

**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

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。

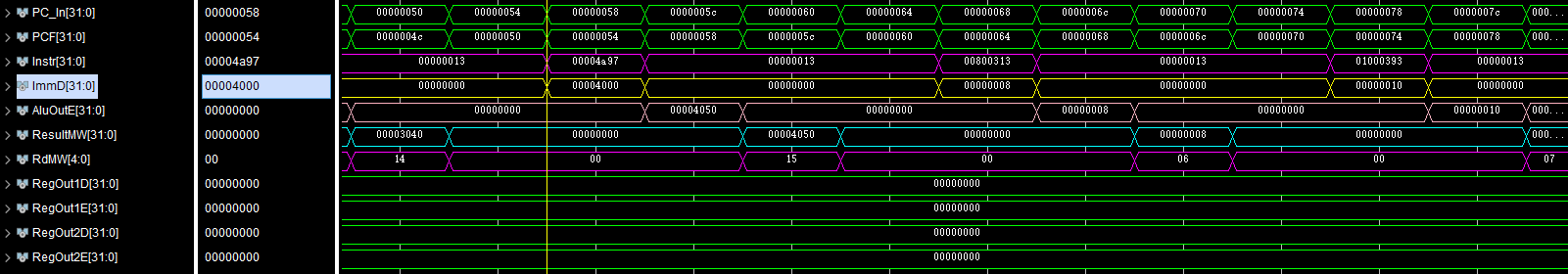


图 1

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。

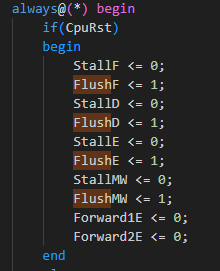


图 3

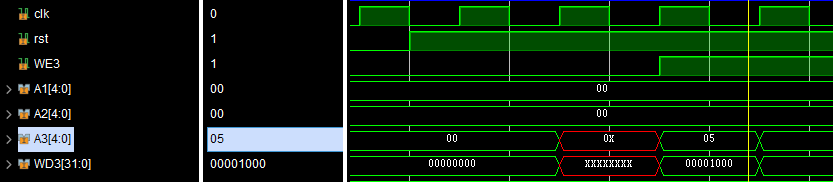


图 2

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。

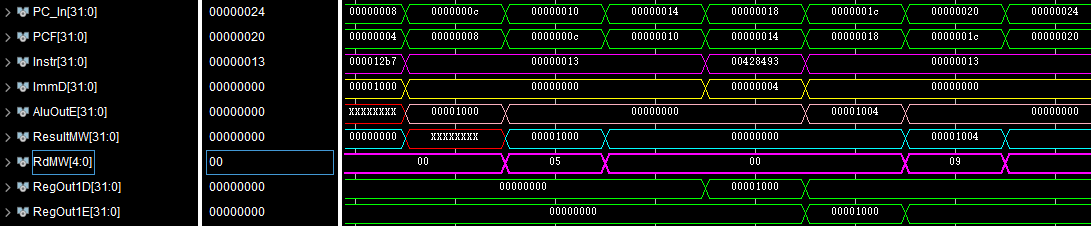


图 4

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

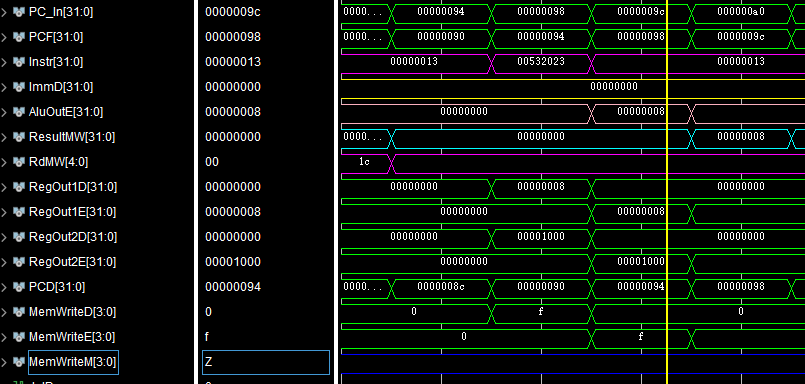


图 5

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 把当前指令地址放大（左移）12位赋值给a0寄存器。这里的0x2是立即数，表示放大两次，即左移24位。

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。在这里# 13704 <\_\_bss\_start+0x27c>是注释，它表示这条指令的目的，实际的汇编代码执行并不会考虑这种注释。

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。

四、实验总结

（1）代码编写阶段

代码编写阶段总体花费时间不长，在8h以内。基本上就是根据机器码、硬件实现图将其代码化，有一些课堂上没有讲的细节需要花一点时间。例如代码Core中Branch实现的方式和课堂上讲授的连线有所区别，但实际测试发现二者都是正确的。再例如在DataRam的实现的具体细节上，由于其是32位划分的，在处理地址A[1:0]不等于0以及SB，SH的时候需要重新考虑WE，WD的值（由于A直接舍弃了末两位）。总之代码写的时候越小心，debug遇到的问题就越少，写的不仔细，debug就会付出很大的代价。我就是后者。

  wire [3:0] wea;

    wire [31:0] dina;

    assign wea = WE<<A[1:0];

    assign dina = WD<<(A[1:0]\*8);

    DataRam DataRamInst (

        .clk    (clk),                      //请补全

        .wea    (wea),                      //请补全

        .addra  (A[31:2]),                      //请补全

        .dina   (dina),                      //请补全

        .douta  ( RD\_raw         ),

        .web    ( WE2            ),

        .addrb  ( A2[31:2]       ),

        .dinb   ( WD2            ),

        .doutb  ( RD2            )

    );

上图是DataSeg的部分代码。

（2）代码自行调试阶段

对于Part2部分的调试花费了比较多的时间，一部分是编写汇编时编写太多跳转代码比较容易搞混的问题，另一部分则是Debug不知道程序跳到哪里去的问题。总之这部分花费的时间是在意料之中的，带来的收获则是B-type指令和JAL，JALR几乎在后续的debug中没有出问题。

此外，还需要提到的是，在debug期间，由于我改变了Core，加入了新的ImmOperandUnit，在写这部分代码时比较糟糕，导致调试的时候发现许多情况下Imm加起来都不是32位而是31位！好在这样的错误比较容易发现。

此阶段用时6h以内。

（3）官方代码调试阶段

此阶段用时无法统计。

在调试test2时遇到了超级多的bug，而其大部分都是关于SB，LB的。此外还有AUIPC、LUI、BEQ、LH、SLT等命令的。以下是几个耗时比较久的debug测试点：

1test128, 2test274, 2test310, 2test372, 2test373, 2test375, 2test491

其中标红的三个测试点都是SB出了问题。这个问题的难点在于，由于其前后指令就是SB，LB，如果不深入底层的DataRamInst比较难以判断究竟是save的问题还是load的问题。常常在判断bug的位置就花费了很多的时间，而改这个bug又由于有一个\*8的操作太过扎眼想改掉但又难以想到更简单的方案（<<3由于位数限制不可行，除非另外再声明）。

五、实验改进

1. 课堂上讲的端口设计和连接方式可以和代码尽量统一（Branch）

2. 代码具体实现上的细节课上可以提一下，例如比较困难的DataExt和DataRamIn的预处理

3. 单周期完全可以把ImmOperandUnit加上去，方便后面Pipeline的实验。