实验报告成绩：	成绩评定日期：

2025～2026 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 2301 班、2302 班

组长：李天园

组员：李芊芊 楚可欣

报告日期：2025.12.28

目录

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
李天园	主要负责 IF 与 ID 模块，构思总体架构，实验报告的编写	35%
李芊芊	MEM 模块、WB 模块、CTRL 模块代码及报告书写，GitHub 各分支合并管理	35%
楚可欣	EX 模块，HI_LO_REG 模块代码实现以及对应部分报告的编写	30%

1.2 总体设计框架

```
mycpu_core
  ├─ IF  <-- br_bus --------- ID
  │     ├─ inst_sram (addr=pc_reg) -> inst_sram_rdata -> IF -> if_to_id_bus -> ID
  │     └─ if_to_id_bus -> ID
  ├─ ID  <-- wb_to_rf_bus -- WB
  │     ├─ if_to_id_bus -> ID -> id_to_ex_bus -> EX
  │     ├─ br_bus -> IF (分支跳转)
  │     └─ stallreq_for_bru -> CTRL
  ├─ EX
  │     ├─ id_to_ex_bus -> EX -> ex_to_mem_bus -> MEM
  │     ├─ ex_to_rf_bus -> (forward -> ID / 写回路径)
  │     ├─ data_sram (addr/wdata/wen) <- EX (访存请求)
  │     ├─ 使用 alu / div / mul -> 产生 stallreq_for_ex
  │     └─ ex_load_bus/ex_save_bus -> MEM
  ├─ MEM
  │     ├─ ex_to_mem_bus_r -> MEM -> mem_to_wb_bus -> WB
  │     ├─ 从 data_sram 读取 data_sram_rdata -> 处理 load -> mem_result
  │     └─ 若 load 需等待 -> stallreq_for_load -> CTRL
  ├─ WB
  │     ├─ mem_to_wb_bus -> WB -> 写回寄存器 (regfile)
  │     └─ 产生 wb_to_rf_bus -> 回馈 ID（写回转发）
  ├─ hi_lo_reg  <-> EX/WB（用于 mul/div 的 HI/LO 保存与读写）
  └─ CTRL
        ├─ 接收 stallreq_for_ex / stallreq_for_bru / stallreq_for_load
        └─ 生成 stall[`StallBus`] -> IF/ID/EX/MEM/WB 控制停顿

其他库模块：
  ├─ regfile  <- 写回(WB) / 读出(ID) / 写回信号来自 wb_to_rf_bus/ex_to_rf_bus/mem_to_rf_bus
  ├─ alu / div / mul  (被 EX 使用)
  └─ mmu / decoder / defines.vh 等为工程支持文件
```

1.3 运行环境及工具

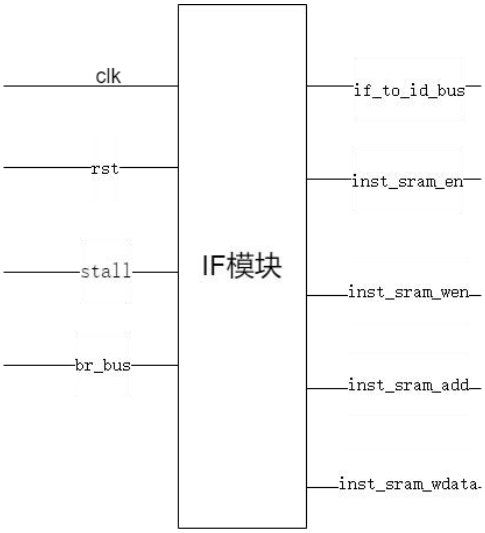
由于提供的 CG 环境较卡，并且保存记录、查看输出调试等不太方便，最终在本地 使用 vivado HLS 2019.2 在本地进行仿真运行。

使用 github 创建仓库进行管理，并使用 gitkraken、github desktop 等可视化工具来 拉取、上传代码等。

2. 流水线各阶段说明

2.1 IF 模块

IF 是 CPU 流水线中取指阶段的核心模块，核心职责是根据程序计数器（PC）地址从指令存储器中读取指令，计算下一条指令的 PC 地址，并将取指相关信息（指令存储器使能、当前 PC 值）传递给解码阶段（ID），同时响应分支跳转信号和流水线停顿控制，保障指令读取的连续性与正确性。



核心流程：基于当前 PC 地址读取指令 → 计算下一条指令的 PC 地址 → 向 ID 阶段传递取指状态与 PC 信息；

关键特性：支持流水线停顿控制（响应 stall 信号）、分支跳转地址更新（接收 br_bus 信号），仅负责指令读取（不涉及指令写入，inst_sram_wen 恒为 0）。

