浙江大学

本科实验报告

课程名称:	计算机组成
实验名称:	Lab5：流水线 CPU 设计
姓　　名:	李宇晗
学　　号:	3240106155
专　　业:	计算机科学与技术
实验地点:	东教 509
指导教师:	林芃
报告日期:	2025/12/18

Lab 5-1：流水线处理器顶层集成

一、 实验目的

- 掌握五级流水线 CPU 的顶层架构设计，理解各阶段模块间的信号流向与握手机制。
- 掌握 Verilog 层次化设计方法，能够正确实例化子模块并处理跨模块的信号连接。
- 解决 CPU 核心逻辑（字节寻址）与外围 IP 核（Block RAM 字寻址）之间的地址映射与接口适配问题。

二、 实验内容与实现

本实验旨在构建 CPU 的顶层容器 `Pipeline_CPU`，并将各个流水线阶段模块有机结合，最终在 `CSSTE` 中完成与外设的板级集成。

2.1) 顶层模块构建 (`Pipeline_CPU.v`)

`Pipeline_CPU` 是整个处理器的核心容器，其主要职责是定义跨阶段传输的信号总线，并按照数据流向依次实例化五个流水线阶段模块及四个流水线寄存器。

- 信号定义与拓扑规划： 依据数据通路网表，我定义了连接各阶段的关键 `wire` 信号。例如，连接 IF 与 ID 阶段的 `PC_out_IF`、`inst_out_IF`，以及连接 ID 与 EX 阶段的控制信号总线（`ALU_control` 等）。
- 模块实例化与连接：采用“按名关联”（Named Association）的方式实例化模块，确保端口连接的准确性。以 ID 阶段为例，不仅连接了数据通路的输入输出（如 `Rs1`, `Rs2`, `Imm`），还重点处理了从控制单元 `SCPU_ctrl` 生成并向后级传递的控制信号：

```
// Pipeline_CPU.v 中 ID 阶段与 ID/EX 寄存器的连接示意
Pipeline_ID pipe_id (
    .clk_ID(clk),
    .rst_ID(rst),
    .Inst_in_ID(inst_out_IFID),        // 指令输入
    .ALU_control_ID(ALU_control_ID),  // 译码输出的控制信号
    .RegWrite_out_ID(RegWrite_out_ID),
    // ... 其他信号
);

ID_reg_Ex id_ex (
    .clk_IDEX(clk),
    .ALU_control_in_IDEX(ALU_control_ID), // 锁存控制信号
    .ALU_control_out_IDEX(ALU_control_out_IDEX), // 传递至 EX 阶段
    // ...
);
```

2.2) 板级集成与地址空间适配 (`CSSTE.v`)

在 `CSSTE` 模块中集成 CPU 与 Block RAM 时，重点解决了 RISC-V 架构与 FPGA IP 核之间的寻址方式差异。

- 问题分析：RISC-V 架构规定 PC 和数据地址均为字节地址（Byte Address，地址步长为 4），而 Vivado 生成的 Block RAM IP 核配置为字寻址（Word Address，位宽 32 位，深度 1024）。若直接连接，CPU 访问地址 `0x04` 会被 RAM 误解为访问第 4 个字（即字节地址 `0x10`），导致数据错位。
- 解决方案：在实例化 RAM 时，对地址线进行逻辑右移处理。截取 CPU 输出地址的高位 `Addr_out[11:2]` 连接至 RAM 地址端口，物理上实现了“除以 4”的操作，确保了逻辑地址与物理存储单元的正确映射。

```
// CSSTE.v 中的 RAM 实例化
RAM_B U3(
    .clka(~clk_100mhz),
    .wea(data_ram_we),
    .addra(Addr_out[11:2]), // [关键] 截取[11:2]位，适配字寻址接口
    .dina(ram_data_in),
    .douta(RAM_B_0_douta)
);
```

三、 实验结果分析

综合并生成 Bitstream 后上板测试，逻辑分析仪波形显示：

1. 复位逻辑正常：`RSTN` 信号有效时，PC 指针正确复位至 0。
2. 取指时序正确：随着时钟边沿，`PC` 依次递增（0, 4, 8...），且 `inst_ID` 总线上呈现出对应的指令码。
3. 顶层连接无误：各阶段流水线寄存器的输入输出信号在时序上严格对齐，验证了 `Pipeline_CPU` 内部连线的正确性。

