**计算机系统结构试验**

**Lab06: 简单的类 MIPS 多周期流水化处理器的实现**

姓名：N/A

摘要

在Lab06中，我实现了MIPS多周期流水化处理器的主要功能，使用Forwarding和Stall机制解决了Data Hazard和Control Hazard，使用分支预测减少Control Hazard，并在此基础之上将指令集扩展到了31条。实验包括了Instruction Fetch、Instruction Decode、Execution、Memory和Buffers等模块的设计和实现，以及修改Lab5中的其余模块。通过本次实验，我进一步学习了对Verilog语言的运用，掌握MIPS流水线处理器的组成、设计和实现方法，并且深入理解了流水线Stall和Forwarding机制，给我带来宝贵的经验和收获。

目录

[摘要 1](#_Toc168264842)

[**1.** 实验目的 3](#_Toc168264843)

[**2.** 原理分析 3](#_Toc168264844)

[2.1 Vivado工程的基本组成 3](#_Toc168264845)

[2.2 流水化处理器的原理 3](#_Toc168264846)

[2.2.1 指令流水线 3](#_Toc168264847)

[2.2.2 流水线寄存器 4](#_Toc168264848)

[2.2.3 时序控制 4](#_Toc168264849)

[2.3 控制冒险的处理 4](#_Toc168264850)

[2.3.1 Jump指令控制冒险 4](#_Toc168264851)

[2.3.2 Branch指令控制冒险 5](#_Toc168264852)

[2.4 数据冒险的处理 5](#_Toc168264853)

[2.4.1 寄存器Load-Use冒险 5](#_Toc168264854)

[2.4.2 内存Load-Use冒险 6](#_Toc168264855)

[2.5 分支预测的原理 6](#_Toc168264856)

[**3.** 功能实现 6](#_Toc168264857)

[3.1 流水线处理器的实现 6](#_Toc168264858)

[3.1.1 IF阶段 6](#_Toc168264859)

[3.1.2 ID/WB阶段 7](#_Toc168264860)

[3.1.3 EX阶段 8](#_Toc168264861)

[3.1.4 MEM阶段 9](#_Toc168264862)

[3.1.5 流水线寄存器 9](#_Toc168264863)

[3.2 消除控制冒险的实现 10](#_Toc168264864)

[3.3 消除数据冒险的实现 10](#_Toc168264865)

[3.3.1 消除寄存器Load-Use冒险 10](#_Toc168264866)

[3.3.2 消除内存Load-Use冒险 11](#_Toc168264867)

[3.4 分支预测的实现 11](#_Toc168264868)

[**4.** 结果验证 12](#_Toc168264869)

[4.1 提供的测试代码 12](#_Toc168264870)

[4.2 冒险与分支预测的测试 13](#_Toc168264871)

[4.3 其他指令的测试 14](#_Toc168264872)

[4.3.1 jr，j和jal的测试 14](#_Toc168264873)

[4.3.2 sll，srl和sra的测试 14](#_Toc168264874)

[4.3.3 addi，addiu和lui的测试 15](#_Toc168264875)

[4.3.4 slti和sltiu的测试 15](#_Toc168264876)

[4.5 综合测试 15](#_Toc168264877)

[**5.** 总结与反思 16](#_Toc168264878)

