**重庆大学课程设计报告**

课程设计题目： 基于MIPS指令架构的SOC CPU设计

学 院： 计算机学院

专业班级： 计算机科学与技术 5班

年 级： 20

姓 名： 石岳

学 号： 20201717

完成时间： 2023 年 1 月 9 日

成 绩：

指导教师： 冯永

重庆大学教务处制

**MIPS SOC CPU设计报告**

By 石岳

一、设计简介

在本次硬件综合设计课程中，实现了基于MIPS架构的流水线CPU，包含52条基本指令、5条特权指令及异常处理机制[1,2]，此外,在此基础上实现了快速除法器、AXI接口[3,4]和指令Cache的添加，提升了CPU的性能。

（一）设计任务的问题分析

该项目的主要任务是在计算机组成原理实验四的基础上通过添加指令和异常处理来完善CPU的功能，通过进行soc封装，使实现的CPU能够嵌入到硬件中(文献[4])。我们整体设计流程是从添加52条指令开始，依次实现SRAM-SOC、5条特权指令、异常处理、AXI接口及Cache，并在每个阶段后进行相应的功能测试（52条、57条、AXI、Cache）和性能测试（AXI、Cache）。项目的主要难点在于AXI接口的实现和Cache的实现及功能测试和性能测试的debug。

二、设计方案

（一）总体设计思路

项目实现过程，首先实现52条指令的添加，依次完成逻辑运算指令、移位运算指令、数据移动指令与HILO寄存器、算术指令、转移指令、访存指令。在完成第一步之后，添加到57条指令，实现CP0寄存器、延迟槽、精确异常等[5]。CP0寄该存器中包括BadVAddr、Count、Status、Cause、EPC寄存器。利用上述工具，完成异常、中断的处理。然后，实现了SRAM端口到类SRAM端口的转换。最后，完成了AXI和指令Cache的实现。

总体的结构如图1：在mycpu\_top模块中调用mmu模块进行虚拟地址到物理地址的转换，调用Cache模块，访问Cache，调用给定的转换桥及CPU和AXI的接口模块实现AXI接口，调用MIPS，将所需要的信号，传入给定的顶层模块中；在MIPS模块中，调用controller模块和datapath模块，调用inst(data)sram\_like模块实现类sram接口。

该项目主要分为几个重要的模块：controller、 maindecorder 、aludecorder、datapath 、alu、div 、hazard 、 exception 、cp0reg、 hilo 、mmu、inst(data)sram\_like等，由于本项目是在计算机组成原理实验四的基础上实现的，故不在对实验四中的代码再进行重复描述，下文仅对为了实现本项目而新添加的代码进行详细描述。使用“\*”标记出了提供的模块代码。

MIPS CPU

类SRAM

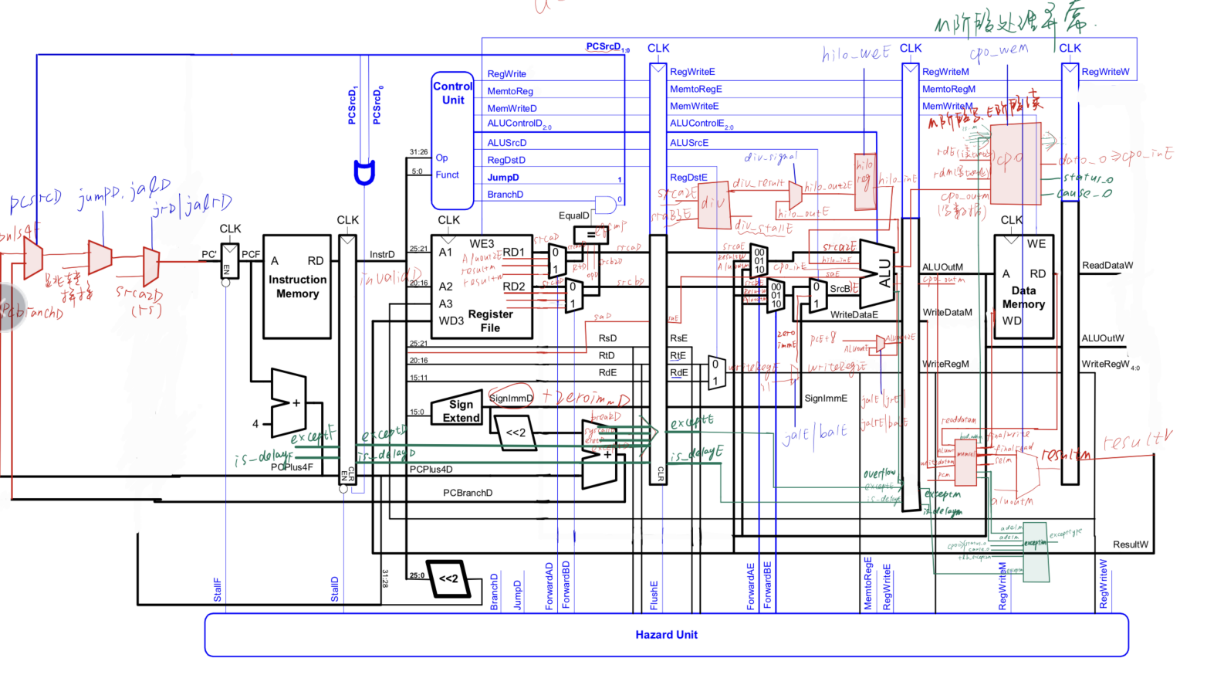
Cache

CPU🡪AXI转换桥

SRAM端口

图1

**整体数据通路图：**

图2

（二）指令设计

本CPU设计中，一共实现了52+5=57条指令，实现顺序与具体指令如下表所示。

|  |  |  |
| --- | --- | --- |
| 指令类型 | 数量 | 具体指令 |
| 逻辑运算指令 | 8 | AND, OR, XOR, NOR, ANDI, XORI, LUI, ORI |
| 移位运算指令 | 6 | SLL, SRL, SRA, SLLV, SRLV, SRAV |
| 数据移动指令 | 4 | MFHI, MFLO, MTHI, MTLO |
| 算术运算指令 | 14 | ADD, ADDU, ADDI,ADDIU, SUB, SUBU, ST, SLT, SLTI, SLTIU, DIV, DIVU, MULT, MULTU |
| 转移指令 | 12 | BEQ, BNE, BGEZ, BGTZ, BLEZ, BLTZ, BLTZAL, BGEZAL, J, JAL, JR, JALR, |
| 访存指令 | 8 | LB, BU, LH, LHU, LW, SB, SH, SW |
| 特权指令 | 5 | BREAK, SYSCALL, ERET, MTC0, MFC0 |

前五十二条指令的设计基本上思路是一致的，也不需要对计组实验四的数据通路做什么太大的调整，都是通过在controller中的maindec模块内根据指令的op和funct进行判断，判断出是什么指令后，给数据通路中的各种控制信号赋值并赋值alucontrol信号，而后，在alu中根据alucontrol信号判断执行什么操作。

几点变动在于：

1. 在实现逻辑运算指令时，由于需要进行原先没有的零扩展操作，于是在datapath中新增了零扩展模块zero\_extend，并将选择alu操作数的控制信号alusrc扩充为2位，实现三选一操作。
2. 在实现移位指令时，由于要指定的移位量原先并没有传入alu模块中，于是，在alu模块的输入信号增加了移位sa信号。
3. 在实现数据移动指令时，需要对HILO寄存器进行读写操作，于是在alu输入输出信号分别新加了读和写HILO寄存器的数据。
4. 在实现乘除法指令时，由于乘法的结果最多为64位、除法需要64位分别存储32位商和余数，于是，将其结果存入新加的写HILO寄存器输出信号中，存入HILO。
5. 在实现转移指令时，如果是需要跳回的指令，需要先将返回地址存入寄存器中。如果需要从寄存器中获取跳转地址，则应先去从寄存器中读出跳转地址后再进行跳转。
6. 在实现访存指令时，新增了memory\_select模块选择读出/写入内存的数据。
7. 在实现特权指令中的MTC0和MFC0操作时，在alu模块内增加了分别新加了读和写cp0寄存器的数据。

（三）datapath模块设计

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| clk | Input | 1 | 时钟信号 |
| rst | Input | 1 | 复位信号 |
| ext\_int | Input | 6 | 外部硬件中断信号 |
| pcF | Output | 32 | F阶段指令地址 |
| instrF | Input | 32 | F阶段指令 |
| pcsrcD | input | 1 | D阶段产生的判断是否跳转，是产生新的pc值的控制信号 |
| branchD | Input | 1 | D阶段是否是分支指令 |
| jumpD,jrD,jalD,jalrD,balD | Input | 1 | D阶段是否是各种跳转指令 |
| equalD | Output | 1 | D阶段中提前判断rs和rt中的值是否相等 |
| instrD | Output | 32 | D阶段指令 |
| stallD | Output | 1 | D阶段暂停信号 |
| invalidD | Input | 1 | D阶段发生指令异常信号 |
| flushD | Output | 1 | D阶段清空信号 |
| memtoregE | Input | 1 | E阶段的指令是否将内存中读出的数据写回寄存器信号 |
| alusrcE | Input | 2 | E阶段选择alu的第二个操作数b是来自寄存器、有符号扩展还是零扩展的信号 |
| regdstE | Input | 1 | E阶段选择写回寄存器的地址的信号 |
| regwriteE | Input | 1 | E阶段的指令是否写寄存器的信号 |
| alucontrolE | Input | 5 | E阶段alu控制信号 |
| flushE | Output | 1 | E阶段清空信号 |
| hilo\_weE | Input | 1 | E阶段写HILO寄存器使能信号 |
| div\_validE | Input | 1 | E阶段指令是否是除法操作信号 |
| div\_signE | Input | 1 | E阶段除法指令是否是有符号除法操作信号 |
| stallE | Output | 1 | E阶段暂停信号 |
| memtoregM | Input | 1 | M阶段的指令是否将内存中读出的数据写回寄存器信号 |
| regwriteM | Input | 1 | M阶段的指令是否写寄存器的信号 |
| aluoutM | Output | 32 | M阶段alu产生的结果 |
| final\_writedata | Output | 32 | M阶段最终写入内存的数据 |
| readdataM | Input | 32 | M阶段从内存中读出的数据 |
| selM | Output | 4 | M阶段判断写内存操作写入内存的字节数 |
| cp0weM | Input | 1 | M阶段写CP0寄存器使能信号 |
| flush\_except | Output | 1 | M阶段产生异常/中断时的清空信号 |
| flushM | Output | 1 | M阶段清空信号 |
| stallM | Output | 1 | M阶段暂停信号 |
| memtoregW | Input | 1 | W阶段的指令是否将内存中读出的数据写回寄存器信号 |
| regwriteW | Input | 1 | W阶段的指令是否写寄存器的信号 |
| pcW | Output | 32 | W阶段指令地址 |
| writeregW | Output | 5 | W阶段写入寄存器堆的寄存器号 |
| resultW | Output | 32 | W阶段指令最终产生的结果 |
| flushW | Output | 1 | W阶段清空信号 |
| stallW | Output | 1 | W阶段暂停信号 |
| longest\_stall | Output | 1 | CPU中的最长暂停信号，避免类SRAM在CPU暂停时重复请求 |
| i\_stall | Input | 1 | CPU通过类SRAM访问指令内存时的暂停信号 |
| d\_stall | Input | 1 | CPU通过类SRAM访问数据内存时的暂停信号 |

**模块分析：**

datapath模块代表着我们设计的整个流水线CPU的数据通路，主要分为取指（F）、译指（D）、执行（E）、访存（M）、写回（W）五个阶段，使程序中的变量可以在不同的阶段中顺序传递，其它模块（controller）通过接口与datapath模块交互。此外，在该模块中，调用了协处理器CP0来支持与异常相关的指令，冒险模块hazard来解决数据冲突、异常处理时pc的跳转，HILO寄存器存储产生的数据，memory\_select模块生成访问内存的相关信号等等。

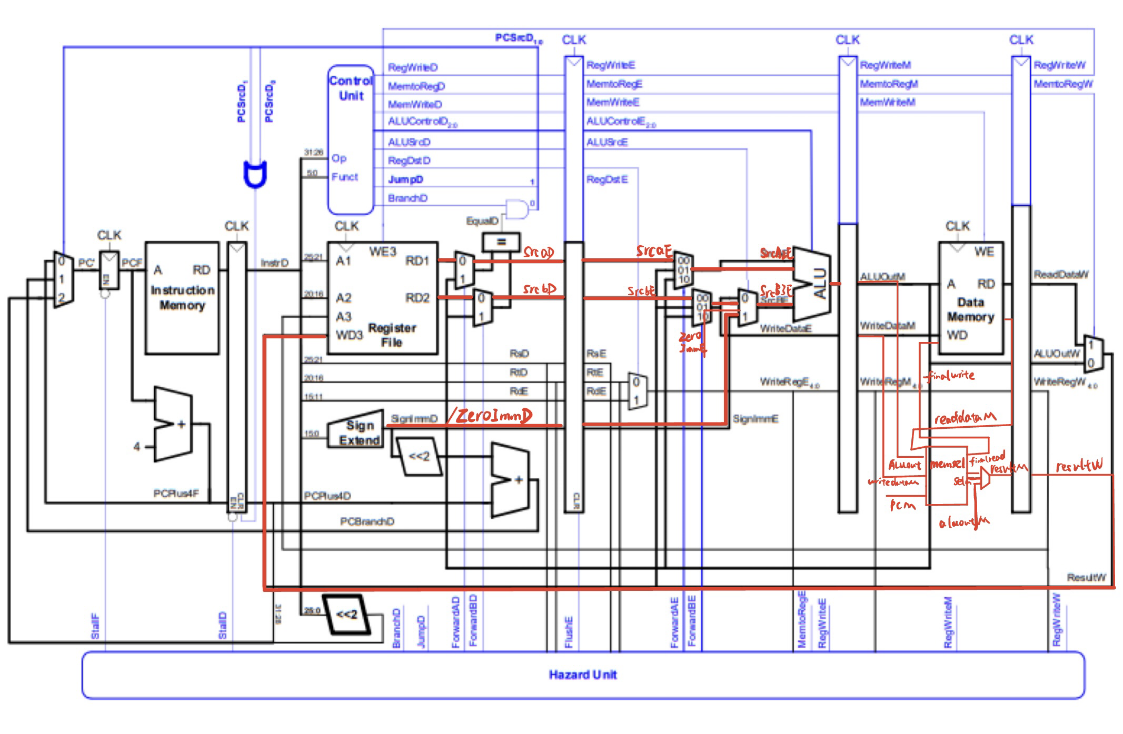
**数据通路图：**

图3

（四）hazard模块设计

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| stallF | Output | 1 | F阶段暂停信号 |
| flushF | Output | 1 | F阶段清空信号 |
| rsD | Input | 5 | D阶段输入的rs寄存器地址 |
| rtD | Input | 5 | D阶段输入的rt寄存器地址 |
| forwardaD | Output | 2 | D阶段alu第一个操作数数据前推信号 |
| forwardbD | Output | 2 | D阶段alu第二个操作数数据前推信号 |
| stallD | Output | 1 | D阶段暂停信号 |
| flushD | Output | 1 | D阶段清空信号 |
| rsE | Input | 5 | E阶段输入的rs寄存器地址 |
| rtE | Input | 5 | E阶段输入的rt寄存器地址 |
| writeregE | Input | 5 | E阶段写入寄存器堆的寄存器号 |
| regwriteE | Input | 1 | E阶段的指令是否写寄存器的信号 |
| memtoregE | Input | 1 | E阶段的指令是否将内存中读出的数据写回寄存器信号 |
| forwardaE | Output | 2 | E阶段alu第一个操作数数据前推信号 |
| forwardbE | Output | 2 | E阶段alu第二个操作数数据前推信号 |
| flushE | Output | 1 | E阶段清空信号 |
| div\_stallE | Output | 1 | E阶段除法指令暂停信号 |
| stallE | Output | 1 | E阶段暂停信号 |
| rdE | Input | 5 | E阶段目标寄存器号 |
| forwardcp0E | Output | 1 | E阶段读CP0寄存器的数据前推信号 |
| writeregM | Input | 5 | M阶段写入寄存器堆的寄存器号 |
| regwriteM | Input | 1 | M阶段的指令是否写寄存器的信号 |
| flushM | Output | 1 | M阶段清空信号 |
| stallM | Output | 1 | M阶段暂停信号 |
| writeregW | Input | 5 | W阶段写入寄存器堆的寄存器号 |
| regwriteW | Input | 1 | W阶段的指令是否写寄存器的信号 |
| flushW | Output | 1 | W阶段清空信号 |
| stallW | Output | 1 | W阶段暂停信号 |
| longest\_stall | Output | 1 | CPU中的最长暂停信号，避免类SRAM在CPU暂停时重复请求 |
| i\_stall | Input | 1 | CPU通过类SRAM访问指令内存时的暂停信号 |
| d\_stall | Input | 1 | CPU通过类SRAM访问数据内存时的暂停信号 |

**模块分析：**

由于数据指令间相互联系，因此可能产生冒险， hazard模块的主要功能就是通过数据前推和暂停来解决出现的冒险。此外，还用于产生flush信号，清空不应该被执行的指令。我们还将异常处理时，pc值的跳转也放入了此模块中。

数据前推：

****

对CP0寄存器需要进行读和写操作，而我们采取的设计方式是在E阶段读，在M阶段写[1]。因此，当前一条指令写CP0寄存器而后一条指令需要读出前一条写入的值时，便会产生数据冒险。于是，在本模块中添加了数据前推信号，将前一条指令写入的值前推给后一条需要读出的值。

