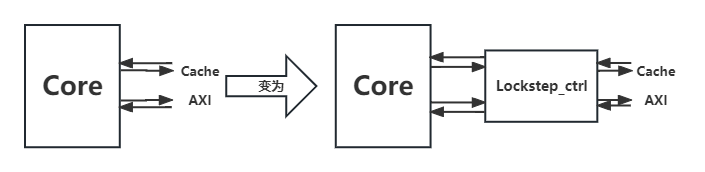
1. 流水线双核锁步

将双核的regbank进行比较，如果发生了错误，则流水线停顿stall并进行冲刷，同时退回至8周期前的PC值。

23.2.23 目前lockstep\_ctrl模块已经实现：两个Core的输出和输入替换成\*\_lockstep，以便输入输出先进入lockstep\_ctrl模块中，而与Cache或外部AXI总线交互的是lockstep\_ctrl模块，目前这些线仅仅assign直连，做这个隔离一层是为了以后的故障隔离，避免将故障传递到Cache或AXI总线。此外，dccm模块比较的输入输出也替换成了\*\_lockstep，即Core直接输出的去进行错误比较。



已将regbank的wdata和wen引出至lockstep\_ctrl模块，来自两个Core的进行比较，当错误的时候，可以冲刷流水线，即ls\_flush\_ret，此信号会传入至dpu\_ctl模块，会kill everything。但此时错误的结果已经写入了regbank。

已将pc\_instr0\_de引出至lockstep\_ctrl模块，并pipeline。

23.2.24 已将regbank引出至lockstep\_ctrl模块，用于之后pipeline后恢复正确的寄存器值。

已将写入pre-de的指令数据dat\_fe2改变为当错误发生时会变为跳转指令，并传入Core。

加固基本思路大概有了。尽量不去更改Core的具体细节部分，而是多运行一次跳转指令及控制冲刷流水线方面。当向regbank写数据时发生错误时，此时会首先拉高ls\_flush\_ret，即冲刷流水线，目前必须得持续五个周期才能完全将流水线冲刷干净，之后当冲刷结束后，临时改变传入pre-de的指令，这个临时指令是跳转指令，未来会改成BX指令，暂时用B指令代替。（不能使用B指令原因是，B指令译码后实际执行的是相对PC的偏移值，而发生故障错误的前几个周期，PC值不一定规律递增，中间可能存在函数跳转和流水线停顿等情况。未来会先临时使用r12寄存器，将pipeline的PC值存入，然后使用BX指令跳转，同时再恢复r12值）此时就会跳转至发生错误的五个周期前，即从在发生错误的指令处重新开始执行指令，错误得以恢复。

暂时存在的问题1，R4为超标量双发射处理器，会有同时运行两个指令的情况，例如

“MOV r0, 0x1; MOV r1, 0x2;”，

实际执行时r0和r1寄存器是同时被写入的。当发生错误时不能判断两个指令的前后顺序，若是有ADD等指令，恢复会可能失败。

问题2，冲刷流水线时发生错误的后两个指令并不能被冲刷掉，例如

“MOV r0, 0x1; MOV r1, 0x2; MOV r2, 0x3; MOV r3, 0x4;”,

当第一条或第二条指令发生错误时，后两条指令仍会执行完毕，即r2和r3会被写入值。此问题可这样解决：当冲刷流水线时，不允许写入regbank，但不清楚其他指令是否会有影响。

问题3，发生错误的指令当检测到错误时，已经将错误的结果写入了regbank中，暂时想到的办法是将regbank打拍，之后再重新写入这个指令之前的值。

问题4，目前方案技术上可实现，但从检测到错误再到再次恢复到正常，至少需要5+8个周期，当然由于pipeline，跳转指令后就可以当作正常，实际可能少于此数，但仍需花费较多时间，不知是否可以减少至5周期以内甚至更少。

问题5，当恢复失败时，可能会陷入“检测到错误->恢复->失败，仍检测到错误”的死循环，未来考虑使用一个计数器，当死循环次数过多时可进行复位。

23.2.26 R4取值时是64位地址对齐，例如，退回至0x28，会从0x28开始执行，但退回至0x2c，仍是一次取了64位数据，即还是从0x28开始执行。因此恢复错误时，虽然是双通道同时执行，但无论两个哪个发生了错误，都退回至地址对齐处重新执行。

