**LAB6报告**

学号： 2021K8009929010

姓名： 贾城昊

学号： 2021K8009929016

姓名： 李金明

学号： 2021K8009929007

姓名： 牛浩宇

箱子号： 13

1. 实验任务

Lab6的整体要求是使我们理解异常和中断的软硬件协调处理机制，理解精确异常的概念和处理方法，并掌握在流水线CPU中添加异常和中断支持的方法。

实验12要求所设计的CPU能够支持简单的异常和中断处理。需要增加csr控制状态寄存器CRMD、PRMD、ESTAT、ERA、EENTRY和SAVE0~3；实现对csr的三个读写命令csrrd、csrwr和csrxchg；实现中断返回指令ertn和系统调用指令syscall。

实验13要求在实验12的基础上增加对取指地址错（ADEF）、地址非对齐（ALE）、断点（BRK）和指令不存在（INE）这几种异常处理的支持；增加对2个软中断、8个硬件中断和定时器中断的支持；在csr中增加控制状态寄存器ECFG、BADV、TID、TCFG、TVAL和TICLR。

1. 实验设计
   1. 总体设计思路

对CPU发生中断和异常时进行状态控制的信息存放在csr寄存器中，这部分内容由硬件电路自动完成，不属于用户可见的软件上下文。为实现这部分电路，代码中新增一个csr模块，其内包含了所有的csr寄存器以及对csr寄存器进行读写的控制电路，并例化在WB流水级中。CPU在WB流水级收集到本条指令在所有流水级中发生的异常或中断信息后，对csr模块中的控制状态寄存器进行集中读写。

为了实现精确异常，首先需要保证异常指令前面所有的指令全部执行完毕，因此在WB级之后进行中断和异常处理，这样前面的指令已经完成了所有的操作。此外，还需要保证异常指令后面“错取”的指令不能改变机器的状态，所以如果WB级发现本级指令出现异常时，必须将前面各个流水级清空，即通过向前面各流水级传递一个flush信号来“等效”把各级的valid标志置0（因为valid是一个寄存器，下一拍值才能改变，所以这里的“等效”指的是与valid信号相关的判断逻辑改为valid & ~flush，这样看起来就像当拍将valid清0了）。

需要注意的是，如果“错取”的指令包含store类型指令，那么该指令在清空各流水级之前已经将数据写入到内存当中，导致机器的状态出现了更改。为了解决这一问题，WB级和MEM级都需要把当前所执行的指令是否发生异常的判断信号传递到EXE级，如果EXE级发现后面两个流水级中存在出现异常的指令，便将数据ram的使能置0，防止误写数据破坏了机器的状态。

除此之外，如果“错取”的指令包含div类型指令，那么该指令在清空各流水级之前已经将。。。。（补充）

1. Exp 12
2. 增加控制状态寄存器CRMD、PRMD、ESTAT、ERA、EENTRY和SAVE0~3

在本实验中，只有触发例外或中断，以及遇到csrrd、csrwr和csrxchg指令才会对控制状态寄存器中所存数据进行读取和修改，因此在csr模块中设计控制状态寄存器的读写逻辑可以从这两方面进行考虑。对于要增加的状态寄存器，参考指令集手册和讲义，各控制状态寄存器对应的读写条件如下所述：

CRMD（当前模式信息）：该寄存器的PLV域刻画了CPU当前所处的特权级。指令手册定义在复位时PLV域被置为0，表示最高特权级，当触发例外时该域也被置为0，从而保证trap后机器会处于最高特权级。该寄存器的IE域表示全局中断使能，发生例外时该域被置为0，从而让trap后屏蔽中断。此外，执行ertn指令从中断处理程序返回时，需要将PRMD寄存器中的PPLV和PIE域分别写入PLV和IE域。因此例外信号和ertn指令信号需要作为csr模块的输入。

PRMD（例外前模式信息）：触发例外时，PRMD寄存器中的PPLV和PIE域被赋值成CRMD寄存器的PLV和IE域中的值。

ESTAT（例外状态）：在本实验中只考虑该寄存器IS域的后两位需要被csr读写指令进行读取和修改。此外，在发生例外时，该寄存器的ecode和esubcode域会被赋值成此异常的类型代号，因此异常的类型代号需要作为csr模块的输入。

ERA（例外返回地址）：当触发例外时，将触发例外的指令的PC值，即当前WB级的PC值保存在该寄存器中，因此wb\_pc需要作为csr模块的输入。

EENTRY（例外入口地址）：该寄存器的后6位恒为0，前26位可被csr读写指令进行读取和修改。

SAVE0~3（数据保存）：数据保存寄存器可以存放通用寄存器中的数据，只能由csr读写指令进行读取和修改。

1. 增加csrrd、csrwr和csrxchg指令

csrrd、csrwr和csrxchg指令用于软件访问csr模块，需要通过num来指定被操作的控制状态寄存器。其中，num可以从指令类型（如ertn指令）或者指令本身来定义。由于csrwr和csrxchg指令需要用到寄存器rj和rd中的数据，因此在取数时需要考虑数据相关的冲突。

此外，这三条指令都需要写rd寄存器，会与后续指令产生数据相关，流水线需要阻塞。这三条csr读写指令的写数据只能在WB级访问csr寄存器后才能获得，所以后续指令如果与这些指令产生了数据相关，需要阻塞一拍或者两拍，才能从WB级前递的数据中读出需要的数据。阻塞逻辑的更改：在EXE级和MEM级增加前递到ID级的res\_from\_csr信号，表示在EXE或MEM级中的指令需要对csr进行读写。如果ID级发现上述两个res\_from\_csr信号只要有一个为1，就需要将自己的ready\_go置0，从而实现阻塞，如下所示：

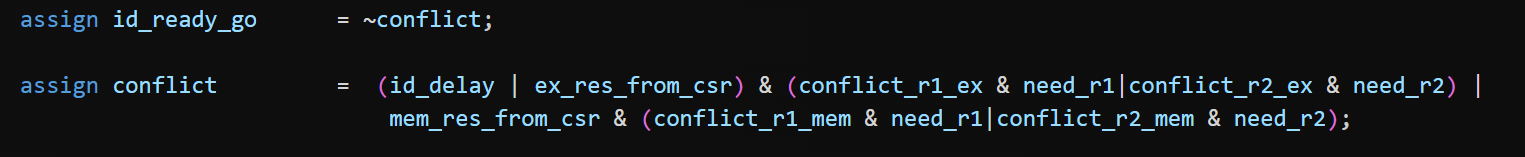


图1 exp12对ID的阻塞逻辑

1. 增加ertn指令和syscall指令

如果当前执行的指令是ertn指令，那么WB流水级会从csr模块中读出ERA的值，并传递给IF流水级更新nextpc，在下一时钟周期跳转到返回地址继续执行。同时，WB流水级要拉高向前四个流水级发出的flush信号，从而清空流水线。

如果当前执行的指令是syscall指令，ID级在进行译码时会触发中断并产生例外编码，中断信号顺着流水线传递到WB级时，WB级会把产生例外的PC存入控制状态寄存器中的ERA寄存器作为中断处理完毕后的返回地址，同时根据例外编码eecode和esubcode从EENTRY寄存器中读出中断处理程序的入口地址，并传递给IF级更新nextpc，在下一时钟周期跳转到中断处理程序开始执行。此外，WB级也要拉高flush信号来清空流水线，从而实现精确异常。

清空流水线的具体操作是根据WB级传递的flush信号，将每一级的allow\_in信号置1，同时将每一流水级向下一级传递的xx\_to\_yy\_valid信号置0(即“等效”把各级的valid标志置0）。以EXE级为例，代码实现如下：

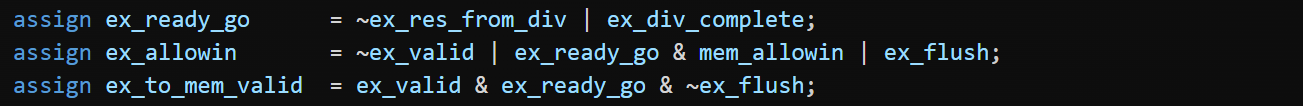


图2 exp12对流水线的清空操作

2、Exp 13

（1）增加控制状态寄存器ECFG、BADV、TID、TCFG、TVAL、TICLR，修改ESTAT

在本实验中，需要修改寄存器的值的情况与exp12中类似，包括触发例外或中断，以及执行csrrd、csrwr和csrxchg指令。结合上述条件，并参考指令集手册和讲义，得到新增的控制状态寄存器的读写条件如下：

ECFG（例外配置）：ECFG的第0~9位和第11~12位均为局部中断使能位LIE，每位控制一个中断源，中断源与ESTAT的对应位一一对应；其余位为保留位。对于LIE域，其读写由CSR指令实现，因此只需对此添加相关支持；其余保留位不允许写入，读出的值为0，因此直接用0填充即可。

BADV（出错虚地址）：本实验中的BADV共32位，用于存储触发地址相关例外时的虚地址。在本实验中相关的例外只有ADEF和ALE。如果是ADEF，则把wb\_pc写入BADV；如果是ALE，则把访存地址传到wb级并写入BADV。

TID（定时器编号）：TID寄存器存储定时器编号，需要支持CSR指令读写。本实验中可由rdcntid指令读出。

TCFG（定时器配置）：软件配置定时器的接口，从低到高依次为定时器使能位、循环模式控制位、初始值、只读位。需要支持CSR指令的读写，并用对应位控制定时器TVAL。

TVAL（定时器值）：TVAL存储定时器的值，初值为TCFG中的InitVal域，在TCFG的En域为1时递减。减到0后，如果TCFG的Periodic域为1，则从初始值开始重新倒计时，否则保持不变。

TICLR（定时中断清除）：从该寄存器中读出的值始终为0，通过对该寄存器最低位写1的动作来清除时钟中断标记。因此只需要用一个1bit信号csr\_ticlr\_clr来实现该寄存器，并始终赋值为0，如果要读则在高位拼接上31bit的0。至于清除时钟中断，只需要捕捉对寄存器写1的动作，不需要真的写入1。

另外，exp12中只考虑了ESTAT的最低2位，本实验中需要将ecode和subcode写入对应域，并给硬件中断、软件中断、时钟中断的中断状态位赋值，其中时钟中断状态位在倒计时到0时拉高。

（2）增加异常支持ADEF、ALE、BRK、INE

ADEF：PC值后两位非0时发生异常。在pre-IF级对异常进行判定，如果存在异常则把adef\_excep拉高，并逐级传递下去，直到在WB级拉高flush来刷掉前面的流水级，并跳转到异常处理程序入口，这部分操作可以复用exp12中的通路。此外，还要在WB级把出错的PC存入BADV，并把ecode和subcode存入ESTAT的对应域。

ALE：访存地址未对齐时发生ALE异常。ALE在EXE级检测，如果发现异常则把ale\_excep拉高，并把出错的访存地址传到WB级写入BADV，其余的异常处理操作和ADEF使用相同的通路。

BRK：如果在译码级发现当前的指令是break，则拉高inst\_break，并逐级传递到WB级，在WB级把ecode和subcode存入ESTAT的对应域，然后通过拉高flush来刷新流水级，并跳转到异常处理程序。break指令是新增的指令，需要在ID级添加对应的译码逻辑。

INE：如果在译码级发现指令不属于任何一条已经实现的指令，则拉高ine\_excep，其余操作同BRK，包括在WB级保存ecode、subcode，刷新流水级，跳转到异常处理程序等。

（3）增加中断支持

本实验需要支持硬件中断、软件中断与时钟中断。软件中断由软件控制，通过对ECFG的使能位写入1来实现，只需支持CSR指令读写。时钟中断在TVAL倒计时到0且TCFG使能拉高时触发，中断发生时将ESTAT中对应的状态位拉高，通过对TICLR最低位位写1的动作来将该状态位清0。硬件中断则是通过hw\_int\_in信号传入csr模块，把ESTAT的对应IS域值拉高即可。

如果发生以上任何一种中断，且没有被屏蔽中断（CRMD的IE域为1），将csr模块的has\_int信号拉高，传到ID级处理。只要发生中断或异常，对应流水级的对应excep信号就会拉高，并逐级传递到WB级处理。