1. 输入信号与输出信号

信号名	位宽	功能描述
clk	1 位	同步时钟信号，驱动模块内部时序逻辑（PC 寄存器、使能寄存器更新）。
rst	1 位	高电平有效复位信号，初始化 PC 寄存器和指令存储器使能信号。
stall	StallBus-1:0	流水线停顿信号，stall[0] 对应 IF 阶段：stall[0]==NoStop` 时允许更新，否则保持当前状态。
br_bus	BR_WD-1:0	来自分支预测/分支执行模块的分支信息总线，包含分支有效标志和分支目标地址。

信号名	位宽	功能描述
if_to_id_bus	IF_TO_ID_WD-1:0	传递给 ID 阶段的总线，包含指令存储器使能信号（ce_reg）和当前 PC 值（pc_reg）。
inst_sram_en	1 位	指令存储器使能信号，ce_reg 直接映射，控制指令存储器是否启动读取。
inst_sram_wen	4 位	指令存储器写使能信号，恒为 4'b0，IF 阶段仅读不写指令存储器。
inst_sram_addr	32 位	指令存储器读取地址，直接映射当前 PC 寄存器值（pc_reg）。
inst_sram_wdata	32 位	指令存储器写数据，恒为 32'b0，无指令写入需求。

2.程序计数器（PC）的更新

程序计数器（pc_reg）是 IF 模块的核心寄存器，存储当前正在读取指令的地址，其更新逻辑严格遵循“复位优先、停顿保持、正常更新”的原则

复位逻辑：当 rst 为高电平时，pc_reg 被初始化为固定地址 32'hbfbf_fffc，该地址为程序预设的起始执行地址，确保程序从指定入口启动。

正常更新逻辑：在每个时钟上升沿，若检测到无流水线停顿（stall[0] == NoStop），则 pc_reg 被更新为预计算的下一条指令地址（next_pc`），实现指令的连续读取。

停顿保持逻辑：若检测到流水线停顿信号（stall[0] 为停顿状态），则 pc_reg 保持当前值不变，避免取指流程错乱，确保流水线各阶段协同一致。

3. 指令存储器使能信号的管理

指令存储器使能信号（ce_reg）用于控制指令存储器的启动与关闭，其更新逻辑与 PC 寄存器协同联动：

复位逻辑：当 rst 为高电平时，ce_reg 被置为低电平（1'b0），此时指令存储器处于禁用状态，不进行任何指令读取操作。

正常更新逻辑：在每个时钟上升沿，若检测到无流水线停顿（stall[0] == NoStop），则 ce_reg 被置为高电平（1'b1`），启用指令存储器，配合 PC 地址完成指令读取。

停顿保持逻辑：若检测到流水线停顿信号，ce_reg 保持当前状态（通常为高电平，确保停顿结束后可快速恢复取指），避免频繁切换使能状态导致的时序问题。

4.下一个程序计数器（next_pc）的计算

next_pc 是下一条指令的地址预计算结果，其值由分支状态决定，实现“顺序执行”与“分支跳转”的动态切换：

分支跳转场景：当分支有效标志 br_e 为高电平时，表示当前需执行分支跳转，next_pc 直接赋值为分支目标地址 br_addr，实现指令跳转到指定地址。

顺序执行场景：当 br_e 为低电平时，表示无分支跳转需求，next_pc 为当前 PC 值加 4（pc_reg + 32'h4），因 CPU 指令通常占 4 字节，该计算确保指令按顺序连续读取。

5. 指令存储器接口信号的分配

IF 阶段仅负责从指令存储器读取指令，不涉及指令写入操作，因此相关接口信号按“只读”需求固定配置或动态映射：

使能信号 inst_sram_en：直接映射 ce_reg 的值，ce_reg 为高电平时，inst_sram_en 有效，指令存储器启动读取；反之则禁用。

写使能信号 inst_sram_wen：固定配置为 4'b0，明确 IF 阶段不执行指令存储器的写操作，避免误写指令数据。

指令存储器地址 inst_sram_addr：直接映射当前 PC 寄存器值 pc_reg，指向即将读取的指令在存储器中的物理地址。

指令存储器写数据 inst_sram_wdata：固定配置为 32'b0，因无写操作需求，无需传递有效写数据。

6. 数据总线的打包与传递

IF 模块通过 if_to_id_bus 总线将取指关键信息传递给 ID 阶段，实现两阶段的数据协同：

总线构成：if_to_id_bus 由 ce_reg（指令存储器使能信号）和 pc_reg（当前 PC 值）打包而成，高位为 ce_reg，低位为 pc_reg，总线位宽由宏定义 IF_TO_ID_WD 统一规定。

