成绩评定日期：

实验报告成绩：

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班

组长：郭一丁

组员：沈建辉 张森

报告日期：2025.1.4

目录

[1. 实验设计 3](#_Toc19256)

[1.1小组成员工作量划分 3](#_Toc7172)

[1.2 总体设计 3](#_Toc32491)

[1.3 运行环境及工具 5](#_Toc19652)

[2. 流水线各个阶段的说明 5](#_Toc23612)

[2.1 IF模块 5](#_Toc27838)

[2.2 ID模块 6](#_Toc16552)

[2.3 EX模块 11](#_Toc11788)

[2.4 MEM模块 13](#_Toc32299)

[2.5 WB模块 15](#_Toc2519)

[2.6 CTRL模块 16](#_Toc25572)

[2.7 HILO寄存器模块 18](#_Toc19197)

[3. 实验感受及建议 19](#_Toc12453)

[3.1 郭一丁部分 19](#_Toc32453)

[3.2 沈建辉部分 20](#_Toc26449)

[3.3 张森部分 20](#_Toc1109)

[4. 参考资料 21](#_Toc31488)

1. **实验设计**

**1.1小组成员工作量划分**

|  |  |  |
| --- | --- | --- |
| 姓名 | 任务分工 | 任务量占比 |
| 郭一丁 | 添加算术运算、数据移动、逻辑、跳转、访存指令，参与实现hilo寄存器、参与实现stall | 50% |
| 沈建辉 | 主要负责在流水线中添加stall相关指令，参与hilo的相关指令，参与实验报告的编写 | 25% |
| 张森 | 主要负责在流水线中添加hilo相关指令，参与实现stall的相关指令，参与实验报告的编写 | 25% |

**1.3 运行环境及工具**

运行环境：装有Vivado 的Linux服务器。 FPGA的Family为Artix 7，Package为fbg676，型号为xc7a200tfbg676-2。

编程工具：使用VSCode编写代码，使用Vivado 模拟仿真，使用git进行版本管理，使用GitHub搭建项目仓库。

**1.2 总体设计**

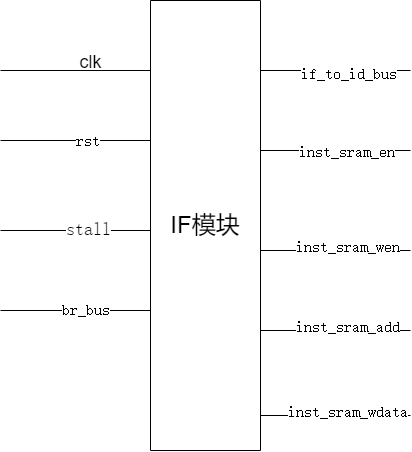
项目包括IF.v，ID.v，EX.v，MEM.v，WB.v，hi\_lo\_reg.v，mycpu\_core.v，mycpu\_top.v，这部分搭建了一条流水线的基本框架；及位于/lib目录下的alu.v，decoder\_2\_4.v，decoder\_5\_32.v，decoder\_6\_64.v，defines.vh，div.v，mmu.v，regfile.v，这部分构建了ALU和寄存器，定义了包含总线宽度信息在内的头文件；及位于/lib/mul目录下的add.v，fa.v，mul.v，这部分实现了乘法的运算。

图 1 CPU流水线示意图

1. **流水线各个阶段的说明**

**2.1 IF模块**

**整体说明：**取指令，控制指令延迟槽和跳转指令。接口如图所示。

**功能说明：**

IF段首先接收时钟信号和复位信号。当复位信号激活时，pc寄存器的值将被重置为其初始状态。随后，系统会检查暂停（stall）信号；若stall标志为高电平，则执行延迟槽机制，即下一指令的pc值将保持与当前pc值相同，以此实现暂停功能。

接下来，系统评估分支总线（br\_bus）的状态。如果检测到跳转请求，则从br\_bus中提取目标地址，并将其赋值给next\_pc变量，之后再将next\_pc的值传递给pc\_reg。如无跳转需求，pc\_reg则直接采用当前的next\_pc值，同时next\_pc自身递增4以指向下一个指令的位置。