（4）新增计时器相关指令rdcntvl.w、rdcntvh.w、rdcntid

在新增的三天计时器相关指令中，rdcntvl.w和rdcntvh.w读取一个64位计数器的低32位和高32位。这个计数器**并不是时钟中断中的倒计时计数器，而是另一个正向计数的计数器**。本组成员将这个计数器设置在EXE级，其初值为0，每拍加一。指令在EXE级得到结果后在WB级写回寄存器。如果发生数据相关，可以将EXE级的结果前递到ID级，这部分通过复用已有的数据通路实现。

rdcntid读取TID控制状态寄存器中的内容，可以复用exp12中三条CSR指令的通路。由于它在写回级才能拿到从TID中读出的数据，因此一旦它后面的指令和它发生数据相关，需要阻塞后面的指令，直到它在写回级读数据。这部分也通过复用已有CSR指令的数据通路实现。

* 1. 重要模块1设计：CSR模块
     1. 工作原理

在写回级处理异常，将信号传入CSR模块。如果发生异常，就把异常码和出错的地址写入对应寄存器，并把异常处理程序的入口地址传给pre-IIF级。如果写回级的是CSR读写指令，就对csr\_num指明的寄存器进行对应操作。如果是etern指令，就把保存的地址传给pre-IF级。此外，还要在发生中断时拉高has\_int信号。。

* + 1. 接口定义

表1 CSR模块接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clock | IN | 1 | 时钟输入 |
| resetn | IN | 1 | 复位信号 |
| csr\_num | IN | 14 | 控制状态寄存器读写地址 |
| csr\_we | IN | 1 | 控制状态寄存器写使能 |
| csr\_wmask | IN | 32 | 控制状态寄存器写掩码 |
| csr\_wvalue | IN | 32 | 控制状态寄存器写数据 |
| ertn\_flush | IN | 1 | ertn指令刷新信号 |
| wb\_ex | IN | 1 | 例外发生信号 |
| wb\_ecode | IN | 6 | 例外发生ecode代码 |
| wb\_esubcode | IN | 9 | 例外发生esubcode代码 |
| wb\_pc | IN | 32 | WB级PC值，用于记录例外发生的PC |
| csr\_rvalue | OUT | 32 | 控制状态寄存器读数据 |
| ex\_entry | OUT | 32 | 例外跳转入口地址 |
| has\_int | OUT | 1 | 发生中断信号 |
| ipi\_int\_in | IN | 1 | 核间中断输入 |
| coreid\_in | IN | 32 | 核编号，也是TID初值 |
| hw\_int\_in | IN | 8 | 硬件中断输入 |
| wb\_vaddr | IN | 32 | 出错地址 |

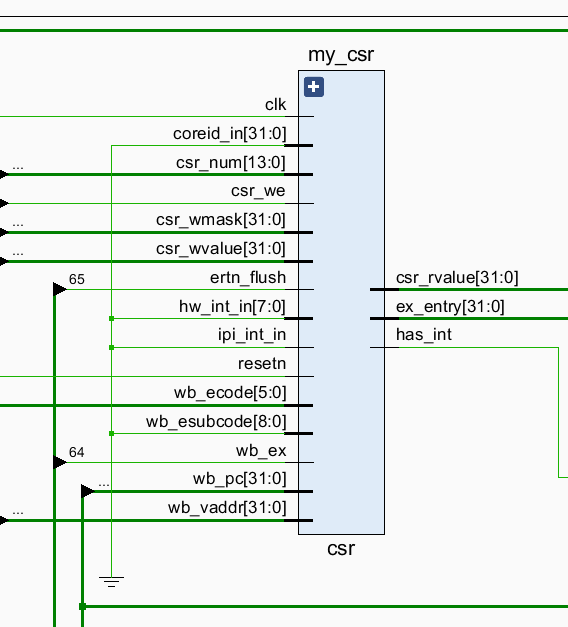


图3 CSR模块接口

* + 1. 功能描述

csr模块包含控制状态寄存器CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0~3、ECFG、BADV、TID、TCFG、TVAL、TICLR。每个模块的每个域分别用一个寄存器实现，在读出时再拼接为完整的寄存器，通过多路选择器赋值给csr\_rvalue。每个域在wmask中对应的bit，以及每个寄存器的csr\_num，都用宏在csr.h中定义。

各寄存器的含义与功能以及其更新逻辑已在总体设计思路中阐明，在此不再赘述了。

* 1. 重要模块2设计：除法器
     1. 工作原理

本实验中使用的是1位恢复余数绝对值迭代触发器。除法器工作时，首先根据被除数和除数确定商和余数的符号，并计算被除数和除数的绝对值，然后对绝对值进行除法运算，迭代运算得到商和余数的绝对值，最后根据被除数和除数原先的符号确定商和余数最终的符号和值。在这个除法器中，若想得到无符号除法的结果，只需要在计算被除数和除数的绝对值时保持值不变即可。

* + 1. 接口定义

表2 除法器接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| div\_clk | IN | 1 | 除法器模块时钟信号 |
| resetn | IN | 1 | 复位信号 |
| div | IN | 1 | 除法运算的命令 |
| div\_signed | IN | 1 | 控制有符号除法和无符号除法 |
| x | IN | 32 | 被除数 |
| y | IN | 32 | 除数 |
| s | OUT | 32 | 除法结果，商 |
| r | OUT | 32 | 除法结果，余数 |
| complete | OUT | 1 | 除法完成信号 |

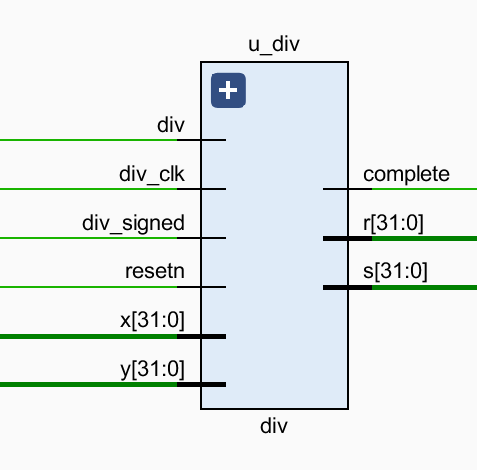


图6 除法器接口

* + 1. 功能描述

当div信号拉高时，根据EXE模块传递的操作数进行除法运算。当运算结束后，拉高complete信号，此时r和s的值分别是除法运算得到的余数和商。

* 1. 重要模块3设计：ID流水级
     1. 工作原理

为新增加的指令添加译码信号，将它们传给EXE阶段。修改与alu、立即数有关的信号，实现算术逻辑运算指令。对于转移指令则需要判断是否进行跳转，并计算跳转地址。

本次实验中采用前递的方法，故当遇到访存指令和乘法指令且与紧跟的吓一跳指令有数据相关时，也需要阻塞一周期。阻塞是通过控制id\_ready\_go 的信号实现的，其拉低即可确保流水线被阻塞。

* + 1. 接口定义

表3 ID流水级接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| resetn | IN | 1 | 复位信号 |
| ex\_allowin | IN | 1 | EXE流水级允许ID流水级传入数据 |
| id\_allowin | OUT | 1 | ID流水级允许IF流水级传入数据 |
| if\_to\_id\_valid | IN | 1 | 标记IF流水级传入ID流水级的数据是否有效 |
| if\_to\_id\_wire | IN | 64 | IF流水级传给ID流水级的数据 |
| id\_to\_ex\_valid | OUT | 1 | 标记ID流水级传入EXE流水级的数据是否有效 |
| id\_to\_ex\_wire | OUT | 160 | ID流水级传入EXE流水级的数据 |
| br\_taken | OUT | 32 | 传给IF流水级的跳转信号 |
| br\_target | OUT | 1 | 传给IF流水级的跳转地址 |
| wb\_rf\_zip | IN | 38 | WB流水级向ID流水级传递的regfile的写回信息和前递数据 |
| ex\_rf\_zip | IN | 39 | EXE流水级前递到ID流水级的数据 |
| mem\_rf\_zip | IN | 38 | MEM流水级前递到ID流水级的数据 |



图7 ID流水级接口

* + 1. 功能描述

对于exp10增加的算术逻辑运算类指令，基本上都可以复用现有的数据通路，从寄存器中读出数据，根据指令类型对立即数进行扩展，计算出两个操作数后送入EXE阶段，使用alu算出结果。andi、ori、xori的立即数是由12位立即数无符号扩展得到的，之前没有这类立即数的数据通路，需要添加对应的信号和逻辑。

对于exp11增加的跳转指令（blt、bltu、bge、bgeu），可以复用之前的br\_taken与br\_target信号，只需在br\_taken的赋值逻辑中增加对应的条件判断的逻辑，即判断rj寄存器中的值在有无符号的两种情况下是否小于rd寄存器中的值。逻辑如下：

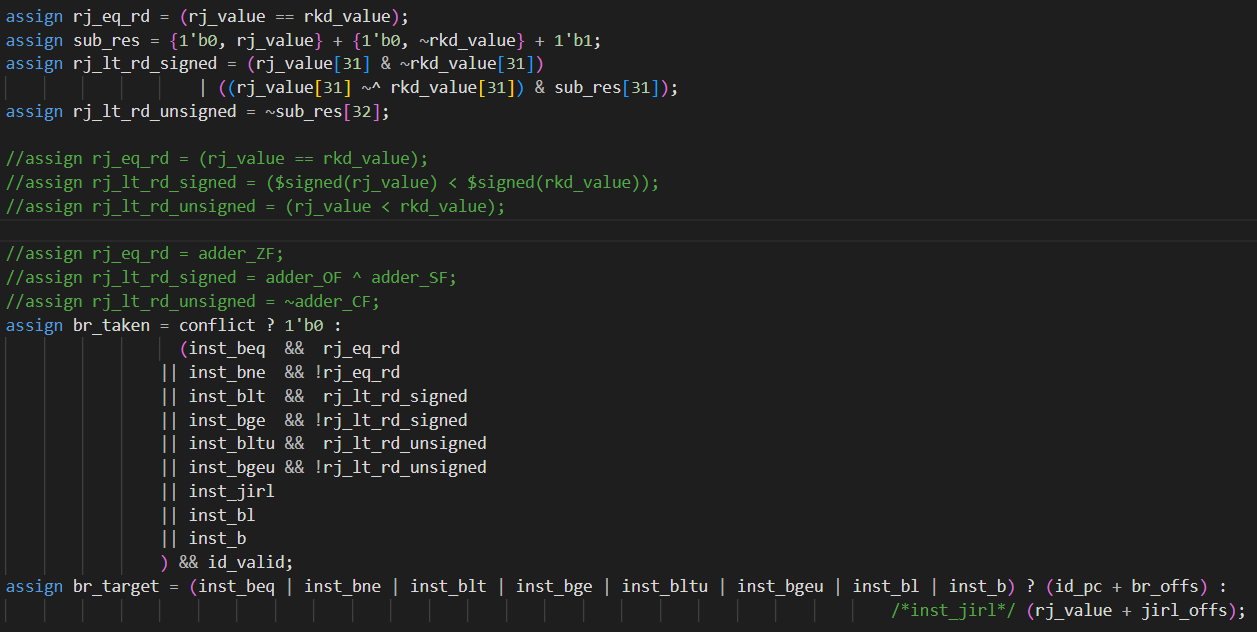


图8 跳转指令判断逻辑

共尝试了三种判断逻辑，包括直接使用小于号、连接加法器模块使用Flag值、以及不连接单独的加法器模块，而是直接进行运算并判断。最终发现第三种赋值方法时序性能最好，故采用。

exp11中还增加了几种load和store指令，内存读写不再固定为32位。 改动为不再在ID流水级产生res\_from\_mem、mem\_we以及data\_sram\_wdata信号，而是将5种load指令信号与3种store指令信号向后传递到对应流水级再进行处理。

* 1. 重要模块4设计：EXE流水级
     1. 工作原理

EXE阶段从ID阶段取得有效操作数和操作码后，进行算术逻辑运算，以及在EXE流水级向数据ram发送读写请求，读写地址以及写数据。在Exp10中，EXE流水级需要新添对乘法器和除法器的例化，对于乘法器，EXE流水级只需要把相关信号传递给乘法器，而对于除法器，若为除法指令，则需要等待其算出结果，然后选择结果传递给下一流水级。

* + 1. 接口定义

表4 EXE流水级接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| resetn | IN | 1 | 复位信号 |
| mem\_allowin | IN | 1 | MEM模块允许EXE模块传入数据 |
| ex\_allowin | OUT | 1 | EXE模块允许ID模块传入数据 |
| id\_to\_ex\_valid | IN | 1 | 标记ID模块传入EXE模块的数据是否有效 |
| id\_to\_ex\_wire | IN | 160 | ID模块传入EXE模块的数据 |
| ex\_to\_mem\_valid | OUT | 1 | 标记EXE模块传入MEM模块的数据是否有效 |
| ex\_to\_mem\_wire | OUT | 110 | EXE模块传入MEM模块的数据 |
| ex\_rf\_zip | OUT | 39 | EXE模块前递到ID模块的数据 |
| data\_sram\_en | OUT | 1 | 数据ram片选信号 |
| data\_sram\_we | OUT | 4 | 数据ram写使能 |
| data\_sram\_addr | OUT | 32 | 数据ram地址信号 |
| data\_sram\_wdata | OUT | 32 | 数据ram写数据 |

