Lab 7 报告

马迪峰 2021K8009929033
饶嘉怡 2021K8009929005
范子墨 2021K8009929006

箱子号 5

一、实验任务（10%）

CPU 内部通过总线与系统中的内存、外设等进行交互，如何定义读取内存的接口，对 CPU 的速度和设计有很大的影响。举例来说，如果我们的地址在本周期发出，指令或者值便可以在本周期取回。这种设计现看起来是大大帮助我们减轻了 CPU 设计的负担，但是却在另一方面加剧了 FPGA 板上资源的消耗。FPGA 板子上有各种 IP 核（固化好的 Ram 资源等），这样设计的异步读的接口会让我们的 CPU 将不能利用这些高速资源，而是会用触发器层层堆叠。最终导致了资源的浪费和速度的降低。

在以往的设计中，我们的接口表现为 CPU 地址本周期发出，内存中的值一定会在下一周期返回，这也就是之前一直使用的 SRAM 接口。这种接口使得我们可以利用 FPGA 上的 IP 核，节省了资源，提速了 CPU，也得以完成板级验证。

然而，实际上我们需要考虑这样一个问题，CPU 的运算速度是远远高于 RAM 的读取速度的，如果采用 Sram 接口，每个周期时长要怎么设计才能保证数据一定在发出地址的下一周期返回呢？显然，我们的 CPU 周期需要大于等于 RAM 的读取周期。可是在绝大多数情况下，CPU 的运算是远远快于 RAM 的，所以采用这种接口，我们速度就直接受限于 RAM 的速度。因此，我们需要寻求一种新的总线接口，让 CPU 的周期与 RAM 的周期脱钩，读取一次数据需要多个周期才能返回，让 CPU 跑得更快。

另一方面，大量第三方 IP 核的读取都是同步读，即地址在本周期发出，数据返回在下一周期或者更后面的某一周期。按照工业标准进行定义的总线接口有利于与大量第三方 IP 进行集成。AXI 总线协议是一个是一种面向高性能、高带宽、低延迟的片内总线，很适合用于本实验中在 FPGA 中进行以数据为主导的大量数据的传输应用。因此，设计 AXI 总线就是以上问题的最好解决方案。

设计 AXI 总线接口有两种思路可供采取：

1、改造 CPU 的 SRAM 接口为类 SRAM 接口，之后搭建类 SRAM 转 AXI 的转接桥，与 CPU 一起封装，使得 CPU 在外观上表现为 AXI 接口。

2、直接设计 CPU 的 AXI 接口。

为了方便起见，本实验采用的就是第一种思路，因此也就分为三个阶段来进行

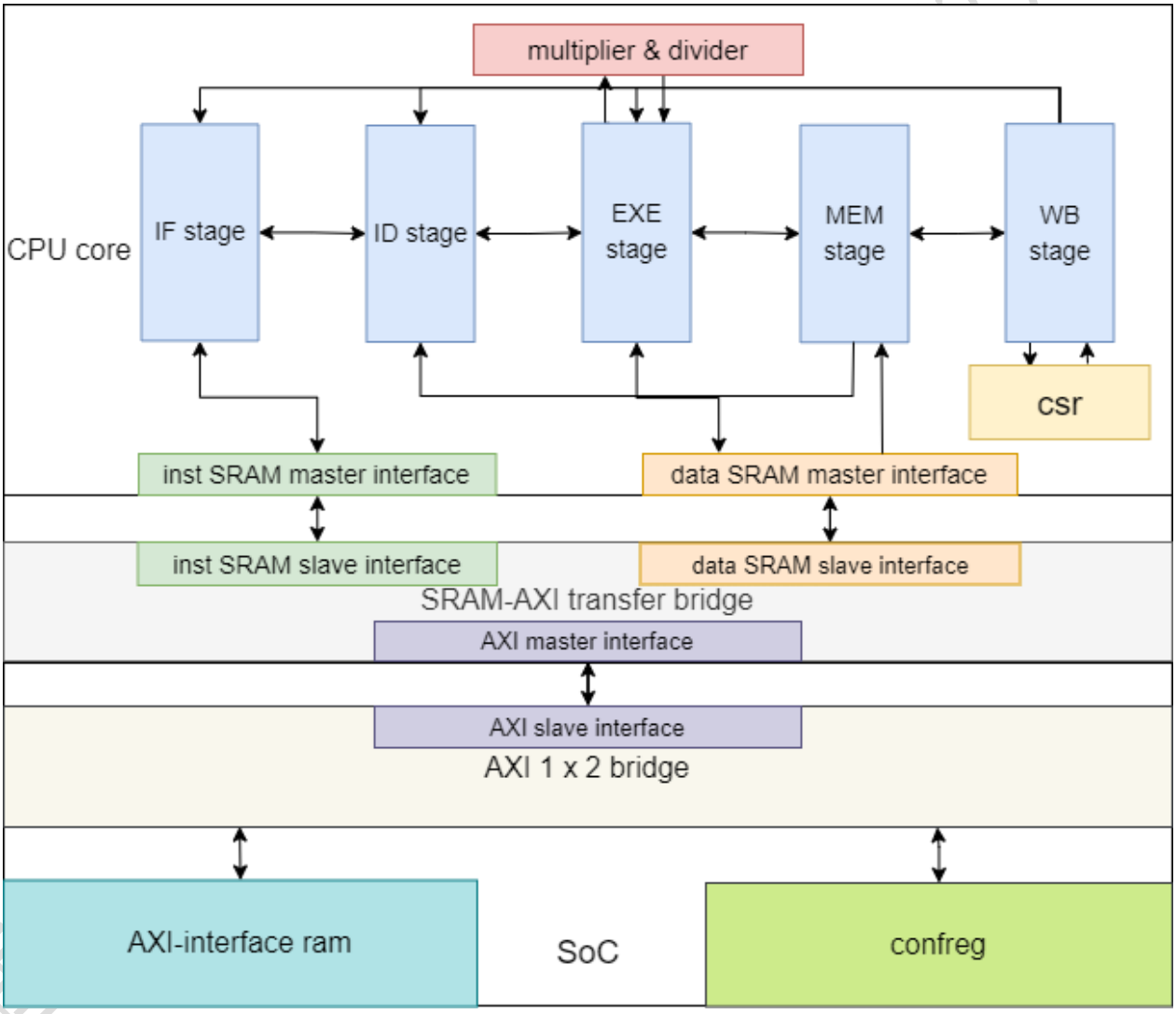
Exp14：将原有的 CPU 访问 SRAM 的接口调整为类 SRAM 接口，在访问 ram 的过程中增加握手机制，并通过随机延迟的上板验证。这一阶段的实验可以降低直接实现 AXI 总线的设计复杂度。

Exp15：设计转接桥模块，将 CPU 中的类 SRAM 接口和 AXI 总线接口进行相互转化，与 CPU 封装在一个模块中，使得 CPU 对外表现为 AXI 接口。该实验只需通过无延迟的上板验证。

Exp16：调试代码使之能够通过任意延迟的上板验证。

二、实验设计（40%）

（一）总体设计思路



图表 1 总体设计框架图

Exp14:

这个实验将 CPU 的访存接口改为类 SRAM 总线接口，因此需要调整原有访问 inst_sram 和 data_sram 的逻辑。我们很自然地能够想到，一旦 CPU 的周期和 RAM 的周期脱钩，那么每次我们向 RAM 请求读取数据时，数据的返回时刻就已经不再是固定在几周期后，而是随机的了，那 RAM 怎么知道哪一个周期的地址是有效的

呢？或者说 CPU 怎么才能知道数据什么时候返回呢？这就需要 master(CPU 端)与 slave(RAM 端)互相握手，在地址传递过来的时候，和 RAM 握个手，告诉 RAM 当前地址有效。在数据传过来的时候，和 CPU 握个手,告诉 CPU 当前周期数据有效。

观察类 SRAM 接口信号可以发现，相比原本的接口信号，多出来的信号就是由于握手机制的引入。

需要注意到一次读写一共握手两次：

size:来源于 AXI 总线协议，在 AXI 的设计中，有 arsize 和 awsize 信号，我们生成 size 信号是为了方便生成这两个信号。

req:来源于握手机制，master 向 slave 端传递，传递本周期地址信号有效的信息。表示需要对该地址进行操作，表现为读操作或者是写操作。

addr_ok:来源于握手机制，slave 向 master 端传递，用于和 req 信号一起完成读写请求的握手。表示 slave 端已经收到正确的 addr。对于读请求，这当然表示 SRAM 已经正确地收到了地址，随时可能返回读到的数据，而对于写端来说，这就是指 SRAM 此时收到的写数据会随时写入到其中。

data_ok:来源于握手机制，slave 向 master 端传递，有两种含义，在读 RAM 中这个信号有效保证当前周期读出的数据有效。在写 RAM 中表示写入完成。

信号	位宽	方向	功能
clk	1	input	时钟
req	1	master → slave	请求信号，为 1 时有读写请求，为 0 时无读写请求
wr	1	master → slave	为 1 表示该次是写请求，为 0 表示该次是读请求
size	[1:0]	master → slave	该次请求传输的字节数，0: 1byte；1: 2bytes；2: 4bytes
addr	[31:0]	master → slave	该次请求的地址
wstrb	[3:0]	master → slave	该次写请求的字节写使能
wdata	[31:0]	master → slave	该次写请求的写数据
addr_ok	1	slave → master	该次请求的地址传输 OK，读：地址被接收；写：地址和数据被接收
data_ok	1	slave → master	该次请求的数据传输 OK，读：数据返回；写：数据写入完成
rdata	[31:0]	slave → master	该次请求返回的读数据

图表 2 类 SRAM 总线接口定义

所谓另一次握手，在类 sram 接口中，其实是我们认为 master 是时刻准备好接受信号的,所以一旦 addr_ok 有效，master 端（CPU 端)就会将数据存起来。可以认为这是一个隐形的握手，不过是 matser 一直在伸手，只等 slave 伸手（addr_ok）。

在这里需要补充一点，在对 RAM 发起一次读写请求事务的请求握手成功之前，是可能再多次地完成其他读、写事务的握手的 ，这样就导致会出现相当复杂的握手信号序列。解决方案也是很显然的，只需要在 Master 段拉低 req 信号来暂停发送新事务的请求,这样对 IF 流水级的约束就更加严格了。