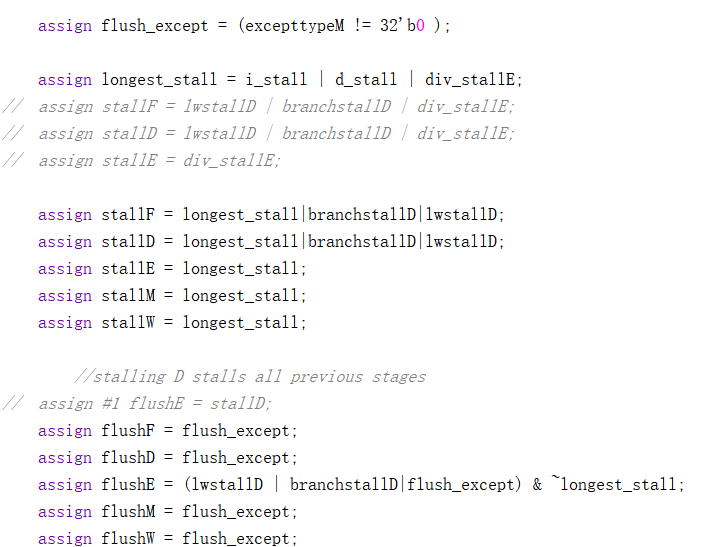
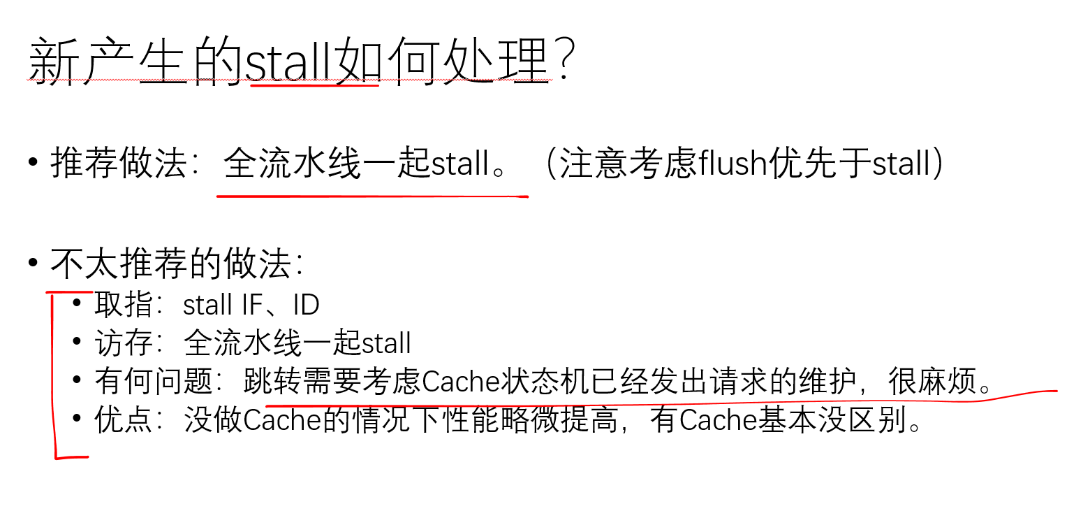


图 4

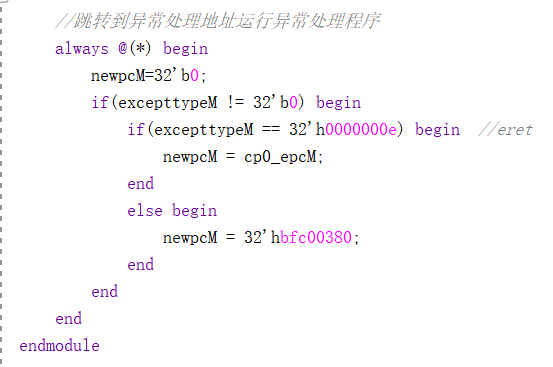
stall信号：

如图所示，hazard模块的主要变动在于各阶段stall信号和flush信号赋值条件的改变。首先，在添加了类SRAM接口后，由于存储器的处理速度滞后于CPU的一个时钟周期，所以在访存未结束时需要暂停CPU流水线，在这里，我们选择陈泱宇学长推荐的取指和数据访存都暂停流水线所有阶段的操作如图4，而不是取指时只暂停前两个阶段，因为这样做可能会产生更多的bug且二者的CPU吞吐量无太大区别(文献[3])。而longest\_stall通过我们的CPU中可能产生多周期暂停的两个访存暂停及除法暂停进行或计算得到，表示CPU中的最长暂停信号，避免在类SRAM在CPU暂停时重复请求，造成死循环。所以，我们把longest\_stall赋值给各阶段的暂停信号，在进行类SRAM访存时暂停所有阶段。此外，前两个阶段保留了实验四中的赋值条件。

flush信号：

由于我们设计的CPU实现的是精确异常,即指令产生的所有异常都在该指令的M阶段进行处理[1,4]，所以是按照指令的顺序而不是异常发生的顺序处理异常。因此，在发生异常时，excepttypeM变量不为0，flush\_except变量被赋值为1。首先，由于M阶段的指令存在异常，所以将flush\_except的值赋值给flushW，防止M阶段的指令的数据继续传递到下个阶段。此外，该异常指令后的指令也应该被清除掉，于是，将flush\_except的值赋值给前面几个阶段的flush变量。需要注意的是，flushE的信号不仅仅取决于异常的发生，实验四中的lw和branch指令暂停时，同样需要flush掉E阶段，防止继续向后传递。而~longest\_stall是在后续添加AXI接口后，debug出现错误（详见三、错误记录 错误12），通过观察仿真图得出lw指令没有被stall就被flush掉了，使其丢失，因此，要在longest\_stall没结束时，不能将其flush掉。

异常处理地址跳转：



异常处理地址跳转有两种情况，一种是ERET指令返回之前cp0\_epcM寄存器中保存的发生异常的地址，另外一种是其它异常/中断跳转到32’hbfc00380运行异常处理程序[3]。

（五）controller模块设计

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| clk | Input | 1 | 时钟信号 |
| rst | Input | 1 | 复位信号 |
| instrD | Input | 32 | D阶段指令 |
| pcsrcD | Output | 1 | D阶段产生的判断是否跳转，是产生新的pc值的控制信号 |
| branchD | Output | 1 | D阶段是否是分支指令 |
| equalD | Input | 1 | D阶段中提前判断rs和rt中的值是否相等 |
| jumpD,jrD,jalD,jalrD,balD | Output | 1 | D阶段是否是各种跳转指令 |
| stallD | Input | 1 | D阶段暂停信号 |
| invalidD | Output | 1 | D阶段发生指令异常信号 |
| flushE | Input | 1 | E阶段清空信号 |
| memtoregE | Output | 1 | E阶段的指令是否将内存中读出的数据写回寄存器信号 |
| alusrcE | Output | 2 | E阶段选择alu的第二个操作数b是来自寄存器、有符号扩展还是零扩展的信号 |
| regdstE | Output | 1 | E阶段选择回写寄存器堆地址的信号 |
| regwriteE | Output | 1 | E阶段的指令是否写寄存器的信号 |
| alucontrolE | Output | 5 | E阶段alu控制信号 |
| hilo\_weE | Output | 1 | E阶段写HILO寄存器使能信号 |
| div\_validE | Output | 1 | E阶段指令是否是除法操作信号 |
| div\_signE | Output | 1 | E阶段除法指令是否是有符号除法操作信号 |
| stallE | Input | 1 | E阶段暂停信号 |
| memtoregM | Output | 1 | M阶段的指令是否将内存中读出的数据写回寄存器信号 |
| regwriteM | Output | 1 | M阶段的指令是否写寄存器的信号 |
| memenM | Output | 1 | M阶段是否访存的使能 |
| cp0weM | Output | 1 | M阶段写CP0寄存器使能信号 |
| flushM | Input | 1 | M阶段清空信号 |
| stallM | Input | 1 | M阶段暂停信号 |
| memtoregW | Output | 1 | W阶段的指令是否将内存中读出的数据写回寄存器信号 |
| regwriteW | Output | 1 | W阶段的指令是否写寄存器的信号 |
| flushW | Input | 1 | W阶段清空信号 |
| stallW | Input | 1 | W阶段暂停信号 |

**模块分析：**

Controller模块的主要功能是在D阶段生成数据通路（包括hazard模块、hilo寄存器）以及存储器所需的各种控制信号，从而控制数据在数据通路上流动。Controller共分为三个部分，分别是负责指令控制信号译码的maindec，负责alu运算操作译码的aludec和三个寄存器：regE，regM，regW。

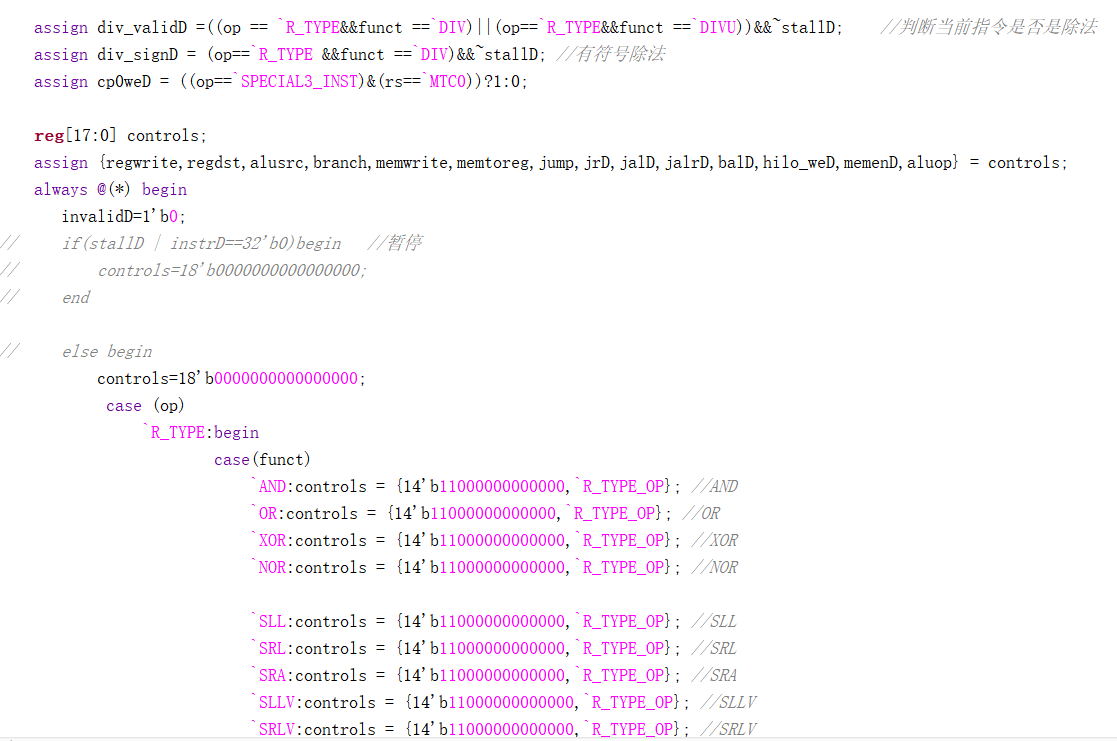
（六）maindec模块设计

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| stallD | Input | 1 | D阶段暂停信号 |
| instrD | Input | 32 | D阶段指令 |
| memtoreg | Output | 1 | D阶段的指令是否将内存中读出的数据写回寄存器信号 |
| branch | Output | 1 | D阶段是否是分支指令 |
| alusrc | Output | 2 | D阶段选择alu的第二个操作数b是来自寄存器、有符号扩展还是零扩展的信号 |
| regdst | Output | 1 | D阶段选择回写寄存器堆地址的信号 |
| regwrite | Output | 1 | D阶段的指令是否写寄存器的信号 |
| jumpD,jrD,jalD,jalrD,balD | Output | 1 | D阶段是否是各种跳转指令 |
| aluop | Output | 4 | D阶段不同类别指令的标志信号 |
| hilo\_weD | Output | 1 | D阶段写HILO寄存器使能信号 |
| div\_validD | Output | 1 | D阶段指令是否是除法操作信号 |
| div\_signD | Output | 1 | E阶段除法指令是否是有符号除法操作信号 |
| cp0weD | Output | 1 | D阶段写CP0寄存器使能信号 |
| invalidD | Output | 1 | D阶段发生指令异常信号 |

**模块分析：**

maindec模块的主要功能是在D阶段根据指令的类型对各类使能信号和选择器信号赋值，具体结构如下，其中alusrc占2位，aluop占4位，总共18位。



……

maindec模块首先判断instrD第26-31位，对于R型指令继续判断instrD第0-5位，对于bltz、bltzal、bgez、bgezal继续判断instrD第16-20位，对于特权指令继续判断instrD第21-25位。在分辨出所有指令后，对于不同的指令赋不同的controls值

（七）aludec模块设计

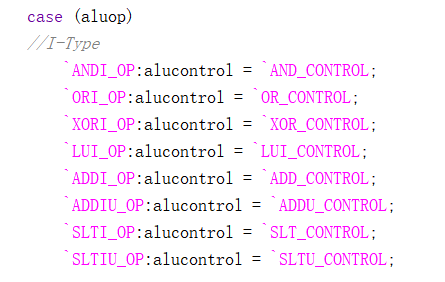
**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| instrD | Input | 32 | D阶段指令 |
| aluop | Input | 4 | D阶段不同类别指令的标志信号 |
| stallD | Input | 1 | D阶段暂停信号 |
| alucontrol | Output | 5 | D阶段alu控制信号 |

**模块分析：**

aludec模块的主要功能是确定不同类型的指令对应的alucontrol（alu 控制信号），控制信号通过datapath传入alu，在alu中通过对alucontrol的判断对每种指令执行相应的操作。

由于I型指令可以根据指令第26-31位的op code分辨，而R型指令第26-31位都是零，需要进一步判断第0-5位的funct code，所以在aludec模块中通过有关aluop的case语句将I型指令直接分辨出来，再判断instrD第0-5位的funct code功能码来分辨R型指令。此外对于Decode阶段的流水线暂停也有相应的控制。



（八）Exception模块设计

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| rst | Input | 1 | 复位信号 |
| except | Input | 8 | M阶段异常标志，每位代表不同的异常，1为产生该异常 |
| adel | Input | 1 | M阶段读地址异常信号 |
| ades | Input | 1 | M阶段写地址异常信号 |
| cp0\_status | Input | 32 | CP0内部处理器状态与控制寄存器 |
| cp0\_cause | Input | 32 | CP0内部存放上一次例外原因寄存器 |
| excepttype | Output | 32 | M阶段异常类型 |
| cp0weW | Input | 1 | W阶段写CP0寄存器使能 |

**模块分析：**

此模块按照不同的例外编码，对例外进行分类，如图5，进行后续的处理。

首先判断cause寄存器的8到15位，cause的8，9位代表待处理软件中断标志，每一位对应一个软件中断。第10到15位代表硬件处理中断标志，每一位代表一个中断线。当他们的值不为8'h00，代表会出现中断。接着对cp0的status寄存器进行判断，寻找到其第0和1位。status寄存器第1位在发生例外的时候会被置为1，当处于1的状态是，处理器自动处于核心态，所有硬件和软件的中断会被屏蔽epc、causebd在发生新的例外的时候不做更新。而当第0位是1的时候就代表使其所有硬件和软件中断。当满足上述条件的时候，excepttype便会被置为1。

用变量adel来表示读内存地址错例外，当except[7] == 1'b1 或者 adel便代表发生了读数据或者取指令的错误。用ades来代表写内存地址错例外，发生了写数据的错误。在对取址判断完毕之后，同理用sys代表系统调用例外，bp代表断点例外，ri代表保留指令例外，ov代表算出例外。特别注意的是保留指令例外中包含了syscall、Break、eret、invalid，通过判断except对应位数，便可确认例外的类型。



图5

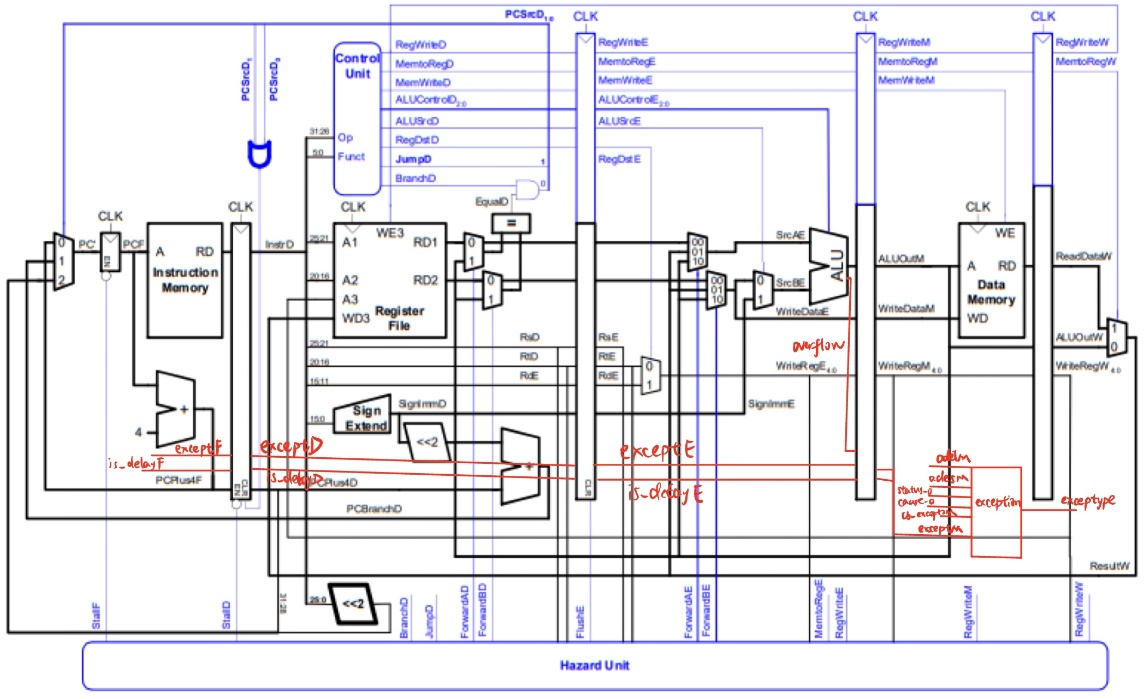
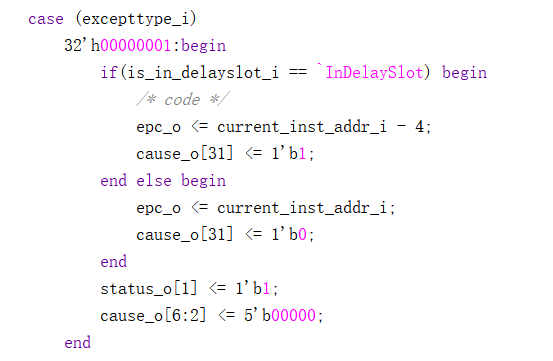
**数据通路图：**