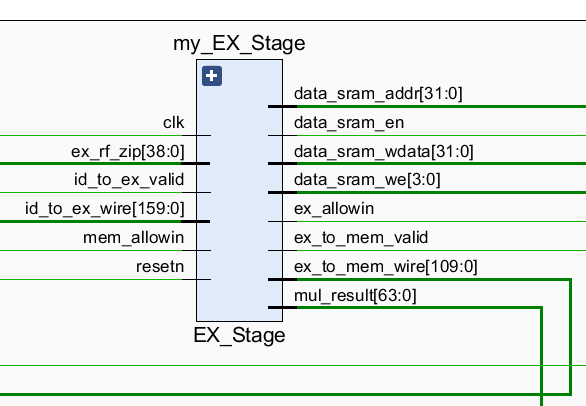


图9 EXE流水级接口

* + 1. 功能描述

结合从ID级传递来的控制信号和数据进行指令的执行。新增对于乘法指令和除法指令的处理：例化乘法器和除法器模块，并传递相应信号，乘法指令的阻塞处理在ID进行，除法指令的阻塞则需要在EXE进行，若为除法指令，需要等到除法器模块发送complete信号后，才能将ex\_ready\_go拉高，同时传递给MEM的数据也需要进行选择。具体如下所示：

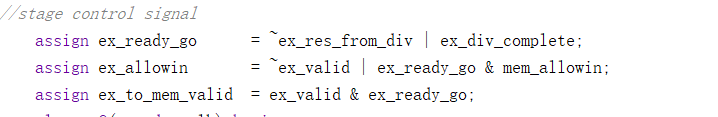


图 10 EXE流水级对除法指令的阻塞

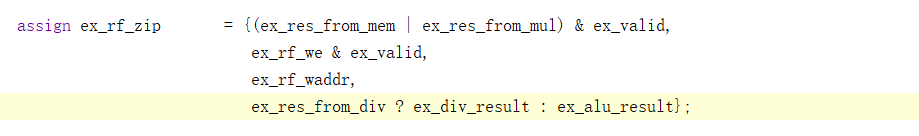


图 11 EXE流水级对除法指令的处理

对于访存：根据alu计算得到的地址第二位与内存写的位数来对mem\_we进行赋值：

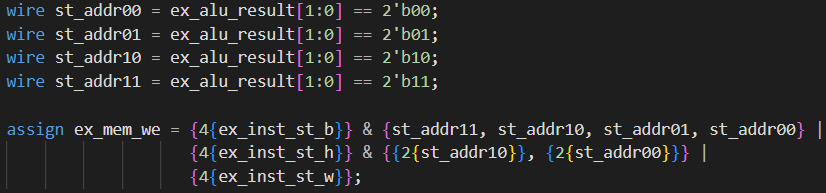


图12 mem\_we赋值逻辑

32位的wdata可以将8位数据重复4次，16位数据重复两次，再由mem\_we进行选择。

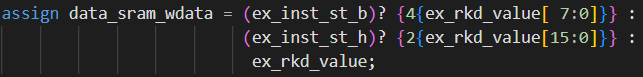


图13 数据RAM写数据赋值逻辑

* 1. 重要模块5设计：MEM流水级
     1. 工作原理

接收数据ram返回的读出数据，根据控制信号生成最终的写回数据。

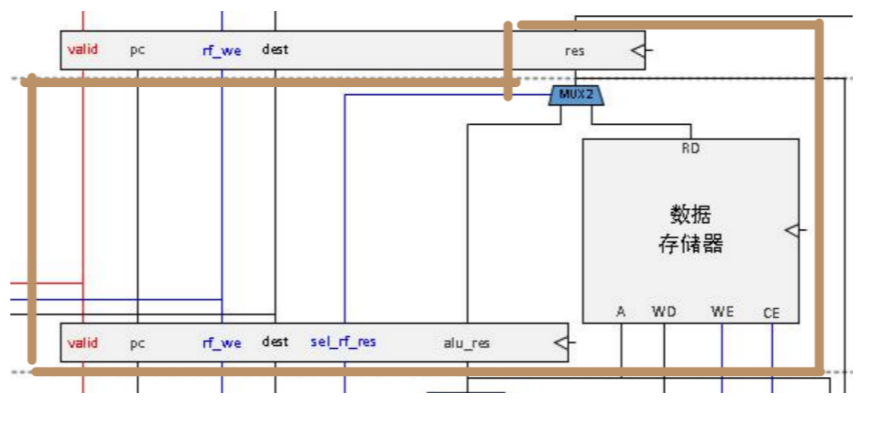


图14 MEM流水级结构示意图

* + 1. 接口定义

表5 MEM流水级接口定义

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| clk | IN | 1 | 时钟信号 |
| resetn | IN | 1 | 复位信号 |
| wb\_allowin | IN | 1 | WB模块允许MEM模块传入数据 |
| mem\_allowin | OUT | 1 | MEM模块允许EXE模块传入数据 |
| ex\_to\_mem\_valid | IN | 1 | 标记EXE模块传入MEM模块的数据是否有效 |
| ex\_to\_mem\_wire | IN | 104 | EXE模块传入MEM模块的数据 |
| mem\_to\_wb\_valid | OUT | 1 | 标记MEM模块传入WB模块的数据是否有效 |
| mem\_to\_wb\_wire | OUT | 70 | MEM模块传入WB模块的数据 |
| mem\_rf\_zip | OUT | 38 | MEM模块前递到ID模块的数据 |
| data\_sram\_rdata | IN | 32 | 数据ram读数据 |

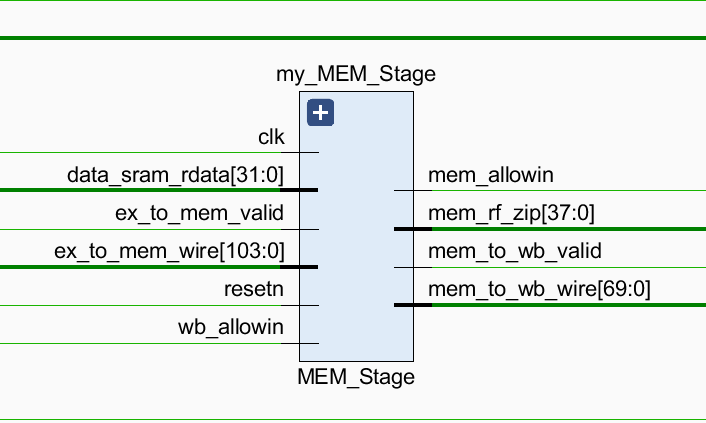


图15 MEM流水级接口

* + 1. 功能描述

根据EXE模块传递来的数据执行相应的访存操作。并将访存指令结果、写回控制信号、PC等信息在下一个时钟上升沿传递给WB模块。

内存读结果赋值逻辑如下：

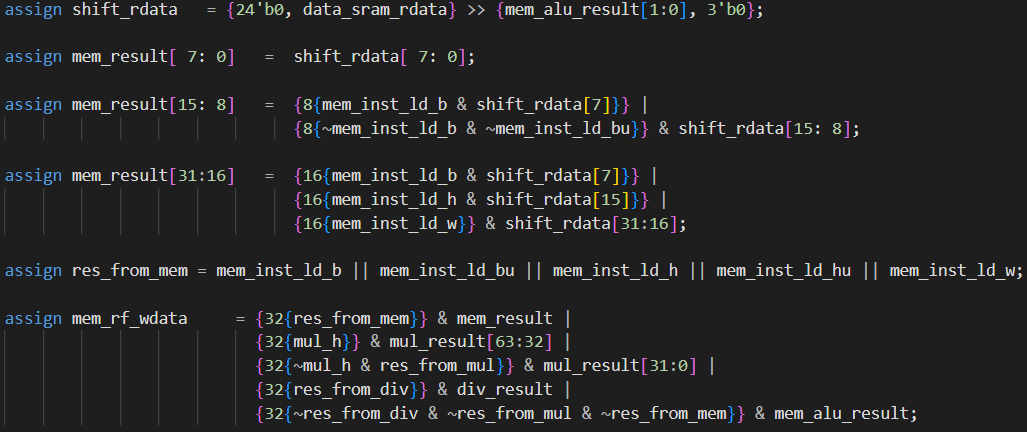


图16 内存读结果赋值逻辑

先根据地址后两位进行移位，使rdata的最低位与应读数据的最低位对齐。然后根据指令的不同分别对rdata进行扩展，生成mem\_result。之后进行寄存器堆写回数据的选择。

1. 实验过程
   1. 实验流水账

贾城昊

2023年10月13日 19：00-21：00 阅读讲义

2023年10月14日 8：00-11：00 完成exp10的译码逻辑部分大致逻辑

2023年10月14日 12：00-14：00 完成exp10的对非乘除法指令的处理

2023年10月14日 19：00-23：00完成exp10中对乘法除法指令的处理以及对exp10的debug

李金明

2023年10月13日 19：00-21：00 阅读讲义

2023年10月14日 12：00-16：00 完成乘法器

2023年10月14日 17：00-19：00完成除法器

2023年10月14日 19：00-23：00完成exp10中对乘法除法指令的处理以及对exp10的debug

牛浩宇

2023年10月12、13日晚上完成exp11大致逻辑

2023年10月15日下午-晚上完成exp11逻辑补充与debug

2023年10月16-19日，小组成员一起尝试对exp11的时序进行优化

* 1. 错误记录
     1. 错误1：exp10乘法器模块中booth算法输入错误
        1. 错误现象

运行仿真后，一个结果是对的，一个结果是错的，它们都是无符号数相乘。

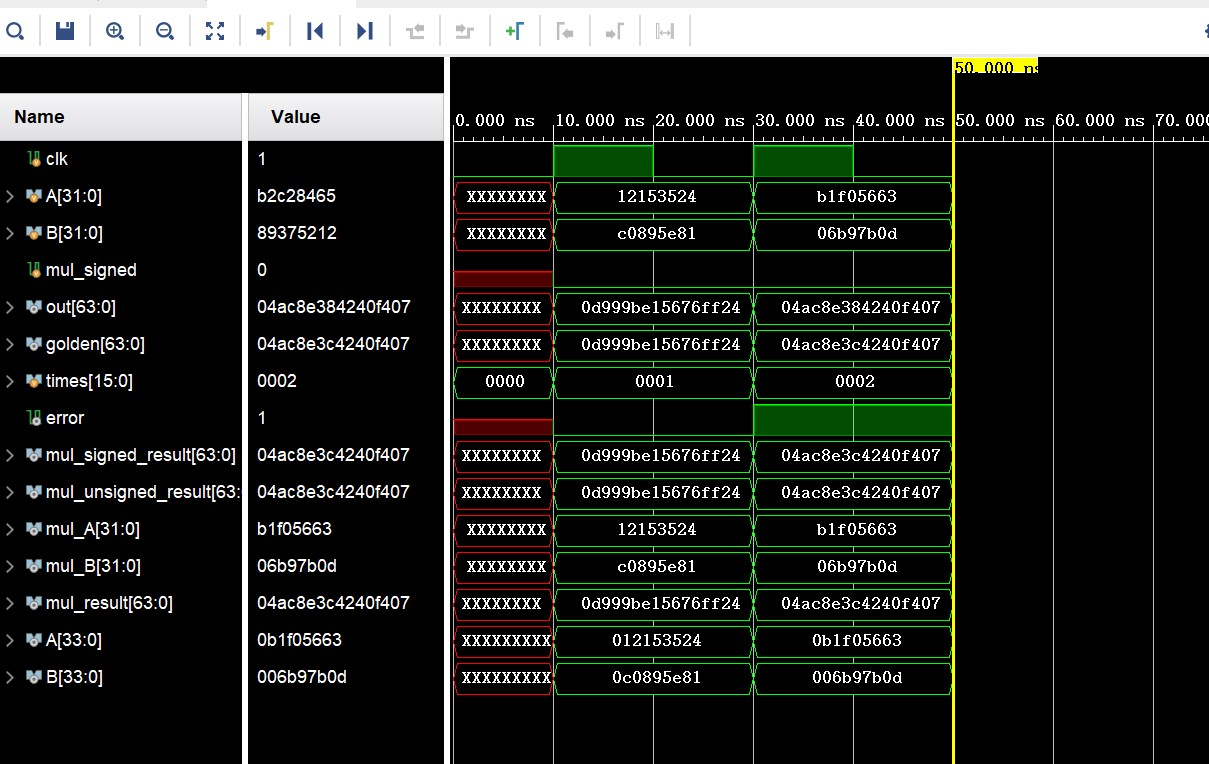


图17 错误1波形图

* + - 1. 分析定位过程

乘法器出错时较难依据信号定位问题所在。观察到出错结果和金标准相差在第34位，而这个乘法器模块是在体系结构理论课作业中已实现的32位定点补码乘法器改进来的，所以我认为出错的位置在于对操作数进行扩展时。但是阅读代码时发现以往写的64、32、16等数据该改的基本上都改了，经过一段时间才发现这一行代码出错：