理解了类 SRAM 接口是怎么握手的，设计起来就比较容易了，这里主要分为两种情况来进行考虑也就是指令 RAM 和数据 RAM。

1、指令 RAM：

指令 sram 对于取指，也是有一个伪流水级 pre-IF 级来负责发送请求，IF 级负责接收返回的指令。pre-IF 级会计算 nextpc，并对 SRAM 发出请求从 nextpc 取指。 从这里开始，实际上 pre_IF 已经可以算是一个完整的的流水级了，也就是 pre-IF 级，当取指未完成的时候，fs_valid 不会是有效的，通过这个就可以控制在 IF 级的 PC 等待接口取回指令，取好后一起送到 ID 级运行。在 fs_allowin 为 1 时再发送取指请求。这可以保证在发完请求后 pre-IF 级一定能流入 IF 级，至多也只会有一条需要取消的指令。

IF 级负责从 inst_sram 接收读出的指令，由于读完后可能无法立即流入 ID 级，原因是 ID 和之后的流水级可能正在处理对应的某条指令，需要等待多个周期。因此我们需要一个寄存器 fs_inst_buf 来保存读出的指令，另外还需要一个 fs_inst_valid 来记录 buffer 中的指令是否有效。

异常和跳转指令的处理：

由于 IF 取指要多周期，而一旦 ID 级集齐了 PC 和指令，ID 就会发车,那么对于跳转信号来说，它只会持续一周期。那么就极有可能产生当 IF 读取下一地址时。由于 ID 级已经不是跳转指令，导致了 IF 错误读取 PC+4。因此我们需要在 IF 阶段设计寄存器来保存每一次跳转指令的相关控制信号，对于异常控制信号，道理也是如此。在我们的设计中，取指暂停的只是 IF 级。ID 级之后是可以正常运行的，所以很有可能出现在取指的等待指令阶段，之后的 WB 突然发送了异常，那么我们需要手动丢弃这一次的取指，因为由精确控制我们知道，我们认为 WB 之前的指令未发生，WB 之后的指令均已完成。为了解决这个问题，需要在 IF_stage 上新增一层逻辑，称为 IF_inst_cancel ,而它的作用就是在异常和跳转等指令发生且当前 pfs_stage 已经发送了取指请求时，取消这条得到的指令。

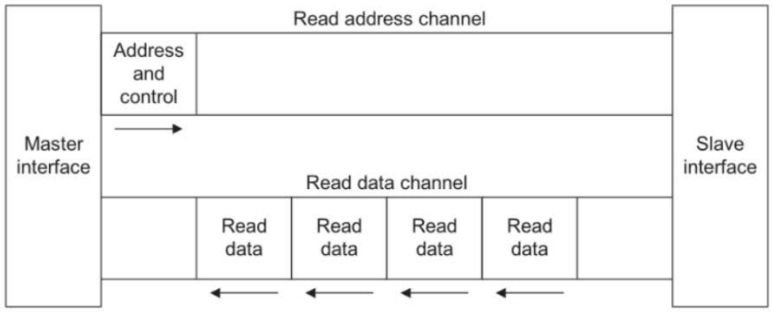
注还需要补充到的是，如果 ID 级是转移指令，而它的前一条指令是 load 指令，又恰好存在数据相关，那么 ID 级由于延迟会无法通过前递获得数据。为此在 ID 级加入一个 br_stall 信号并传给 pre-IF 级，使其在发生这种相关时将 req 拉低，不要发送取指请求，直到相关解除后才发送。也就是真正的跳转信号有效其实是 br_taken 拉高且 br_stall 为低时才会有效。

2、数据 RAM：

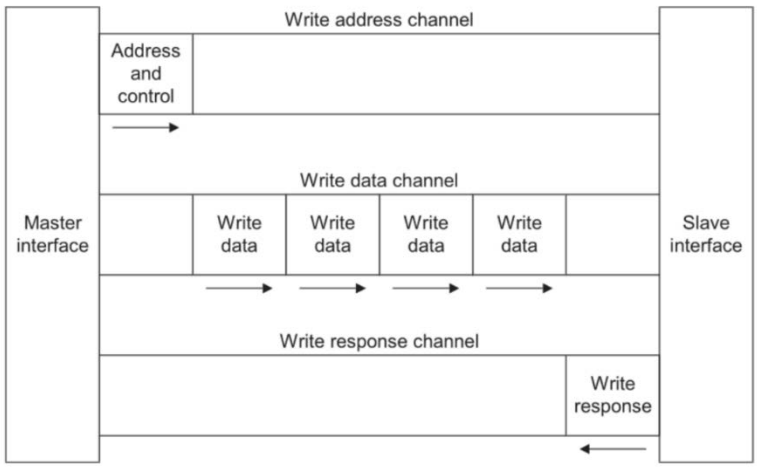
数据 sram 对数据 sram 的访问逻辑的修改与指令 sram 类似，也是将请求和接收分配到两个流水级中完成，EXE 级负责发送请求，MEM 级负责接收返回的信号。这里涉及到的指令只有 load 和 store，并且也不存在指令 sram 中需要取消请求那样的情形，因此设计相对简单一些。与指令 RAM 的处理类似，EXE 级也只在 ms_allowin 为 1 时发请求，这样可以保证发完请求后可以直接进入 MEM 级，不需要考虑数据在 EXE 级就返回的情况。对于 ready_go 信号，如果需要访存，则要等待请求握手成功后才能为 1，否则可以直接沿用之前实验的逻辑，等除法计算完后置为 1。MEM 级也考虑这两种情况，在需要访存时等待 data_ok，否则直接流入下一级。由于在目前的设计中，WB 级的 ready_go 始终为 1，这导致 ws_allowin 也始终为 1，因此 MEM 级拿到数据后一定能流入下一级，不需要像 IF 级一样用寄存器存放读出的数据。

Exp15&exp16:

在类 SRAM-AXI 转接桥中，设计四个状态机，分别处理读事务和写事务。设计详情放在转接桥模块中，在此不再赘述。



图表 3 读频道



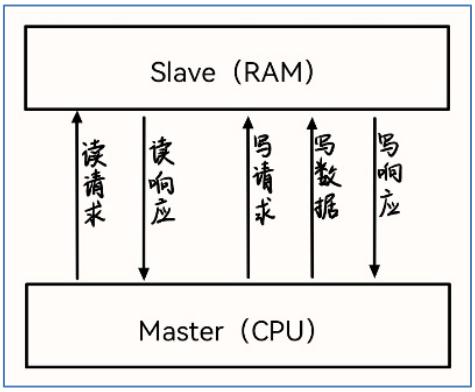
图表 4 写频道

在 Sram 或者类 Sram 中，读写频道是合并在一起的，要么读要么写，两者不能并行，但是在 AXI 总线协议中，实现了读写频道的分离。

（二）重要模块设计：转接桥模块

1、工作原理

前文已述，AXI 总线协议上将传输通道分为五个：读请求、读响应、写请求、写数据、写响应。之所以写事务要在写数据的基础上多出一个写响应通道，是为了防备写后读相关。在 AXI 总线上，可能会出现主方先发出的写地址和写数据被堵在通道上，而后发出的读请求畅通无阻地传输过去，导致从方先看到了读请求，从而返回一个错误的旧值。由此引入写响应，发出写响应表示从方已经收到写数据，主方在这之后发送读请求就是安全的。



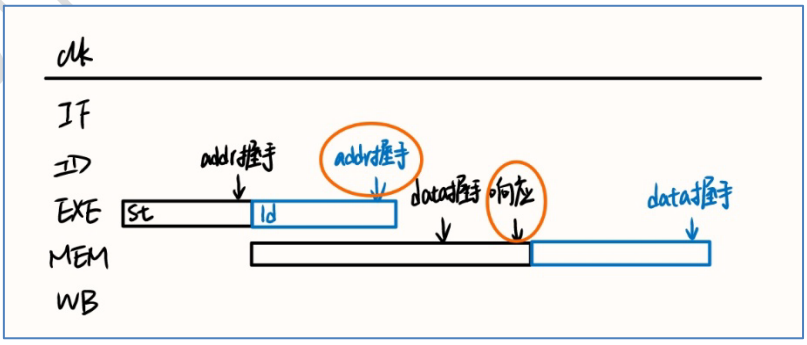
我们采取讲义的建议，将控制 AXI 读写的状态机分为四个：读请求通道一个，读响应通道一个，写请求和写数据通道共用一个，写响应通道一个。

我们设计状态机的大致思路是，当没有相应操作的时候，状态机处于复位（RST）状态；当收到请求时，转移到开始（START）状态；当信号握手时，转移到结束（END）状态。在此基础上，根据具体情况增删修改。

在详细介绍每个状态机之前，要先提一下两处特殊的设计。

其一是计数器。类 SRAM 接口将取指和访存分开，而 AXI 接口将二者不做区分。因此可能会出现这种情况：当一条取指的读事务刚刚请求握手、尚未数据握手的时候，就有一条访存的读事务又请求握手了。此时读请求的状态机的状态尚未恢复到 RST，就又要处理新的请求握手，也就是需要从 END 回到 START。因此很有必要维护 1bit 的计数器，用来记录已经请求握手但尚未数据握手的指令条数，以指示状态机的状态转移。

其二是写后读相关的阻塞。由于之前的访存操作能够在发出请求的下一拍读/写出结果，因此不涉及到数据相关。但现在发出访存请求之后，需要等待一个不定期的延迟才能读/写出结果，可能会存在这种情况： st 指令已经地址握手但尚未响应握手，此时下一条 ld 指令发出请求，如果请求的是同一个地址，就可能会产生冲突，读取的是该地址的旧值而非刚刚写进去的数据，如下图所示。此时应该先阻塞后者，令前者的事务全部执行完毕后，再让后者继续进行。



图表 5 写后读阻塞示例

阻塞的实现方法是，设计一个触发器 awaddr_block，记录下当前写事务的访存地址，直到响应到达后才清零。当有一个读请求时，将 awaddr_block 与读请求的类 SRAM 接口地址 data_sram_addr 比较，若相等说明发生写后读相关，让读请求在 CHECK 状态循环等待，直到响应到达，awaddr_block 清零，不再满足地址相等的条件，让读请

