## 计算机体系结构课程第 三 次实验报告

**实验名称：流水线CPU的实现设计**

学号：2013029 姓名：刘伟

(注：1-6页为指导手册理论、7-12页为流水线改错修正、13-end页为流水线优化设计)

1. 实验目的
2. 在多周期CPU实验完成的提前下，深入理解CPU流水线的概念。
3. 熟悉并掌握流水线CPU的原理和设计。
4. 最终检验运用verilog语言进行电路设计的能力。
5. 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和体系结构理论知识的理解。
6. 培养对CPU设计的兴趣，加深对CPU现有架构的理解和深思。
7. 实验内容说明

本次实验基于多周期CPU的基础上，前期的课程设计准备同多周期CPU的实验，主体部分可以直接使用多周期CPU实验的设计方案，但在多周期CPU中只要求实现了30多条指令，此处要求扩展到40多条指令。

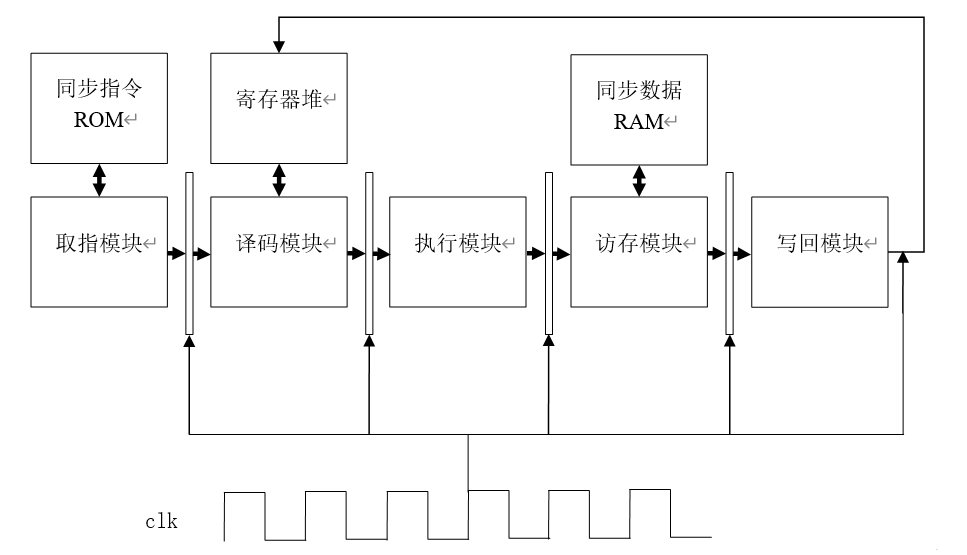
多周期CPU在单周期基础上提高了时钟频率，但并没有改善执行一条指令的时间，且存在资源闲置的问题，例如当指令在执行级有效时，译码级实际上在空转。若每一级都在执行有效的指令，将解决资源闲置的问题。最理想的情况是，当第一条指令从取指级转换到下一级译码时，第二条指令进入取指级，当第一条指令完成译码进入执行级时，第二条指令进入译码级，第三条指令进入取指级……静态5级流水CPU就是基于这样的设计思路。



(b)多周期; (c)5级流水CPU时空图

在流水线CPU中，当在一时钟周期内完成了某一条指令的全部执行时（写回级完成），则有望在下一时钟周期内完成下条指令的执行，因此依然相当于是一个周期完成一条指令，而时钟频率更高，因此CPU可以运行的更快。

在流水线CPU中，每条指令完全执行仍旧是需要5个时钟周期。但是在同一时间是5个部分是可以做到共同运转的，故整体的设计框图没有改变。



1. 实验需要注意的细节

本次课程设计的关键和难点在于流水线的控制，比如一条指令何时可以从译码级进入执行级，一条指令何时需要堵在流水线中。本次课程设计暂不考虑前递技术，因此有数据相关时就需要堵在流水线中。

MIPS架构中有延迟槽的设定，其本意是加快流水CPU的执行速度。在之前单周期和多周期CPU实验中未支持延迟槽，但在流水CPU中需要支持该设定，因为只有硬件支持了延迟槽技术，用通用编译器编译出来的MIPS二进制执行文件才能在自己设计的CPU中运行正确。

MIPS架构中分支和跳转指令参与计算的PC值均为延迟槽指令对应的PC(即分支跳转指令的PC+4),在本课程设计中尤其需要注意这一点。比如一条指令“beq,r0,r0,#2”在不考虑延迟槽的多周期CPU中，其跳转的目标地址为beq指令后面的第2条。而在考虑延迟槽的流水CPU中，其跳转的目标地址为beq指令后面的第3条（即延迟槽指令后面的第2条）。在编写测试程序时就需要注意分支跳转指令的偏移量。

1. 实验设计要求
2. 确认流水CPU的设计框图的正确性；
3. 编写verilog代码，将表9.2中自己编写的汇编程序翻译为二进制，以coe文件的方式初始化到指令ROM中；
4. 对该模块进行仿真，得出正确的波形，截图作为实验报告结果一项的材料；
5. 完成调用流水CPU的外围模块的设计，并编写代码；
6. 对代码进行综合布局布线下载到实验箱里FPGA板上，进行上板验证；
7. 实验原理

流水CPU实现的指令集是在多周期CPU的指令集上增加9条指令，主要是乘法指令，与HI、LO、协处理器0（Coprocessor 0, cp0）寄存器传送数据的指令以及特权指令，共实现了45条指令。

5级流水线CPU实现的mips指令特性归纳

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **指令类型** | **汇编指令** | **指令码** | **源操作数1** | **源操作数2** | **源操作数3** | **目的寄存器** | **功能描述** |
| R型指令 | addu rd,rs,rt | 000000|rs|rt|rd|00000|100001 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]+GPR[rt] |
| subu rd,rs,rt | 000000|rs|rt|rd|00000|100011 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]-GPR[rt] |
| slt rd,rs,rt | 000000|rs|rt|rd|00000|101010 | [rs] | [rt] |  | rd | GPR[rd]=(sign(GPR[rs])<sign(GPR[rt])) |
| sltu rd,rs,rt | 000000|rs|rt|rd|00000|101011 | [rs] | [rt] |  | rd | GPR[rd]=(zero(GPR[rs])<zero(GPR[rt])) |
| jalr rs | 000000|rs|00000|11111|00000|001001 | [rs] |  |  | 31 | GPR[31]=PC,PC=GPR[rs] |
| jr rs | 000000|rs|0000000000|00000|001000 | [rs] |  |  |  | PC=GPR[rs] |
| and rd,rs,rt | 000000|rs|rt|rd|00000|100100 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]&GPR[rt] |
| nor rd,rs,rt | 000000|rs|rt|rd|00000|100111 | [rs] | [rt] |  | rd | GPR[rd]=~(GPR[rs]|GPR[rt]) |
| or rd,rs,rt | 000000|rs|rt|rd|00000|100101 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]|GPR[rt] |
| xor rd,rs,rt | 000000|rs|rt|rd|00000|100110 | [rs] | [rt] |  | rd | GPR[rd]=GPR[rs]^GPR[rt] |
| sll rd,rt,shf | 000000|00000|rt|rd|shf|000000 |  | [rt] |  | rd | GPR[rd]=zero(GPR[rt])<<shf |
| sllv rd,rt,rs | 000000|rs|rt|rd|00000|000100 | [rs] | [rt] |  | rd | GPR[rd]=zero(GPR[rt])<<(GPR[rs]%32) |
| sra rd,rt,shf | 000000|00000|rt|rd|shf|000011 |  | [rt] |  | rd | GPR[rd]=sign(GPR[rt])>>shf |
| srav rd,rt,rs | 000000|rs|rt|rd|00000|000111 | [rs] | [rt] |  | rd | GPR[rd]=sign(GPR[rt])>>(GPR[rs]%32) |
| srl rd,rt,shf | 000000|00000|rt|rd|shf|000010 |  | [rt] |  | rd | GPR[rd]=zero(GPR[rt])>>shf |
| srlv rd,rt,rs | 000000|rs|rt|rd|00000|000110 | [rs] | [rt] |  | rd | GPR[rd]=zero(GPR[rt])>>GPR[rs] |
| mult rs,rt | 000000|rs|rt|0000000000|011000 | [rs] | [rt] |  | (HI,LO) | (HI,LO)=sign(GPR[rs])\*sign(GPR[rt]) |
| mflo rd | 000000|0000000000|rd|00000|010010 | [LO] |  |  | rd | GPR[rd]=[LO] |
| mfhi rd | 000000|0000000000|rd|00000|010000 | [HI] |  |  | rd | GPR[rd]=[HI] |
| mtlo rs | 000000|rs|000000000000000|010011 | [rs] |  |  | LO | [LO]=GPR[rs] |
| mthi rs | 000000|rs|000000000000000|010001 | [rs] |  |  | HI | [HI]=GPR[rs] |
| I型  指令 | addiu rt,rs,imm | 001001|rs|rt|imm | [rs] | sign\_ext (imm) |  | rt | GPR[rt]=GPR[rs]+sign\_ext (imm) |
| slti rt,rs,imm | 001010|rs|rt|imm | [rs] | sign\_ext (imm) |  | rt | GPR[rt]=(sign(GPR[rs])<sign\_ext (imm)) |
| sltiu rt,rs,imm | 001011|rs|rt|imm | [rs] | sign\_ext (imm) |  | rt | GPR[rt]=(zero(GPR[rs])<sign\_ext (imm)) |

