MulResultReady`（即乘法单元的结果已准备好），那么：
  - 设置 mul_opdata1_o 和 mul_opdata2_o 为寄存器的值。

- 设置 mul_start_o 为停止状态（MulStop）。
- 设置 signed_mul_o 为 1。
- 取消暂停流水线（stallreq_for_mul = NoStop;`），表示可以继续执行后续指令。

- 如果 mul_ready_i 的状态不是上述两种情况，则将所有相关信号设置为默认值。

如果为 2'b01，则是无符号乘法，与有符号乘法的处理逻辑类似，只不过 signed_mul_o 设置为 0，表示执行无符号乘法。如果都不是，则不启动乘法单元。这些值的操作完成后，传递给 mymul 模块中。

4.mymul.v

该模块是一个硬件乘法器，支持有符号和无符号整数的乘法运算，并且在时钟信号和复位信号控制下，能够完成乘法的启动、计算、结束等一系列状态切换。它通过逐步将乘数乘到被乘数的不同部分，从而完成乘法运算。

接口介绍：

输入接口：

1. rst：复位信号，当为高电平时，模块状态会被复位，运算结果清零。
2. clk：时钟信号，所有的操作都会在时钟的上升沿进行。
3. signed_mul_i：标志位，指示是否执行有符号乘法。如果为 1，则为有符号乘法；否则为无符号乘法。

4. a_o：被乘数，32 位输入，参与乘法运算的第一个操作数。

5. b_o：乘数，32 位输入，参与乘法运算的第二个操作数。

6. start_i：开始信号，当为高电平时，启动乘法运算。

输出接口：

1. result_o：乘法结果，64 位输出，保存最终的乘法结果。

2. ready_o：运算是否结束的标志信号，若运算完成，则为高电平；否则为低电平。

所用变量：

temp_opa 和 temp_opb：分别存储被乘数 a_o 和乘数 b_o，在有符号乘法时，对负数进行补码处理。

pv：64 位的中间变量，用于保存乘法的部分结果。

ap：64 位的中间操作数，初始时将被乘数 a_o 扩展为 64 位。

i：一个计数器，用来控制乘法步骤的进度，最大值为 32。

state：状态寄存器，控制乘法器的状态。它有 3 个状态：MulFree：空闲状态，等待启动信号。MulOn：乘法运算进行中。MulEnd：乘法运算结束，准备输出结果。

乘法器各个状态所进行的操作：

空闲状态 (MulFree):

a. 如果 start_i 为高电平，模块会开始乘法运算。

b. 在此状态下，模块首先会根据输入信号 signed_mul_i 判断是否为有符号运算。如果是有符号运算，并且输入数为负数，则会对其取补码。

c. 接着，将被乘数 a_o 扩展为 64 位，并初始化乘法的中间变量。

运算状态 (MulOn):

a.进入乘法状态后，通过逐位判断乘数 b_o，逐步进行乘法运算。具体方法是利用加法器将中间结果 pv 与当前的操作数 ap 相加，然后将 ap 左移一位，将乘数 b_o 右移一位，重复此过程直到乘法运算完成。

b.每执行一步，计数器 i 自增，最多执行 32 次，完成乘法。

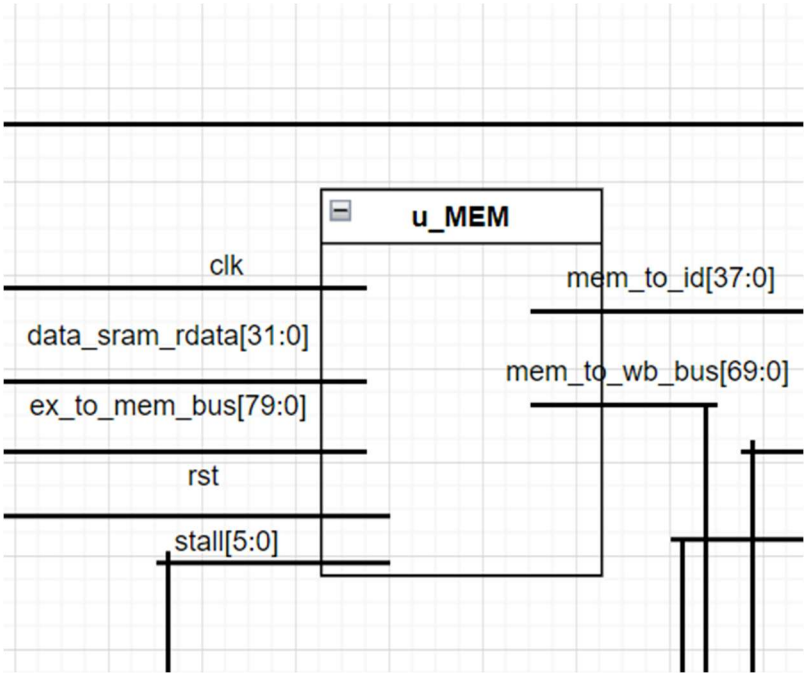
结束状态 (MulEnd):

a.当计数器 i 达到 32 时，乘法运算结束。如果是有符号乘法，且结果应为负数，则需要对结果进行取反加一（补码处理）。

b.乘法结果 pv 被输出到 result_o，并且 ready_o 被置为高电平，表示运算完成。

c.如果 start_i 为低电平，则模块回到 MulFree 状态，等待下一次的乘法请求。

5.MEM.v



输入端口：

1. clk（时钟信号）：用于同步时序。

2. rst（复位信号）：用于初始化模块状态，复位时将所有寄存器清零。

3. stall（暂停信号）：用于控制流水线的暂停，具体是控制在某些条件下是否暂停某些流水线阶段的数据传输。该信号是一个多位宽度的信号，不同的位表示不同的流水线暂停条件。

