

体系结构第二次实验报告

姓名：梁朝阳 2311561 专业：密码科学与技术

目录

1 实验要求	2
2 旁路设计图	2
3 实验过程	3
3.1 设计思路（我的考虑与尝试）	3
3.1.1 前递阶段选取与转移计算未完成情况	3
3.1.2 具体判断逻辑	4
3.2 实现代码	5
3.2.1 旁路单元详细实现	5
3.2.2 总线传递与模块接口变化	7
3.3 仿真实验	8
3.3.1 周期变短验证	8
3.3.2 延迟情况验证	9
3.3.3 最后举例	10
4 实验总结与心得体会	11

摘要

设计思路与架构：旁路设计图见图 2.1，设计思路在后面详细讲述，这里只是简单介绍架构：首先创建独立的 `bypass_unit.v` 模块。主要的注意点在于优先级选取，和转移未完成的 `stall` 的检测。还有就是也对比了下我在国庆前的不完全正确实现（EXE 阶段前递），说明了为什么我选择了 ID 阶段前递。

代码实现：没有完全按照 CPU 设计实战中的建议，而是采用了我自己的实现方式，主要是为了简化逻辑，避免冒险修改级联控制中 `ready_go` 信号的复杂性（与出错的可能性），最后结果也是正确的。

仿真对比效果：从三个方面对比了仿真效果：整体直观上总周期变短；`branch` 与乘法延迟情况，最后从细节举例确实减少了周期的指令。

1 实验要求

请在第一次实验的五级流水线 CPU 的基础上，根据《CPU 设计实战》这本书上第四章后面的内容，验证现有 CPU 的指令相关和流水线冲突问题，以添加旁路的方式和前递技术，让 CPU 规避这样的问题。注意：