求继续进行。

然而实际上，似乎是因为 st 指令紧接 ld 指令对同一地址操作的情况本来就很少发生，测试中也没有涉及。

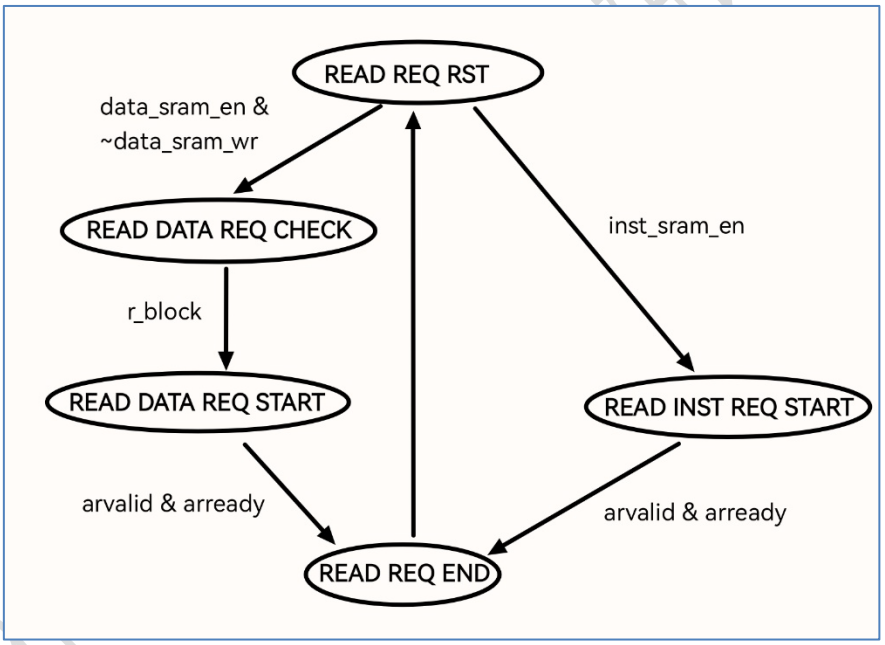
【读请求状态机】

由于取指和访存均会涉及到读操作，因此读请求状态机需要同时考虑到二者。再由于访存可能会涉及到写后读相关，因此需要添加一个 CHECK 状态用来等待写响应。

当没有读请求任务的时候，读请求的状态机会一直处于 RST 状态。

当收到一个取指请求或者访存读请求的时候，状态机会分别转移到相应状态。如果是取指请求，会直接转移到 START 状态；但如果是访存读请求，由于会涉及到写后读相关，会先转移到 CHECK 状态。访存读请求如果检查不存在写后读相关，会转移到读数据 START 状态；如果检查存在写后读相关，就要一直等到写响应到来之后才会转移到读数据开始状态。

当 arvalid & arready，即读请求握手之后，读请求任务结束，进入结束状态，再下一拍回到复位状态。



图表 6 读请求状态转移图

上面讨论了状态机的转移方式，下面讨论地址等变量应该如何随着状态转移发送。

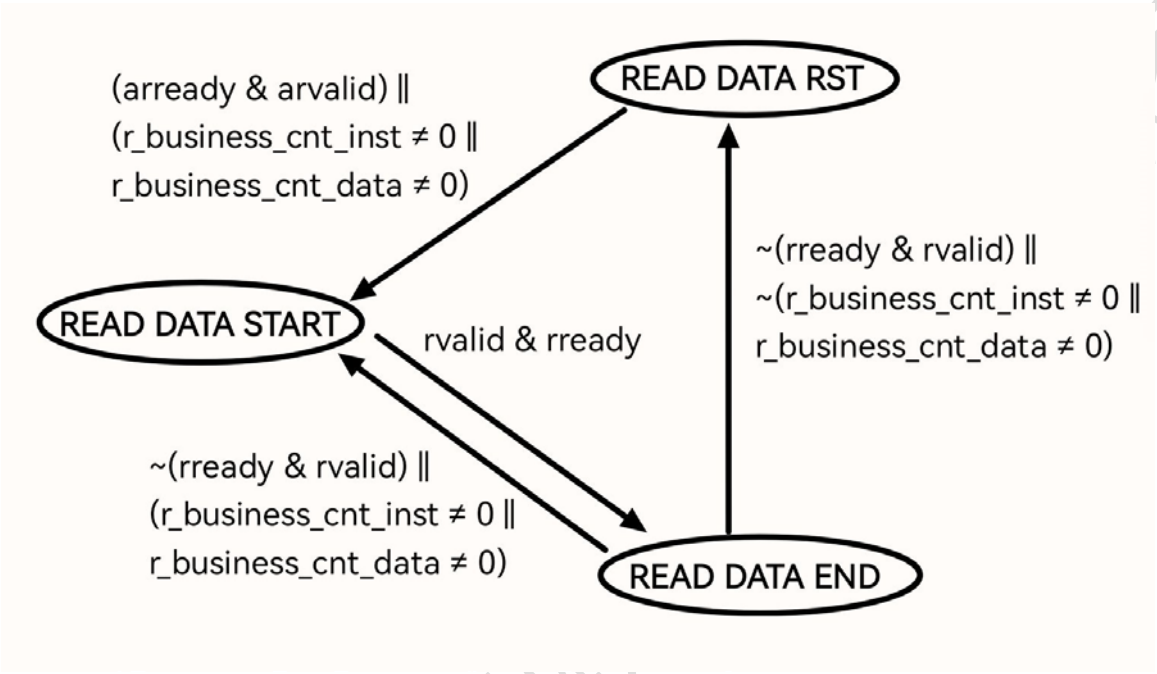
在读请求的时候，一共需要考虑四个 wire 型变量：arid、araddr、arsize、arvalid。这四个变量分别有一个自己的 reg 型变量用于在时序逻辑中非阻塞赋值。

当前或下一个状态是 START 时，分别给 arid、araddr、arsize 的 reg 型变量赋值（数据的 arid 为 1，指令的 arid 为 0）。之所以下一个状态为 START 的时候就提早赋值，就是因为非阻塞赋值要等到下一个时钟上升沿来到时才能真正赋值，只有这样才能让状态为 START 时地址等变量就已经有效。

当前指令为 START 时，拉高 arvalid 的 reg 型变量；当 arready 拉高时，也就是握手时，拉低 arvalid 的 reg 型变量。

【读响应状态机】

计数器用来计数请求握手但尚未返回数据的读事务个数。只有当读请求握手且两个读计数器不全为零的时候，才会从RST进入START状态。每当读数据握手，就会进入END状态。如果计数器尚未清零，或仍有读数据握手，就会在 START 和 END 之间循环往复，直到计数器全为零，暂时不再有读任务，才回到 RST 状态。



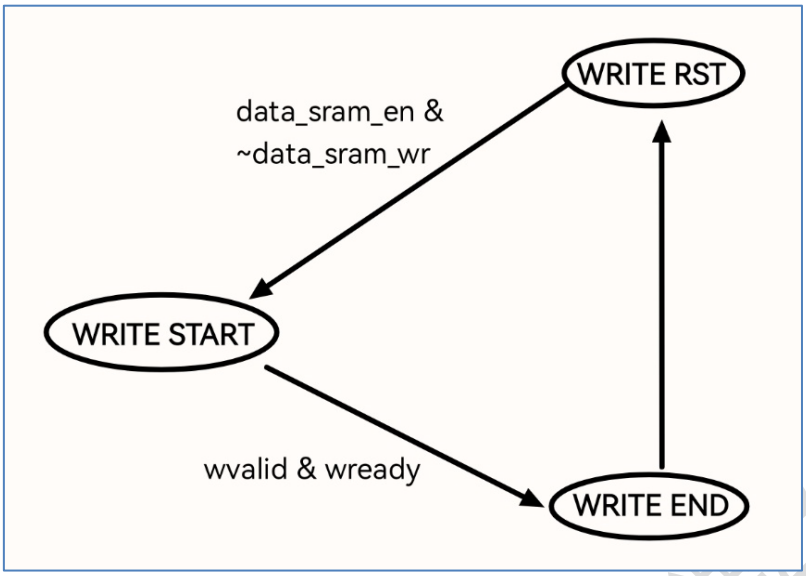
图表 7 读响应状态转移图

计数器如何维护自然也就清晰了：当请求和数据同时握手的时候，计数器不变；当请求握手的时候，计数器加 1；当数据握手的时候，计数器减 1。

【写请求+写数据状态机】

由于只有访存会涉及到写操作，因此该状态机只需要考虑访存，不需要考虑取指，这一点和读请求状态机有所区别。

还有一个和读请求状态机不同的点就是，写请求和写数据放在一个状态机里了，也就是 START 和 END 两个状态之间，不仅完成了写请求的握手，还完成了写数据的握手。因此需要设计一个 write_pending 变量，使之在写请求握手之后直到 END 状态一直拉高，以防写请求再次拉高。



图表 8 写请求+写数据状态转移图

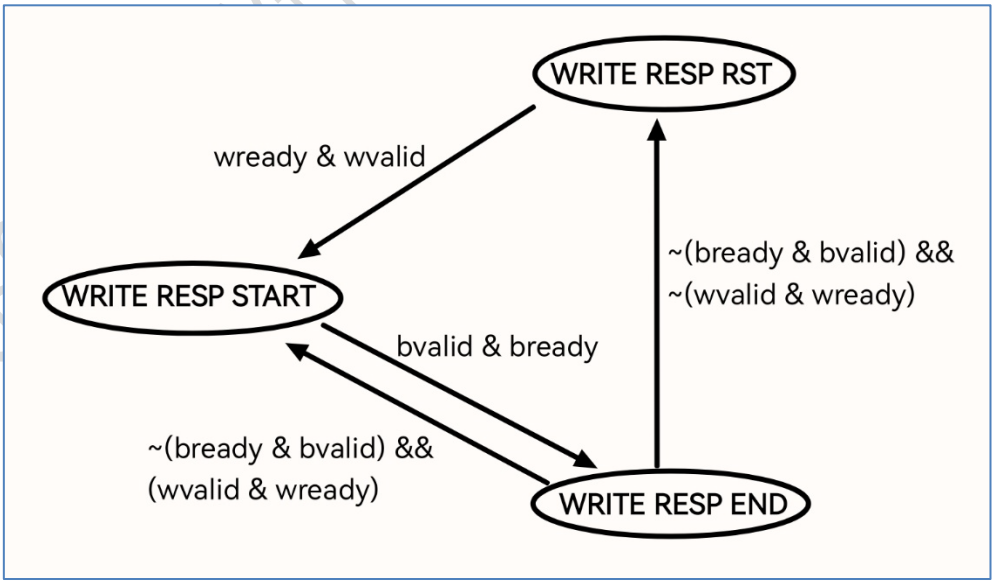
在写请求的时候，一共需要考虑六个 wire 型变量：awaddr、awsize、awvalid、wdata、wstrb、wvalid。这三个变量分别有一个自己的 reg 型变量用于非阻塞赋值。

当前或下一个状态是数据读请求开始或指令读请求开始时，分别给 awaddr、arsize、wdata、wstrb 的 reg 型变量赋值。

当前指令为写开始时，给 awvalid、wvalid 的 reg 型变量赋值为 1；当 awready 拉高时，也就是握手时，给 arvalid 的 reg 型变量赋值为 0；当 wready 拉高时，也就是握手时，给 wvalid 的 reg 型变量赋值为 0。

