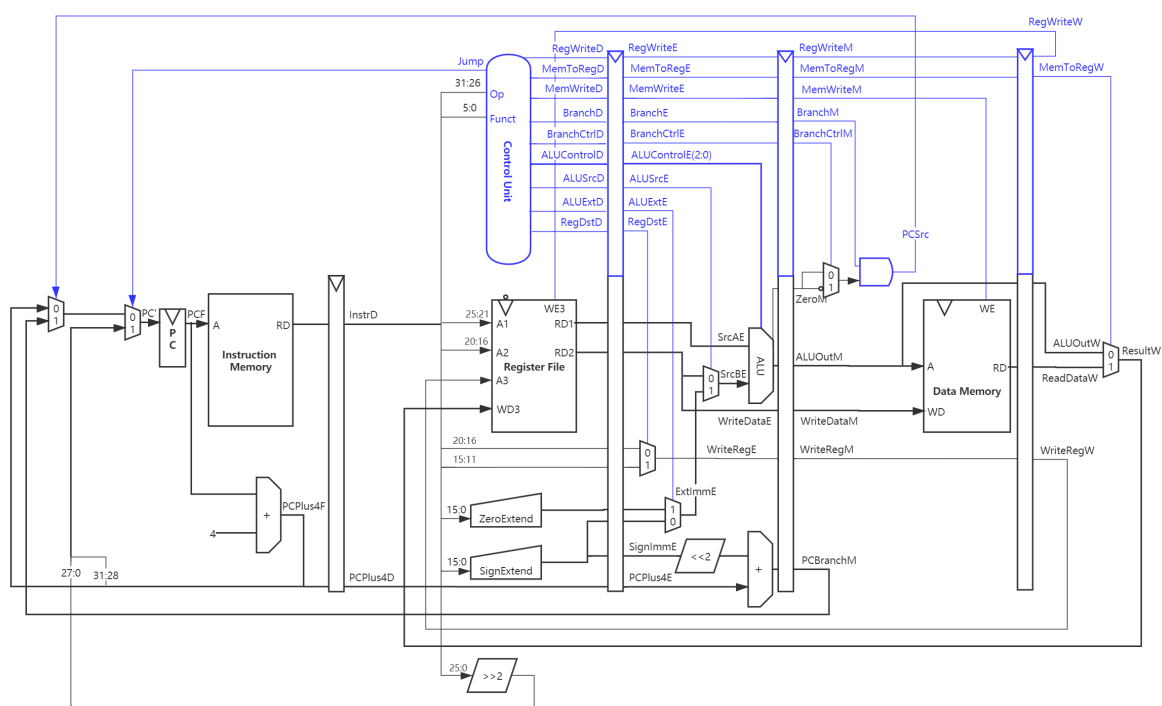


MIPS流水线处理器 实验报告

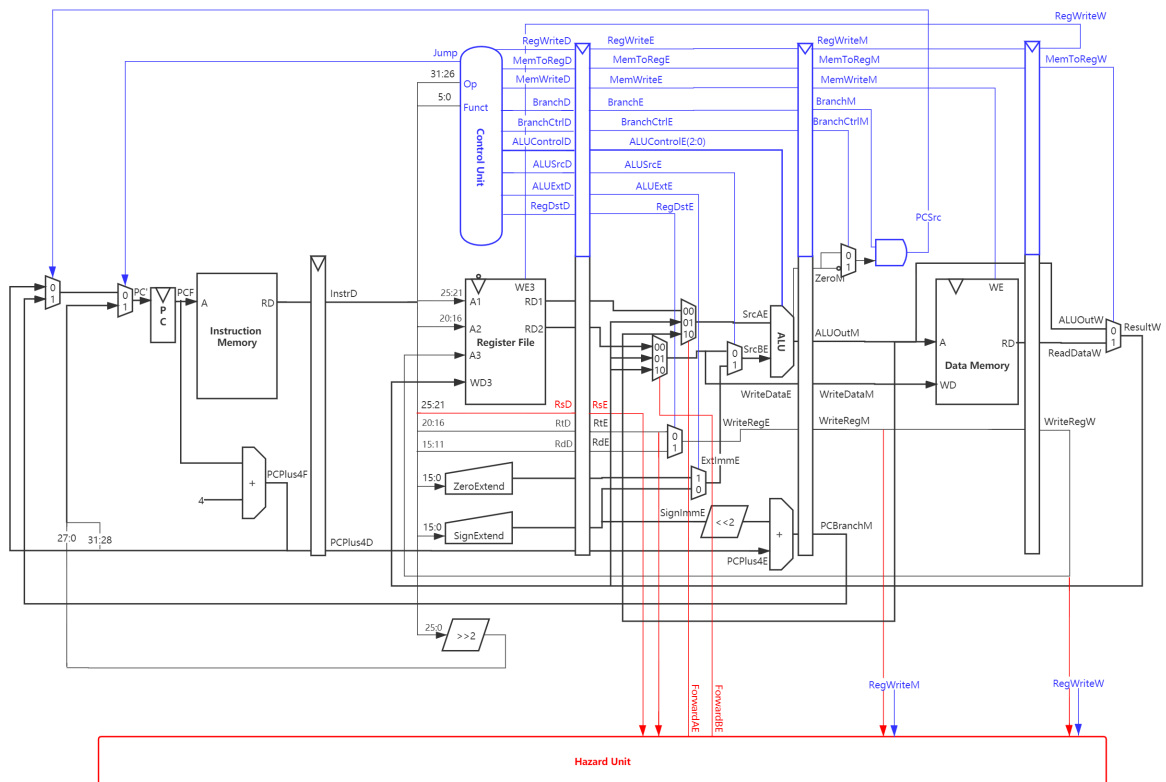
一、设计原理

流水线技术的思想是：将CPU部件划分为若干阶段，在指令执行过程中，各个指令分别处于不同的阶段，从而可以做到重叠执行多条指令。与非流水线相比，它降低了CPI (cycles per instruction)，从而大大减少了总CPU时间消耗。所有现代高性能微处理器都采用流水线技术。