最终，pc\_reg中的地址信息被发送至指令存储器，从中检索出对应于该地址的指令数据，并将此信息传递给ID段进行解码和进一步处理。

表 1 IF模块输入输出

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 暂停信号，控制指令是否暂停 |
| 4 | br\_bus | 33 | 输入 | 分支跳转信号，控制延迟槽是否跳转 |
| 5 | if\_to\_id\_bus | 33 | 输出 | IF段到ID段的数据总线 |
| 6 | inst\_sram\_en  inst\_sram\_en  \_en | 1 | 输出 | 读写使能信号 |
| 7 | inst\_sram\_wen | 4 | 输出 | 写使能信号 |
| 8 | inst\_sram\_addr | 32 | 输出 | 存放指令寄存器的地址 |
| 9 | inst\_sram\_wdata | 32 | 输出 | 存放指令寄存器的数据 |

**2.2 ID模块**

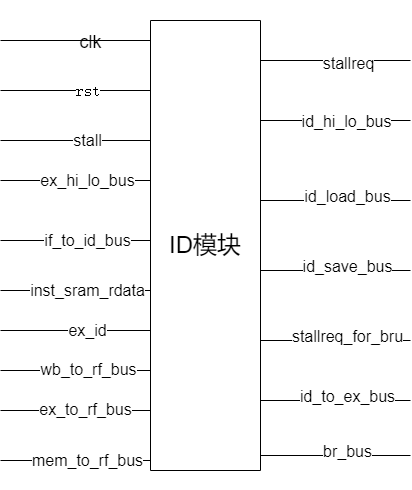
**整体说明：**

对指令进行译码，将结果传给EX段，实现寄存器读写，处理数据相关。接口下图所示。

表 2 ID模块输入输出

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 暂停信号，控制指令是否暂停 |
| 4 | stallreq | 1 | 输出 | 暂停请求信号 |
| 5 | if\_to\_id\_bus | 33 | 输入 | IF段到ID段的数据总线 |
| 6 | inst\_sram\_rdata | 1 | 输入 | 读写使能信号 |
| 7 | ex\_id | 1 | 输入 | 写使能信号 |
| 8 | wb\_to\_rf\_bus | 38 | 输入 | WB段存放进寄存器的数据 |
| 9 | ex\_to\_rf\_bus | 38 | 输入 | EX段存放进寄存器的数据 |
| 10 | mem\_to\_rf\_bus | 38 | 输入 | MEM段存放进寄存器的数据 |
| 11 | ex\_hi\_lo\_bus | 66 | 输入 | EX段存放进hilo寄存器的数据的总线 |
| 12 | id\_hi\_lo\_bus | 72 | 输出 | ID段存放进hilo寄存器的数据的总线 |
| 13 | id\_load\_bus | 5 | 输出 | ID段执行load命令的数据总线 |
| 14 | id\_save\_bus | 3 | 输出 | ID段执行save命令的数据总线 |
| 15 | stallreq\_for\_bru | 1 | 输出 | 执行load命令时的暂停请求 |
| 16 | id\_to\_ex\_bus | 159 | 输出 | ID段到EX段的数据总线 |
| 17 | br\_bus | 33 | 输出 | 分支跳转信号，控制延迟槽是否跳转 |

**功能说明：**

ID段的执行比较复杂，下面我们分成几部分来分别细述。

**第一部分 流水线暂停的判断与实现**

当ID段接收到由CTRL模块传递的stall信号后，它会根据该信号值来决定是否暂停流水线。若对应于ID段的stall信号为0，表示没有检测到流水线暂停的需求，此时IF段传送给ID段的数据（if\_to\_id\_bus）将被正常地复制到内部寄存器if\_to\_id\_bus\_r中，随后可以继续进行指令译码和操作数获取的过程。

然而，如果发现stall信号对于ID段而言为1，这表明当前存在存储访问冲突——即需要读取的寄存器数据尚未准备好，必须等到下一个时钟周期才能从内存中获得这些数据，且无法通过数据前递（forwarding）机制解决这一问题。在这种情况下，ID段的流水线将暂停一个周期，等待在下一时钟周期获取所需的寄存器值之后再行处理。一旦确定了暂停，if\_to\_id\_bus\_r会被设置为0以阻止本周期内的任何更新，并且整个ID段的操作将在下一周期恢复正常。