**1.** 实验目的

（1）理解CPU Pipeline、流水线冒险(hazard)及相关性，在Lab5基础上设计简单流水线CPU；

（2）在1的基础上设计支持Stall的流水线CPU。通过检测竞争并插入停顿（Stall）机制解决数据冒险/竞争、控制冒险和结构冒险；

（3）在2的基础上，增加Forwarding机制解决数据竞争，减少因数据竞争带来的流水线停顿延时，提高流水线处理器性能；

（4）在3的基础上，通过predict-not-taken或延时转移策略解决控制冒险/竞

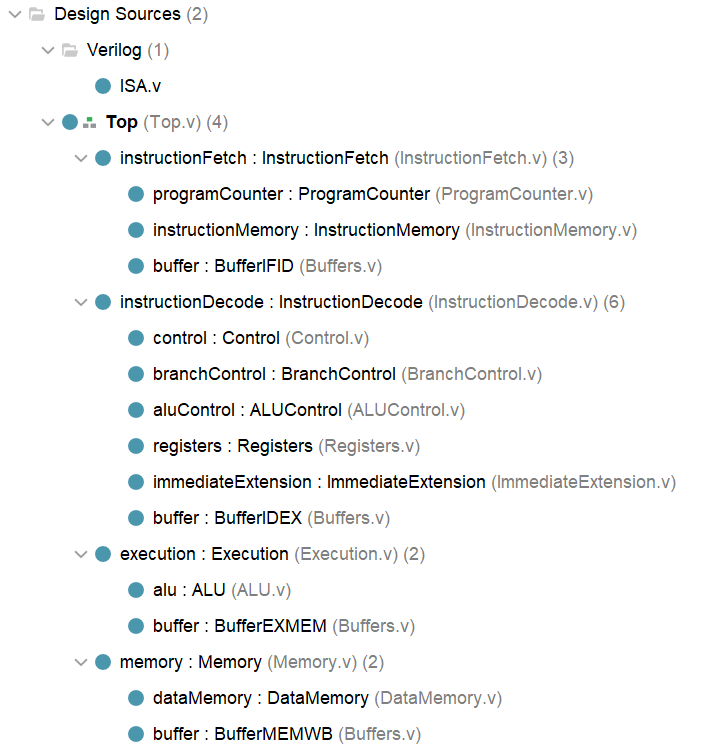
争，减少控制竞争带来的流水线停顿延时，进一步提高处理器性能；

（5）在4的基础上，将CPU支持的指令数量从9条或16条扩充为31条，使处理器功能更加丰富。

**2.** 原理分析

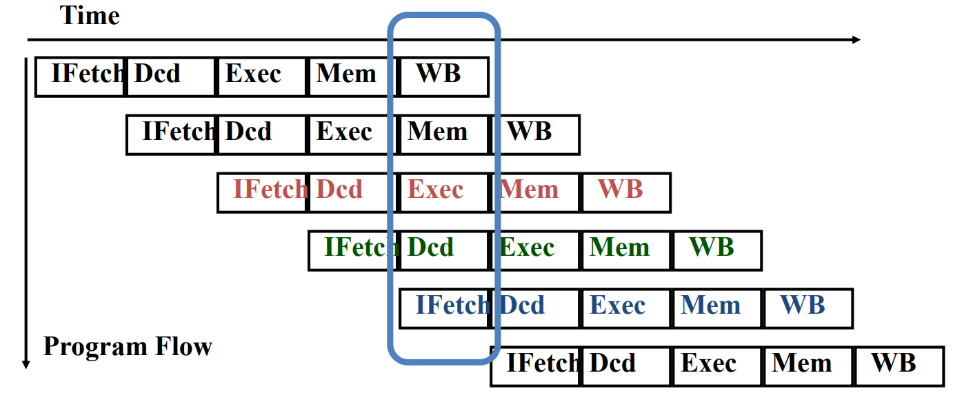
2.1 Vivado工程的基本组成

Vivado工程的结构如下：



2.2 流水化处理器的原理

2.2.1 指令流水线



MIPS多周期流水线处理器将指令的执行分成5个阶段：

（1）取指阶段(IF)：从指令存储器中取出指令。

（2）译码阶段(ID)：指令译码、寄存器读取，生成控制信号。

（3）执行阶段(EX)：读取操作数，进行算术或逻辑运算，或计算内存地址。

（4）访存阶段(MEM)：读取内存地址，执行内存读/写操作。

（5）写回阶段(WB)：读取计算结果或内存数据，将结果写回寄存器文件。

2.2.2 流水线寄存器

图示

描述已自动生成

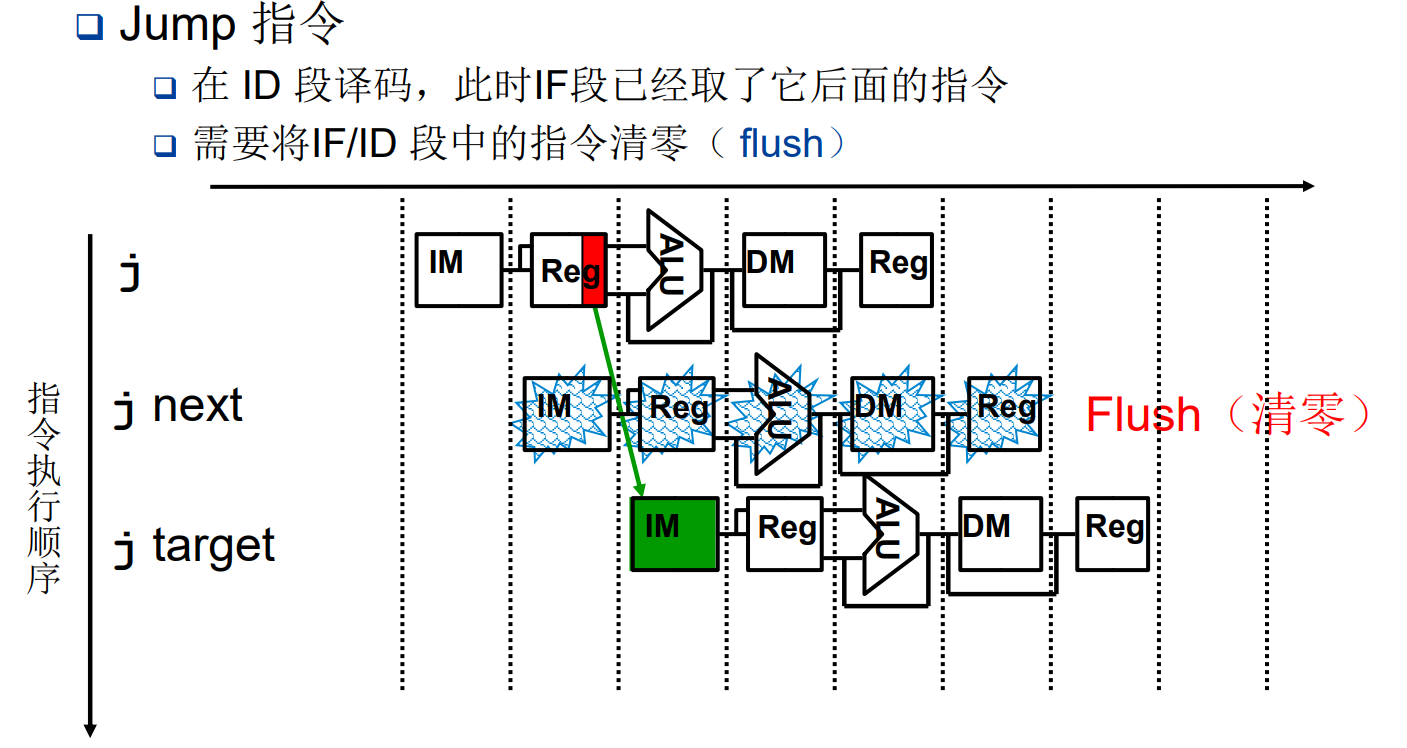
为了在各个阶段之间传递指令信息，在每个阶段之间引入流水线寄存器，用来存储指令的某些部分、控制信号和中间结果，确保在不同的周期里各阶段之间的数据信息可以正确传递。图中展示了四个关键的流水线寄存器：IF/ID, ID/EX, EX/MEM和MEM/WB中保存的控制信号。

2.2.3 时序控制

在5个流水线阶段中，IF、ID和MEM阶段需要注意时序问题。Program Counter和各buffer应该在同一时钟沿更新，保证每个流水阶段执行的指令在同一时刻更新到下一阶段，本次实验中设置为时钟下降沿更新。为了防止读写的Hazard问题，Registers和Data Memory的写操作为时钟上升沿更新。

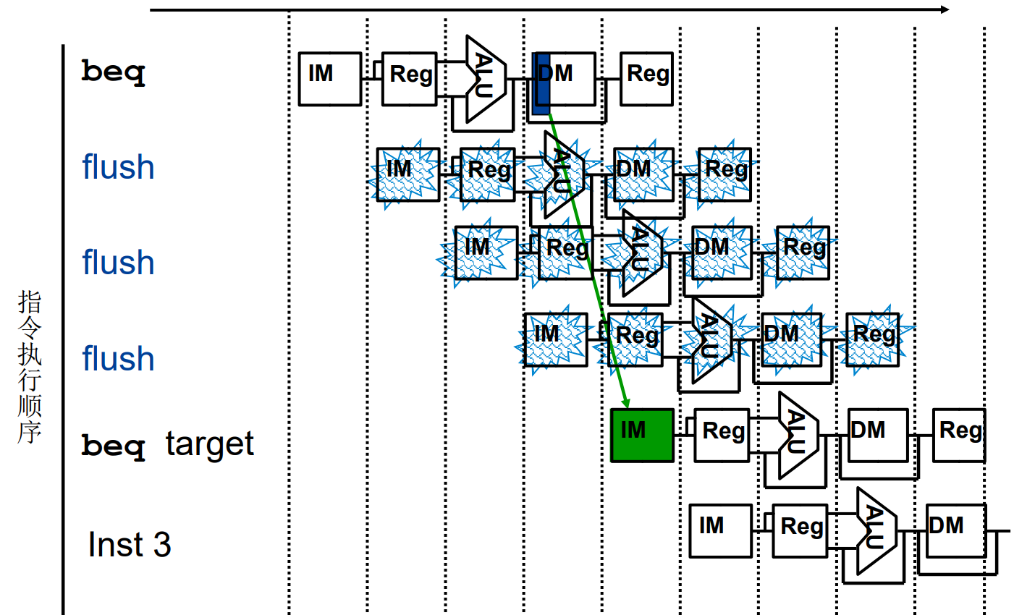
2.3 控制冒险的处理

2.3.1 Jump指令控制冒险



在Jump类指令（jr，j，jal）中，指令在ID阶段译码完成并得到跳转地址，然而此时IF阶段已经得到了顺序执行下一条指令的内容并保存在IF/ID寄存器内。因此，在将jump信号和目标发给IF阶段的同时，还要清空IF/ID寄存器中错误的指令。此时ID/EXE寄存器中还需保留有jal指令的写回操作，无需清空ID/EXE寄存器。