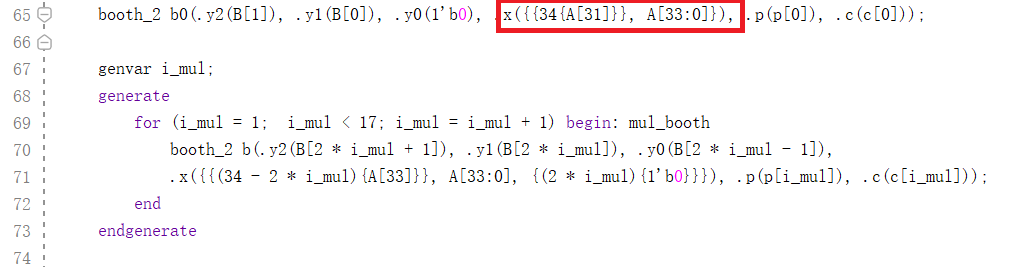


图18 错误1错误代码

* + - 1. 错误原因

Booth编码一开始要将操作数A扩展为68位，这个扩展则是有符号扩展（因为被扩展的数是34位的操作数），该扩展需要补充34位操作数的符号位，符号位应当是A的第33位，而不是31位，这里没有修改导致booth编码扩展时出错。

* + - 1. 修正效果

将出错的那一行修改为：



图19 错误1修改代码

该方法有效，来到下一个bug

* + 1. 错误2：exp10乘法器模块中booth算法输入错误
       1. 错误现象

运行仿真后，一个结果是对的，一个结果是错的，它们都是无符号数相乘。

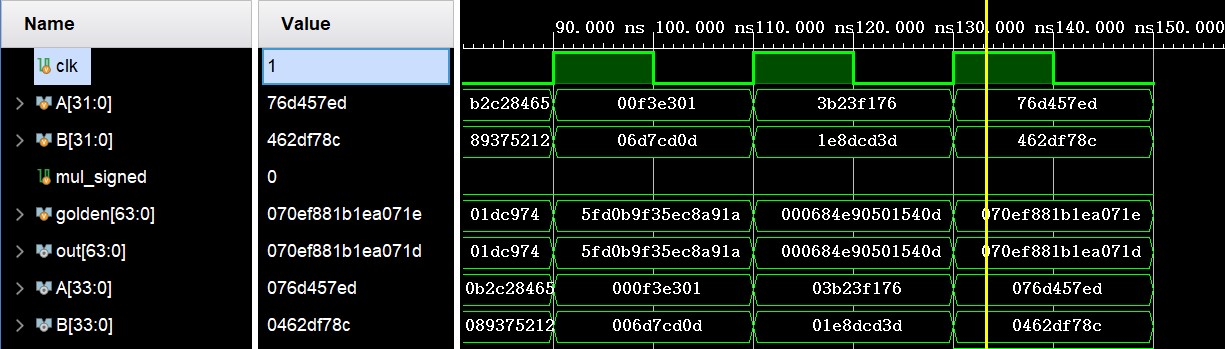


图20 错误2波形图

* + - 1. 分析定位过程

观察到出错结果和金标准相差在第0位，这种出错我认为是booth算法生成的C没有正确地传递给64位加法器导致的，查看相关代码：

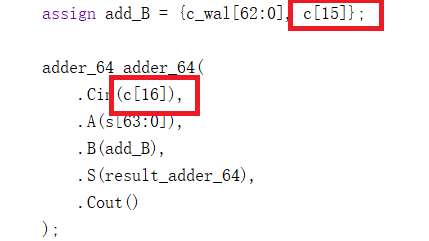


图21 错误2错误代码

* + - 1. 错误原因

Booth算法会产生17位的C，它们相当于是结果第0位的进位。其中0-14位通过华莱士树传递，第17位由于是扩展得到的，故一直为0，需要通过构造加数和进位加上的是第15、16位。但是华莱士数是一个二级流水，前一周期传入17个部分积，要在下一个周期才能拿到结果，即64位加法器的源操作数。而此时已经操作数，C也可能发生改变，因此需要使用一个寄存器存储C的第15位和第16位。

* + - 1. 修正效果

使用一个寄存器存储C的第15位和第16位：

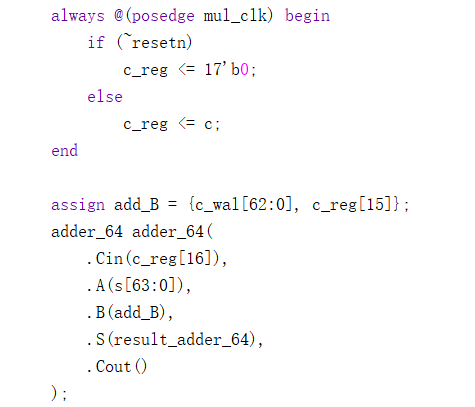


图22 错误2修改代码

该方法有效，乘法器单独验证通过。

* + 1. 错误3：exp10中ID与EXE流水级的寄存器信息解析出错
       1. 错误现象

运行仿真后，仿真一直在进行，但是PC值却没有变化，如下所示：：

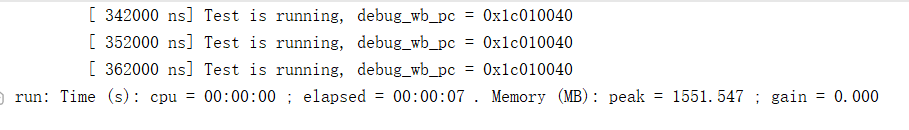


图 23 错误3 对应的Console报错

* + - 1. 分析定位过程

根据经验可知，是比对信号出现问题，导致一直没有与金标准进行比对，于是本人通过波形，观察是哪一个流水级首次出现错误，波形图如下：

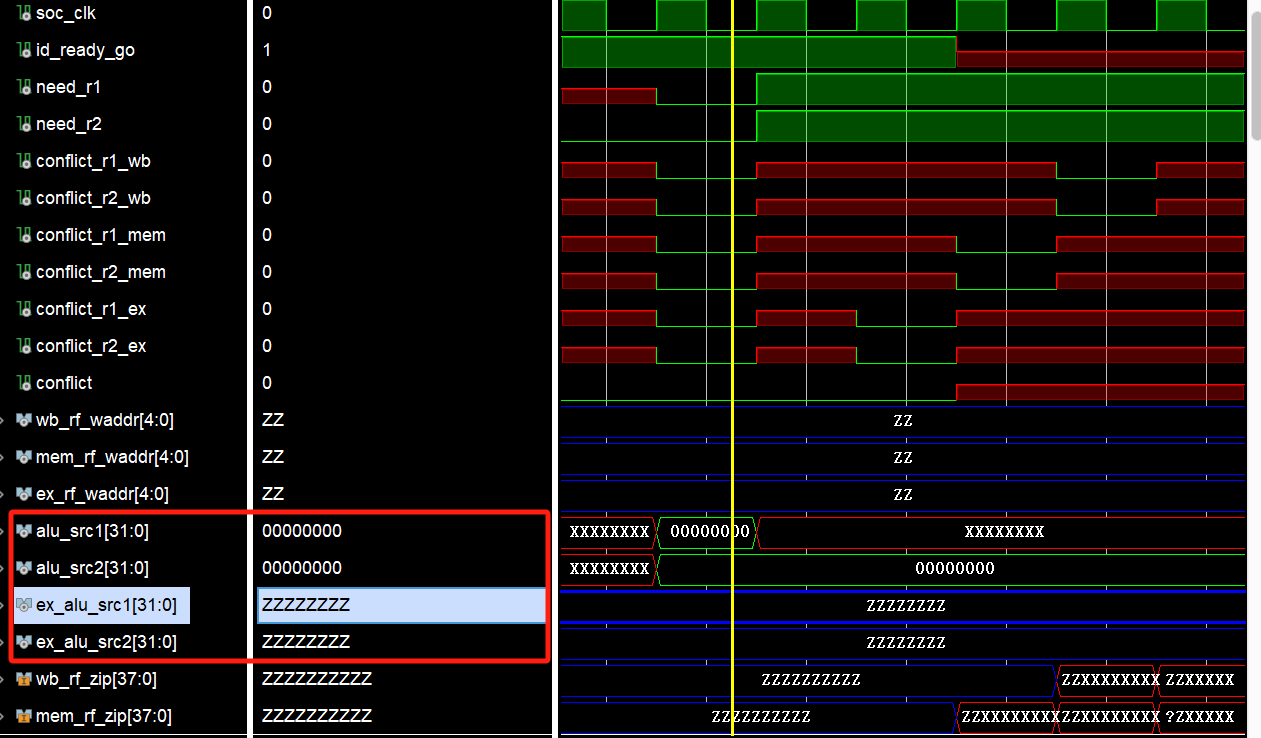


图24 错误3中alu\_src全为高阻态

可以看到，EXE流水级的ex\_alu\_src1和ex\_alu\_src2均为高阻态，所以本人查看了这两个信号的赋值，发现是出现了笔误，导致ID与EXE流水级的寄存器信息解析出错：

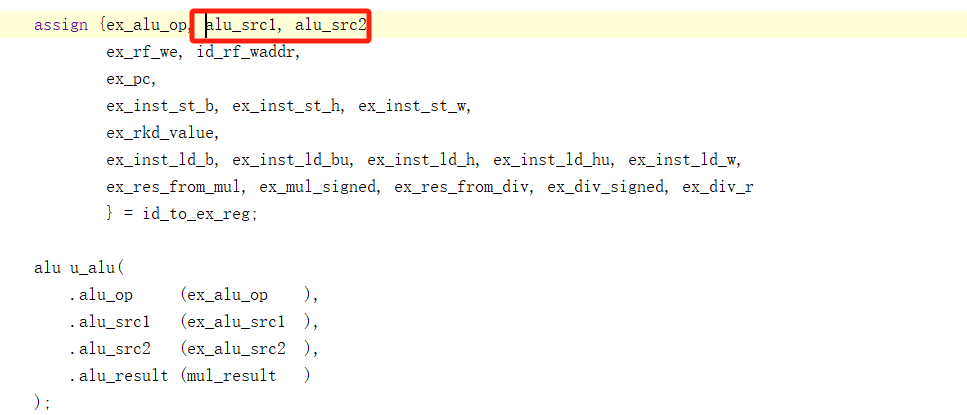


图25 错误3修改前代码(1)

于是本人把alu\_src1和alu\_src2改为ex\_alu\_src1和alu\_src2，再次进行仿真，却仍然在这里出错，于是本人再次观察波形，发现如下结果：