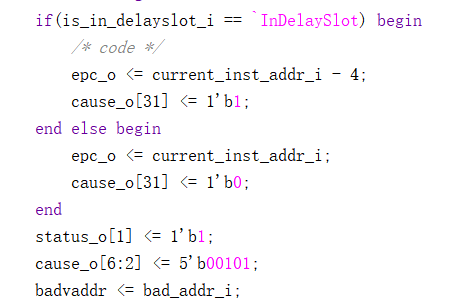
图6

（九）cp0\_reg模块设计\*

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| clk | Input | 1 | 时钟信号 |
| rst | Input | 1 | 复位信号 |
| we\_i | Input | 1 | 写CP0寄存器使能信号 |
| waddr\_i | Input | 5 | 写CP0寄存器的地址 |
| raddr\_i | Input | 5 | 读CP0寄存器的地址 |
| data\_i | Input | 32 | 写入CP0寄存器的数据 |
| int\_i | Input | 6 | 外部中断信号 |
| excepttype\_i | Input | 32 | 异常/中断类型信号 |
| current\_inst\_addr\_i | Input | 32 | 当前指令的指令地址 |
| is\_in\_delayslot\_i | Input | 1 | 当前指令是否在延迟槽中的信号 |
| bad\_addri | Input | 32 | 发生异常的访存地址 |
| data\_o | Output | 32 | 从CP0寄存器中读出的数据 |
| count\_o | Output | 32 | 处理器内部计数器寄存器 |
| compare\_o | Output | 32 | 对比寄存器 |
| status\_o | Output | 32 | CP0内部处理器状态与控制寄存器 |
| cause\_o | Output | 32 | CP0内部存放上一次例外原因寄存器 |
| epc\_o | Output | 32 | 存放上一次发生异常的指令的pc |
| config\_o | Output | 32 | （未用到） |
| prid\_o | Output | 32 | （未用到） |
| badvaddr | Output | 32 | 发生异常的访存的虚拟地址 |
| time\_int\_o | Output | 1 | 时钟中断信号 |

**模块分析：**

此模块直接使用参考代码中的CP0模块。在此模块中，可以根据异常的类型对CP0各个寄存器进行赋值。BadVAddr用于记录最新地址相关例外的出错地址；count是处理器内部计数器；status用于记录处理器状态与控制寄存器；cause用于存放上一次例外原因；epc存放上一次发生例外指令的 pc。按照例外的类型，分别对各种例外进行处理，对于普通的异常来说，通过except变量来判断的是否发生，但是软件中断是在CP0中上升沿触发写cause，当存在不止一个clk上升沿，因为exception模块是个组合逻辑会导致hazard模块中的刷新信号赋值提前，刷新掉前面的指令（详见三、错误14）。所以需要新定义一个变量cp0weW,把MEM阶段CP0寄存器的写使能信号传到W阶段，并传入exception模块中，与except一样在W阶段的一开始再更新写入cause的值。

cause的第31位标记最近发生例外的指令是否处于分支延迟槽，1代表在延迟槽中。判断完毕后将是否在延迟槽的信号is\_in\_delayslot\_i传到M阶段，作为CP0的输入信号如果触发异常的指令不在延迟槽中的话，epc直接触发例外的指令的 pc；如果触发异常的指令在延迟槽中的话，记录该指令前一条分支或跳转指令的 pc，同时 cause.BD 置为 1。其余例外指令同理。

**数据通路图：**

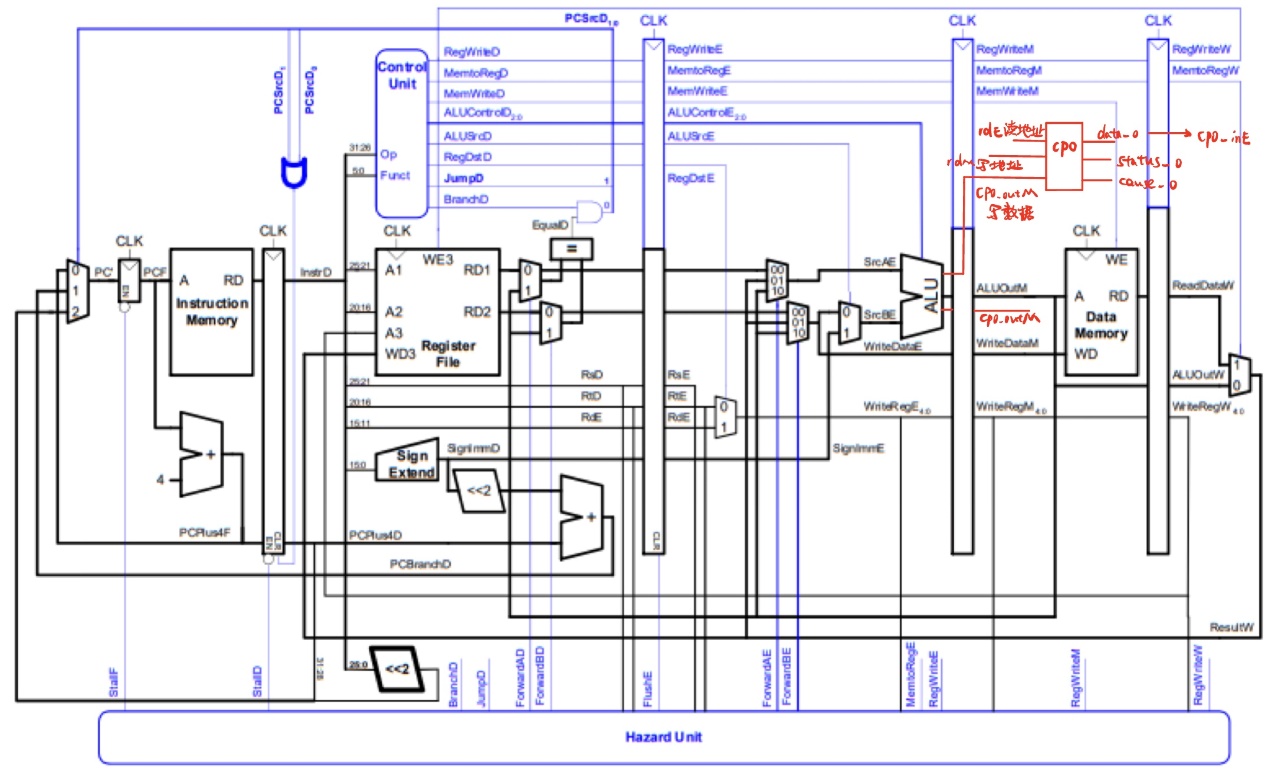
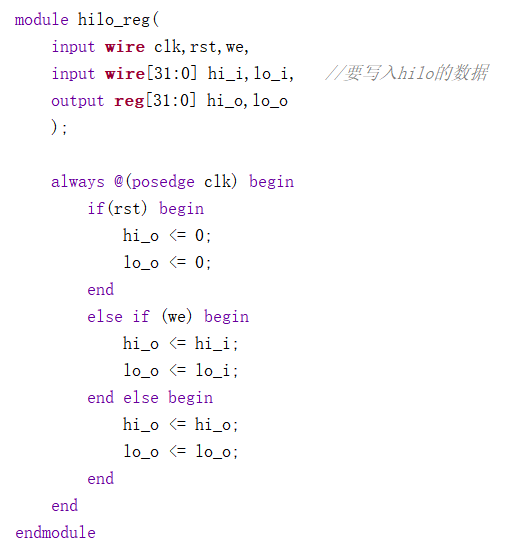


图7

（十）HILO寄存器模块设计\*

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| clk | Input | 1 | 时钟信号 |
| rst | Input | 1 | 复位信号 |
| we | Input | 1 | 写HILO寄存器使能信号 |
| hi\_i | Input | 32 | 写入HI寄存器的数据 |
| lo\_i | Input | 32 | 写入LO寄存器的数据 |
| hi\_o | Output | 32 | 读出HI寄存器的数据 |
| lo\_o | Output | 32 | 读出LO寄存器的数据 |

**模块分析：**

该模块直接使用参考代码中的HILO寄存器，其中定义了两个32位的特殊的寄存器—HI与LO。在乘除法计算中，两个32位的数相乘结果为64位，而除法的时候，除了商以外，还存在余数。为了避免32位通用寄存器存不下这些数的问题，使用 HI、LO寄存器来存放乘除法运算结果。

在该模块中包含两个部分，一部分是写入HI、LO，一部分是读出HI、LO。用hi\_i,lo\_i, 代表要写入HILO的数据，用hi\_o,lo\_o代表从HILO读出的数据，首先初始化为零，在接收到使能信号之后，为输出信号赋值。从寄存器堆或HI、LO寄存器中读出数据后，沿着流水线向后传递至W阶段，中间不需要做额外的运算，最终写入对应的寄存器中。hilo寄存器将在E阶段读取数据，在W阶段写回数据。MF开头的指令代表从HI、LO写回通用寄存器， MT开头的指令代表从通用寄存器写入HI、LO寄存器。

对于HI、LO写回通用寄存器的指令在E读取时会遭遇数据冒险，如果前面的指令需要对hilo寄存器进行写操作，因此需要将MEM阶段或者WB阶段将要写回hilo寄存器的数据前推到当前指令的EX阶段。

**数据通路图：**

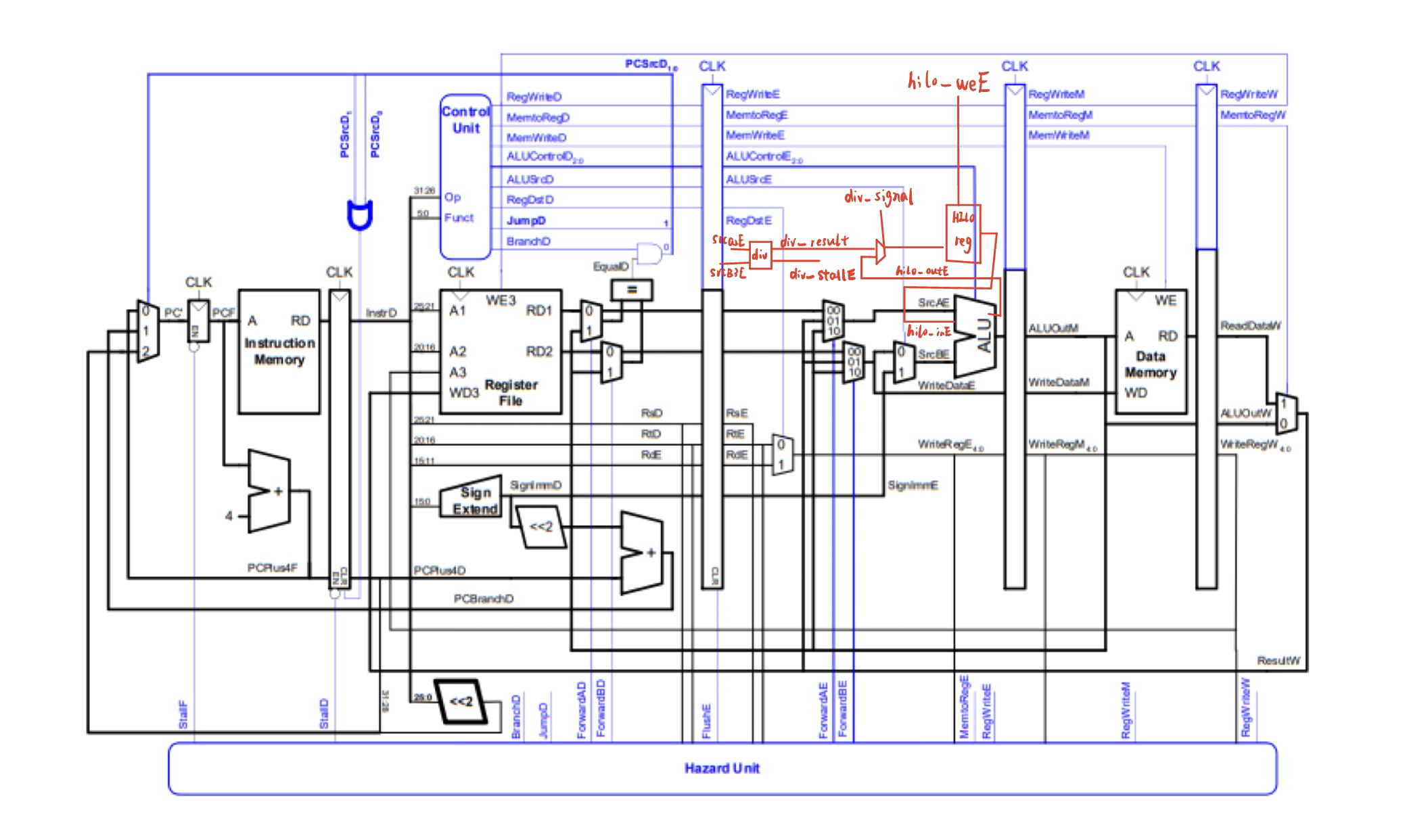


图8

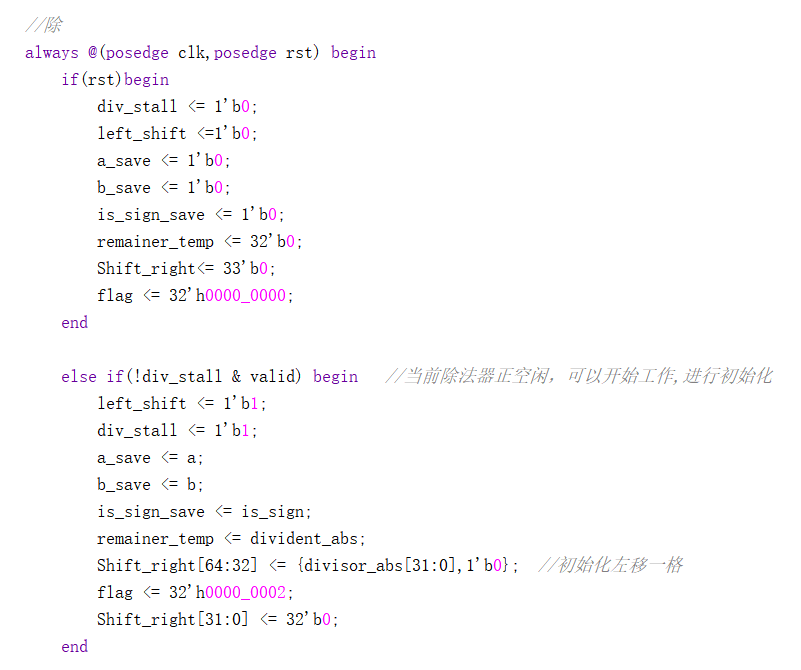
（十一）quick\_div模块设计（扩展）

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| clk | Input | 1 | 时钟信号 |
| rst | Input | 1 | 复位信号 |
| a | Input | 32 | 被除数 |
| b | Input | 32 | 除数 |
| valid | Input | 1 | 除法有效使能 |
| is\_sign | Input | 1 | 标志是否是有符号的除法运算的信号 |
| div\_stall | Output | 1 | 标志除法器是否正在计算中 |
| result | Output | 64 | 高32位是余数，低32位是商 |

**模块分析：**

该模块是按照计算机组成原理中快速除法器的部分，自己实现的快速除法器。

首先，将存放在除数是移位寄存器的高33位（详见三、错误1），商用移位寄存器的低32位的逆序来表示。在进行计算之前先对除数和被除数都取绝对值，符号在最后进行添加根据是否为有符号除法及被除数与除数的符号位来进行判断。

主要思路为当处于右移阶段说明在做除法：carry\_out为1，说明被除数减去除数大于0：更新被除数；反之不变。在更新被除数的时候用被除数-除数。在实际过程中，使用除法器还要先判断当前是否已经在使用。若（!div\_stall & valid）为1，则说明当前除法器正空闲，可以开始工作,进行初始化。当除法指令运行时，后续指令必须等待除法指令运行完再继续，所以需要将所有的变量暂停，因此除法器十分关键的一部分是暂停，当前E阶段在执行的时候，前面的D、F阶段需要增加stallF和stallD信号进行暂停。

在每次进行除法的时候先左移除数:扩大除数Shift\_right <= {Shift\_right[63:0],1'b0}; 直到carry\_out=0说明被除数小于除数，左移结束。当判断完毕被除数和除数之后，开始做除法，持续更新最新的商。最后除法结束的时候，使用一个标记位flag的1移动到第0位，除法结束。

