# 实验 10 报告

学号：2017K8009922026，2017K8009922032
姓名：康齐瀚，赵鑫浩
箱子号：63

## 一、实验任务（10%）

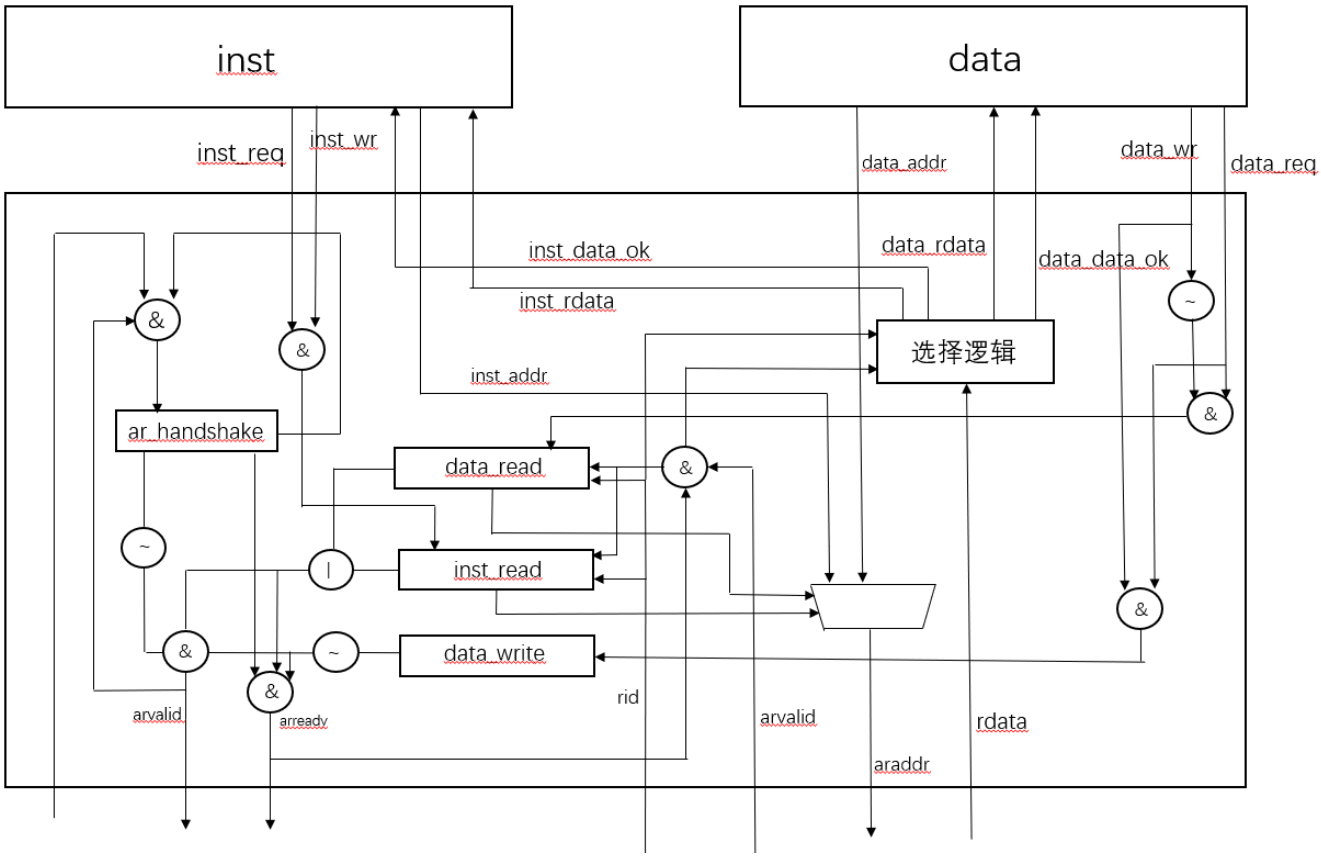
添加类 SRAM-AXI 转接桥，最终能成功通过随机延迟的仿真和上板测试

## 二、实验设计（40%）

### （一）总体设计思路

实现从两个类 SRAM 接口向 AXI RAM 的数据传输的转接桥。

本次实验的设计图如下所示，其中只展示了读数据通道的部分，对于写数据通道的部分，由于二者在处理逻辑上并无太大区别，因此设计也是类似的。




### （二）重要模块 1 设计：read_inst，read_data 模块

## 1、工作原理

从 cpu 中得到 inst_size，inst_addr(读指令),data_size,data_addr(读数据)，从 ram 中读出指令和数据。

## 2、功能描述

首先我们实现了关于 read_inst 的状态机，来表示正在读指令，在做读指令时，不允许其他操作，即同一时刻值进行一种操作，其他操作等待。Read_data 的状态机同理。

```
always@(posedge clk) begin

    if(~resetn) begin
        ar_handshake <= 1'b0;
    end
    else if(arvalid && arready) begin
        ar_handshake <= 1'b1;
    end
    else if(rready && rvalid) begin
        ar_handshake <= 1'b0;
    end

end
```

```
always@(posedge clk) begin
    if(~resetn) begin
        inst_read <= 1'b0;
    end
    else if(~inst_read && inst_req && ~inst_wr && ~data_req && ~data_read) begin
        inst_read <= 1'b1;
    end
    else if(inst_read && rready && rvalid) begin
        inst_read <= 1'b0;
    end
end
```

```
always@(posedge clk) begin
    if(~resetn) begin
        data_read <= 1'b0;
    end
    else if(~data_read && data_req && ~data_wr && ~inst_read && ~data_write)  begin
        data_read <= 1'b1;
    end
    else if(data_read && rready && rvalid) begin
        data_read <= 1'b0;
    end
end
```

之后我们需要得到两个重要的握手信号 arvalid 和 rready 来控制 ar 和 r 通道的握手，在握手的同时，我们需要将地址和数据的 size 存在一个 reg 变量中，并给 AXIRAM addr_ok 和 data_ok 信号来从 RAM 中读出对应地址的数据，最后将读回的数据传回 cpu。

```
assign arid    =     data_read ? 4'h1 :
                     inst_read ? 4'h0 : 4'h2 ;

assign arvalid  =  ( data_read | inst_read ) & ~ar_handshake & ~data_write;
assign rready   =  ( data_read | inst_read ) &  ar_handshake & ~data_write;


assign araddr   =   data_read ? raddr_reg :