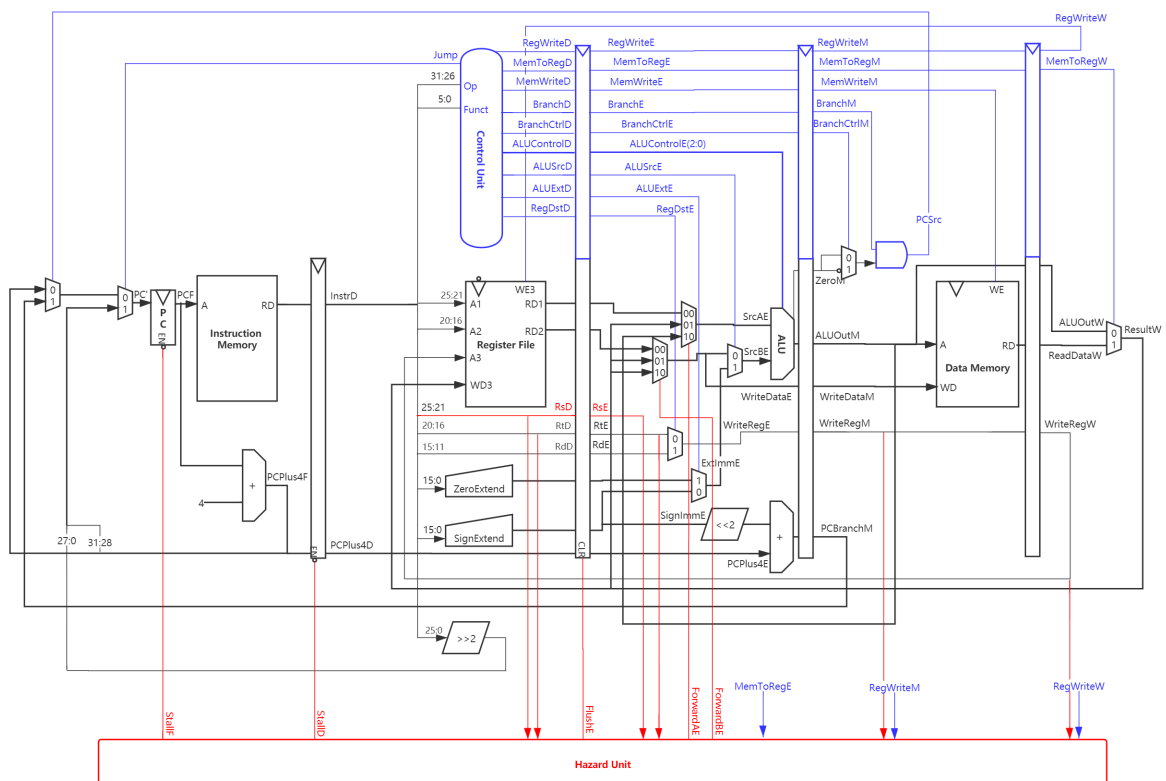


在MIPS单周期处理器的基础上，我们以数据路径为基础，将CPU电路划分为取指、译码、执行、访存、回写5个阶段，就得到了一个经典的5级MIPS流水线。这样划分的原因是，将5个主要部件各划分为一个阶段（imem、RF读、ALU、dmem、RF写），从而能够使5段的执行时间尽可能相等，避免形成瓶颈。