当计算结束之后，需要对符号进行处理。当余数符号与被除数相同时，对二进制取反加一取其负数形式作为结果。

最后的余数表示为：



最后的商表示为：



最后的结果将商和余数结合写入HILO寄存器：



（十二）memory\_select模块设计

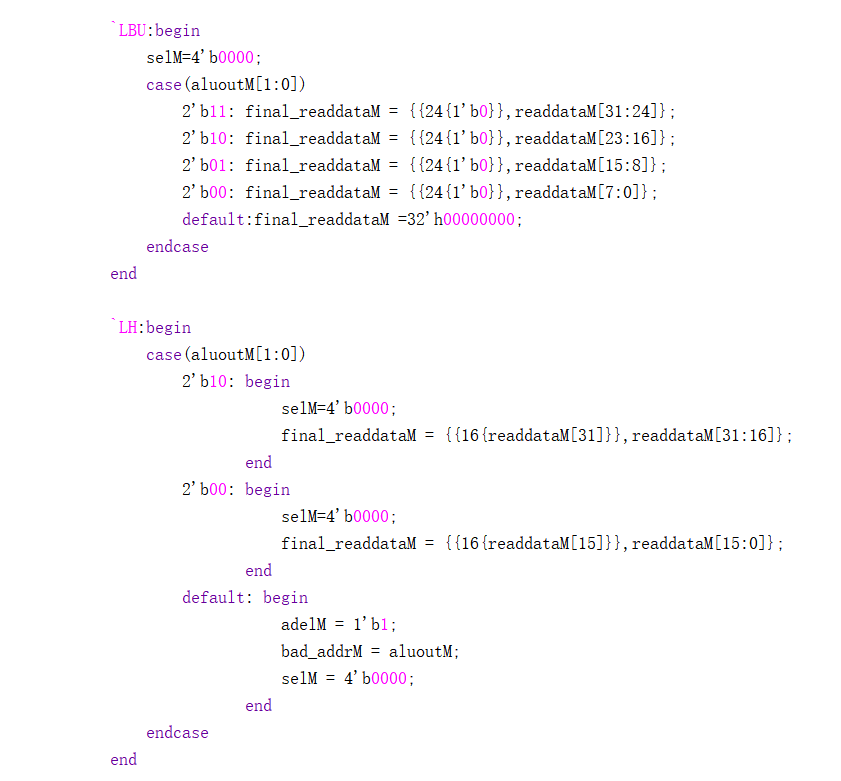
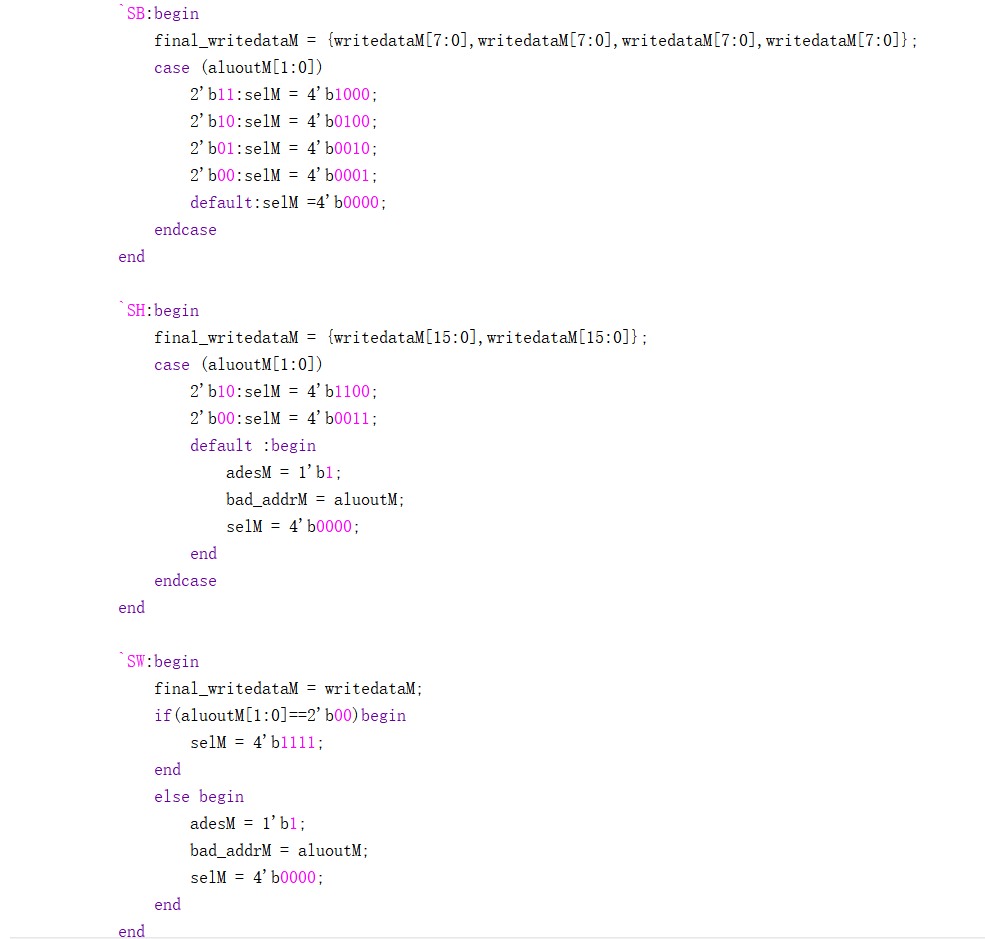
**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| pcM | Input | 32 | M阶段的指令地址 |
| opM | Input | 6 | M阶段的指令的高六位，用来区分是何种指令 |
| aluoutM | Input | 32 | M阶段alu输出，即访存地址 |
| readdataM | Input | 32 | M阶段从内存中读出的数据 |
| wrtiedataM | Input | 32 | M阶段要写入内存的数据 |
| selM | Output | 4 | 内存读/写数据的选择位 |
| final\_readdataM | Output | 32 | 最终从内存中读出的选择后的数据 |
| final\_writedataM | Output | 32 | 最终传给内存进行选择的写入数据 |
| adelM | Output | 1 | M阶段读地址异常信号 |
| adesM | Output | 1 | M阶段写地址异常信号 |
| bad\_addri | Input | 32 | M阶段发生异常的访存地址 |

**模块分析：**

此模块是指令需要访存时，根据不同的指令，如LB、LBU、LH、LHU、LW、SB、SH、SW这八条访存指令，进行访存相关信号的赋值。首先，根据6位的指令op判断是否是这八条访存指令，而后可以分为读内存和写内存两类进行处理。

对于读内存指令，是先把从内存中读取出来一个字后，即32位的readdataM，在此模块内，根据读内存指令需要读取一个字、半个字还是一个字节，结合读取地址的低两位和是有符号还是无符号操作进行选择与扩展。如下图显示的LBU和LH指令的操作。



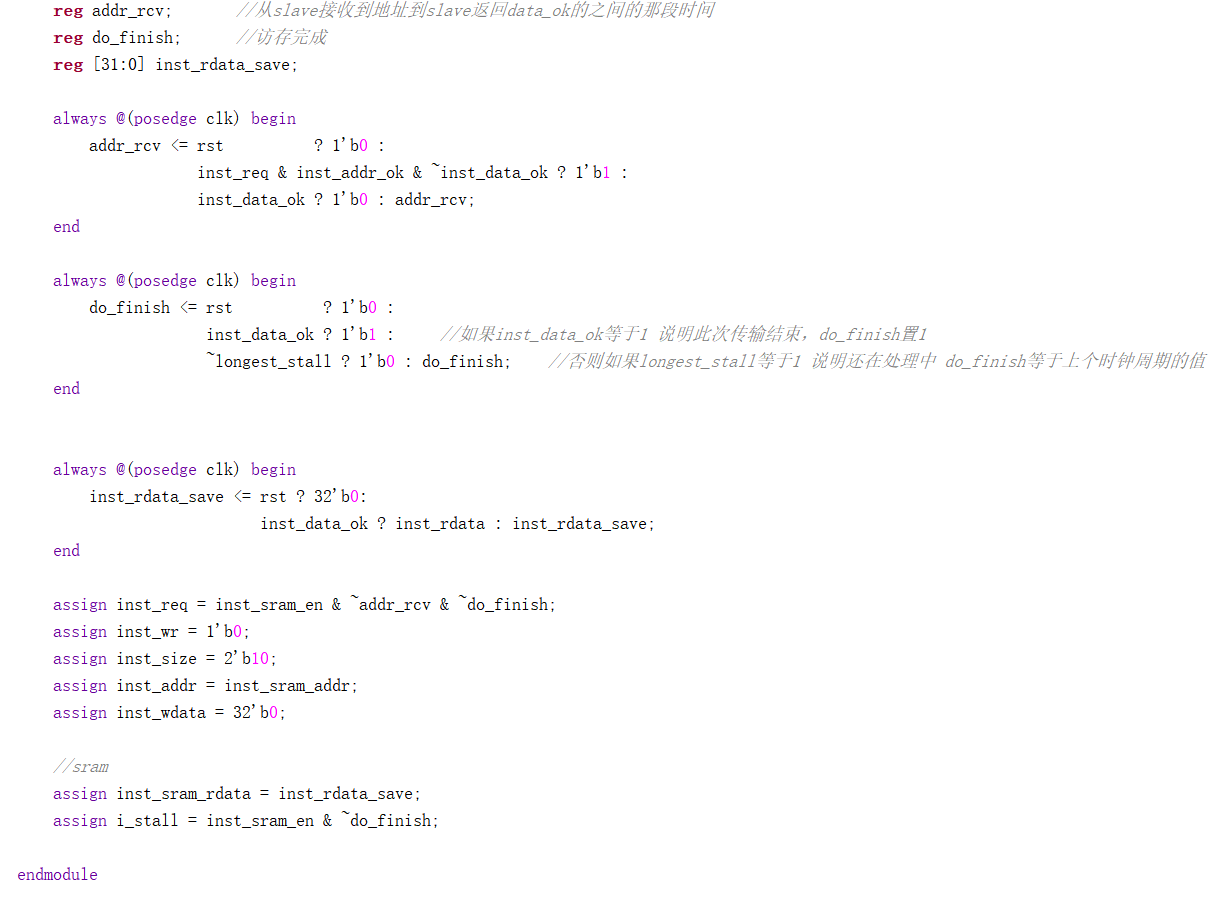
对于写内存指令，与读内存不同，是先对writedataM进行处理后，根据selM的值写入内存中。由于除了SW指令，其余指令写入的都是最低的1、2个字节[1,2],因此，final\_writedataM的值为对写入字节的扩展。而后，根据不同的写入字节数和写入地址的低两位赋值selM变量，将selM和final\_writedataM作为输出，传至内存的输入使能信号。

此外，在实现异常处理中，还在此模块增加了对地址低两位是否符合要求的判断，若不符合，则触发异常，使用adelM和adesM

（十三）inst\_sram\_like模块设计

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| clk | Input | 1 | 时钟信号 |
| rst | Input | 1 | 复位信号 |
| inst\_sram\_en | Input | 1 | 访问指令内存使能信号 |
| inst\_sram\_addr | Input | 32 | 访问指令内存地址 |
| inst\_sram\_rdata | Output | 32 | 从指令内存中读出的数据 |
| i\_stall | Output | 1 | 取指暂停信号 |
| inst\_req | Output | 1 | master向slave发送的请求指令信号 |
| inst\_wr | Output | 1 | master向slave发送的是否为写内存的信号 |
| inst\_size | Output | 2 | master向slave发送的读取/写入的字节数 |
| inst\_addr | Output | 32 | master向slave发送的访问内存的地址 |
| inst\_wdata | Output | 32 | master向slave发送的写内存的数据 |
| inst\_addr\_ok | Input | 1 | slave向master反馈的收到地址的信号 |
| inst\_data\_ok | Input | 1 | slave向master反馈的收到数据的信号 |
| inst\_rdata | Input | 32 | slave向master反馈的从内存中读出的数据 |
| longest\_stall | Input | 1 | CPU最长的暂停信号 |

**模块分析：**

在实现本模块前，指令内存和数据内存都内嵌在流水线中，单个周期从内存中读取数据，即SRAM接口的方式，但这种方式读写之间是有一个时钟延迟的。此外，读写共用地址通道，无法同时读写。又由于存储器和CPU之间的速度是不匹配的，即存储器的访存速度慢于CPU的速度，导致在CPU一个阶段内，内存并不能取出数据，因此不能把内存内嵌在流水线中（文献[4]），因为其限制了 CPU 的频率，使CPU 频率不能超过内存的读写频率。

因此，需要多个周期来访存，因此，构建此类SRAM接口模块，将CPU封装成类SRAM接口，进而利用类SRAM和AXI转换桥实现AXI接口。

在本模块的实现中，首先master端根据SRAM端口的使能和是否处于slave正在工作且CPU最长暂停结束发送req请求，同时发送inst\_wr、inst\_size、inst\_addr、inst\_wdata等信号进行握手，暂停CPU继续向后执行。在读内存操作时，slave端接收到req请求和相关变量后后，将inst\_addr\_ok信号拉高，master端各信号拉低，读地址握手成功。而后slave端进行处理，直至读出数据，将data\_ok信号拉高，并将数据传输给master端，读数据握手成功。在写内存操作时（inst没有、data有），master在发起请求时，要将wr使能信号拉高，同时传入写入的数据w\_data，slave返回的data\_ok信号，代表数据已成功写入。从图9中可以看出，类SRAM的读写操作是可以同时进行的[7]。

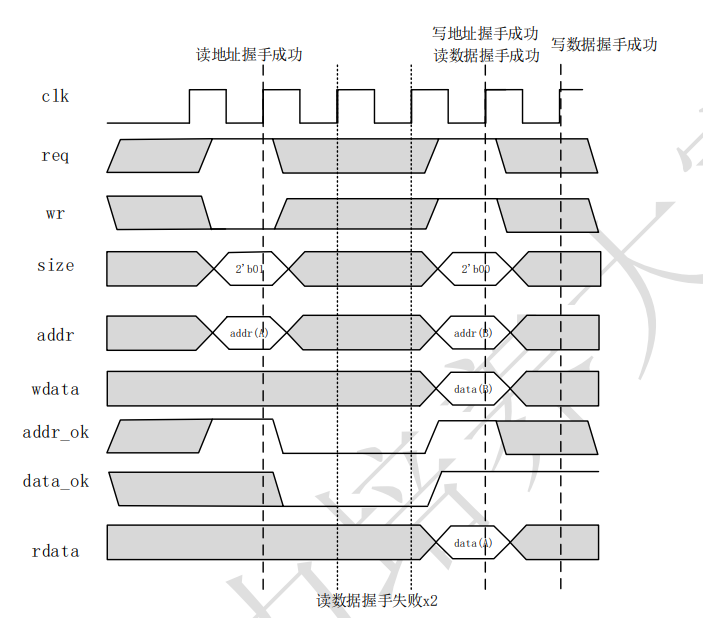


图9

（十四）data\_sram\_like模块设计

同（十三）

（十五）icache模块设计\*

**接口定义：**

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 方向 | 位宽 | 功能描述 |
| clk | Input | 1 | 时钟信号 |
| rst | Input | 1 | 复位信号 |
| cpu\_inst\_req | Input | 1 | cpu发送的请求指令信号 |
| cpu\_inst\_wr | Input | 1 | cpu发送的是否为写内存的信号 |
| cpu\_inst\_size | Input | 2 | cpu发送的读取/写入的字节数 |
| cpu\_inst\_addr | Input | 32 | cpu发送的访问内存的地址 |
| cpu\_inst\_wdata | Input | 32 | cpu发送的写内存的数据 |
| cpu\_inst\_rdata | Output | 32 | Cache命中反馈给cpu的读取的数据 |
| cpu\_inst\_addr\_ok | Output | 1 | 反馈cpu的收到地址的信号 |
| cpu\_inst\_data\_ok | Output | 1 | slave向master反馈的收到数据的信号 |
| cache\_inst\_req | Output | 1 | Cache发送的请求指令信号 |
| cache\_inst\_wr | Output | 1 | Cache发送的是否为写内存的信号 |
| cache\_inst\_size | Output | 2 | Cache发送的读取/写入的字节数 |
| cache\_inst\_addr | Output | 32 | Cache发送的访问内存的地址 |
| cache\_inst\_wdata | Output | 32 | Cahce发送的写内存的数据 |
| cache\_inst\_rdata | Input | 32 | 内存中读出反馈给cpu的读取的数据 |
| cache\_inst\_addr\_ok | Input | 1 | 内存反馈的收到地址的信号 |
| cache\_inst\_data\_ok | Input | 1 | 内存反馈的收到数据的信号 |

此模块直接调用参考代码中提供的写直达指令Cache，将CPU发送的请求作为输入传送到指令Cache中，若命中，直接返回给CPU所需的数据，否则，Cache再向内存发送请求，将返回的数据反馈给CPU。

（十六）cpu\_axi\_interface模块设计\*

直接调用参考代码中提供的cpu\_axi\_interface模块，将从指令cache中输出的信号和数据的类SRAM信号传入，将类SRAM接口转换为AXI接口。AXI具有地址/控制和数据总线相分离的特点、突发传输、独立的读写通道等特点，可以有效地提高性能[4]。

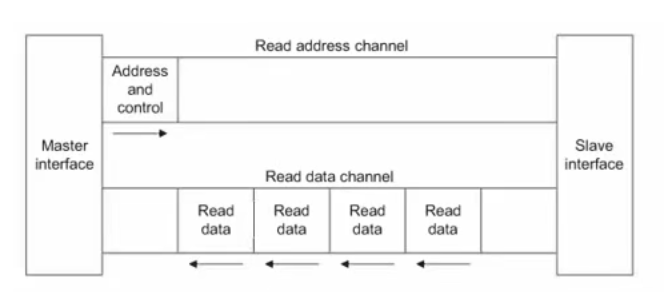


图10

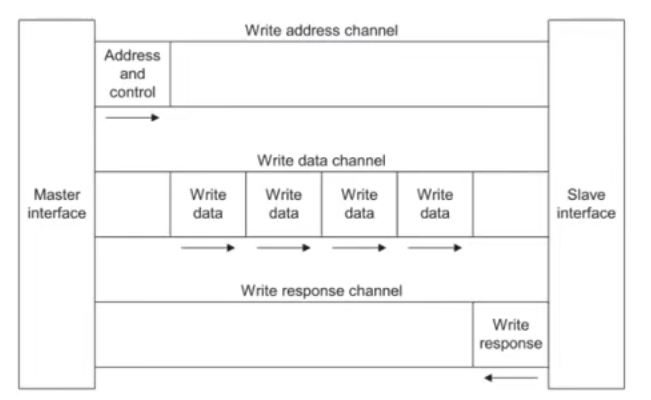


图11

三、实验过程（40%）

（一）设计工作日志

52条指令阶段：

**2022年12月21日**

* 20:00 – 22:00 完成所有逻辑运算指令的实现 使用funcTest\_independent测试通过，总计指令数8/52

**2022年12月22日**

* 20:00 – 21:00 完成所有移位运算指令的实现 使用funcTest\_independent测试通过，总计指令数14/52

**2022年12月23日**

* 15:00 – 20:00完成所有数据移动指令 + HILO寄存器的实现 使用funcTest\_independent测试通过，总计指令数18/52
* 22:00 – 23:00 完成除乘除法外的算术运算指令的实现 完成的指令使用funcTest\_independent测试通过，总计指令数28/52

**2022年12月24日**

* 15:00 – 22:00 完成乘法指令和除法指令的除法器的实现 使用funcTest\_independent测试通过，总计指令数30/52

**2022年12月25日**

* 10:00 – 12:00 15:00 – 20:00 完成除法指令的实现（找了很长时间bug，错误1、2）使用funcTest\_independent测试通过，总计指令数32/52

**2022年12月26日**

* 10:00 – 12:00 完成转移指令中BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ指令的添加，总计指令数38/52
* 15:00 – 20:00 完成转移指令中J、JR、JAL、JALR、BGEZAL、BLTZAL指令的添加，总计指令数44/52

**2022年12月28日**

* 19:30 – 00:00 完成访存指令中LB、LBU、LH、LHU、LW、SB、SH、SW指令的添加，总计指令数52/52

**2022年12月29日**

* 18:30 – 19:30 使用funcTest\_independent通过观察仿真图对LB、LBU、LH、LHU、LW、SB、SH、SW进行测试，解决错误3，测试通过 总计指令数52/52