                    inst_read ? inst_addr_reg : 32'b0;
```

```
assign arsize   =   data_read ? {1'b0 ,data_read_size} :
                    inst_read ? {1'b0, inst_size_reg } : 3'b0;

assign inst_addr_ok  =  ~inst_read & ~data_write & ~data_read & inst_req & ~inst_wr & ~data_req;


//assign inst_addr_ok  =  ~inst_read & ~data_write & ~data_read & inst_req & ~inst_wr ;
assign inst_data_ok  =  inst_read & rready & rvalid  & (rid == 4'h0)    ;
//assign inst_data_ok  =  rready & rvalid  ;
assign inst_rdata    =  rdata ;
assign data_rdata    =  rdata ;
```

## （三）重要模块 2 设计：write_data 模块

### 1、工作原理

从 cpu 中得到,data_size,data_addr 和将要写入 RAM 的数据 data_wdata，向 ram 中制定位置写入数据。

### 2、功能描述

首先我们需要写出 write_data 的状态机，data_req 和 data_wr 满足写数据条件且 data_read 为 0 时，可以拉高 data_write 信号表示开始写数据。

```
always@(posedge clk) begin
    if(~resetn) begin
        data_write <= 1'b0;
    end
    else if(~data_write && data_req && data_wr && ~data_read) begin
        data_write <= 1'b1;
    end
    else if(data_write && bready && bvalid) begin
        data_write <= 1'b0;
    end
end
```

我们使用 aw_handshake 和 wdata_handshake 两个信号分别表示 aw 和 w 通道的握手开始和结束，并用这两个信号控制两个通道的握手信号。

```
always@(posedge clk) begin
    if(~resetn) begin
        aw_handshake <= 1'b0;
    end
    else if(awvalid && awready) begin
        aw_handshake <= 1'b1;
    end
    else if(bready && bvalid) begin
        aw_handshake <= 1'b0;
    end
end
```

```
always@(posedge clk) begin
    if(~resetn) begin
        wdata_handshake <= 1'b0;
    end
    else if(wvalid && wready) begin
        wdata_handshake <= 1'b1;
    end
    else if(bready && bvalid) begin
        wdata_handshake <= 1'b0;
    end
end
```

```
assign awvalid  =  data_write & ~aw_handshake ;
assign wvalid   =  data_write & ~wdata_handshake ;
assign bready   =  data_write & wdata_handshake ;
```

