

# Verilog Pipeline CPU Design Document

## 项目概述

本项目为计算机组成原理 P6 实验作业，实现了一个基于 MIPS 指令集的 **五级流水线 CPU**。设计采用 Verilog HDL 编写，强调模块化、高内聚低耦合的设计原则，并保留了良好的拓展性以支持后续指令添加。

## 设计要求

### 1. 指令集支持

核心指令集包括：

- 算术运算: `add, sub, addi`
- 逻辑运算: `ori, and, or, lui`
- 访存指令: `lw, sw` (扩展支持 `lb, lh, sb, sh`)
- 分支跳转: `beq, bne, jal, jr`
- 乘除法: `mult, multu, div, divu, mfhi, mflo, mthi, mtlo` (MDU模块)
- 空指令: `nop`

### 2. 存储器规格

- 指令存储器 (IM): 容量 16KB (32bit × 4096 字)，PC 初始地址 `0x00003000`。
- 数据存储器 (DM): 容量 12KB (32bit × 3072 字)，支持按字节/半字/字访问。
- 字节序: 小端序 (Little Endian)。

### 3. 顶层模块接口

顶层模块名为 `mips`，遵循标准评测接口定义：

```
module mips(
    // input api
    input clk,
    input reset,
    input [31:0] i_inst_rdata,      // Instruction
    input [31:0] m_data_rdata,      // Read Data

    // output api
    output [31:0] i_inst_addr,      // PC addr
    output [31:0] m_data_addr,      // Memo addr
    output [31:0] m_data_wdata,      // Memo data
    output [3:0] m_data_byteen,      // Memo byte enable
    output [31:0] m_inst_addr,      // Memo stage PC
    output w_grf_we,                // GRF write enable
    output [4:0] w_grf_addr,        // GRF write addr
    output [31:0] w_grf_wdata,        // GRF write data
    output [31:0] w_inst_addr       // write stage PC
);
```

## 模块详解

CPU 采用标准的五级流水线架构 : **F (Fetch)** -> **D (Decode)** -> **E (Execute)** -> **M (Memory)** -> **W (Writeback)**。

### 1. 取指阶段 (Module I)

- **PC (Program Counter)**: 负责维护当前指令地址。支持同步复位和冻结 (Stall)。
- **逻辑**: 下一条地址 **NPC** 由 D 级计算并回传。
- **接口**: 接收 **h\_stall\_I** 信号, 当发生 Load-Use 冒险时冻结 PC。

### 2. 译码阶段 (Module D)

- **GRF (General Register File)**: 通用寄存器堆, 实现了**内部转发** (Write-Through), 即在同一时钟沿写入的数据可即时读出。
- **NPC (Next PC)**: 负责计算下一条指令地址。支持顺序执行 **PC+4**、分支跳转 **Branch**、链接跳转 **JAL** 以及寄存器跳转 **JR**。
- **CMP (Comparator)**: 所有的分支判定 (**beq**, **bne** 等) 均在 D 级完成, 以减少分支延迟槽的影响, 接收来自 Hazard 单元的转发数据 **hv\_fwd1\_D** 和 **hv\_fwd2\_D**, 确保判定时使用最新数据。
- **Extender**: 支持 Zero-Extend 和 Sign-Extend。

### 3. 执行阶段 (Module E)

- **ALU**: 算术逻辑单元, 负责常规计算及地址计算。
- **MDU (Multiply Divide Unit)**: 独立的乘除法单元, 支持 **start** / **busy** 握手信号。当 MDU 忙时, 会向 Hazard 发出 Stall 请求。
- **Store Data Forwarding**: 将要写入内存的数据 (**rt**) 传递给 M 级。
- **高阻态优化**: 针对 Load 指令, E 级输出结果 **v\_WB\_EM** 设为 **32'bz**, 配合 Hazard 单元进行精准的 Load-Use 冒险检测。

### 4. 访存阶段 (Module M)

- **Store Alignment**: 针对 **sb** (Store Byte) 和 **sh** (Store Halfword) 指令实现了**数据镜像逻辑**。
- 将寄存器低位数据复制到 **v\_MEM\_wdata** 的所有对应字节通道, 配合 **byteen** 信号实现任意地址对齐写入。
- **Load Extension**: 针对 **lb**, **lh** 指令, 根据地址低两位进行截取和符号扩展。
- **Forwarding Source**: 若当前指令不是 Load 类型, M 级的计算结果可直接转发回 D 级或 E 级。