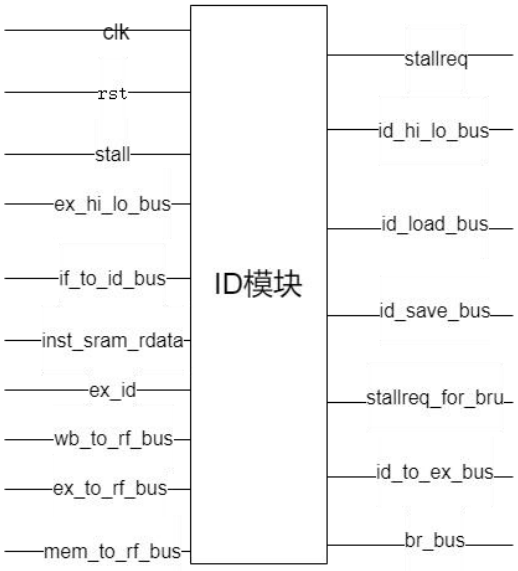
传递目的：ce_reg 用于告知 ID 阶段当前取指操作是否有效，pc_reg 为 ID 阶段解析指令、计算分支地址、处理指令时序提供核心地址参考。

2.2 ID 模块

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	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：

（一）基础配置与信号接口

1. 配置引入与输入输出信号定义

引入宏定义头文件 lib/defines.vh，统一复用 Stop/NoStop、寄存器地址、指令编码等定义，提升代码可维护性；

输入输出信号定义

① 时序控制信号：clk（同步时钟，驱动内部时序逻辑）、rst（高电平有效复位，初始化模块状态）；

② 流水线交互信号：stall（流水线停顿控制，stall[1] 控制本阶段暂停）、if_to_id_bus（承接 IF 阶段的使能信号与 PC 值）、ex_id（EX 阶段标识信号，支撑数据转发）；

③ 跨阶段数据总线：wb_to_rf_bus/ex_to_rf_bus/mem_to_rf_bus（分别接收 WB/EX/MEM 阶段到寄存器文件的写回数据）、ex_hi_lo_bus（接收 EX 阶段 HI/LO 寄存器相关信号）；

④ 输出专用总线：id_hi_lo_bus（传递 HI/LO 信号到 EX 阶段）、id_load_bus/id_save_bus（传递访存指令类型信号）、br_bus（传递分支跳转相关信号）。

2. 内部核心组件

暂存寄存器：if_to_id_bus_r（缓存 IF 阶段数据总线）、pc_reg（程序计数器）、ce_reg（使能信号寄存器）、flag/buf_inst（停顿期间指令缓存与状态标志）；

功能模块实例化：regfile（寄存器文件，读取 rs/rt 源操作数）、hi_lo_reg（HI/LO 专用寄存器，处理乘法 / 除法 64 位结果的读写）。

(二) 核心功能处理

1. 指令译码

字段拆分：按指令集规范（如 MIPS），从 32 位指令中提取操作码（opcode）、功能码（func）、寄存器地址（rs/rt/rd）、移位量（sa）、立即数（imm）、跳转索引（instr_index）等功能字段；

编码转换：通过 6-64 译码器将 opcode/func 转为 64 位独热码（op_d/func_d），通过 5-32 译码器将 rs/rt/rd/sa 转为 32 位独热码（rs_d/rt_d 等），降低硬件判断复杂度；

指令识别：通过独热码逻辑与运算，结合字段约束（如 mult 指令约束 rd=0/sa=0），精准判定指令类型，生成 inst_add/inst_lw/inst_j 等单比特指令标识信号，覆盖算术运算、逻辑运算、移位、分支跳转、数据移动、访存等所有指令类别。

2. 数据准备与相关性处理

原始数据读取：通过 regfile 模块，按 rs/rt 地址读出原始操作数 rdata1/rdata2；

数据转发修正：比较 rs/rt 地址与 EX/MEM/WB 阶段的写回地址，选择最新数据（ex_rf_wdata/mem_rf_wdata/wb_rf_wdata）转发，生成修正后的运算数据 ndata1/ndata2，解决流水线数据冒险；

HI/LO 数据处理：通过 hi_lo_reg 模块，接收 EX 阶段的 hi_we/lo_we（写使能）与 hi_wdata/lo_wdata（写数据），提供 HI/LO 寄存器读写接口，适配乘法 / 除法指令的 64 位结果存储需求；

寄存器写使能控制：根据指令类型，仅对需保存结果的指令（如加法、读内存、跳转保存返回地址）置位 rf_we，避免无效写操作。

3. 控制信号生成

ALU 控制：生成 sel_alu_src1/sel_alu_src2（定义 ALU 运算数来源：寄存器数据 / 立即数 / PC / 移位量）、alu_op（12 位操作码，映射加法 / 减法 / 移位 / 逻辑运算等 ALU 执行动作）；