1. 实验报告中简要画一下旁路设计的示意图，有对应的介绍。
2. 进行详细分析，原始情况下指令是如何执行的，增加了旁路和前递之后是如何执行的。

2 旁路设计图

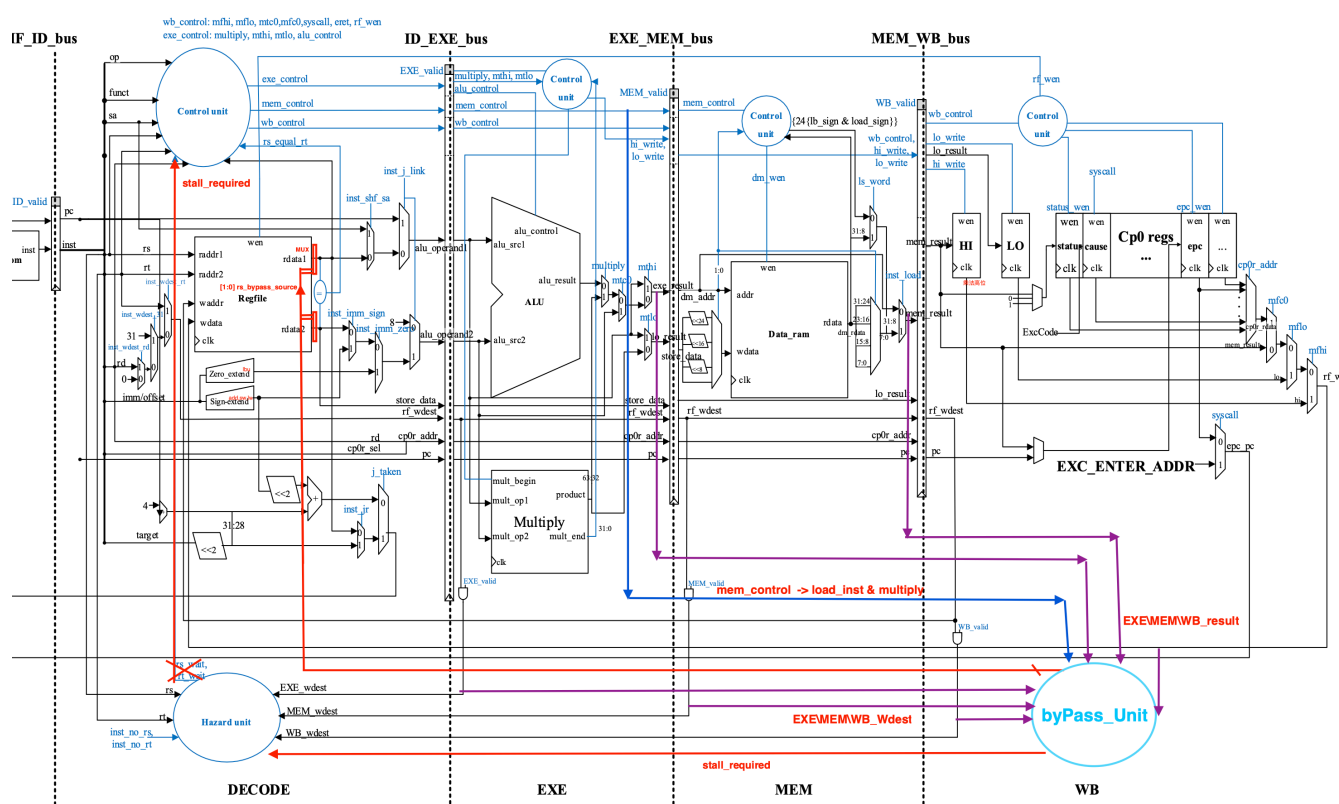


图 2.1: EXE/MEM/WB -> ID 旁路设计图

设计思路（为什么这么设计以及详细考量）在后面详细讲述，这里只是简单介绍架构：首先创建独立的 `bypass_unit.v` 模块，ID 选择操作数结果优先级：EXE > MEM > WB > 寄存器堆。且由于在 ID 阶段实现了 branch，因此具有「转移计算未完成 (load-branch)」这类特殊情况（需要 stall）。因此我的旁路内对类似这种情况¹进行了检测。

`bypass_unit` 考虑级联控制中的 `valid` 值接受 EXE/MEM/WB 阶段的写回地址 `Wdest` 与写回值 (`result`)。同时接受 `mem_control` 信号中的判断类型信号。最终输出 `rs_bypass_source` 与 `stall_required`，替代 `wait` 信号。

¹实际上 CPU 设计书中说的不是特别全，还有乘法运算也同样需要检测。实际实现下来，我的做法是对所有 branch 前有运算相关性的都进行 stall，下文会详细叙述原因。

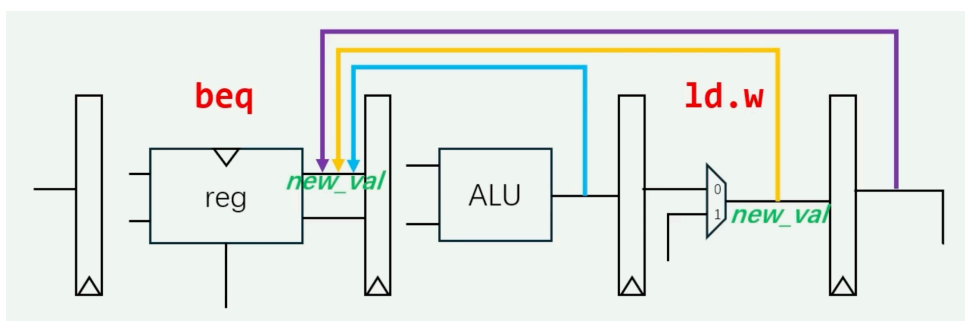


图 3.4: 正确实现

因此，正确实现应该是发出一个 stall 信号，使得 ld 指令执行到 MEM 阶段，这样就可以解除错误。

3.1.2 具体判断逻辑

Listing 1: 是否出现相关性

```

1 // RS寄存器旁路检测
2 wire exe_bypass_rs = EXE_valid & (rs != 5'd0) & (rs == EXE_wdest);
3 wire mem_bypass_rs = MEM_valid & (rs != 5'd0) & (rs == MEM_wdest) &
  ~exe_bypass_rs;
4 wire wb_bypass_rs = WB_valid & (rs != 5'd0) & (rs == WB_wdest) &
  ~exe_bypass_rs & ~mem_bypass_rs;
5 // 同理于rt寄存器

