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 2302 班

组长：王丽莎

组员：邓雅文 丁学郅

报告日期：2025.12.28

目　录

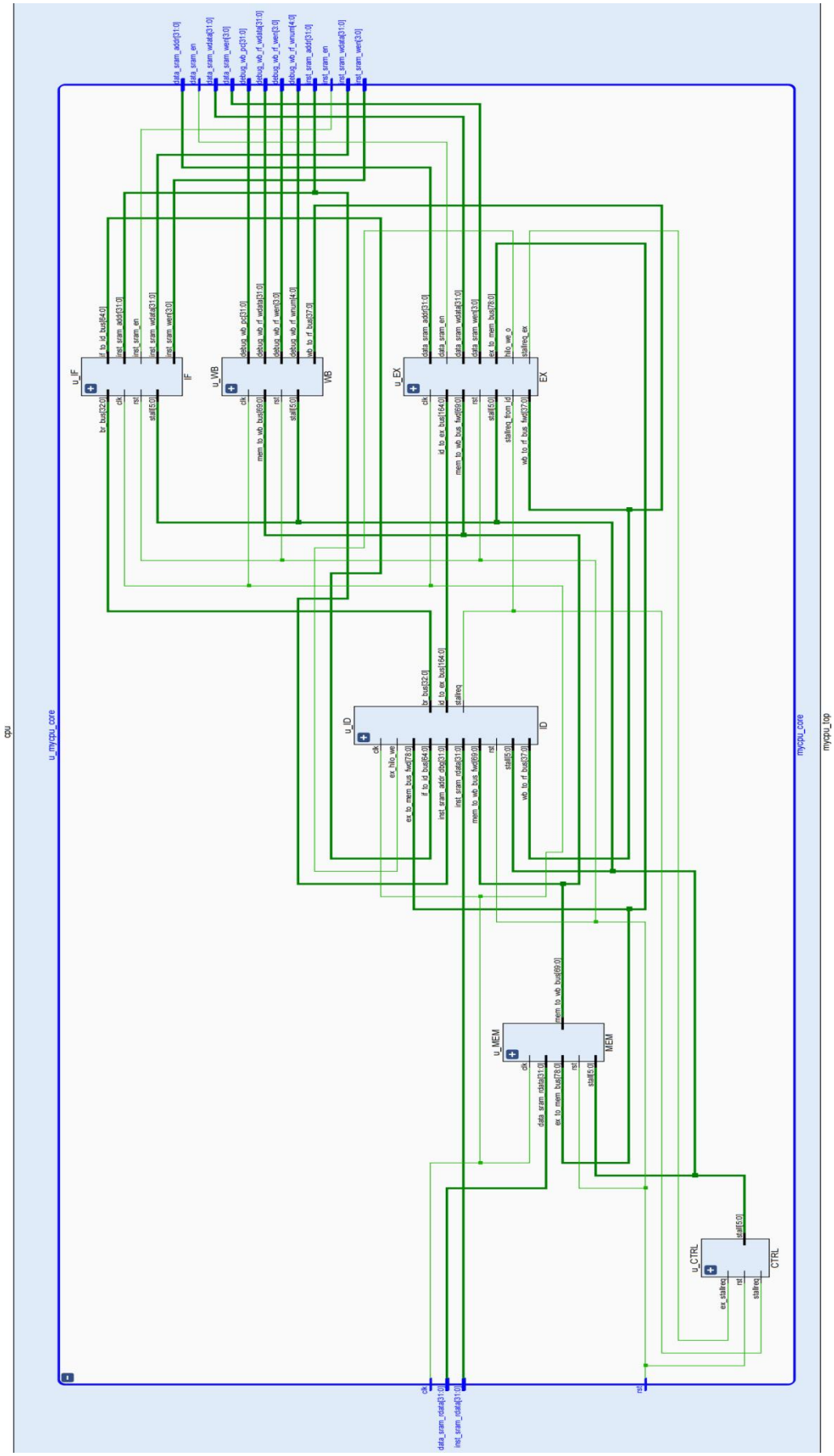

图 1 CPU 五级流水线总体设计示意图

一、项目总述

1.1 小组成员工作量划分

成员姓名	任务内容	工作量占比
王丽莎	负责 EX/MEM/WB 数据通路，完善前递、访存对齐与扩展、store 写掩码、HI/LO 与 div 多周期 stall，保证写回数据与 trace 完全一致。	35%
邓雅文	负责 ID 译码与控制信号生成，补全指令集、目的寄存器选择、分支比较前递及多类 hazard 检测，输出 stallreq 与 br_bus。	35%
丁学郅	负责 IF/CTRL/顶层联调，解决 PC/X 传播、stall/flush 框架、调试接口与仿真稳定性问题，建立 trace 驱动的定位流程。	30%

1.2 总体设计与实现

1.2.1 CPU 总体结构：

五级流水线结构：IF (取指)、ID (译码)、EX (执行)、MEM (访存)、WB (写回)

顶层框图：

- IF: 负责指令的获取与传递到 ID
- ID: 负责指令的译码，读取寄存器数据，生成控制信号
- EX: 执行算术运算、地址计算、控制信号
- MEM: 处理访存（load/store），与内存交互
- WB: 将数据写回寄存器堆

1.2.2 流水段之间的连线与总线组织：

关键总线连接：

1) pc + inst：从 IF 到 ID
2) control_signals：ID 生成控制信号并传递至 EX/MEM/WB
3) data_sram_addr/data_sram_wdata：数据内存接口
4) stall[ ]：用于流水线的停顿与同步

1.2.3 流水线的冒险处理与控制信号：

1) 冒险检测：load-use hazard、分支控制、HI/LO hazard（div）
2) 停顿与前递：使用 stallreq 和前递信号保证流水线的正确性