访存控制：生成 data_ram_en（数据 RAM 使能，仅访存指令置 1）、data_ram_wen（数据 RAM 写使能，仅写内存指令置 1）；

寄存器控制：生成 rf_waddr（寄存器写地址，按指令类型选择 rd/rt/31 号寄存器）、sel_rf_res（写数据来源选择：ALU 结果 / RAM 读结果）；

分支跳转控制：计算 pc_plus_4（下一条指令地址），基于 rs_eq_rt/rs_lt_z 等条件判断是否满足跳转要求，生成分支使能 br_e 与跳转地址 br_addr。

4. 流水线风险控制

停顿请求生成：当 EX 阶段需写回的寄存器与当前 ID 阶段的 rs/rt 重合时，发出 stallreq_for_bru 停顿请求，让流水线暂停一拍，待数据写回后再继续执行，避免数据相关性错误，与数据转发形成互补，覆盖所有数据冒险场景。

（三）输出打包与流水线协同

1. 核心数据总线打包

id_to_ex_bus：整合当前 PC 值、原始指令、alu_op、sel_alu_src1/sel_alu_src2、data_ram_en/data_ram_wen、rf_we/rf_waddr/sel_rf_res 及 ndata1/ndata2，一次性传递给 EX 阶段，简化模块间信号连接；

专用总线打包：id_hi_lo_bus（HI/LO 寄存器相关信号）、id_load_bus（读内存指令类型）、id_save_bus（写内存指令类型），适配 EX/MEM 阶段特殊处理需求。

2. 分支跳转信号打包

br_bus：整合分支条件满足标志 br_e 与跳转地址 br_addr，传递给 IF 阶段，用于更新 PC（跳转时取 br_addr，否则取 pc_plus_4），保障分支指令的正确执行。

2.3 EX 模块

设计说明

EX 模块是五级流水 CPU 中的执行阶段，主要负责算术逻辑运算、访存地址计算以及相关控制信号的生成。在最初实现中，EX 模块完成了基本的 ALU 运算、Load/Store 地址计算以及乘除法操作，但控制逻辑较为集中，功能耦合度较高，不利于后续扩展和维护。因此，在此基础上对 EX 模块进行了进一步完善和优化。

在改进后的设计中，EX 模块将指令类型的译码工作尽量前移到 ID 阶段，通过多条总线分别传递运算控制、Load/Store 类型以及 HI/LO 相关信息，并在 EX 阶段统一打一拍缓存。EX 阶段只根据这些控制信号完成具体执行操作，使模块职责更加清晰。同时，通过对 ALU 运算结果与 HI/LO 寄存器读取结果的选择，实现了对 mfhi、mflo 等指令的正确支持。

针对访存指令，EX 模块采用 ALU 计算访存地址，并利用地址低两位生成字节选择信号，从而支持字节、半字和字级别的精细访存控制。Store 指令的写使能由指令类型和字节选择共同决定，写数据根据访问宽度进行复制处理，保证了 sb、sh、sw 等指令的正确行为。同时，EX 模块将相关控制信息传递到 MEM 阶段，避免在后续阶段重复译码。

在乘法和除法指令的支持方面，EX 模块分别实例化乘法器和除法器。由于除法操作为多周期运算，当除法结果尚未准备好时，EX 阶段会主动向流水线控制单元发出暂停请求，阻止后续指令继续推进，从而保证运算结果的正确性。除法完成后，结果通过 HI/LO 寄存器写回，流水线恢复正常运行。

模块接口

类别	信号名称	位宽	方向	描述
时钟复位	clk	1	输入	系统时钟
	rst	1	输入	复位信号，高有效
流水线控制	stall	StallBus	输入	流水线停顿控制
	stallreq_for_ex	1	输出	EX 阶段停顿请求

类别	信号名称	位宽	方向	描述
数据输入	id_to_ex_bus	ID_TO_EX_WD	输入	ID→EX 数据总线
	id_load_bus	LoadBus	输入	加载指令类型
	id_save_bus	SaveBus	输入	存储指令类型
	id_hi_lo_bus	72	输入	HI/LO 寄存器信号
数据输出	ex_to_mem_bus	EX_TO_MEM_WD	输出	EX→MEM 数据总线
	ex_to_rf_bus	EX_TO_RF_WD	输出	寄存器文件写回数据
	ex_hi_lo_bus	66	输出	HI/LO 操作结果
存储器接口	data_sram_en	1	输出	数据 SRAM 使能
	data_sram_wen	4	输出	数据 SRAM 写使能
	data_sram_addr	32	输出	数据 SRAM 地址
	data_sram_wdata	32	输出	数据 SRAM 写数据
控制信号	ex_id	1	输出	EX 阶段标识
	data_ram_sel	4	输出	数据 RAM 选择
	ex_load_bus	LoadBus	输出	EX 阶段加载类型