```

如果后面的阶段有效，且不是 0 寄存器，且写回地址与 rs/rt 相同，则认为出现相关性。

Listing 2: 旁路源编码（优先级检测）

```

1 output [1:0] rs_bypass_source, // rs旁路源 (00:无, 01:EXE, 10:MEM, 11:WB)
2 output [1:0] rt_bypass_source // rt旁路源 (00:无, 01:EXE, 10:MEM, 11:WB)
3
4 // 旁路源编码
5 assign rs_bypass_source = exe_bypass_rs ? 2'b01 :
6     mem_bypass_rs ? 2'b10 :
7     wb_bypass_rs ? 2'b11 : 2'b00;
8 // 同理与rt

```

由于上文提及了，如果 EXE 与后面的阶段同时有相关性，则优先级：EXE > MEM > WB > 寄存器堆。具体设计为了 00 为无相关，01:EXE, 10:MEM, 11:WB。用于控制我在图 2.1 中画出的两个 MUX。

3.2 实现代码

3.2.1 旁路单元详细实现

Listing 3: 模块

```
1  module bypass_unit(  
2      // 当前指令信息  
3      input [4:0] rs,           // 源寄存器1地址  
4      input [4:0] rt,           // 源寄存器2地址  
5      input [31:0] rs_value,     // 源寄存器1原始值  
6      input [31:0] rt_value,     // 源寄存器2原始值  
7  
8      // 后续流水级信息  
9      input EXE_valid,          // EXE级有效  
10     input MEM_valid,          // MEM级有效  
11     input WB_valid,           // WB级有效  
12     input [4:0] EXE_wdest,     // EXE级写回目标寄存器  
13     input [4:0] MEM_wdest,     // MEM级写回目标寄存器  
14     input [4:0] WB_wdest,      // WB级写回目标寄存器  
15     input [31:0] EXE_result,   // EXE级结果  
16     input [31:0] MEM_result,   // MEM级结果  
17     input [31:0] WB_result,    // WB级结果  
18  
19     // 指令类型信息（用于特殊处理）  
20     input inst_load,           // 当前指令Load指令  
21     input inst_mult,           // 当前指令乘法指令  
22     input inst_mfhi,           // 当前指令MFHI指令  
23     input inst_mflo,           // 当前指令MFLO指令  
24     input inst_mfc0,           // 当前指令MFC0指令  
25  
26     // EXE级指令类型信息（用于Load-Use相关检测）  
27     input EXE_inst_load,        // EXE级Load指令  
28     input EXE_inst_mult,        // EXE级乘法指令  
29  
30     // 旁路结果输出  
31     output [31:0] bypassed_rs_value, // 旁路后的rs值  
32     output [31:0] bypassed_rt_value, // 旁路后的rt值  
33  
34     // 控制信号输出  
35     output stall_required,      // 需要阻塞信号  
36     output rs_bypass_valid,     // rs旁路有效  
37     output rt_bypass_valid,     // rt旁路有效
```

```

38     output [1:0] rs_bypass_source, // rs旁路源 (00:无, 01:EXE, 10:MEM, 11:WB)
39     output [1:0] rt_bypass_source // rt旁路源 (00:无, 01:EXE, 10:MEM, 11:WB)
40 );

```

虽然这个比较占地方，但是由于本次实验主要就是这个代码，因此依然还是把声明（所有接口）原封不动给出了。主要的特殊信息就是 load 和乘法累指令，还有协处理器指令:inst_mfc0。

Listing 4: 阻塞控制信号-产生

```

1 // 需要阻塞的情况
2 assign stall_required = load_use_hazard_rs | load_use_hazard_rt |
3     mult_use_hazard_rs | mult_use_hazard_rt;