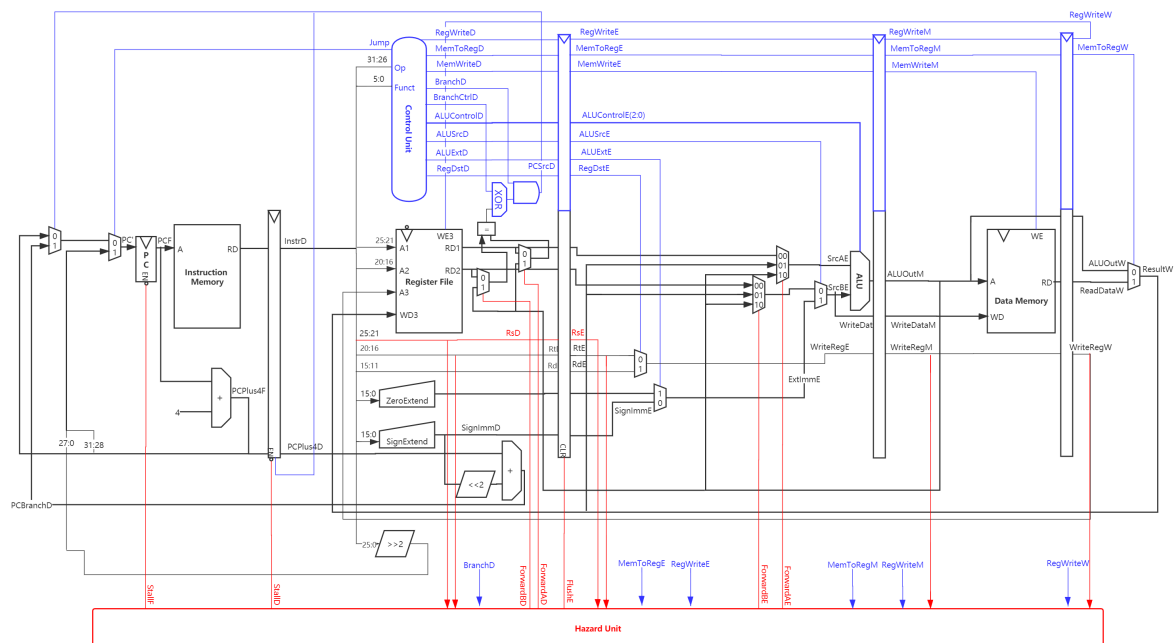
具体的电路图如上图所示。其中横亘整张电路图的4个长寄存器即为流水寄存器，将电路划分成5个阶段。