### 5. 回写阶段 (Module W)

- **功能**: 将最终结果写入 GRF。
- **Trace**: 输出用于评测机比对的写回信息。

### 6. 冒险控制单元 (Hazard Unit)

本设计采用 **集中式冒险控制**。

- **转发策略 (Forwarding)**: 遵循“最新数据优先”原则 (E > M > W)。

- **E -> D / M -> D**: 用于 D 级分支判定。
- **M -> E / W -> E**: 用于 ALU 运算数。
- **W -> M**: 用于 Store 指令的数据转发。
- **暂停策略 (Stall)**:
- **Load-Use**: 当 D 级分支或 E 级运算需要的数据，正由上一条 Load 指令在 E/M 级读取时，触发 Stall。
- **MDU Busy**: 当 MDU 处于 Busy 状态且 D 级为 MDU 相关指令时，触发 Stall。
- **优先级修复**: 实现了 `&& ~h_stall_E` 逻辑，确保当 E 级暂停时，D 级不会错误地发出 Flush 信号，防止流水线数据丢失。

## 7. 控制单元 (Controller)

- 采用集中式译码，但在各级模块中分布实例化，因此已算是分布式译码
- 负责生成 `ALUOp`, `MemWrite`, `RegWrite`, `Branch` 等控制信号。
- 负责生成 `c_storeSign` 和 `c_loadSign` 以支持半字/字节指令。

---

# 关键技术点

## 1. 解决 Load-Use 冒险

通过检测 E 级或 M 级是否为 Load 指令（利用 `v_WB == 32'b1` 特性），当 D 级急需该数据时：

- 冻结 PC 和 IF/ID 寄存器。
- 清空 ID/EX 寄存器（插入 NOP）。

## 2. 支持延迟槽

- 分支跳转判定在 D 级完成。
- 分支指令不 Flush 紧随其后的指令（延迟槽），确保延迟槽指令被正确执行。

## 3. 分支预测的正确性

- 修复了 M 级 Load 指令向 D 级转发的 Bug：当 M 级是 Load 指令时，数据尚未从内存读出，禁止转发，强制 Stall D 级，防止分支指令使用错误的地址数据进行比较。

## 4. 字节/半字存取

- **Store**: 实现了数据位宽复制，解决了 `sb` 指令写入高位地址时的 `0x00` 错误。
- **Load**: 实现了符号扩展和移位逻辑，支持 `1b`, `1h` 的正确读取。

---

# 其他信息

## 编译与运行

项目支持在 ISE、Vivado 及 VSCode (配合 Icarus Verilog) 环境下运行。

## VSCode 调试命令:

```
"{VScode path}\Code.exe" -r . -r -g $1:$2
```

## 目录结构:

- `mips.v`: 顶层模块
- `Module*.v`: 各流水级模块
- `Reg*.v`: 流水线寄存器
- `Hazard.v`: 冒险控制器
- `Controller.v`: 主控制器
- `GRF.v, ALU.v, MDU.v, DM.v, IM.v`: 基础部件

## 思考题答案

这是一份针对你的 P5 流水线 CPU 项目的思考题答案汇总。这些答案结合了我们之前调试过程中发现的具体问题（如 Store 字节对齐、M 级 Load 转发问题、MDU 握手等），不仅符合一般理论，更契合你的代码实现。

## 思考题答案汇总

1. 为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器？

### 单独乘除法部件 (MDU) :

1. **时序与临界路径**：ALU 主要是单周期的组合逻辑（加减、逻辑运算），能够在极短时间内完成。而乘除法（尤其是除法）通常需要多个时钟周期才能完成（迭代试商法等）。如果整合进 ALU，会极大地拉长 ALU 的关键路径，导致整个 CPU 的时钟频率大幅下降，拖累所有指令的执行速度。2. **并行性**：将 MDU 独立出来，可以在 MDU 进行长周期运算时，流水线继续执行后续不相关的 ALU 指令（乱序执行的基础），提高了流水线的吞吐率。