```

需要阻塞的情况就是乘法和 load，协处理器情况不需要阻塞，因为都会 cancel 掉，所有 valid 值应该都在 cancel 信号的控制下。

Listing 5: 阻塞控制信号-细节

```

1 // Load-Use相关: EXE级Load指令的结果需要2个周期才能获得
2 wire load_use_hazard_rs = exe_bypass_rs & EXE_inst_load;
3 wire load_use_hazard_rt = exe_bypass_rt & EXE_inst_load;
4 // 多周期操作相关: EXE级乘法指令需要多周期完成
5 wire mult_use_hazard_rs = exe_bypass_rs & EXE_inst_mult;
6 wire mult_use_hazard_rt = exe_bypass_rt & EXE_inst_mult;

```

因此，具体实现的两个信号是这样的，主要有相关性，且 EXE 阶段（注意不是本阶段）的指令运行的事乘法或者 load，都需要 stall。

Listing 6: 阻塞控制信号-效果

```

1 // assign rs_wait = ~inst_no_rs & (rs!=5'd0) & ( (rs==EXE_wdest) |
2     (rs==MEM_wdest) | (rs==WB_wdest) );
3 // assign rt_wait = ~inst_no_rt & (rt!=5'd0) & ( (rt==EXE_wdest) |
4     (rt==MEM_wdest) | (rt==WB_wdest) );
5 assign ID_over = ID_valid & (~inst_jbr | IF_over) & ~stall_required;

```

删除掉原先的 rs/rt_wait 信号，而是用我新定义的 stall 来控制 ID 是否 over (ready_go)。如果 ID 没有 ready_go，那么由于：

Listing 7: 级联控制

```

1 assign IF_allow_in = (IF_over & ID_allow_in) | cancel;
2 assign ID_allow_in = ~ID_valid | (ID_over & EXE_allow_in);

```

所以 ID 阶段被阻塞，IF 阶段无法进入；这样就成功解决了错误情况（完成阻塞）。

3.2.2 总线传递与模块接口变化

最开始我的实现修改了总线信号，后来思考为了简便与清晰性，可以考虑单独抽出一些信号：

Listing 8: decoder 新添加的信号

```
1 // 旁路数据输入
2 input    [ 31:0] EXE_result, // EXE级结果
3 input    [ 31:0] MEM_result, // MEM级结果
4 input    [ 31:0] WB_result,  // WB级结果
5 // EXE级指令类型信息（用于Load-Use相关检测）
6 input          EXE_inst_load, // EXE级Load指令
7 input          EXE_inst_mult, // EXE级乘法指令
```

由于旁路模块是在 decoder 中实例化的，因此其他所有模块都应给 Decoder 传入相关的信号（如有效值与具体值）。这个具体值需要找到最最后面的，具有决定性的值，以 WB 阶段举例：

Listing 9: WB 阶段向前传递的 result

```
1 assign rf_wen = wen & WB_over;
2 assign rf_wdest = wdest;
3 assign rf_wdata = mfhi ? hi : mflo ? lo : mfc0 ? cp0r_rdata : mem_result;
4 assign WB_result = rf_wdata; // ! 旁路输出
```

除此之外，每个模块的输出也需要增加，以 EXE 为例：

Listing 10: EXE 模块新添加的信号

```
1 // 新增旁路数据输出
2 output    [ 31:0] EXE_result, // EXE级结果，用于旁路
3
4 // EXE级指令类型信息输出
5 output          EXE_inst_load, // EXE级Load指令
6 output          EXE_inst_mult, // EXE级乘法指令
```

最后可以举例说明值是如何被选择的，以 ALU 为例子：

```
assign alu_operand1 = inst_j_link ? pc :
    inst_shf_sa ? {27'd0, sa} :
    bypassed_rs_value;
assign alu_operand2 = inst_j_link ? 32'd8 :
    inst_imm_zero ? {16'd0, imm} :
    inst_imm_sign ? {{16{imm[15]}}, imm} :
    bypassed_rt_value;
```