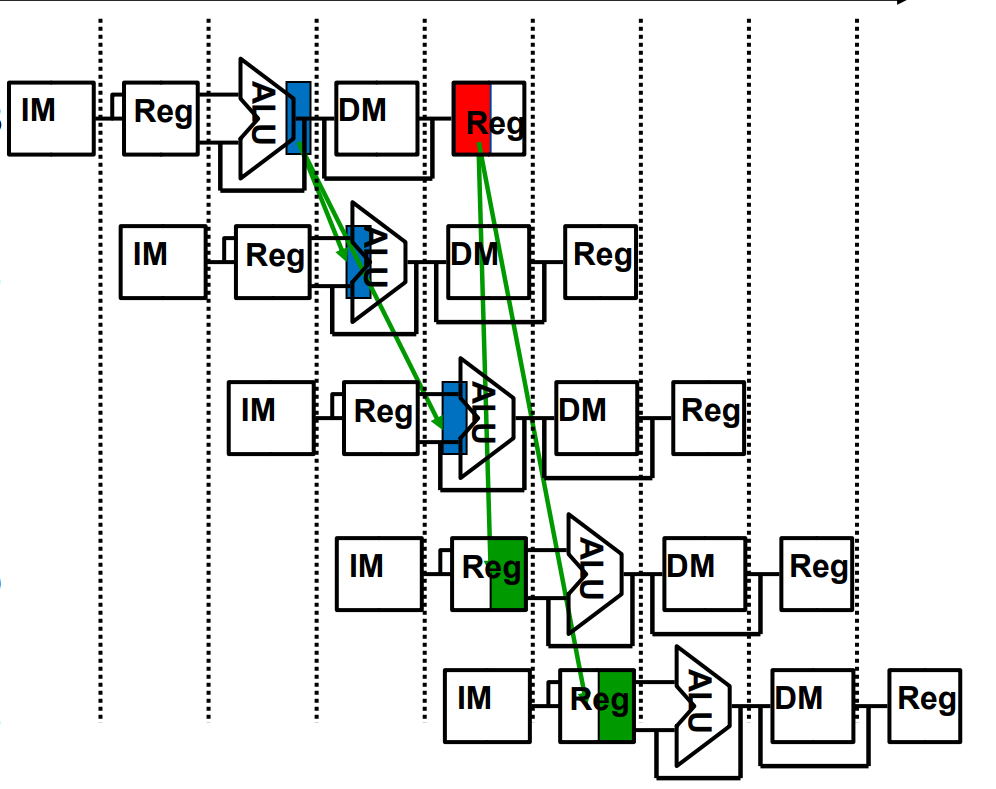
2.3.2 Branch指令控制冒险



在Branch类指令（beq，bne）中，指令在EX阶段计算出zero值并判定是否跳转（实际判定了分支预测是否错误），此时IF、ID阶段可能已经获得了错误的指令，需要清空ID/IF和IF/EX两个寄存器。此时EX/MEM寄存器保存了Branch指令，在MEM和WB阶段没有操作，因此清空与否都可以。

2.4 数据冒险的处理

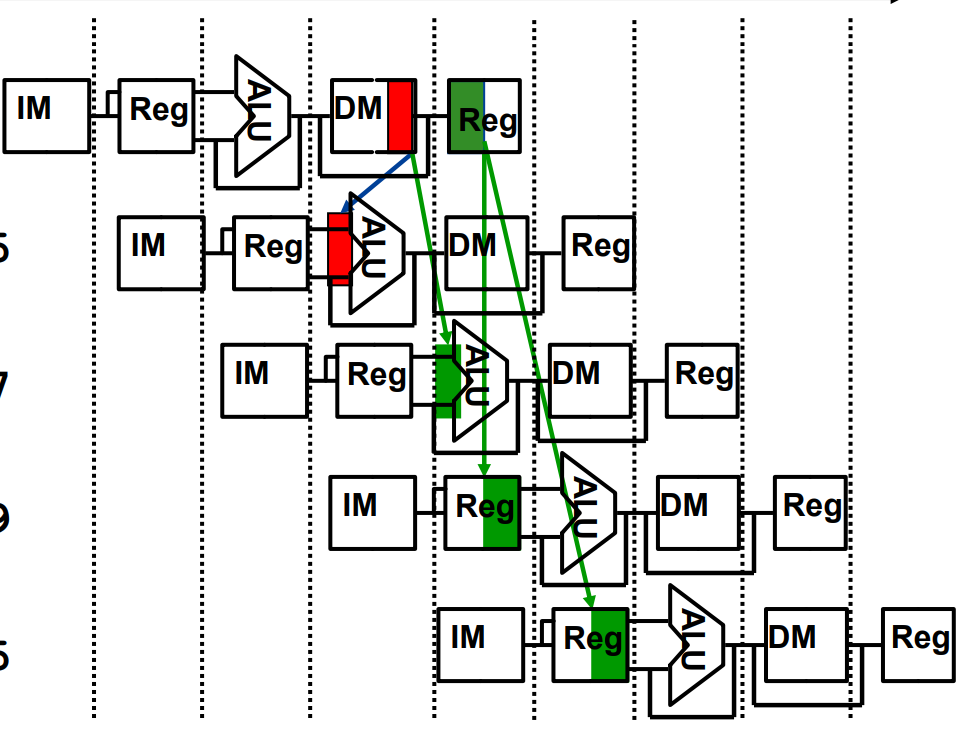
2.4.1 寄存器Load-Use冒险



在寄存器Load-Use冒险中共有三种情况：

1. EX阶段用到前一条指令的结果：在EX模块内部，设置一个Forwarding单元将EX/MEM寄存器中的数据传给ALU的输入。
2. EX阶段用到前两条指令的结果：设置一个Forwarding单元，将此时的MEM/WB寄存器中的数据传给ALU的输入。
3. EX 阶段用到前三条指令的结果：在上一个时钟周期，前三条指令正在执行WB阶段写寄存器，该条指令正在执行ID阶段读寄存器。在Registers模块内部设置一个Forwarding单元将写入数据传给读出数据。

2.4.2 内存Load-Use冒险



在内存Load-Use冒险中，前一条指令的MEM阶段和此条指令的ALU阶段同时发生。如果做Forwarding，那么流水线延迟将变为MEM和EX的延迟之和，会严重降低CPU的频率。因此这里采用Stall机制实现，当出现内存Load-Use冒险时，IF、ID和EX阶段以及相应的寄存器将会暂停一个周期，等待前一条指令进入WB阶段，并通过Registers的Forwarding机制传给EX阶段。

2.5 分支预测的原理

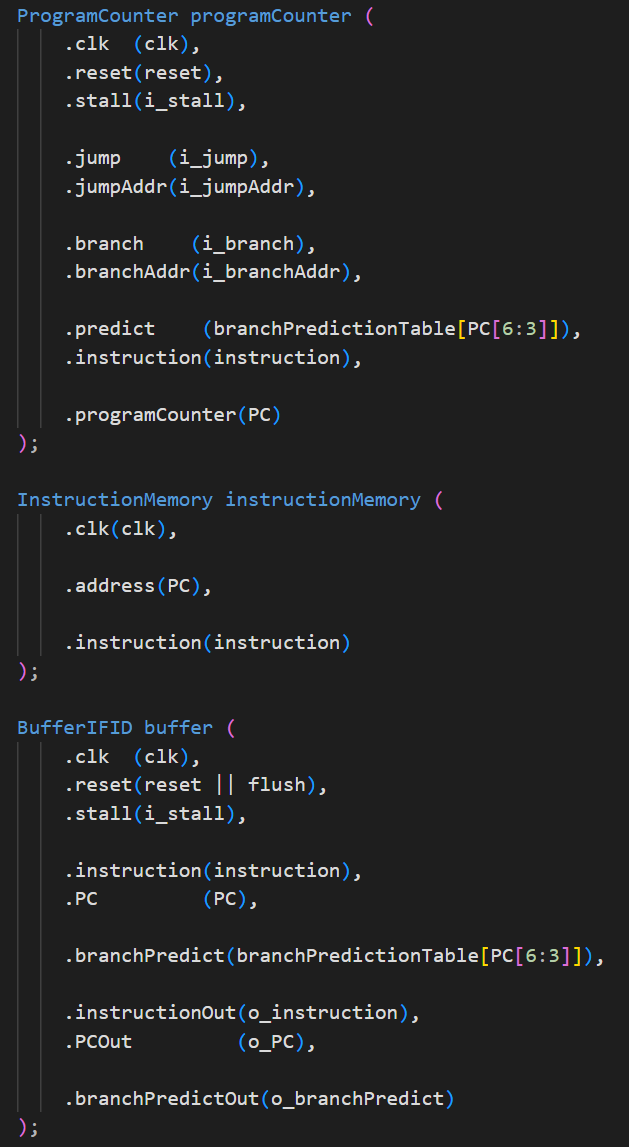
在IF阶段可以保存一个哈希表，记录指令上次跳转情况，并预测Branch指令的行为与上次一致。为了实现提前跳转，IF阶段需要提前译码指令，跳转到相应位置。在EX阶段通过实际计算判断预测是否正确，若错误则flush取到的错误指令，并通知IF阶段修改哈希表中的预测值。

