实验三：流水线 CPU 设计与实现

朱梓涵

学号：24325356

2025 年 12 月 1 日

1 实验目的

本实验旨在深入理解流水线技术在 CPU 设计中的应用，通过实现三级和五级流水线 CPU，掌握流水线中竞争冒险的处理方法。实验目标包括：

1. 理解流水线寄存器的作用，并实现支持阻塞和清空功能的流水线寄存器。
2. 设计并实现三级流水线 CPU，掌握控制冒险的处理方法。
3. 设计并实现五级流水线 CPU，学习使用阻塞和转发技术处理数据冒险。
4. 将分支跳转提前到译码阶段，进一步缩短分支延迟，优化流水线性能。
5. 通过编写测试用例，验证流水线 CPU 各模块及整体功能的正确性。

2 实验环境

- **操作系统**: Windows 11
- **开发工具**: IntelliJ IDEA
- **构建工具**: SBT
- **仿真与测试**: Verilator, chiseltest

3 流水线结构与功能划分

3.1 三级流水线结构

三级流水线将单周期 CPU 的组合逻辑切分为三个阶段：

- **取指 (IF)**: 根据 PC 从指令存储器取出指令。
- **译码 (ID)**: 解码指令，生成控制信号，从寄存器组读取操作数。
- **执行 (EX)**: 执行 ALU 运算、访存和写回操作。

在三级流水线中，主要需要处理控制冒险。当 EX 段执行跳转或分支指令时，IF 和 ID 段的两条指令需要被清空。

3.2 五级流水线结构

五级流水线在三级流水线的基础上，将 EX 段进一步细分：

- **取指 (IF)**: 根据 PC 从指令存储器取出指令。
- **译码 (ID)**: 解码指令，生成控制信号，从寄存器组读取操作数。
- **执行 (EX)**: 执行 ALU 运算。
- **访存 (MEM)**: 访问数据存储器。
- **写回 (WB)**: 将结果写回寄存器组。

五级流水线引入了更复杂的数据冒险，需要使用阻塞和转发技术进行处理。

3.3 缩短分支延迟的五级流水线

在最终版本的五级流水线中，将分支和跳转指令的执行从 EX 段提前到 ID 段：

- 在 ID 段增加加法器，用于计算跳转目标地址。
- 在 ID 段进行分支条件判断，使用转发逻辑从 MEM 和 WB 段获取操作数。
- 如果依赖的数据还未产生，则进行阻塞。

这样做可以将分支延迟从两个时钟周期减少到一个时钟周期。

4 模块实现与分析

4.1 流水线寄存器（PipelineRegister）

4.1.1 功能

流水线寄存器是流水线 CPU 的核心组件，用于在相邻流水段之间缓存数据和控制信号。它支持三种操作：

- **清空 (flush)**: 将寄存器值重置为默认值，用于清除错误路径上的指令。
- **阻塞 (stall)**: 保持当前寄存器值不变，用于暂停流水线。
- **正常更新**: 在时钟上升沿将输入值写入寄存器。

4.1.2 代码实现

```
class PipelineRegister(width: Int = Parameters.DataBits, defaultValue:
   UInt = 0.U) extends Module {
  val io = IO(new Bundle {
    val stall = Input(Bool())
    val flush = Input(Bool())
    val in = Input(UInt(width.W))
    val out = Output(UInt(width.W))
  })
  // Lab3(PipelineRegister)
  val register = RegInit(defaultValue)

  when(io.flush) {
    register := defaultValue
  }.elsewhen(io.stall) {
  }.otherwise {
    register := io.in
  }

  io.out := register
  // Lab3(PipelineRegister) End
}
```

Listing 1: 流水线寄存器实现

4.1.3 设计要点

- 优先级：`flush` 信号优先级最高，其次是 `stall`，最后是正常更新。
- 当 `flush` 为高时，寄存器被清空为默认值（通常是 NOP 指令或 0）。
- 当 `stall` 为高时，寄存器保持当前值不变。
- 其他情况下，寄存器在时钟上升沿更新为输入值。