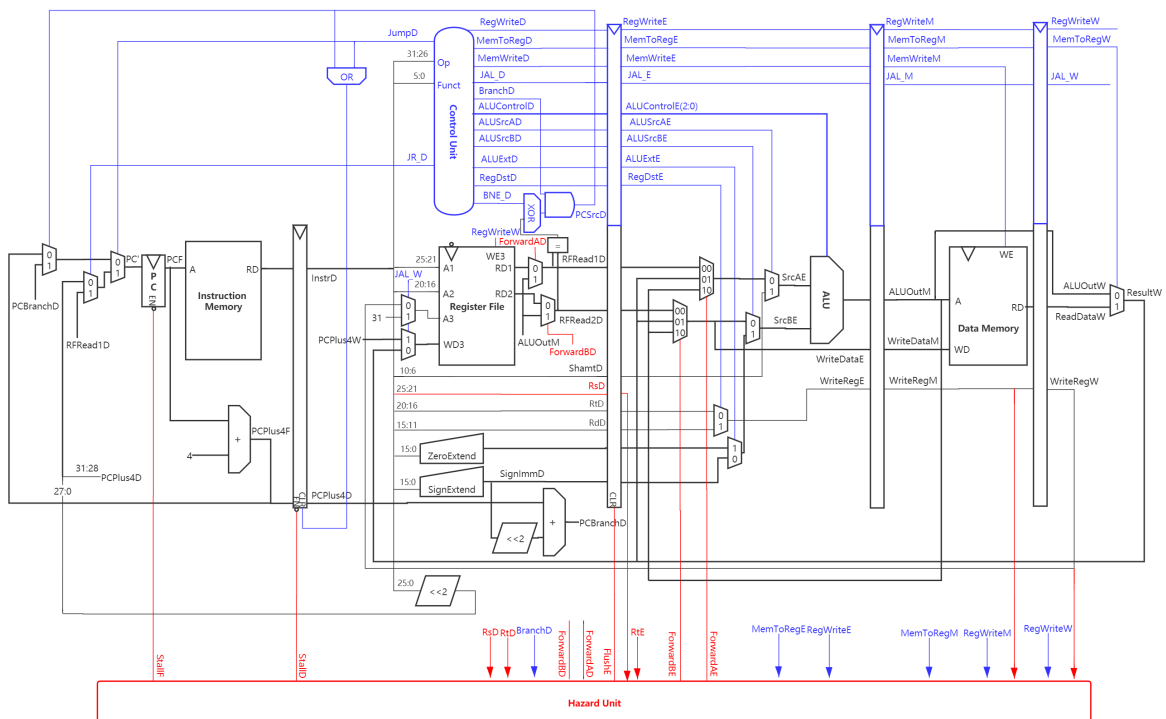
因为多个指令重叠执行，可能产生冒险(hazard)。理论上说处理冒险的最简单方式是阻塞后面的指令、让前面的指令执行完，但效率会很低。所以我们采用专门的硬件电路来处理。首先，是通过转发(forwarding)处理数据冒险，如上图所示。



对于无法通过转发解决的数据冒险，只能让流水线停顿(stall)。为此需要给每个流水寄存器都引入一个停顿信号，如上图所示。












最后是处理控制冒险，我们将分支判断提前到译码段完成，在此段添加一个专门的相等比较器比较两个操作数以判断分支成功，并在分支成功时清空分支指令的后一条指令。



最后，我们再添加元件以处理无条件跳转系列指令j、jr、jal、jalr和位移系列指令sll、sra、srl。至此，我们就得到了一个功能较为完善的MIPS流水线处理器，支持的指令有：add, sub, and, or, slt, sll, sra, srl, addi, andi, ori, slti, sw, lw, j, jr, jal, jalr, nop, beq, bne。

二、开发步骤

Initial commit for cache  jasha64 committed on 7 May
Fix bug of instr slti  jasha64 committed on 27 Apr
Add instr sll, srl, sra, sllv, srlv, srav  jasha64 committed on 3 Apr
Add instr j, jr, jal, jalr  jasha64 committed on 2 Apr
Solved control hazard. Full hazard handling. Add instr beq, bne.  jasha64 committed on 9 Mar
Add stalls  jasha64 committed on 8 Mar
Add data forwarding. Add instr addi, andi, ori, slti, nop.  jasha64 committed on 6 Mar
Initial pipeline w/o hazard unit. Supporting instr R, lw, sw  jasha64 committed on 5 Mar
Rename files, prepared for pipeline  jasha64 committed on 2 Mar

https://github.com/jasha64/MIPS_Pipeline_Cache_BPU

可于该github项目的提交记录中查看所有中间步骤的代码（包括最终成品）。（注：为防止抄袭，该项目目前为未公开状态，将于学期结束后公开）

三、仿真测试

测试结果

我们采用自动化的测试代码（代码见项目中的cpu_tb.sv）和拔尖课程提供的10笔测资进行测试。这10笔测资分别是：

ad hoc：课本7.6.3节提供的测资，覆盖课本上实现过的所有指令（R类指令、addi、sw、lw、j），无循环和函数调用。

bisection：二分查找，考察循环（循环过程用到bne、beq、j指令）和移位指令sll、sra。

bubble sort：冒泡排序，嵌套循环。

en & clear：见下方问题讨论部分。

factorial：分解质因数，考察递归（函数调用过程用到jr、jal指令）。

i-type：针对立即数类型(i-type)指令进行考察，主要考察立即数类型指令的实现是否严格正确。特别地，考察addi是否是零扩展而andi、ori、slti是否是符号位扩展。