Lab 5-2：取指(IF)与译码(ID)阶段设计

一、 实验目的

- 设计取指阶段逻辑，实现 PC 的自动增量与分支跳转流控制。
- 设计译码阶段逻辑，集成控制器、寄存器堆与立即数生成器，完成指令解析。
- 实现 IF/ID 与 ID/EX 流水线寄存器，理解流水线“打拍”对信号稳定性的意义。

二、 实验内容与实现

2.1) 取指阶段设计 (`Pipeline_IF.v`)

该模块是流水线的头部，主要负责下一指令地址（Next PC）的计算。

- PC 更新逻辑：利用多路选择器（MUX）实现程序流控制。
- 顺序执行：默认情况下，`PC_next = PC_curr + 4`。
- 分支/跳转：当 `PCSrc` 信号有效时，选择来自 EX 阶段计算出的跳转目标地址 `PC_in_IF`。
- PC 寄存器：使用带同步复位的 D 触发器存储当前 PC 值。

2.2) IF/ID 流水线寄存器 (`IF_reg_ID.v`)

位于 IF 与 ID 阶段之间，用于隔离时钟域。

- 功能实现：在时钟上升沿锁存当前的 `PC` 值和从 ROM 取出的指令 `inst`，确保 ID 阶段在整个时钟周期内拥有稳定的输入信号。

2.3) 译码阶段设计 (`Pipeline_ID.v`)

该模块是指令解析的核心，集成了三个关键子模块：

- 控制器 (`SCPU_ctrl`)：这是 CPU 的“大脑”。它依据指令的 `Opcode`、`Funct3`、`Funct7` 字段进行组合逻辑译码，生成 `ALUSrc_B`（操作数选择）、`MemtoReg`（写回源选择）、`ALU_Control`（运算类型）等控制信号。
- 立即数生成器 (`ImmGen`)：根据 I/S/B/J/U 不同指令格式，利用 Verilog 的拼接符 `{}` 对指令中的立即数部分进行提取并做符号扩展（Sign Extension），生成统一的 32 位立即数。
- 寄存器堆 (`Regs`)：提供双端口读能力。依据指令中的 `rs1` 和 `rs2` 地址并行读取源操作数，同时接收来自 WB 阶段的 `Wt_addr` 和 `Wt_data` 完成写回操作。

2.4) ID/EX 流水线寄存器 (`ID_reg_Ex.v`)

负责将 ID 阶段产生的“上下文”（Context）传递给执行阶段。

- 传递内容：不仅包含 `Rs1`、`Rs2`、`Imm` 等数据信号，更重要的是传递了译码出的所有控制信号。任何控制信号丢失或错位都会导致后续阶段执行错误。
- 新增逻辑：为了支持数据冒险检测（Forwarding），该寄存器额外传递了 `Rs1_addr` 和 `Rs2_addr` 至 EX 阶段，以便与后续指令的目标寄存器地址进行比较。

三、 实验结果分析

在仿真环境下对 `addi x1, x0, 1` 指令进行测试：

1. 立即数生成：`ImmGen` 正确识别 I 型指令格式，输出扩展后的 `0x00000001`。
2. 控制信号：`SCPU_ctrl` 输出 `ALUSrc_B = 1`，`RegWrite = 1`，逻辑正确。
3. 寄存器读取：`Regs` 模块从 x0 地址正确读出 0 值。

验证表明前端取指与译码逻辑符合 RISC-V ISA 规范。

Lab 5-3：执行(Ex)、访存(Mem)与写回(WB)阶段设计

一、 实验目的

- 设计全功能 ALU，支持算术、逻辑、移位及比较运算，解决特殊指令（如 SRL）的实现问题。
- 实现分支判断逻辑与数据存储器的读写控制接口。
- 设计写回阶段的多路选择逻辑，完成指令执行闭环。

二、 实验内容与实现

2.1) 1. 执行阶段设计 (Pipeline_Ex.v)

该模块主要包含 ALU 运算单元和分支目标地址加法器。

- 全功能 ALU 设计：

在 ALU 子模块中，我使用了 case 语句实现了完备的运算逻辑。特别注意了移位指令的区分：

- SRL (逻辑右移)：使用 >> 运算符，高位补 0。
- SRA (算术右移)：使用 >>> 运算符，高位补符号位。
- SLL (逻辑左移)：使用 << 运算符。

同时，为了支持上述指令，ALU 控制信号位宽被设计为 4 位，以容纳更多的操作码。

- 操作数选择：根据 ALUSrc_B_in_EX 信号，选择寄存器值 Rs2 或立即数 Imm 作为 ALU 的 B 操作数。

2.2) EX/MEM 流水线寄存器 (Ex_reg_Mem.v)