【写响应状态机】

写响应状态机与读响应状态机的逻辑基本一致，在此不再赘述。



图表 9 写响应状态转移图

2、接口定义

名称	方向	位宽	功能描述
clk	IN	1	时钟信号

名称	方向	位宽	功能描述
resetn	IN	1	复位信号
arid	OUT	4	地址读请求标识
araddr	OUT	32	地址读请求地址
arlen	OUT	8	地址读请求长度
arsize	OUT	3	地址读请求大小
arburst	OUT	2	地址读请求突发类型
arlock	OUT	1	地址读请求锁定
arcache	OUT	3	地址读请求缓存
arprot	OUT	2	地址读请求保护
arvalid	OUT	4	地址读请求有效
arready	IN	1	地址读请求准备
rid	IN	4	读响应标识
rdata	IN	32	读响应数据
rresp	IN	2	读响应响应类型
rlast	IN	1	读响应最后一个
rvalid	IN	1	读响应有效
rready	OUT	1	读响应准备
awid	OUT	4	地址写请求标识
awaddr	OUT	32	地址写请求地址
awlen	OUT	8	地址写请求长度
awsize	OUT	3	地址写请求大小
awburst	OUT	2	地址写请求突发类型
awlock	OUT	1	地址写请求锁定
awcache	OUT	3	地址写请求缓存
awprot	OUT	2	地址写请求保护
awvalid	OUT	4	地址写请求有效
awready	IN	1	地址写请求准备
wid	OUT	4	写数据标识
wdata	OUT	32	写数据
wstrb	OUT	3	写数据写使能
wlast	OUT	1	写数据最后一个
wvalid	OUT	1	写数据有效
wready	IN	1	写数据准备
bid	IN	4	写响应标识
bresp	IN	2	写响应响应类型
bvalid	IN	1	写响应有效
bready	OUT	1	写响应准备
inst_sram_en	IN	1	指令 SRAM 使能
inst_sram_wr	IN	1	指令 SRAM 写使能

名称	方向	位宽	功能描述
inst_sram_size	IN	2	指令 SRAM 大小
inst_sram_wstrb	IN	4	指令 SRAM 写数据使能
inst_sram_addr	IN	32	指令 SRAM 地址
inst_sram_wdata	IN	32	指令 SRAM 写数据
inst_sram_rdata	OUT	32	指令 SRAM 读数据
inst_sram_addr_ok	OUT	1	指令 SRAM 地址有效
inst_sram_data_ok	OUT	1	指令 SRAM 数据有效
data_sram_en	IN	1	数据 SRAM 使能
data_sram_wr	IN	1	数据 SRAM 写使能
data_sram_wstrb	IN	2	数据 SRAM 写数据使能
data_sram_size	IN	4	数据 SRAM 大小
data_sram_addr	IN	32	数据 SRAM 地址
data_sram_wdata	IN	32	数据 SRAM 写数据
data_sram_rdata	OUT	32	数据 SRAM 读数据
data_sram_addr_ok	OUT	1	数据 SRAM 地址有效
data_sram_data_ok	OUT	1	数据 SRAM 数据有效

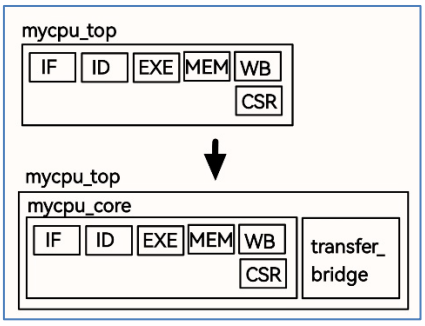
3、功能描述

做类 SRAM 总线和 AXI 总线的相互转换，使得 CPU 对外表现为 AXI 接口。

（三） 重要模块 2 设计：mycpu_top 模块

1、工作原理

如下图所示，原先的 CPU 顶层模块直接封装了五个流水级模块和 CSR 模块，如今即将新增一个转接桥模块。由于转接桥是直接与原先 CPU 顶层模块的接口交互的，因此我们不采用直接将转接桥模块也封装进 CPU 顶层模块。为了封装的简洁和直观，我们将原先的 CPU 顶层模块改写为 mycpu_core 模块，接口不变，与转接桥模块 transfer_bridge 一同封装在 CPU 顶层模块 mycpu_top 下。



图表 10 顶层模块示意图

2、接口定义

名称	方向	位宽	功能描述
aclk	IN	1	时钟
aresetn	IN	1	复位
arid	OUT	4	读请求 ID
araddr	OUT	32	读请求地址

名称	方向	位宽	功能描述
arlen	OUT	8	读请求长度
arsize	OUT	3	读请求大小
arburst	OUT	2	读请求突发类型
arlock	OUT	2	读请求锁定
arcache	OUT	4	读请求缓存
arprot	OUT	3	读请求保护
arvalid	OUT	1	读请求有效
arready	IN	1	读请求就绪
rid	IN	4	读响应 ID
rdata	IN	32	读响应数据
rresp	IN	2	读响应响应类型
rlast	IN	1	读响应最后一个
rvalid	IN	1	读响应有效
rready	OUT	1	读响应就绪
awid	OUT	4	写请求 ID
awaddr	OUT	32	写请求地址
awlen	OUT	8	写请求长度
awsize	OUT	3	写请求大小
awburst	OUT	2	写请求突发类型
awlock	OUT	2	写请求锁定
awcache	OUT	4	写请求缓存
awprot	OUT	3	写请求保护
awvalid	OUT	1	写请求有效
awready	IN	1	写请求就绪
wid	OUT	4	写数据 ID
wdata	OUT	32	写数据
wstrb	OUT	4	写数据字节掩码
wlast	OUT	1	写数据最后一个
wvalid	OUT	1	写数据有效
wready	IN	1	写数据就绪
bid	IN	4	写响应 ID
bresp	IN	2	写响应响应类型
bvalid	IN	1	写响应有效
bready	OUT	1	写响应就绪
debug_wb_pc	OUT	32	调试接口 PC
debug_wb_rf_we	OUT	4	调试接口寄存器文件写使能
debug_wb_rf_wnum	OUT	5	调试接口寄存器文件写地址
debug_wb_rf_wdata	OUT	32	调试接口寄存器文件写数据

3、功能描述

作为 CPU 顶层模块，封装 mycpu_core（五级流水模块）和 transfer_bridge（转接桥模块），使 CPU 对外表现

外 AXI 接口。

三、实验过程（50%）

（一）实验流水账

exp14 实验流水账

- 2023/11/12 15:00 开始撰写 exp14 实验部分代码。
- 2023/11/13 9:00　调试结束，exp14 仿真通过，准备上板验证。
- 2023/11/13 21:00 上板验证通过，　exp14 结束。

exp15 & exp16 实验流水账

- 2023/11/17 15:00 开始撰写 exp15 实验部分代码
- 2023/11/19 9:00　调试结束，exp15 仿真通过，上板验证通过，exp15 结束。
- 2023/11/27 14:00 重新完善代码，上板验证通过。

（二）错误记录

Exp14:

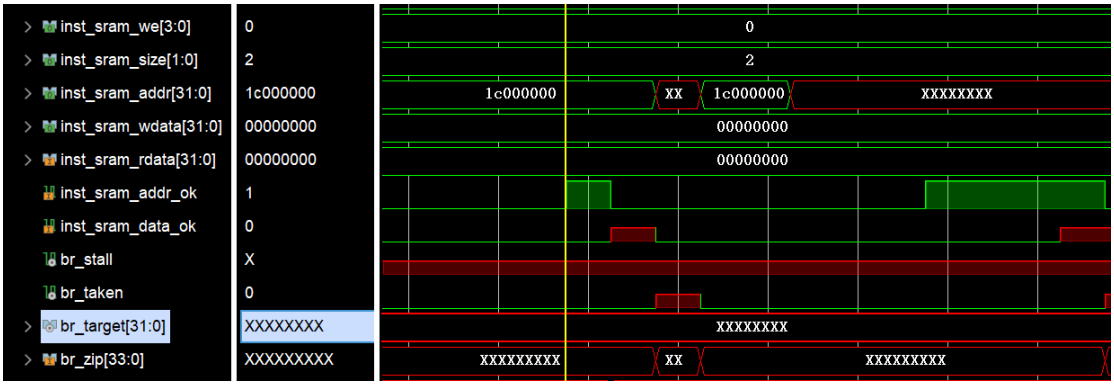
1、错误 1：典中典之 debug_pc 为 0xxxxxxxxx 不定态

（1）错误现象

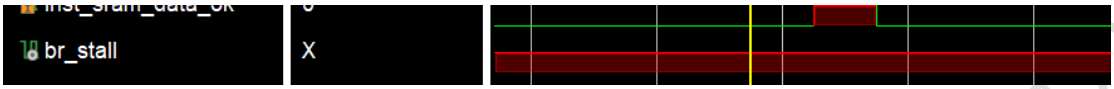
```
[3522000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
[3532000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
[3542000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
[3552000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
[3562000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
[3572000 ns] Test is running, debug_wb_pc = 0xxxxxxxxx
```

（2）分析定位过程

关键信号均为不定态，导致无法继续取指



继续追查，br_stall 为不定态，但是理论上 br_stall 此时应该是拉低



查找 IDstage 中的赋值逻辑，br_stall 主要是由 EXE 和 MEM 阶段前递来的 block 信号决定，

```
assign es_blk = es_mem_block && exe_conflict_rj||exe_conflict_rkd;
assign ms_blk = ms_mem_block && mem_conflict_rj||mem_conflict_rkd;

assign br_stall = branch && (es_blk || ms_blk);
```

继续寻找 es_mem_block 和 ms_mem_block 信号，发现 ms_mem_block 信号拉低，但是 es_mem_block 信号为不定态，也就导致了最终 br_stall 信号为不定态，而原因观察 es_mem_block 信号的赋值就很显然了。另外，对于 es_blk 和 ms_blk 的信号赋值也是同样的问题。

（3）错误原因

```
assign es_mem_block = es_res_from_mem || (es_csr_re | es_csr_we) & es_valid;
```

这串赋值少了一个括号，出现错误，正确的逻辑应该是：

```
assign es_mem_block = (es_res_from_mem || (es_csr_re | es_csr_we) )& es_valid;

assign es_blk = es_mem_block && (exe_conflict_rj||exe_conflict_rkd);
assign ms_blk = ms_mem_block && (mem_conflict_rj||mem_conflict_rkd);
```

（4）修正效果

修改后仿真正常进行。

2、错误 2：传错了 pc 的值

（1）错误现象

```
------------------------------------------------------------
[   2167 ns] Error!!!
    reference: PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
    mycpu    : PC = 0x1c000004, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
------------------------------------------------------------
```

（2）分析定位过程

（3）错误原因

检查 fs_pc 的赋值逻辑，其应当在每一个时钟上升沿更新为 nextpc 中的值，由此推断应该是 nextpc 出现问题。波形中看到 nextpc 并未及时赋值给 fs_pc.