**2022年12月30日**

* 21:00 – 23:00 为了使用vivado的方式进行测试添加了debug\_wb\_pc、debug\_wb\_rf\_wen、debug\_wb\_rf\_wnum、debug\_wb\_rf\_wdata接口与正确答案进行比对

**2022年12月31日**

* 00:00 – 1:00 测试出现下述错误4，debug中…… 前52条指令测试进度：0/64

**2023年1月1日**

* 10:00 – 14:00 14:00 – 19:00 解决错误4 成功跑通前51个测试点 前52条指令测试进度：51/64
* 19:30 –20:00 解决错误5 成功跑通前64个测试点 前52条指令的阶段正式结束 前52条指令测试进度：64/64

57条指令阶段：

**2023年1月2日**

* 10:00 – 13:00 添加了特权指令BREAK、SYSCALL、ERET、MTC0、MFC0，此外添加了exception模块对异常/中断类型进行了判断。对相关模块如controller、datapath、hazard等模块进行更新。总计指令数57/57
* 11:00 – 13:00 对相关模块如controller、datapath、hazard、aludec、maindec等模块进行整合更新。
* 15:00 – 18:00 添加了CP0模块，对异常处理模块进行了测试，debug中…解决错误7、8 57条指令测试进度：74/89
* 18:30 – 20:00 debug中…解决错误9 57条指令测试进度：89/89

AXI+Cache阶段：

**2023年1月3日**

* 10:00 – 12:00 在之前的sram接口上，添加了类sram接口，改变了mips模块和mycpu模块的结构
* 15:00 – 23:00 尝试添加axi接口和Cache，并进行测试，debug中…解决错误10、11 axi功能测试进度：1/89

**2023年1月4日**

* 10:00 – 12:00 继续debug加了axi接口和Cache后的功能测试的错误，解决错误12 axi功能测试进度：43/89
* 15:00 – 1:00 继续debug加了axi接口和Cache后的功能测试的错误，解决错误13 axi功能测试进度：76/89

**2023年1月5日**

* 10:00 – 14:00 继续debug加了axi接口和Cache后的功能测试的错误，解决错误14 AXI功能测试进度：89/89

**2023年1月6日**

* 10:00 – 12:00 使用添加了AXI接口和Cache后的CPU进行了性能测试，失败
* 15:00 – 22:00 去除了Cache，仅对添加了AXI的CPU进行了性能测试，成功

**2023年1月7日**

* 10:00 – 14:00 仅添加了指令Cache，对添加了AXI接口和指令Cache的CPU进行了性能测试，成功

**2023年1月8日**

* 20:00 – 22:00 调通了verilator，使得可以利用verilator工具进行快速编译了

**2023年1月9日**

* 15:00 – 15:30 答辩

（二）主要的错误记录

本部分是在实现本项目的过程中，遇到问题并成功解决后，及时记录的，因此记录了包括模块设计、仿真、测试等特别是debug的所有的、有意义的错误。由于一开始verilator上编译不出来结果，所以都是在vivado上调试的，后来发现是由于在快速除法器中使用了for循环，改变了代码后，可以使用verilator快速编译了。

1、错误1

（1）错误现象

在添加除法指令时，采用除数移位的方式进行运算，一开始并没有将除数左移一位，直接将除数放到64位移位寄存器的高32位，后续又通过左移移位寄存器来判断何时除数大于被除数，再将移位寄存器右移回去开始除法，但这样做的话，如果除数是无符号数且最高位为1的话，这一位就会在左移后，再右移被0覆盖掉而丢失。

（2）分析定位过程

此问题是在对除法进行测试前，与给定的参考代码中的除法器进行对比发现的这个问题，因为给定代码中将移位寄存器设置为了65位，百思不得其解，后来经过反复的思考后发现此问题。如果这个问题现在不解决，以后可能会产生比较大的影响。

（3）错误原因

错误原因就是因为上述由于除法移位会导致无符号除数最高位丢失。

（4）修正效果

修正此问题的方法是使移位寄存器多扩展一位，同时，在将除数放入移位寄存器时，先将其左移一位变成长度为33位后并放置到移位寄存器的[64:32]位上，左移一位后，由于除数变成33位而被除数仍是32位，这如果出现除数是无符号数且最高位为1时，也就是除数的第33位为1，那么一定比被除数大，这样除数就不可能左移，这一位也不会再因为左移后右移而丢失。

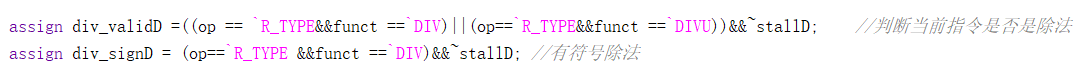
（5）归纳总结（可选）

这个错误是对数据进行移位时，最边上的一位数据会丢失而产生的。今后应对这种边界情况考虑的更全面。

2、错误2

（1）错误现象

在添加除法指令时，一开始在main\_decorder中的div\_validD和div\_signD没有增加stallD条件，导致后续除法器在第一次除法运算后错乱，结果错误。



（2）分析定位过程

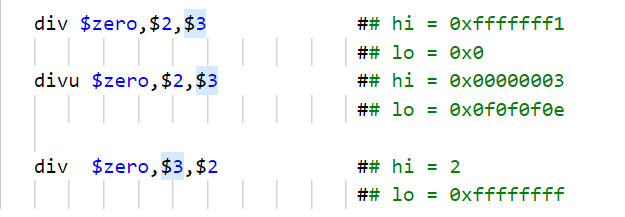
通过仿真，发现div\_validD和div\_signD在暂停阶段的值仍一直为高电平，后来，通过去代码中找到这两个变量，发现赋值时少加了暂停的控制条件。

（3）错误原因

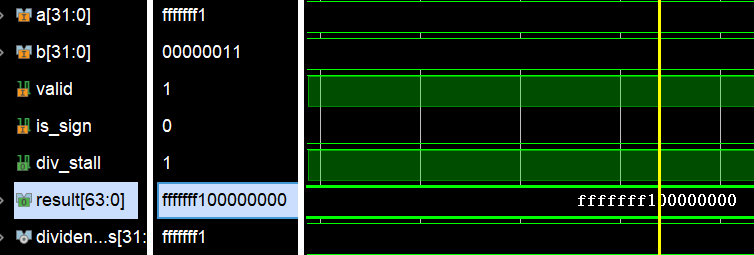
除法要涉及到暂停问题，当当前除法指令在E阶段运行时，此时处于E阶段前的D阶段和F阶段都应该暂停，所以增加了一个stallE信号暂停E阶段。而D阶段工作的模块main\_decorder、alu\_decorder使用stallD暂停工作。

（4）修正效果

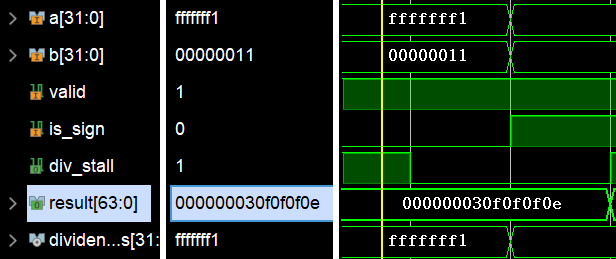
更正后，使用funcTest\_independent仿真正确：



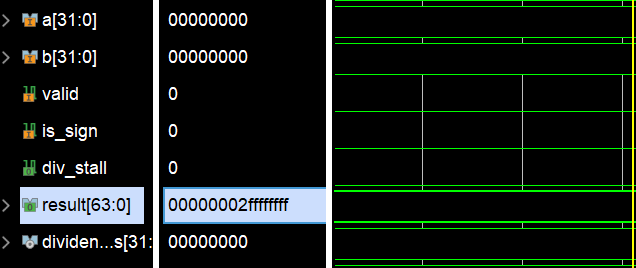
第一条除法指令结果：



第二条除法指令结果：



第三条除法指令结果：



可以看到修正成功。

（5）归纳总结（可选）

这个错误属于流水线暂停问题。当除法指令运行时，后续指令必须等待除法指令运行完再继续，所以需要将所有的变量暂停，等待除法指令运行结束。此错误是因为疏忽，只对各阶段间的寄存器使用stall进行控制，而忘记了变量在赋值时也要考虑到stall，今后考虑要更加细致、全面，否则这种错误找起来很头疼。

3、错误3

（1）错误现象

在添加访存指令时，设置了一个四位sel变量传入到数据存储器data\_ram的wea端口中，用来指定写入的比特位，但一开始使用对指令的OP进行case判断来确定sel的值，但一开始没有设置default:sel <= 4’b0000这一项（不为访存（存储）指令时，sel应该等于4’b0000），导致在测试中，一开始sb指令后，紧随着的srl指令的sel也跟sb一样，又因为wea的值始终为一，导致srl指令也将result写入了数据存储，导致错误。

（2）分析定位过程

通过将data\_ram的变量拉到仿真中，观察各变量随指令变化的变化，发现此错误。

（3）错误原因

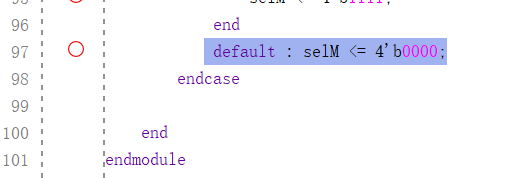
在使用case时，没有设置default项，导致sel在存储指令更改后始终为定值，无法回到默认值（4’b0000）。

错误仿真：

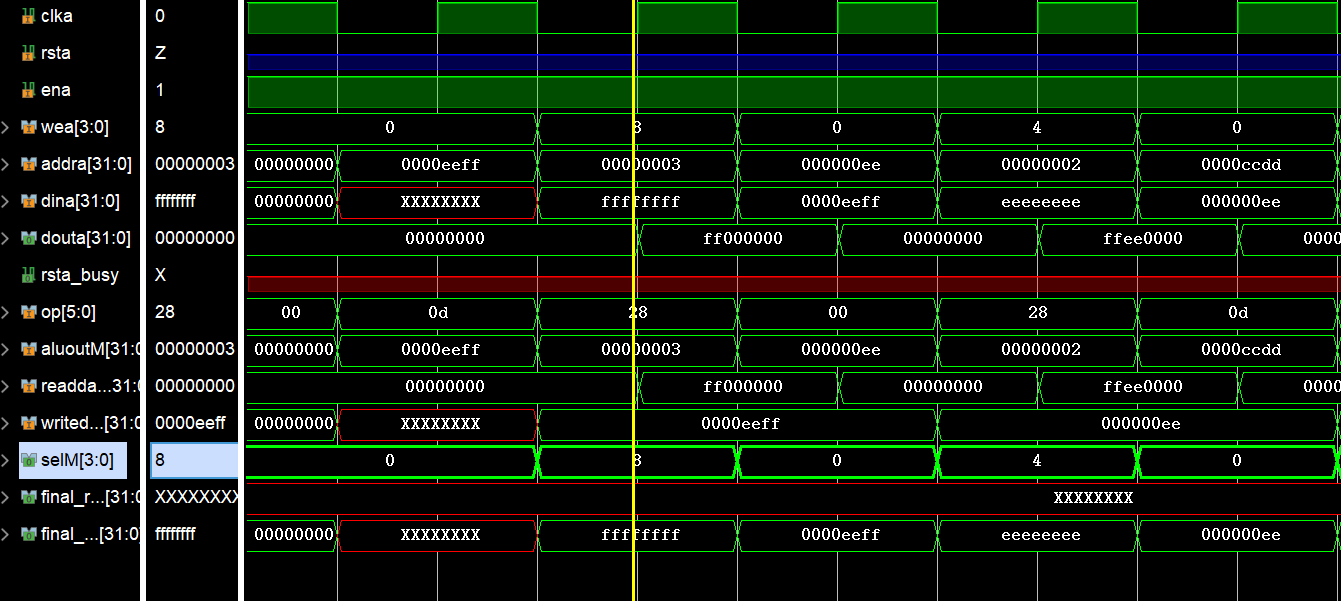


（4）修正效果

添加default项即可修正成功：



修正后仿真：

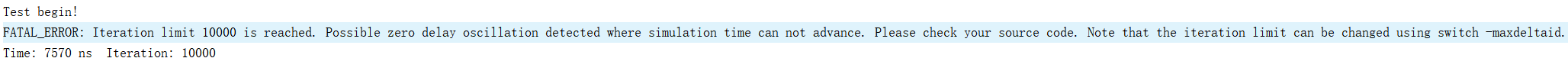


（5）归纳总结（可选）

4、错误4

（1）错误现象

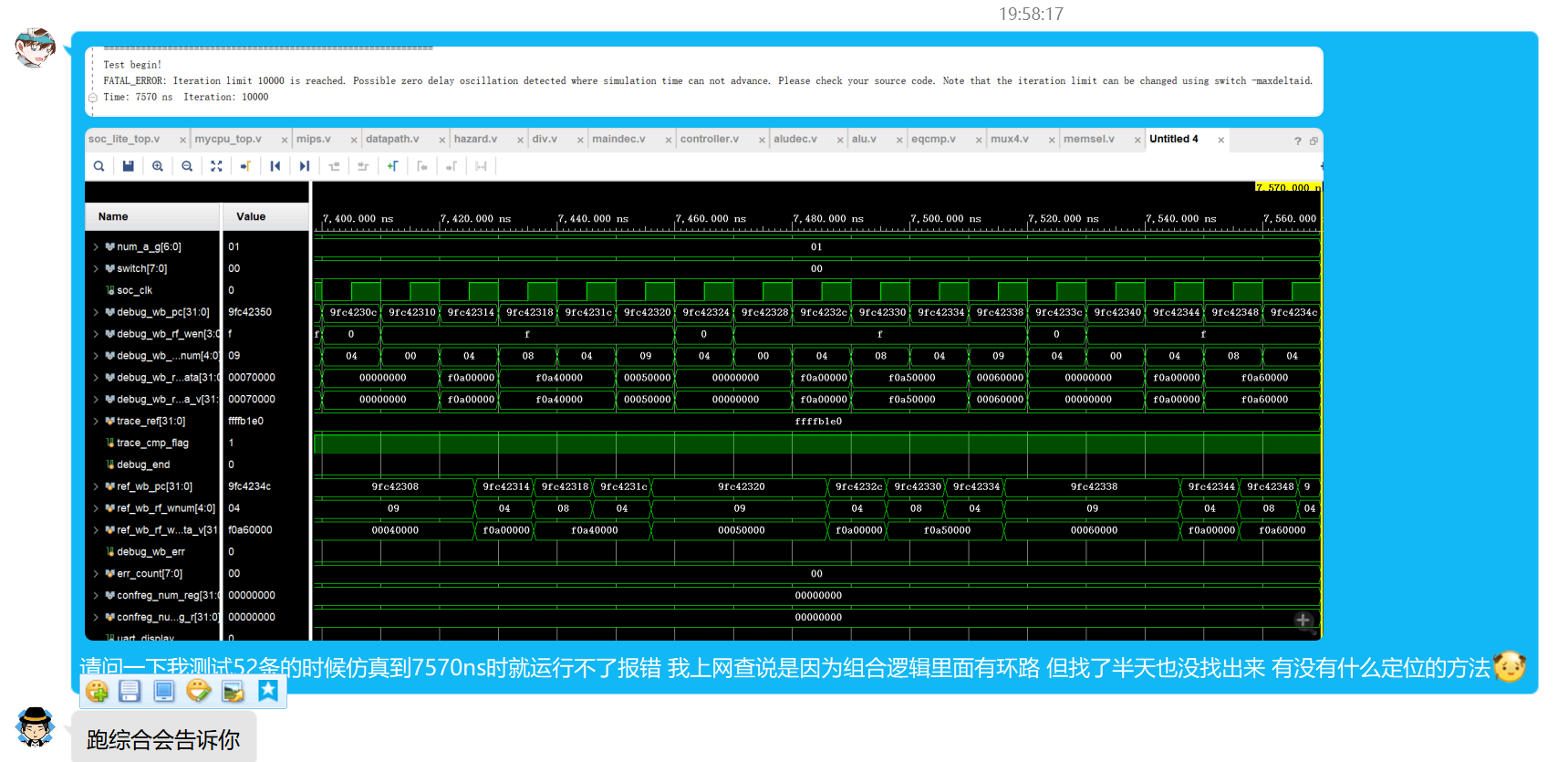
使用sram\_func对52条指令进行测试时报错：FATAL\_ERROR: Iteration limit 10000 is reached. Possible zero delay oscillation detected where simulation time can not advance. Please check your source code. Note that the iteration limit can be changed using switch -maxdeltaid.