接着我们考虑需要输出给 RAM 的 data_ok,addr_ok 信号，我们分读写数据两种情况。

```
assign data_write_addr_ok   =   ~data_write & ~data_read & ~inst_read & data_req & data_wr;
assign data_read_addr_ok    =   ~data_read  & ~data_write & data_req & ~data_wr;
assign data_addr_ok         =   data_write_addr_ok | data_read_addr_ok ;

assign data_write_data_ok   =   data_write & bready & bvalid    ;
assign data_read_data_ok    =   data_read  & rready & rvalid  & (rid == 4'h1);
//assign data_read_data_ok    =   data_read  & rready & rvalid ;
assign data_data_ok         =   (data_write & data_write_data_ok) | (data_read & data_read_data_ok) ;
```

# 三、实验过程（50%）

## （一）实验流水账

2019/11/14　13：30～22：00　　写代码，仿真，出错继续调 bug

2019/11/15　 9：00～15：00　　继续修改代码，第一次仿真通过，改变种子后出错

2019/11/15　19：00～21：00　　通过随机种子测试，最终上板成功

## （二）错误记录

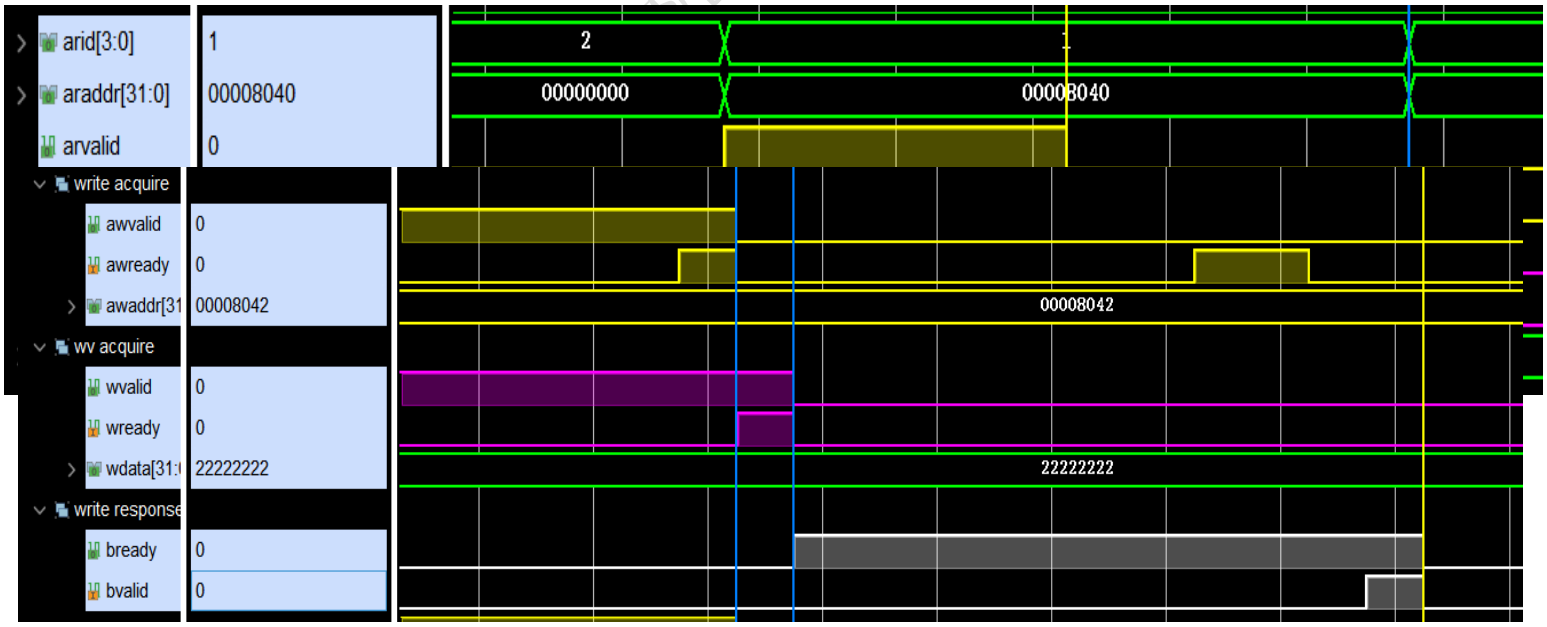
### 1、错误 1：读数据出错

#### （1）错误现象

仿真时控制台信息报错，发现是在四次 data_read 指令时出错。如下图所示：

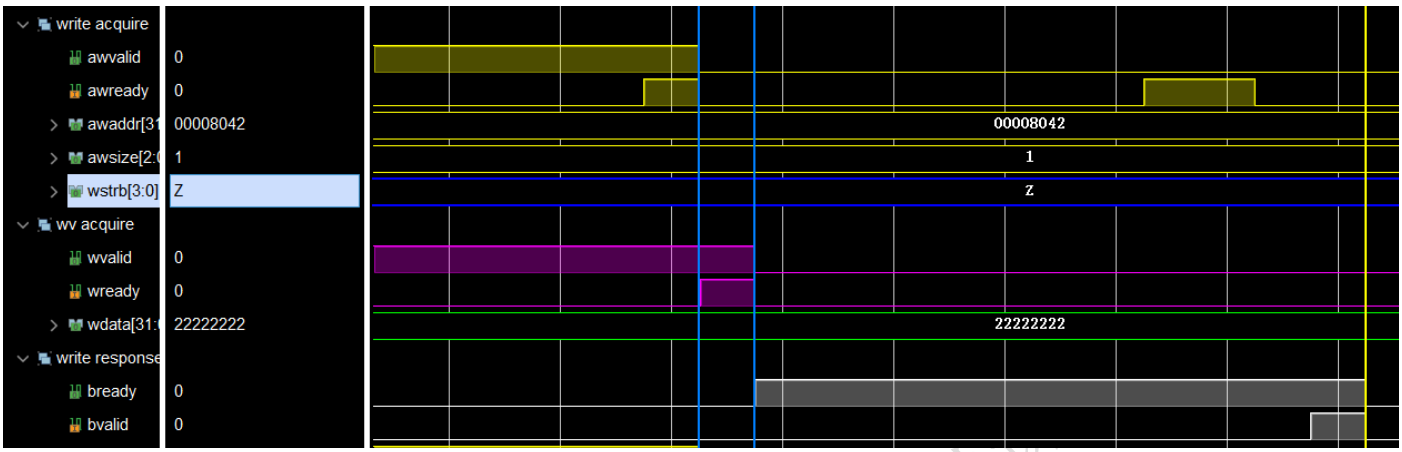
```
[   2815 ns] Fail!!!read from data sram-like, ref_data[31: 0]=32'h22220000, my_data[31: 0]=00000000