mutual recursion: 背景是编译原理文法分析过程, 双重递归 (即两个函数互相递归调用)。

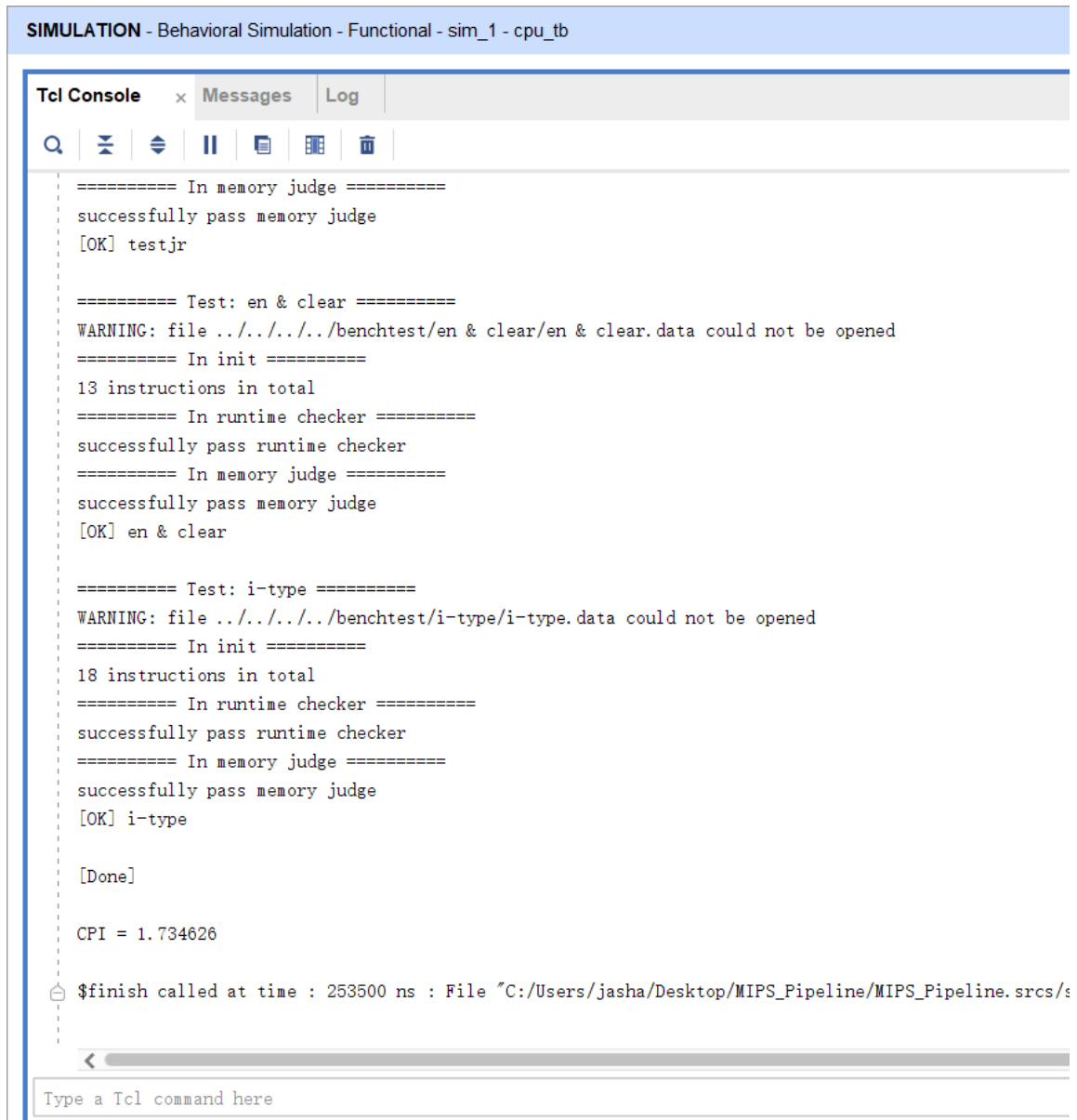
gcd: 求最大公约数, 考察循环。

quick multiply: 快速乘, 考察位移和循环。

testjr: 是一笔直接考察jr指令的测资, 主要考察jr指令出现数据冒险时能否正确处理。

(具体测资文件, 请见提交的项目目录 /benchtest 文件夹下的内容)