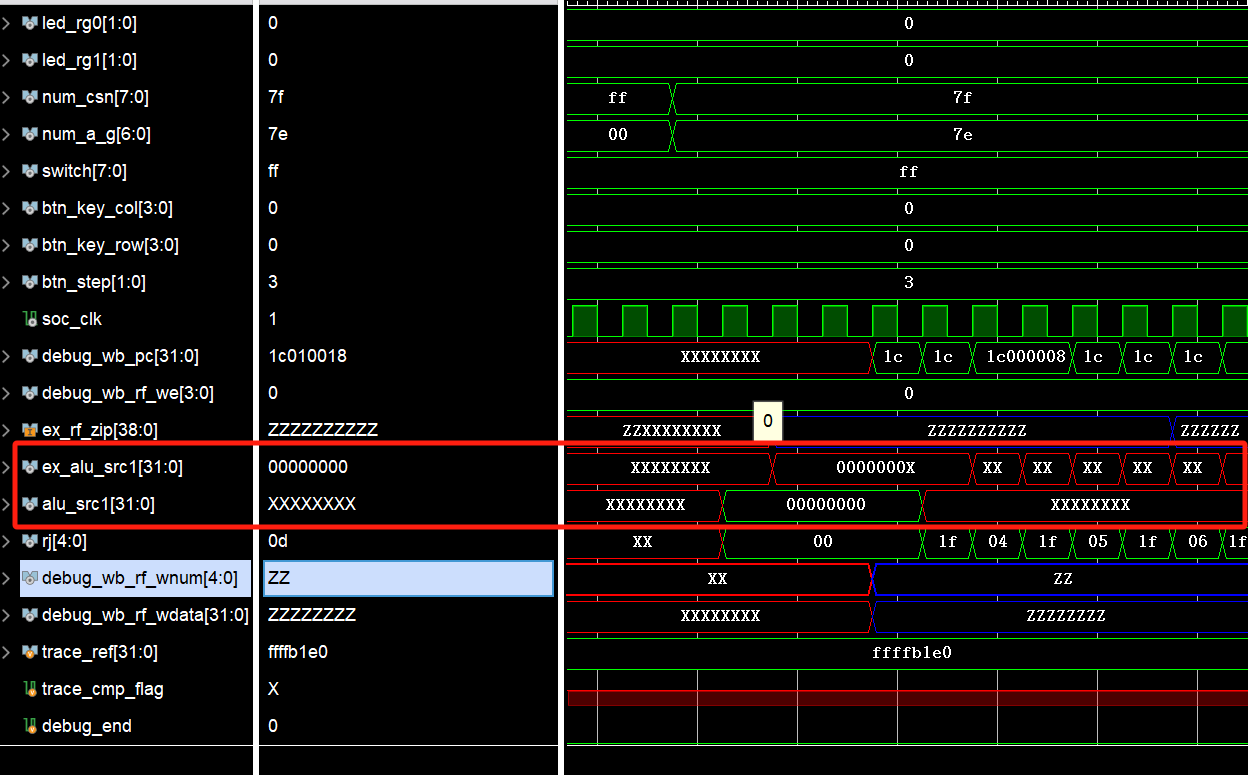


图26 错误3中EXE流水级与ID流水级alu\_src1不对应

可以看到，ex\_alu\_src1与ID的alu\_src1并不对应，所以看来对ID与EXE流水级的寄存器信息解析还是有问题，观察源码发现还是笔误，如下所示：

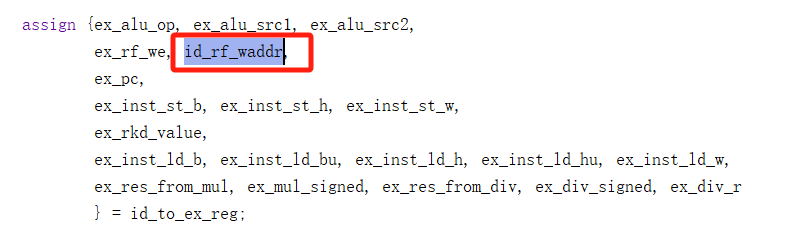


图27 错误3修改前代码(2)

由于id\_rf\_waddr在EXE流水级没有定义，所以其默认位宽为1，才导致了上面波形图出现的ex\_alu\_src1与ID的alu\_src1不对应的问题。

* + - 1. 错误原因

由于笔误，导致ID与EXE流水级的寄存器信息解析出错

* + - 1. 修正效果

将对ID与EXE流水级的寄存器信息解析的笔误修改过来即可，如下所示：

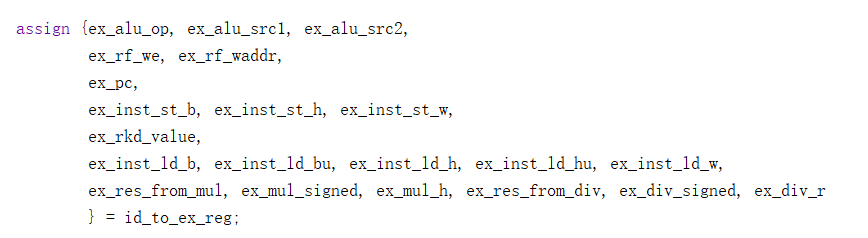


图28 错误3改后代码

* + 1. 错误4：exp10中alu\_op译码出错
       1. 错误现象

Console报错如下：

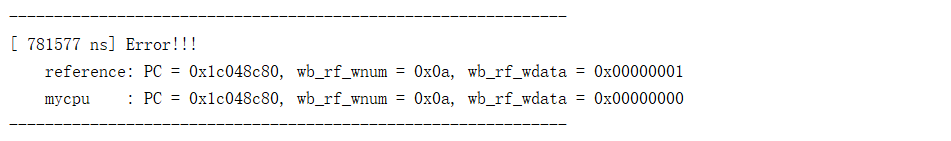


图 29错误4对应的Console报错

* + - 1. 分析定位过程

发现wb\_rf\_wdata错误，由于已经跑了很久的仿真，所以猜想是对新加的指令的处理出错。于是本人首先通过行为仿真，定位出错位置的相关信号，波形图如下：

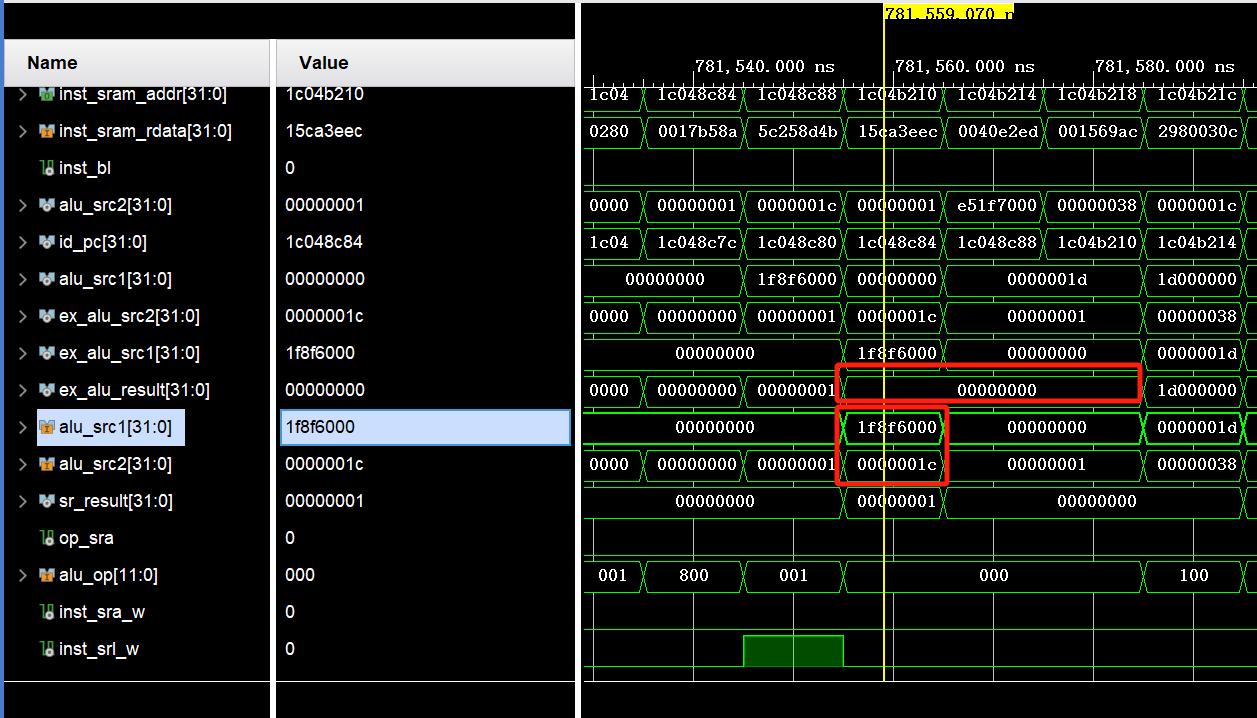


图30 错误4 的alu\_result计算出错

此时的alu模块的alu\_src1和alu\_src2以及alu\_result如上所示，本人紧接着查看反汇编代码，发现此时为srl\_w指令。然而手动验算发现，在该alu\_src下，alu\_result本就应该为0x00000001，所以本人猜测是alu的处理出错。进一步查看波形图信号，可以发现不对的地方，如下所示：

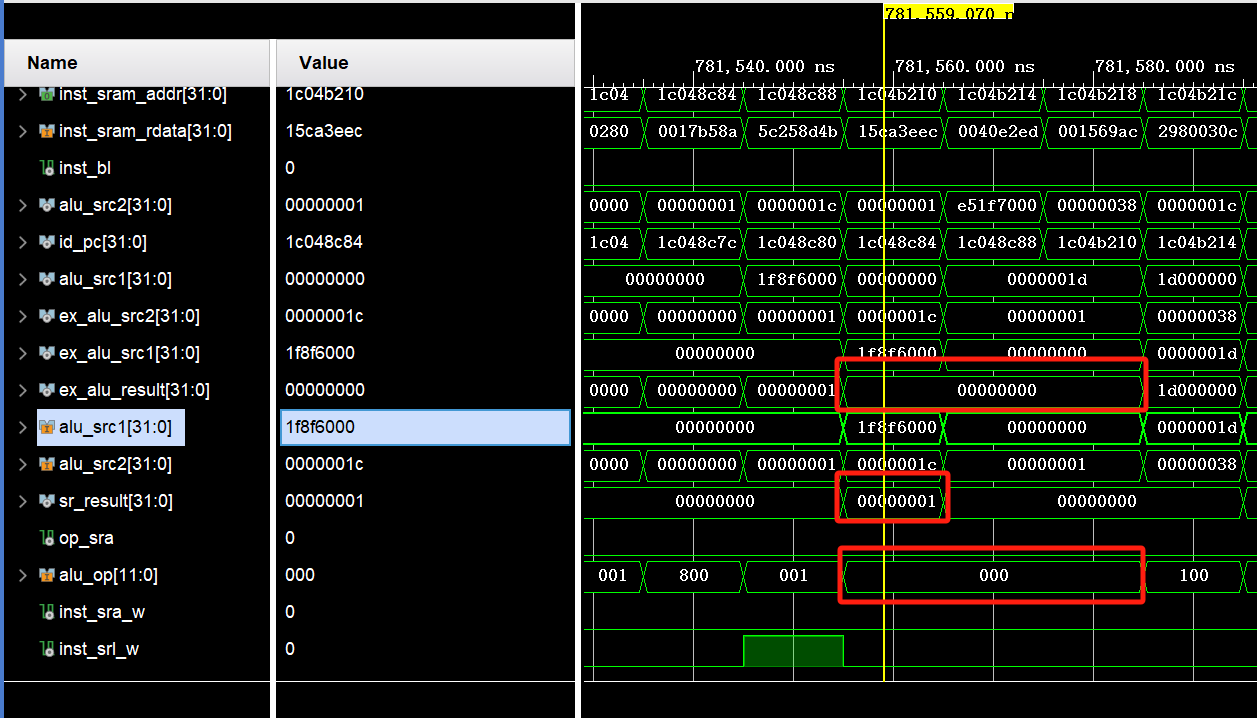


图31 错误4 的alu\_op出错

可以看到，此时的alu模块的sr\_result是正确的，但与alu\_result不符，进一步查看，可以发现，此时的alu\_op居然为000，可以看出是alu\_op的译码出现了问题，于是本人查看了alu\_op的译码逻辑，发现了错误，相关代码如下所示：

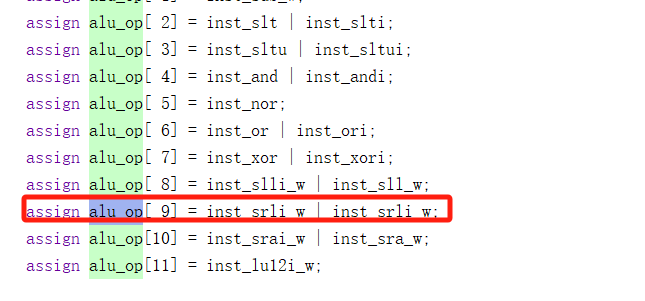


图32 错误4修改前代码

* + - 1. 错误原因

在ID流水级对alu\_op译码的过程中，由于笔误，导致alu\_op[9]的赋值逻辑出错。

* + - 1. 修正效果

将ID流水级的alu\_op[9]的赋值逻辑修正过来即可，如下所示：

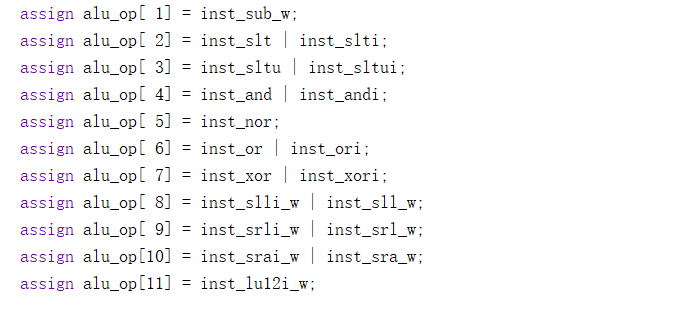


图33 错误4修改后代码

* + 1. 错误5：exp10中传递给除法器模块的信号赋值逻辑出错
       1. 错误现象

Console报错如下：

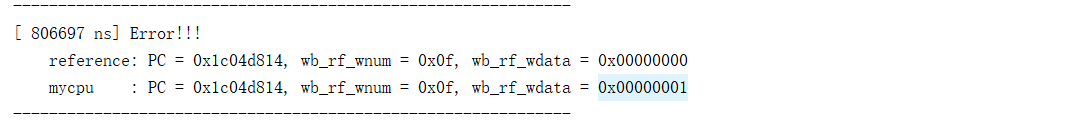


图34 错误5 对应的Console报错

* + - 1. 分析定位过程

发现wb\_rf\_wdata错误，由于已经跑了很久的仿真，所以猜想是对新加的指令的处理出错。于是本人首先查看了反汇编代码，发现此时是div.w指令，看来大概率是除法指令处理出错，于是紧接着通过行为仿真，定位出错位置的相关信号，波形图如下：