3) 控制信号：如 br_taken、br_addr 控制分支跳转，mem_en 控制访存，rf_we 控制写回

1.2.4 实现的 52 条指令

1) 算术/比较（10）

add, addu, addi, addiu; sub, subu; slt, sltu, slti, sltiu

2) 逻辑/立即数（8）

and, or, xor, nor; andi, ori, xori, lui

3) 移位（6）

sll, srl, sra; sllv, srlv, srav

4) 访存（8）

Load：lw, lb, lbu, lh, lhu; Store：sw, sb, sh

5) 跳转/分支（10）

Jump：j, jal, jr, jalr; Branch：beq, bne, bgez, bltz, bgtz, blez; 带链接：bgezal, bltzal

6) 乘除法 + HI/LO（10）

乘除：mult, multu, div, divu; 读写 HI/LO：mfhi, mflo, mthi, mtlo

1.2.5 实验工具与开发环境：

- Vivado 2019.2 进行设计、仿真与调试
- 使用 trace 比对工具进行功能验证
- 使用波形和调试口查看信号变化，确保设计正确性

二、单流水段说明

2.1 IF (Instruction Fetch) 模块

2.1.1 整体功能

1) IF 段维护 PC 寄存器 pc_reg 和 取指使能 ce_reg。
2) 根据 分支总线 br_bus 决定 next_pc：若分支成立取 br_addr，否则 pc+4。
3) 受 stall[0] 控制：暂停时保持 PC 不变，防止流水线前端乱跑。
4) 对外输出指令 SRAM 端口：inst_sram_en/wen/addr/wdata，并把 {ce_reg, pc_reg} 打包送入 ID（if_to_id_bus）。

2.1.2 包含的功能模块

1) PC 寄存器更新逻辑（受 stall/branch 控制）
2) next_pc 选择器（branch vs pc+4）
3) IF→ID 流水寄存器（这里用 bus 打包形式体现）