我的MIPS流水线处理器能通过上面所有的测资, 测试结果如下:



```
===== In memory judge =====
successfully pass memory judge
[OK] testjr

===== Test: en & clear =====
WARNING: file ../../../../benchtest/en & clear/en & clear.data could not be opened
===== In init =====
13 instructions in total
===== In runtime checker =====
successfully pass runtime checker
===== In memory judge =====
successfully pass memory judge
[OK] en & clear

===== Test: i-type =====
WARNING: file ../../../../benchtest/i-type/i-type.data could not be opened
===== In init =====
18 instructions in total
===== In runtime checker =====
successfully pass runtime checker
===== In memory judge =====
successfully pass memory judge
[OK] i-type

[Done]

CPI = 1.734626

$finish called at time : 253500 ns : File "C:/Users/jasha/Desktop/MIPS_Pipeline/MIPS_Pipeline.srsc/:
```

测资设计

非常值得记载的是, 以上10笔测资中, 1笔 (ad hoc) 是课本作者设计的, **2笔测资 (mutual recursion、en & clar) 由本人设计**, 2笔 (i-type、testjr) 由本学期正在修读拔尖班ICS课程的同学设计, 其余6笔由大二拔尖班ICS课程的助教 (也是我的室友) 设计。在此向相关同学表示感谢。

这里重点讲解一下我的 mutual recursion 测资的设计思路。室友让我帮他出一笔测资, 我看已有的测资中有2笔是顺序结构, 5笔是关于循环的, 只有1笔考察函数调用, 遂决定造一笔关于函数调用的测资, 弥补我们测试集的薄弱之处; 同时为了避免和已有的考察递归调用的 factorial 测资重复, 我决定将我的测资设置成双重递归。于是我在网上找了关于双重递归的资料, 并将相关的C++程序自行翻译成了汇编语言, 这样就得到了这笔测资。该测资通过了室友写的CPU的交叉验证, 并得到了同学的好评。

因为这笔测资的汇编程序很长，这里就不粘贴完整代码，读者如有兴趣可打开 .\benchtest\mutual recursion 文件夹查看。

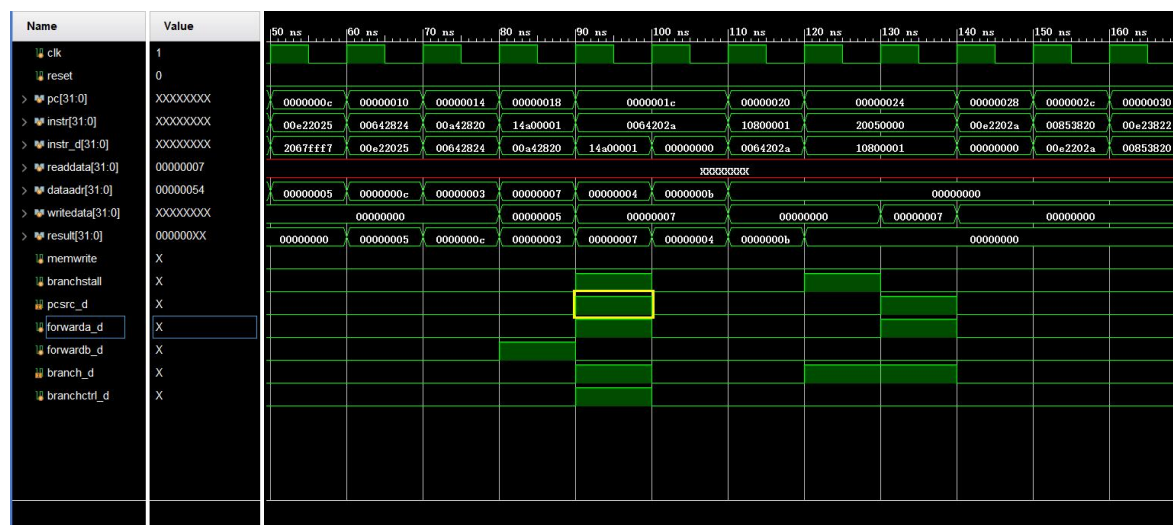
问题讨论

话说在开发控制冒险处理部件时，我使用 en & clear 这笔测资进行了仿真测试：

```
1  addi $v0, $0, 5
2  addi $v1, $0, 12
3  addi $a3, $v1, -9
4  or $a0, $a3, $v0
5  and $a1, $v1, $a0
6  add $a1, $a1, $a0
7  bne $a1, $0, around    # should be taken
8  add $a1, $a1, $a0
9  around:
10 sw $a1, 4($0)
11 nop
12 nop
13 nop
14 nop
```