5级流水线CPU实现的mips指令特性归纳（续）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **指令类型** | **汇编指令** | **指令码** | **源操作数1** | **源操作数2** | **源操作数3** | **目的寄存器** | **功能描述** |
| I型  指令 | beq rs,rt,offset | 000100|rs|rt|offset | [rs] | [rt] |  |  | if GPR[rs]=GPR[rt] then PC= next\_pc+ sign\_ext (offset)<<2 |
| bgez rs,offset | 000001|rs|00001|offset | [rs] |  |  |  | if GPR[rs]≥0 then PC= next\_pc + sign\_ext (offset)<<2 |
| bgtz rs,offset | 000111|rs|00000|offset | [rs] |  |  |  | if GPR[rs]>0 then PC= next\_pc + sign\_ext (offset)<<2 |
| blez rs,offset | 000110|rs|00000|offset | [rs] |  |  |  | if GPR[rs]≤0 then PC= next\_pc + sign\_ext (offset)<<2 |
| bltz rs,offset | 000001|rs|00000|offset | [rs] |  |  |  | if GPR[rs]<0 then PC= next\_pc + sign\_ext (offset)<<2 |
| bne rs,rt,offset | 000101|rs|rt|offset | [rs] | [rt] |  |  | if GPR[rs]≠GPR[rt] then PC= next\_pc + sign\_ext (offset)<<2 |
| lw rt,offset(b) | 100011|b|rt|offset | [b] | sign\_ext (offset) |  | rt | GPR[rt]=Mem[GPR[b]+sign\_ext (offset)] |
| sw rt,offset(b) | 101011|b|rt|offset | [b] | sign\_ext (offset) | [rt] |  | Mem[GPR[b]+sign\_ext (offset)]=GPR[rt] |
| lb rt,offset(b) | 100000|b|rt|offset | [b] | sign\_ext (offset) |  | rt | GPR[rt]=sign(Mem[GPR[b]+sign\_ext (offset)]) |
| lbu rt,offset(b) | 100100|b|rt|offset | [b] | sign\_ext (offset) |  | rt | GPR[rt]=zero(Mem[GPR[b]+sign\_ext (offset)]) |
| sb rt,offset(b) | 101000|b|rt|offset | [b] | sign\_ext (offset) | [rt] |  | Mem[GPR[b]+sign\_ext (offset)]=GPR[rt] |
| andi rt,rs,imm | 001100|rs|rt|imm | [rs] | zero\_ext (imm) |  | rt | GPR[rt]=GPR[rs]&zero\_ext (imm) |
| lui rt,imm | 001111|00000|rt|imm |  | {imm, 16'd0} |  | rt | GPR[rt]= {imm, 16'd0} |
| ori rt,rs,imm | 001101|rs|rt|imm | [rs] | zero\_ext (imm) |  | rt | GPR[rt]=GPR[rs]|zero\_ext (imm) |
| xori rt,rs,imm | 001110|rs|rt|imm | [rs] | zero\_ext (imm) |  | rt | GPR[rt]=GPR[rs]^zero\_ext (imm) |
| J型  指令 | j target | 000010|target |  |  |  |  | [PC]={next\_pc[31:28],target<<2} |
| jal target | 000011|target |  |  |  |  | GPR[31]=PC,PC={next\_pc[31:28],target<<2} |
| cp0指令 | mfc0 rt,cs | 010000|00000|rt|cs|00000000|sel | CPR[cs.sel] |  |  | rt | GPR[rt]=CPR[cs.sel] |
| mtc0 rt,cd | 010000|00100|rt|cd|00000000|sel |  | GPR[rt] |  | CPR[cd.sel] | CPR[cd.sel]=GPR[rt] |
| syscall | 000000|code|001100 |  |  |  |  | CPR[14.0]=PC,CPR[13.0][6:2]=01000,CPR[12.0][1]=1,PC=CPR[15.1]+0x180并跳转 |
| Eret | 010000|1|0000000000000000000|011000 |  |  |  |  | CPR[12.0][1]=0,PC=CPR[14.0]并跳转 |

注：分支跳转指令参与运算的不在是当前PC值，而是next\_pc,即当前PC+4，即延迟槽指令的PC。

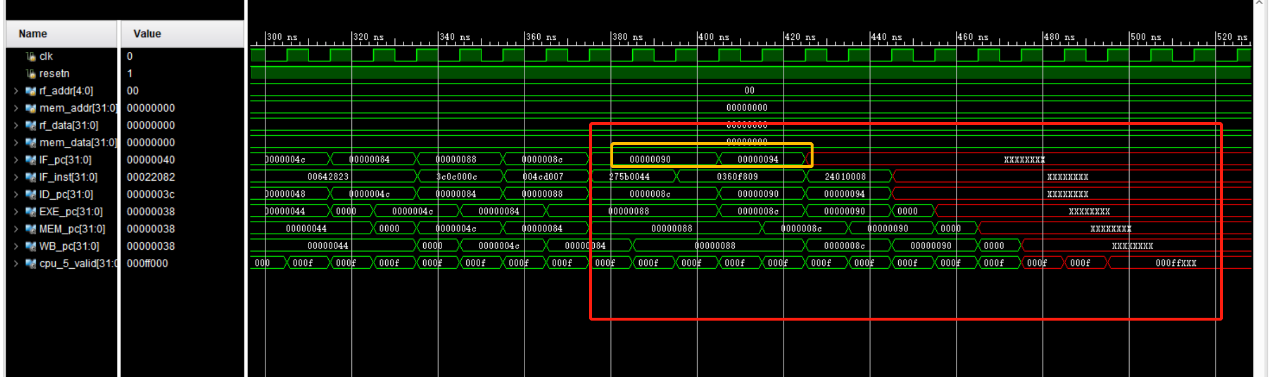
5级流水线CPU实现框图：



1. 实验顶层模块图



1. 实验具体设计步骤（改错部分）：
2. 建立项目工程后，首先创建IP核。建立inst\_rom、data\_rom。建立的时候需要导入测试文件.coe文件，其中包括了仿真所需的测试MIPS指令的二进制代码。
3. 导入其他各个模块：顶层模块tb.v、pipeline\_cpu.v，实例化IF、ID、EXE、MEM、WB、rf、ALU、adder各个模块。
4. 初次建立仿真观察实验样例代码中存在的问题：



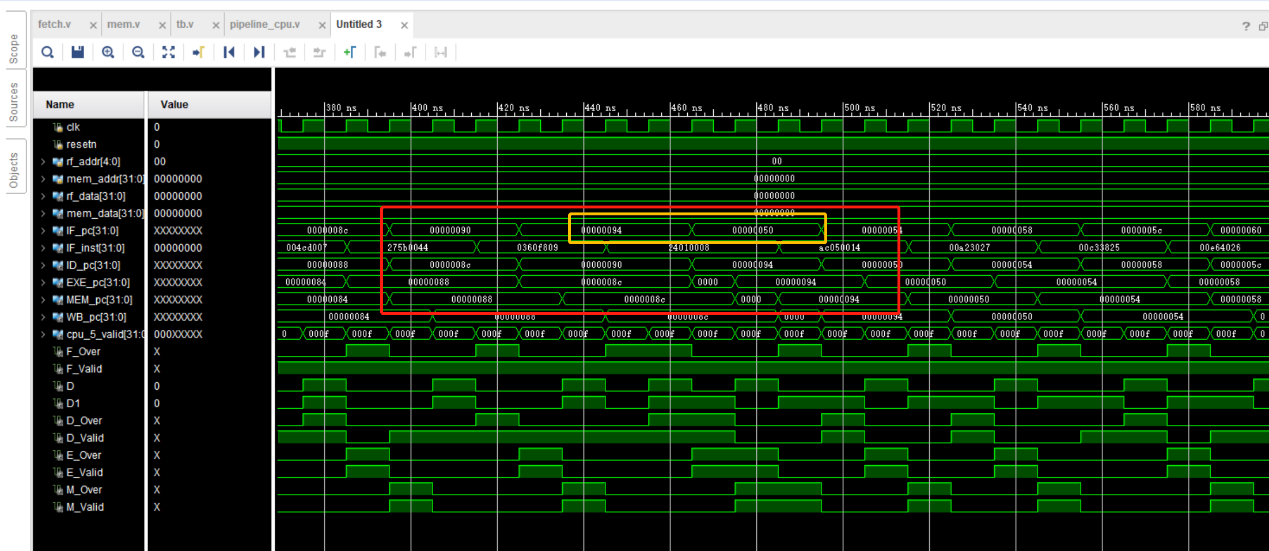
从仿真结果可以看出，指令在90H:jalr $27这条跳转指令时出现了问题，无法正确跳转到目标地址处。（注：联想到多周期CPU设计时，程序一开始在bgez $25,#16指令处发生问题，考虑到可能是IP核与IF、MEM代码中取数据的周期有关。故作者尝试仿照多周期CPU的程序设计，在IF：fetch.v和MEM:mem.v的代码中，延长阶段的结束控制信号。）

延长IF和MEM阶段的结束指令，再次对代码指令进行仿真，并添加大量的输出内容，便于观察分析：

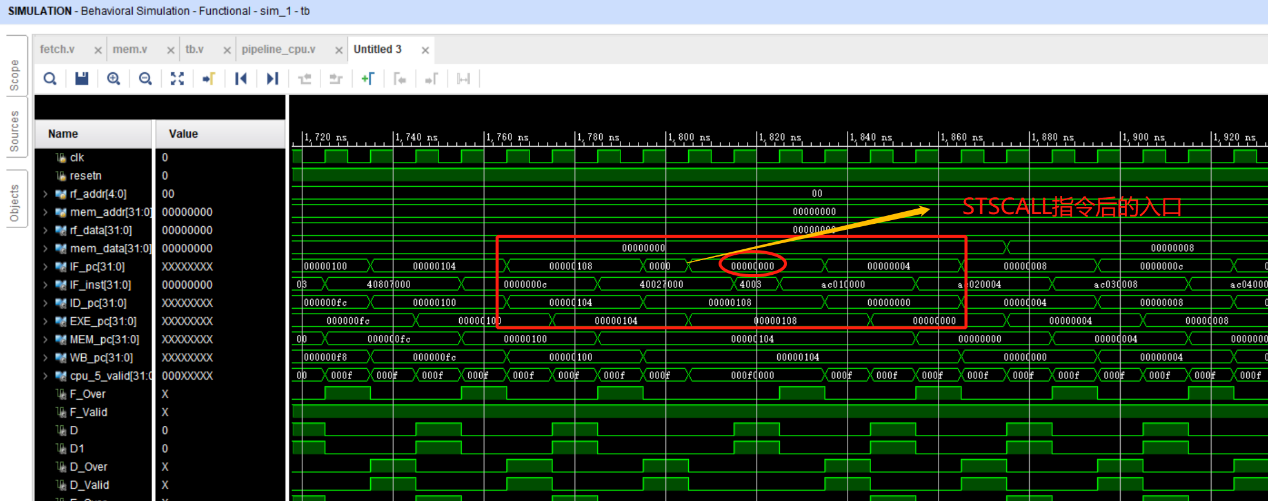
下图一：成果执行90H:jalr $27指令

下图二：成功跳转Exception入口地址

下图三：成功执行测试文件最后一条MIPS指令



图一



图二



图三

从仿真结果可以看出，代码可以正确的执行完测试文件中所有的MIPS指令。

（注：此处作者已经将fetch.v和mem.v的over信号进行了延迟。）

修改代码如下： 代码中新增的Delay为延迟计数器

**IF阶段fetch.v**

always @(posedge clk)

begin

if (!resetn || next\_fetch)

begin

IF\_over <= 1'b0;

Delay <= 1'b0;

end

else

begin

if(Delay == 1'b0)

Delay = 1'b1;

else

begin

IF\_over <= IF\_valid;

Delay <= 1'b0;

end

end

end

**MEM阶段mem.v**

always @(posedge clk)

begin

if (MEM\_allow\_in)

begin

MEM\_valid\_r <= 1'b0;

end

else

begin

if(Delay == 1'b0)

Delay = 1'b1;

else

begin

MEM\_valid\_r <= MEM\_valid;