可以看到在仿真到7570ns时，出现错误。

（2）分析定位过程

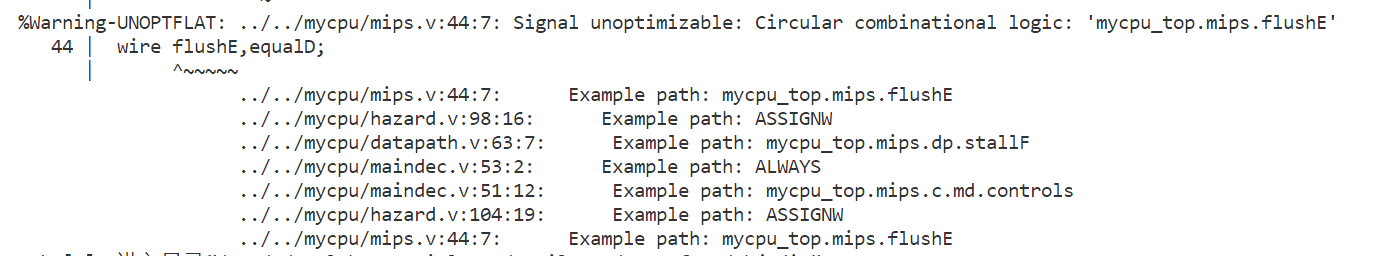
一开始并不知道知道如何定位此错误，后来通过询问助教，从综合中的警告中找到代码有问题的地方，并将其改正。



后来，看了综合之后，只知道是产生了critical warning：time loop的错误，即有组合逻辑环路，但不知道是哪里产生了环路，再进一步询问助教后，使用verilator工具进行编译，定位出了错误地址：



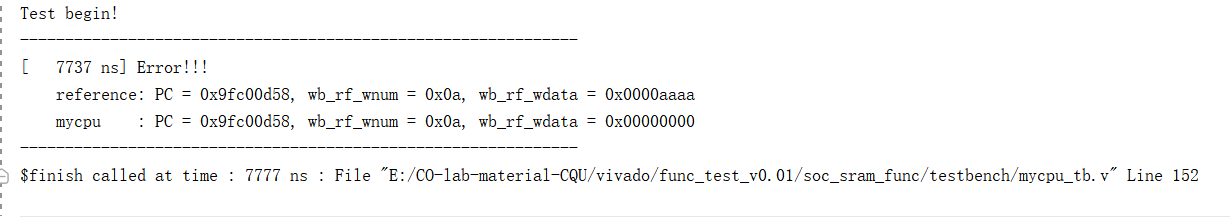
Verilator：

****

（3）错误原因

通过上网查阅资料得知此错误是因为在代码里面出现了回环，通常是组合电路的问题。比方说在一个组合逻辑块里面，对敏感变量进行赋值。作为敏感变量，只要变化，就会触发组合逻辑块的赋值，而赋值又会立马让敏感变量变化，然后再触发组合逻辑块赋值。这样循环往复，每次触发变化的时间，是几乎可以忽略不计的。一旦敏感变量触发组合电路的赋值，便会不断地，触发--赋值--触发---赋值……并且组合电路动作，几乎不花时间，这样实际上就是一个死循环。

（4）修正效果



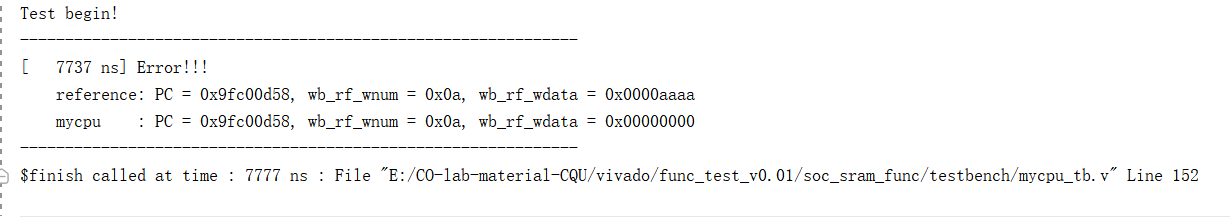
可以看到又有新的错误了，不过经过了一天时间好在终于跑过7570ns这一道坎了。

（5）归纳总结（可选）

这是一个组合逻辑中的错误。

5、错误5

（1）错误现象



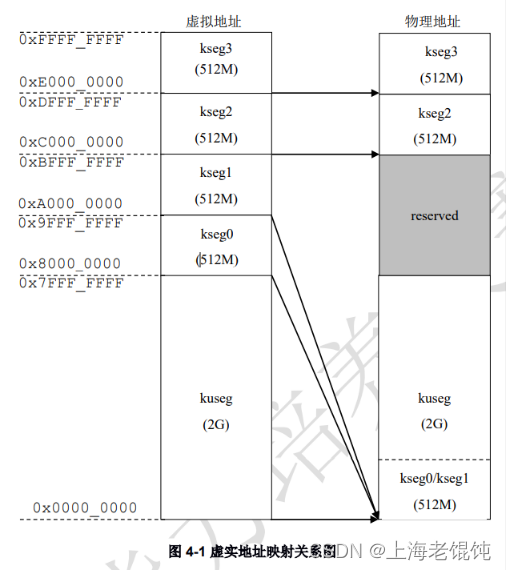
可以看到此条指令的wb\_rf\_wdata出现错误，即写回寄存器的值出现错误，而这条指令是lw指令，也就是从内存中读出来的数据是错误的。

（2）分析定位过程

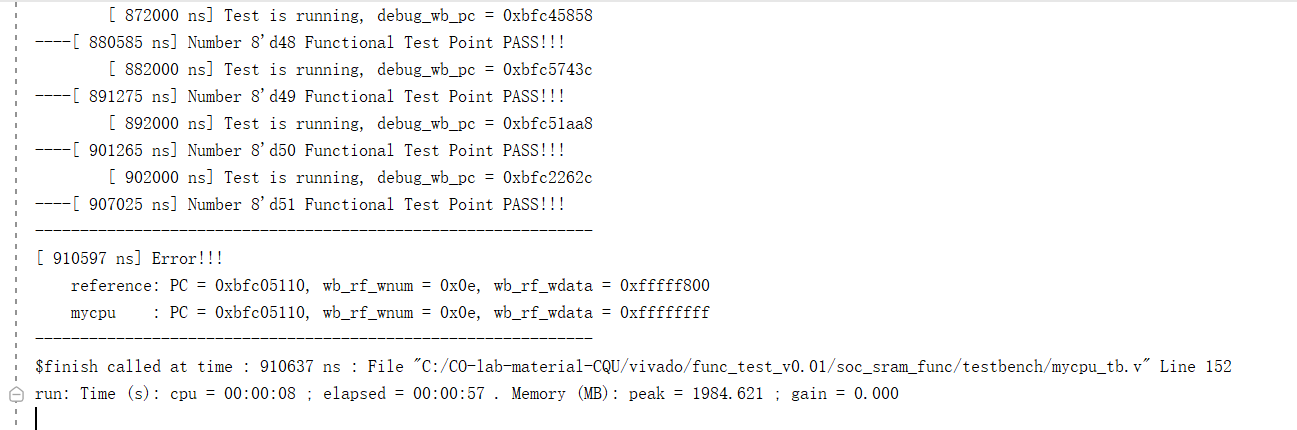
这是条访存指令，与内存有关。突然想到之前说过有个mmu模块，通过阅读第一次课的ppt了解到mmu是为了将虚拟地址转换成物理地址访问内存，而我的代码此时并没有添加mmu模块，全部是用虚拟地址访问的内存，不仅会越界，还会导致读写混乱。

（3）错误原因

未添加mmu模块进行虚拟地址到物理地址的映射：



（4）修正效果



修正后一口气跑通了50个测试点，遇到了新的错误。

（5）归纳总结（可选）

6、错误6

（1）错误现象

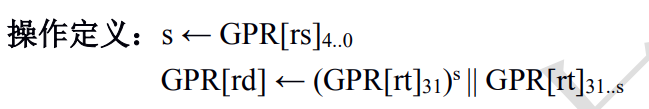


（2）分析定位过程

根据报错的指令pc，查看测试文档，得知此条指令是SRAV指令，而后通过阅读“系统能力培养大赛”MIPS指令系统规范和硬综文档发现错误。

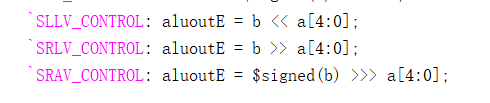
（3）错误原因

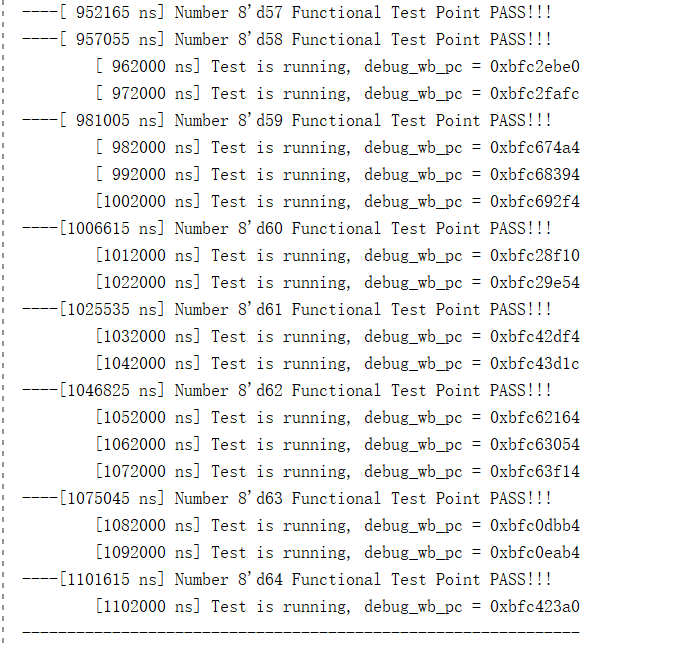
此错误是因为没有注意到，SRAV、SLLV、SRLV指令的移位位数应该是rs号寄存器中的值低第五位，一开始以为移位位数是rs号寄存器的值。



（4）修正效果

将移位位数取0-4位即可:

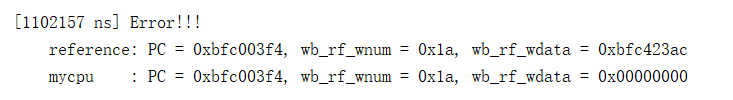




可见，前52条指令已经成功通过前64个测试点，进入下个阶段。

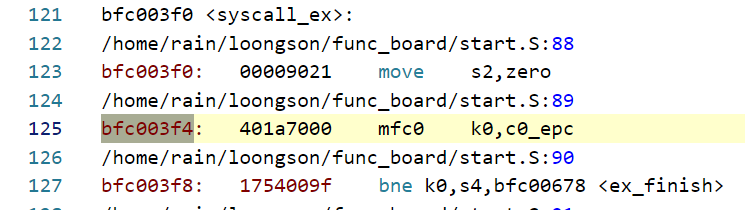
7、错误7

（1）错误现象

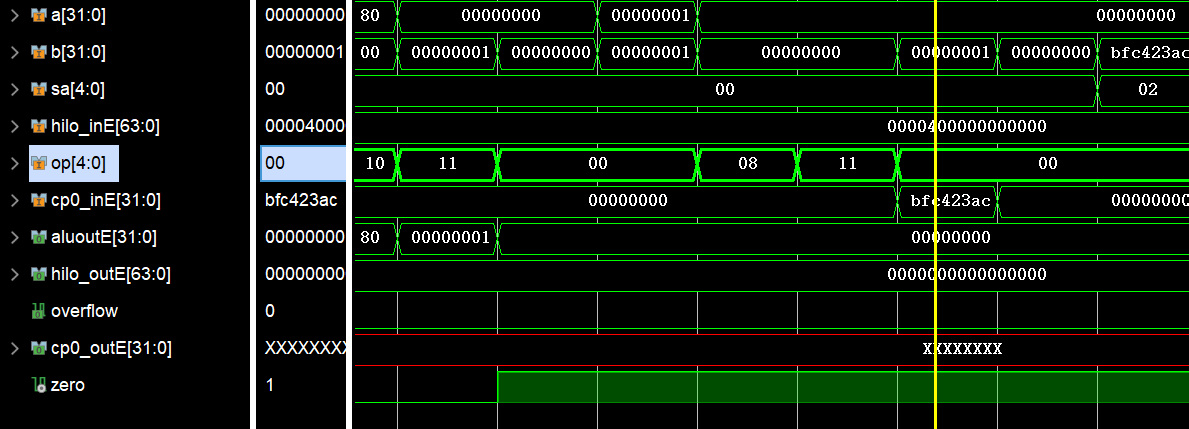


（2）分析定位过程

首先查看测试文件，得知此条指令是mfc0指令。



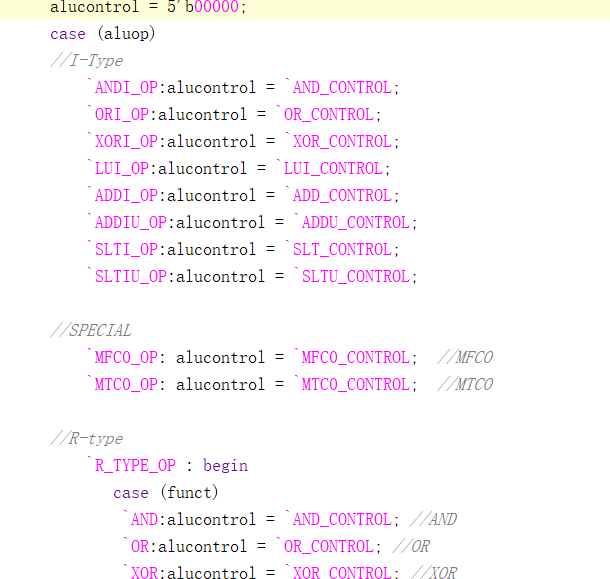
然后，通过观察alu的各变量取值，发现cp0\_inE的值实际上是正确的，但op（alucontrol）为0，说明问题出在这里。



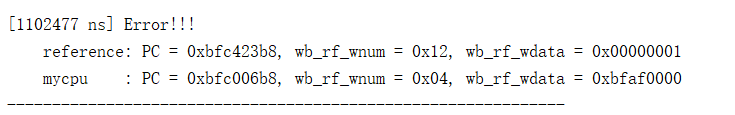
（3）错误原因

错误原因是因为在aludec文件中，在使用case进行判断给alucontrol赋值时，将MTCO和MFC0的op放在了错误的位置，alucontrol没有被正确赋值，为default情况下的默认值5’b00000。

（4）修正效果

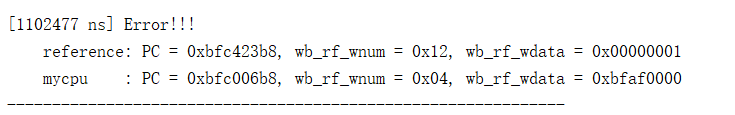


改正后即可，遇到新错误如下：

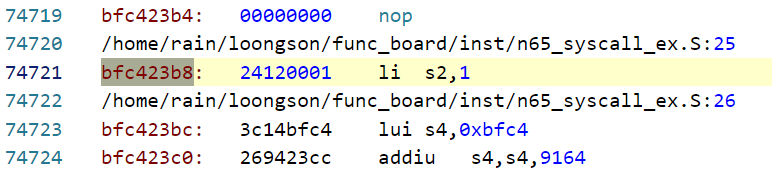


8、错误8

1. 错误现象



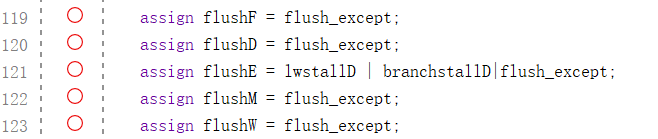
1. 分析定位过程



首先查看测试文件，得知此条指令是li指令。通过观察仿真图发现我的cpu在后面也跑到了此条ref指令：0xbfc243b8，写回寄存器的值也都是正确的，只不过是相隔了几个周期。因此，我感觉是stall或者flush的问题，于是去检查了各阶段stall和flush变量由于新增加的异常所赋的值，发现了错误。

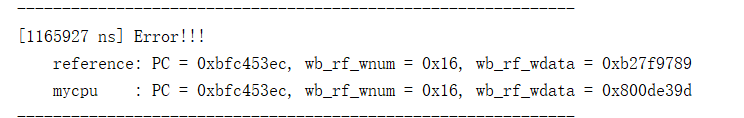
（3）错误原因

错误原因是因为在hazard模块中，在产生异常时并没有给flushE赋值，导致D阶段到E阶段的指令并没有被flush掉而是继续执行，导致产生了偏差。



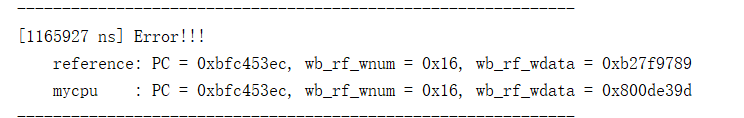
（4）修正效果

更改flushE后，此条指令正常，产生新错误如下：

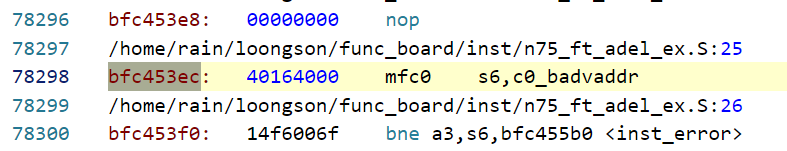


9、错误9

1. 错误现象



1. 分析定位过程

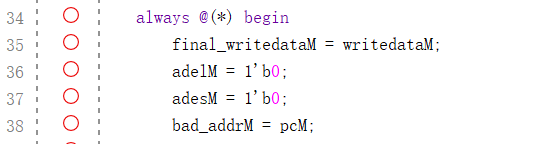


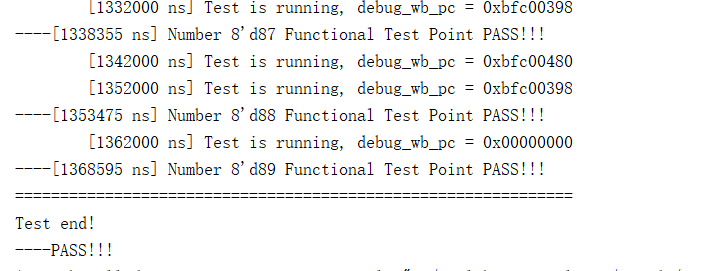
可以看出此条指令的pc是正确的，所以不是跳转之类的问题，然后通过查询测试文件，得知此条指令是mfc0指令，通过观察仿真图，得知alucontrol等信号均是正确的，说明cp0寄存器中存储的值不正确。而从测试文件中可以得知此条mfc0指令指令读出的值是cp0中的badvaddr这一寄存器，所以是badvaddr出现了问题。而badvaddr是记录的错误地址，通过追踪badvaddr变量发现错误存在于memory\_select模块中。

1. 错误原因

Badaddr分为读写数据和取指令地址异常，我一开始只记录了读写数据地址异常而忽视了取指令地址异常。在初始化memory\_select模块时，badaddr并没有赋初值（当前pc的值），也就是pc取指令时出现的错误的虚地址，仅仅在部分访存指令中（低两位出现问题）记录了aluoutM的值，因此导致错误的地址缺失。