值得注意的是，if\_to\_id\_bus并不携带实际的指令信息；指令（inst）是在上一周期依据IF段提供的pc值从内存中直接读取的。因此，在发生暂停时，inst值不会被清零。为了确保指令流的连续性和正确性，ID段必须保存当前周期的inst值一个周期，使得在接下来的周期中能够使用当前周期的inst值，从而保持指令与其对应的pc值之间的同步和匹配。

**第二部分 指令的译码**

在处理一般指令时，首先根据指令中的操作码字段识别指令类型，并激活相应的inst\_\*\*变量来标识具体是哪条指令。译码完成后，依据指令中指定的源寄存器地址（rs和rt），即通过regfile模块读取位于inst[25:21]和inst[20:16]的通用寄存器，获取对应的rdata1和rdata2值。在此过程中，系统会检查是否存在数据相关性问题，并根据需要调整rdata1或rdata2的值。

同时，根据解析出的操作要求，为ALU配置合适的运算模式。每个ALU都有一个标志位，0表示该指令不使用特定的ALU，而1则表示使用；这些标志位组合成一个12位宽的alu\_op信号，用以表达16种可能的ALU操作之一，并作为指令进入执行（EX）阶段的一部分信息。

对于目的寄存器的写入控制，rf\_we信号决定了指令是否需要将结果写回通用寄存器。sel\_rf\_dst是一个三位的选择信号，其中：

sel\_rf\_dst[0]表示计算结果应写入由rd字段指定的通用寄存器；

sel\_rf\_dst[1]表示计算结果应写入由rt字段指定的通用寄存器；

sel\_rf\_dst[2]表示计算结果应写入31号通用寄存器（通常用于返回地址）。

rf\_waddr指定了实际要写入结果的通用寄存器地址。至于内存访问控制，data\_ram\_en信号用于指示指令是否涉及从内存读取或向内存写入数据；若指令包含此类操作，则data\_ram\_en设为1’b1。data\_ram\_wen是一个四位宽的信号，用于指定哪些字节需要写入内存——每一位对应一个字节，当需要写入特定字节时，相应位被设置为1。

对于跳转指令的处理，首先使用br\_e标志来标识当前指令是否为一条跳转指令。接下来，根据rdata1（由源寄存器rs提供的数据）和rdata2（由源寄存器rt提供的数据）之间的比较结果，设置几个条件标志：

rs\_ge\_z用于表示rdata1的值是否大于或等于0；

rs\_le\_z用于表示rdata1的值是否小于或等于0；

rs\_lt\_z用于表示rdata1的值是否严格小于0；

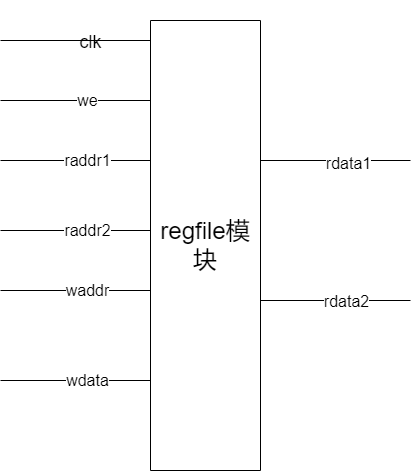
rs\_eq\_rt用于判断rdata1与rdata2的值是否相等。

这些条件标志将用于决定是否执行跳转以及跳转的目标地址。br\_addr则代表了计算后的跳转目标地址，具体的地址计算方法取决于指令类型。一旦确定了跳转逻辑和条件满足情况，计算出的新地址将被赋值给br\_addr，以此作为后续指令取指阶段的基础。

**第三部分 判断操作数来源**

通过使用sel\_alu\_src1和sel\_alu\_src2选择信号，系统可以确定ALU操作数的来源。对于第一个操作数，存在三种可能的来源；而对于第二个操作数，则有四种不同的来源。这些选择信号依据指令类型的不同来决定具体的操作数来源，确保每个指令能够正确地获取其所需的操作数。