（该测资也是我设计的，其实也就是把书上 ad hoc 这笔测资中的 beq 改成了等效的 bne，来测试我的 CPU 的 bne 指令实现是否正确。）

当时我的 CPU（参考了课件上的实现）未能通过该笔测资。波形图如下：



这里我们发现问题是，在黄框指向的那一时钟周期，分支指令是成功的，应当跳转并清空后续的一条指令，但实际上没有清空后续指令（看下个时钟周期的 instr_d，理论应当为 00000000）。

从而就定位了问题所在：因为复位(clear)信号和停顿(stall)信号同时为1，在流水线正在停顿中的90-100ns，会被清除掉，但当时流水线还在等待相关指令的计算结果，这一周期是基于过时的寄存器值得到的clear信号，是错误的。参考代码有BUG！

```

module flopenrc #(parameter WIDTH = 8)
    (input logic          clk, reset, en, clear,
     input logic [WIDTH-1:0] d,
     output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset)
        if      (reset) q <= 0;
        else if (clear) q <= 0;
        else if (en)    q <= d;
endmodule

```

我将以上的参考实现修改成了以下写法：

```

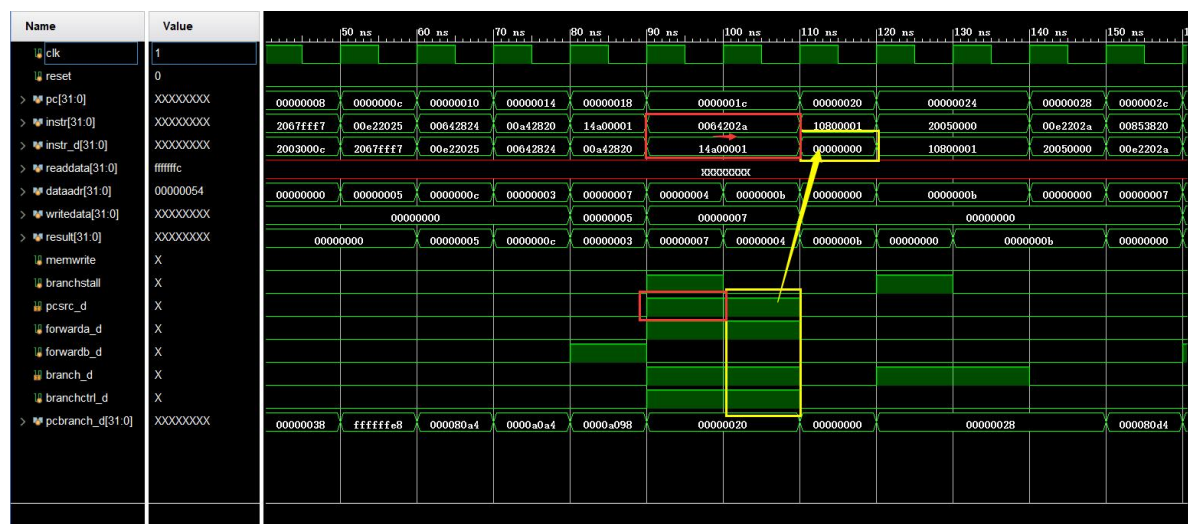
module flopenrc #(parameter WIDTH = 8)
    (input logic          clk, reset, en, clear,
     input logic [WIDTH-1:0] d,
     output logic [WIDTH-1:0] q);

    always_ff @(posedge clk, posedge reset)
        if      (reset) q <= 0;
        else if (en & clear) q <= 0;
        else if (en)    q <= d;
endmodule

```

其中，唯一一处改动是将"clear"改为"en & clear"，即只有当不在停顿状态时才允许复位。

这样一来，我的CPU就通过了这笔测资，现在的波形图如下：



我们看到，在作此修改后，CPU就不会基于过时的寄存器值清空流水寄存器，而是在停顿结束后再根据最新的寄存器值计算 flush 信号，也就解决了这个BUG。

P.S. 另外一种可行的解决方案是直接在水寄存器的 flush 信号那里直接加一个" & ^stall"。

P.S. 这也是我的这笔测资得名的由来：因为需要把参考代码中的"en"改成"en & clear"，所以我就把这笔测资命名为"en & clear"，希望能给大二的同学们一些提示。