4.2 控制单元（Control）

4.2.1 功能

控制单元负责检测流水线中的冒险，并生成相应的控制信号：

- 检测数据冒险，生成阻塞信号。
- 检测控制冒险，生成清空信号。

4.2.2 代码实现（最终版本）

```
class Control extends Module {
  val io = IO(new Bundle {
    val jump_flag = Input(Bool())
    val jump_instruction_id = Input(Bool())
    val rs1_id = Input(UInt(Parameters.PhysicalRegisterAddrWidth))
    val rs2_id = Input(UInt(Parameters.PhysicalRegisterAddrWidth))
    val memory_read_enable_ex = Input(Bool())
    val rd_ex = Input(UInt(Parameters.PhysicalRegisterAddrWidth))
    val memory_read_enable_mem = Input(Bool())
    val rd_mem = Input(UInt(Parameters.PhysicalRegisterAddrWidth))

    val if2id_flush = Output(Bool())
    val id2ex_flush = Output(Bool())
    val pc_stall = Output(Bool())
    val if2id_stall = Output(Bool())
  })

  // Lab3(Final)
```

```
  val stall = Wire(Bool())

  val load_use_hazard = io.memory_read_enable_ex && io.rd_ex =/= 0.U &&
    (io.rd_ex === io.rs1_id || io.rd_ex === io.rs2_id)

  val id_jump_needs_ex_alu = io.jump_instruction_id && io.rd_ex =/= 0.U
    &&
    !io.memory_read_enable_ex &&
    (io.rd_ex === io.rs1_id || io.rd_ex === io.rs2_id)

  val id_jump_needs_mem_load = io.jump_instruction_id && io.
   memory_read_enable_mem && io.rd_mem =/= 0.U &&
    (io.rd_mem === io.rs1_id || io.rd_mem === io.rs2_id)

  stall := load_use_hazard || id_jump_needs_ex_alu ||
   id_jump_needs_mem_load

  val flush = io.jump_flag && !stall
  io.pc_stall := stall
  io.if2id_stall := stall
  io.if2id_flush := flush

  io.id2ex_flush := stall
  // Lab3(Final) End
}
```

Listing 2: 控制单元实现（缩短分支延迟版本）

4.2.3 设计要点

在缩短分支延迟的版本中，由于分支和跳转指令在 ID 段执行，控制单元需要处理以下几种情况：

1. **Load-use 冒险**: 当 ID 段的指令需要使用 EX 段 load 指令的结果时，必须阻塞一个周期，等待数据从 MEM 段产生后通过转发获取。

2. **跳转指令依赖 EX 段 ALU 结果**: 当 ID 段的跳转/分支指令需要使用 EX 段的 ALU 计算结果时，需要阻塞一个周期。虽然 EX 段的结果会在下个周期进入 MEM 段，可

以通过转发提供给 ID 段，但由于跳转判断需要在 ID 段完成，因此必须等待一个周期。

3. **跳转指令依赖 MEM 段 load 结果**: 当 ID 段的跳转/分支指令需要使用 MEM 段 load 指令的结果时，需要阻塞一个周期，等待数据进入 WB 段后通过转发获取。