在ID段完成译码后，会根据解析结果将相应的数据赋值给id\_to\_ex\_bus和br\_bus。其中，id\_to\_ex\_bus用于将ID段处理后的信息传递至执行（EX）阶段，而br\_bus则专门用于向IF段发送跳转指令相关的判断信号及目标地址。这意味着，当遇到跳转指令时，br\_bus不仅携带了是否应该执行跳转的决策信息，还包含了计算出的新目标地址，以便IF段据此更新程序计数器（PC）。这样，整个流水线就可以根据最新的跳转指令做出响应，确保指令流的正确性和连贯性。

regfile模块接口如图所示。

regfile模块说明：

regfile模块的主要功能是读取指定寄存器的值。它会根据提供的寄存器地址raddr1和raddr2来确定rs和rt寄存器的内容。具体来说：

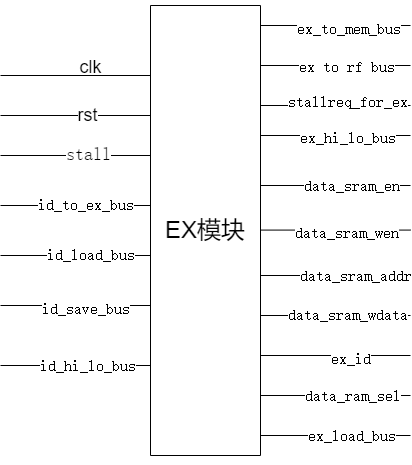
对于raddr1，模块首先检查其是否为零地址。如果是零地址，则将rdata1设置为32位全0（即32’b0），表示无效或空值；如果不是零地址，则从对应的寄存器中读取值，并将其赋给rdata1。

同样地，对于raddr2，也会进行类似的检查。如果raddr2为零地址，则将rdata2设置为32位全0；若不是零地址，则读取该地址对应寄存器中的值，并赋值给rdata2。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk  k | 1 | 输入 | 时钟信号 |
| 2 | we | 1 | 输入 | 寄存器的写使能信号 |
| 3 | raddr1 | 5 | 输入 | 读取的第一个数的地址 |
| 4 | raddr2 | 5 | 输入 | 读取的第二个数的地址 |
| 5 | rdata1 | 32 | 输出 | 读取的第一个数的值 |
| 6 | rdata2 | 32 | 输出 | 读取的第二个数的值 |
| 7 | waddr | 5 | 输入 | 写入的地址 |
| 8 | wdata | 32 | 输入 | 写入寄存器的值 |

通过这种方式，regfile确保了当尝试读取零号寄存器时总是返回零值，同时能够准确地从非零地址的寄存器中读取实际数据，为后续的指令执行提供正确的操作数。这种设计不仅简化了硬件逻辑，也符合许多RISC架构中关于零号寄存器总是硬连线为零的标准实践。

表 3 regfile模块输入输出

**2.3 EX模块**

**整体说明：**

在执行阶段，系统从ID/EX流水线寄存器中读取来自两个源的操作数：第一个操作数可以是来自寄存器1的值，第二个操作数则可能是来自寄存器2的值或是一个经过符号扩展的立即数。这些操作数随后被送入ALU以进行相应的算术或逻辑运算。例如，在加法指令的情况下，这两个操作数将在ALU中相加，产生的结果将被存储到EX/MEM流水线寄存器中，为后续步骤做准备。

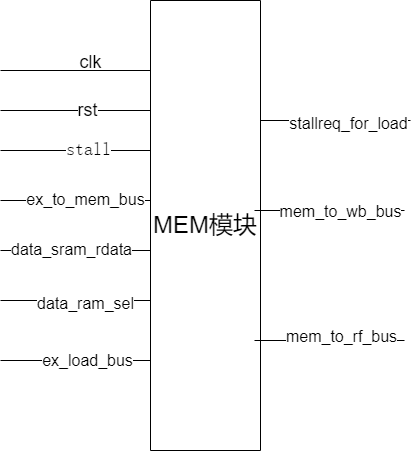
ALU模块已经设计完成，通过提供适当的控制信号，可以轻松地配置ALU来执行各种基本的逻辑和算术运算。对于那些需要访问内存的指令，在此阶段会发出访存请求，确保数据能够正确地从内存中读取或写入。

如图所示的接口用于展示不同组件之间的交互方式，包括如何传递操作数给ALU、接收ALU的结果以及如何发起访存请求。这一过程保证了指令执行的流畅性和准确性，同时维持了流水线架构的高效性。

