

**计算机组成·本科实验报告**

|  |  |
| --- | --- |
| 姓 名： | 李欣奇 |
| 学 院： | 信息与电子工程学院 |
| 专 业： | 微电子科学与工程 |
| 学 号： | 3210105139 |
| 指导教师： | 黄科杰 |

2024 年 4 月 21 日

**实验报告**

日期：2024.4.21

地点：

课程名称： 计算机组成 指导老师： 黄科杰

实验名称： 单周期RISC-V仿真

1. **实验目标**
2. **实验环境和工具**
3. **实验内容和过程（总结自己所做的三个阶段工作）**
4. **实验总结（说说踩的坑，总结收获，分析下自己花了多少时间，都用来做什么事情）**
5. **提出改进实验的意见**

一、实验目标

用verilog实现RV32I **单周期CPU**。

二、实验环境和工具

Vivado 2023.2

Ubuntu -64 22.04.3 LTS

三、实验内容和过程

1. 阶段1、阶段2，无HazardUnit

测试代码位于：

2\_BRAMInputFileGenerator\Mycode\test\_part2\test\_part2\_2.S

这份测试代码是从hw1的单周期的test\_part2\_1.S在每条命令间加入3个nop命令得到的。

问题1：

问题描述：

lui t0, 0x1 # Load upper immediate (t0 = 0x1000)

addi s1, t0, 4 # s1 = t0 + 4 = 0x1004

无法加载出存储在t0中的值（t0=0x1000），如图1所示。

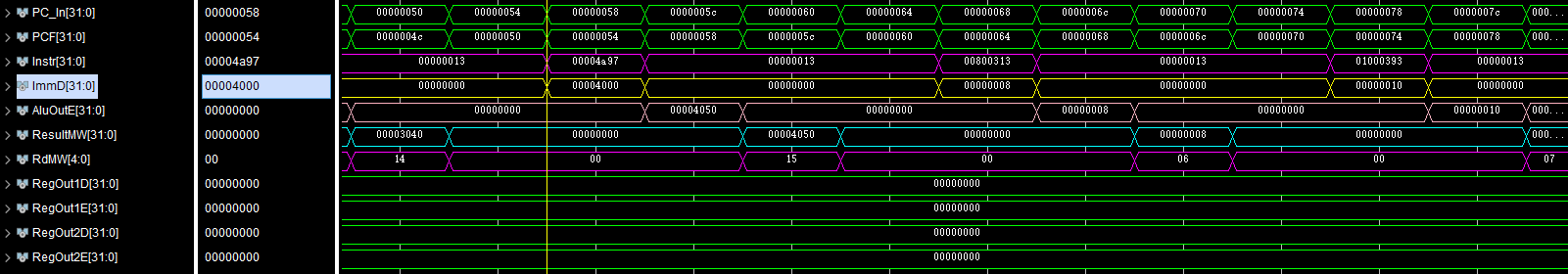


图 1

问题分析：

如图2所示，在RegisterFile信号中，A3地址信号，WD3数据信号和WE3写使能信号都正确，但注意到reset此时还是高电平，因此需要修改系统的复位逻辑。

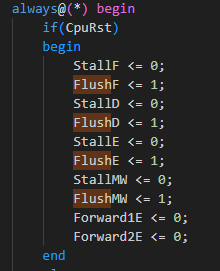


图 3

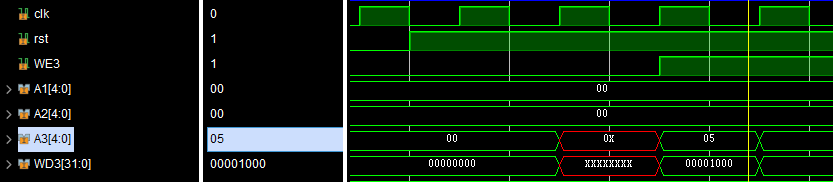


图 2

修改后的代码如右图所示。在Hazard文件的注释中有这一段话：

    //可以最后实现。前期测试CPU正确性时，可以在每两条指令间插入四条空指令，然后直接把本模块输出定为，不forward，不stall，不flush

但这句话是错的，不stall不flush带来的结果就是reset一旦激活程序就立刻开始执行，这导致了RegFile以及其他有reset信号输入的模块本身还在reset的状态就要执行命令。这会导致reset激活状态期间运行错误。