**3.** 功能实现

3.1 流水线处理器的实现

3.1.1 IF阶段

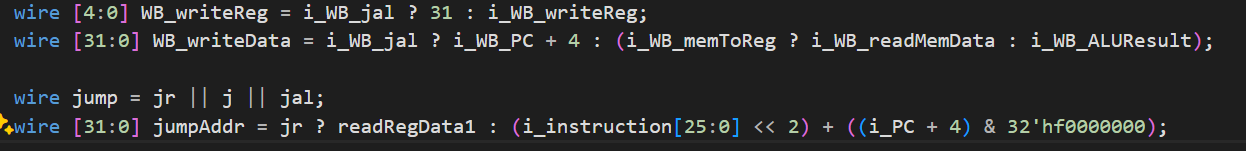
Instruction Fetch模块的部分代码如下：

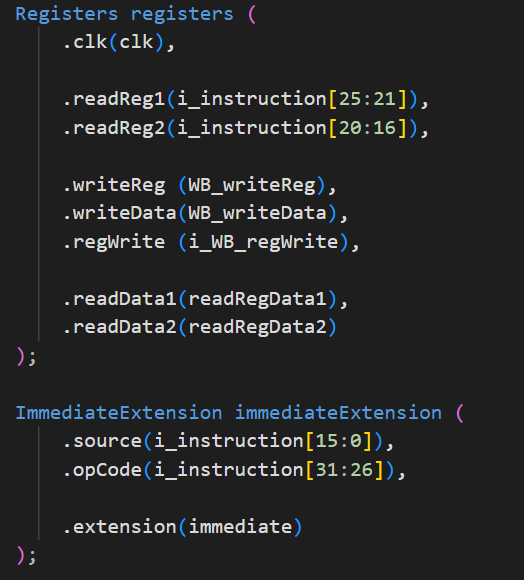
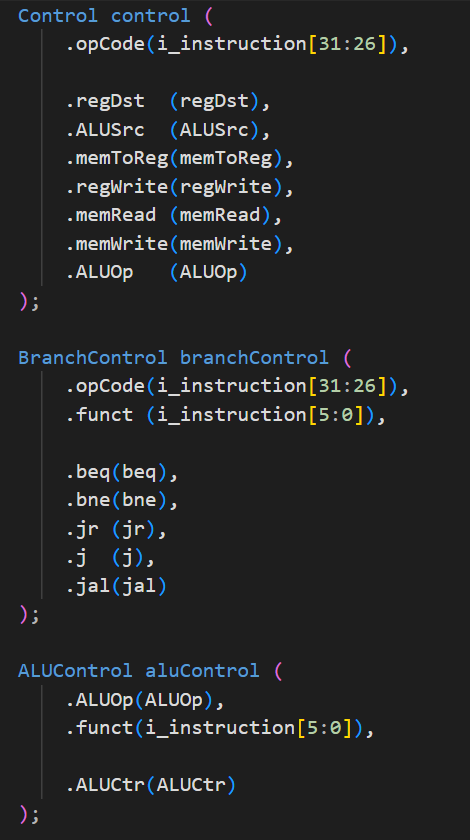


该阶段包括了Program Counter和Instruction Fetch模块，并保存了分支预测的哈希表。输出信号包括PC值、指令和分支预测信号。

3.1.2 ID/WB阶段

Instruction Decode模块部分代码如下：

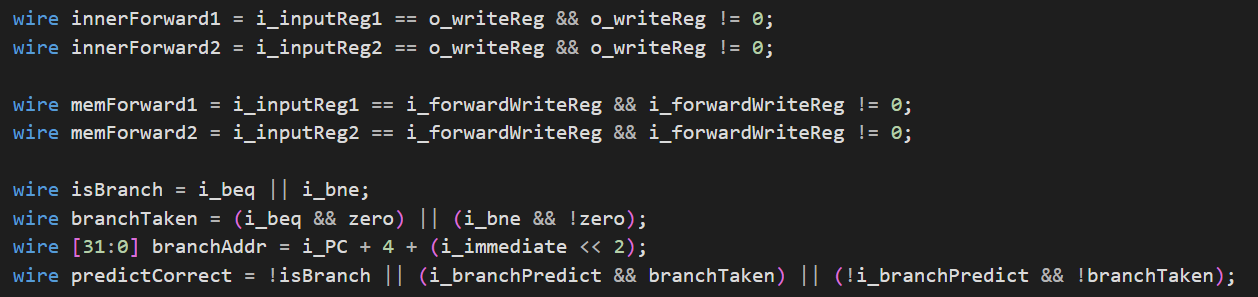


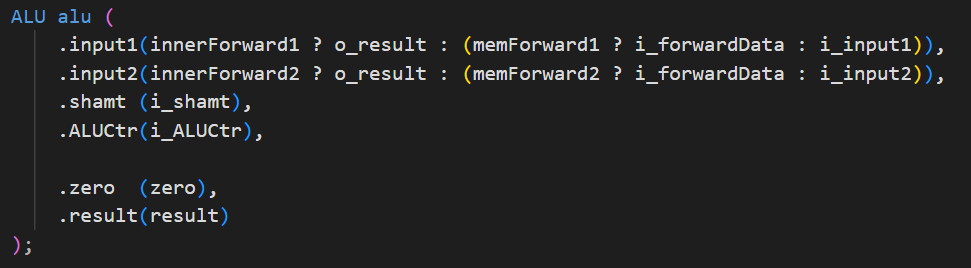
 

该阶段包括了Control、ALU Control、Branch Control、Registers和Immediate Extension模块。此外，Jump类指令的跳转信息由此处发给IF阶段。WB阶段也在该模块实现。

3.1.3 EX阶段

Execution的部分代码如下：

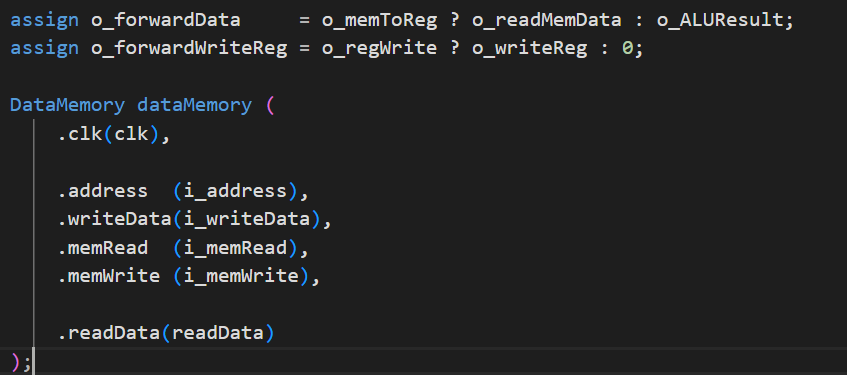




Execution阶段只有ALU模块。在该阶段中，处理了Forwarding的相关机制和Branch的判定。

3.1.4 MEM阶段

Memory模块的部分代码如下：



Memory阶段也只有Data Memory模块。这里处理了由MEM向ALU发送的Forwarding数据，并将WB相关信号发给ID阶段。

3.1.5 流水线寄存器

流水线寄存器将输入的值在时钟下降沿更新到输出寄存器，行为类似D触发器。以IF/ID阶段为例：

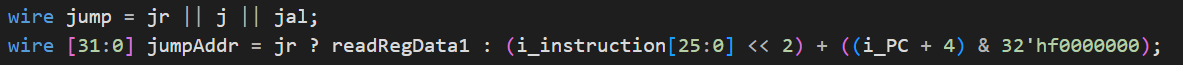
文本

描述已自动生成

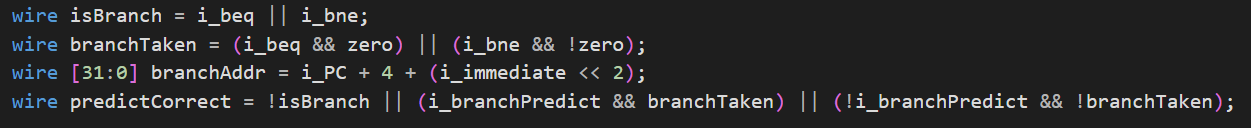
这里接收时钟信号和reset、stall两个控制信号。在reset时清空当前buffer内容，在stall时不做操作，否则在时钟下降沿将输出值更新。