图 3.5: alu 采用旁路值

3.3 仿真实验

3.3.1 周期变短验证

下面进行仿真实验，首先整体上可以看到周期变短了，且更加整齐，先前由于级联控制，不同的指令经常因为种种情况导致延迟，现在可以看到只有 Beq 指令可能遇到延迟（44H-48H）：

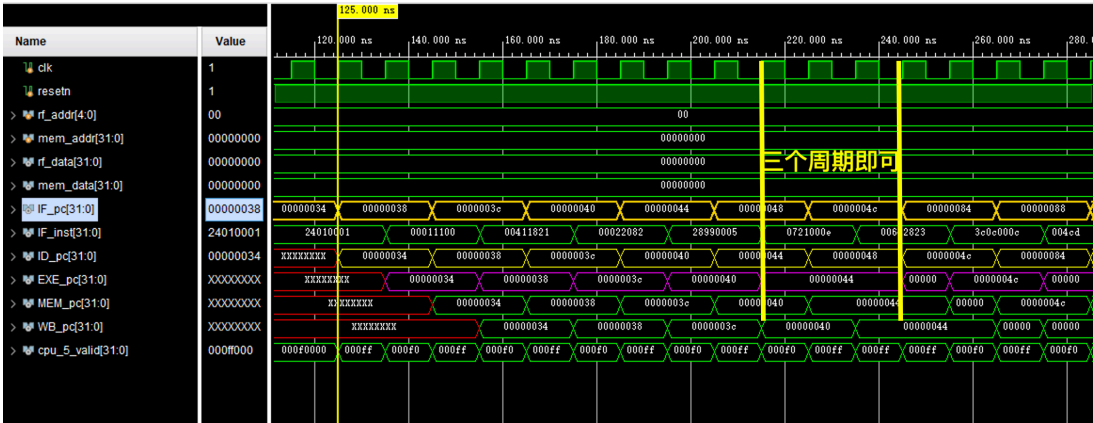


图 3.6: 明显看到周期变短

对比之前的，可以看到先前的 44 占据了 4 个周期，现在占据三个周期。且其余指令也都有长进（具体见本章最后的举例部分）。

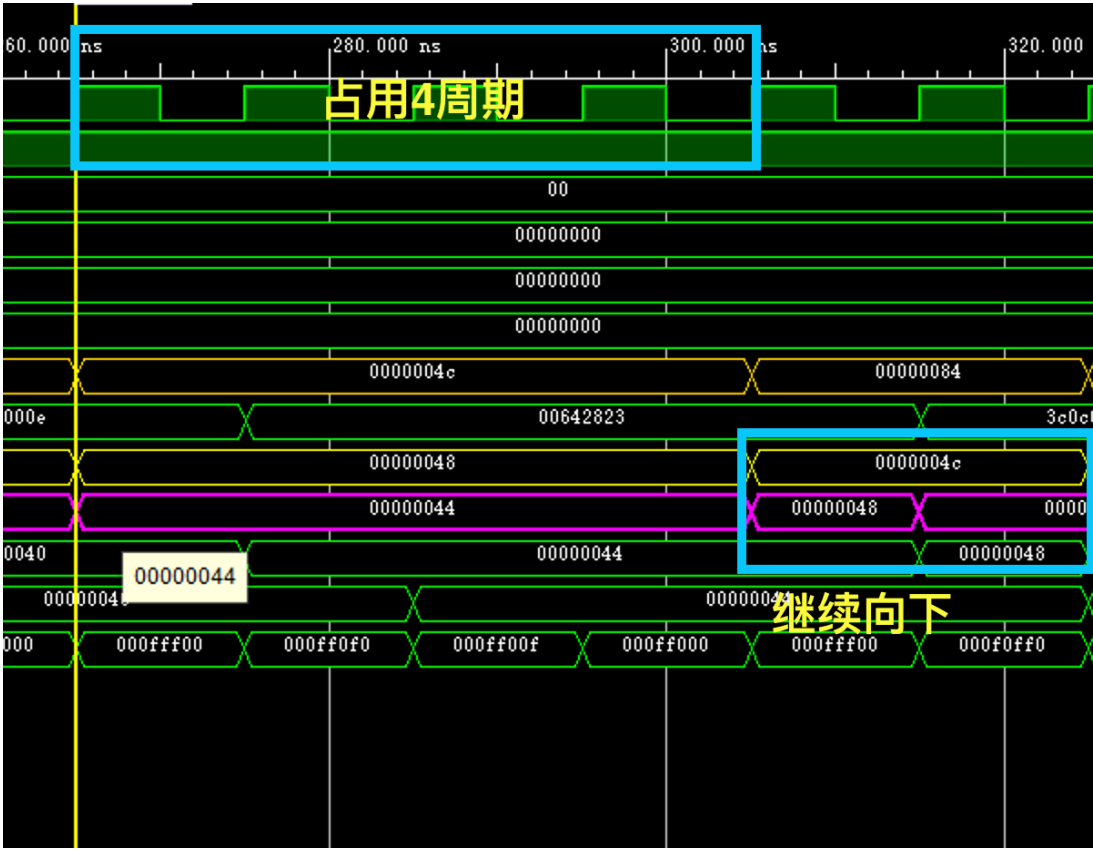