**独立 HI、LO 寄存器：**1. **结果位宽**：MIPS 32 位架构中，两个 32 位数相乘结果为 64 位，除法产生 32 位商和 32 位余数。通用寄存器 (GPR) 只有 32 位，无法一次性容纳结果。2. **端口限制**：如果将结果写入 GPR，需要同时写入两个寄存器，这会增加寄存器堆 (GRF) 的写端口需求，大幅增加硬件复杂度和面积。使用专用的 HI/LO 寄存器可以规避对 GRF 端口的占用。

2. 真实的流水线 CPU 是如何使用实现乘除法的？请查阅相关资料进行简单说明

在真实的现代高性能 CPU 中：

1. **硬件流水线化**：乘法器通常采用 **Wallace 树** 或 **Booth 编码** 配合流水线寄存器实现。虽然延迟 (Latency) 可能需要 3-4 个周期，但吞吐率 (Throughput) 可以达到 1 个周期（即每个周期都能发射一条新乘法指令）。2. **非阻塞与乱序执行**：乘除法单元通常作为独立的执行端口。发射队列 (Issue Queue) 将指令分发给 MDU 后，不会阻塞流水线，后续无关指令可以继续发射和执行。结果计算完成后，通过 **重排序缓冲区 (ROB)** 或 **Common Data Bus (CDB)** 写回。3. **除法优化**：除法通常难以完全流水线化，延迟较高（如 20+ 周期）。现代 CPU 可能使用基 4 或基 8 的 SRT 算法，或者牛顿-拉夫逊迭代法来加速。

3. 请结合自己的实现分析，你是如何处理 Busy 信号带来的周期阻塞的？

在我的设计中，MDU 模块输出 `start` 信号表示运算开始，输出 `busy` 信号表示运算正在进行。处理方式主要在 **Hazard 模块** 中实现：

1. **检测条件**：`Hazard.v` 持续监测 `((start || busy) && MDU)` 信号。即当 MDU 正在忙碌（或刚开始），且当前 D 级译码出的指令是需要使用 MDU 的指令（如 `mult`, `div`, `mfhi`, `mflo` 等）时，判定为冲突。2. **阻塞行为**：将 `h_stall_D` 信号置为 1，冻结 IF/ID 和 PC 寄存器，保持 D 级指令不变。同时 `h_flush_DE` 置为 1，向 E 级插入气泡（NOP），防止错误数据进入流水线。一旦 MDU 运算结束，`busy` 拉低，Stall 解除，D 级指令进入 E 级读取结果。

4. 请问采用字节使能信号的方式处理写指令有什么好处？（提示：从清晰性、统一性等角度考虑）

1. **接口统一性**：无论执行 `sw`、`sh` 还是 `sb`，数据存储器（DM）的写数据接口 `wdata` 始终保持 32 位宽。我们不需要为字节或半字设计单独的物理连线或存储块。

2. **逻辑清晰与解耦**：**控制与数据分离**：`byteen` 专管“写哪里”，`wdata` 专管“写什么”。**简化存储器设计**：存储器内部只需根据 `byteen` 的掩码决定是否更新对应的 8 位存储单元，无需关心指令是 `sb` 还是 `sw`。这使得存储器模块（如 Block RAM）可以做成通用的 IP 核。3. **避免“读-改-写”（Read-Modify-Write）**：如果没有字节使能，要修改一个字中的某个字节，CPU 必须先读出整个字，修改对应字节，再写回整个字（耗时 2 周期）。使用 `byteen` 可以单周期完成写入，效率更高。

5. 请思考，我们在按字节读和按字节写时，实际从 DM 获得的数据和向 DM 写入的数据是否是一字节？在什么情况下我们按字节读和按字节写的效率会高于按字读和按字写呢？

**实际数据位宽：**

**不是一字节，而是 32 位（一字）。** **写时**：CPU 将寄存器的低 8 位数据镜像复制到 32 位数据线的所有字节位置（如 `0x78787878`），通过 `byteen` 选通写入。**读时**：DM 输出包含目标字节的完整 32 位字，CPU 根据地址低两位截取并进行符号/零扩展。