3.2 消除控制冒险的实现

Jump指令在ID阶段的译码如下：



Branch指令在EX阶段的判定如下：



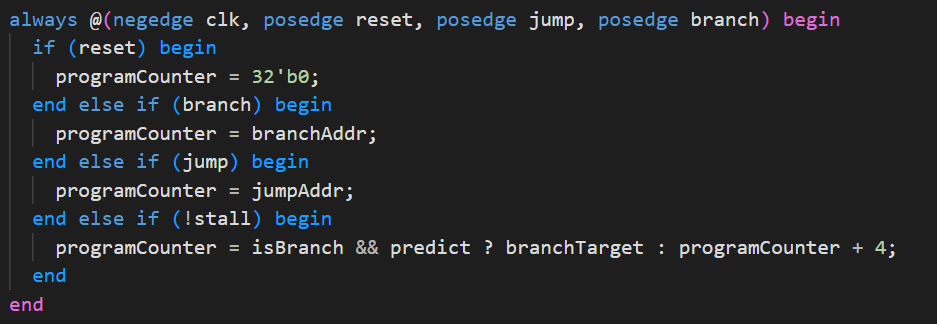
在发生Jump/Branch时，清空相应流水线寄存器：

文本

描述已自动生成文本

描述已自动生成

在Program Counter内根据输入信号跳转：

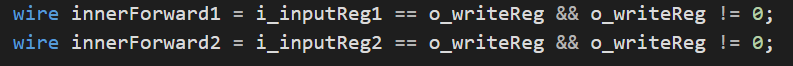


3.3 消除数据冒险的实现

3.3.1 消除寄存器Load-Use冒险

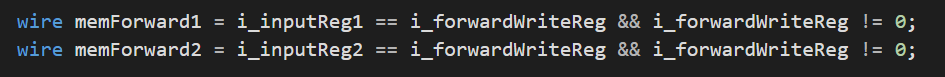
根据三种寄存器Load-Use数据冒险情况，需要实现机制如下：

（1）在EX模块内部，设置一个Forwarding单元将EX/MEM寄存器中的数据传给ALU的输入：

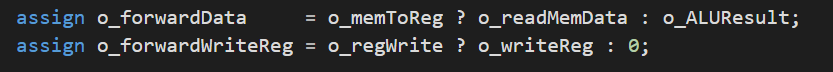


（2）设置一个Forwarding单元将MEM/WB寄存器中的数据传给ALU的输入：

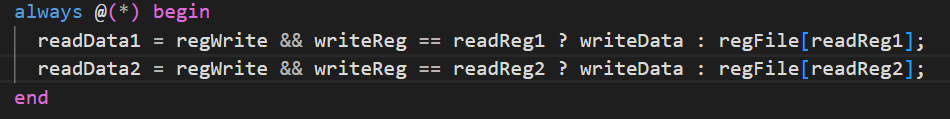
在ALU内：



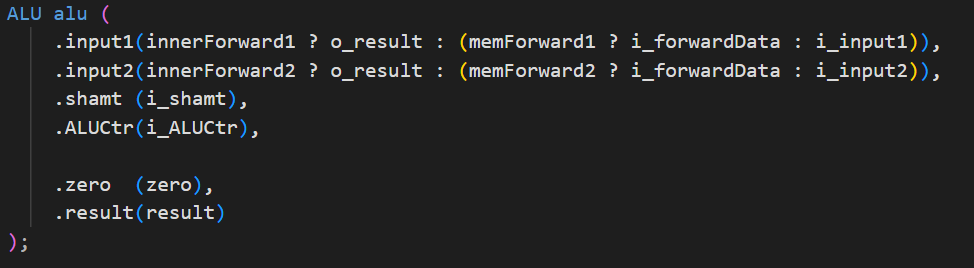
在MEM阶段：



（3）在Registers模块内部设置一个Forwarding单元将写入数据传给读出数据：



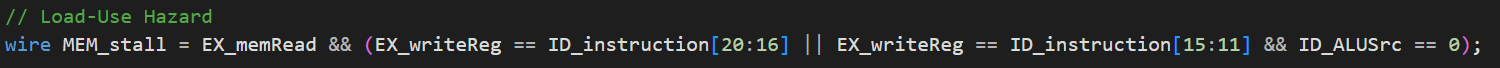
最后在ALU内选择输入：



此外在传递用作写入内存的寄存器的读取值时，也做了相同原理的Forwarding机制，此处省略。

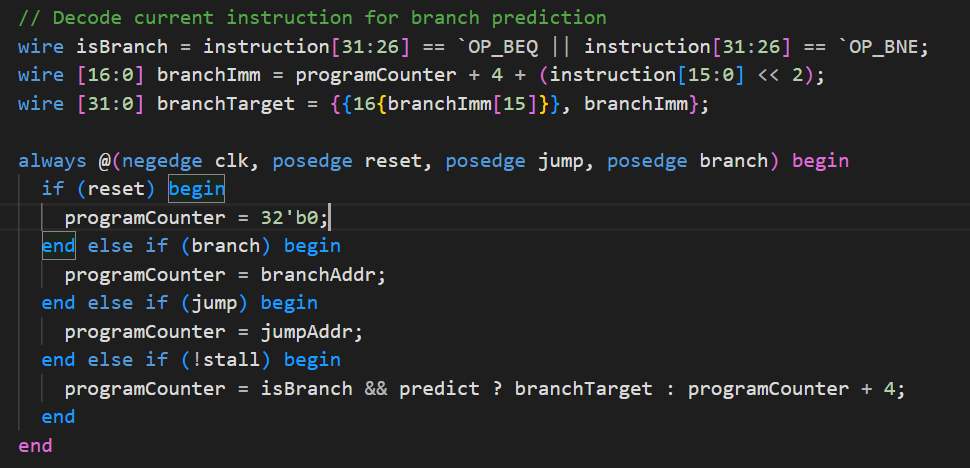
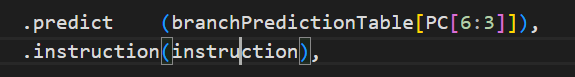
3.3.2 消除内存Load-Use冒险

在内存的Load-Use冒险中，实际上没有数据被Forward，只是IF和ID阶段发生了stall，stall的检测如下：

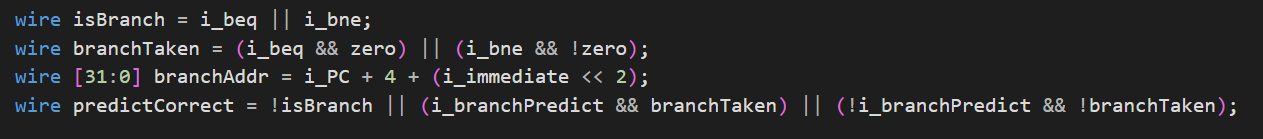


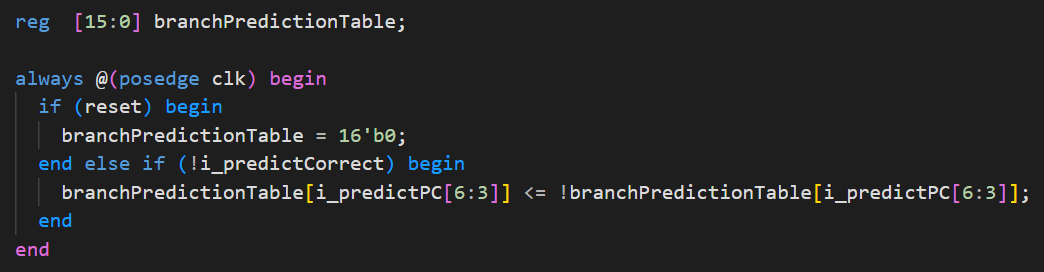
3.4 分支预测的实现

在Instruction Fetch阶段，需要提前译码当前指令，根据Branch Table预测本次跳转是否发生，如果满足条件直接跳转，并且发出predict信号为真给EX阶段。这里哈希值取了指令的6-3位，哈希表大小为16。



