

体系结构第一次实验报告

姓名：禹相祐 学号：2312900 专业：计算机科学与技术

目录

一、实验目的	1
二、实验要求	2
三、任务一：修改 Source Code 的 BUG	2
(一) Bug 1: IF 级提前推进导致 PC/取指错位 (<code>pipeline_cpu.v</code>)	2
(二) Bug 2: 可写 \$0 破坏 ISA 语义 (<code>regfile.v</code>)	3
(三) Bug 3: SW/SB 写数据与写使能不一致 (<code>mem.v</code>)	3
四、任务二：新增指令说明	4
(一) <code>mult rs, rt</code> (有符号乘法, 结果进 HI/LO)	4
(二) <code>mflo rd</code> (把 LO 搬到通用寄存器)	5
(三) <code>mfhi rd</code> (把 HI 搬到通用寄存器)	6
(四) <code>mtlo rs</code> (把 rs 写入 LO)	6
(五) <code>mtli rs</code> (把 rs 写入 HI)	6
(六) <code>mfc0 rt, cs</code> (从 CP0 读到通用寄存器)	7
(七) <code>mtc0 rt, cd</code> (从通用寄存器写到 CP0)	7
(八) <code>syscall</code> (发起系统调用, 进入异常)	7
(九) <code>eret</code> (从异常返回)	8
五、任务二上箱的实验图片	8
六、实验总结	9

一、实验目的

1. 熟悉并掌握流水线 CPU 的原理和设计。
2. 检验运用 Verilog 进行电路设计的能力。
3. 通过亲自设计实现静态 5 级流水线 CPU, 加深对计算机组成原理与体系结构的理解。

二、 实验要求

1. 修改原始 source code 中的 bug，保证指令运行正确。
2. 针对五级流水线中新增加的指令，逐级分析其执行过程与结果，梳理流水线知识点。

三、 任务一：修改 Source Code 的 BUG

在阅读了源码以及实验指导书后，并且运行了如今的带 bug 的源码后，最后经由自己的努力以及同同学之间的讨论以及网上搜索后，最后发现一共有三处 BUG，分布在三个文件内，分别如下：

（一） Bug 1：IF 级提前推进导致 PC/取指错位 (pipeline_cpu.v)

错误写法：

```
1 assign IF_allow_in = (IF_over & ID_allow_in) | cancel;
```

为什么错：

- 同步 ROM 取指通常是“申请一拍，返回一拍”。IF_over 还没到就推进，PC 先变了，但返回的 inst 还是旧地址，对不齐。
- cancel 的作用是冲刷流水线和重定向 PC，不是“强行放行”。把 cancel 混进 allow_in, WB 一触发异常或返回, IF 就无条件前进一步, 容易出现“多取一条”“PC 看起来多跳了 8/12 字节”的怪现象。
- 破坏基本握手约定：推进要满足“本级数据就绪”over 和“下级能接”allow_in 两个条件同时成立。用外部的 cancel 越级干预，会把 valid/allow_in/over 的因果顺序打乱。
- 实际连锁反应：IF 早一步，ID 看到的是不完整或不匹配的 inst，分支目标、延迟槽和异常返回的下一条都会被带偏。

正确写法：

```
1 assign IF_allow_in = (IF_over & ID_allow_in);
```

为什么这么改：

- 保证 IF 只在“拿到有效指令”且 ID 准备好了时推进，严格贴合同步存储器的两拍节奏。
- cancel 继续做它该做的事：清 valid、选异常入口或 EPC 返回地址，但不参与“是否推进”的判定。这样 PC 和 inst 始终配对，不会脱钩。
- 直觉版总结：allow_in 只看“数据到了没、下级能不能接”，别再额外掺杂控制流信号。

(二) Bug 2: 可写 \$0 破坏 ISA 语义 (regfile.v)

错误写法:

```
1 always @(posedge clk) begin
2     if (wen) rf[waddr] <= wdata; // 未屏蔽 waddr==0
3 end
```

为什么错:

- MIPS 语义规定 \$0 恒为 0。若能写 \$0, 就会让 BEQ \$0,\$0,label 这类“永真”判断失效; 很多“清零、搬立即数”的序列也会表现出随机错误。
- 相关与转发逻辑通常默认“读到 \$0 就是 0”, 一旦被写脏, 前后级的旁路比较和停顿决策都会被污染, 出现“偶发一拍错”的诡异现象。
- 复位边界也有坑: 即使大多数时候不去写 \$0, 只要有一次误写或综合在边界插入了不期望的赋值, rf[0] 很难靠后续逻辑自动恢复为 0。

正确写法:

```
1 always @(posedge clk) begin
2     if (wen && (waddr != 5'd0)) rf[waddr] <= wdata; // 禁止写 $0
3     rf[0] <= 32'd0; // $0 每拍硬清零
4 end
```

为什么这么改:

- “屏蔽写入 + 每拍清零”是双保险: 从源头禁止写入 \$0, 同时每拍把 rf[0] 拉回到 0, 堵住边界态和综合细节带来的小概率污染。
- 这样能让分支比较、转发匹配和立即数操作都保持稳定, 完全符合课程里用到的 ISA 约定, 且对时序影响可忽略。
- 直觉版总结: 把 \$0 当成“硬连到地”的特殊寄存器, 任何时候读它都是 0。

(三) Bug 3: SW/SB 写数据与写使能不一致 (mem.v)

错误写法:

```
1 always @(*) begin
2     case (dm_addr[1:0])
3         2'b00: dm_wdata <= store_data;
4         2'b01: dm_wdata <= {16'd0, store_data[7:0], 8'd0};
5         2'b10: dm_wdata <= {8'd0, store_data[7:0], 16'd0};
6         2'b11: dm_wdata <= {store_data[7:0], 24'd0};
7     endcase
8 end
```

为什么错：

- 这段逻辑把所有存储都当成“单字节写”。对于 SW，本应写满 4 个字节且不改变字节顺序；结果被错当成“只写最低字节再移位”，读回自然乱序。
- 写使能和数据路径脱钩：SW 期望 4'b1111，SB 期望 $1 \ll \text{dm_addr}[1:0]$ 。数据只给了 1 字节的内容，但却可能开了 4 位写使能，等于“往 3 个字节写垃圾”或“对着空气写”。
- 容易被忽略的小点：不少 FPGA 的 BRAM 是按“字节通道”工作的，**数据放在哪个通道要和哪个通道被使能**严格对应，否则就会出现“看似偶发”的花屏式数据。

正确写法：

```

1  always @(*) begin
2      if (ls_word) begin
3          dm_wdata <= store_data; // SW: 直接写 32b
4      end else begin           // SB: 按字节对齐
5          case (dm_addr[1:0])
6              2'b00: dm_wdata <= store_data;
7              2'b01: dm_wdata <= {16'd0, store_data[7:0], 8'd0};
8              2'b10: dm_wdata <= {8'd0, store_data[7:0], 16'd0};
9              2'b11: dm_wdata <= {store_data[7:0], 24'd0};
10         endcase
11     end
12 end

```

为什么这么改：

- SW 直接把 32 位原样送出去，配合四位写使能，一次性写满；SB 只把目标字节放到对应通道，配合单比特写使能，做到“只改那一个字节”。
- 数据路径与写使能一一对应，字节序保持一致，后续无论按字读还是按字节读，结果都可预测。
- 直觉版总结：先想清“打开了哪些 byte lane”，再把数据**精确地**放进那些 lane。

四、 任务二：新增指令说明

新增 9 条指令如表 1。

下面我们就对这九条指令分别进行讲解：

(一) mult rs, rt (有符号乘法, 结果进 HI/LO)

- 作用：rs 与 rt 做有符号乘法，64 位结果的高 32 位进 HI，低 32 位进 LO。
- 分段

表 1: 新增指令列表

序号	指令名称	汇编形式
1	有符号乘法	<code>mult rs, rt</code>
2	从 LO 取值	<code>mflo rd</code>
3	从 HI 取值	<code>mfhi rd</code>
4	向 LO 存值	<code>mtlo rs</code>
5	向 HI 存值	<code>mthi rs</code>
6	从 CP0 取值	<code>mfc0 rt, cs</code>
7	向 CP0 存值	<code>mtc0 rt, cd</code>
8	系统调用	<code>syscall</code>
9	异常返回	<code>eret</code>