信号处理流程

ID 阶段传来的操作数、控制信号和指令信息通过 id_to_ex_bus 总线进入 EX 模块，在时钟上升沿被流水线寄存器捕获并暂存，为执行阶段做准备。然后从流水线寄存器中取出源操作数 1 和 2，根据选择信号 sel_alu_src1 和 sel_alu_src2 决定操作数来源（PC、立即数或寄存器值），立即数根据指令类型进行符号扩展或零扩展。扩展后的操作数送入 ALU 单元，根据 alu_op 控制信号执行指定的算术或逻辑运算，产生 32 位结果，这是大多数计算指令的最终结果。如果是乘除指令，操作数被送入乘法器或除法器。乘法单周期完成，产生 64 位结果；除法多周期执行，期间发出停顿请求 stallreq_for_ex，暂停流水线直到计算完成。

根据指令类型选择最终输出结果：MFHI/MFLO 指令选择 HI/LO 寄存器值，其他指令选择 ALU 结果。结果通过 ex_to_mem_bus 传递到下一阶段，同时通过 ex_to_rf_bus 支持数据前递。对于访存指令，ALU 计算的地址作为 data_sram_addr 输出，同时根据指令类型生成字节选择 data_ram_sel、写使能 data_sram_wen 等控制信号，准备下一阶段的内存访问。如果是 MTHI/MTLO 指令，寄存器值写入 HI/LO；如果是乘除指令，运算结果的高低位分别写入 HI 和 LO，更新信号通过 ex_hi_lo_bus 输出。

整个过程在单个时钟周期内完成（除法除外），实现了指令的高效执行和流水线的顺畅流动。

2.4 MEM 模块

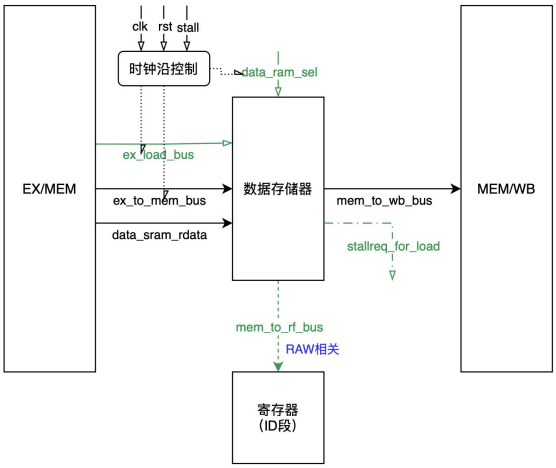


图 1 MEM 模块整体逻辑（绿色为增加的输入输出）

整体流程：

在 MEM 模块，系统从 EX/MEM 流水线寄存器（ex_to_mem_bus_r）中获取来自 EX 模块的控制信号、运算结果和数据选择信号。根据这些信息，MEM 模块处理内存访问操作。对于 load 操作，从数据存储器读取数据（data_sram_rdata），根据 load 指示信号（如 inst_lb）选择提取字节、半字还是字等并扩展（符号或零扩展），最终扩展成 32 位的输出（mem_result）。然后，根据 sel_rf_res 和 data_ram_en 选择写回寄存器的数据（rf_wdata），传递到 MEM/WB 流水线寄存器（mem_to_wb_bus）让 WB 模块完成写回寄存器的操作，同时另一条线直接传到寄存器（mem_to_rf_bus）用于 RAW 数据前推。

对于 store 操作，MEM 模块传递写使能信号（data_ram_wen）和数据选择信号（data_ram_sel_r）到数据存储器，实际写入动作由 sram 硬件完成（MEM 不直接执行写入，代码上只有 load 的完整链路）。此外，系统根据 rf_we 和 rf_waddr 控制是否写回寄存器，及写回寄存器的地址，让访存结果或 EX 输出能正确传递。

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
4	ex_to_mem_bus	80	输入	EX 传给 MEM 的数据总线
5	data_sram_rdata	32	输入	从内存读出来的值
6	data_ram_sel	4	输入	内存数据位置的选择信号
7	ex_load_bus	5	输入	EX 传递的load字长控制信号
8	stallreq_for_load	1	输出	对Load操作发出的停顿请求信号
9	mem_to_wb_bus	70	输出	MEM传给WB的数据总线
10	mem_to_rf_bus	38	输出	MEM传给寄存器的数据总线