在EX阶段，计算出实际跳转是否发生，并输出predict correct信号。如果信号为false，那么IF/ID和ID/EX寄存器将被清空。同时IF阶段会相应修改Branch Table的对应位。

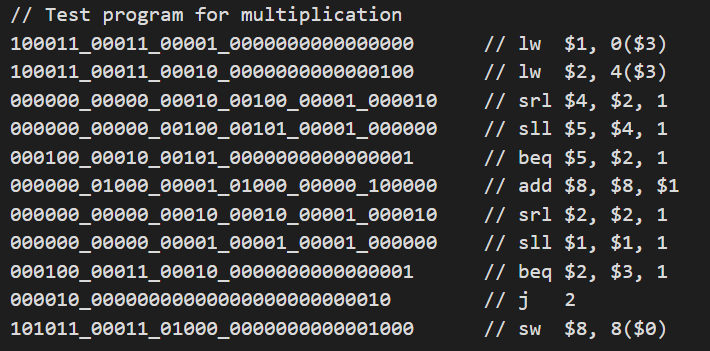




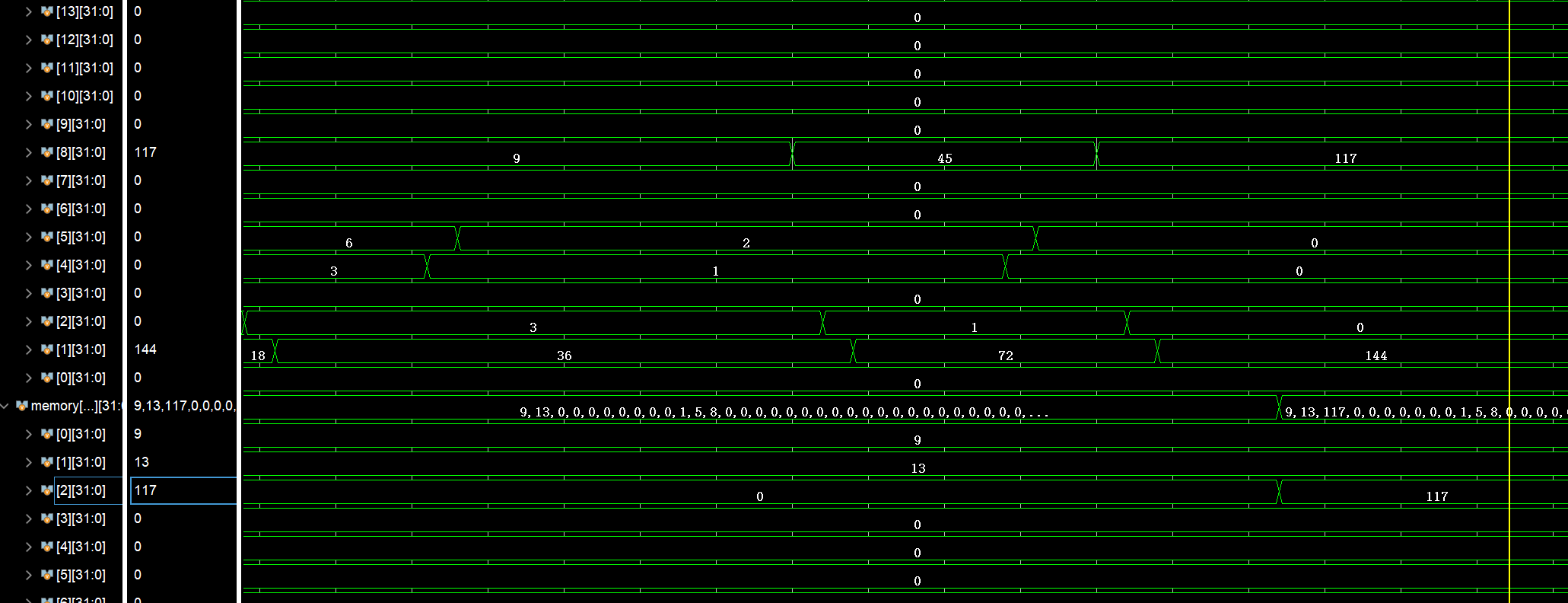
**4.** 结果验证

4.1 提供的测试代码

在乘法器测试中，编写Instruction.txt文件如下：



测试结果如图所示：



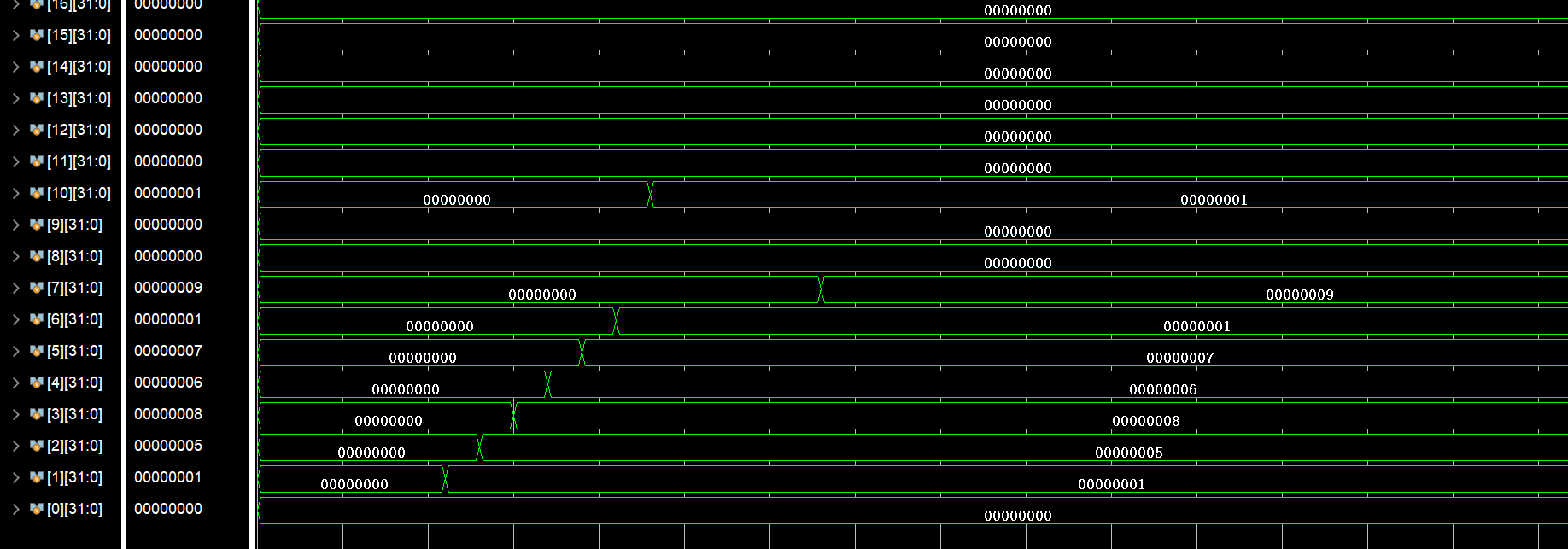
输出和计算过程符合预期。

在提供的测试代码中，编写Instruction.txt文件如下：

日历

中度可信度描述已自动生成

测试结果如图所示：



没有Data Hazard产生。

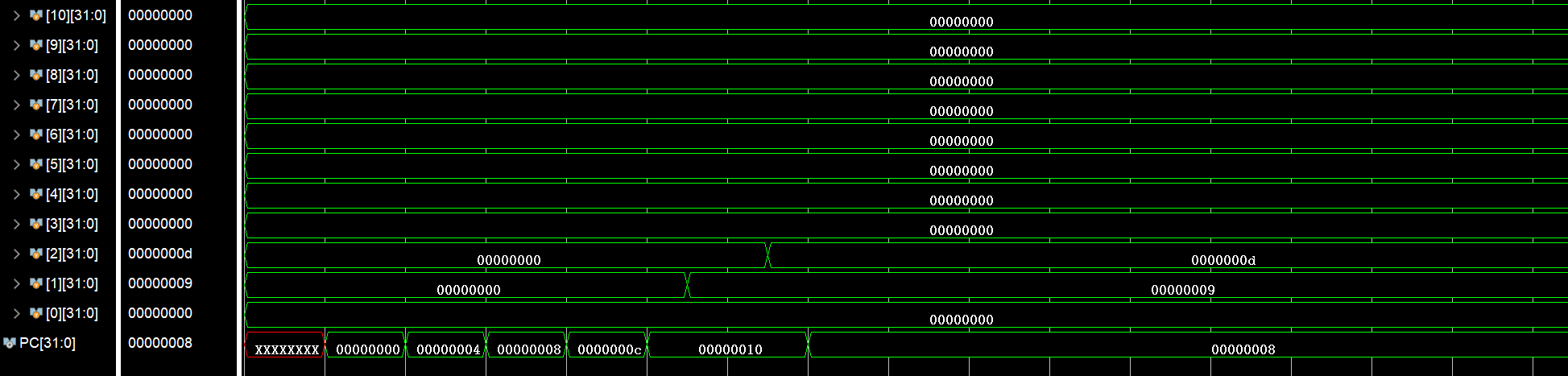
4.2 冒险与分支预测的测试

编写Instruction如下：

图形用户界面, 文本

描述已自动生成

仿真结果如图所示



在测试中，出现了内存的Load-Use冒险，最终bne成功跳转，说明数据冒险已经消除。在Branch中，前面阶段的PC已经取到0x0c和0x10，但因为相关流水线寄存器被清空，addi指令没有执行成功，说明控制冒险已经消除。最后，由于bne指令一直跳转到自身，在第二次分支时分支预测判断会发生分支，因此PC一直保持0x08，分支预测行为符合预期。