- IF: 正常取指。
- ID: 识别 `mult`, 读 `rs`、`rt`。
- EX: 多周期乘法, 未结束就不放行后继。
- MEM: 透传。
- WB: 写 HI 与 LO。

- 注意点

- 与后继 `mfhi`、`mflo` 有读后写；无前递则插泡到本条写回。
- 与 `mthi`、`mtlo` 改同一目标有写后写；以先到 WB 为准。

(二) `mflo rd` (把 LO 搬到通用寄存器)

- 作用: 把 LO 的值写进 `rd` ($rd \leftarrow LO$)。

- 分段

- IF: 正常。
- ID: 识别 `mflo`, 锁存 `rd`。
- EX/MEM: 透传。
- WB: 读 LO, 写 `rd`。

- 注意点

- 紧跟 `mult` 或 `mtlo` 时需等前条写回；通常插泡。

(三) mfhi rd (把 HI 搬到通用寄存器)

- 作用：把 HI 的值写进 rd ($rd \leftarrow HI$)。
- 分段
 - IF：正常。
 - ID：识别 mfhi, 锁存 rd。
 - EX/MEM：透传。
 - WB：读 HI, 写 rd。
- 注意点
 - 紧跟 mult 或 mthi 时等写回；无前递就插泡。

(四) mtlo rs (把 rs 写入 LO)

- 作用：用 rs 覆盖 LO ($LO \leftarrow rs$)。
- 分段
 - IF：正常。
 - ID：识别 mtlo, 读 rs。
 - EX：准备写入值，打标记。
 - MEM：透传。
 - WB：写 LO。
- 注意点
 - 后继 mflo 需等本条 WB。
 - 与 mult 写 LO 冲突时看 WB 到达先后。

(五) mthi rs (把 rs 写入 HI)

- 作用：用 rs 覆盖 HI ($HI \leftarrow rs$)。
- 分段
 - IF：正常。
 - ID：识别 mthi, 读 rs。
 - EX：准备写入值，打标记。
 - MEM：透传。
 - WB：写 HI。

- 注意点
 - 后继 mfhi 需等本条 WB。
 - 与 mult 写 HI 冲突时看 WB 到达先后。

(六) mfc0 rt, cs (从 CP0 读到通用寄存器)

- 作用：把 CP0 中编号为 cs 的寄存器读到 rt (如 EPC、Status、Cause)。
- 分段
 - IF：正常。
 - ID：识别 mfc0，解析 cs，锁存 rt。
 - EX/MEM：透传。
 - WB：读 CP0[cs]，写 rt ($rt \leftarrow CP0[cs]$)。
- 注意点
 - 若前序 mtc0 写同一寄存器，需隔一拍或做回写前递再读。

(七) mtc0 rt, cd (从通用寄存器写到 CP0)

- 作用：把 rt 写入 CP0 中编号为 cd 的寄存器 ($CP0[cd] \leftarrow rt$)。
- 分段
 - IF：正常。
 - ID：识别 mtc0，解析 cd，读 rt。
 - EX/MEM：透传。
 - WB：写 CP0[cd]。
- 注意点
 - 后继 mfc0 读同一寄存器需隔一拍或做回写前递。
 - 写 Status 的 EXL 或写 EPC 会改变异常进出路径。

(八) syscall (发起系统调用，进入异常)

- 作用：主动触发异常，跳到异常入口。
- 分段
 - IF/ID/EX/MEM：按普通指令推进。
 - WB：设置 Status.EXL 为 1，Cause 为 8，EPC 记当前地址；冲刷流水线并把取指重定向到异常入口。

- 注意点
 - 控制相关在 WB 生效；被冲刷的指令不产生副作用。

(九) eret (从异常返回)

- 作用：把控制流拉回 EPC 指向的位置，并清除 EXL。
- 分段
 - IF/ID/EX/MEM：按普通指令推进。
 - WB： $PC \leftarrow EPC$ ，清 Status.EXL；冲刷流水线后恢复执行。
- 注意点
 - 与 syscall 配对检查冲刷与重定向路径是否正确。