修改后命令运行正常。如图4所示。

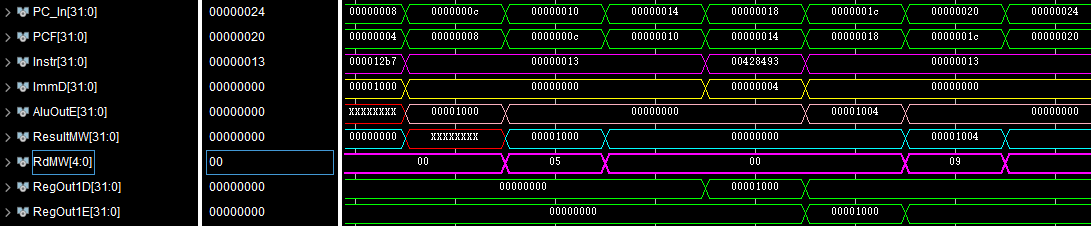


图 4

2. 加入HazardUnit后的测试代码：

2\_BRAMInputFileGenerator\Mycode\test\_part3\test\_part3\_1.txt：

00010054 <\_start>:

10054: 00400493 li s1,4

10058: 00500913 li s2,5

1005c: 00600a13 li s4,6

10060: 012482b3 add t0,s1,s2

10064: 414289b3 sub s3,t0,s4

10068: 00000013 nop

1006c: 0054a023 sw t0,0(s1)

10070: 0004a303 lw t1,0(s1)

10074: 009363b3 or t2,t1,s1

问题2：

问题描述：

sw t0, 0(t1) # Save word from t0 to [t1]

lw s8, 0(t1) # Load word into t8 from [t1]

所有的S-type， I-type中的load中的命令出现问题，具体问题定位查看问题分析。

问题分析：

如图5所示。MemWriteM信号传输不正确（高阻）。但后续的调查发现此信号没有传输到任何模块。注释掉此信号的声明后重新仿真，问题依然存在。

一个线索发现本该在MW阶段执行的Mem写入在Ex阶段就执行了，可能导致错误。（如图6所示）

后续的调查发现其实是MemToReg使能信号在MW阶段才到达，导致了使能信号和数据信号错位，因此RegisterFile的WD3接口无法接受到正确的数据信号。

修改方案：

将原来的控制Mem的EN，addr，data信号全部设为reg，在时钟的上升沿来的时候才改变，从而使其同步在MW阶段完成Mem的操作。



图 6

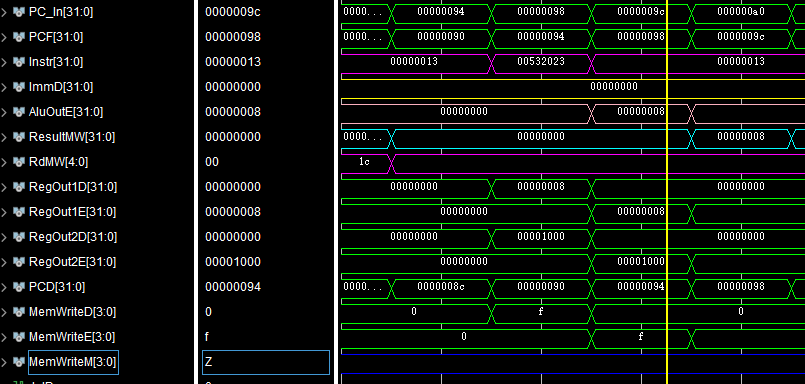


图 5

3. 综合测试

Error 1：

000105d8 <test\_38>:

105d8: 01000093 li ra,16

105dc: 01e00113 li sp,30

105e0: 00208033 add zero,ra,sp

105e4: 00000e93 li t4,0

105e8: 02600193 li gp,38

105ec: 01d00463 beq zero,t4,105f4 <test\_39>

105f0: 5590106f j 12348 <fail>



图 7

Test解释：

li ra,16 加载立即数，把16加载到寄存器ra中。

li sp,30 加载立即数，把30加载到寄存器sp中。

add zero,ra,sp 把ra和sp加起来，然后把结果存入zero寄存器。但这实际上应该没有效果，因为无法改变zero寄存器的值--它永远都是0。看起来这只是检查加法器的一个技巧。

li t4,0 加载立即，把0加载到寄存器t4中。

li gp,38 加载立即，把38加载到寄存器gp中。

beq zero,t4,105f4 <test\_39> 如果zero和t4相等（实际上肯定相等，因为它们都是0），则跳转到105f4地址，这就是一个条件分支。

若不满足条件分支，则会执行 j 12348 <fail>，无条件跳转到 12348。

显然是写入zero成功了（图7中的0000002e）导致了这个问题。

解决方法：

在MWSegReg中将MemToRegMW变量的直接传递加上一个判断条件：

MemToRegMW <= ((RdE)?MemToRegE:0);

加入后发现问题没有得到解决，再次调查发现竟然在某一个时刻t4寄存器才是那个被写入0000002e的寄存器，从而导致了ble的错误。

再经过长时间的debug，发现ALU的输出是0000002e，但立即数是0，而Operand是0000002e；ForwardData1是0000002e，但RegOut1是0，从而定位到是HazardUnit中Forward1E信号产生的逻辑错误。

原来的判断条件是：

if ( !MemToRegE && (((RegReadE[1]) && (Rs1E == RdMW) && (RegWriteMW != 3'b0)) || ((RegReadE[0])  &&(Rs2E == RdMW) && (RegWriteMW != 3'b0) )))

但其实应该是

if ( **!MemToRegMW** && (((RegReadE[1]) && (Rs1E == RdMW) && (RegWriteMW != 3'b0)) || ((RegReadE[0])  &&(Rs2E == RdMW) && (RegWriteMW != 3'b0) )))

注意标红的部分的区别，这个错误产生的原因仅仅是错误的把MemToRegMW以为成了MemToRegE。加入MemToReg到这个判断条件的原因是用于区别**可以前递**的情况和**只能Stall**的情况。因为如何前面那条命令的结果是从Memory中读出来的，那么前递传递的是地址而不是读出来的值。而又因为没有第二条前递的数据通路，因此需要从Mem中读数据的命令就不能前递。因此判断条件中加入了**!MemToRegMW**

然而可惜的是这个test产生错误的根本原因还是因为当Rs==0时前递的数据是无效的。因此需要加入对Rs的判断。

Error2：

00010fe4 <test\_102>:

10fe4: 00000013 nop

10fe8: 00002517 auipc a0,0x2

10fec: 71c50513 addi a0,a0,1820 # 13704 <\_\_bss\_start+0x27c>

10ff0: 004005ef jal a1,10ff4 <test\_102+0x10>

10ff4: 40b50533 sub a0,a0,a1

10ff8: 00002eb7 lui t4,0x2

10ffc: 710e8e93 addi t4,t4,1808 # 2710 <\_start-0xd970>

11000: 06600193 li gp,102

11004: 01d50463 beq a0,t4,1100c <test\_103>

11008: 3400106f j 12348 <fail>

Test解释：

nop 无操作。

auipc a0,0x2 把立即数放大（左移）12位（前20位）再加上当前PC赋值给a0寄存器。

addi a0,a0,1820 把a0寄存器的值和即数1820相加，结果存回a0寄存器。

jal a1,10ff4 <test\_102+0x10> 跳转并链接，跳转到10ff4地址执行，并把下一条指令的地址（也就是此条指令的地址加4）保存到a1寄存器。