目前恢复方案细节为：当错误发生时，回退至64位地址对齐的地方，可能是错误发生的地方，也可能是前一个指令处，从此地方重新执行。当错误发生时，错误信号会被拉高，此时除了冲刷流水线外，regbank的wen需要被禁能，不允许数据被写入。同时，还需记录每次64位地址对齐时的regbank里寄存器数据。对于pc的值，由于跳转或流水线停顿等因素，pipeline的方式并不太准确，而且还需要知道错误指令的地址，以便知道64位地址对齐的地方，因此只对PC进行pipeline的方式就不可再用，因为无法确定正在执行的指令的地址。目前方案想到的是，将PC值顺着流水线传入进去，这样就能知道正在执行的PC值，不过这个代码里应该有，需要看代码找到，找不到就自己写。此外，当错误发生时，PFU仍会继续取指令，需要对PFU取到的指令清空以及将valid禁掉。

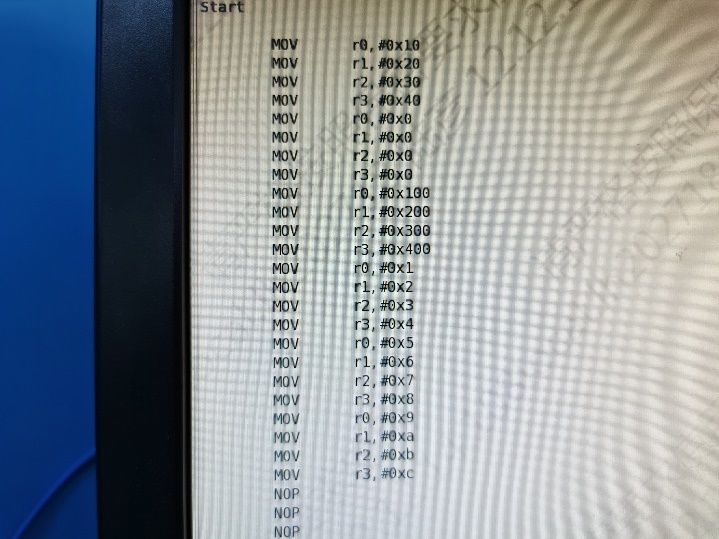
新问题1，实际取的指令是PC值减去0x8，例如，PC值跳转到了0x30，实际PFU会从0x28开始执行，但其实B指令跳转的就是0x28，这个原因好像是PFU会从PC减0x8的地方取指令。中间的指令都没问题，但复位刚开始的指令应该是从0x0开始，但PC值自从复位就是0x8，因此如果按照中间指令的方式去恢复错误，那么复位刚开始的两条指令如果出错就恢复不了了。