五、 任务二上箱的实验图片

下面是三张实验上箱的照片，对应的三条指令分别是 mfhi rd, mtlo rs 以及 mthi rs, 实验图如下：

LOONGSON	
IF PC:000000F4	IF IN:0000F010
ID PC:000000F0	EXEPC:000000EC
MEMPC:000000E8	WB PC:000000E8
MADDR:00000000	MDATA:00000000
VALID:000FFF00	
HI:00000000	LO:00000000
REG00:00000000	REG01:00000008
REG02:00000010	REG03:00000011
REG04:00000004	REG05:0000000D
REG06:FFFFFFE2	REG07:FFFFFFF3
REG08:00000011	REG09:00000000
REG0A:00000011	REG0B:000000C0
REG0C:000C0000	REG0D:00000000
REG0E:FFFFFFE20	REG0F:FFFFFFF88
REG10:FFFFFFF7F	REG11:00000000
REG12:00000000	REG13:00000000
REG14:0000FF88	REG15:FFFFFFF7F
REG16:FFFF0077	REG17:00000000
REG18:00000001	REG19:00000001
REG1A:0000000C	REG1B:00000050
REG1C:0000000D	REG1D:000C000D
REG1E:00000000	REG1F:00000084
START INPUTING	

图 1: mfhi rd

LOONGSON	
IF PC: 000000F8	IF IN: 0000F010
ID PC: 000000F4	EXPC: 000000F0
MEMPC: 000000EC	WB PC: 000000E8
MADDR: 00000002	MDATA: 00000008
VALID: 000FFFF0	
HI: 00000050	LO: 0000000C
REG00: 00000000	REG01: 00000008
REG02: 00000010	REG03: 00000011
REG04: 00000004	REG05: 0000000D
REG06: FFFFFFFE2	REG07: FFFFFFFF3
REG08: 00000011	REG09: 12000000
REG0A: 00000011	REG0B: 000000C0
REG0C: 000C0000	REG0D: 00000000
REG0E: FFFFFFFE20	REG0F: FFFFFFFF88
REG10: FFFFFFFF	REG11: 00000000
REG12: 00000000	REG13: 00000000
REG14: 0000FF88	REG15: FFFFFFFF
REG16: FFFF0077	REG17: 009C0000
REG18: 00000001	REG19: 00000001
REG1A: 0000000C	REG1B: 00000050
REG1C: 0000000D	REG1D: 000C000D
REG1E: 00000090	REG1F: 00000084

图 2: mtlo rs

LOONGSON	
IF PC: 00000100	IF IN: 03600011
ID PC: 000000FC	EXPC: 000000F8
MEMPC: 000000F8	WB PC: 000000F4
MADDR: 00000002	MDATA: 00000008
VALID: 000FF0F0	
HI: 00000090	LO: 009C0000
REG00: 00000000	REG01: 00000008
REG02: 00000010	REG03: 00000011
REG04: 00000004	REG05: 0000000D
REG06: FFFFFFFE2	REG07: FFFFFFFF3
REG08: 00000011	REG09: 12000000
REG0A: 00000011	REG0B: 000000C0
REG0C: 000C0000	REG0D: 00000000
REG0E: FFFFFFFE20	REG0F: FFFFFFFF88
REG10: FFFFFFFF	REG11: 00000000
REG12: 00000000	REG13: 00000000
REG14: 0000FF88	REG15: FFFFFFFF
REG16: FFFF0077	REG17: 009C0000
REG18: 00000001	REG19: 00000001
REG1A: 0000000C	REG1B: 00000050
REG1C: 0000000D	REG1D: 000C000D
REG1E: 00000090	REG1F: 00000084
START INPUTING	

图 3: mthi rs

六、 实验总结

本次流水线 CPU 实验使我深刻认识到理论设计与实际实现之间的差异。在调试 IF 级指令提前推进、\$0 寄存器写保护以及存储指令数据对齐这三个关键问题的过程中，我深入理解了流水线各阶段同步机制的重要性，并意识到硬件设计必须严谨考虑各种边界条件。

通过对九条新增指令的逐级分析，我从多周期乘法的特殊处理到 CP0 异常管理的实现机制，对 CPU 内部工作机制有了更全面的认识。

这次实验不仅深化了我对流水线原理的理解，更重要的是培养了系统级的调试思维：当遇到异常现象时，需要从整体架构出发，综合分析数据流与控制流的交互关系。整个调试过程虽然充满挑战，但最终取得的成果表明，这些付出都具有重要意义。