[   3145 ns] Fail!!!read from data sram-like, ref_data[23:16]=8'h11, my_data[23:16]=00
[   3275 ns] OK!!!read data 2
[   3355 ns] Fail!!!read from data sram-like, ref_data[31: 0]=32'h11113311, my_data[31: 0]=00000000
[   3685 ns] Fail!!!read from data sram-like, ref_data[31: 0]=32'h55004444, my_data[31: 0]=00000000
```

#### （2）分析定位过程




从图中可以看出在 arvalid 和 arready 信号握手时，传入的读数据地址是 0x00008040。而在 rvalid 和 rready 握手时返回的 rdata 数据是 0x00000000。产生该现象的原因可能是由于最近一次的向该地址写入数据出错了。追踪最近的一次向该地址写入数据如下图所示：

从上图可以看出最近的一次向 0x00008040 写入数据时，写入的数据是 0x22222222,如果该数据被真实写入了，那么读出的数据就恰好是 0x22220000，与 ref_data 相同。因此，出现该错误说明数据并没有被正确写入。进一步考察其他与写数据相关的信号：




可以看出在 awvalid 和 awready 信号握手时，wstrb 信号一直是 Z, 因此 slave 端无法判断写使能字节，导致写入数据失败。

（3）错误原因

wstrb 信号未被正确设置，导致写入数据时出错

（4）修正效果

需要根据有写请求时的 waddr 信号以及 size 信号恰当设置 wstrb 的大小，具体是参考上课时老师提供的讲义写的。

（5）归纳总结（可选）

没有仔细阅读讲义，其实 wstrb 是一个很重要的输出信号，包括之前在读数据时的 asize 信号，其实最开始也没有进行设置，但却能够运行通过仿真测试，这次出现了 wstrb 信号未设置的 bug 后增加了对其他重要信号的检查，否则可能出现仿真通过但是上板不通过的现象。