4.3 其他指令的测试

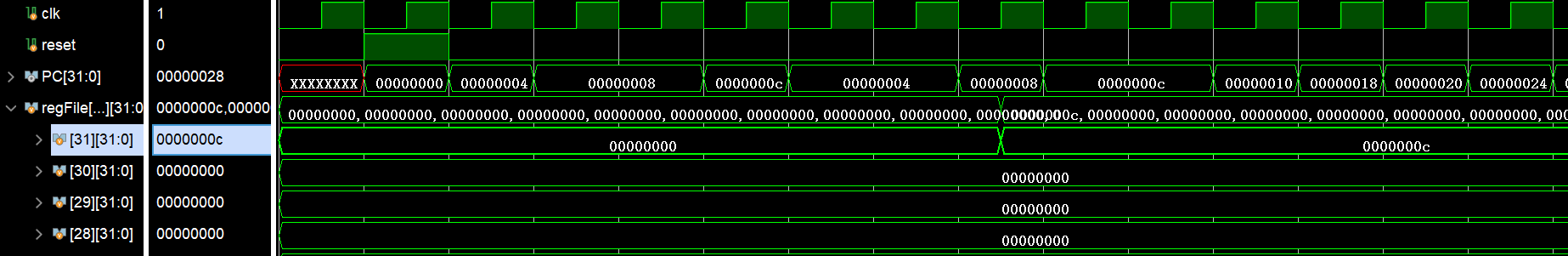
4.3.1 jr，j和jal的测试

编写Instruction如下：

图形用户界面, 文本, 应用程序, 聊天或短信

描述已自动生成

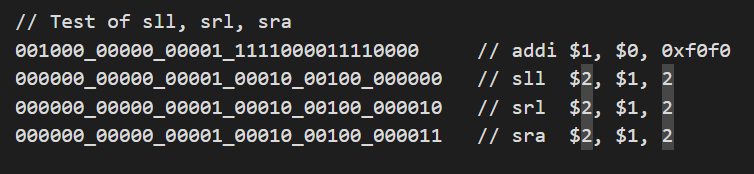
结果如图所示：



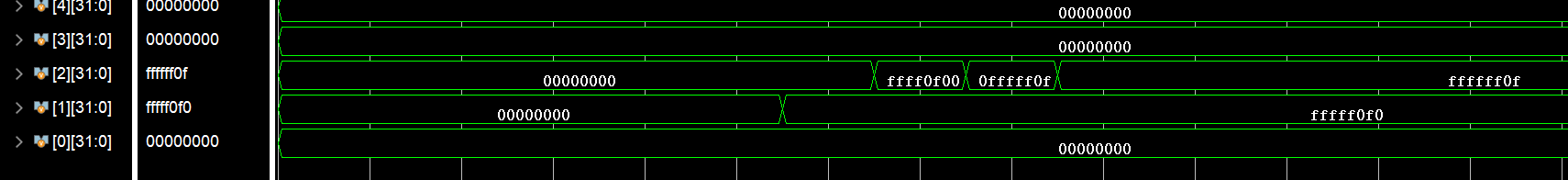
测试代码先使用j 2无条件跳转到PC = 2 << 2 = 8处，此时执行第三条指令jal 1，将PC + 4 = 0x0c存入ra寄存器，并跳转到PC = 1 << 2 = 4处。接下来执行第二条指令jr $31将PC设为ra的值0x0c。可以看到相应的跳转位置因为控制冒险导致了流水线停顿。

4.3.2 sll，srl和sra的测试

编写Instruction如下：



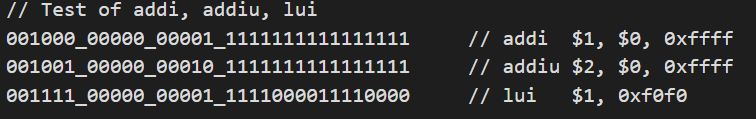
结果如图所示：



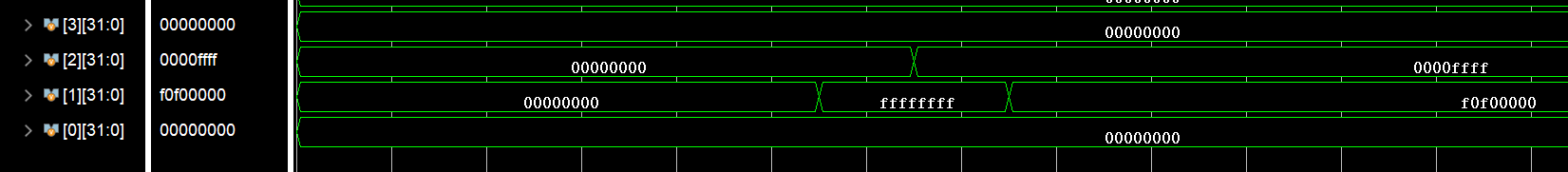
测试代码将1号寄存器设为0xfffff0f0（有符号数扩展）。在sll指令中，1号寄存器左移4位得到0xffff0f00。在srl指令中，1号寄存器右移四位得到0x0fffff0f。在sra指令中，1号寄存器右移4位，并在高位填充符号位1，得到0xffffff0f。仿真符合预期，也没有数据冒险错误。