4. **控制冒险**: 当跳转确实发生时（`jump_flag` 为真且无阻塞），需要清空 IF2ID 流水线寄存器，丢弃已取出的错误路径指令。

5. **阻塞时插入气泡**: 当发生阻塞时，需要清空 ID2EX 流水线寄存器，在 EX 段插入一条 NOP 指令（气泡），防止 ID 段的指令被重复执行。

4.3 转发单元（Forwarding）

4.3.1 功能

转发单元负责检测数据冒险，并生成转发控制信号，使得 EX 或 ID 段可以直接从流水线寄存器中获取所需数据，而不必等待数据写回寄存器组，从而减少流水线阻塞。

4.3.2 代码实现（最终版本）

```
object ForwardingType {
  val NoForward = 0.U(2.W)
  val ForwardFromMEM = 1.U(2.W)
  val ForwardFromWB = 2.U(2.W)
}

class Forwarding extends Module {
  val io = IO(new Bundle() {
    val rs1_id = Input(UInt(Parameters.PhysicalRegisterAddrWidth))
    val rs2_id = Input(UInt(Parameters.PhysicalRegisterAddrWidth))
    val rs1_ex = Input(UInt(Parameters.PhysicalRegisterAddrWidth))
    val rs2_ex = Input(UInt(Parameters.PhysicalRegisterAddrWidth))
    val rd_mem = Input(UInt(Parameters.PhysicalRegisterAddrWidth))
    val reg_write_enable_mem = Input(Bool())
    val rd_wb = Input(UInt(Parameters.PhysicalRegisterAddrWidth))
    val reg_write_enable_wb = Input(Bool())

    val reg1_forward_id = Output(UInt(2.W))
```

```
    val reg2_forward_id = Output(UInt(2.W))
    val reg1_forward_ex = Output(UInt(2.W))
    val reg2_forward_ex = Output(UInt(2.W))
  })

  // Lab3(Final)
  val ex_mem_hazard_rs1 = io.reg_write_enable_mem && io.rd_mem =/= 0.U
   && (io.rd_mem === io.rs1_ex)
  val ex_mem_hazard_rs2 = io.reg_write_enable_mem && io.rd_mem =/= 0.U
   && (io.rd_mem === io.rs2_ex)
  val ex_wb_hazard_rs1 = io.reg_write_enable_wb && io.rd_wb =/= 0.U &&
   (io.rd_wb === io.rs1_ex)
  val ex_wb_hazard_rs2 = io.reg_write_enable_wb && io.rd_wb =/= 0.U &&
   (io.rd_wb === io.rs2_ex)
  io.reg1_forward_ex := Mux(ex_mem_hazard_rs1, ForwardingType.
   ForwardFromMEM,
    Mux(ex_wb_hazard_rs1, ForwardingType.ForwardFromWB, ForwardingType.
   NoForward))
  io.reg2_forward_ex := Mux(ex_mem_hazard_rs2, ForwardingType.
   ForwardFromMEM,
    Mux(ex_wb_hazard_rs2, ForwardingType.ForwardFromWB, ForwardingType.
   NoForward))

  val id_mem_hazard_rs1 = io.reg_write_enable_mem && io.rd_mem =/= 0.U
   && (io.rd_mem === io.rs1_id)
  val id_mem_hazard_rs2 = io.reg_write_enable_mem && io.rd_mem =/= 0.U
   && (io.rd_mem === io.rs2_id)

  val id_wb_hazard_rs1 = io.reg_write_enable_wb && io.rd_wb =/= 0.U &&
   !id_mem_hazard_rs1 && (io.rd_wb === io.rs1_id)
  val id_wb_hazard_rs2 = io.reg_write_enable_wb && io.rd_wb =/= 0.U &&
   !id_mem_hazard_rs2 && (io.rd_wb === io.rs2_id)
  io.reg1_forward_id := Mux(id_mem_hazard_rs1, ForwardingType.
   ForwardFromMEM,
    Mux(id_wb_hazard_rs1, ForwardingType.ForwardFromWB, ForwardingType.
   NoForward))
  io.reg2_forward_id := Mux(id_mem_hazard_rs2, ForwardingType.
```

```
    ForwardFromMEM,
     Mux(id_wb_hazard_rs2, ForwardingType.ForwardFromWB, ForwardingType.
    NoForward))

  // Lab3(Final) End
}
```

Listing 3: 转发单元实现（缩短分支延迟版本）

4.3.3 设计要点

转发单元需要同时处理到 EX 段和到 ID 段的转发：

1. **EX 段转发**:
   - 如果 EX 段的源寄存器与 MEM 段的目标寄存器相同，从 MEM 段转发。
   - 如果 EX 段的源寄存器与 WB 段的目标寄存器相同，从 WB 段转发。
   - MEM 段转发优先级高于 WB 段转发（保证获取最新的值）。
2. **ID 段转发**:
   - 由于跳转指令在 ID 段执行，需要将 MEM 和 WB 段的结果转发到 ID 段。
   - 如果 ID 段的源寄存器与 MEM 段的目标寄存器相同，从 MEM 段转发。
   - 如果 ID 段的源寄存器与 WB 段的目标寄存器相同（且不与 MEM 段冲突），从 WB 段转发。
   - MEM 段转发优先级同样高于 WB 段转发。
3. **寄存器 x0 的特殊处理**:
   - RISC-V 中寄存器 x0 恒为 0，写入 x0 的结果会被丢弃。
   - 因此，转发逻辑中需要检查目标寄存器是否为 0（`rd =/= 0.U`），避免不必要的转发。