表 4 EX模块输入输出

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 控制暂停信号 |
| 4 | id\_to\_ex\_bus | 169 | 输入 | ID 段传给 EX 段的数据 |
| 5 | id\_load\_bus | 5 | 输入 | ID段传递读的数据 |
| 6 | id\_save\_bus | 3 | 输入 | ID段传递写的数据 |
| 7 | ex\_to\_mem\_bus | 80 | 输出 | EX 段传给 MEM 段的数据 |
| 8 | ex\_to\_rf\_bus | 38 | 输出 | EX 段传给regfile段的数据 |
| 9 | id\_hi\_lo\_bus | 72 | 输入 | ID 段传给hilo段的数据 |
| 10 | ex\_hi\_lo\_bus | 66 | 输出 | EX 段传给hilo段的数据 |
| 11 | stallreq\_for\_ex | 1 | 输出 | 对EX段的stall请求 |
| 12 | data\_sram\_en | 1 | 输出 | 内存数据的读写使能信号 |
| 13 | data\_sram\_wen | 4 | 输出 | 内存数据的写使能信号 |
| 14 | data\_sram\_addr | 32 | 输出 | 内存数据存放的地址 |
| 15 | data\_sram\_wdata | 32 | 输出 | 要写入内存的数据 |
| 16 | ex\_id | 38 | 输出 | EX 段传给 ID 段的数据 |
| 17 | data\_ram\_sel | 4 | 输出 | 内存数据的选择信号 |
| 18 | ex\_load\_bus | 5 | 输出 | EX 段读取的数据 |

**功能说明：**

MEM模块，该模块有18个输入输出端口，包括clk，rst，stall，id\_to\_ex\_bus，id\_load\_bus，id\_save\_bus，id\_hi\_lo\_bus等7个输入端口和ex\_to\_mem\_bus，ex\_to\_rf\_bus，ex\_hi\_lo\_bus，stallreq\_for\_ex，data\_sram\_en，data\_sram\_wen，data\_sram\_addr，data\_sram\_wdata，ex\_id，data\_ram\_sel，ex\_load\_bus等11个输出端口，该模块还包含乘除法的部分实现。

**2.4 MEM模块**

**整体说明：**

在访问内存阶段，系统从EX/MEM流水线寄存器中获取地址信息，并据此从数据存储器中读取或写入数据。对于读操作，读取的数据将被存储到MEM/WB流水线寄存器中，以备后续的写回阶段使用。这一过程确保了数据能够正确地在存储器和流水线之间传递。

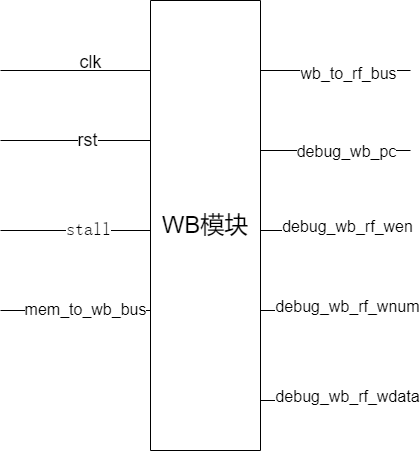
对于需要访问内存的指令，在此阶段会接收并处理访存的结果。具体来说，当执行读内存操作时，所读取的数据会被送入MEM/WB流水线寄存器；而对于写内存操作，则是在此阶段完成实际的写入动作。此外，系统还会根据指令要求选择是否将结果写回寄存器文件，以及确定写回的具体位置。

如上图所示的接口设计，清晰地展示了不同组件间的交互流程，包括地址和数据的传递、访存请求的发起及结果的处理。这不仅保证了指令执行的连贯性和准确性，也体现了高效流水线架构中各阶段之间的紧密协作。

表 5 MEM模块输入输出

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 控制暂停信号 |
| 4 | ex\_to\_mem\_bus | 80 | 输入 | EX 传给 MEM 段的数据 |
| 5 | data\_sram\_rdata | 32 | 输入 | 从内存中读出来要写入 寄存器的值 |
| 6 | data\_ram\_sel | 4 | 输入 | 内存数据的选择信号 |
| 7 | ex\_load\_bus | 5 | 输入 | EX 段读取的数据 |
| 8 | stallreq\_for\_load | 1 | 输出 | 对EX段的stall请求 |
| 9 | mem\_to\_wb\_bus | 70 | 输出 | MEM 传给 WB 段的数据 |
| 10 | mem\_to\_rf\_bus | 38 | 输出 | MEM段传给regfile段的数据 |