（4）修正效果

检查 fs_pc 的更新逻辑，发现其在 ds_allowin 拉高是只能是在 fs_ready_go 为真时才能更新，但这实际上晚了一拍，因为 fs_ready_go 是在传指令给 IF 后才会更新，但实际上 fs_pc 的值已经变成了下一拍新的 pc 值了。

```
always @(posedge clk) begin
 if(fs_ready_go & ds_allowin) begin
    fs_pc <= nextpc;
  end
end
```

正确的赋值逻辑应该如下：

当 to_fs_valid 拉高后即可更新 fs_pc 的值。

```
always @(posedge clk) begin
 if(to_fs_valid & ds_allowin) begin
    fs_pc <= nextpc;
  end
end
```

3、错误 3：右移指令的 PC 出错

（1）错误现象

```
[  60367 ns] Error!!!
    reference: PC = 0x1c0102bc, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x00000000
    mycpu    : PC = 0x1c0102b8, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x00000000
```

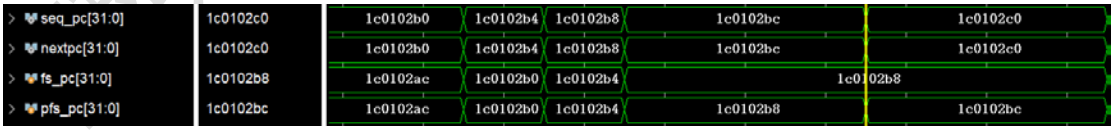
该指令是一条正常的左移指令，错误的 pc 对应的则是异或指令，但是可以看到写地址和写数据均正确，同样是 pc 出现错误。

（2）分析定位过程

```
1c0102b0:   02bffdef    addi.w  $r15,$r15,-1(0xfff)
1c0102b4:   2880018e    ld.w    $r14,$r12,0
1c0102b8:   0015b5ce    xor $r14,$r14,$r13
1c0102bc:   0040a5ce    slli.w  $r14,$r14,0x9
1c0102c0:   0012b9f0    sltu    $r16,$r15,$r14
```

查看仿真波形：

发现 fs_pc 和 next_pc 没有同步更新,观察指令发现其很明显的特征是出现了前递，并且 MEM 阶段对应的正是 ld 取数指令，因此猜测是因为 ID 中处理 id_allowin 信号不当造成的.



这是由于 ds_ready_go 判定太严格导致的，然而实际上如果是 ld 指令，理应会在此时拉低，因为要进行数据的取出，因此问题不应该出现在 ds_allowin 上

（3）错误原因

转念一想，怎么能跟 ds_allowin 有关呢，只需要 fs_allowin 拉高就可以更新 fs_pc 的值了，发现这是改完 bug2 后的纰漏。(`^`>)!

（4）修正效果

修改 ds_ready_go 的逻辑

```
assign ds_ready_go    = ~(      exe_rf_we && (es_block | es_csr_re |es_mem_block ) && (exe_conflict_rj || exe_conflict_rkd)     ||
                                mem_rf_we && (ms_csr_re | ms_mem_block)           && (mem_conflict_rj || mem_conflict_rkd)     ) ||ds_reflush;
```

4、错误 4：除法指令出错

（1）错误现象

```
[4485327 ns] Error!!!
    reference: PC = 0x1c04dca0, wb_rf_wnum = 0x0f, wb_rf_wdata = 0x00000003
    mycpu    : PC = 0x1c04dca0, wb_rf_wnum = 0x0f, wb_rf_wdata = 0x00000029
------------------------------------------------------------
```

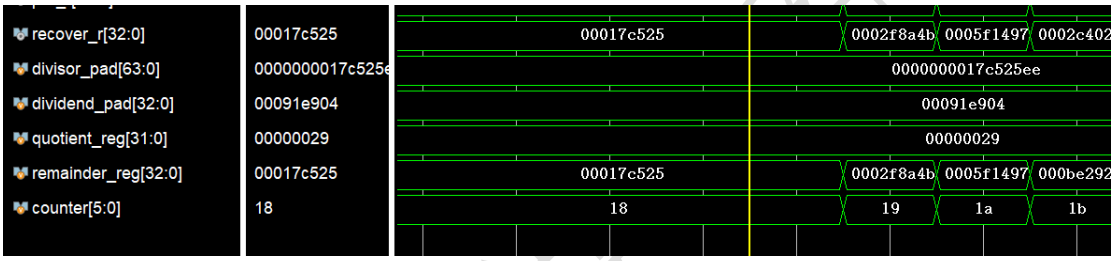
出错地址是一条除法指令？？？

（2）分析定位过程

```
63932    1c04dc9c:    0386e9ad    ori $r13,$r13,0x1
63933    1c04dca0:    0020358f    div.w   $r15,$r12,
63934    1c04dca4:    03800c10    ori $r16,$r0,0x3
```

发现是除法器中，counter 在除法器开始计算时未设置为 0，导致并未计算完毕就输出了结果。原因是上一次除法运算时，未算出结果就关闭了除法器，导致 counter 计数器残留

（3）错误原因

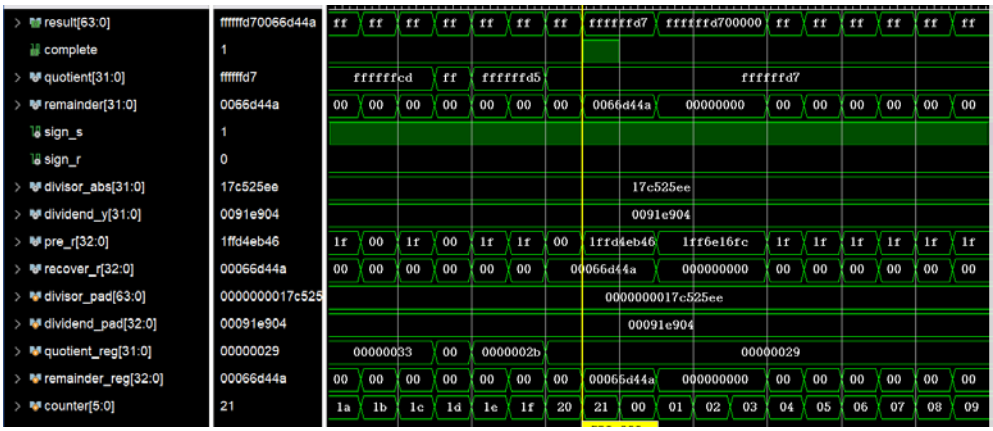


在波形中可以看到 div_en 突然拉低，猜测这可能是 es_ready_go 信号出现问题导致寄存器中数据被刷新换成了新的指令，导致传给 alu 的指令更新 div_en 拉低，但实际上并不应该在 div 运算未完毕后就刷新寄存器。

原因是 ds2es_valid 在除法开始时奇怪的拉高，导致 bus 寄存器中 值更新从而 div_en 拉低判断除法结束，但实际上并未结束。

寻找上一次除法运算，发现其并未算完结果，也是直接就被中断，但是由于其商刚好对应参考结果，所以并未报错，但是这里的逻辑并不正确。查看除法器的波形，发现

`1c04dc84:  0020358f  div.w  $r15,$r12,$r13` 这条指令被连续执行了两次，



原因是 ds2es_valid 的信号一直未拉高，导致 ds2es_bus 寄存器中的值始终未更新，所以除法器判断是一条新的指令继续进行计算直到下一次 ds2es_bus 拉高为止。因此原因很显然，就是在除法器 complete 信号只考虑到计数器达到了 33 次，而且一到 33 次就立即刷新计数器，但实际上如果此时 div_en 信号仍为开启，应当保持原来的结果。

（4）修正效果

修改除法器中计数器的更新逻辑为：

```
always @(posedge clk) begin
    if(~resetn) begin
        counter <= 6'b0;
    end
    else if(div_en) begin
        if(complete)
            counter <= counter;
        else
            counter <= counter + 1'b1;
        end
    else begin
            counter <= 6'b0;
        end
end
```

Exp14 时序总结：

时序情况：

加入类 AXI 总线设计后，时序变得更差了，究其根本是因为一方面加入了更多了控制信号，另一方面 AXI 总线的握手信号要求更加严格的时延界限。但是考虑到这个时延貌似也没有差得太离谱，一切就等上板测试再做说法！

上板测试意外地顺利，并未出现上板不过的情况，测试了很多种子，均正常通过。Exp14 大胜利！

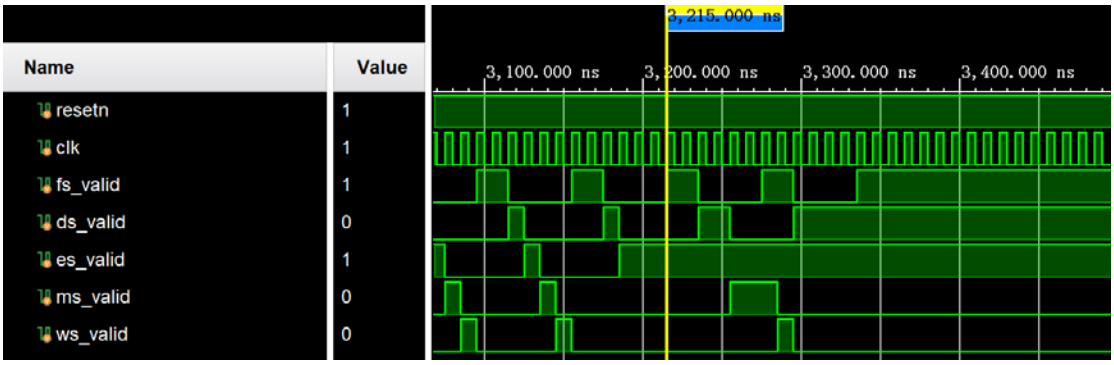
Exp15：

5、错误 5：写数据+写请求状态机设计有缺陷

（1）错误现象

```
Test begin!
         [  22000 ns] Test is running, debug_wb_pc = 0x1c01003c
         [  32000 ns] Test is running, debug_wb_pc = 0x1c01003c
         [  42000 ns] Test is running, debug_wb_pc = 0x1c01003c
         [  52000 ns] Test is running, debug_wb_pc = 0x1c01003c
```