2.1.3 结构示意图

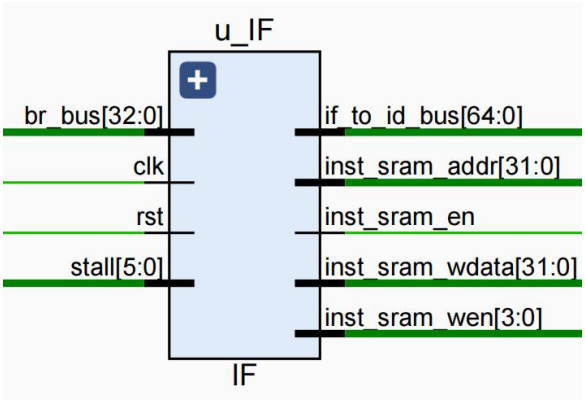

图 2 IF 段结构示意图

2.1.4 next_pc 的选择器

节选代码：

```
wire br_addr_valid = (^br_addr !== 1'bX);
wire br_taken = (br_e === 1'b1) && br_addr_valid;
assign next_pc = br_taken ? br_addr : (pc_reg + 32'h4);
```

解释：

br_taken 是“分支成立”信号；这里额外用 br_addr_valid 防止 br_addr 还是 X 时把 PC 带成 X。

这是一个 二选一选择器（MUX）：分支优先，否则顺序执行 pc+4。

IF 段在 always @(posedge clk) 中，只要 stall[0]==NoStop 才会把 pc_reg<=next_pc，暂停时 PC 保持不变。

2.1.5 端口和信号介绍

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[5:0]	6	输入	控制流水线暂停信号（stall）
4	br_bus[32:0]	33	输入	分支跳转信号，决定是否跳转
5	if_to_id_bus[64:0]	64	输出	将指令从 IF 模块传递到 ID 模块
6	inst_sram_en	1	输出	指令存储器的读取使能信号
7	inst_sram_addr[31:0]	32	输出	指令存储器地址
8	inst_sram_wdata[31:0]	32	输出	指令存储器的写数据
9	inst_sram_wen[3:0]	4	输出	指令存储器的写使能信号

2.2 ID (Instruction Decode) 模块

2.2.1 整体功能

1) ID 段从 inst_sram_rdata 取得指令，与 IF 传来的 PC 结合，完成 指令字段切分（rs/rt/rd/imm 等） 与 控制信号生成。
2) 通过 regfile 读出源操作数（rdata1/rdata2），并决定目的寄存器号、是否写回、ALU 操作类型、访存控制、写回选择等，最终打包成 id_to_ex_bus。
3) ID 同时负责 分支判断 并输出 br_bus（br_e + br_addr），指导 IF 改变 next_pc。
4) ID 负责冒险检测（典型是 load-use、以及 HI/LO 相关冲突），必要时拉高 stallreq 让 CTRL 发出暂停信号。

2.2.2 包含的功能模块

1) 指令译码（op/func/sel 等译码器）
2) 通用寄存器堆 regfile（读 rs/rt，写回来自 WB）
3) 分支比较与分支目标生成
4) 冒险检测（生成 stallreq）
5) 分支用的操作数前递（因为分支比较发生在 ID，不能等 WB）

2.2.3 结构示意图

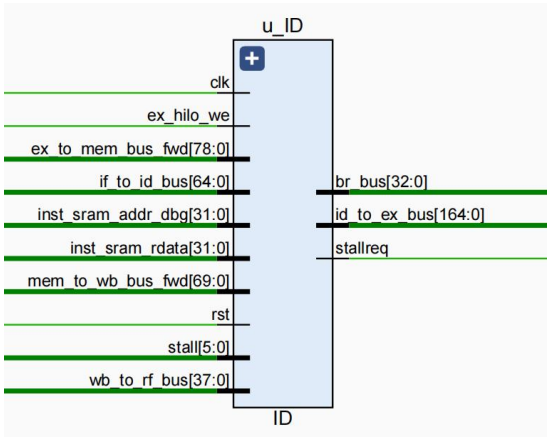

图 3 ID 段结构示意图

2.2.4 分支比较的前递选择器

节选代码：

```
always @(*) begin
    rdata1_br = rdata1;
    if (wb_rf_we    && wb_rf_waddr!=5'd0 && wb_rf_waddr==rs)
```

```
        rdata1_br = wb_rf_wdata;
    if(mem_rf_we_fwd&&mem_rf_waddr_fwd!=5'd0&& mem_rf_waddr_fwd==rs)
        rdata1_br = mem_rf_wdata_fwd;
    if (ex_rf_we_fwd && ex_sel_rf_res_fwd==1'b0 &&
        ex_rf_waddr_fwd!=5'd0 && ex_rf_waddr_fwd==rs)
        rdata1_br = ex_result_fwd; // 只旁路 ALU 结果，load 不能旁路
end
```