图 3.7: 先前的跳转 1 周期分析

3.3.2 延迟情况验证

延迟情况的话，就是 beq 指令部分和乘法部分：

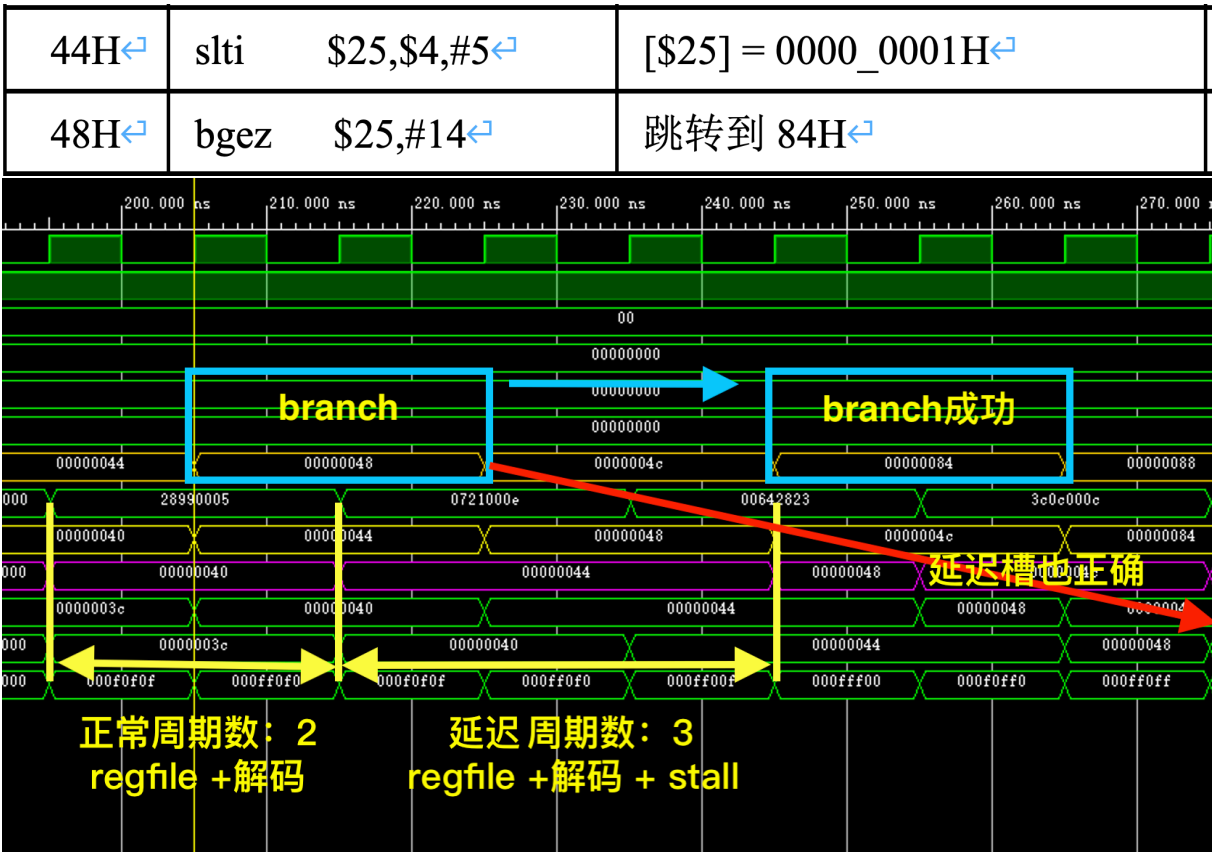


图 3.8: beq 的时候多延迟一个周期

48H 是 beq 指令，可以看到在他前面的具有相关性的指令多停留了一些，这符合我说的对 beq 指令的时候单独检测 stall 的逻辑。

可以对比一下最早我实现的 EXE 阶段前递版本：

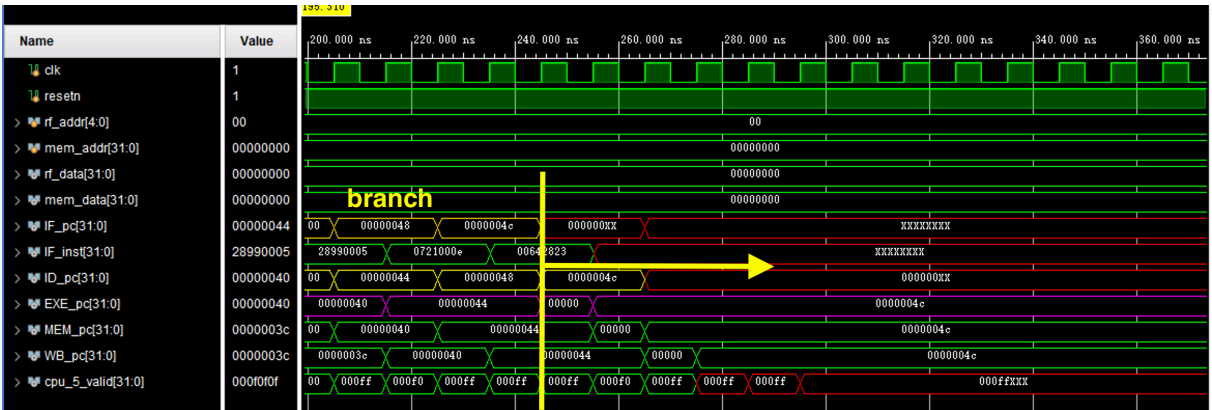


图 3.9: EXE 阶段前递版本（最开始的错误实现）