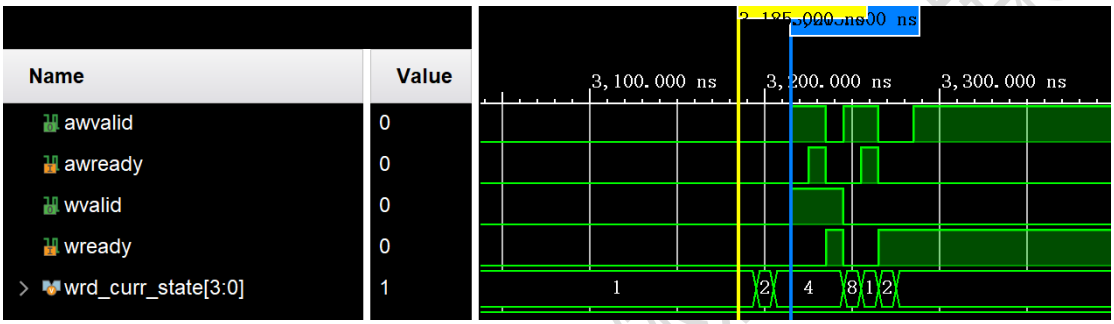
指令卡在 0x1c01003c 一直执行。



valid 信号在这之后一直错误拉高。

（2）分析定位过程

观察到出错指令是测试中出现的第一条 st 指令，st 涉及访存写入，判断可能是写事务相关的状态机出错。调出写事务相关的握手信号查看，确实一直异常拉高。



跟踪这几次 awvalid 的拉高，发现蓝色光标处的 awvalid 拉高是因为黄色光标处的访存请求 data_sram_en 拉高了，但蓝色光标的下一次拉高却没有任何原因。

回溯 awvalid_reg 的赋值逻辑，发现哪怕写请求已经握过手并且写请求的握手信号已经一并拉低了，但只要写请求+写数据状态机（由于写数据还没握手）还处于 WRITE_START 状态，在目前的赋值逻辑下 awvalid 就会再次“意义不明”地拉高，造成后续错误。

```
always @(posedge clk) begin
    if(reset | awready) // until slaver returns awready
        awvalid_reg <= 1'b0;
    else if(wrd_curr_state == WRITE_START)
        awvalid_reg <= 1'b1;
end
```

（3）错误原因

写请求+写数据状态机设计有缺陷，在 WRITE_SRART 状态中没有控制住请求信号，使其错误地拉高。

（4）修正效果

因此只要设计一个变量 write_pending，其拉高时表示“在当前 WRITE_START 状态中，写请求已经握手，当前正在等待写数据握手，切勿再拉高 awvalid”，就能解决这个问题。该变量的赋值逻辑是，当写请求握手时拉高，当进入 WRITE_END 状态时拉低，如下图所示。

```
reg write_pending;
always @(posedge clk) begin
    if(reset)
        write_pending <= 1'b0;
    else if(awvalid && awready)
        write_pending <= 1'b1;
    else if(wrd_next_state == WRITE_END)
        write_pending <= 1'b0;
end
```

awvalid_reg 的赋值逻辑需要增添一条：当 write_pending 拉高时赋值为零，如下图所示。

```
always @(posedge clk) begin
    if(reset | awready | write_pending) // until slaver returns awready
        awvalid_reg <= 1'b0;
    else if(wrd_curr_state == WRITE_START)
        awvalid_reg <= 1'b1;
end
```

（5）归纳总结（可选）

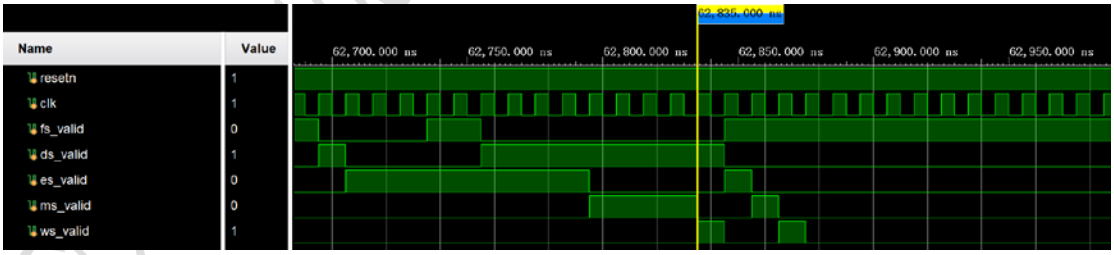
之所以读事务中没有遇到这个错误，是因为写请求与写数据合用一个状态机，而读请求单用一个状态机，请求握手之后直接进入 END 状态，不需要等待数据。

6、错误 6：读数据状态机设计有缺陷

（1）错误现象

```
        [  62000 ns] Test is running, debug_wb_pc = 0x1c034784
----[  62355 ns] Number 8'd01 Functional Test Point PASS!!!
        [  72000 ns] Test is running, debug_wb_pc = 0x1c0102a4
        [  82000 ns] Test is running, debug_wb_pc = 0x1c0102a4
        [  92000 ns] Test is running, debug_wb_pc = 0x1c0102a4
        [ 102000 ns] Test is running, debug_wb_pc = 0x1c0102a4
```

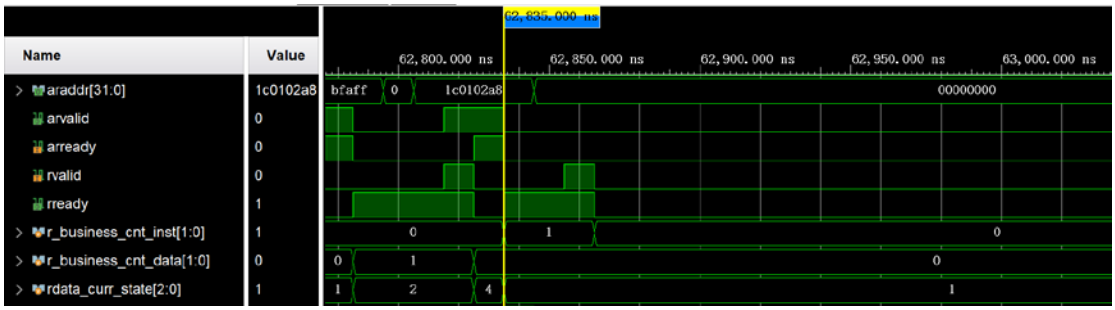
指令卡在 0x1c0102a4 一直执行。



fs_valid 一直拉高，其余流水级的 valid 再也不拉高。

（2）分析定位过程

像是取指附近出的错误，推测错误在读事务的状态机上，与指令部分有关。调出与读事务相关的信号，发现读数据（读响应）状态机会在某一个时刻变为 1 即 RST 状态后，不再变化。



注意到黄色光标之前读指令计数器 r_business_cnt_inst 的值为 0，之后变为 1。也就是说在光标之前的一拍进行了地址握手，在光标之后就存在了一条已经地址握手但尚未收到数据的读事务。可以看到很快就数据握手了，计数器也清零了。但不再有取指请求，因此读数据状态机一直错误地维持在了 RST 状态。

找出 RST 状态机的逻辑，发现当读数据（读响应）状态机处于 RST 状态时，只有地址握手才会导致状态转移到 SRART。这样漏掉了一种情况，也就是现在出错的情况：在状态回到 RST 的当拍，读请求握手，在转移到 RST 状态时计数器已经为 1，却不会转移到 START，就取不到这条指令。取不到这条指令，就不会有下一个读请求进来，导致状态机永远停留在 RST，死锁住。

```
READ_DATA_RST: begin
    if(arready && arvalid) // exists unaddressed read business
        rdata_next_state = READ_DATA_START;
    else
        rdata_next_state = rdata_curr_state;
end
```

（3）错误原因

读数据状态机设计有缺陷，导致若状态转移到 RST 与地址握手同一拍发生，会造成死锁。

（4）修正效果

修改的方法是，在 RST 状态也检测计数器，只要不为 0 就支持转移到 START 状态即可，如下图所示。

```
READ_DATA_RST: begin
    if((arready && arvalid) || r_business_cnt_inst != 2'b0 || r_business_cnt_data != 2'b0)
        rdata_next_state = READ_DATA_START;
    else
        rdata_next_state = rdata_curr_state;
end
```

（5）归纳总结（可选）

之所以写事务不需要考虑种情况，是因为写响应状态机不涉及到取指，因此不会有两个“地址握手但尚未响应的读事务”同时存在，也就压根没有设置计数器。

7、错误 7：清空流水线时出错

（1）错误现象

```
[6808907 ns] Error!!!
    reference: PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000
    mycpu    : PC = 0x1c008110, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000007
```

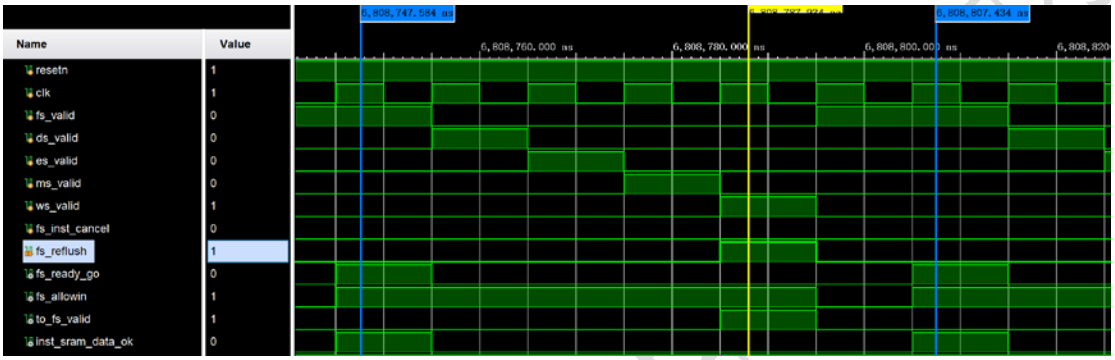
PC 错误，与金标准不符。

（2）分析定位过程

上一条指令是 1c008000，bne，但是 bne 不写回，因此没有金标准；再上一条指令是 1c071d40，syscall，也不写回；再上一条是 1c071d3c，addi.w，写回，是对的，并且可以保证 syscall 确实是应该执行的。

```
1c071d40:       002b0000        syscall 0x0
1c071d44:       5c01133e        bne     $r25,$r30,272(0x110) # 1c071e54 <inst_error>
1c071d48:       14003a0c        lu12i.w $r12,464(0x1d0)
```