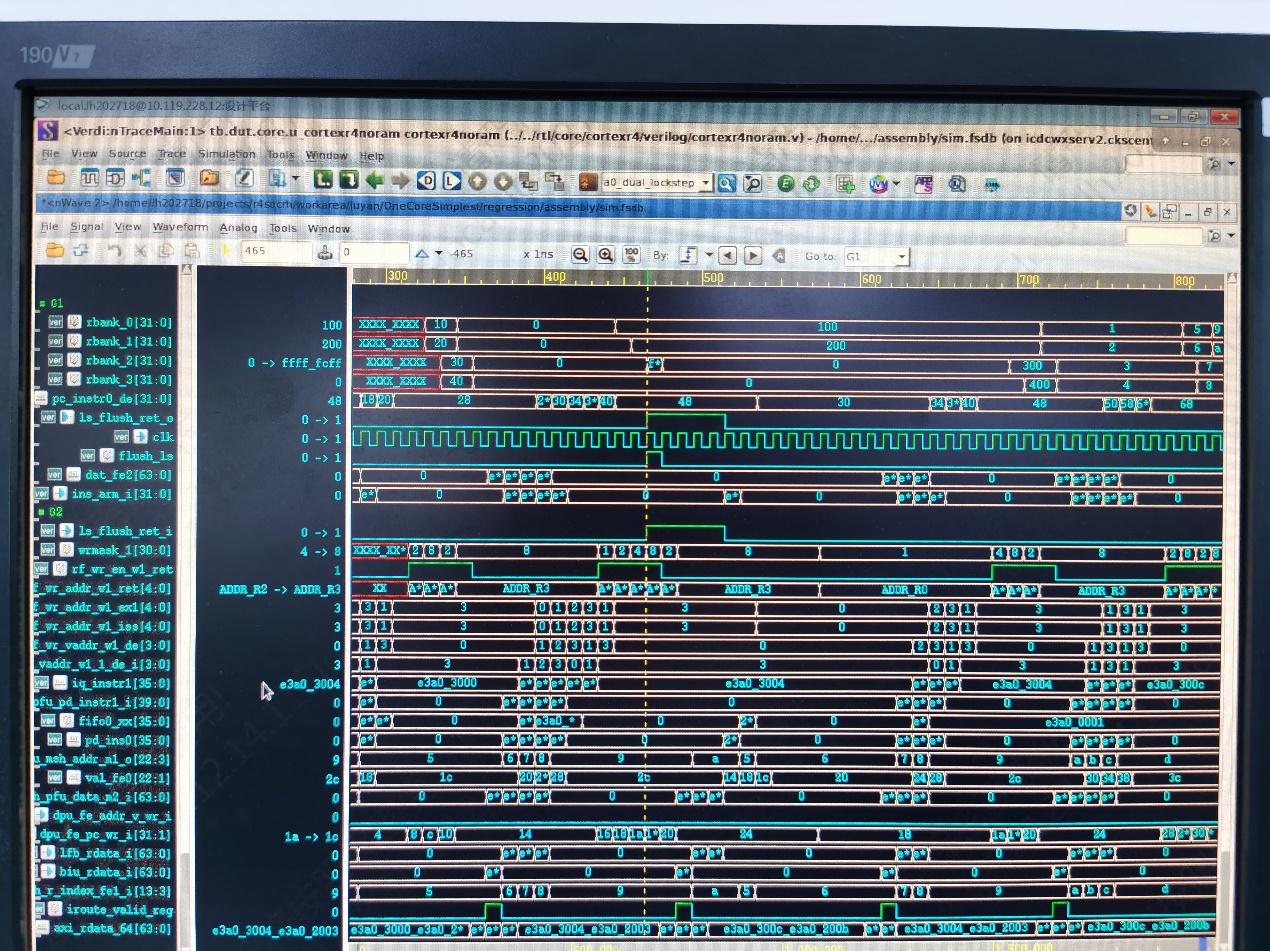
已完成，当错误发生时，冲刷流水线时regbank不会被再写入数据。

明天需要先去看没有错误的时候跳转指令到不对齐64位地址的情况是怎么执行的，关于这个错误恢复的方案可能还需更改。

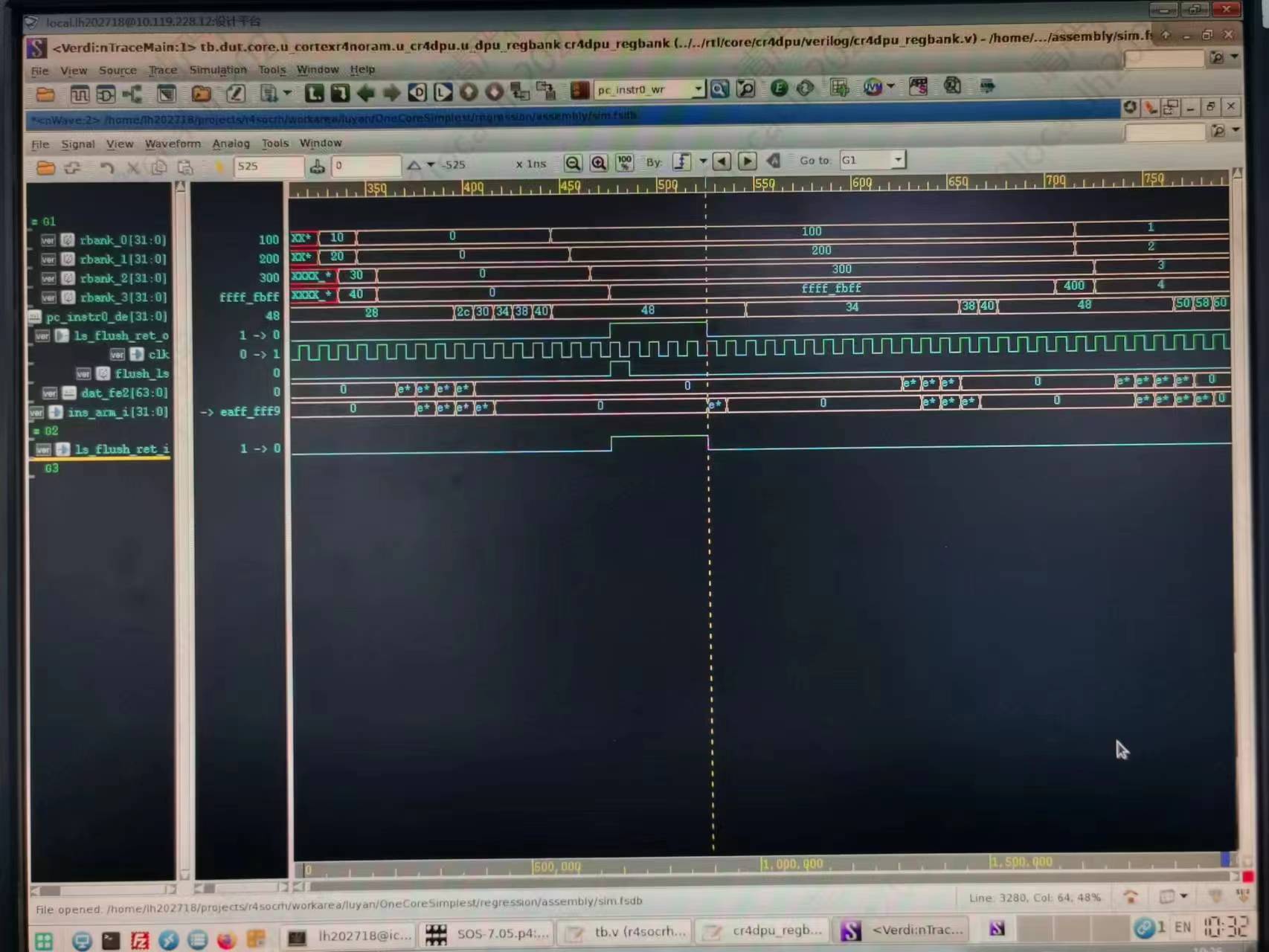
23.2.27 疑惑点搞懂，跳转指令到不对齐64位地址时，取指令仍然是取64位，送往两个通道，即两个pre-de模块，只不过有判断是否有效的信号来确定哪个指令有效。而且发生错误时手动传入的跳转指令，仍会屏蔽无效信号，因此64位信号不对齐不是问题。