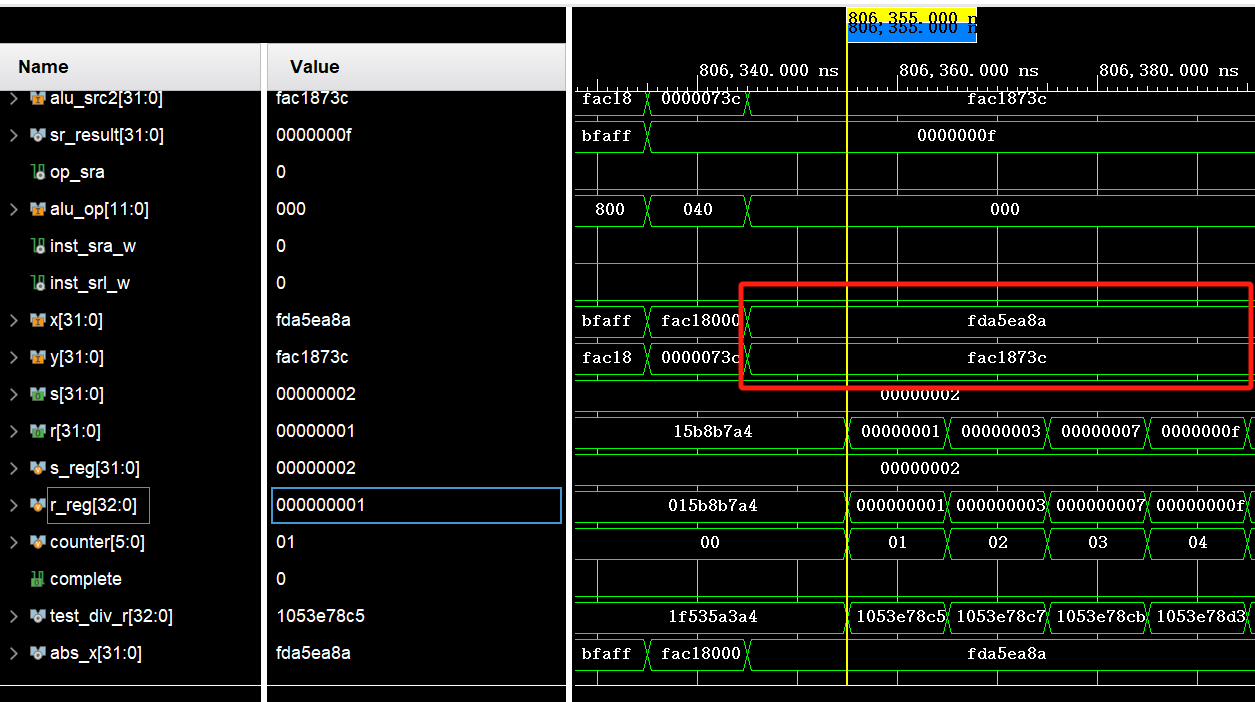


图35 错误5中除法器的输入

此时的除法器模块的x和y如上所示，本人通过计算，发现计算结果本就应该为0x00000000，但输出是0x00000001，于是经过讨论，本组成员首先认为是除法器的实现出错，但是通过给除法器单独写的test\_bench可以看出，除法器的功能并没有出错。

于是本组成员将目光又聚焦到出错位置的除法器信号，通过较长时间的排查，发现了除法器模块中信号不对的地方：

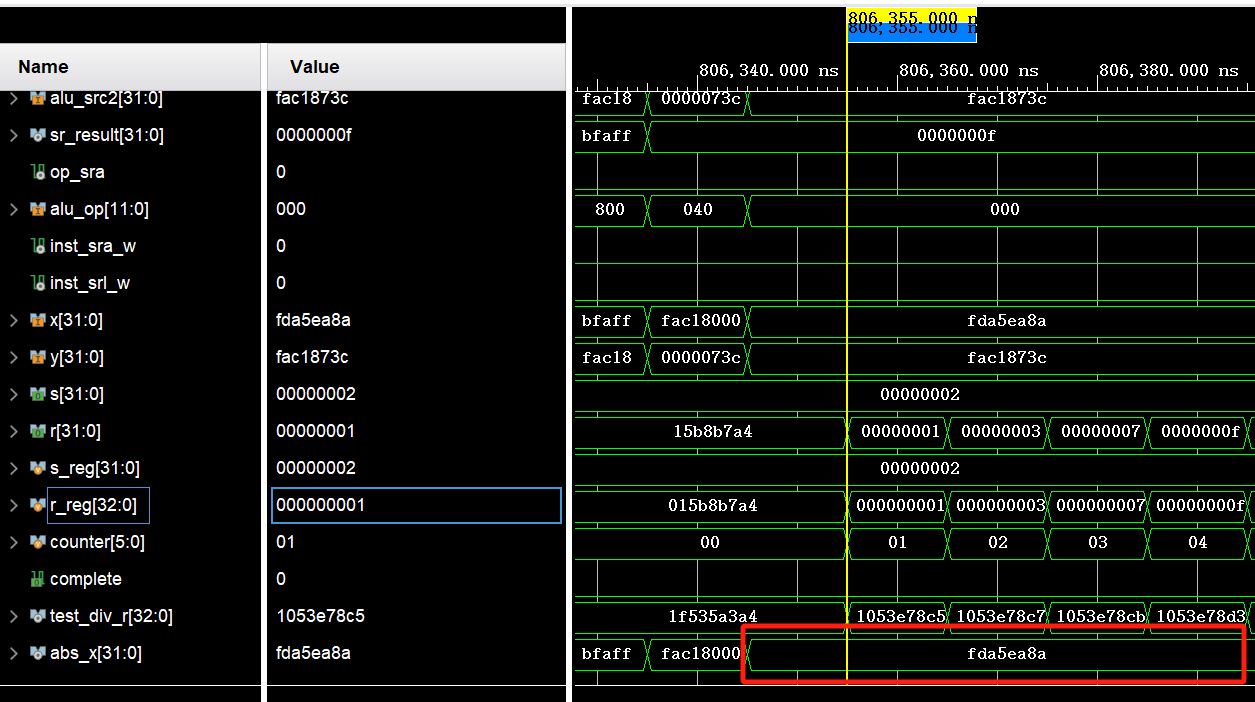


图36 错误5 中除法器被除数的绝对值出错

除法器模块中abs\_x代表的是x的绝对值，但显然对于div.w指令来说，此时x为有符号数，但此时abs\_x明显为负数，查看除法器模块对于abs\_x的处理如下：



图37 错误5 中除法器abs\_x逻辑

可见其逻辑没有出错，那么出错的地方只可能是div\_signed信号出现问题了，查看ID阶段的div\_signed信号赋值逻辑，果然发现了问题：

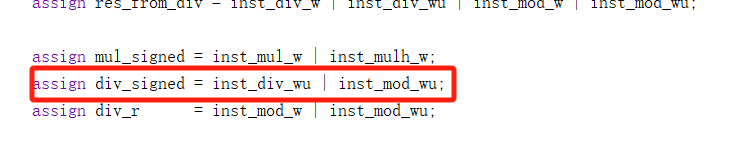


图38 错误5修改前代码

最后复盘发现，是因为写译码逻辑的同学与写除法器模块的同学的信号约定没有商量好，导致出现了歧义，所以才有了上面的代码。

* + - 1. 错误原因

写译码逻辑的同学与写除法器模块的同学的信号约定没有商量好，导致出现了歧义，所以导致传递给除法器模块的信号出错。

* + - 1. 修正效果

将div\_signed赋值逻辑进行如下修改即可：

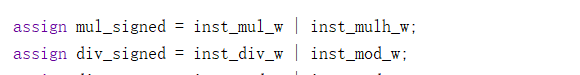


图39 错误5修改后代码

* + 1. 错误6：exp10中乘法器结果未进行高位和低位选择
       1. 错误现象

Console报错如下：

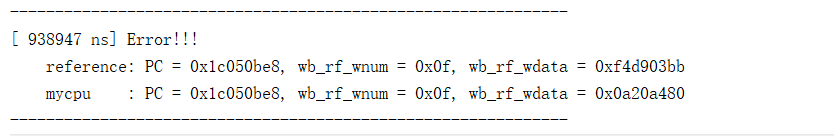


图 40 错误6对应的Console报错

* + - 1. 分析定位过程

发现wb\_rf\_wdata错误，由于已经跑了很久的仿真，所以猜想是对新加的指令的处理出错。于是本人首先查看了反汇编代码，发现此时是mulh.w指令，看来大概率是乘法指令处理出错，于是紧接着通过行为仿真，定位出错位置的相关信号，波形图如下：

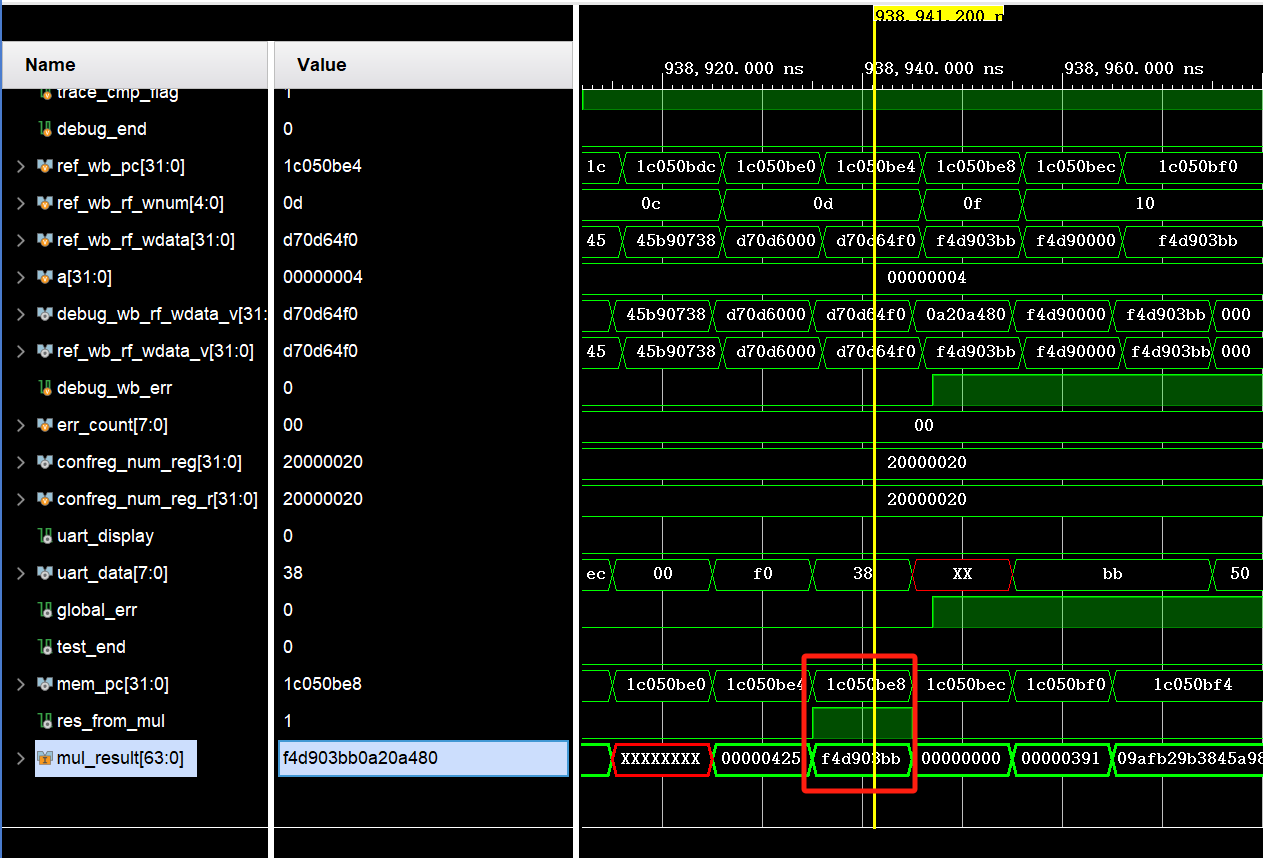


图41 错误6中乘法器计算结果

此时MEM流水级的的相关信号如上所示，可以看出mul\_result计算并未出错(其高位为0xf4d903bb)，但写回结果是0x0a20a480。所以很显然，最终写回结果选择了乘法器计算结果的低位，然后查看相关代码，发现问题所在：