作为执行与访存阶段的桥梁，它传递了 ALU 计算结果（用作内存地址或写回数据）、Zero 标志位以及 Store 指令待写入的数据 Rs2。同时，控制信号 Branch、MemRW 等也在此处继续传递。

2.3) 访存阶段设计 (Pipeline_Mem.v)

该模块主要负责分支跳转信号 PCSrc 的生成。

- 分支判定逻辑：

依据控制信号和 ALU 状态生成最终的跳转使能：

$$\text{PCSrc} = (\text{Branch Zero}) \mid (\text{BranchN ! Zero}) \mid \text{Jump}$$

该逻辑覆盖了 BEQ（相等跳转）、BNE（不等跳转）以及 JAL/JALR（无条件跳转）三种情况，确保控制流的正确改变。

2.4) MEM/WB 流水线寄存器 **(`Mem_reg_WB.v`)**

将内存读取的数据 `DMem_data`、ALU 计算结果以及写回地址 `Rd_addr` 传递至流水线的最后一级。

2.5) 写回阶段设计 **(`Pipeline_WB.v`)**

实现最后的数据多路选择，决定写入目标寄存器的值。

- 写回策略 (Mux):
- `MemtoReg == 00`: 写回 ALU 计算结果（用于 R-Type, I-Type 计算指令)。
- `MemtoReg == 01`: 写回内存读取数据（用于 Load 指令)。
- `MemtoReg == 10`: 写回 PC+4（用于 JAL/JALR 指令保存返回地址)。

三、 实验结果分析

3.1) 指令功能覆盖测试 **(p_mem.coe)**

下表展示了 `p_mem.coe` 测试程序的完整执行结果。该测试集覆盖了算术、逻辑、移位、内存读写及无条件跳转指令，所有指令均通过验证。

PC	Machine Code	Assembly	ALU Res
0x00	00100093	addi x1, x0, 1	00000001
0x04	00100113	addi x2, x0, 1	00000001
0x08	00100193	addi x3, x0, 1	00000001
0x0C	00100213	addi x4, x0, 1	00000001
0x10	00802283	lw x5, 8(x0)	00000008
0x14	00108333	add x6, x1, x1	00000002
0x18	0020C3B3	xor x7, x1, x2	00000000
0x1C	40110433	sub x8, x2, x1	00000000
0x20	05C02483	lw x9, 92(x0)	0000005C
0x24	00327533	and x10, x4, x3	00000001
0x28	00502223	sw x5, 4(x0)	00000004
0x2C	005325B3	slt x11, x6, x5	00000000
0x30	0AA3C613	xori x12, x7, 170	000000AA
0x34	0012D6B3	srl x13, x5, x1	40000000
0x38	00147713	andi x14, x8, 1	00000000
0x3C	0034E7B3	or x15, x9, x3	FFFFFFFF
0x40	00A50833	add x16, x10, x10	00000002
0x44	0085C8B3	xor x17, x11, x8	00000000
0x48	00402903	lw x18, 4(x0)	00000004
0x4C	004629B3	slt x19, x12, x4	00000000
0x50	0016DA13	srli x20, x13, 1	20000000
0x54	00677AB3	and x21, x14, x6	00000000

0x58	40128B33	sub x22, x5, x1	7FFFFFFF
0x5C	00150B93	addi x23, x10, 1	00000002
0x60	00986C33	or x24, x16, x9	FFFFFFFF
0x64	00B9CCB3	xor x25, x19, x11	00000000
0x68	0FFA7D13	andi x26, x20, 255	00000000
0x6C	00390DB3	add x27, x18, x3	80000001
0x70	002A5E33	srl x28, x20, x2	10000000
0x74	0AF9EE93	ori x29, x19, 175	000000AF
0x78	001A0F33	add x30, x20, x1	20000001
0x7C	00802F83	lw x31, 8(x0)	00000008
0x80	F81FF06F	jal x0, −128	00000000