目前可实现恢复的错误情况：

运行代码如下

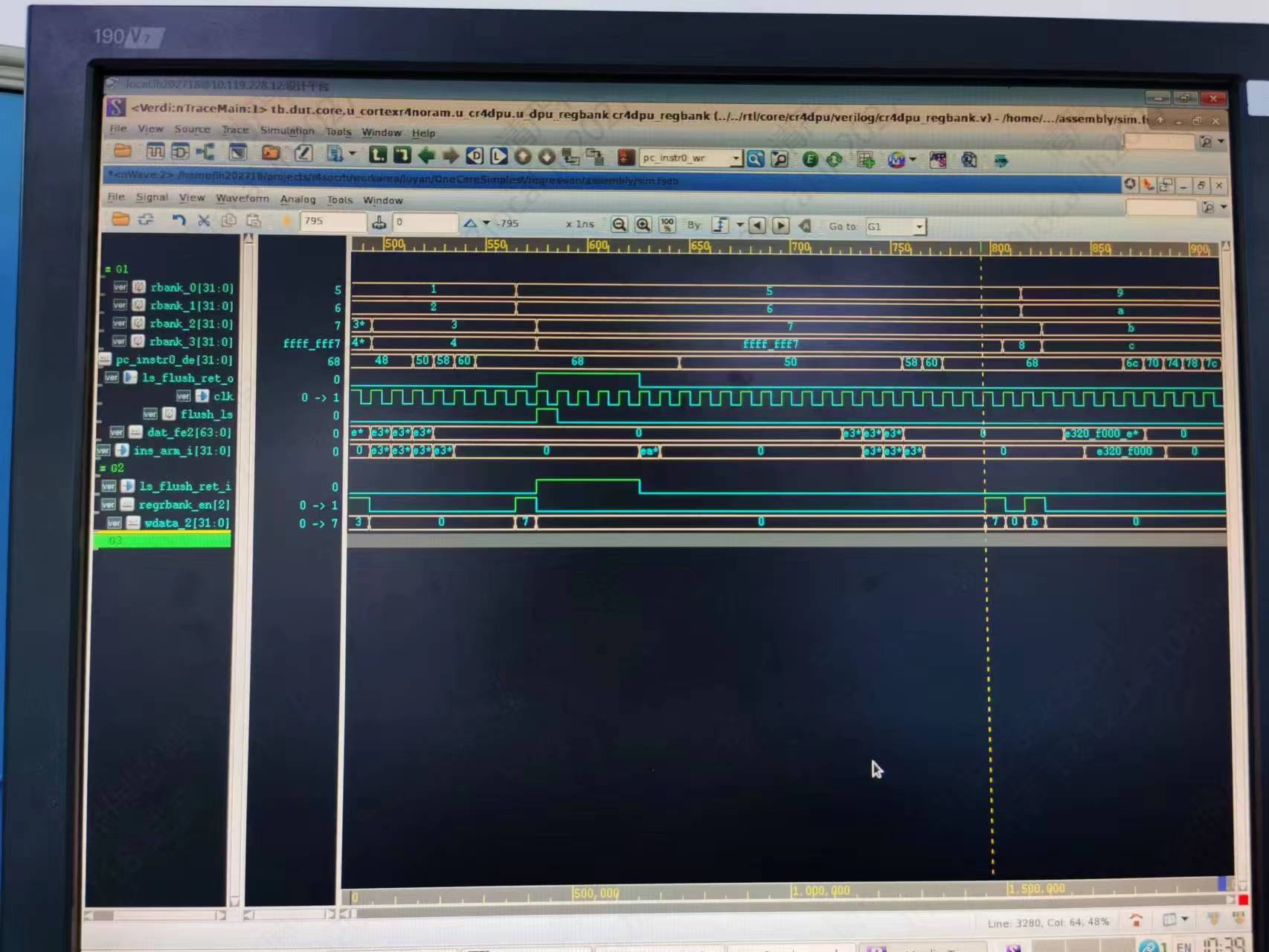
1. 