4.4 译码单元（InstructionDecode）

在缩短分支延迟的版本中，译码单元需要在 ID 段完成分支和跳转指令的执行。

4.4.1 关键代码

```
val reg1_data = MuxLookup(io.reg1_forward, io.reg1_data)(
  Seq(
    ForwardingType.ForwardFromMEM -> io.forward_from_mem,
    ForwardingType.ForwardFromWB -> io.forward_from_wb
  )
)
val reg2_data = MuxLookup(io.reg2_forward, io.reg2_data)(
  Seq(
    ForwardingType.ForwardFromMEM -> io.forward_from_mem,
    ForwardingType.ForwardFromWB -> io.forward_from_wb
  )
)

val is_jump_instruction = opcode === Instructions.jal || opcode ===
   Instructions.jalr || opcode === InstructionTypes.B
io.ctrl_jump_instruction := is_jump_instruction

val jump_condition_met =
  (opcode === Instructions.jal) ||
    (opcode === Instructions.jalr) ||
    (opcode === InstructionTypes.B) && MuxLookup(
      funct3,
      false.B,
      IndexedSeq(
        InstructionsTypeB.beq -> (reg1_data === reg2_data),
        InstructionsTypeB.bne -> (reg1_data =/= reg2_data),
        InstructionsTypeB.blt -> (reg1_data.asSInt < reg2_data.asSInt),
        InstructionsTypeB.bge -> (reg1_data.asSInt >= reg2_data.asSInt)
   ,
        InstructionsTypeB.bltu -> (reg1_data < reg2_data),
        InstructionsTypeB.bgeu -> (reg1_data >= reg2_data)
      )
    )

val jump_address = Mux(
```

```
  opcode === Instructions.jalr,
  (reg1_data.asSInt + io.ex_immediate.asSInt).asUInt,
  (io.instruction_address.asSInt + io.ex_immediate.asSInt).asUInt
) & (~1.U(Parameters.DataWidth)).asUInt

io.if_jump_flag := jump_condition_met || io.interrupt_assert
io.if_jump_address := Mux(
  io.interrupt_assert,
  io.interrupt_handler_address,
  jump_address
)
```

Listing 4: ID 段跳转逻辑

4.4.2 设计要点

- 在 ID 段使用转发逻辑获取寄存器数据，确保使用的是最新的值。
- 根据指令类型（jal/jalr/分支）判断跳转条件是否满足。
- 在 ID 段计算跳转目标地址，无需等到 EX 段。
- 跳转地址需要与 `~1` 进行与运算，确保地址为偶数（RISC-V 要求）。

5 CSR 指令的冒险分析

5.1 数据冒险

CSR（Control and Status Register）指令用于读写控制和状态寄存器。在本实验的实现中，CSR 指令可能产生以下数据冒险：

1. **RAW（Read After Write）冒险**:
   - 当后续指令需要读取 CSR 指令写入的通用寄存器时，会发生 RAW 冒险。
   - 例如：`csrrw x1, mstatus, x2` 后跟 `add x3, x1, x4`。
   - 解决方法：与普通指令相同，通过转发或阻塞解决。CSR 指令在 WB 段将结果写回通用寄存器，可以通过 MEM-EX 和 WB-EX 转发路径提供数据。
2. **CSR 寄存器的 RAW 冒险**:

- 当连续的 CSR 指令访问同一个 CSR 寄存器时，后续指令可能读取到过时的值。
- 例如：`csrrw x1, mstatus, x2` 后跟 `csrrs x3, mstatus, x4`。
- 本实验未专门处理 CSR 寄存器间的冒险。在实际实现中，可以通过以下方式处理：
  - 检测 CSR 地址冲突，插入阻塞。
  - 为 CSR 单元添加转发逻辑。
  - 简化方案：CSR 指令较少出现连续访问，可以通过编译器重排指令避免。