1. 修正效果



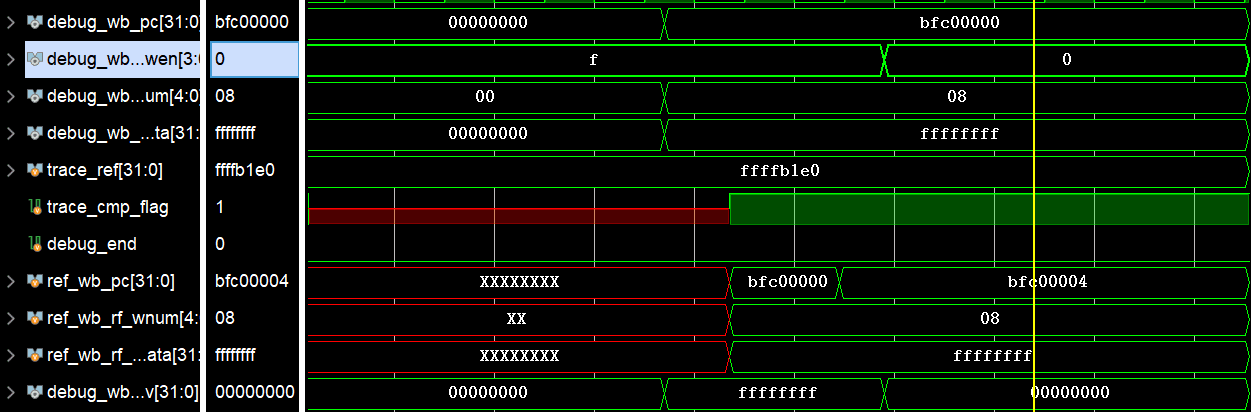


由此可见，57条指令测试通过。

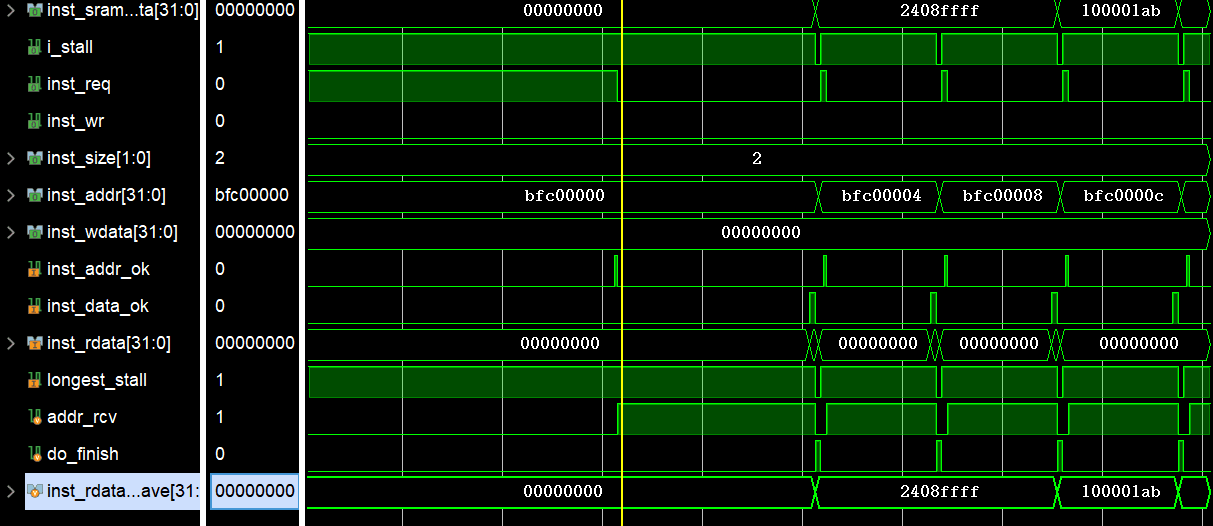
10、错误10

(1) 错误现象

在添加了axi接口后的第一次仿真进行功能测试时，发现虽然debug写回阶段的各个变量都是正确的，但debug和ref的pc值相隔了一个时钟周期，而查看指令类sram接口也并没有百思不得其解。



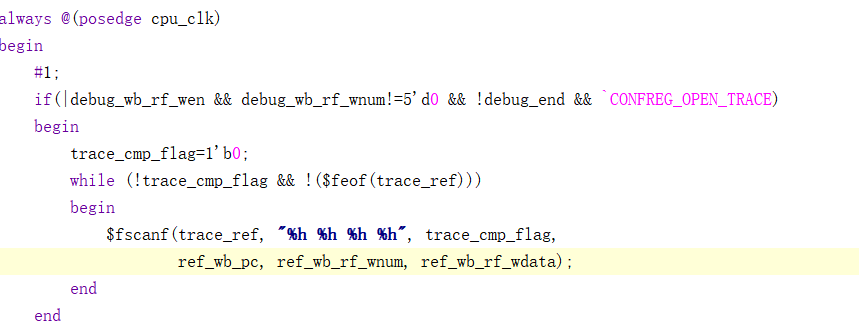
指令类sram接口信号：



1. 分析定位过程

通过询问助教得知我的问题在于在stallW阶段时，我的regwrite信号仍为高电平，导致在测试时，尽管W阶段在取指时暂停了，但测试文件中由于regwrite仍为高电平，满足了导入trace文件的条件，所以在debug的W阶段仍暂停时，就导入了trace进行对比，这也就使他们之间错位了。





1. 错误原因

在stallW阶段时，regwrite信号仍为高电平，导致在测试时，尽管W阶段在取指时暂停了，但测试文件中由于regwrite仍为高电平，满足了导入trace文件的条件，所以在debug的W阶段仍暂停时，就导入了trace进行对比，使他们之间错位了。

1. 修正效果

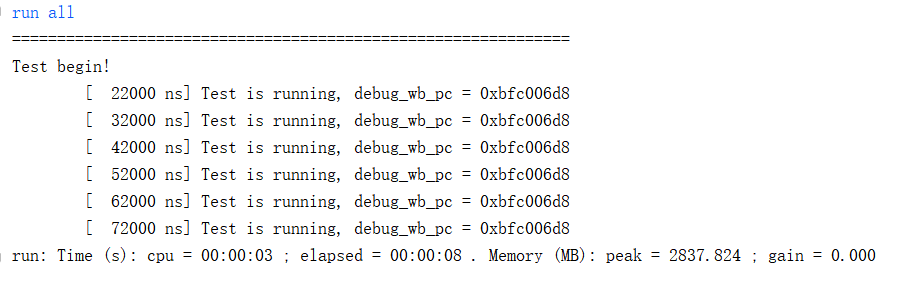
对w阶段的写回寄存器信号使用stallW变量进行约束，即可解决此问题。



11、错误11

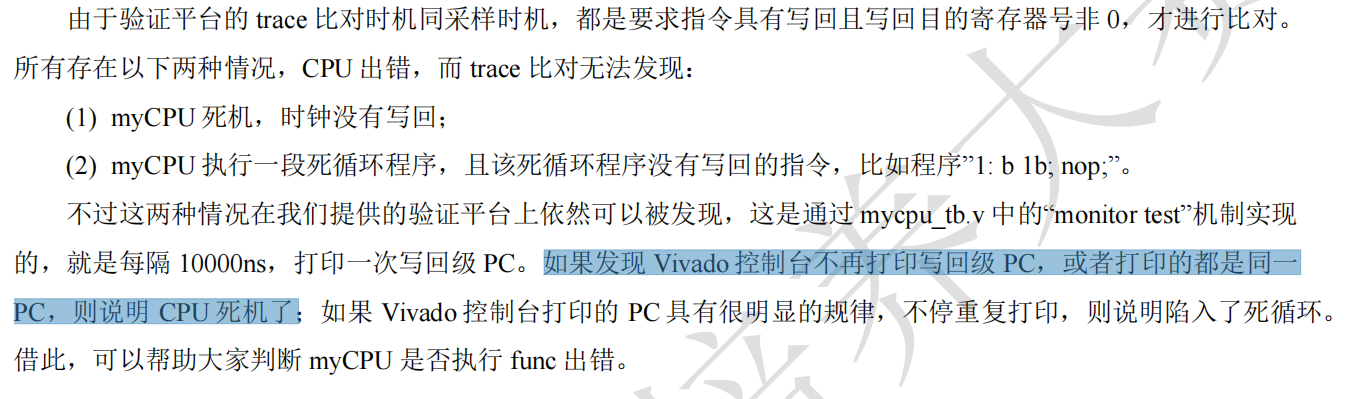
(1) 错误现象

测试时，控制台不断打印相同pc值，cpu陷入死循环。



1. 分析定位过程

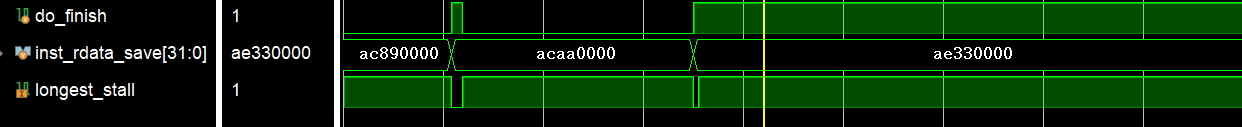
通过查看调试文档，得知此错误可能发生的原因。



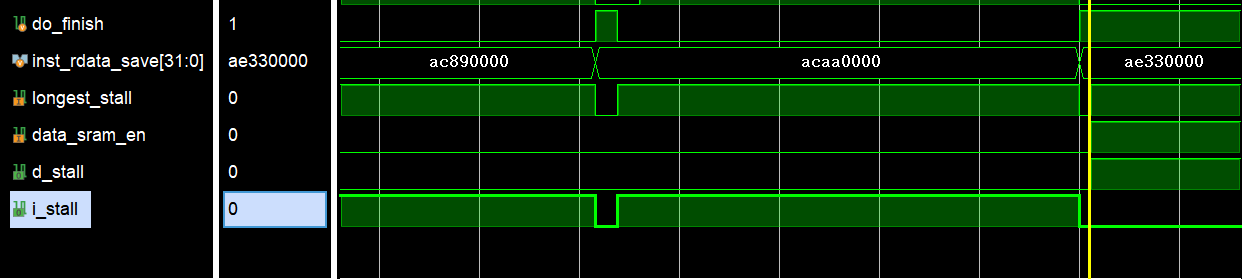
进一步通过观察仿真图，得知真正错误原因。

1. 错误原因

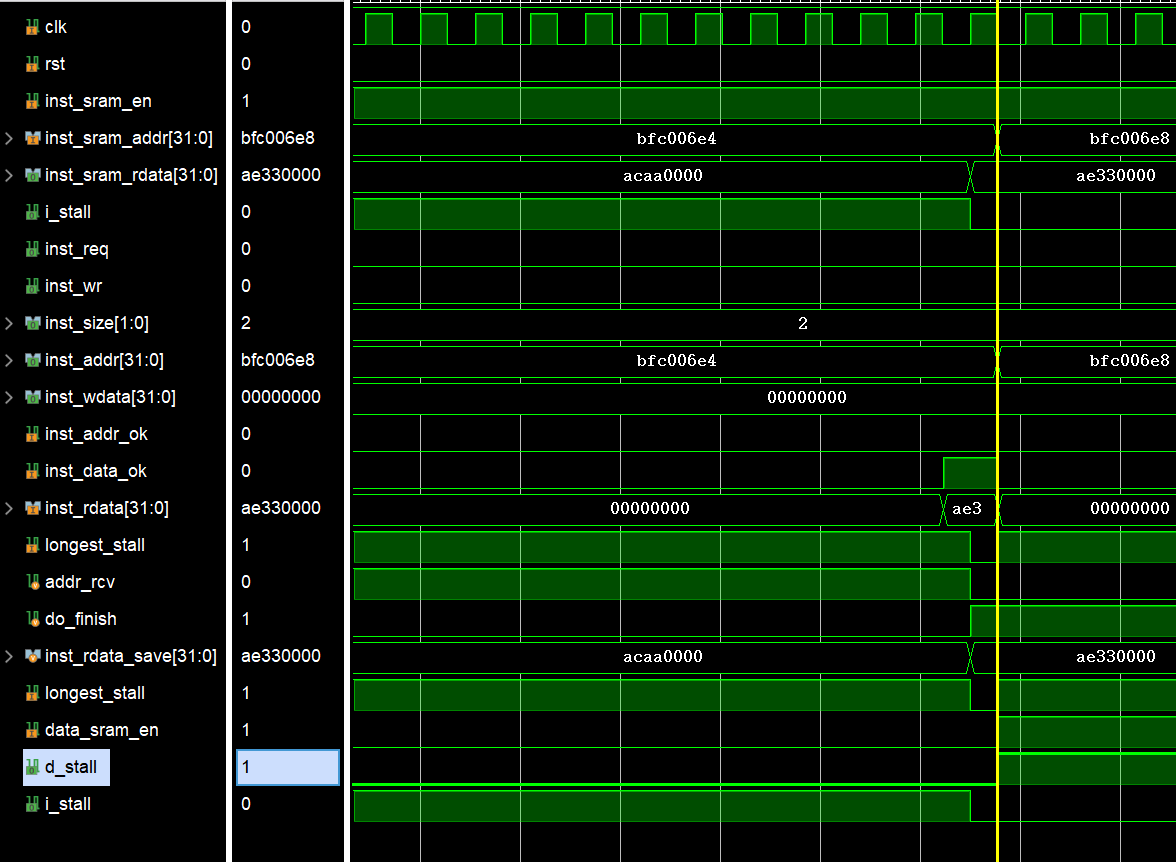
首先查看了类sram的仿真图，发现do\_finish和longest\_stall变量在后续一直为高电平，而正确应该是在短暂的高电平后，就处理下一条指令请求而变为低电平。



然后，查看决定这两个变量的变量。通过和之前longest\_stall变量置0时的对比发现是由于数据请求而导致的d\_stall和data\_sram\_en高电平的到来使得longest\_stall刚变为0还没来得及req就又变为高电平陷入死循环了。

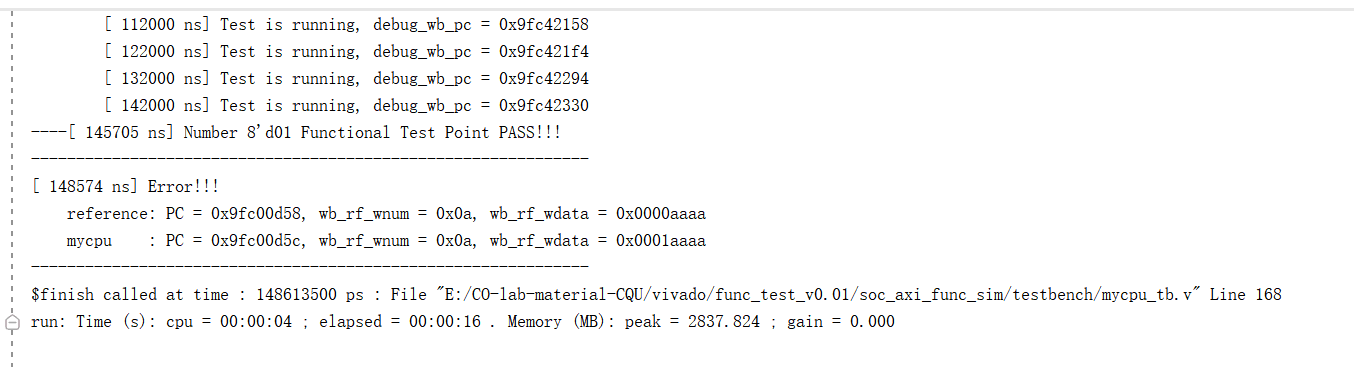


我们可以看出为d\_stall和data\_sram\_en为clk的下降沿触发，是因为我之前尝试解决上个问题时，将这两个类sram接口改成下降沿触发忘记改回来了……导致缺失了半个周期。



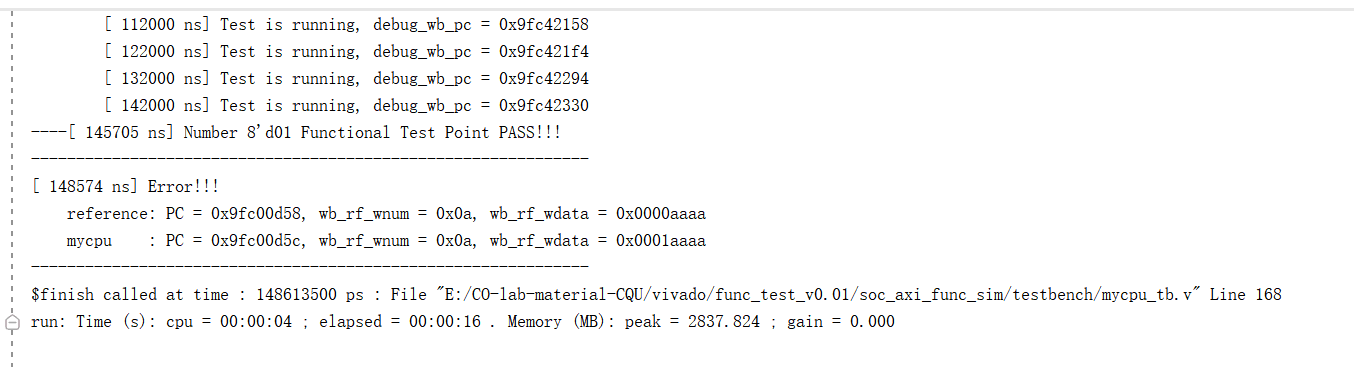
1. 修正效果

改完之后不再循环，进入新的错误。



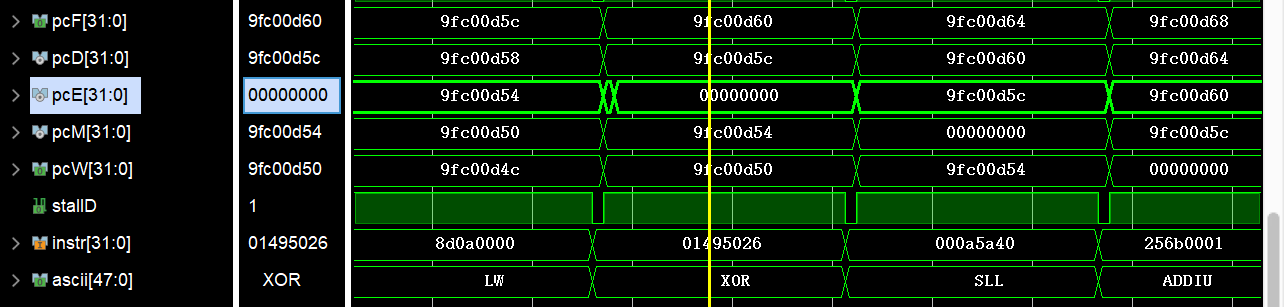
12、错误12

(1) 错误现象



(2) 分析定位过程

首先通过查看测试文件得知此条指令（pc=9fc00d58）是lw指令，然后通过观察仿真图，得知是因为stall和flush使用的不合适导致此条指令没有被stall就flush了，使其丢失。