表 3 MEM 输入输出信息

总线	包含	占位	含义	备注
ex_to_mem_bus_r ->	mem_pc	75:44	当前指令的PC	->为总线解析出右侧包含的变量；<-为右侧包含的变量传到总线中
	data_ram_en	43	数据RAM使能信号	
	data_ram_wen	42:39	数据RAM写使能信号	
	sel_rf_res	38	寄存器结果选择信号	
	rf_we	37	寄存器写使能信号	
	rf_waddr	36:32	寄存器写地址	
	ex_result	31:0	来自EX阶段的运算结果	
ex_load_bus_r ->	inst_lb	4	Load Byte	
	inst_lbu	3	Load Byte Unsigned	
	inst_lh	2	Load Halfword	
	inst_lhu	1	Load Halfword Unsigned	
	inst_lw	0	Load Word	
mem_to_wb_bus <-	mem_pc	69:38		
	rf_we	37		
	rf_waddr	36:32		
	rf_wdata	31:0	寄存器写数据	
mem_to_rf_bus <-	rf_we	37		
	rf_waddr	36:32		
	rf_wdata	31:0		

表 4 MEM 总线传输内部信息

功能实现：

最初的 MEM 部分代码只有一个 EX->MEM->WB 的骨架以及时钟控制逻辑，其中 mem_result 声明了却没有实现，这显然是不行的，不可能跑通。MEM 的最基础功能就是实现与内存交互，为了解决 RAW 相关，还要加一条向前的通路。因此我在原 MEM 骨架的基础上，增加了 load 指令的实现，并增加了 MEM 的 RAW 数据相关的通路。

具体实现：先正常操作，把输入的控制数据、EX 传输的数据分别暂存到寄存器 ex_load_bus_r、data_ram_sel_r、ex_to_mem_bus_r 中，以方便 CTRL 对流水线的统一控制。EX 传递过来的 ex_to_mem_bus 解析出 data_ram_en=1 时，允许读取内存提供的 data_sram_rdata。而指向内存的 data_ram_sel_r 的 4 位控制负责给出真正读取的字节是 32 位的 data_sram_rdata 中 4 个段中的哪一段（因为字节是 8 位），半字是 32 位的前半段还是后半段，然后将这些字节、半字、字的临时变量都暂存下来；然后通过判断 ex_load_bus 解析出的 inst_lw 等中哪一个为 1，对应选择要哪个类型的数据（字节、无符号字节、半字、无符号半字、字）。以此将内存提供对应长度的数据（已经选择好是前段还是后段等）选出来，再通过符号扩展或者零扩展扩展（无符号就零扩展，否则符号扩展）成 32 位的统一长度的 word，作为 mem_result；然后选择写回寄存器的 rf_wdata 是 EX 输出的结果 ex_result 还是刚从内存读取的 mem_result；最后将 rf_wdata 一条线正常传入到下一阶段的 WB 中（WB 负责写回寄存器，完成 load 操作的最后一步），还有一条新增的 forwarding 通路直接传到 ID 段的寄存器，解决这部分 RAW 的问题。

2.5 WB 模块

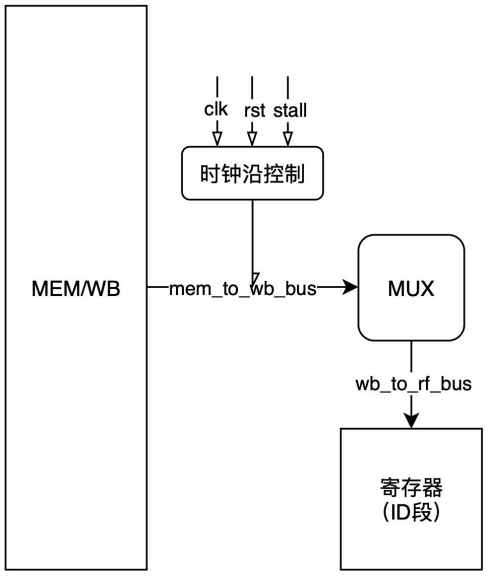


图 2 WB 模块整体逻辑（MUX 用来表示中间变量）

整体流程：

从 MEM/WB 流水线寄存器中读取数据并写回到 ID 段的寄存器中。

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
4	mem_to_wb_bus	70	输入	MEM传给WB的数据总线
5	wb_to_rf_bus	38	输出	WB传给寄存器的数据总线
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

表 5 WB 输入输出信息

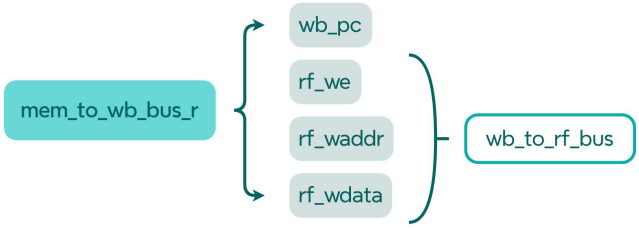


图 3 WB 总线解析、打包图；其中 wb_pc 为当前指令的 PC，rf_we 为寄存器写使能信号，rf_waddr 为寄存器写地址，rf_wdata 为寄存器写数据