四、上板验证

验证过程

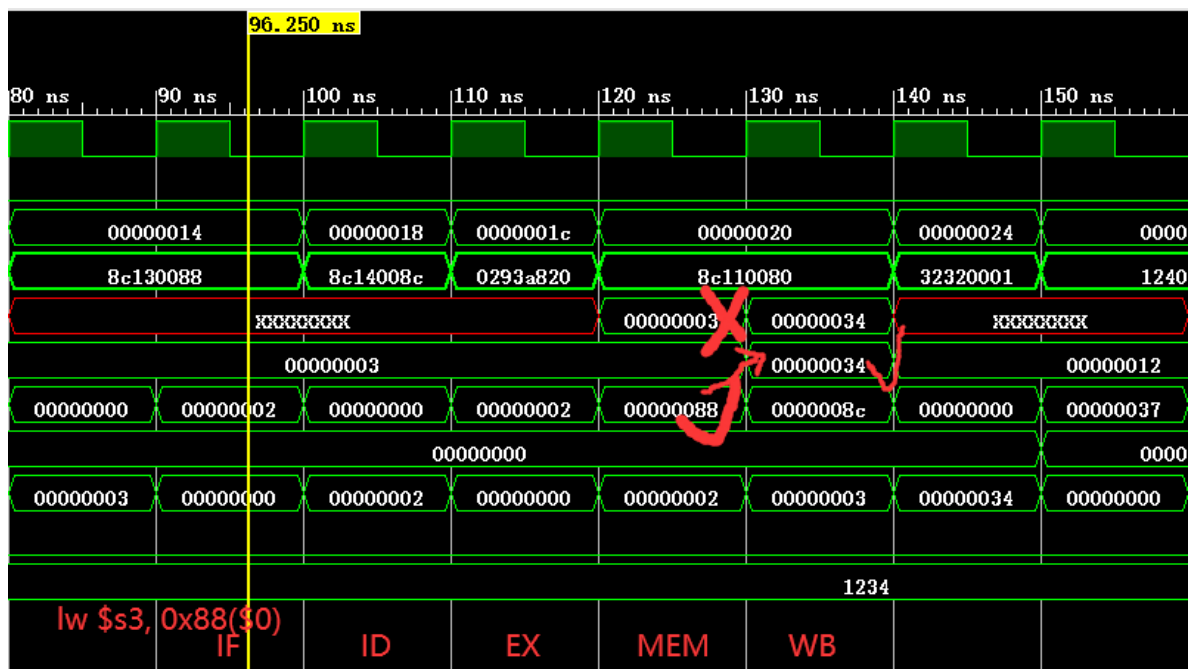
我们参照老师提供的资料，实现一个IO接口，从开发板上的开关(switch)读入数据（并显示到七段数码管左半部分）、将计算结果输出到LED灯（并显示到七段数码管右半部分）。将以下测资内容烧录到指令内存中：（由老师课件提供，已经预先放置在
.\\MIPS_Pipeline\\MIPS_Pipeline.runs\\Obj\\memfile.dat，可直接综合）

```
main:
    addi $s0, $0, 0
    sw    $s0, 0x80($0)
chkSwitch:
    lw    $s1, 0x80($0)
    andi  $s2, $s1, 0x2
    beq   $s2, $0, chkSwitch
    lw    $s3, 0x88($0)
    lw    $s4, 0x8C($0)
    add   $s5, $s4, $s3
chkLED:
    lw    $s1, 0x80($0)
    andi  $s2, $s1, 0x1
    beq   $s2, $0, chkLED
    sw    $s5, 0x84($0)
    j     chkSwitch
```

我们将开关拨到"12"和"34"，按下右侧按钮(开关输入)，再按下左按钮(LED输出)，即可看到相加结果，具体可查看实验录像（.\\IO.mp4）。

问题讨论

一开始上板验证时，发现计算结果不正确。经过仿真分析，发现这样一个问题：在对I/O设备进行读取时，原本应在访存(MEM)段读出的值到回写(WB)段才读出来。



经过分析，我推断原因如下：我们知道always @ (posedge clk)下的所有非阻塞赋值(<=)右边的表达式计算，是以时钟上升沿前一瞬间的值为标准的。因为老师提供的IO代码模板使用了这种写法，导致读取的数据要迟一个周期反馈出来。（作为对比，如果是读数据内存(dmem)而非I/O设备，因为dmem是组合电路，就可以在地址变化的同时立刻读出应该读的数据。）