sub a0,a0,a1 把a0寄存器的值和a1寄存器的值进行减法运算，结果保存在a0寄存器。

lui t4,0x2 把立即数（0x2）左移20位存到t4寄存器。

addi t4,t4,1808 把t4寄存器的值和即数1808相加，结果存回t4寄存器。

li gp,102 把立即数102加载到gp寄存器。

beq a0,t4,1100c <test\_103> 如果a0寄存器的值等于t4寄存器的值，则跳转到1100c的地址执行。这是一个条件跳转。

如果上面的比较条件不成立，则执行j 12348 <fail>，无条件跳转到12348地址，执行fail的代码。

产生这样的错误的原因是原来的代码中ForwarE的两个信号会阻碍JAL信号的产生，导致没有Flush信号产生。

修改方案：

将Flush、Stall信号与Forward的信号的产生逻辑分开，写两个always块，使其产生逻辑并行（Forward可以和Flush、Stall信号共存）

Error3：

00012248 <test\_232>:

12248: 00100093 li ra,1

1224c: 0140006f j 12260 <test\_232+0x18>

12250: 00108093 addi ra,ra,1

12254: 00108093 addi ra,ra,1

12258: 00108093 addi ra,ra,1

1225c: 00108093 addi ra,ra,1

12260: 00108093 addi ra,ra,1

12264: 00108093 addi ra,ra,1

12268: 00300e93 li t4,3

1226c: 0e800193 li gp,232

12270: 0dd09c63 bne ra,t4,12348 <fail>

00012274 <jalr\_test\_2>:

12274: 00200193 li gp,2

12278: 00000293 li t0,0

1227c: 00000317 auipc t1,0x0

12280: 01030313 addi t1,t1,16 # 1228c <jalr\_target\_2>

12284: 000302e7 jalr t0,t1

00012288 <jalr\_linkaddr\_2>:

12288: 0c00006f j 12348 <fail>

0001228c <jalr\_target\_2>:

1228c: 00000317 auipc t1,0x0

12290: ffc30313 addi t1,t1,-4 # 12288 <jalr\_linkaddr\_2>

12294: 0a629a63 bne t0,t1,12348 <fail>

问题出现在0x1227c到0x12288这三行，fail从0x12288跳到。

问题定位：

经过分析发现t1的前递、ALU值都是正确的，FlushE、FlushD信号也产生正常，如果正常执行jalr命令，那么就不会执行12288，会直接跳到1228c。因此有可能是12288的j指令抢占了12284的jalr。因为jalD和jalrE其实是同时产生的。因此考察NPCGenerator模块中，源代码如下：

    always @(\*) begin

        if (BranchE)

            PC\_In <= BranchTarget;

        else if (JalD)

            PC\_In <= JalTarget;

        else if (JalrE)

            PC\_In <= JalrTarget;

        else

            PC\_In <= PCF+4;

    end

只需要把jalrE提前，和jalD换一下位置即可。

解决方案：

    always @(\*) begin

        if (BranchE)

            PC\_In <= BranchTarget;

        else if (JalrE)

            PC\_In <= JalrTarget;

        else if (JalD)

            PC\_In <= JalTarget;

        else

            PC\_In <= PCF+4;

    end

至此1testAll全部成功。如图8所示。

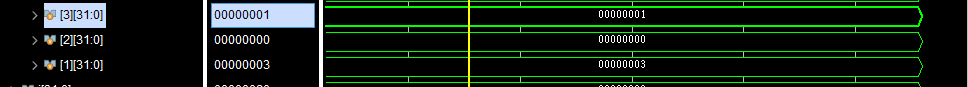


图 8

2testAll成功。如图9所示。

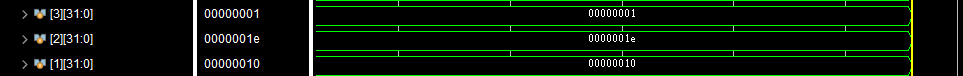


图 9

3testAll成功。如图10所示。

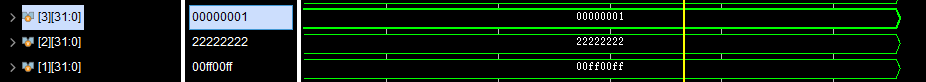


图 10

四、实验总结

（1）无HazardUnit代码编写阶段

代码编写阶段总体花费时间不长，在8h以内。有了第一次的经验，在编写流水线的代码时，更加注重细节。因此在无HazardUnit的测试部分消耗的时间相对第一次少了很多。