**功能说明：**

MEM模块是处理加载（load）和存储（store）指令的核心组件，具备10个输入输出端口。其输入端包括时钟信号（clk）、复位信号（rst）、流水线暂停信号（stall），以及来自执行阶段的数据总线（ex\_to\_mem\_bus）和加载数据总线（ex\_load\_bus）。该模块的输出端则负责将处理后的数据传递给写回阶段（mem\_to\_wb\_bus）和寄存器文件（mem\_to\_rf\_bus）。

在功能上，MEM模块能够执行多种类型的加载和存储指令，具体包括：

加载指令：支持lb（带符号字节加载）、lbu（无符号字节加载）、lh（带符号半字加载）、lhu（无符号半字加载）和lw（字加载）。对于这些指令，MEM模块会根据地址的最低两位来确定需要加载的字节数，并正确地对齐数据，然后将结果写入指定的寄存器文件（RF）寄存器中。

存储指令：支持sb（字节存储）、sh（半字存储）等。在执行这类指令时，MEM模块同样依据地址的最低两位来设置字节写使能信号，确保只更新内存中特定位置的数据，将数据准确写入数据RAM中。

通过这样的设计，MEM模块不仅实现了高效的内存访问，还保证了数据的正确性和完整性，同时与流水线架构中的其他阶段紧密协作，以维持整个系统的性能和稳定性。

**2.5 WB模块**

**整体说明：**

将结果写回寄存器,从MEM/WB流水线寄存器中读取数据并将它写回图中部的寄存器堆中。接口如右图所示。

表 6 WB模块输入输出

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 控制暂停信号 |
| 4 | mem\_to\_wb\_bus | 70 | 输入 | MEM 传给 WB 的数据 |
| 5 | wb\_to\_rf\_bus | 38 | 输出 | WB 传给 rf 的数据 |
| 6 | debug\_wb\_pc | 32 | 输出 | 用来 debug 的 pc 值 |
| 7 | debug\_wb\_rf\_wen | 4 | 输出 | 用来 debug 的写使能信号 |
| 8 | debug\_wb\_rf\_wnum | 5 | 输出 | 用来 debug 的写寄存器地址 |
| 9 | debug\_wb\_rf\_wdata | 32 | 输出 | 用来 debug 的写寄存器数据 |

**功能说明：**

WB（写回）模块是流水线架构中的最后一个阶段，拥有9个输入输出端口，其中包括时钟信号（clk）、复位信号（rst）、暂停信号（stall），以及来自MEM/WB流水线寄存器的数据总线（mem\_to\_wb\_bus）。此外，还提供调试信号用于开发和故障排查。该模块的输出包括用于将数据写回到寄存器堆的数据总线（wb\_to\_rf\_bus），以及用于调试目的的额外信号。

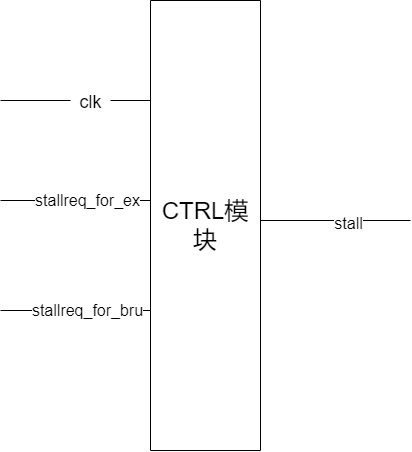
WB模块的主要职责是从MEM/WB流水线寄存器中读取处理完成的数据，并将其写回到寄存器堆中，从而完成指令的最终结果更新。具体流程如下：

数据接收：定义一个名为mem\_to\_wb\_bus\_r的内部寄存器，用于暂存从MEM/WB流水线寄存器中接收到的数据。这一步骤确保了即使在流水线暂停的情况下，数据也不会丢失。

数据写回：一旦确定可以继续执行（即没有流水线暂停等情况），WB模块会将存储在mem\_to\_wb\_bus\_r中的数据写回到指定的寄存器堆位置。这一步骤实现了指令执行结果的实际保存。