Delay = 1'b0;

end

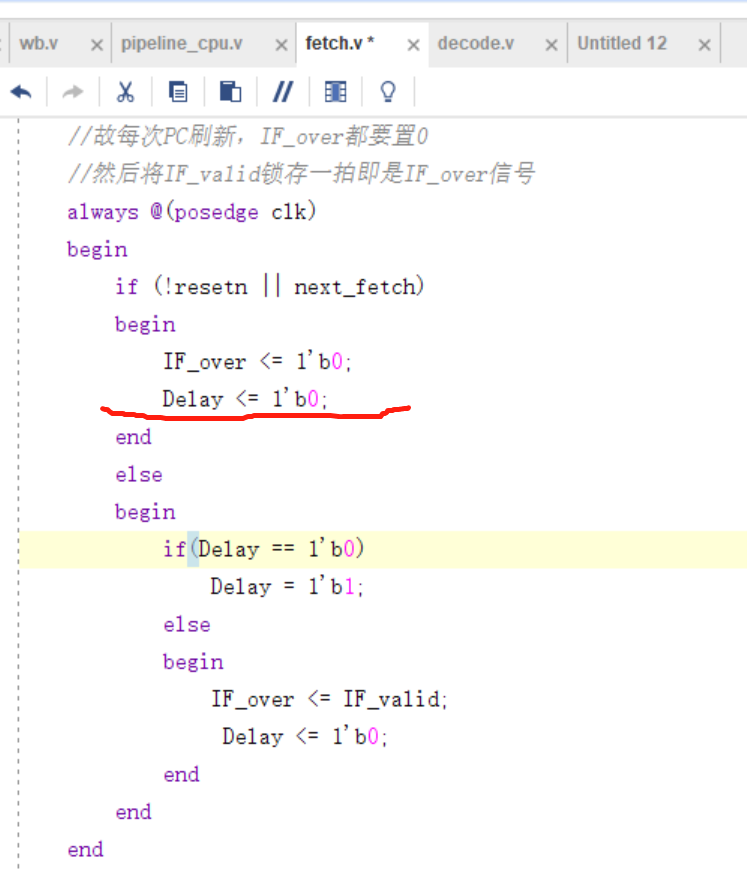
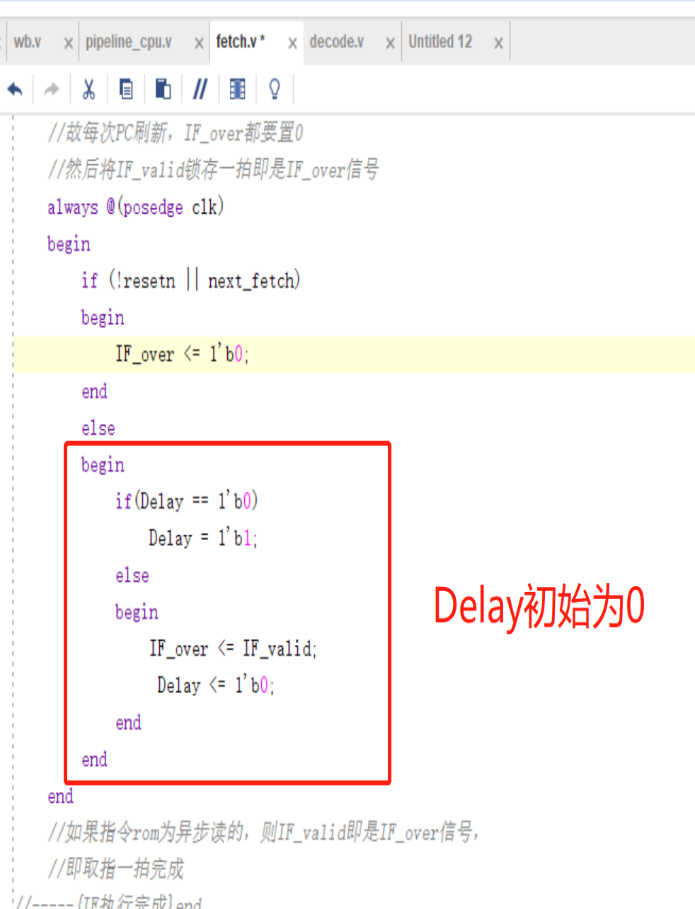
end

end

实际修改IF和MEM阶段over控制信号的时候，作者曾遇到过一个bug，在此阐述一下：（注：该区别是在贺祎昕同学的帮助下发现的）

注意看下面俩图中的不同点：

在第二张图中，在if(!resetn || next\_fetch)分支下添加了关于延迟计数器的重新置零。实验发现不在此处添加的话，程序会在后续测试代码中出现问题，无法正常执行。