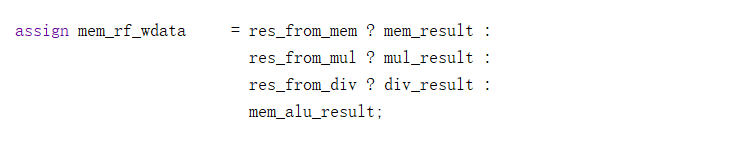


图42 错误6修改前代码

可以看出，对最终结果的选择。并没有考虑选择乘法器的高位还是低位，而mem\_rf\_wdata位宽为32位，所以默认选择了乘法器计算结果的低位。

* + - 1. 错误原因

MEM流水级没有考虑选择乘法器的高位还是低位，而mem\_rf\_wdata位宽为32位，所以默认选择了乘法器计算结果的低位。

* + - 1. 修正效果

修改数据通路的位宽，添加mul\_h信号，其拉高代表选择乘法器计算结果的高位，并把MEM流水级写回数据的赋值逻辑作如下修改即可：

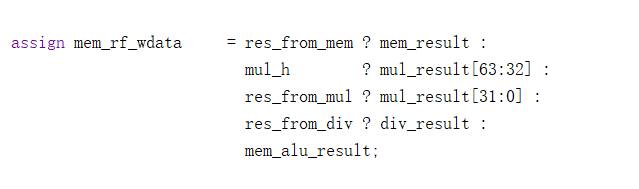


图43 错误6修改后代码

* + 1. 错误7：exp11中译码逻辑补充不全
       1. 错误现象

此错误发生两次，包含以下现象：

跳转指令的rf\_we错误拉高：

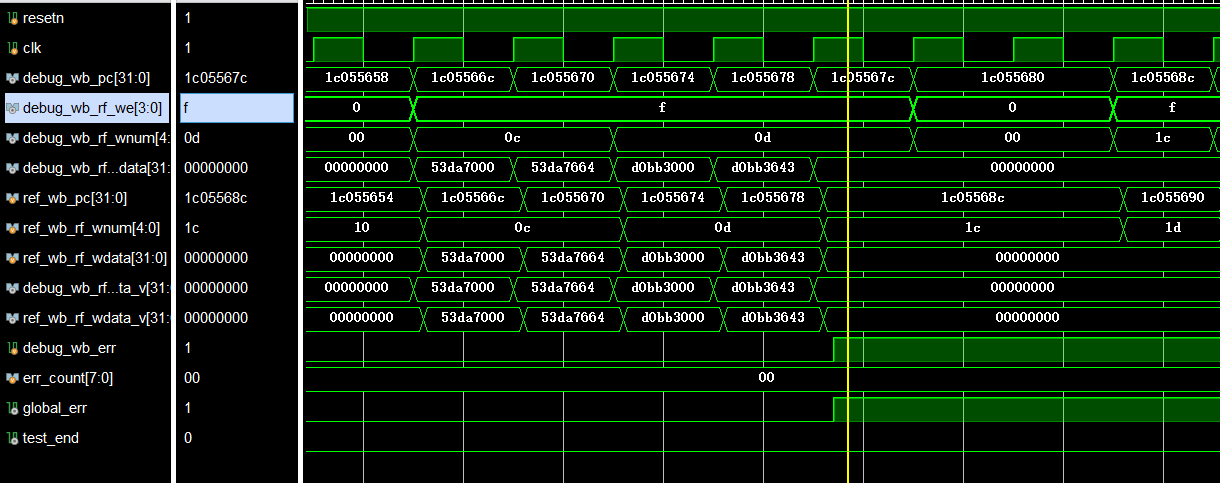


图44 错误7例1

blt指令未考虑raddr2应为rd寄存器：

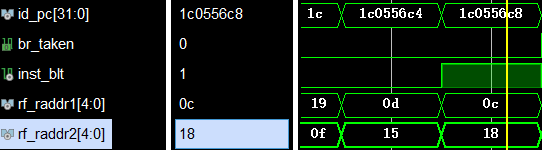


图45 错误7例2

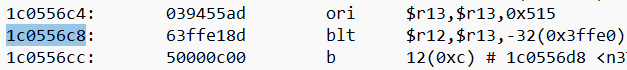


图46 例2对应指令

* + - 1. 分析定位过程

第一次，检查rf\_we的赋值逻辑，发现没有将新增指令的rf\_we拉低：

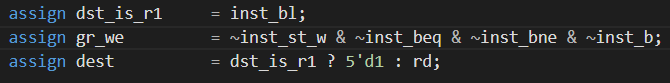


图47 rf\_we信号赋值错误

第二次，发现raddr2地址不对，检查src\_reg\_is\_rd信号，发现没有加入新增的分支与内存写信号：

2-2

图48 src\_reg\_is\_rd信号赋值错误

* + - 1. 错误原因

添加指令时对译码逻辑考虑不完全。

* + - 1. 修正效果

以上两次错误修正如下：

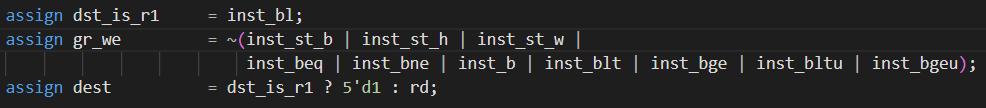


图49 rf\_we信号赋值逻辑修改

2-3

图50 src\_reg\_is\_rd信号赋值逻辑修改

之后在ID流水级对已有的分支和访存指令进行搜索，将需要补充新增指令的地方都进行补充。

* + 1. 错误8：exp11中内存读未对齐4字节
       1. 错误现象

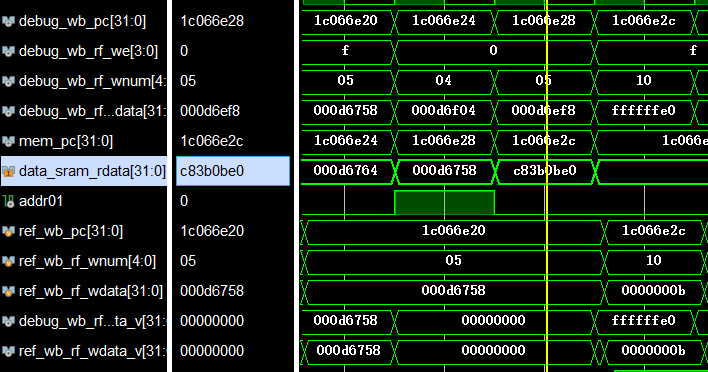


图51 错误8波形图

alu计算得到的后两位地址为01，这里应该选择0b为读数据，但实际选择了最低字节的e0。

* + - 1. 分析定位过程

检查内存读数据产生逻辑，即发现在MEM阶段处理读数据时没考虑到地址后两位未对齐的问题。

* + - 1. 错误原因

在MEM阶段处理读数据时没考虑到地址后两位未对齐的问题。

* + - 1. 修正效果

增加了按照地址后两位处理读数据的逻辑：

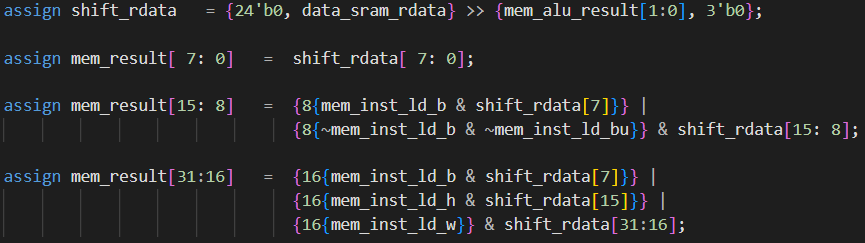


图52 错误8修改

（逻辑经过优化）

* 1. 对exp11的时序进行优化

在exp10中，本小组对乘法器切成了两级流水，所以exp10中的时序较好

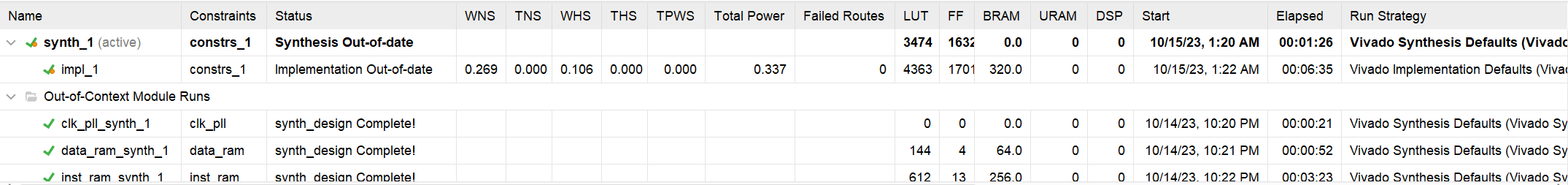


图53 exp10的时序结果

但在exp11中，由于访存指令的处理逻辑变为更加复杂，跳转的判断也更加复杂，导致exp11中的时序出现了问题，如下所示：

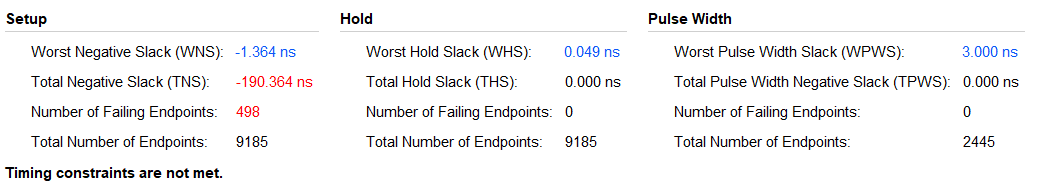