4.3.3 addi，addiu和lui的测试

编写Instruction如下：



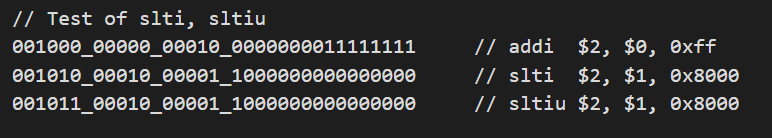
结果如图所示：



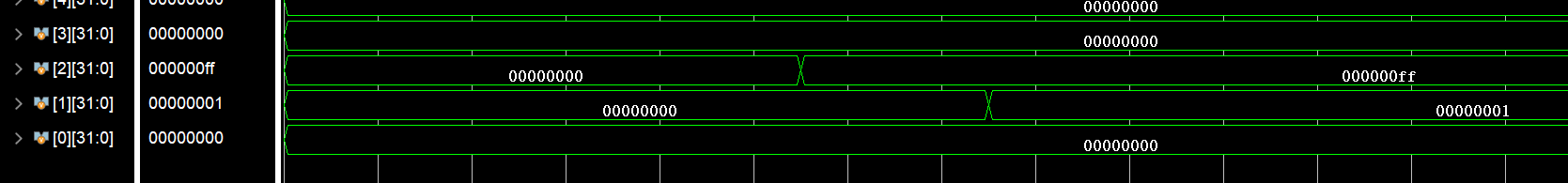
仿真代码中addi指令将0xffff做有符号数扩展得到0xffffffff，addiu指令将0xffff做无符号数扩展得到0x0000ffff。lui指令将立即数0xf0f0填入高16位，并把低16位清零得到0xf0f00000。仿真符合预期，也没有数据冒险错误。

4.3.4 slti和sltiu的测试

编写Instruction如下：



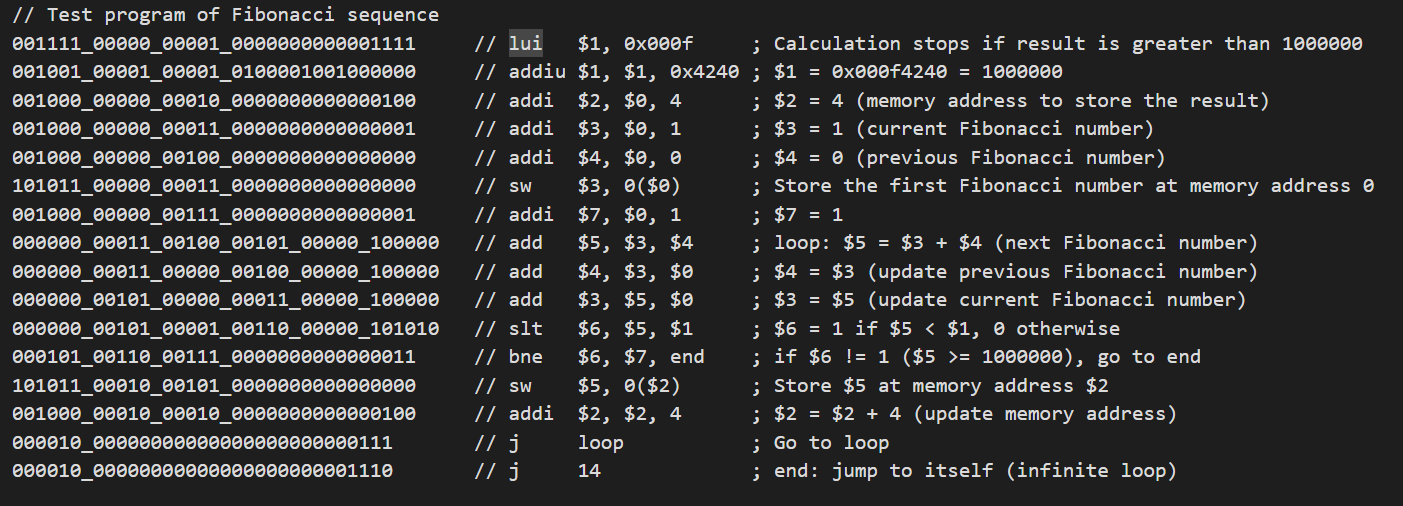
结果如图所示：



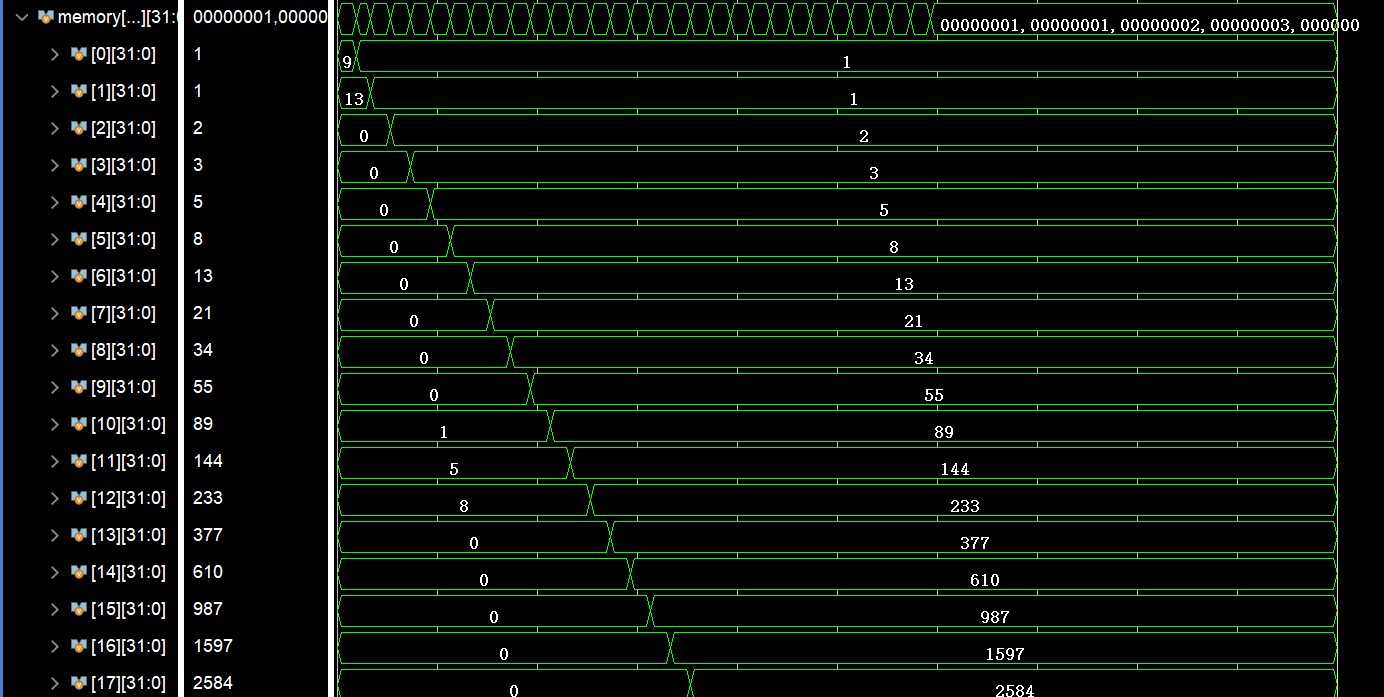
测试代码把2号寄存器设为0x00ff。在slti指令中，0x8000做有符号数扩展，为负数，0x00ff < 0xffff8000 不成立，故1号寄存器不设为1。在sltiu指令中，0x8000做无符号数扩展 0x00ff < 0xffff成立，故1号寄存器设为1。仿真符合预期，也没有数据冒险错误。

4.5 综合测试

编写斐波那契数列计算代码如下：



测试结果如图所示：



电脑游戏的屏幕

描述已自动生成

测试结果符合预期。

**5.** 总结与反思

在Lab06中，我深入了解了MIPS多周期流水线处理器的各部件功能和关系。通过完成消除控制冒险、消除数据冒险、进行分支预测等功能，我熟悉了处理器IF、ID、EX、MEM和WB阶段的具体流程，掌握了冒险的成因和解决方法，也了解了如何写出减少流水线停顿的代码。通过使用Vivado开发环境，我能够更好地编写仿真文件调试Verilog HDL的代码和模块。

我要感谢课程组为我们提供的详细指导书，它为我提供了清晰的实验步骤，使我能够更好地理解和实践所学的知识。通过这次实验，我不仅巩固了Verilog和MIPS处理器的基础知识，还为的学习和设计打下了坚实的基础。