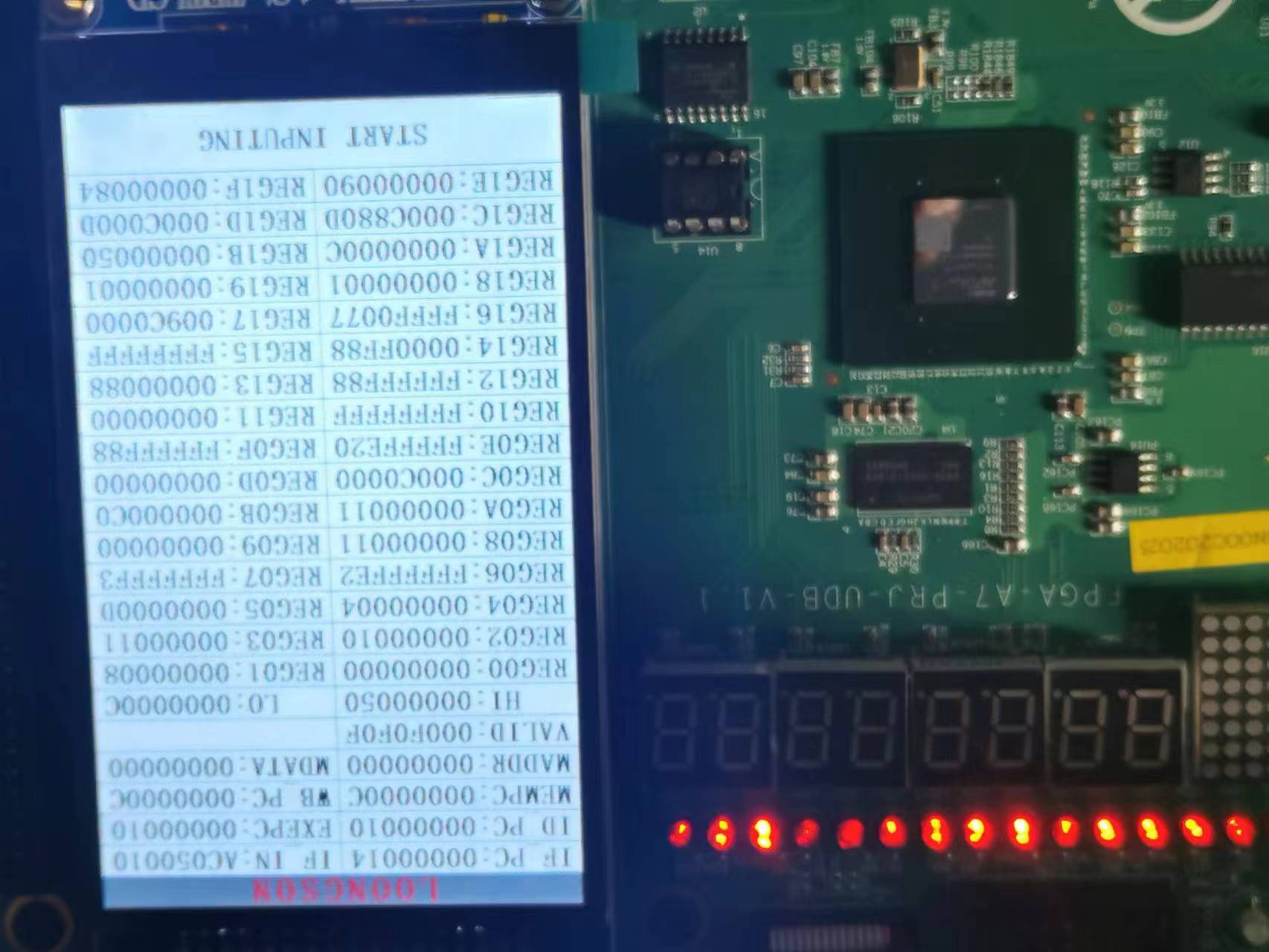
作者开始利用仿真输出，观察问题出现的根本原因。

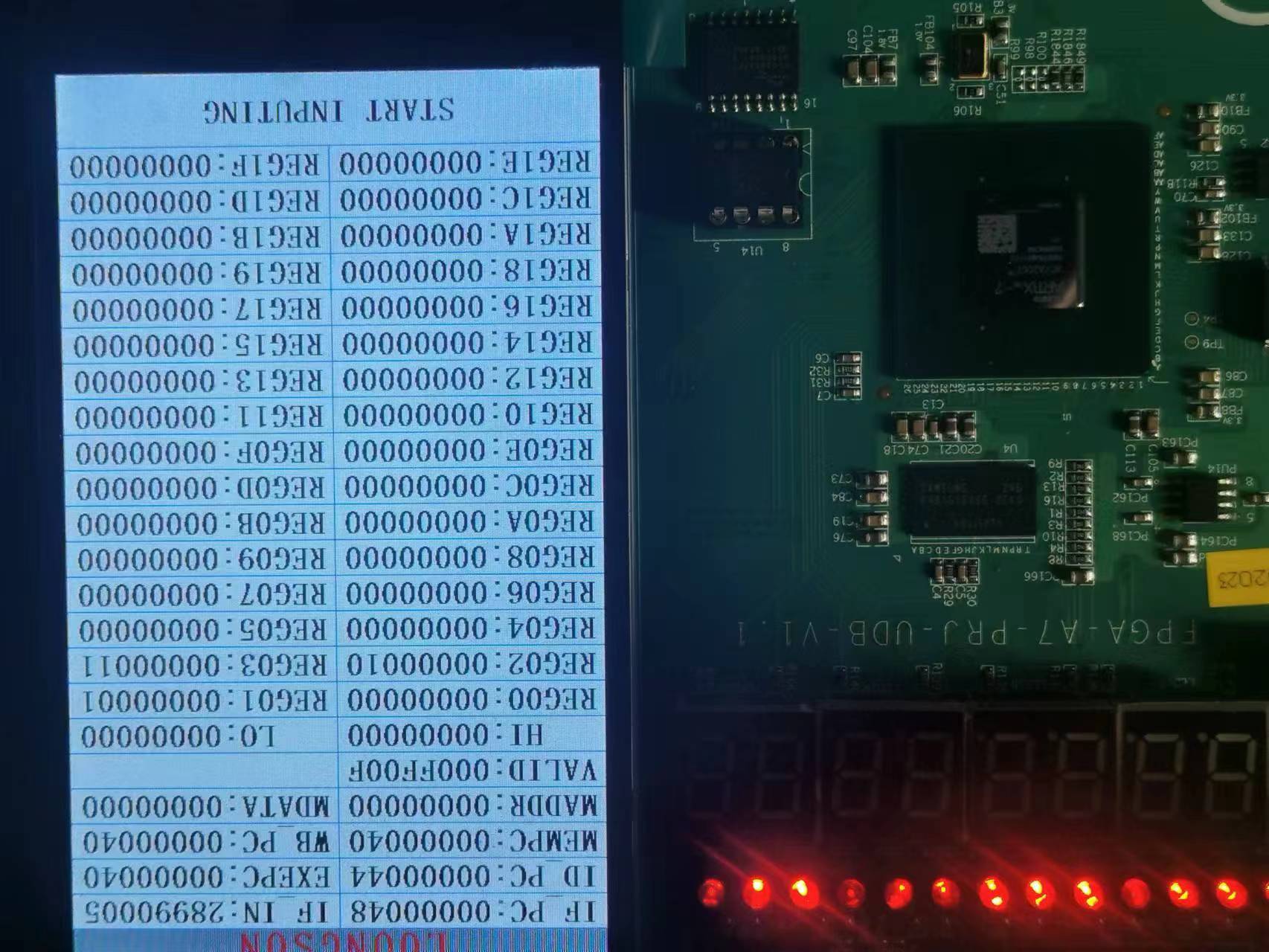


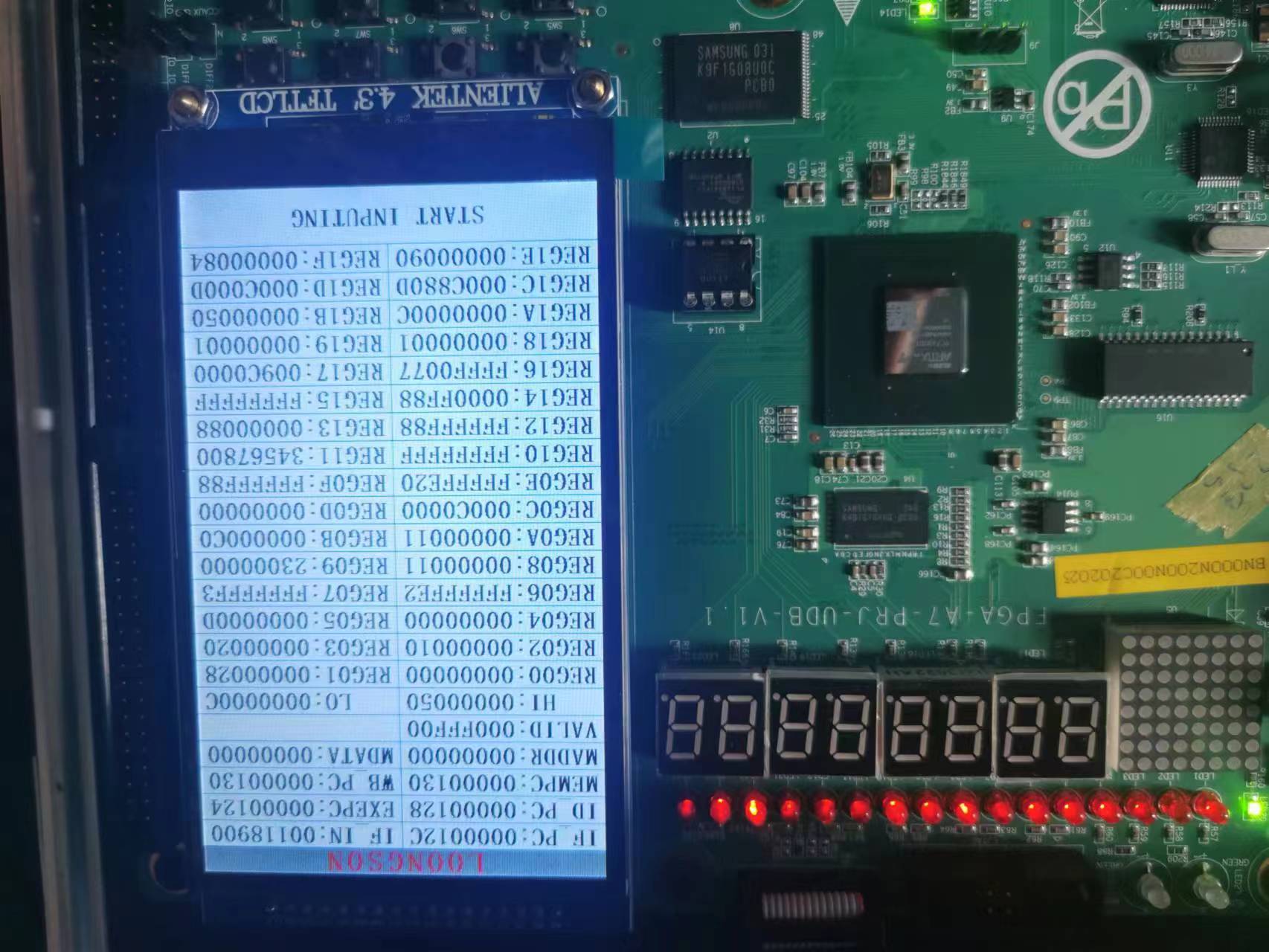
在仿真图中正确情况下的延迟计数器D与未加next\_fetch重置的D1相比，在90H:jalr $27这条指令处出现了差异，由于无条件跳转ID阶段后立马回IF阶段，导致二者控制信号与clk时钟之间出现了新的对应关系，不在next\_fetch下添加置零信号，那么延迟计数器将不会对下一条指令产生作用，出现指令执行内容的混乱，造成最后的无法执行。

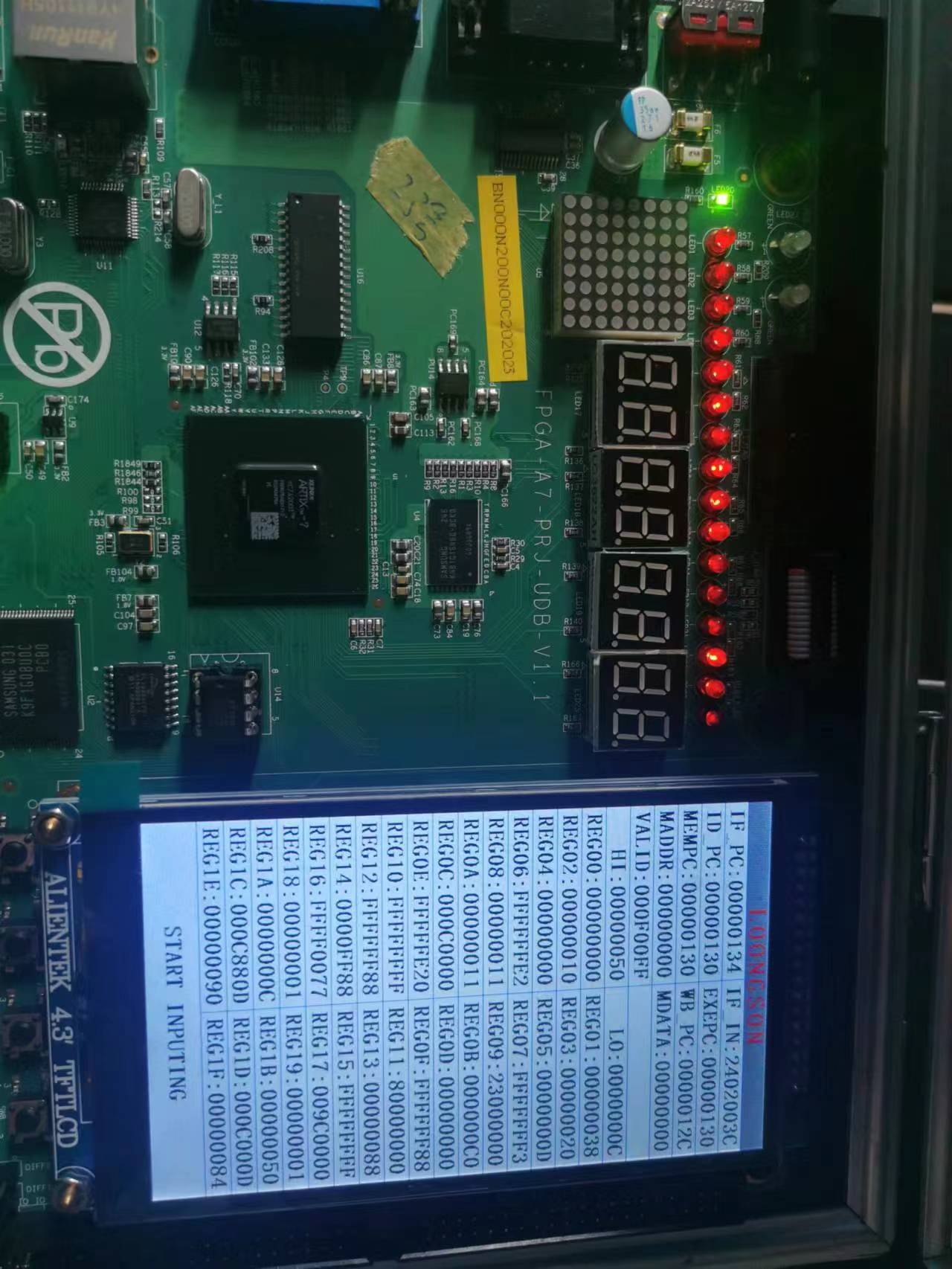
1. 利用display模块，对代码进行综合布局布线下载到实验箱里FPGA板上，进行上板验证。观察实验箱FPGA验证结果。

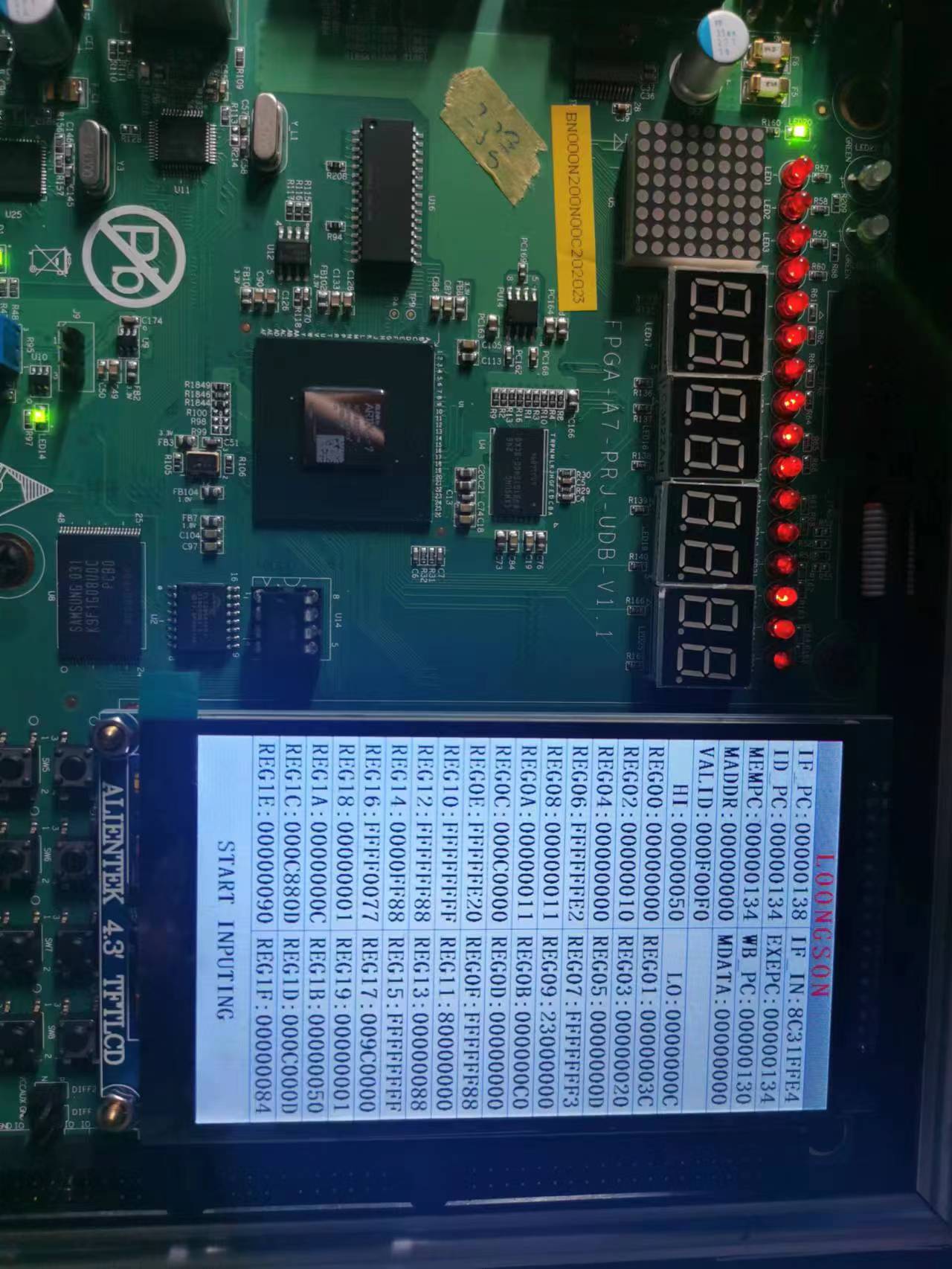
截取部分图，从FPGA实验板上各个寄存器的显示值可以看出，代码是正确可行的，成功实现5级静态流水线。