从波形可以看到 syscall 的入口地址是 1c008000，那么应该执行那条 lu12i 才对，但是这里虽然 PC 对了，指令码却取的是 syscall 下一条的 bne 的指令码，导致出错。这里错误的原因是 syscall 之后已经发起了 bne 的取指请求，实际上并且没有把它取消掉。



（3）错误原因

异常清空流水线时，如果已经发起的错误请求在异常的当拍正好地址握手，无法取消掉。究其原因，是 fs_inst_cancel 由于非阻塞赋值滞后 fs_reflush 一拍。

（4）修正效果

修改方式就是要让真正的 fs_inst_cancel 信号，由 fs_reflush 和由时序逻辑赋值的中间信号相或而成，确保在清空信号拉高的这一拍，取消信号就已经拉高。

```
assign fs_inst_cancel = fs_reflush | fs_reflush_reg;
```

Exp16：

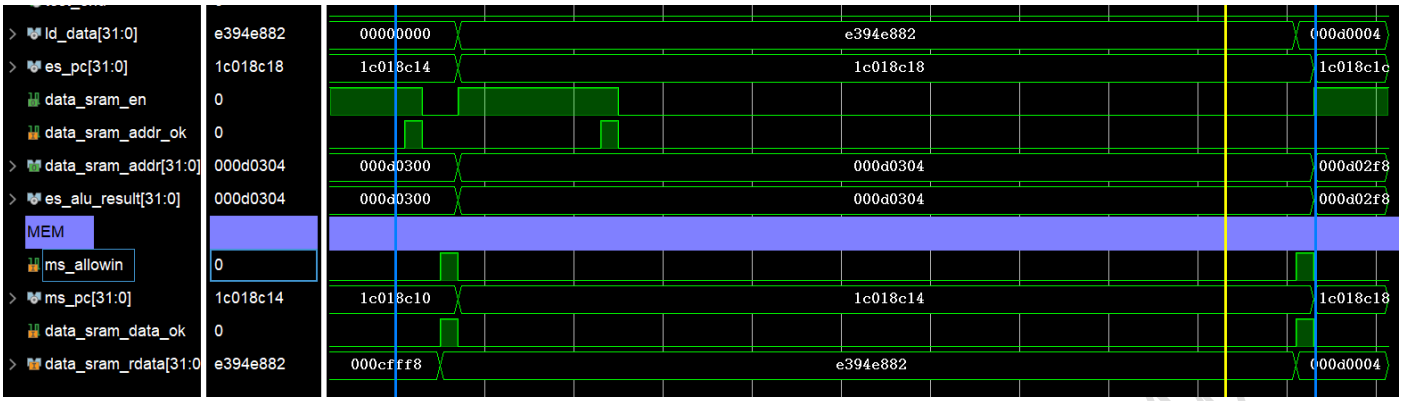
8、错误 8：读请求限制错误

（1）错误现象

```
------------------------------------------------------------
[4369517 ns] Error!!!
    reference: PC = 0x1c018c14, wb_rf_wnum = 0x0a, wb_rf_wdata = 0xe394e882
    mycpu    : PC = 0x1c018c14, wb_rf_wnum = 0x0a, wb_rf_wdata = 0x000d0004
------------------------------------------------------------
```

上板测试后发现种子 16’00FC 没过，于是更改 confreg 中种子，出现上述错误。

（2）分析定位过程



如图在 addr_ok 和 en 拉高时，发出读请求，而后在 data_ok 时读数据返回，同时 ms_allowin 拉高，EXE 阶段的数据进入 MEM 阶段，但在 MEM 阶段数据进入 WB 阶段时出现错误，也就是说，在第一次返回的数据还没进入 WB 阶段时，第二次返回的数据已经将 MEM 阶段的数据刷新，从而使得进入 WB 阶段的数据出现错误。这里看起来像是错了一拍，当请求对应的 data_ok 返回时应当控制数据从 MEM 级流入 WB 级，而不是从 EXE 级流入 MEM 级。那我们如何通过通过书上的建议，这里我们可以在 req 请求上进行限制，即修改数据 RAM 的读写使能信号。

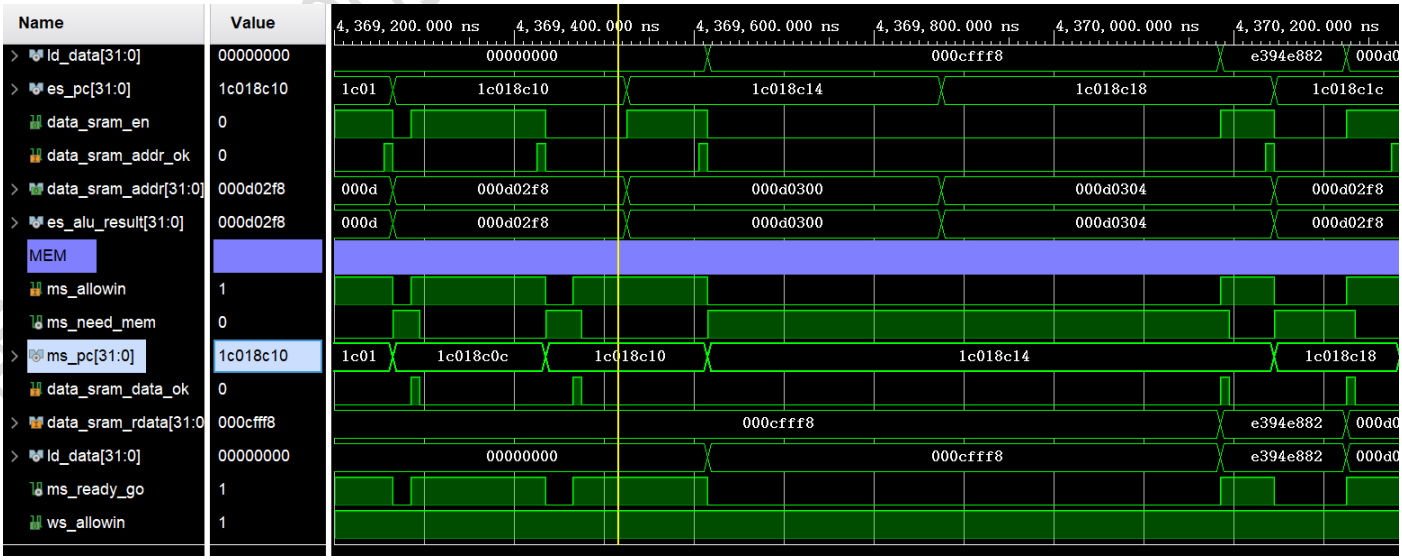
（3）错误原因

```
assign data_sram_en    =  ~es_finish && es_need_mem ;//1'b1;
```

（4）修正效果

```
assign data_sram_en    =  ~es_finish && es_need_mem && ms_allowin;//1'b1;
```

在读写使能信号中加入 ms_allowin，也就说只有当下一级可以流入时才将读请求握手成功的数据自 EXE 级流入 MEM 级，而后当数据返回时（data_ok 拉高），数据从 MEM 级流入 WB 级。也就是原来控制 EXE 级到 MEM 级的为与数据对应的 data_ok，修改之后 data_ok 变成控制 MEM 级到 WB 级，而这样才是 data_ok 正常应该控制的地方。正确修改后的波形图如下