**效率比较**：在 **32 位宽总线** 的架构下（如本实验），按字节读写的效率与按字读写 **相同**（都是 1 个时钟周期）。**效率更高的情况**：1. **总线宽度受限**：如果外部数据总线只有 8 位宽，读写字节只需 1 次传输，而读写字需要 4 次。2. **避免 RMW**：如前所述，如果硬件不支持字节使能，修改字节需要“读-改-写” 2 个周期；支持字节使能（按字节写）则只需 1 个周期，此时效率高于“模拟的字节写”。

6. 为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？

1. **模块化设计**：将流水线分为 `ModuleI ~ ModuleM` 五个独立模块，通过流水线寄存器互联。

**帮助**：每个模块只需关注当前级的逻辑，降低了心智负担。

2. **集中式冒险控制 (Hazard Unit)**：

将分散在各级的暂停、冲刷、转发逻辑剥离出来，统一在 `Hazard.v` 中处理。特点：只需在 Hazard 单元中列出真值表（如 `Branch @ D vs Load @ E`），即可统一管理 `stall` 和 `flush` 信号，避免了逻辑分散导致的死锁或漏判。

3 **控制器抽象**：使用 `Controller` 模块，将 Opcode/Funct 翻译为 `c_ALU_op`, `c_branch` 等语义明确的控制信号。

帮助：在处理冲突时，我们只需判断 `is_load` 或 `use_rs` 等抽象信号，而不需要去比对具体的二进制指令码。

4. 标准化接口命名：如 `i_inst_addr`, `m_data_wdata` 等，遵循顶层规范，便于调试和对接。

7. 在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？

冲突 1：Load-Use 导致的 Branch 误判

**描述：**当 `lw` 指令在 M 级，`beq` 指令在 D 级时，Hazard 单元错误地将 M 级计算出的内存地址转发给了 D 级进行比较，导致分支预测错误。**解决：**在 Hazard 单元中增加检测：当 M 级是 Load 指令时 (`v_WB_M == 32'b1000_0000_0000_0000_0000_0000_0000_0000`)，禁止转发并强制 Stall D 级，等待数据回写。**样例：**`lw $1, 0($0); beq $1, $0, label.`

冲突 2：Store Byte 数据未对齐

**描述：**执行 `sb` 指令写入非对齐地址（如 `offset=3`）时，只写入了 `00` 而非寄存器低位数据。**解决：**在 `ModuleM` 中增加数据镜像逻辑，将寄存器低 8 位/16 位复制填充到整个 32 位数据线。**样例：**`ori $t1, $0, 0xaa; sb $t1, 3($0)`，检查内存地址 3 是否为 `0xaa`。

冲突 3：多重暂停优先级

**描述：**当 E 级 Stall（如 Load-Use）和 D 级 Stall 同时触发时，D 级发出的 Flush 信号错误地清空了 E 级正在等待的数据。**解决：**引入 `&& ~h_stall_E` 作为 D 级 Stall/Flush 的前置条件，确保后级暂停时前级不会破坏数据。

## 8. 测试样例构造策略与随机性结合

我是通过“受限随机 (Constrained Random)”策略来构造测试样例的。

**完全随机的不足：**完全随机生成的指令流（如随机 Opcode 随机寄存器）虽然数量大，但在稀疏空间中很难命中**极端冒险情况**（如连续的三级数据依赖、特定延迟槽冲突）。且出错后难以定位问题指令。

**我的策略：**

1. **构造冲突模板**：预定义高风险的指令序列模板。

模板 A (Load-Use): `Load $r1 -> ALU $r1` 模板 B (Branch-Hazard): `ALU $r1 -> Branch $r1` 模板 C (Structural): `Mult -> Mflo -> Mult`

2. **随机填充**

在模板的框架下，随机生成寄存器编号、立即数和具体的操作码（如 `add` 变 `sub`）。

3. **结合随机性**

- 在两个冲突模板之间，插入随机数量的 `nop` 或无关指令，测试转发逻辑在不同距离下的正确性（E 级转发 vs M 级转发 vs W 级转发）。
- 随机生成内存读写地址，测试 `sb/sh/sw` 在不同地址偏移下的边界情况。

这种策略既保证了对核心冒险逻辑的强测，又利用随机性覆盖了寄存器组合和数据边界，比单纯随机更高效，比单纯手动构造更全面。