解释：

这是给分支比较用的 “多级覆盖式选择器”：后面的 if 会覆盖前面的赋值，所以形成优先级：EX(只限 ALU) > MEM > WB > 原始寄存器值。

为什么 EX 只旁路 ALU：如果 EX 指令是 load，数据还在 MEM 才会从 SRAM 返回，EX 阶段拿不到正确值，所以要么 stall，要么只允许 ALU 类旁路。

2.2.5 端口和信号介绍

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[5:0]	6	输入	控制流水线暂停信号（stall）
4	br_bus[32:0]	33	输入	分支跳转信号，决定是否跳转
5	if_to_id_bus[64:0]	64	输入	IF 模块传递到 ID 模块的指令和数据
6	inst_sram_en	1	输入	指令存储器的读取使能信号
7	inst_sram_addr[31:0]	32	输入	指令存储器地址
8	inst_sram_wdata[31:0]	32	输入	指令存储器的写数据
9	inst_sram_wen[3:0]	4	输入	指令存储器的写使能信号
10	regfile_rdata1[31:0]	32	输入	寄存器文件读数据 1
11	regfile_rdata2[31:0]	32	输入	寄存器文件读数据 2
12	rf_wdata[31:0]	32	输出	要写回寄存器的数据（来自 EX/MEM/WB）

2.3 EX (Execution) 模块

2.3.1 整体功能

1) EX 段接收 id_to_ex_bus，完成 ALU 运算、乘除/HI-LO 操作、以及 访存地址计算。
2) EX 段负责关键的 数据前递（forwarding）：当源寄存器依赖上一条/上上条指令结果时，从 MEM/WB 旁路数据，避免错误计算。
3) 对 store 指令（sb/sh/sw）生成正确的 data_sram_wen（写字节/半字掩码）与 data_sram_wdata（按字节复制/对齐）。
4) 输出 ex_to_mem_bus，其中包含 load_type、sel_rf_res（写回选 ALU 还是 MEM）、rf_we/rf_waddr/ex_result 等关键信息。

2.3.2 包含的功能模块

1) ALU
2) 乘除/HI-LO 寄存器更新与读出（mfhi/mflo/mthi/mtlo/mult/div…）
3) 前递选择器（rdata1_fwd / rdata2_fwd）
4) Store 写掩码与写数据生成器（sb/sh）

2.3.3 结构示意图

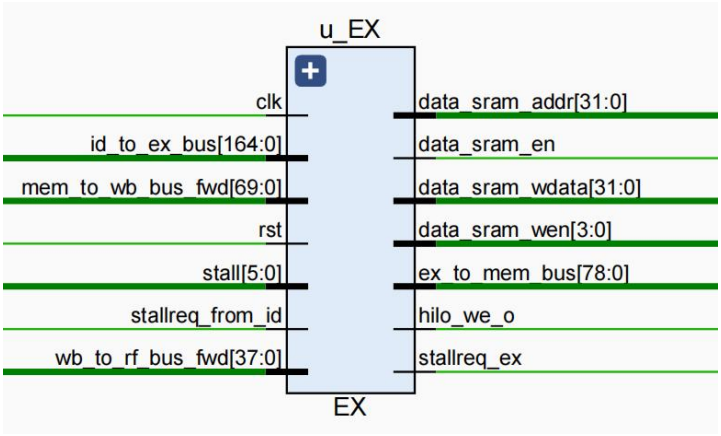

图 4 EX 段结构示意图

2.3.4 EX 段前递选择器

节选代码：

```
always @(*) begin
    rdata1_fwd = rf_rdata1;
    if (wb_rf_we_fwd    && wb_rf_waddr_fwd!=5'd0 && wb_rf_waddr_fwd==rs)
        rdata1_fwd = wb_rf_wdata_fwd;
    if(mem_rf_we_fwd&&mem_rf_waddr_fwd!=5'd0&& mem_rf_waddr_fwd==rs)
        rdata1_fwd = mem_rf_wdata_fwd;
end
```