图54 exp11的最初时序结果

通过观察最差时序的路径，发现是访存指令的处理路径（访存指令的结果通过前递到ID流水级的延迟过高），于是本小组对最初的访存数据的处理逻辑进行了优化，如下所示：

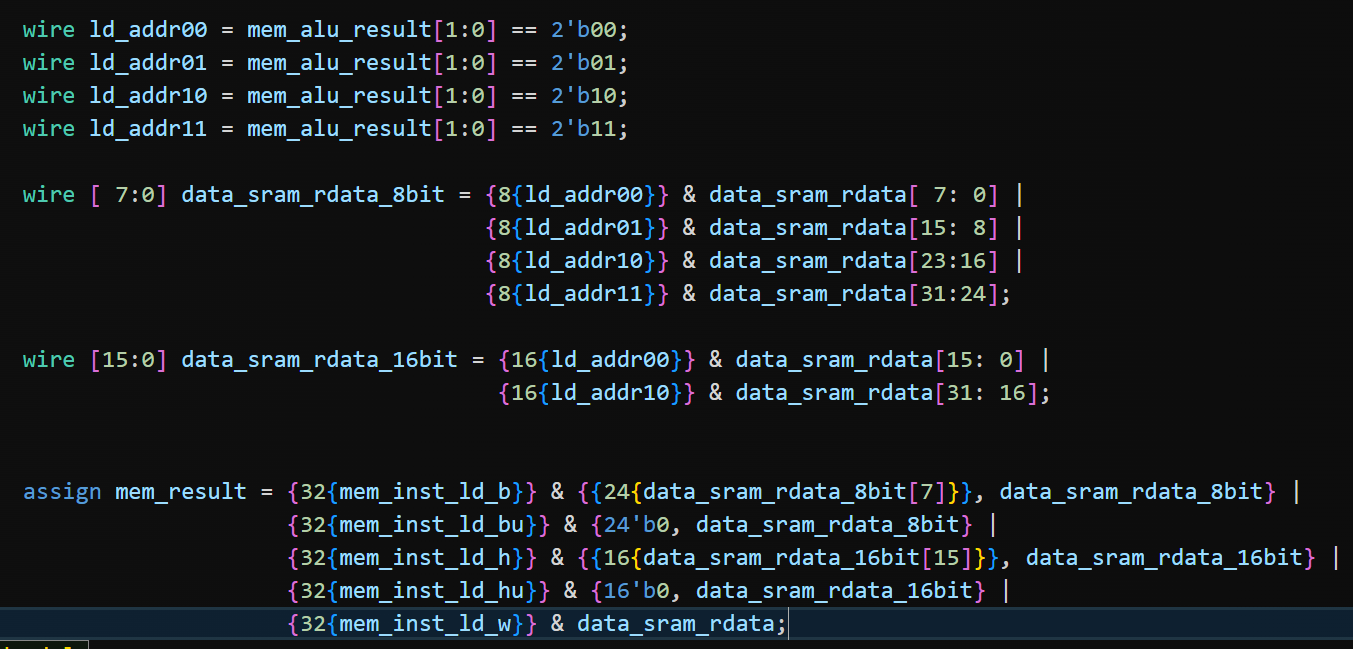


图55 exp11中对访存指令处理最初版本

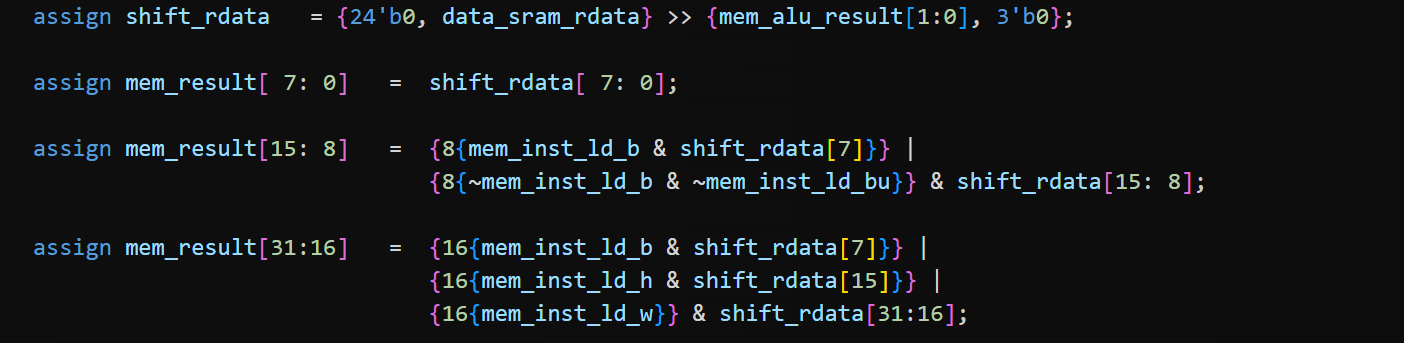


图56 exp11对访存指令的处理第一次优化

但是时序结果仍然不好：

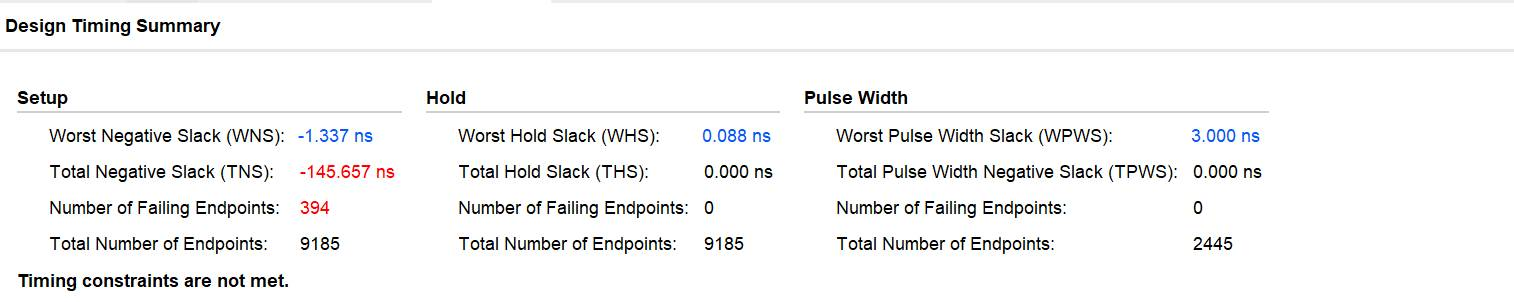


图57 exp11对访存指令的处理第一次优化后时序

通过对代码进行分析，本小组认为可能是mem\_rf\_data使用了很多个多路选择器，可能导致延迟较高，于是尝试改为独热码进行与或操作，如下所示：

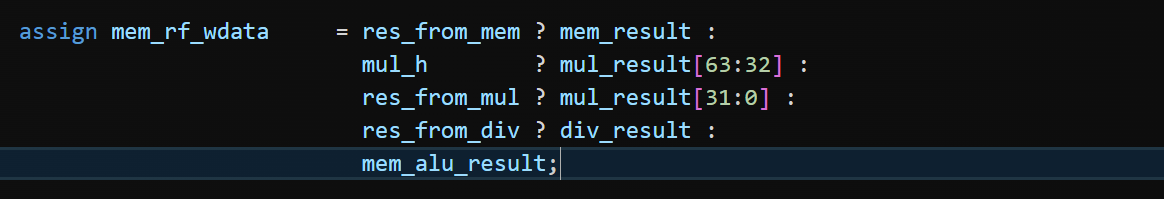


图57 exp11对mem\_rf\_wdata修改前代码

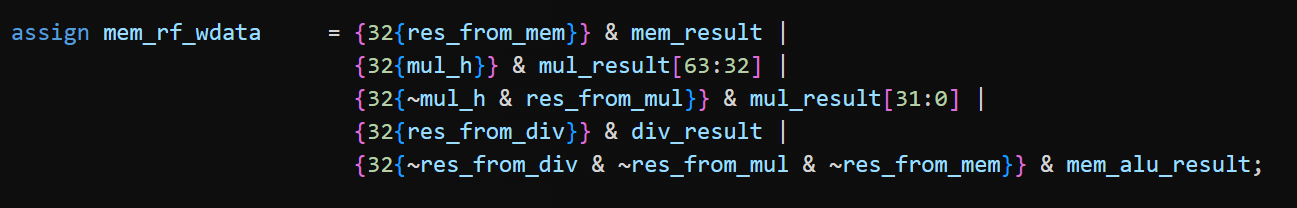


图58 exp11对mem\_rf\_wdata修改后代码

时序结果得到了改善，如下所示：

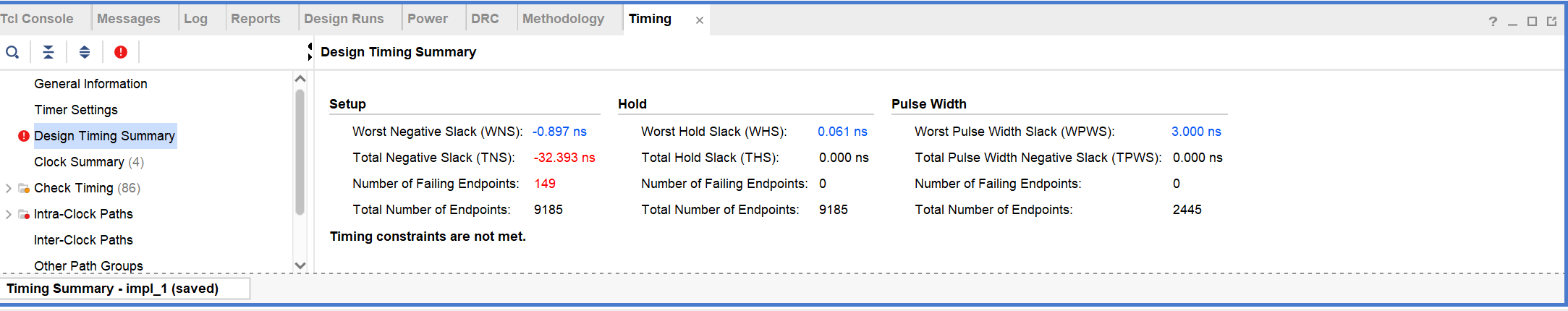


图59 exp11对mem\_rf\_wdata修改后时序

修改后，时序结果较最开始好了很多：继续对时序最差的路径进行分析，发现是br\_taken的判断逻辑的问题，于是本小组成员开始对br\_taken进行优化，最初本组成员的br\_taken的判断逻辑直接使用了 < 进行判断，如下所示：

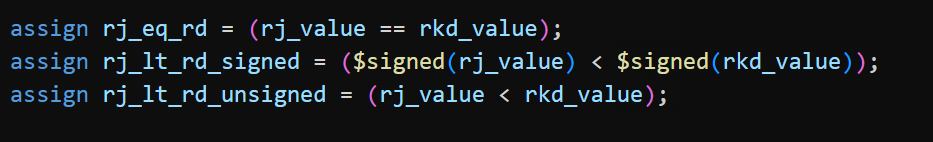


图60 exp11对跳转分支指令处理最初版本

于是本小组尝试自己实现32位的块间并行，块间并行的先行进位加法器，通过CF,OF,SF等进行判断，如下所示：

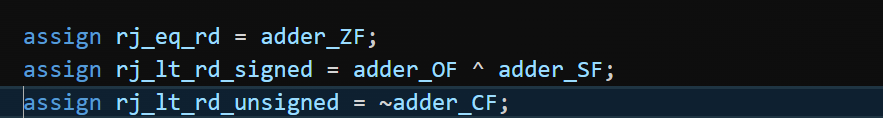


图61 exp11对跳转分支指令的处理第一次尝试优化

但最后发现时序并不如直接使用 < ，于是最后本小组成员尝试直接使用 + 而不是自己写的加法器进行处理，逻辑如下：

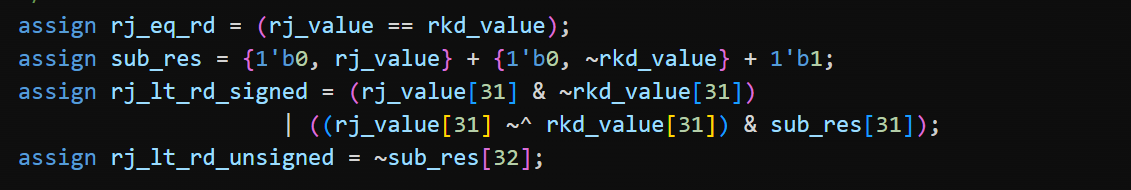


图62 exp11对跳转分支指令的处理第二次优化

最后发现时序变好了：

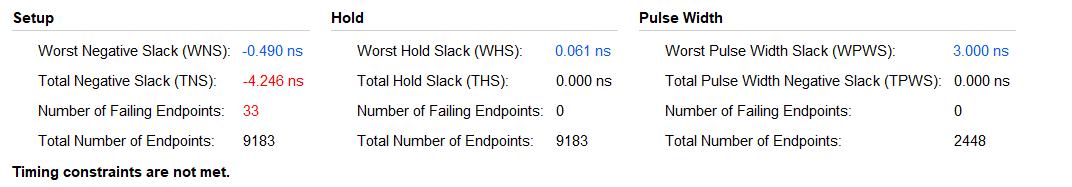


图63 exp11对访存指令的处理第二次优化后时序

后续本小组尝试对其它地方进行修改，如把华莱士数从7层改为6层，把乘法器流水切成三级，均没有让时序变得更好（可能此时主要问题是跳转分支的判断逻辑）。然后把其它的多路选择器改为独热码与或逻辑也没有改善。由于本小组成员对时序优化并不太熟悉，且其它课程的作业压力较大，最后没有进行更进一步的尝试了。

1. 实验总结

本实验添加了算术逻辑运算、乘除法运算、转移、访存四类指令，基本上都可以复用现有的数据通路，只需对控制信号作一些修改，但也有一些地方需要设计新的通路和器件，比如在ALU中添加乘法器和除法器。

在实现乘法器和除法器的过程中，较难对其进行debug，其数据较大，步骤较为繁琐，难以准确通过波形图定位bug，只能从结果和金标准的差别入手推断错误发生的位置。

小组成员也在本次实验中因为缺少交流导致模块接口定义理解出现分歧，导致了bug的产生，这一过程提醒我们要加强交流，避免此类错误的再次产生。

通过本次实验，本小组成员还对git的使用进一步熟悉，对如何使用git进行合作有了更深的理解。