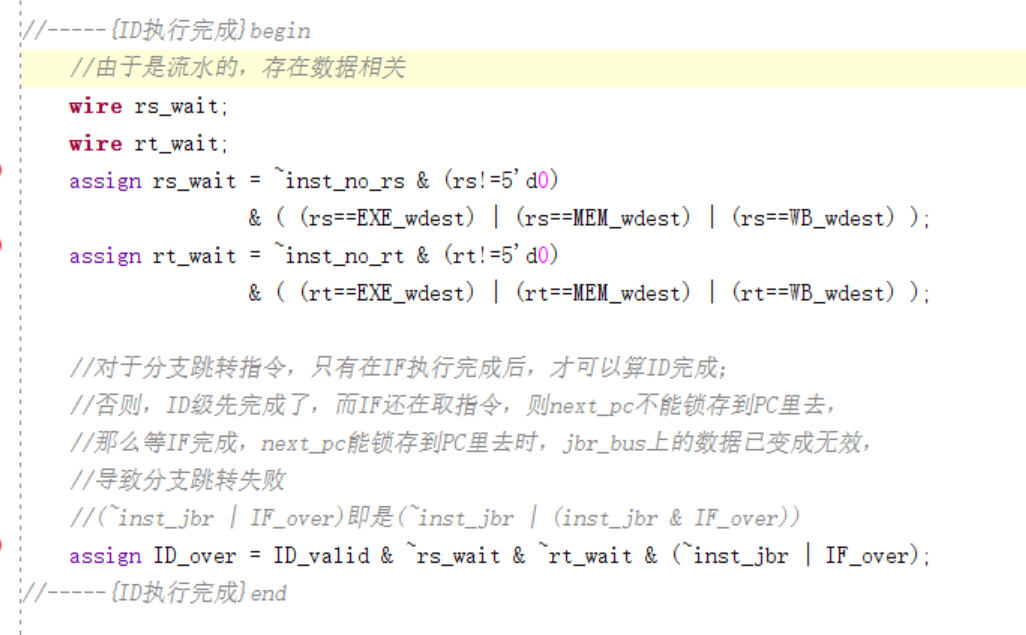




1. 实验优化设计（旁路bypass&分支预测Branch Prediction）
2. 旁路bypass设计

分析原始静态5级流水是如何处理指令间数据相关的冲突

分析代码，发现原始静态5级流水是在decode.v的代码中处理数据相关的冲突问题。



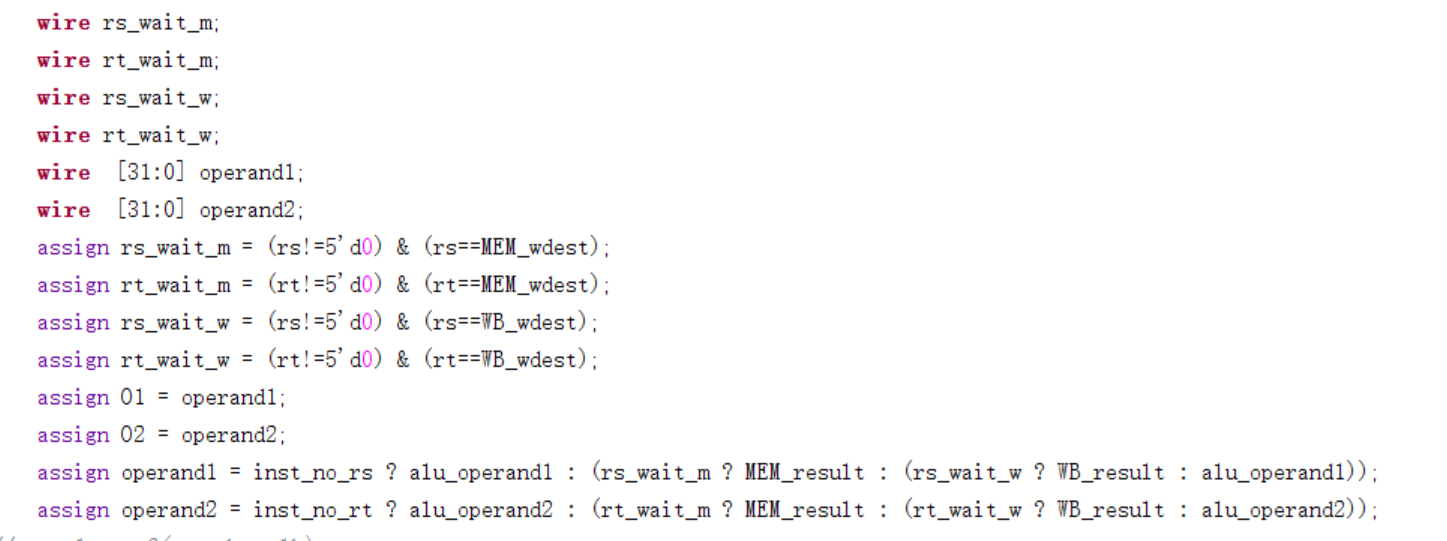
从代码段中可以看出，ID阶段在decode解码的过程中，利用EXE、MEM、WB回传的目的寄存器号与decode解码出的源寄存器号rs、rt进行了比较。

如果rs、rt与目的寄存器号是相同的，那么decode阶段的ID\_over信号将会进行延迟，确保当前指令解码完送进EXE阶段的内容是不具有数据相关的。

结合以上特点，作者开始尝试有关旁路bypass的设计。

通过掌握在体系结构课学到的知识，作者将EXE计算操作所需的操作数operand进行了旁路连接，将MEM\_result和WB\_result传递到EXE阶段。在EXE阶段去进行数据相关的判断，进而对操作数加以选择。

核心代码exe.v：

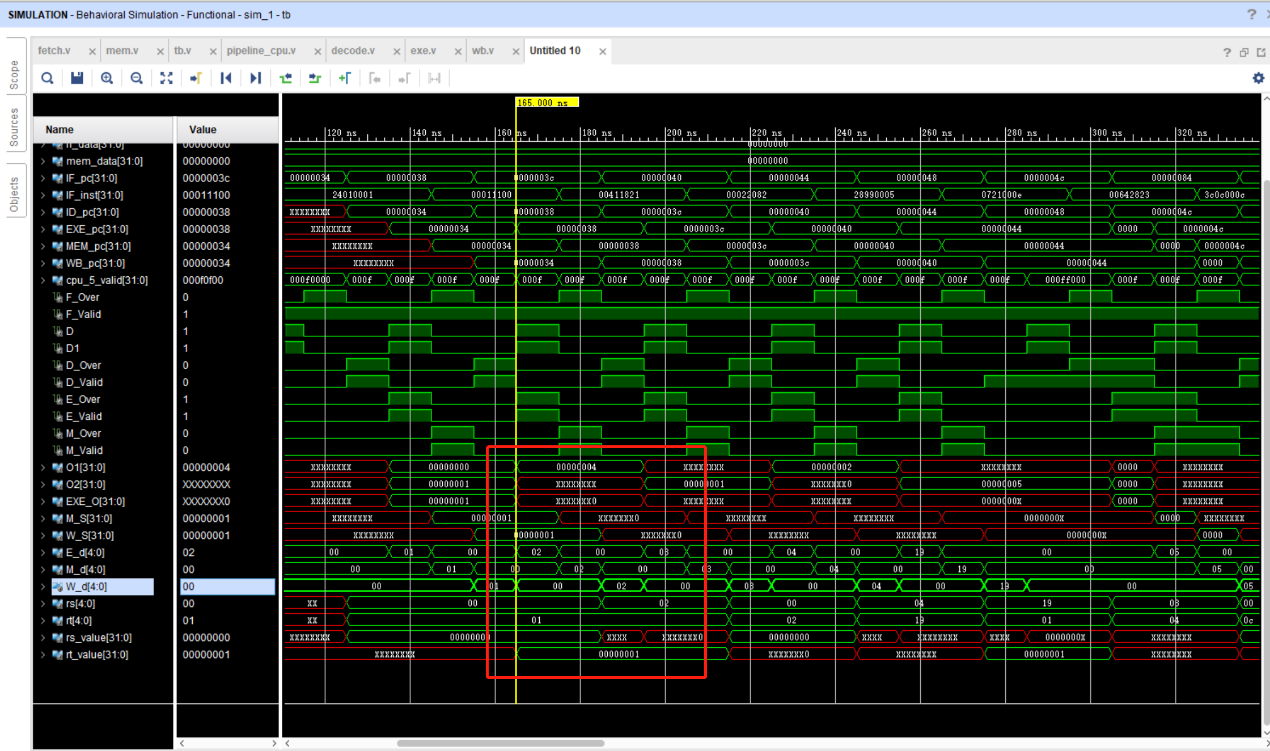


代码中的rs\_wait\_m、rt\_wait\_w、rt\_wait\_m、rt\_wait\_w是对数据相关的判断。如果寄存器值与mem或wb的目的寄存器相同的情况下。

inst\_no\_rs、inst\_no\_rs则是因为在本次实验的测试代码中，部分MIPS指令其不需要rs、rt寄存器，但是rs、rt所在字段的内容不为0，需要额外的进行判断。