解释：

这也是一个有优先级的 MUX：MEM > WB > 原始值（因为 MEM 更“新”）。这段是 always @(*)，代表纯组合逻辑：输入一变，输出立即重新计算（不等时钟）。

2.3.5 端口和信号介绍

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[5:0]	6	输入	控制流水线暂停信号（stall）
4	id_to_ex_bus[164:0]	165	输入	从 ID 阶段传递到 EX 阶段的数据，包括寄存器数据、控制信号等
5	ex_to_mem_bus[78:0]	79	输出	EX 阶段传递给 MEM 阶段的数据总线
6	ex_to_rf_bus[31:0]	32	输出	EX 阶段传递给 RF 阶段的总线数据
7	alu_result[31:0]	32	输出	ALU 运算结果
8	ex_stallreq	1	输出	来自 EX 阶段的 stall 请求信号
9	data_sram_en	1	输出	数据存储器的读取使能信号
10	data_sram_addr[31:0]	32	输出	数据存储器地址
11	data_sram_wdata[31:0]	32	输出	数据存储器的写数据
12	data_sram_wen[3:0]	4	输出	数据存储器的写使能信号

2.4 MEM (Memory Access) 模块

2.4.1 整体功能

MEM 段接收 EX 的 ex_to_mem_bus 与 data_sram_rdata。

对 load 指令，根据 load_type 和地址低两位 addr_offset 从 32 位读数据中抽取 byte/half，并做符号/零扩展。

通过 sel_rf_res 决定最终写回数据来源：load 用 mem_result，非 load 用 ex_result。

输出 mem_to_wb_bus，供 WB 写回与 debug/trace 使用。

2.4.2 包含的功能模块

Load 数据抽取器（byte/half 选择）

符号扩展/零扩展单元

写回数据选择器（MEM vs ALU）

2.4.3 结构示意图

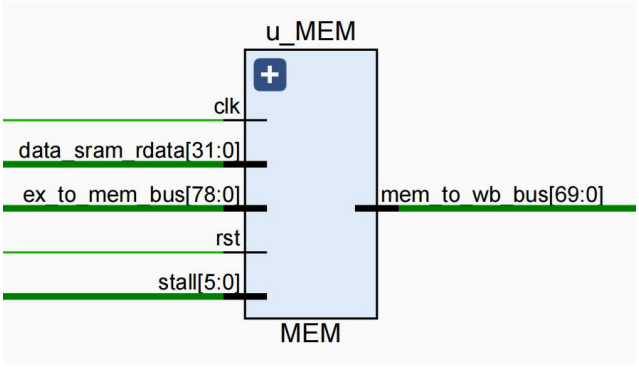

图 5 MEM 段结构示意图

2.4.4 load_type 选择器

节选代码：

```
always @(*) begin
    case (load_type)
        3'b001: mem_result_reg = {{24{byte_data[7]}}, byte_data};    // lb
        3'b010: mem_result_reg = {24'b0, byte_data};                 // lbu
        3'b011: mem_result_reg = {{16{half_data[15]}}, half_data}; // lh
        3'b100: mem_result_reg = {16'b0, half_data};                 // lhu
        default: mem_result_reg = data_sram_rdata;                   // lw
    endcase
end
```

解释：load_type 决定“取多大 + 怎么扩展”，这类错误最容易导致 trace 对不上。

2.4.5 端口和信号介绍

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[5:0]	6	输入	控制流水线暂停信号（stall）
4	ex_to_mem_bus[78:0]	79	输入	EX 阶段传递给 MEM 阶段的数据总线
5	data_sram_rdata[31:0]	32	输入	从数据存储器读取的数据
6	byte_data1_i[7:0]	8	输入	数据字节 1（用于存储指令字

				节/数据字节）
7	mem_result_reg[31:0]	32	输出	MEM 阶段计算结果数据
8	mem_to_wb_bus[89:0]	90	输出	MEM 阶段传递给 WB 阶段的数据总线