5.2 控制冒险

CSR 指令本身不产生控制冒险（它们不是跳转或分支指令）。但是，某些 CSR 指令（如 `mret`）会改变程序执行流：

- `mret` 指令用于从异常处理返回，会跳转到 `mepc` 寄存器指定的地址。
- 这类指令在本实验中被当作特殊的跳转指令处理，会触发流水线清空。

5.3 结论

- CSR 指令写入通用寄存器的数据冒险可以通过现有的转发和阻塞机制解决。
- CSR 寄存器间的 RAW 冒险在本实验中未专门处理，实际应用中需要额外的检测和阻塞逻辑。
- `mret` 等特殊 CSR 指令产生的控制冒险可以通过流水线清空解决。

6 测试结果

本实验通过了所有测试用例，包括：

- **流水线寄存器测试**: 验证了流水线寄存器的阻塞和清空功能。
- **三级流水线 CPU 测试**: 验证了控制冒险的处理，包括递归计算斐波那契数列、快速排序、单字节加载存储等。
- **五级流水线（阻塞）CPU 测试**: 验证了使用阻塞解决数据冒险的正确性。
- **五级流水线（转发）CPU 测试**: 验证了使用转发减少阻塞的优化效果。

- **五级流水线（缩短分支延迟）CPU 测试**: 验证了将跳转提前到 ID 段的优化。

所有测试均在 60 秒内完成，共 20 个测试用例全部通过，验证了流水线 CPU 各阶段实现的正确性。

7 遇到的问题与改进建议

7.1 遇到的问题

1. **问题：转发逻辑的优先级容易混淆。**

   在实现转发单元时，需要处理 MEM 段和 WB 段同时满足转发条件的情况。根据流水线原理，应优先使用 MEM 段的数据（更新），但在编写代码时容易写反优先级。

   - **解决方法：**仔细分析数据流，绘制流水线状态图，明确各阶段数据的新旧关系。使用嵌套的 `Mux` 语句时，外层 `Mux` 判断优先级更高的条件。

2. **问题：跳转指令提前到 ID 段后，冒险情况增多。**

   将跳转判断从 EX 段提前到 ID 段后，需要考虑 ID 段的跳转指令与 EX、MEM 段指令的依赖关系，冒险检测逻辑变得更加复杂。

   - **解决方法：**按照实验指导的提示，列出所有可能的冒险组合表格，逐一分析每种情况，确保覆盖所有冒险场景。

3. **问题：调试困难，波形图信号繁多。**

   五级流水线 CPU 信号众多，在 GTKWave 中查看波形时很难快速定位错误信号。

   - **解决方法：**使用 Chisel 的 `printf` 调试功能，在关键部件（如寄存器文件、控制单元）添加打印语句，输出关键信号的值。结合简单的测试程序（如 `sb.S`），逐步排查错误。

7.2 改进建议

1. **建议：提供更多调试案例和方法指导。**

   实验指导中的调试部分较为简略，建议增加具体的调试案例，例如如何追踪一条指令在流水线中的完整执行过程，如何分析波形图定位冒险问题等。

2. **建议：增加可视化工具。**

   流水线状态可视化对理解和调试非常有帮助。建议提供或推荐一些工具，能够将流水线各阶段的指令和数据流以图形化方式展示。

3. **建议：补充 CSR 指令冒险处理的说明。**

   实验指导中未详细讨论 CSR 指令的冒险处理，建议在后续版本中补充相关内容，或明确说明本实验中 CSR 指令的简化假设。

8 实验结论

通过本次实验，我成功实现了三级和五级流水线 CPU，深入理解了流水线技术和竞争冒险的处理方法。在实现过程中，我掌握了：

- 流水线寄存器的设计与实现。
- 控制冒险的检测与清空机制。
- 数据冒险的阻塞和转发解决方案。
- 分支延迟优化技术（将跳转提前到 ID 段）。

通过编写和分析测试用例，我学会了如何验证流水线 CPU 的正确性，并掌握了使用打印和波形图进行调试的方法。本次实验使我对计算机组成原理中的流水线技术有了更深入的实践理解，为后续更复杂的处理器设计奠定了坚实的基础。