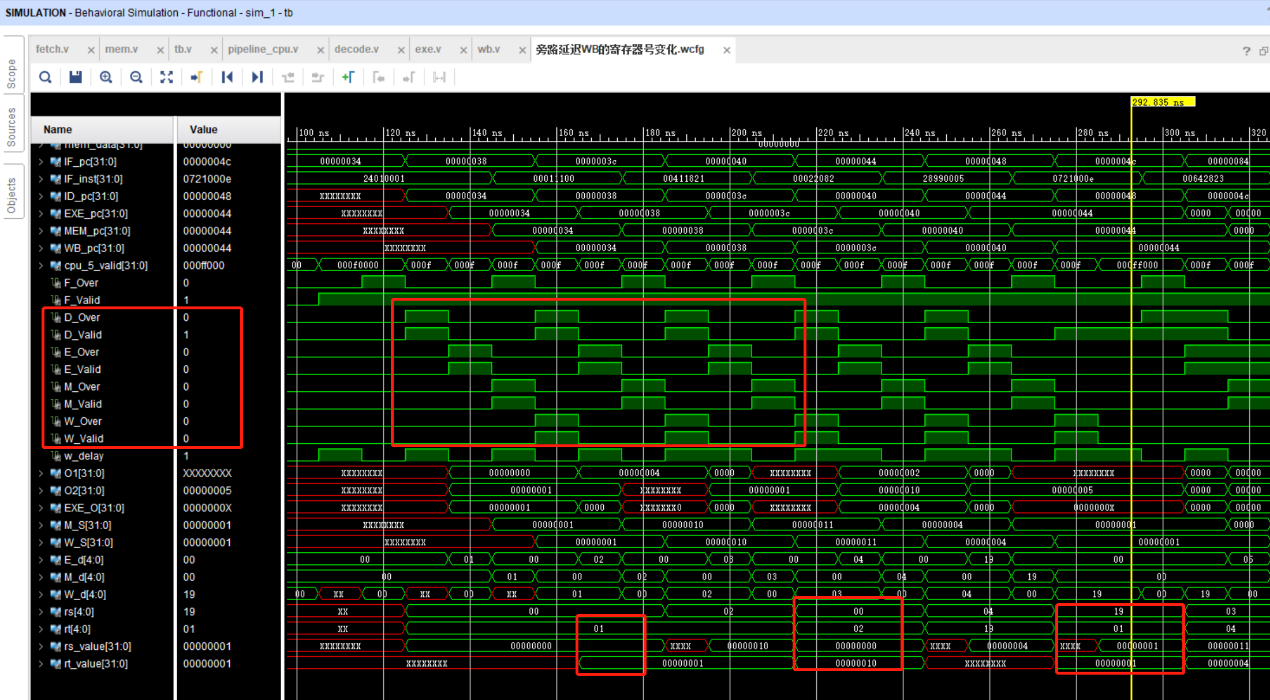
作者通过修改ID到EXE阶段的总线，将decode阶段解码出的rs、rt、inst\_no\_rs、inst\_no\_rt值传递给EXE。

仿真验证：



在结果图中，第二条指令 sll $2,$1,#4和第一条指令addiu $1,$0,#1有数据相关。但是在仿真图中，代码在exe阶段的俩个操作数O1、O2并未将数据前递旁路进来。分析原因得知，由于原本是在ID阶段判断寄存器数据相关的，作者修改的代码将该数据相关的判断延迟到了EXE阶段，导致进入EXE阶段时，该时钟周期时，数据相关冲突以及消失。

进一步修改：将WB写回寄存器号单独往后再保留一个时钟。再次仿真：

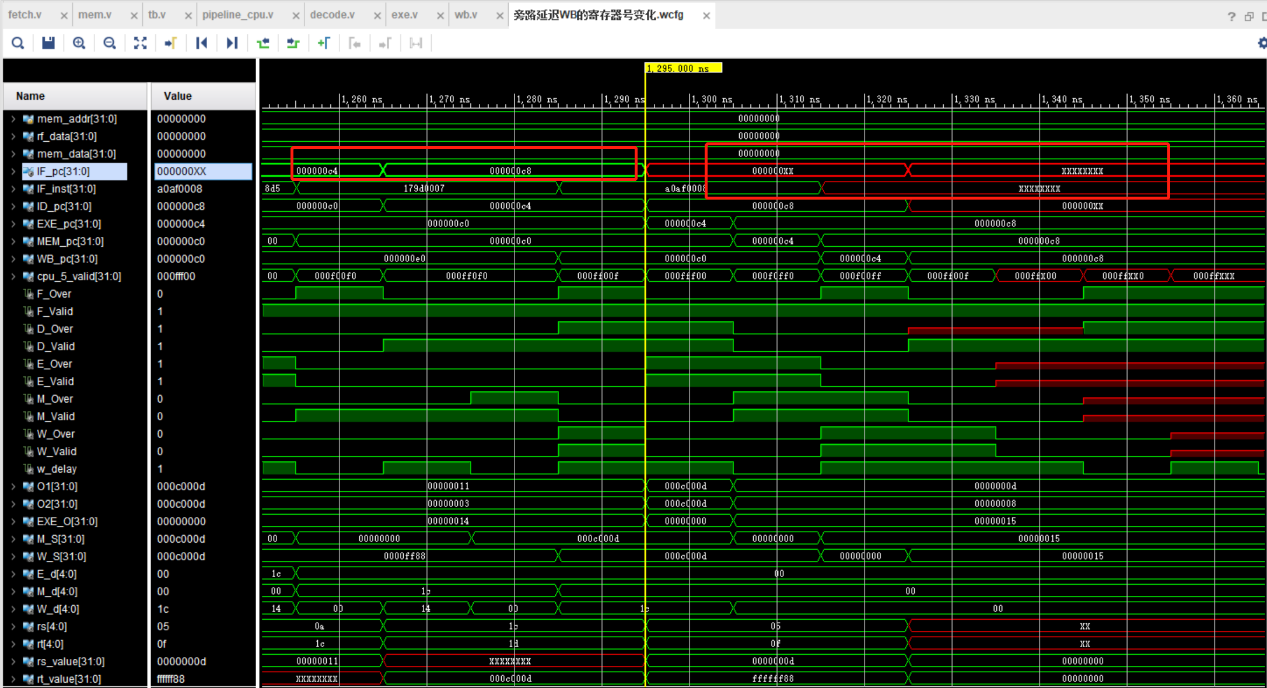


为了便于说明仿真结果的正确性，列出测试文件前六条指令：

|  |
| --- |
| addiu $1, $0,#1 |
| sll $2, $1,#4 |
| addu $3, $2,$1 |
| srl $4, $2,#2 |
| slti $25,$4,#5 |
| bgez $25,#14 |

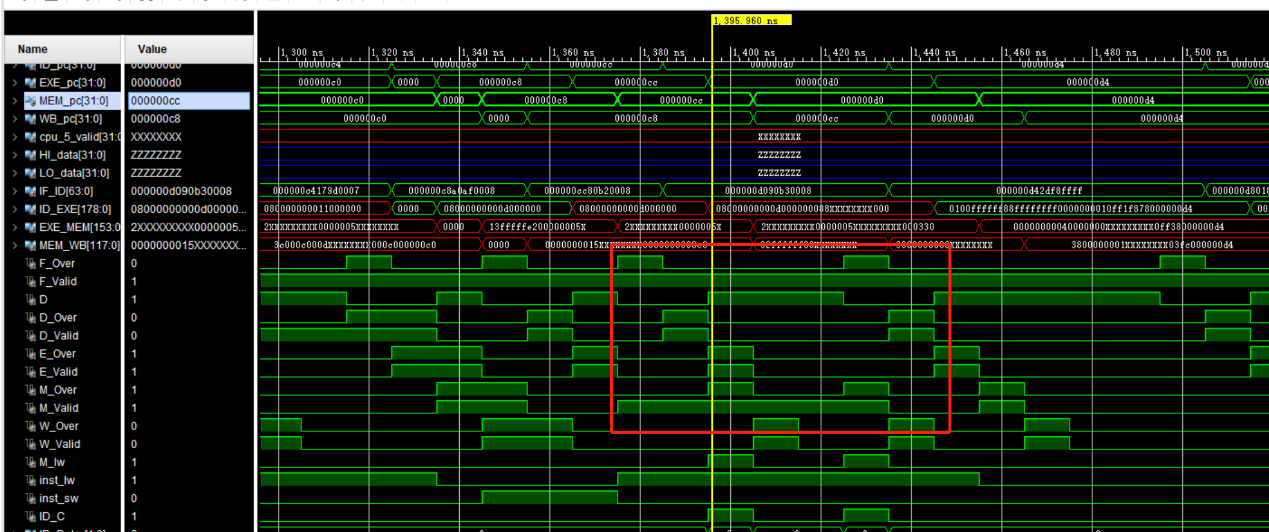
很明显，前六条指令的数据相关非常明显。第二条指令的源寄存器$1是第一条指令的目标寄存器，第三条指令的源寄存器$1$2分别是第一条和第二条的目的寄存器······，从仿真结果可以看出，旁路成功的将先执行的指令的结果前递到了有数据相关的后续指令上。旁路实现成功！

但是目前的修改版本，并没有完全通过测试文件！



当前版本的旁路代码在测试文件中遇到lw指令时，出现了问题。从各阶段的valid、over信号变化可以看出，错误原因是因为：lw指令mem阶段有延迟，但是这个延迟ID阶段并不知道，所以ID阶段更早的更新了rs、rt寄存器的值，导致EXE阶段在开始判断计算数据相关的时候WB阶段还未更新目的寄存器号，导致EXE误认为没有发生数据相关的冲突，直接进行了计算，最终计算结果出现了未知X情况，无法确定跳转。