2.5 WB (Writeback) 模块

2.5.1 整体功能

WB 段把 mem_to_wb_bus 打一拍变成稳定的写回信息（mem_to_wb_bus_r）。

产生 wb_to_rf_bus = {rf_we, rf_waddr, rf_wdata} 写回寄存器堆。

同时输出 debug_wb_* 给 testbench 做 trace 比对：对外展示“我在 WB 这拍写了哪个寄存器、写了什么值、PC 是多少”。

支持 stall：当上游暂停/插入 bubble 时，WB 也会被置零防止错误写回。

2.5.2 包含的功能模块

MEM→WB 流水寄存器（带 stall/bubble 处理）

写回总线打包

debug 信号生成（特别关键，trace 就靠它）

2.5.3 结构示意图

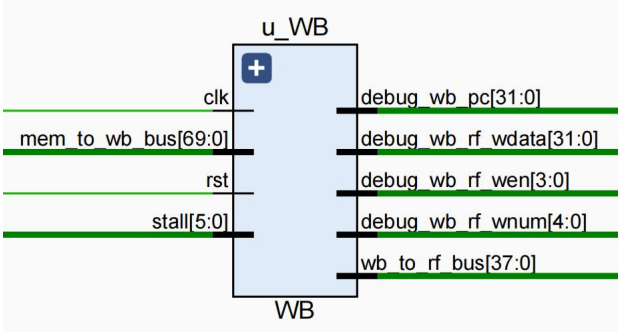

图 6 WB 段结构示意图

图注：mem_to_wb_bus → wb_reg → wb_to_rf_bus + debug_wb_*。

2.5.4 stall 时插 bubble

```
else if (stall[4]==`Stop && stall[5]==`NoStop) begin
    mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
end
```

解释：

当前级暂停但后级不停时，把本级输出清零相当于插入 bubble，避免把“旧指令结果”重复写回。

2.5.5 端口和信号介绍

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[5:0]	6	输入	控制流水线暂停信号（stall）
4	mem_to_wb_bus[89:0]	90	输入	MEM 阶段传递给 WB 阶段的数据总线
5	debug_wb_pc[31:0]	32	输出	从 WB 阶段输出的 PC 数据
6	debug_wb_rf_wdata[31:0]	32	输出	从 WB 阶段输出的寄存器数据
7	debug_wb_rf_wnum[4:0]	5	输出	从 WB 阶段输出的写回寄存器号

2.6 CTRL (Control Unit) 模块

2.6.1 整体功能

CTRL 汇总 ID 的 stallreq 和 EX 的 ex_stallreq，生成 6 位 stall 总线分发到各级。

你们代码中，只要任意请求暂停，就让 IF/ID/EX 停（6'b000111），保证前端不继续推进造成乱序/错误覆盖。

2.6.2 结构示意图

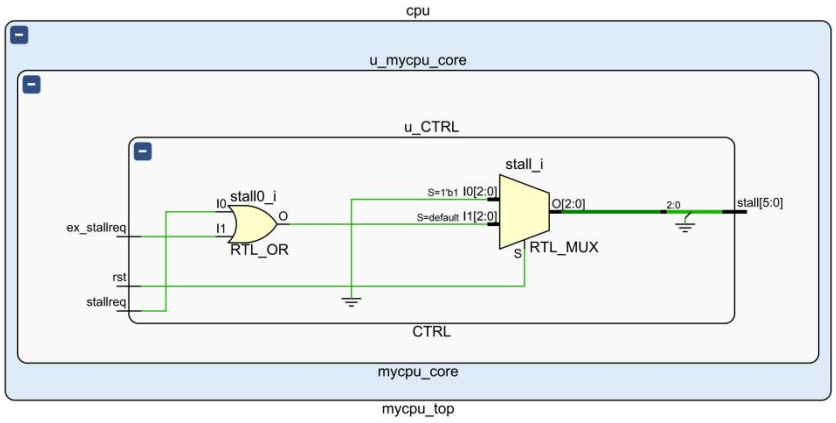

图 7 CTRL 段结构示意图

2.6.3 端口和信号介绍

序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号
2	stallreq	1	输入	来自 EX 阶段的 stall 请求