功能实现：

WB 阶段的代码相较原代码没有太大修改，原代码已经实现了写回寄存器的通路。和 MEM 类似，有一个时钟控制逻辑，将 mem_to_wb_bus 总线的数据暂存到 mem_to_wb_bus_r 寄存器中，方便 CTRL 统一进行流水线控制。然后，将 mem_to_wb_bus_r 中的数据解析出来，再打包进 wb_to_rf_bus 总线传给 ID 段的寄存器即可。

2.6 CTRL 模块

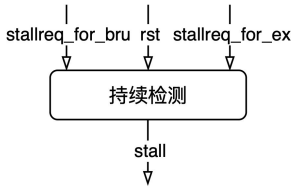


图 4 CTRL 模块整体逻辑

整体流程：

接收流水线各段的 stall 请求信号，更新 6 位的 stall 信号，得以控制流水线各阶段的运行。

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	EX的指令是否请求流水线暂停
3	stallreq_for_bru	5	输入	Branch分支命令是否请求流水线暂停
4	stall	6	输出	暂停信号

表 6 CTRL 输入输出信息

功能实现：

原代码只实现了复位信号 rst 或者没有任何 stall 请求输入时，stall 输入被重置为 0。为了使流水线中某一阶段能够顺利用多个时钟周期完成，我在代码中加入了 EX 的 stall 请求信号和分支阶段的 stall 请求信号。

具体来说，流通于各个阶段的 stall 信号是一个六位信号，其中 stall[0]表示 IF 阶段 PC 是否保持不变，为 1 表示保持不变。为了防止新的指令被取出，在当前指令完成的过程中，PC 值不应该发生改变。stall[1]、stall[2]、stall[3]、stall[4]、stall[5]分别表示 IF、ID、EX、MEM、WB 段是否暂停，对应位为 1 表示暂停，对应位为 0 表示正常运行。

当EX阶段的stall信号stallreq_for_ex传入时，输出的stall信号更新为001111，这里的位号是最右端为 0 号，因此当 EX 阶段请求暂停时，EX 阶段及之前的 ID、IF、PC 值的 stall 值都被设置成 1，全部暂停，而后续的 stall[4]、stall[5]仍然是 0，因此 MEM 段和 WB 段可以正常运行；同理当分支阶段的 stall 信号 stallreq_for_bru 为真时，由于是改进后的流水线，分支处理在 ID 段，因此 stall 输出更新为 000111，ID 段及以前的阶段 stall 设置为 1，暂停执行，ID 以后的阶段正常执行。最后输出这个更新后的六位 stall 信号。由于 stall 信号是流水线中每个阶段的输入，其中某位的变化在对应阶段的代码文件中的时间沿控制逻辑可以直接被检测到，从而控制实际的数据传输。

2.7 HI_LO_REG 模块

功能说明

HI/LO 寄存器存储功能：与通用寄存器不同，这两个寄存器专门用于存储乘法和除法运算的 64 位结果。在乘法指令中，64 位乘积的高 32 位存入 HI 寄存器，低 32 位存入 LO 寄存器；在除法指令中，商存入 LO 寄存器，余数存入 HI 寄存器。这种存储结构使得 32 位处理器能够处理 64 位的运算结果，为乘除法指令提供了必要的数据存储空间。

同步写控制功能：模块采用时钟同步机制控制寄存器写入，支持四种精确的写操作模式。当 hi_we 和 lo_we 信号同时有效时，两个寄存器同时更新，用于乘除法运算完成时的结果写入；当只有 hi_we 有效时，仅更新 HI 寄存器，对应 MTHI 指令执行；当只有 lo_we 有效时，仅更新 LO 寄存器，对应 MTLO 指令执行；当两个使能信号都无效时，寄存器保持当前值不变。这种精细的控制机制确保了数据写入的准确性和时序一致性。

异步读输出功能：模块提供组合逻辑输出端口，实时反映 HI 和 LO 寄存器的当前值。hi_rdata 和 lo_rdata 信号直接连接到内部寄存器的存储单元，无需时钟控制即可读取数据。这种设计使得 MFHI 和 MFLO 指令能够在执行阶段立即获取寄存器值，无读取延迟，支持数据前递和流水线高效执行。读取操作与写入操作完全独立，可同时进行，无访问冲突。

乘除法结果分离功能：门为乘除法指令的结果分离提供硬件支持。乘法运算产生的 64 位乘积自动分离为高 32 位和低 32 位，分别存储到 HI 和 LO 寄存器；除法运算产生的商和余数也自动分离存储。这种硬件级的分离机制简化了指令集设计，使得程序员可以通过简单的 MFHI 和 MFLO 指令分别访问结果的高位和低位部分，无需软件进行复杂的位操作。