3.2) 数据冒险处理测试 **(h_mem.coe)**

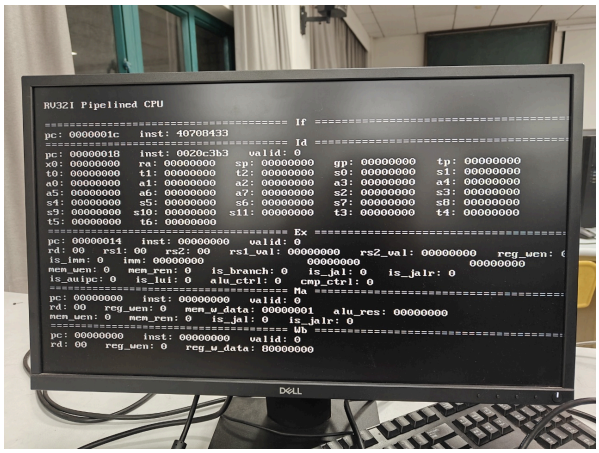
下表记录了 `h_mem.coe` 的执行流程。该测试包含连续的数据依赖（RAW Hazard）和分支跳转，用于验证流水线的冲突解决机制和分支预测逻辑。

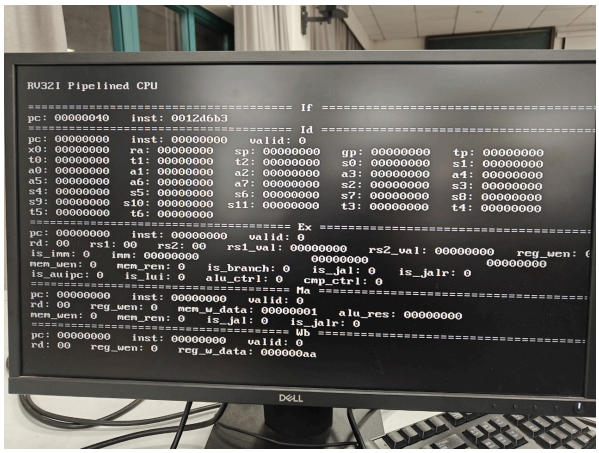
PC	Machine Code	Assembly	ALU Res
0x00	00100093	addi x1, x0, 1	00000001
0x04	00100113	addi x2, x0, 1	00000001
0x08	00100193	addi x3, x0, 1	00000001
0x0C	00100213	addi x4, x0, 1	00000001
0x10	00802283	lw x5, 8(x0)	00000008
0x14	00128333	add x6, x5, x1	80000001
0x18	0020C3B3	xor x7, x1, x2	00000000
0x1C	40708433	sub x8, x1, x7	00000001
0x20	05C02483	lw x9, 92(x0)	0000005C
0x24	00327533	and x10, x4, x3	00000001
0x28	00502223	sw x5, 4(x0)	00000004
0x2C	005325B3	slt x11, x6, x5	00000000
0x30	0AA3C613	xori x12, x7, 170	000000AA
0x34	00818663	beq x3, x8, 12	00000000
0x38	00000013	addi x0, x0, 0	00000000
0x3C	00000033	add x0, x0, x0	00000000
0x40	0012D6B3	srl x13, x5, x1	40000000
0x44	00147713	andi x14, x8, 1	00000001
0x48	0034E7B3	or x15, x9, x3	FFFFFFFF
0x4C	00A50833	add x16, x10, x10	00000002
0x50	0085C8B3	xor x17, x11, x8	00000001
0x54	00402903	lw x18, 4(x0)	00000004