4. ex_to_mem_bus: 来自 EX 段的总线信号，包含了该阶段计算出的结果以及与内存访问相关的信息。

5. data_sram_rdata（来自数据存储器的读数据）：存储器从 SRAM 中读取的数据，用于传输回写阶段或用于生成新的数据。

输出端口：

1. mem_to_id（传递给 ID 阶段的数据）：从 MEM 阶段传递给指令解码阶段的数据，包括寄存器写使能、寄存器地址和写数据。

2. mem_to_wb_bus（传递给回写阶段的总线数据）：从 MEM 阶段传递给回写阶段的数据，包括内存地址、寄存器写使能、寄存器地址以及写回的数据。

在端口定义中，主要在原有接口的基础添加了 mem_to_id 输出接口传递寄存器相关信息。

ex_to_mem_bus_r 存储来自执行阶段的总线信号，并在时钟上升沿更新。它的值在复位时清零，在 stall 信号控制下也会清零或保持当前值，保证数据在流水线各阶段的正确传递。流水线暂停控制：当 stall[3] == Stop 且 stall[4] == NoStop 时，ex_to_mem_bus_r 被清零。否则，当 stall[3] == NoStop 时，ex_to_mem_bus_r 更新为 ex_to_mem_bus` 的值。

data_ram_en 控制内存是否启用。当内存使能信号为 1 时，内存可以进行操作。

data_ram_wen 表示内存的写使能信号。此信号的 4 位宽度表示四个字节的写操作。只有当 data_ram_wen 对应的字节位置为 1 时，才会对内存中的数据进行写操作。

data_ram_readen 控制从内存中读取的数据的位宽。可以通过其不同的位值来选择不同的读取方式，例如读一个字节、半字或字。

根据 data_ram_readen 和 data_ram_en 来决定如何从 SRAM 中读取数据。当 data_ram_readen==4'b1111，则为 lw 指令(lw 指令是将从内存中的值全 部写入相应寄存器)，直接传回 data_sram_rdata；当 data_ram_readen==4'b0001，则为 lb 指令（lb 指令是根据要写入寄存器的地址的最后两位判断要将从内存中读出来的数据的哪一个字节写入寄存器的最低一个字节，其他字节采用符号扩展的方式），根据 ex_result[1:0]的值来确定符号扩展方式，比如说等于 2'b00 时，最后 8 位用 data_sram_rdata 的前 8 位的值，其余采用 data_sram_rdata 的第 8 位的值扩展，依次类推；当 data_ram_readen==4'b0010 时，则为 lbu 指令（lbu 指令是根据要写入寄存器的地址的最后两位判断要将从内存中读出来的数据的哪一个字节写入寄存器的最低一个字节，其他字节采用零扩展的 方式），仍然是对 ex_result[1:0]的不同值采取不同操作，只是符号扩展变为零扩展；当 data_ram_readen==4'b0011 时，则为 lh 指令（lh 指令是根据要写入寄存器的地址的最后两位判断要将从内存中读出来的数据的哪两个字节写入寄存器的最低两个字节，其他字节采用符号扩展的方式），ex_result[1:0]==2'b00 时前 16 位用数据第 16 位扩展，后 16 位为数据前 16 位值，ex_result[1:0]==2'b10 时前 16 位用数据第 32 位扩展，后 16 位为数据后 16 位值；当 data_ram_readen==4'b0100 时，为 lhu 指令（lhu 指令是根据要写入寄存器的地址的最后两位判 断要将从内存中读出来的数据的哪两个字节写入寄存器的最低两个字节，其他 字节采用零扩展的方式），处理方式和 lh 指令相似，只是换为零扩展。

rf_wdata 是最终要写回的寄存器数据。如果是内存读操作且读使能信号有效，则从 data_sram_rdata 中获取数据并进行扩展后传送给寄存器写入端口。

如果是内存写操作，则不更新 rf_wdata，而是使用来自执行阶段的结果 ex_result。

MEM 到 WB 阶段数据传递:

mem_to_wb_bus 总线将 MEM 阶段的数据传递到 WB 阶段。它包含:

mem_pc: 当前指令的程序计数器值。

rf_we: 寄存器写使能信号，指示是否允许将数据写回寄存器文件。

rf_waddr: 要写回的寄存器地址。

rf_wdata: 要写回的数据。

MEM 到 ID 阶段数据传递:

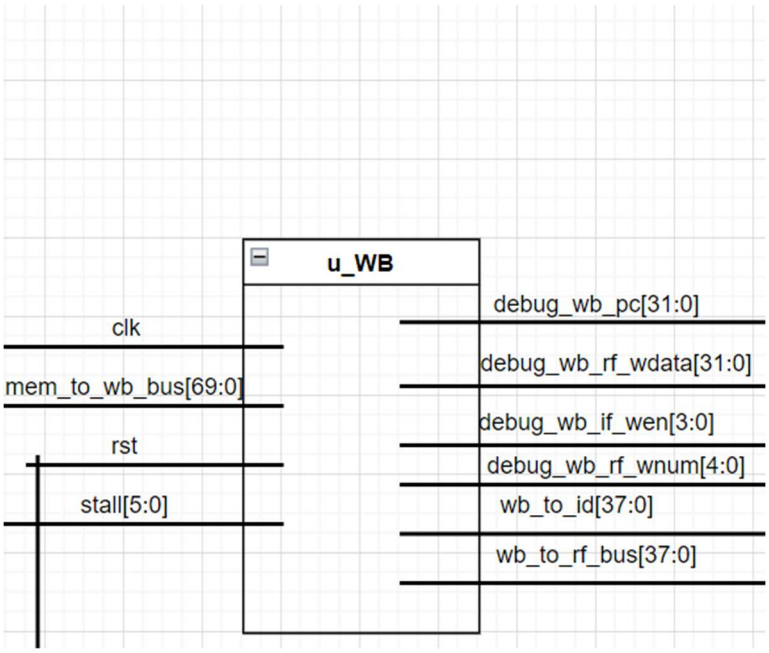
mem_to_id 总线将 MEM 阶段的一部分信息传递到 ID 阶段，供后续操作使用。它包含:

rf_we: 寄存器写使能信号。

rf_waddr: 寄存器写地址。

rf_wdata: 寄存器写数据。

6.WB.v



该模块为流水线中的回写阶段。它接收来自 MEM 阶段的数据，并将其送到 RF（寄存器文件），或者供调试使用。它还负责处理控制信号、寄存器的写使能、以及相关的数据传输。

输入端口：

1. **clk**（时钟信号）：用于同步时序。

2. **rst**（复位信号）：用于初始化模块状态，复位时将所有寄存器清零。

3. **stall**（流水线暂停信号）：该信号是一个多位信号，指示是否暂停流水线中的某些阶段，防止数据冲突或不一致的情况。此信号包含对流水线的多重控制，比如是否暂停某个阶段的数据传递。

4. **mem_to_wb_bus**（来自 MEM 阶段的数据总线）：这是来自 **MEM** 阶段的数据总线，包含了 MEM 阶段的各种信息（如程序计数器值、寄存器写使能、写寄存器地址以及数据），将这些数据传递到 **WB** 阶段。

输出端口：

1. **wb_to_rf_bus**（传递给寄存器文件的总线数据）：该信号用于传递回写数据到寄存器文件，包含：

    a. 寄存器写使能信号（rf_we）

    b. 寄存器写地址（rf_waddr）

    c. 寄存器写数据（rf_wdata）

2. **wb_to_id**（传递给 ID 阶段的总线数据）：该信号用于传递给 **ID**（指令解码）阶段，包含：

    a. 寄存器写使能信号（rf_we）

    b. 寄存器写地址（rf_waddr）

    c. 寄存器写数据（rf_wdata）

3. **debug_wb_pc**（调试信号：PC）：用于输出 **WB** 阶段的程序计数器值，便于调试时查看当前指令的位置。

4. **debug_wb_rf_wen**（调试信号：寄存器写使能）：用于输出寄存器写使能信号，便于调试。

5. **debug_wb_rf_wnum**（调试信号：寄存器写地址）：用于输出寄存器的写地址，便于调试时查看正在写入数据的寄存器。

6. **debug_wb_rf_wdata**（调试信号：寄存器写数据）：用于输出写回寄存器的数据，便于调试时查看具体的写回数据。

在原有的基础上只是添加了传输回 ID 段的输出端口，传递了寄存器写使能、写地址和写数据。

四、实验感受和改进意见

余敏波：

实验感受：通过这次实验，我们更清晰地了解到了 CPU 五级流水线的工作原理和流程。在完成实验的过程中，我们先是学习了新的编程语言 Verilog，然后是对各个模块进行添加、完善功能。对于我负责的部分，正确添加必要的信号、数据并设计其位数以及赋值至关重要，同时还要考虑到前面部分传来的信息，还需要往前面部分传回一些信息。这种环环相扣的结构需要缜密的设计和细致的理解。

张腾飞： 实验感受：本次实验重点研究如何制作简易 CPU，具体包括 IF.v,ID.v,EX.v,MEM.v,WB.v 等五个部分，且这五个部分相辅相成、循序渐进。除此之外还有一些其余文件如 defines.vh 等。补全一个文件前应该熟悉该文件作用再进行补全，能起到事半功倍的效果。

改进意见：可以把部分检查点的实现作为示例让学生跟着做，然后再实行自主设计，这样示例文件拿到手时不至于一头雾水、不知所措。也可以联系初始代码和 CPU 流水线给出一定的说明和写作方向。