输出更新：最后，通过wb\_to\_rf\_bus将写回的数据输出，以更新寄存器堆中的相应寄存器值。这样，后续指令就可以使用最新更新的数据进行计算。

通过这种方式，WB模块不仅完成了对指令执行结果的最终确认和保存，还确保了整个流水线架构的连贯性和高效性，同时为系统提供了必要的调试支持。

**2.6 CTRL模块**

**整体说明：**

接收各段传递过来的流水线请求信号，从而控制流水线各阶段的运行。接口如右图所示。

表 7 CTRL模块输入输出

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | stallreq\_for\_ex | 1 | 输入 | 执行阶段的指令是否请求流水线暂停 |
| 3 | stallreq\_for\_bru | 5 | 输入 | Load命令是否请求流水线暂停 |
| 4 | stall | 6 | 输出 | 暂停信号 |

**功能说明：**

当流水线中位于第n阶段的指令需要多个周期来完成，并因此请求暂停流水线时，系统必须采取措施确保处理的正确性和连续性。具体而言：

保持PC不变：为了防止新的指令被取出，程序计数器（PC）应保持指向当前正在执行的指令地址，不进行更新。

维持寄存器状态：第n阶段及其之前的所有阶段的寄存器内容也应保持不变，以确保这些阶段中的指令状态不会因流水线暂停而丢失或改变。

后续阶段继续运行：与此同时，第n阶段之后的流水线阶段可以继续正常运行，从而充分利用硬件资源，减少不必要的等待时间。

为了实现上述控制逻辑，使用了一组名为stall的信号来指示各个流水线阶段是否应该暂停：

stall[0]为1表示没有任何阶段暂停，流水线正常运作；

stall[1]至stall[5]分别对应IF、ID、EX、MEM和WB五个流水线阶段的暂停控制。若某一位为1，则表示相应阶段被暂停，其操作将延迟到下一时钟周期。

这种机制允许精确控制流水线中不同阶段的行为，确保在多周期指令处理期间维持系统的稳定性和指令执行的准确性，同时尽可能减少对整体性能的影响。通过这种方式，即使某些指令需要额外的时间来完成，流水线仍能高效地处理其他指令，保持良好的吞吐率。

**2.7 HILO寄存器模块**

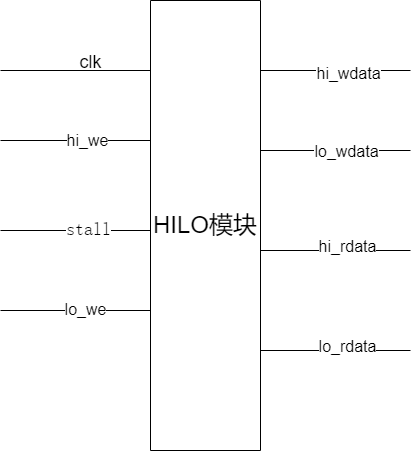
**整体说明：**

HI和LO寄存器作为协处理器的一部分，独立于通用寄存器之外，主要用于处理乘法和除法运算的结果。以乘法为例，当两个整数相乘时，乘法结果的低位部分会存储在LO寄存器中，而高位部分则存储在HI寄存器中。

这两个寄存器支持独立的读取和写入操作：

读取：使用mfhi指令可以从HI寄存器读取值，使用mflo指令可以从LO寄存器读取值。值得注意的是，与通用寄存器不同，mfhi和mflo指令是在执行（EX）阶段就开始从HI和LO寄存器获取数值的。

写入：通过mthi指令可以向HI寄存器写入值，通过mtlo指令可以向LO寄存器写入值。这些写入操作与其他通用寄存器类似，在写回（WB）阶段完成。

这种设计使得HI和LO寄存器能够专门用于保存乘法和除法运算产生的较大结果，同时保持了对它们的访问灵活性，允许程序员根据需要直接操作这些特殊寄存器。通过将读取操作置于执行阶段，并将写入操作安排在写回阶段，确保了流水线架构中的高效数据处理流程。

可以直接改写lib下的regfile.v，也可以添加hiloreg.v，创建u\_hi\_lo\_reg，但是MEM、WB也要跟着改，这里我们采用第二种方法，即添加hiloreg.v文件。接口如右图所示。