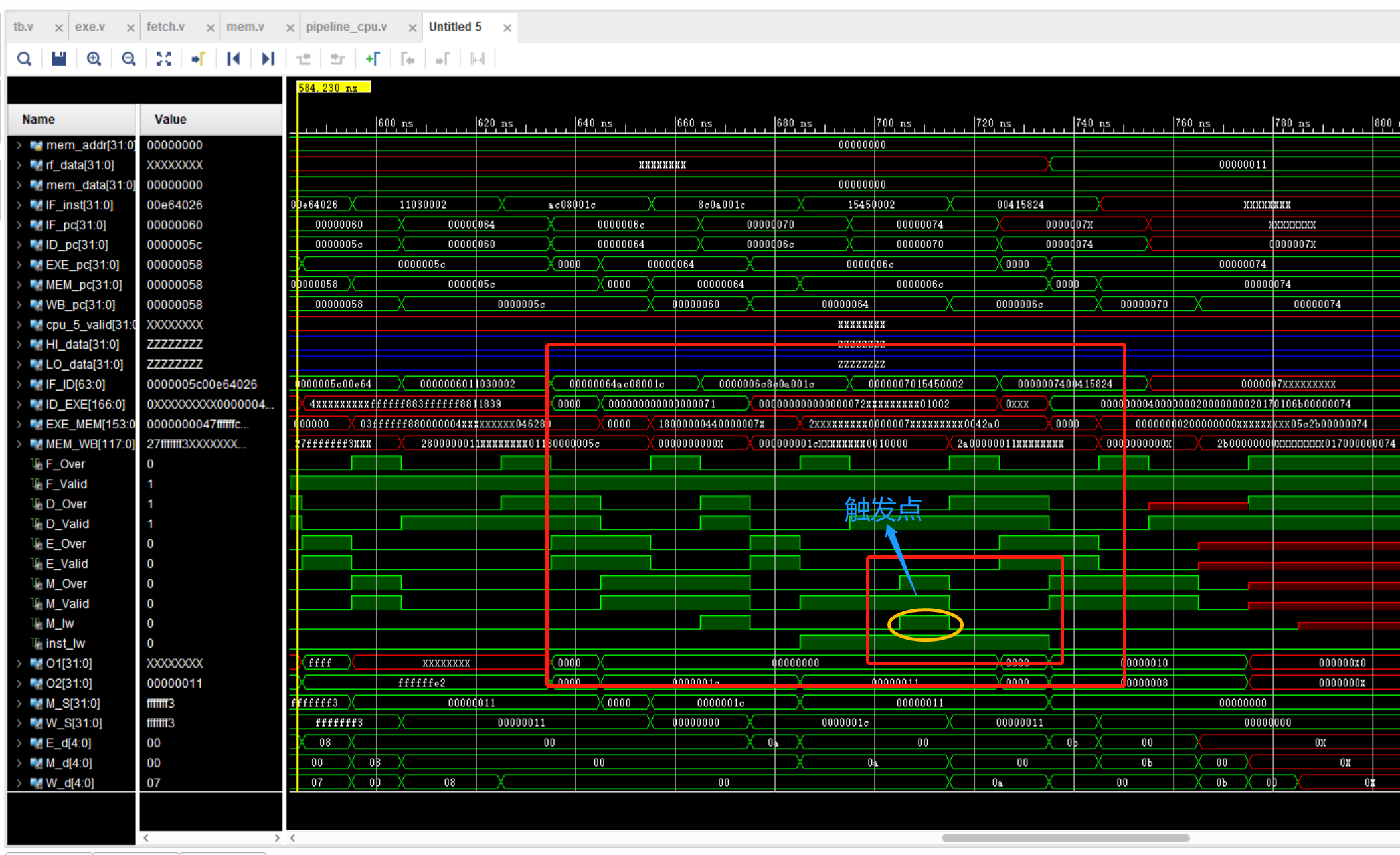
对lw指令进行单独的处理，如果发生该情况，那么就要延迟ID解码阶段的进入，也即是延缓IF\_over使能信号的更新。



延迟设计代码思路：只有在当前指令为load加载类型指令时，才会触发上述设计的延迟，那么通过代码分析得知在mem阶段对控制信号解读时会对其是否为加载指令进行判断。

assign {inst\_load,inst\_store,ls\_word,lb\_sign} = mem\_control;

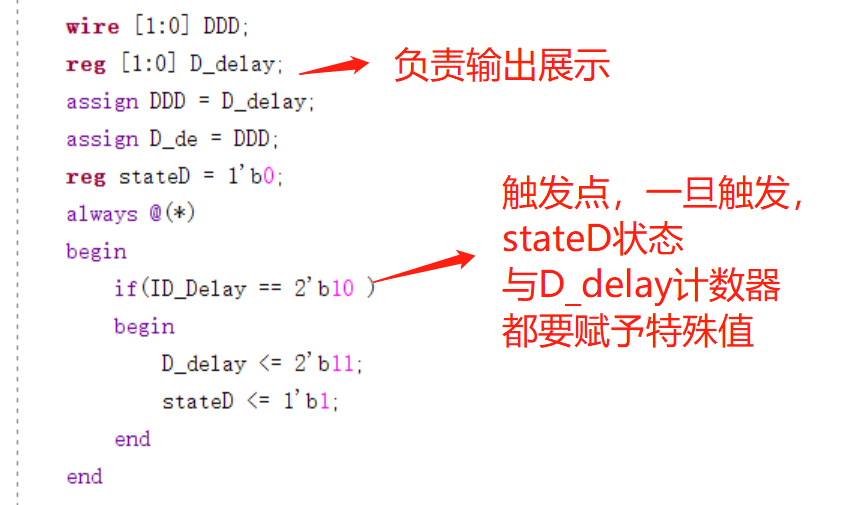
至此，得知inst\_load和mem\_valid双重有效时，则就要触发延迟机制。

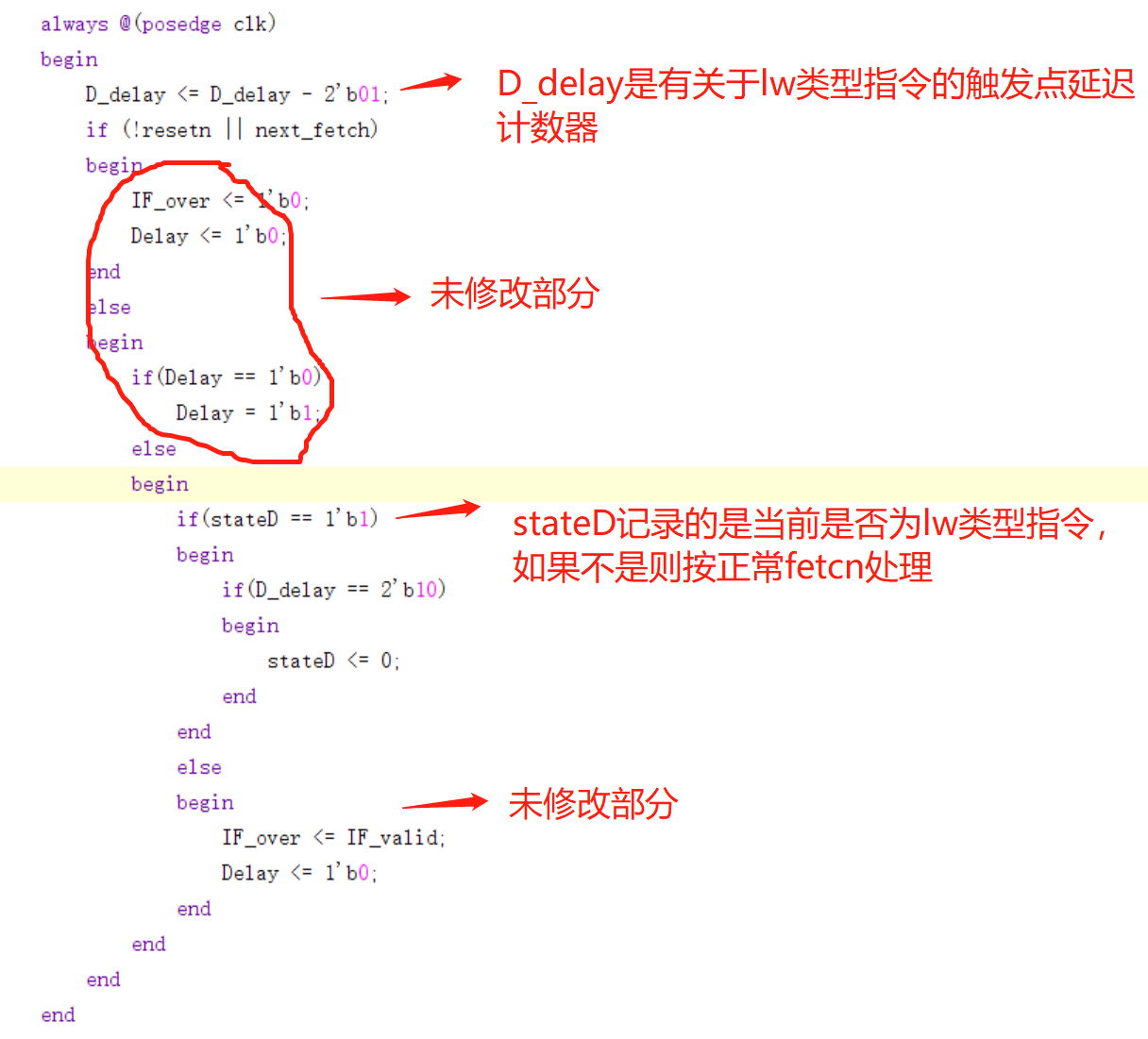


通过使能信号的传递至IF阶段，再结合clk寻找代码的触碰时机。在IF阶段的IF\_over信号实现延迟，如果当前mem阶段正在执行的是load加载数据类型的指令时。