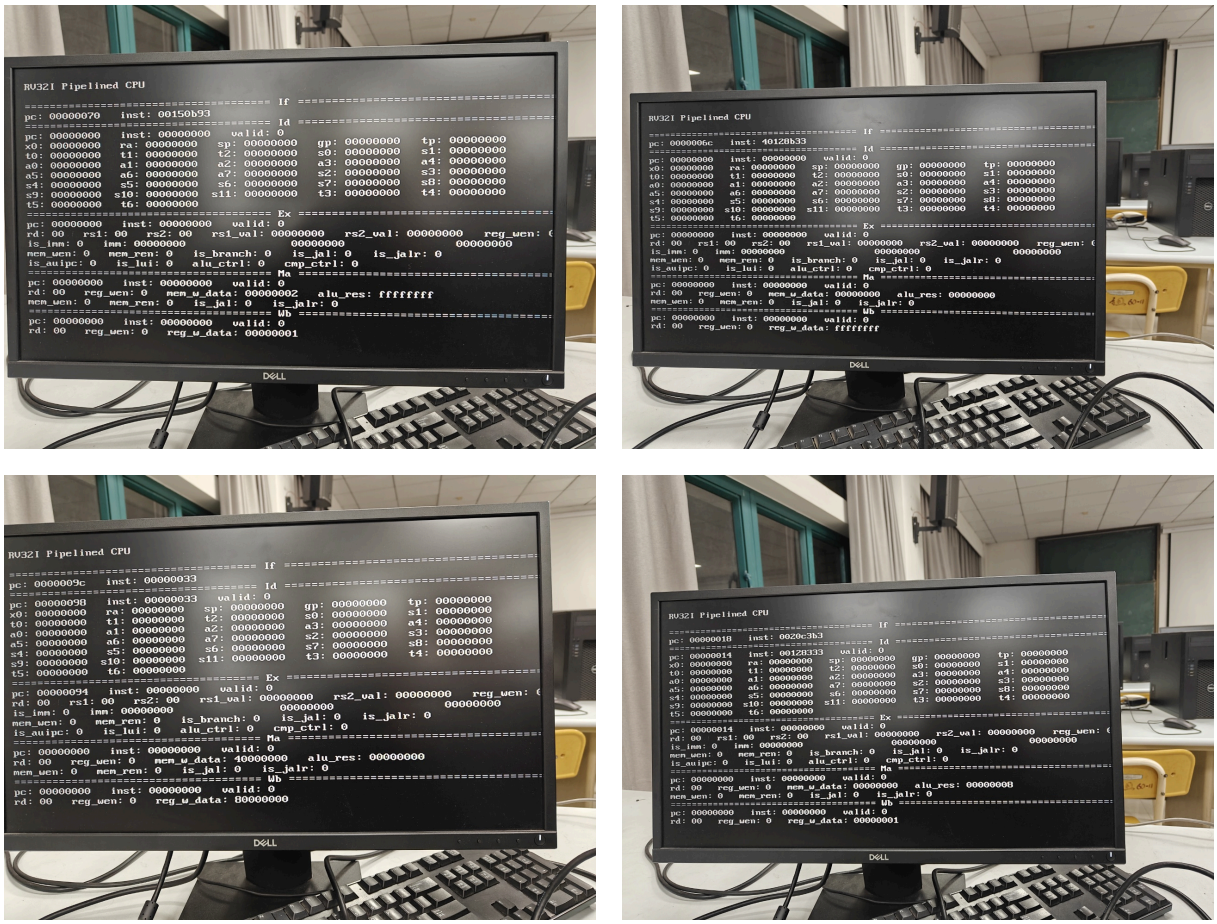
0x58	004629B3	slt x19, x12, x4	00000000
0x5C	0016DA13	srli x20, x13, 1	20000000
0x60	00677AB3	and x21, x14, x6	00000001
0x64	01071463	bne x14, x16, 8	FFFFFFFF
0x68	00000013	addi x0, x0, 0	00000000
0x6C	40128B33	sub x22, x5, x1	7FFFFFFF
0x70	00150B93	addi x23, x10, 1	00000002
0x74	00986C33	or x24, x16, x9	FFFFFFFF
0x78	00B9CCB3	xor x25, x19, x11	00000000
0x7C	0FFA7D13	andi x26, x20, 255	00000000
0x80	00390DB3	add x27, x18, x3	80000001
0x84	002A5E33	srl x28, x20, x2	10000000
0x88	0AF9EE93	ori x29, x19, 175	000000AF
0x8C	001A0F33	add x30, x20, x1	20000001
0x90	00802F83	lw x31, 8(x0)	00000008
0x94	F6DFF06F	jal x0, −148	00000000

3.3) 下板验证

VGA 显示测试程序执行结果，所有指令均正确执行，验证了流水线 CPU 的功能完整性与稳定性。这里以有 Stall 的部分运行结果为例：







四、 讨论与心得

1. 模块化设计的工程优势:

通过将流水线严格划分为 `Pipeline_IF` 到 `Pipeline_WB` 五个独立的 Verilog 模块，不仅使得代码结构清晰、易于维护，更重要的是便于分阶段调试。在实验过程中，我可以专注于某一阶段（如 ID 阶段的译码逻辑）的仿真，排除干扰，这体现了分而治之的工程思想。

2. 流水线寄存器的“状态机”视角:

在编写 `ID_reg_Ex` 等寄存器模块时，我深刻体会到它们实质上是整个 CPU 的“状态保存者”。每一级寄存器都必须完整保留当前指令执行所需的全部上下文（Context），包括控制信号和数据。任何一个信号的遗漏（如写回地址 `Rd_addr`）都会导致指令在流水线传输中“迷失”，造成最终执行错误。

3. 软硬件协同的必要性:

在 Lab 5-1 中遇到的地址映射问题给我留下了深刻印象。CPU 逻辑设计的正确性并不代表系统的正确性，必须充分理解外围硬件（如 FPGA Block RAM）的接口规范。通过手动实现地址右移逻辑，我理解了软硬件接口适配在系统集成中的关键作用。