这里 branch 后，虽然前面正常执行，但是 branch 读到的数值没有定义（一个是有有效的值已经随流水走了，另一个是没有处理，即使读到了也是错误的）。

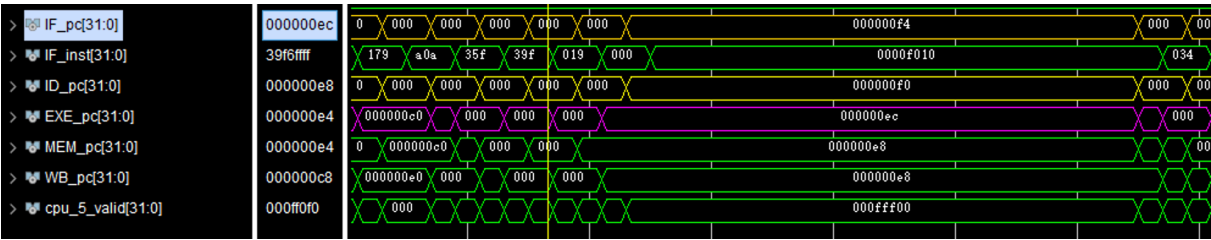


图 3.10: 乘法也可以正常工作

乘法情况，也可以正确控制，如上图，不再赘述。

3.3.3 最后举例

最后再举例一个确实减少周期的例子：

54、58H 这两个指令，在之前绝对是需要 stall 的，现在每个都只需要两个周期（regfile+decode，因为这个代码吧 regfile 单独给了一个周期，我感觉其实没必要，但是表示尊重就没改）：