在465ns注入故障，即向r2写0x300时发生错误，可以正常跳转，重新从“MOV r2,0x300”处运行。

1. 

在475ns注入故障，即向r3写入0x400时发生错误，可以正常跳转，并重新从“MOV r3,0x400”处运行，而且前一条指令“MOV r2,0x300”并不会被执行。ins\_arm\_i的0xeafffff9即为跳转指令。

问题情况：



如果向两个寄存器同时写数据时，且出现故障的是后一条指令时，发生错误会跳转到前一条指令，两个指令都会被同时执行。在“MOV r3,0x8”出现错误，但“MOV r2,0x7”和这条指令是同时被执行的，发生错误跳转后，“MOV r2,0x7”仍会被执行，假如这条指令是ADD等其他指令，就会恢复失败。

已完成错误发生时恢复regbank。

23.2.28 关于写入寄存器堆时，这个写入动作到底是哪个指令产生的，目前想到的方案，沿着写入寄存器使能和地址一路找上去，直到找到decode阶段，此时就可确定是来自哪个指令。写入使能经过了怎样的，就将PC怎样。但由于会很复杂，不知可操作性有多大。就算找到了指令，还需要知道这个指令的确切地址。

疑惑：两个Core错一拍是怎么错的？怎么让输入的指令错开一拍，是复位错开一拍？与Memory交互的是只Master吗？

23.3.01 疑惑解答，两个Core错一拍是所有输入信号均错一拍，包括复位信号。且与Memory交互的只有Master。

传递PC这个方法暂时看来可行，会能准确地确定指令的PC值。

23.3.10 ARM的跳转指令B为0xeaxxxxxx形式：

0xeafffffd 跳到-0x4

0xeafffffe 不跳

0xeaffffff 跳到+0x4

0xea000000 跳到+0x8

0xea000001 跳到+0xc

但，pc\_instr0\_de并不是真实指令地址，假若pc\_instr0\_de为0x30，实际取的是0x28处指令，取值是64bit对齐的，假若pc\_instr0\_de为0x2c，实际取得是0x28处指令。

不使用pc\_instr0\_de打拍了，流水线中各阶段都有可能暂停，因此使用pc\_instr0\_wr，只需将这个打一拍，这一拍是实际向regbank写入的时间，不会再有暂停之类的操作。但恢复的跳转指令不能简单的相减，还需再减去0x8。

23.3.14 考虑的不够全面，现在问题是使用插入B指令，PC是跳转了，但取指令地址并没有跳转，就是cache的地址，这样的话实际恢复后取得是不对的。而且，内部存在ARM指令集和Thumb-2指令集，这两个指令集转换也是问题。

好像有一种新的方法，还是得用软件处理错误，硬件可以去记录发生错误这一时刻的寄存器堆值和PC值，并且冲刷流水线，同时进入中断程序，在软件这边通过汇编恢复跳转。