最终设计代码成功实现load加载数据时旁路的正确执行，流水线的正常工作。

修改代码设计：

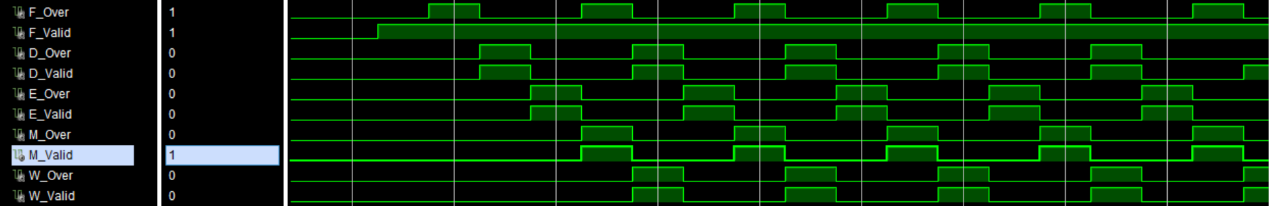




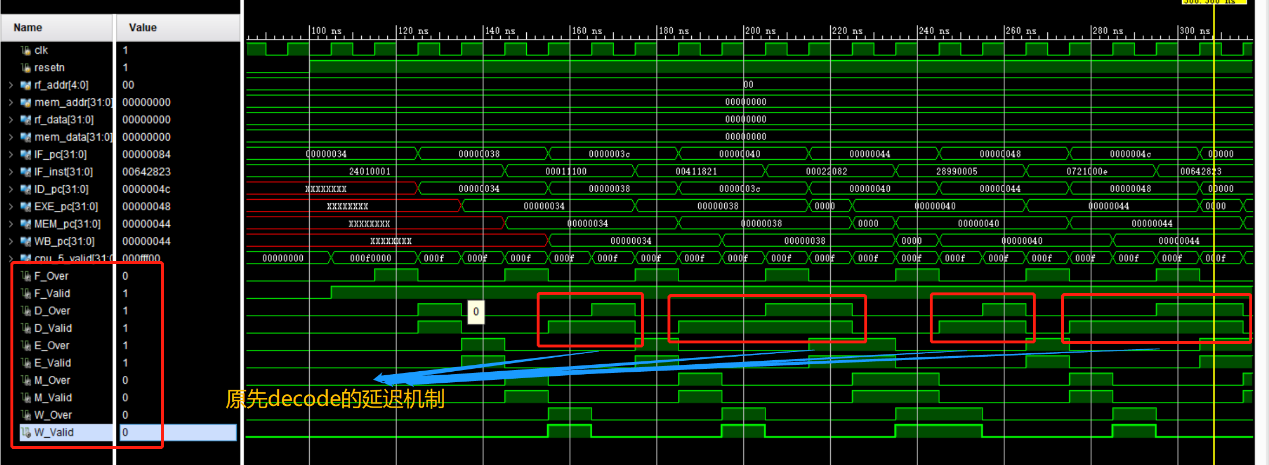
**旁路的优化效果展示：**

**（截取前5条明显的数据相关进行清晰的展示）**

最终旁路执行周期：



原始执行周期：



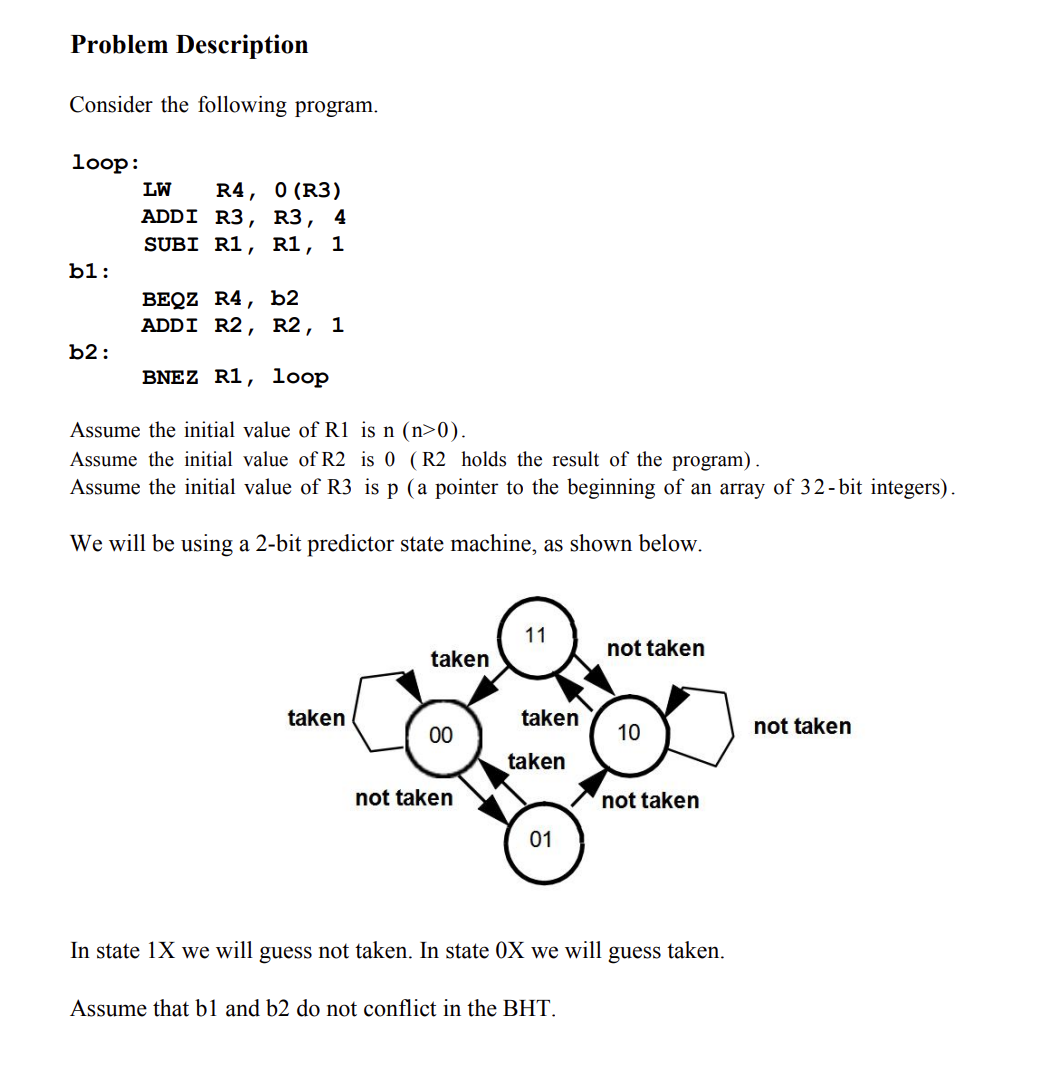
很明显的可以看出修改添加旁路后的优化效果。关于数据相关的冲突可以很好的通过旁路避免阻塞，加速流水线机制的执行，不需要额外的阻塞停等。

1. 分支预测Branch Prediction设计

（**注：该部分是在同学王旭尧（2012527）的指导帮助下完成的，在本次实验中作者着重负责了旁路的实验设计，分支预测则是由王旭尧同学攻坚设计**）

结合体系结构课程上学习到的理论知识：**基于俩位状态机的分支预测**，设计流水线相关的预测机制。

由于我们实验设计时的测试样例中没有MIPS的延迟槽设计，故代码设计中也不涉及延迟槽。



设计的算法状态机如题目中描述的一样，在00、01、10、11四个状态下根据实际的分支跳转进行状态转移。测试指令如下：

0 : 24010008; % main: addiu $1, $0, 8 | 00100100000000010000000000001000 %

1 : 24020000; % addiu $2, $0, 1 | 00100100000000100000000000000001 %

2 : 24050001; % addiu $5, $0, 1 | 00100100000001010000000000000001 %

3 : 24030000; % addiu $3, $0, 0 | 00100100000000110000000000000000 %

4 : ac050000; % sw $5, 0($0) | 10101100000001010000000000000000 %

5 : ac000004; % sw $0, 4($0) | 10101100000000000000000000000100 %

6 : ac050008; % sw $5, 8($0) | 10101100000001010000000000001000 %

7 : ac00000c; % sw $0, 12($0) | 10101100000000000000000000001100 %

8 : ac050010; % sw $5, 16($0) | 10101100000001010000000000010000 %

9 : ac000014; % sw $0, 20($0) | 10101100000000000000000000010100 %

a : ac050018; % sw $5, 24($0) | 10101100000001010000000000011000 %

b : ac00001c; % sw $0, 28($0) | 10101100000000000000000000011100 %

c : 8c640000; % loop: lw $4, 0($3) | 10001100011001000000000000000000 %

d : 24630004; % addiu $3, $3, 4 | 00100100011000110000000000000100 %

e : 00250823; % subu $1, $1, $5 | 00000000001001010000100000100011 %

f : 10800002; % b1: beq $4, $0, b2 | 00010000100000000000000000000010 %

10 : 24e70000; % addiu $7, $7, 0 | 00100100111001110000000000000000 %

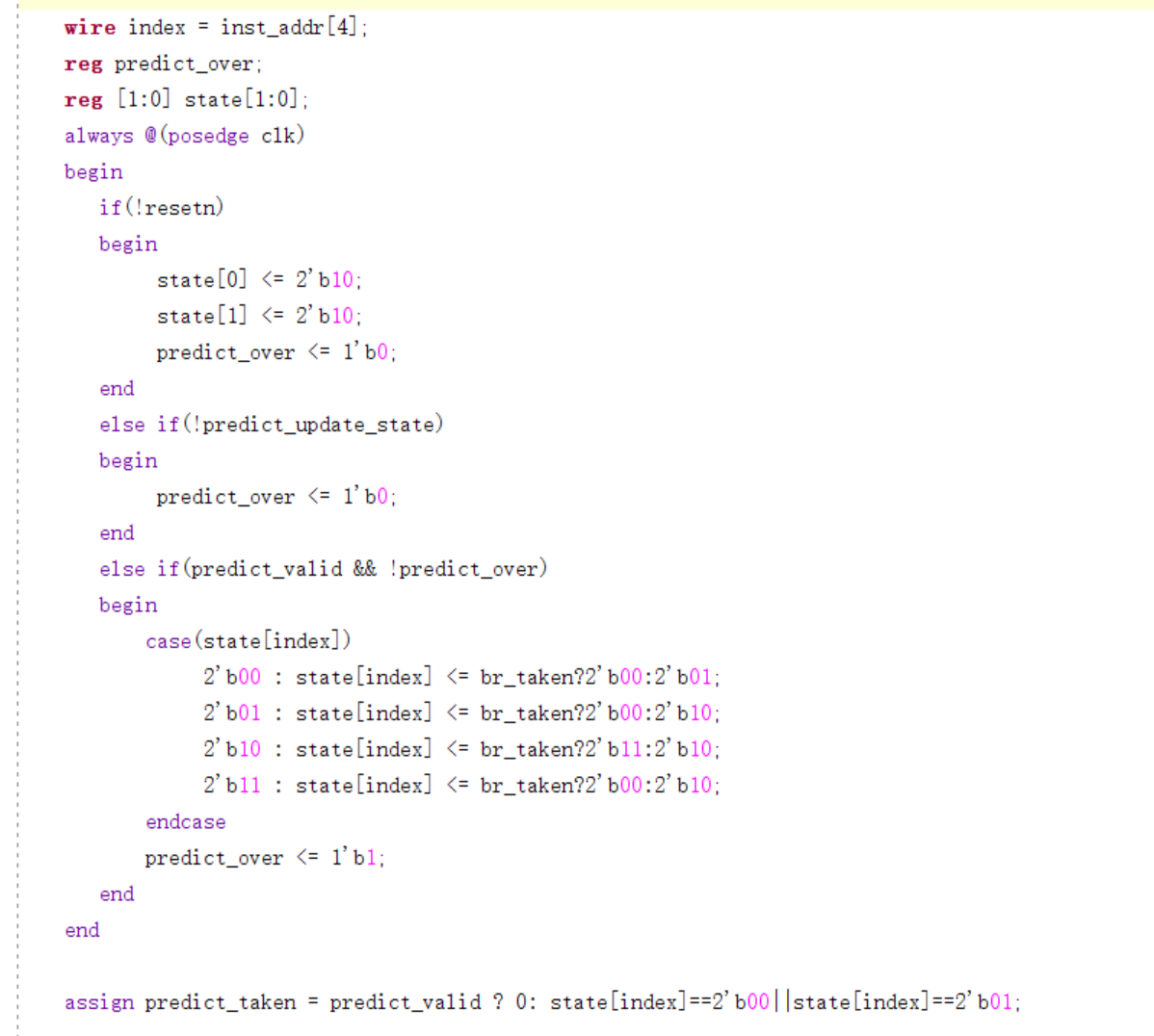
11 : 24420001; % addiu $2, $2, 1 | 00100100010000100000000000000001 %

12 : 1420fff9; % b2: bne $1, $0, loop | 00010100001000001111111111111001 %

13 : 24e70000; % addiu $7, $7, 0 | 00100100111001110000000000000000 %

14 : 08000000; % j main | 00001000000000000000000000000000 %

核心代码部分：



完整的分支预测实验设计部分以及效果图可参见王旭尧同学设计报告或者参见[GitHub仓库。](https://github.com/XuyaoWang/pipeline/tree/predict)

1. 实验总结感悟

本次实验完成过程中，作者对有关于CPU流水线的设计的理论知识进一步的掌握了解。本次实验中作者感到尤其重要的是，作者在一次次错误的仿真图中，不断根据各个变量值的改变分析设计中存在缺陷的过程。在每一次与董老师讨论咨询的过程中，在每一次修改重新仿真的过程中，作者利用vivado设计pipeline流水线CPU的能力都会更进一步。

唯有一次次的修改与仿真，在仿真中观察分析，才能真正的掌握流水线CPU在每一个时钟周期到底进行了什么操作。

这次的实验让作者感悟颇深，也加深了作者对于现代CPU设计的敬佩！

附上本人GitHub链接：[kyrielw24/computer-architecture (github.com)](https://github.com/kyrielw24/computer-architecture)