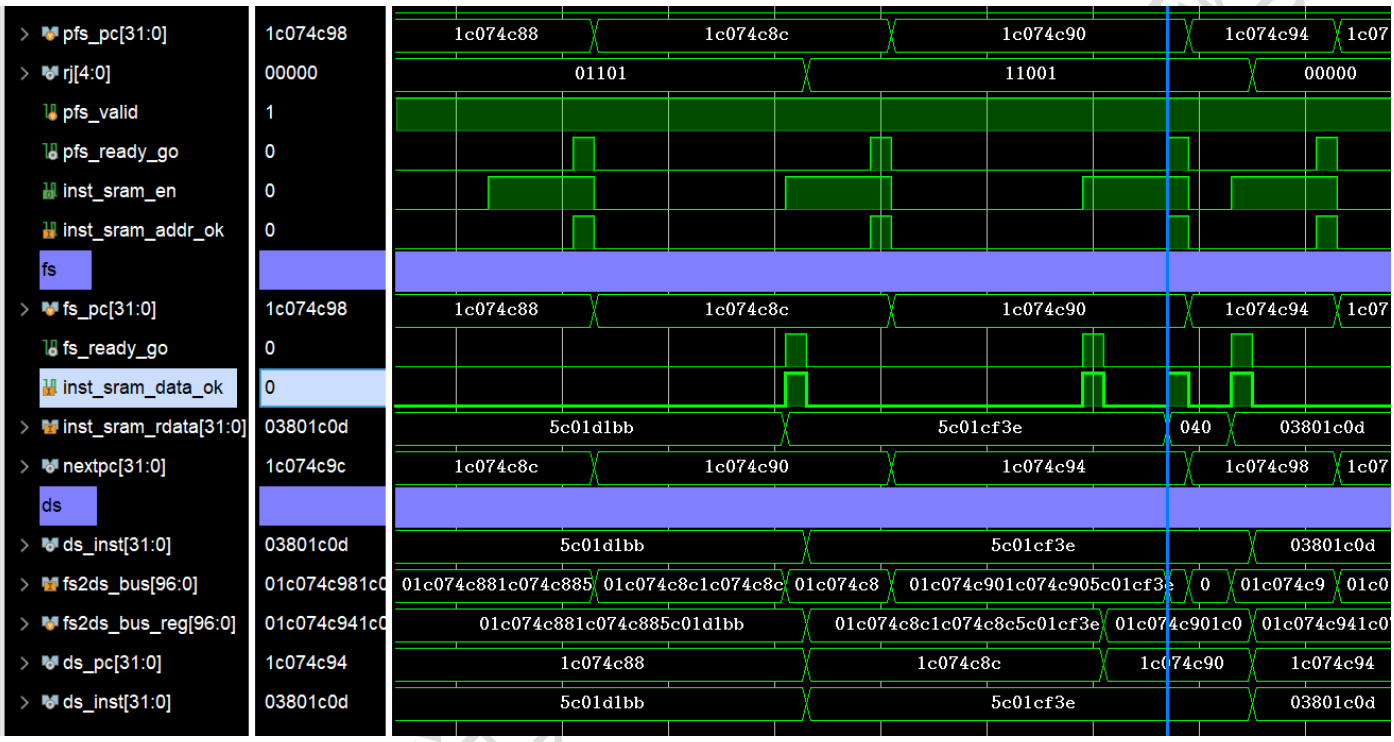
9、错误 9：br_stall 相关调整

（1）错误现象

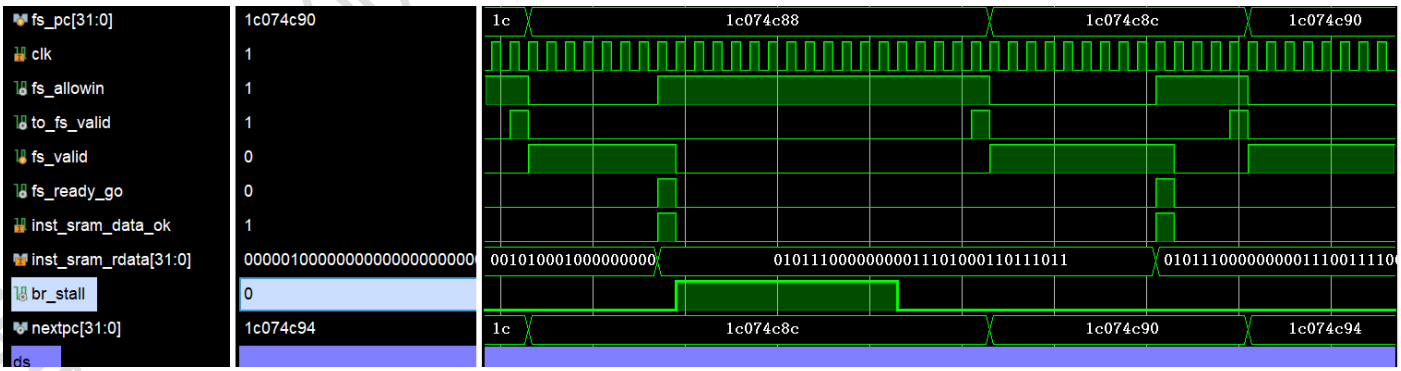
```
[17787757 ns] Error!!!
    reference: PC = 0x1c074c90, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x0000000c
    mycpu    : PC = 0x1c074c94, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000007
```

种子未变，发生错误

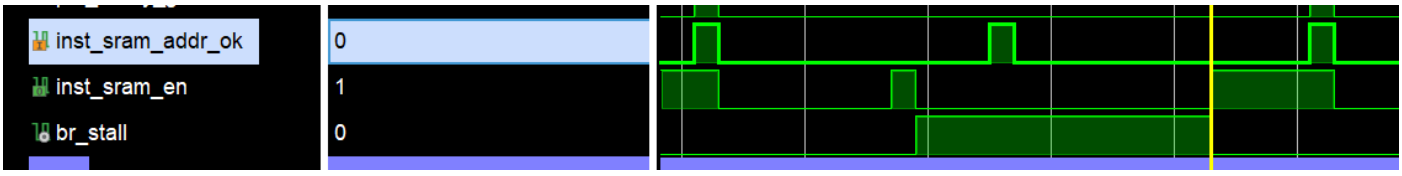
（2）分析定位过程



对于正确的 PC 值，该处指令应为 0x1c074c90，为 CSRRD 指令，但该处 csr_re 没有拉高。往前回溯至 IF 阶段，发现该指令周期中出现了两次 data_ok 拉高，实际上第二次的值是正常值，但我们会取第一次的值。



往前查 br_stall，发现拉高了一次，而对比 en，req 在拉高了一次后遇到 br_stall 拉低，后又再次拉高，被 RAM 视作了两次请求，因此返回了两次值，且值没有改变。



（3）错误原因

```
assign inst_sram_en       = fs_allowin & ~br_stall & pfs_valid & ~pfs_reflush;
```

在出现 br_stall 的时候，拉低了 req 请求，在 br_stall 结束后再次拉高，被视作进行了两次请求。

（4）修正效果

```
assign inst_sram_en       = fs_allowin & pfs_valid & ~pfs_reflush;
```

在出现 br_stall 的时候，不终止请求的发送，而是忽视返回的数据，因此后续再 pfs_reflush 加入 br_stall 的情形，从而在 addr_ok 时忽略后续返回的数据。

```
always @(posedge clk) begin
    if(~resetn)
        pfs_reflush <= 1'b0;
    else if(inst_sram_en && (fs_reflush | br_taken & ~br_stall | (br_stall |
br_stall_reg)& inst_sram_addr_ok )
)
        pfs_reflush <= 1'b1;
    else if(inst_sram_data_ok)
        pfs_reflush <= 1'b0;
end
```

这一部分主要逻辑在于，发出读请求后，由于类 SRAM 总线允许中途更改请求，而 AXI 总线不允许中途更改，因此哪怕错误的请求还没被接收，我们也让其继续发完，然后再阻塞 pre-IF 级，忽视后续返回的指令，再重新发正确的请求。也就是为什么需要 pfs_reflush 信号的原因。

而设计 br_stall 指令的初衷是如果 ID 级是转移指令，而它的前一条指令是 load 指令，又恰好存在数据相关，那么 ID 级由于延迟会无法通过前递获得数据。为此在 ID 级加入一个 br_stall 信号并传给 pre-IF 级，使其在发生这种相关时将 req 拉低，不要发送取指请求，直到相关解除后才发送。

但是现在考虑到因为在指令进入 ID 级、拉高 br_stall 前，它的后一条指令可能已经开始发送取指请求，原本的读请求的判断逻辑是 assign inst_sram_en = fs_allowin & ~br_stall & pfs_valid; 而现在不能仅仅通过在 br_stall 为 1 时拉低 req 来处理 br_stall，而是需要先发完再取消。

现在的 pfs 阶段主要的控制逻辑更改为：

assign pfs_ready_go = inst_sram_en & inst_sram_addr_ok & ~(fs_reflush | br_taken & ~br_stall | br_stall | pfs_reflush);

assign inst_sram_en = fs_allowin & pfs_valid & ~pfs_reflush ;

对于 pfs_ready_go 来讲,当其他流水及遇到异常需要刷新和跳转指令时,信号将会拉低,代表上一条发出的取指指令无效,不会流水到 IF 阶段.

fs_inst_cancel 信号

assign fs_inst_cancel = fs_reflush | fs_reflush_reg | br_taken & ~br_stall | br_taken_reg;

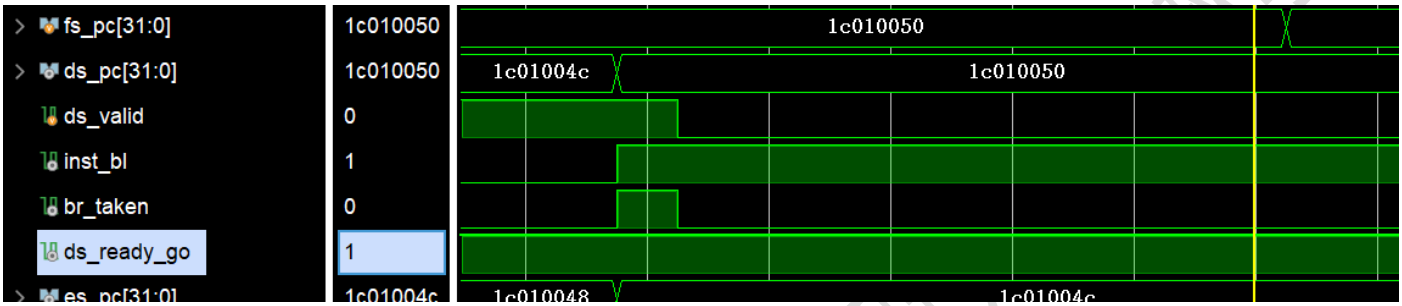
现在 cancel 信号不再需要用一个寄存器存住,因为已经保证了 pfs_stage 发出来的指令都将是有效的,br_stall_reg 也存住已产生的跳转指令请求的信号.

10、错误 10：跳转指令错误

（1）错误现象

```
[   3737 ns] Error!!!
     reference: PC = 0x1c010050, wb_rf_wnum = 0x01, wb_rf_wdata = 0x1c010054
     mycpu    : PC = 0x1c033850, wb_rf_wnum = 0x17, wb_rf_wdata = 0x00000001
```

（2）分析定位过程



这里由于 ds_valid 的写法，导致 ds_valid 在 br_taken 被拉高的下一拍就会被拉高，从而产生问题。

（3）错误原因

```
if (reset || br_taken || ds_reflush) begin
      ds_valid <= 1'b0;
    end else if (ds_allowin) begin
      ds_valid <= fs2ds_valid;
    end
```

这里 br_taken 拉高，则在下一拍如果指令没有流入 EXE 级，则当前指令会被置为无效，因此要删去 br_taken。

（4）修正效果

删去后仿真及上板均通过。

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.498 ns	Worst Hold Slack (WHS):	0.017 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	16803	Total Number of Endpoints:	16803	Total Number of Endpoints:	3688

All user specified timing constraints are met.

原先时序较差时，违约路径大多都是由于要通过组合逻辑进出 RAM，导致一个周期之内的路径太长。本实验中新设计了转接桥与 RAM 做交互，而转接桥中的状态机全部为时序逻辑。这样相当于把原来一条很长的路径“切割”成了小段。时序自然有了很大提升。

四、实验总结（可选）

AXI 总线接口的设计是片上芯片重要的一环， AXI 总线可以支持高性能高频的系统设计，能够简化硬件

的复杂性，连接各种不同的硬件组件，具有很好的兼容性，也可以有效地处理高延迟的存储器访问。当然，第一次接触这样的模块设计，也是需要花很大功夫和时间，不过好在只要理解 AXI 总线设计的内涵，具体实现起来也很能很快摸着门路。但是目前设计的转接桥，尚不支持使用突发传输来传输数据，一些其相关的信号也尚未使用，需要等到 cache 模块设计后继续改进这些模块。

这次实验需要更改随机种子来进行不同的测试，需要先上板查看哪些随机种子出现错误，而后更改仿真文件中的种子，再进行仿真测试，这种 debug 的方式实行起来确实比较麻烦，也还不理解为什么对不同的种子的测试会大不相同的结果。在 debug 中常常就是知道问题出现在哪，但是不知道如何修改，经过很久对整个过程的梳理，才最终知道问题的根源在哪，从而有思路如何修改。

exp14 完成仿真测试通过后进行上板测试，所有的种子测试都是完全没有问题的。然而，在 exp15 实验完成后，就产生了仅仅只有无延迟测种子能够通过，但是其他所有种子（可能）都存在问题的情况。Exp16 时改正了一些实际上出现的纰漏（主要是 AXI 总线传输时不允许修改发送请求的地址而产生的问题），却又出现了其他所有有延迟种子通过，但无延迟测试种子无法通过的情况，对此感觉很有意思，可能需要好好理解一下无延迟和有延迟会对具体哪些模块产生怎样的影响。