表 8 HILO寄存器输入输出

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | stall | 6 | 输入 | 控制暂停信号 |
| 3 | hi\_we | 1 | 输入 | hi寄存器的写使能信号 |
| 4 | lo\_we | 1 | 输入 | lo寄存器的写使能信号 |
| 5 | hi\_wdata | 32 | 输出 | Hi寄存器写的数据 |
| 6 | lo\_wdata | 32 | 输出 | Lo寄存器写的数据 |
| 7 | hi\_rdata | 32 | 输出 | Hi寄存器读的数据 |
| 8 | lo\_rdata | 32 | 输出 | Lo寄存器读的数据 |

**功能说明：**

当控制信号hi\_we和lo\_we均为1时，寄存器reg\_hi和reg\_lo会同时将hi\_wdata和lo\_wdata写入各自的位置。具体来说：

如果hi\_we为0且lo\_we为1，则只有reg\_lo会接收并写入lo\_wdata。

如果hi\_we为1且lo\_we为0，则只有reg\_hi会接收并写入hi\_wdata。

此外，hi\_rdata和lo\_rdata分别用于输出reg\_hi和reg\_lo中的数据值，允许独立访问这两个寄存器的内容。

这种设计确保了HI和LO寄存器能够根据需要灵活地进行单独或同时的数据写入操作，同时也提供了简便的数据读取机制，以支持复杂的算术运算结果的处理。通过精确控制写入使能信号（hi\_we和lo\_we），系统可以在乘法、除法等指令执行过程中有效地管理HI和LO寄存器的状态更新。

。

1. **实验感受及建议**

**3.1 郭一丁部分**

在实验过程中，我熟练掌握了GitHub的使用技巧，能够高效地创建仓库、审阅代码并管理版本，这显著提升了我们团队的工作效率。

在调试阶段，我们通过在波形图中添加可能存在问题的信号来辅助定位故障。特别地，我们关注程序计数器（PC）附近的信号波形，这种方法帮助我精准地确定了错误的位置及其原因，从而有效地解决了问题。

这种结合版本控制和细致调试的方法，不仅增强了我对工具和技术的理解，还极大地提高了问题解决的效率和质量。

**3.2 沈建辉部分**

通过本次实验，我深入掌握了流水线的整体运行机制，将课堂上学习的理论知识成功应用于实践之中。尽管我在团队中负责的任务相对较少，但为了确保任务的顺利完成，我还是全面理解了整个代码库的运行逻辑，深入了解了流水线每个阶段的具体运作方式。这使我能够准确地在适当的位置插入相关指令，并针对遇到的诸多问题，通过查阅网络资料找到了解决方案。

这次实验也深刻体现了团队合作的重要性。为了确保项目的成功，我们明确了各自的分工，并保持了与队友之间的频繁交流。这种协作不仅提高了工作效率，还充分发挥了团队的价值，使我们能够共同克服挑战。

总而言之，这次实验不仅让我掌握了一种新的编程方法，更让我对流水线的工作原理及其细节有了更为深入的理解。同时，它还让我深切体会到了团队合作的力量，认识到明确分工和积极沟通对于项目成功的至关重要性。这段经历极大地丰富了我的技术技能和团队协作经验。

**3.3 张森部分**

此次实验不仅在平台和内容上与以往截然不同，其考核方式也独具匠心，这些新颖的元素促使我掌握了众多新技能和方法：通过使用GitHub进行代码管理，小组成员间的代码同步变得既快速又便捷，大大提高了团队的工作效率；借助CG实验平台，我们避免了繁琐的软件安装和调试过程，能够更专注于实验本身，提升了实验的流畅度；而在debug过程中，我进一步熟悉并运用了CPU五级流水线的概念，这加深了我对计算机系统底层运作机制的理解。

实验期间，我们小组分工明确、配合默契，每位成员都积极贡献自己的力量，这种高效的团队合作让我深刻体会到协作的重要性，这也是我经历过的最为顺畅且富有成效的团队项目之一，大家互帮互助，共同克服了一个个挑战。

总的来说，这是一次令人愉悦且充满收获的实验经历，不仅体验到了团队合作的乐趣，还感受到了实验设计者的用心良苦。从创新的实验形式到指导老师和助教们的认真负责，每一个细节都彰显出对教学质量的高度重视。这次计算机系统实验无疑为我们的学习旅程增添了一抹亮色，相信未来我们将更加出色，并在不断的学习中收获更多宝贵的经验。