此外，这个部分的代码编写更多是直接在单周期的代码的基础上进行修改，只不过要非常注意各个变量的各个阶段，需要仔细想想，最好是和CPU的设计图一条一条信号的比较。

（2）HazardUnit代码编写阶段和调试阶段

HazardUnit代码编写和调试是这次流水线编写的重中之重，因此很多的时间都花在了这个部分。最初我的设计是只用了一个always块分case对stall，flush，forward信号赋值，但后来发现其实forward信号的赋值其实是和stall和flush分开的。此外，我还修改了Core部分的代码，因为在forward的判断条件额外增加了一个MemToReg接口。

以下是完整代码：

    //Stall and Flush signals generate

    always@(\*) begin

        if(CpuRst)

        begin

            StallF <= 0;

            FlushF <= 1;

            StallD <= 0;

            FlushD <= 1;

            StallE <= 0;

            FlushE <= 1;

            StallMW <= 0;

            FlushMW <= 1;

        end

        else

        begin

            if(BranchE || JalrE)

            begin

                StallF <= 0;

                StallD <= 0;

                StallE <= 0;

                StallMW <= 0;

                FlushF <= 0;

                FlushD <= 1;

                FlushE <= 1;

                FlushMW <= 0;

            end //branch

            else if(JalD)

            begin

                StallF <= 0;

                StallD <= 0;

                StallE <= 0;

                StallMW <= 0;

                FlushF <= 0;

                FlushD <= 1;

                FlushE <= 0;

                FlushMW <= 0;

            end //jal

            else if (MemToRegE && ((RdE == Rs1D && Rs1D!=0) || (RdE == Rs2D && Rs2D!=0)))

            begin

                StallF <= 1;

                StallD <= 1;

                StallE <= 0;

                StallMW <= 0;

                FlushF <= 0;

                FlushD <= 0;

                FlushE <= 0;

                FlushMW <= 0;

            end //stall

            else

            begin

                StallF <= 0;

                FlushF <= 0;

                StallD <= 0;

                FlushD <= 0;

                StallE <= 0;

                FlushE <= 0;

                StallMW <= 0;

                FlushMW <= 0;

            end

        end

    end

    //Forward Register Source 1

    always@(\*) begin

        if(CpuRst)

        begin

            Forward1E <= 0;

            Forward2E <= 0;

        end

        else

        begin

             if ( !MemToRegMW && (((RegReadE[1]) && Rs1E!=0 && (Rs1E == RdMW) && (RegWriteMW != 3'b0)) || ((RegReadE[0]) && Rs2E!=0 &&(Rs2E == RdMW) && (RegWriteMW != 3'b0))))

             begin

                Forward1E <= (Rs1E == RdMW) ? 1'b1 : 1'b0;

                Forward2E <= (Rs2E == RdMW) ? 1'b1 : 1'b0;

             end

             else

             begin

                Forward1E <= 0;

                Forward2E <= 0;

             end

        end

    end

此阶段用时10h以内。

（3）官方代码调试阶段

此阶段用时可以忽略不计。

由于在（2）中就以及利用到了综合调试的1testAll的代码，在通过了HazardUnit的调试后，2testAll和3testAll直接通过了，没有遇到bug。

五、实验改进

1. 在Hazard文件的注释中有这一段话：

    //可以最后实现。前期测试CPU正确性时，可以在每两条指令间插入四条空指令，然后直接把本模块输出定为，不forward，不stall，不flush

但这句话是错的，不stall不flush带来的结果就是reset一旦激活程序就立刻开始执行，这导致了RegFile以及其他有reset信号输入的模块本身还在reset的状态就要执行命令。这会导致reset激活状态期间运行错误。