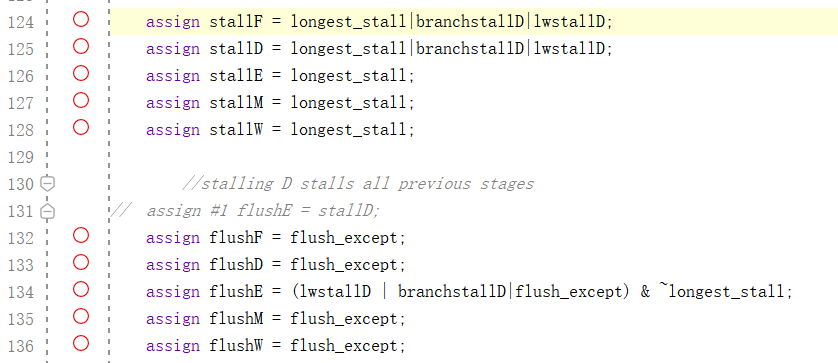


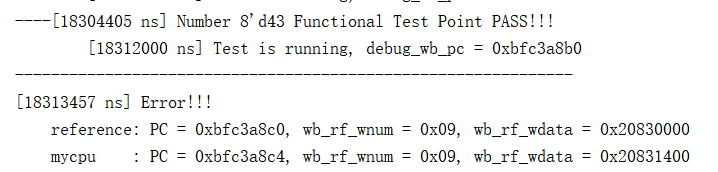
1. 错误原因

错误原因是在hazard模块中的flushE和stallD的赋值有问题，需要进行改变，通过观察和以前没加axi和类sram的仿真图对比，对stallD和flushE变量更新了赋值条件，这里就不再细致赘述怎么来的了，找bug太疲惫了。



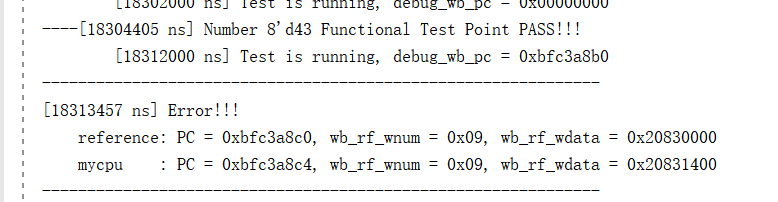
1. 修正效果





13、错误13

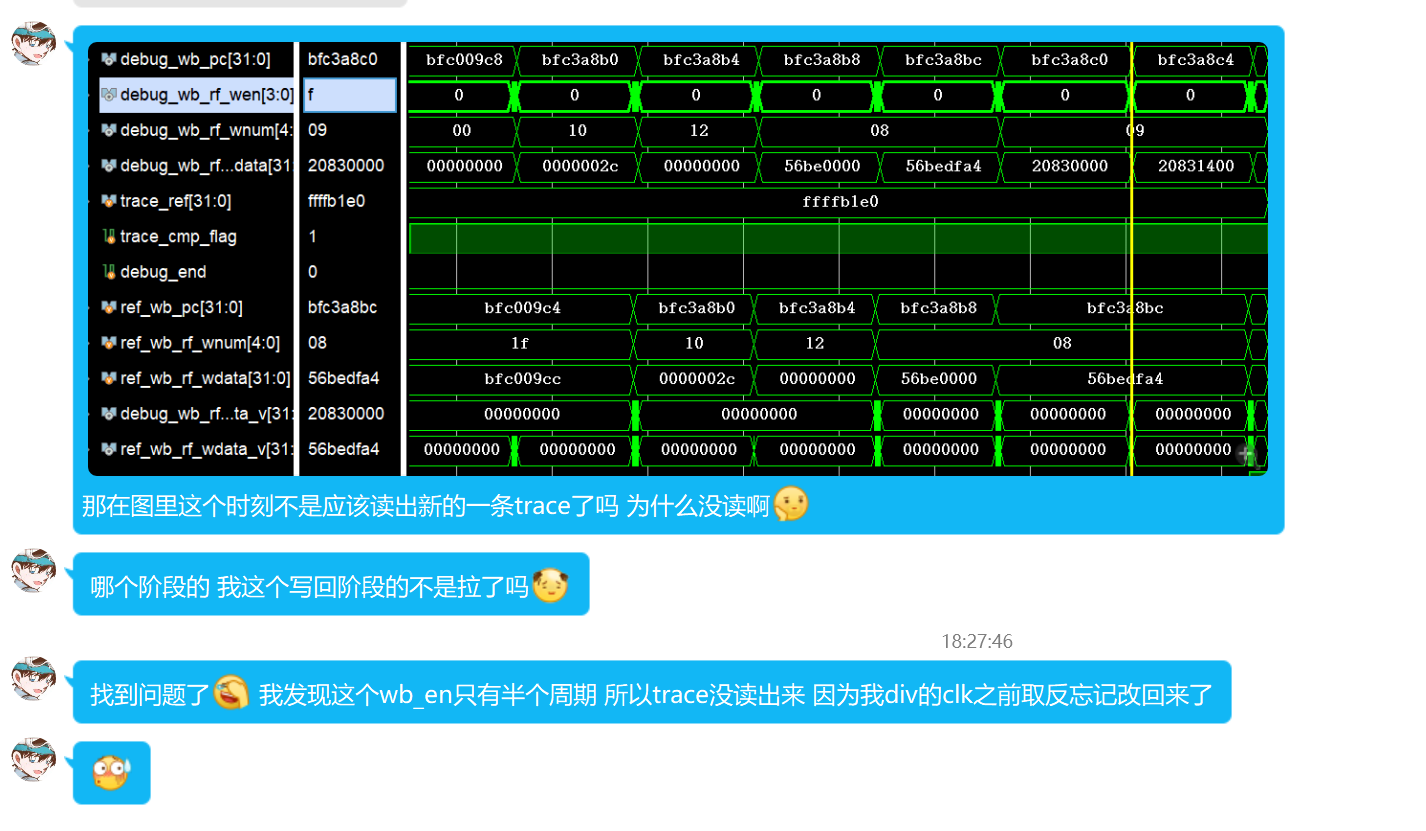
(1) 错误现象



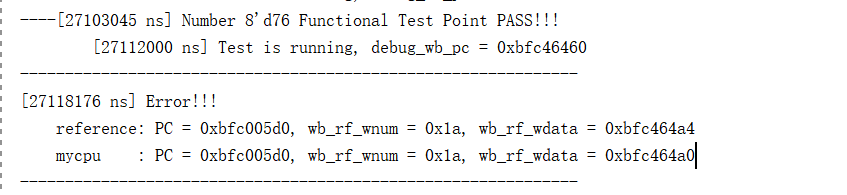
(2) 分析定位过程

通过查看测试文件，得知此模块是对除法指令进行测试，因此对quick\_div模块进行了排查。

(3) 错误原因

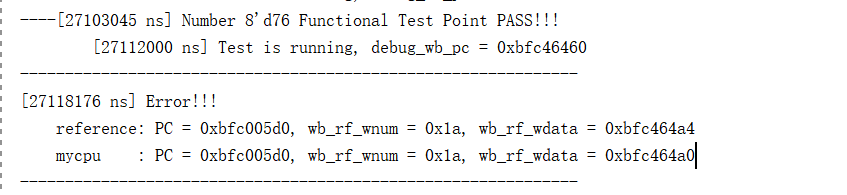


(4) 修正效果



14、错误14

(1) 错误现象



(2) 分析定位过程

通过查看仿真图和测试文件得知此条指令为软件中断指令，通过查看软件中断与其它中断发生时各信号的变化，发现了错误原因。

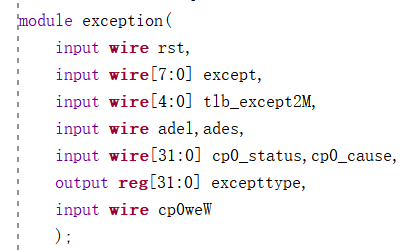
(3) 错误原因

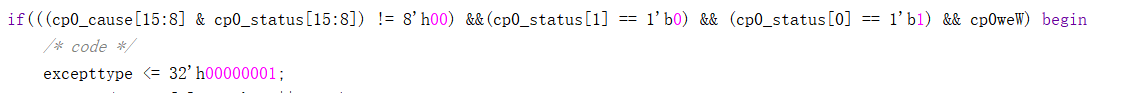
接了axi接口之后 其它异常是通过每个阶段传的except变量来判断的是否发生异常 所以跟原来一样不会有什么影响，但是软件中断我原来是在cp0中上升沿触发写cause，但现在一个阶段不止一个上升沿了，也就是写cause的指令还在E阶段的第二个上升沿时，cause就已经被写好了 而exception模块又是个组合逻辑 然后就导致flush也提前了 直接就把后面的指令给flush了。

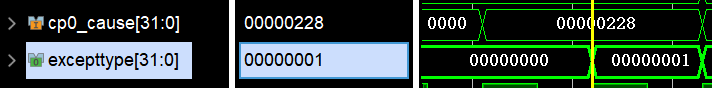
(4) 修正效果

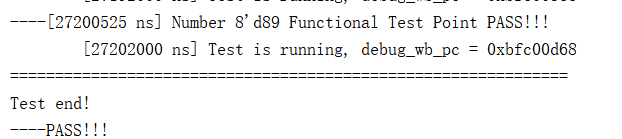
于是我新定义了一个变量cp0weW,即将在M阶段写cp0寄存器的写使能信号传到W阶段，并传入exception模块中，与except一样在W阶段的一开始再更新写入cause的值。











四、设计结果

（一）设计交付物说明

-硬件综合设计报告

-硬件综合设计报告.docx

-mycpu

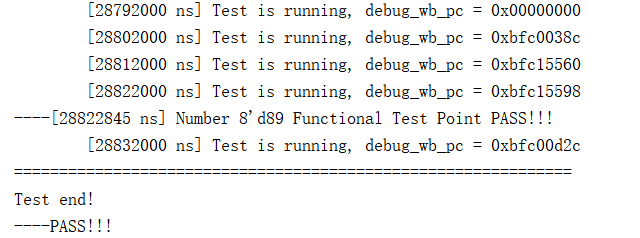
-项目源代码文件（.v文件）

直接使用verilator[6]对源代码进行编译后，即可测试，理论上可以上板。

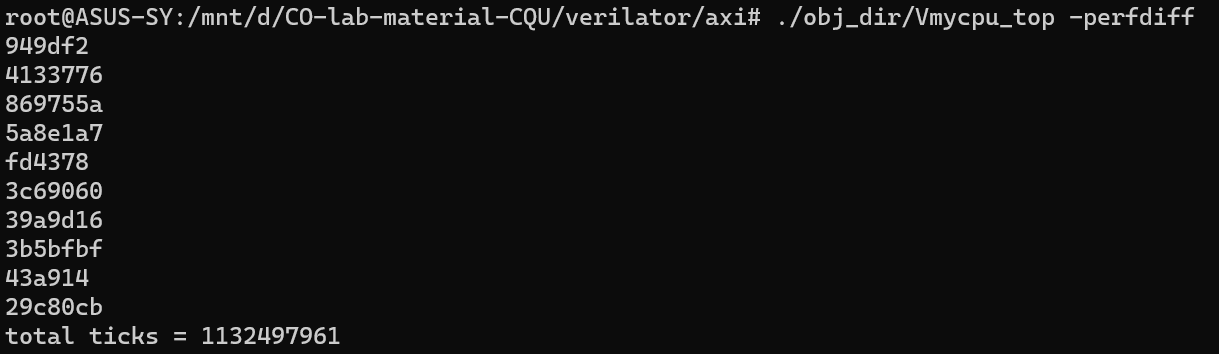
（二）设计演示结果

由于连接SOC后的性能测试通过，故前面52条的6类独立测试也能通过，且在完成52条指令时，已经对各类指令独立测试过（详见 三、错误记录），故在本部分不再一一展示，仅展示功能测试与性能测试结果。

**添加AXI接口后的功能测试：**



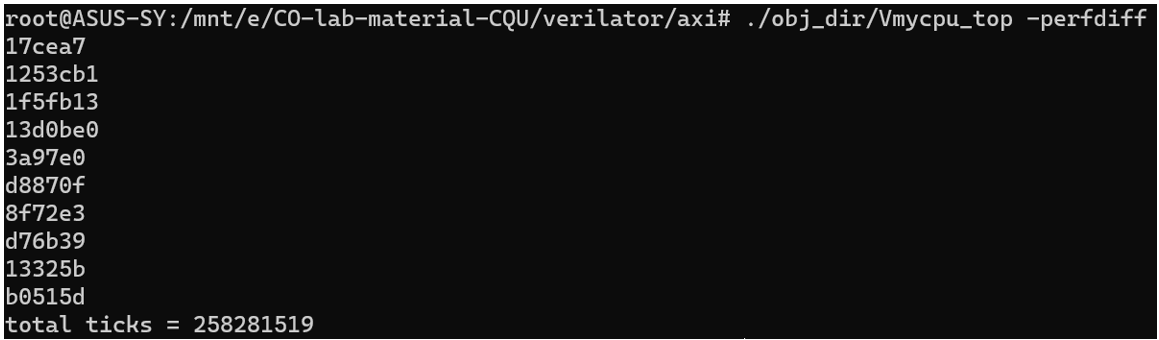
**添加AXI接口后的性能测试：**



**AXI接口+指令Cache功能测试：**

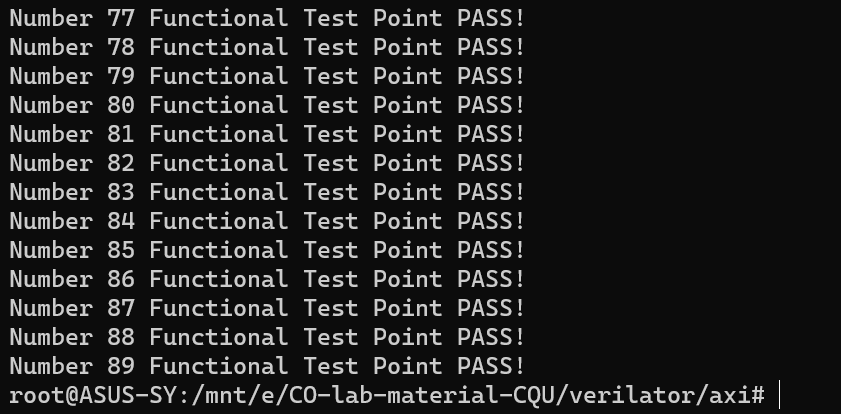
****

**AXI接口+指令Cache性能测试：**

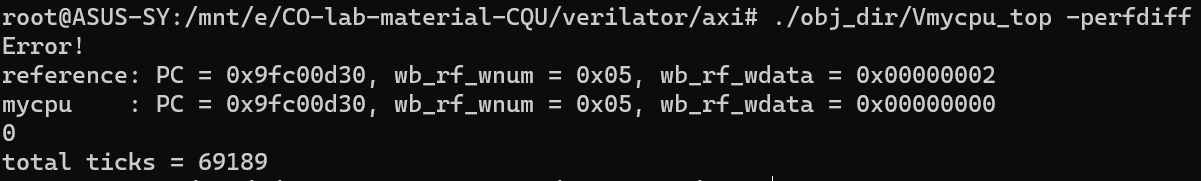
****

可以看出，指令Cache对性能的提升是数量级的

**AXI接口+指令Cache+数据Cache功能测试：**



**AXI接口+指令Cache+数据Cache性能测试：**



由于时间原因，并未对添加数据Cache后报错进行debug，主要思路是与指令Cache一样，将CPU发送的请求作为输入传送到数据Cache中，若命中，直接返回给CPU所需的数据，否则，Cache再向内存发送请求，将返回的数据反馈给CPU。唯一不同的是，数据Cache有一个no\_dcache信号，来决定该请求是否通过Cache，防止一些MMIO、DMA以及ID一致性（例如写入指令后执行的场景）的异常（文献[1]）。因此，推测可能是在处理这方面时出现错误。因此，本项目还有着很大的提升空间。此外，Cache还可以从写直达机制变为写回机制、改变块大小等。

**扩展内容---快速除法器：**

以任意一条除法指令为例，使用原始的除法器必定是需要36个时钟周期的，而在应用快速除法器后，通过flag标志的移动，我们可以看出，此条除法指令只用了8个时钟周期便计算出来结果。



五、参考设计说明

1. hilo\_reg.v

hilo寄存器的参考代码，来自项目资料的ref\_code。

2. cp0\_reg.v

cp0模块的参考代码，来自项目资料的ref\_code。

3. cpu\_axi\_interface.v

AXI接口的参考代码，来自项目资料的ref\_code。

4. i\_cache.v & d\_cache.v

Cache的参考代码，来自项目资料的ref\_code。

5. mmu.v

mmu的参考代码，来自项目资料的ref\_code。

6. quick\_div.v

一开始实现时遇到困难，参考文献[8]和学长的部分代码后自己重新实现

7. inst\_sram\_like &data\_sram\_like

参考计算机体系结构实验二Cache提供的代码后自己实现

8. 参考文献[3]中部分代码思路

六、参考文献

1. 重庆大学硬件综合设计实验文档 [EB/OL]. [(ccslab.cn)](http://co.ccslab.cn/).
2. A03\_“系统能力培养大赛”MIPS指令系统规范\_v1.01.
3. CQU硬件综合设计录播 2019-12-11 [EB/OL]. [2019-12-12]

<https://www.bilibili.com/video/BV1XJ411k7kR?p=1&vd_source=40110d1722fdce829e736c2a08fb55ae>.

1. CQU硬件综合设计2020讲解 [EB/OL]. [2020-12-28].

<https://www.bilibili.com/video/BV1TA411x7T7/?p=3&spm_id_from=333.1007.top_right_bar_window_history.content.click&vd_source=40110d1722fdce829e736c2a08fb55ae>.

1. 雷思磊. 自己动手写CPU. 电子工业出版社. 2014.
2. 2022重庆大学硬件综合设计-MIPS组资源包的使用

https://www.bilibili.com/video/BV19G4y1E7jj/?spm\_id\_from=333.788.top\_right\_bar\_window\_history.content.click&vd\_source=40110d1722fdce829e736c2a08fb55ae.

1. A12\_类SRAM接口说明.
2. 《计算机组成与设计-硬件软件接口》，David A.Patterson、John L.Hennessy编，康继昌、 樊晓桠等译，机械工业出版社，2012年1月出版.