流水线集成功能：模块设计考虑了与处理器流水线的紧密集成。虽然写入操作在写回阶段完成（与通用寄存器一致），但读取操作在执行阶段即可进行，这种时序安排优化了指令执行流程。模块还预留了流水线停顿接口（stall 信号），支持与整体流水线控制机制的协同工作，确保在多周期操作（如除法）时的数据一致性。

模块接口

信号类别	信号名称	位宽	方向	功能描述
时钟控制	clk	1	输入	系统时钟信号
流水线控制	stall	StallBus	输入	流水线停顿控制信号
写控制	hi_we	1	输入	HI 寄存器写使能信号
	lo_we	1	输入	LO 寄存器写使能信号

信号类别	信号名称	位宽	方向	功能描述
写数据	hi_wdata	32	输入	写入 HI 寄存器的数据
	lo_wdata	32	输入	写入 LO 寄存器的数据
读数据	hi_rdata	32	输出	HI 寄存器当前值
	lo_rdata	32	输出	LO 寄存器当前值

3. 实验感受与思考

3.1 李天园部分

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。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。

3.2 李芊芊部分

通过本次流水线 CPU 设计实验，我负责实现了 MEM（访存）、WB（写回）和 CTRL（控制）模块的代码，这让我对流水线架构有了更深刻的理解和实践体验。

首先，在实现 MEM 模块时，我深入学习了访存阶段的运作机制。从 EX/MEM 流水线寄存器中解包地址和控制信号，到处理 load 指令的数据提取、符号扩展和选择，最终生成写回数据并传递到 WB 阶段和寄存器文件，这个过程让我将理论知识转化为实际代码。特别是在处理字节/半字数据的选择和扩展时，我通过查阅 Verilog 语法和 DLX 流水线资料，解决了数据对齐和类型转换的问题，确保了访存结果的正确性。同时，WB 模块的实现让我理解了写回阶段的简单性：它主要负责暂存数据并最终写入寄存器，但也涉及 stall 控制下的数据保持，这强化了我对流水线同步的认识。

CTRL 模块的实现是最具挑战性的部分。作为流水线的控制核心，它通过组合逻辑根据不同冒险请求生成 stall 信号。我需要仔细分析 stall 位序和优先级，确保在 EX 多周期运算或分支冒险时正确暂停上游阶段，避免数据冒险。过程中，我遇到了位序混淆和逻辑优先级的问题，通过调试和参考标准 DLX 设计文档，最终调整了 stall 设置，使流水线能够稳定运行。这不仅提升了我的 Verilog 编程技能，还让我体会到控制逻辑在复杂系统中的关键作用。

总的来说，这次实验让我从理论走向实践，掌握了流水线 CPU 的设计方法，并深刻认识到模块化设计和调试的重要性。团队合作让我体会到沟通的价值，而查阅资料解决问题的过程增强了我的自学能力。这段经历不仅丰富了我的技术储备，还培养了工程思维，为未来的复杂项目奠定了基础。

3.3 楚可欣部分

开发 EX 执行单元让我全面理解了 CPU 计算核心的工作机制。这一模块不仅是 ALU 运算，更集成了地址计算、乘除处理、冒险控制和流水线协调等复杂系统，让我深刻体会到必须同时平衡功能实现与时序约束。其中乘除法单元的设计最具挑战——乘法器需支持有符号/无符号运算并生成 64 位结果，除法则需通过状态机处理多周期操作并与流水线协同。这让我学会了在性能、面积和时序之间进行精细权衡。

HI/LO 寄存器模块虽小，却体现了硬件设计的精妙。两个专用寄存器有效解决了 32 位处理器处理 64 位结果的问题，其读写时序分离设计（写回阶段写入、执行阶段读取）在确保数据稳定的同时支持数据前递。接口设计的清晰简洁大大降低了模块集成难度，也让我认识到良好的信号定义与文档规范在团队协作中的重要性。

调试过程是最宝贵的学习经历。面对除法器状态机异常、数据前递竞争等问题，我通过波形分析、分层测试逐步掌握了硬件调试方法。尤其是多周期操作与

流水线的协调，让我深入理解了时序同步与状态控制的复杂性。这次实践将课堂上的流水线冲突、数据冒险等理论转化为具体的工程实现，每个设计决策都需在时钟、面积、功耗等多重约束中寻求最优解。

最终看到指令正确执行时，我深切感受到计算机体系结构在理论与实践中交织的魅力。这次经历不仅提升了我的硬件设计能力，更培养了解决复杂工程问题的系统思维，为后续深入学习与开发奠定了扎实基础。