				信号
3	ex_stallreq	1	输入	来自 EX 阶段的额外 stall 请求信号
4	stall[5:0]	6	输出	控制流水线的停顿信号（最终合并后的）
5	ex_hilo_we	1	输出	HI/LO 写使能信号
6	ex_hilo_op[2:0]	3	输出	HI/LO 操作类型信号

三、组员的实验感受及改进意见

3.1 王丽莎

本阶段主要目标是把 EX/MEM/WB 的数据通路做到“算对、读对、写回对”。实现中，我重点梳理了 ALU 输入选择与前递链路，确保 EX 能拿到最新操作数；同时在 EX 侧完成访存地址与 store 写数据的组织（byte/half/word 对齐、写掩码生成），并把 load_type 等信息随总线传递到 MEM，在 MEM 阶段完成 load 数据的拆分、符号/零扩展与对齐，最终在 WB 统一选择写回来源，输出 wen/wnum/wdata 供 debug 校验。乘除与 HI/LO 方面，遵循握手/占用周期的约束，将 mfhi/mflo 的读、mthi/mtlo 的写以及 mult/div 的结果写入路径统一起来，并通过 stallreq_ex（或控制级 stall）保证多周期运算期间流水不破坏一致性。

实验中最直观的体会是：功能错误往往不是“算术算错”，而是“数据在错误的时间/通道被使用”，尤其是 load-use、分支比较旁路、以及 lb/lh 的扩展对齐，任何一个环节漏掉都会导致偶发错。

改进建议：①用“每条指令一张表”的方式固化控制信号（sel_rf_res、load_type、wen mask 等），减少隐式逻辑；②统一定义并强制检查流水总线宽度与字段，避免宏宽度与实际打包不一致；③前递/停顿策略模块化（ID/EX 分开、优先级明确），并补充定向用例（跨级前递、连续 load、misaligned 边界）；④增加 reset 清零与 write-first bypass 的一致接线检查，减少 X 扩散与仿真/上板差异。

3.2 邓雅文

在本项实验中，我主要负责 ID 译码阶段及全局控制信号生成的核心任务。作为流水线的“指挥中枢”，这一部分的逻辑复杂度较高。

在指令集的补全与译码部分，我深刻体会到独热码译码器的严谨性，这也要求我们提高对 MIPS 指令格式及 R/I/J 型指令差异的敏感度。最具挑战性的是

Hazard（冲突）检测与前递逻辑。为了提升性能，我们将分支比较提前到 ID 阶段。在编写分支比较前递时，我意识到仅靠寄存器堆的数据是不够的，必须实时截获后级流水线中实时产生的数据。当遇到 Load-Use 等无法前递的硬件限制时，通过精确计算输出 stallreq 暂停请求，并协调 CTRL 模块输出 stall 向量，我成功解决了数据冒险和控制冒险，实现了“零空泡”或“最小空泡”的流水线运行。

这次实验让我明白，CPU 并不是孤立模块的堆砌，而是时序与逻辑的精密博弈。在实操中，我不仅初步熟悉了 Verilog 语言，更对计算机底层架构中“协同调度”的精髓有了切身的体会。

3.3 丁学郅

在本次 CPU 五级流水线实验中，我主要负责我负责 IF、CTRL 及顶层联调，重点保障 PC 生成、取指、stall 等控制逻辑的正确性。

初期 PC 因分支地址未初始化而变 X，导致仿真结果随机出错；通过引入“有效性保护”、stall 时保持 PC、flush 时插 NOP，有效阻断了 X 传播。调试中还发现 stall/flush 范围不全易引发流水线卡死，最终在 CTRL 中统一处理各类停顿请求，明确各级停顿与清空规则，掌握了“停-清-保持”三要素的核心作用。此外，因顶层 debug 接口未完整连接，曾导致 trace 比对失败，补齐 WB 阶段的 pc/wnum/wdata/wen 输出后验证效率显著提升。

总的来说，通过深入参与 PC 生成、指令取指、流水线停顿（stall）等关键逻辑的实现与验证，我不仅夯实了对流水线控制机制的理解，更在实战中体会到“干净的控制信号”对于整个系统稳定性的决定性作用。