54H↵	nor	\$6, \$5, \$2↵
58H↵	or	\$7, \$6, \$3↵

```

1 54H nor $6, $5, $2
2 58H ori $7, $6, $3 // 6具有相关性

```

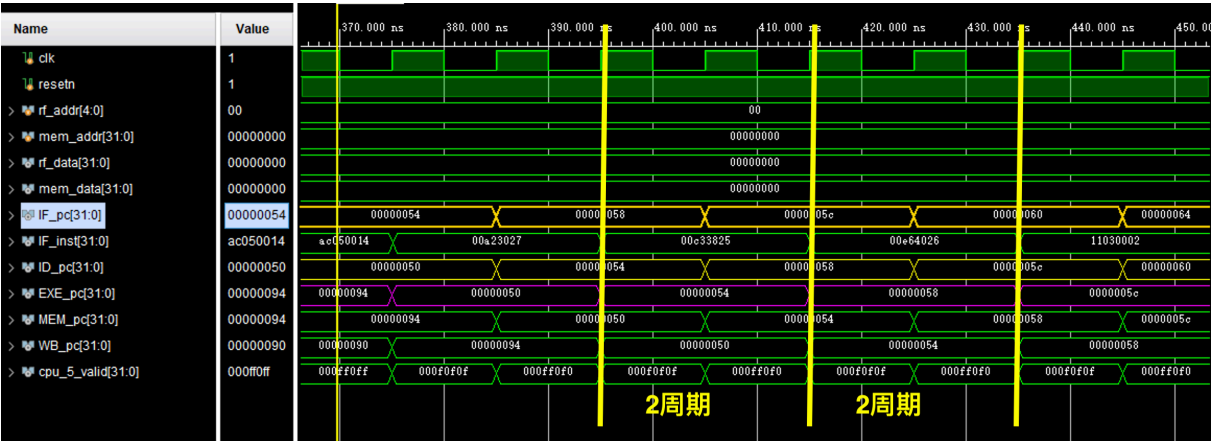


图 3.11: 54、58H 仿真

可以看到 58H 的执行周期完全不受影响，说明旁路在一般情况下确实正确。

4 实验总结与心得体会

1. 最开始实现了不完全正确的旁路 (EXE 阶段), 主要是在国庆前, 我没有想到 ID 阶段的 branch (课上的版本 branch 是在 EXE 阶段), 但是后退版本, 实现了全部前递到 ID 阶段的旁路, 简化了逻辑。最终正确实现了。
2. CPU 设计那个书也给了我一些灵感, 主要是 load-branch 的处理, 但是我并没有采用 CPU 设计书中的去更改 ready_go (代码中叫做 over) 信号。而是采用了一个新的 stall 信号, 我认为这样更加清晰和简单, 主要还是这样能够实现的情况下, 冒险去改级联控制的 ready_go 信号, 很容易出错。而且很明显可以观察到之前代码中 stall 产生信号和 CPU 设计实战那本书中建议的修改方法是冲突的, 只能留一个, 因此我还是选择留 stall 信号, 维护也方便。
3. 实现代码中, 我认为比较需要注意的点就是前递有限级别 (顺序), 和 stall 信号的产生。前者既保证了旁路的正确, 但同时也产生了 load-branch 类型的错误, 我认为我报告的一个优点也在于解释了转移计算未完成的原因 (CPU 设计实战那本书中没有提到)。
4. 感觉有点难度。