## 2、错误 2：读出数据时出错

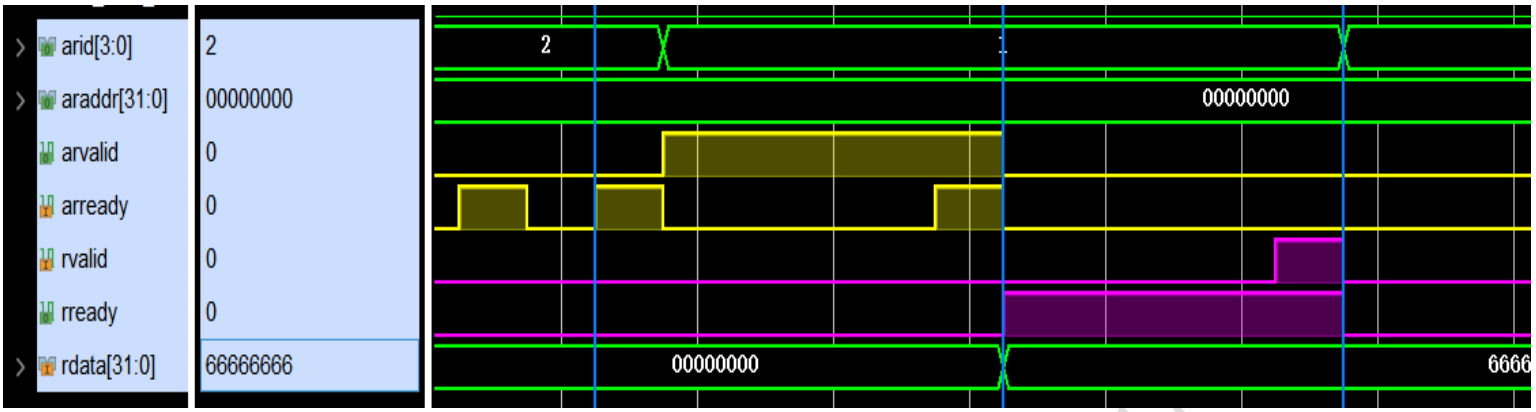
（1）错误现象

仿真时控制台信息报错，发现是在五次 data_read 指令时出错。如下图所示：

```
[   2815 ns] Fail!!!read from data sram-like, ref_data[31: 0]=32'h22220000, my_data[31: 0]=66666666
[   3145 ns] Fail!!!read from data sram-like, ref_data[23:16]=8'h11, my_data[23:16]=66
[   3275 ns] Fail!!!read from data sram-like, ref_data[15: 0]=16'h0000, my_data[15: 0]=6666
[   3355 ns] Fail!!!read from data sram-like, ref_data[31: 0]=32'h11113311, my_data[31: 0]=66666666
[   3685 ns] Fail!!!read from data sram-like, ref_data[31: 0]=32'h55004444, my_data[31: 0]=66666666
```

（2）分析定位过程

查看第一次读数据时的波形图，如下图所示：




从波形可以看出是在读数据握手时的握手地址 araddr 信号出错，一直是 0x00000000。因此怀疑是 araddr 信号的赋值逻辑出错了。查看代码，发现确实是这样：

```
assign araddr    = inst_addr_reg;
```

在给 araddr 赋值时只考虑了 inst_addr_reg 信号，因此一直是在读出 0x00000000 位置的指令数据，从而导致出错了。发生这个现象的原因可能是先写了读指令的处理部分，因此后面完成写数据和读数据时忘了修改前面的相关信号。

（3）错误原因

araddr 信号设置出错，导致读出的数据一直是指令的数据

（4）修正效果

只需要根据当前是正在读数据还是读指令来设置相应的 araddr 信号即可。如下所示：

```
assign araddr    =   data_read ? raddr_reg :
                     inst_read ? inst_addr_reg : 32'b0;
```

（5）归纳总结（可选）

本次实验在写代码时是将几种情况分开处理再合并的，因此难免出现在一些共享信号上设置时忘了根据后面新增加的逻辑进行调整的情况。

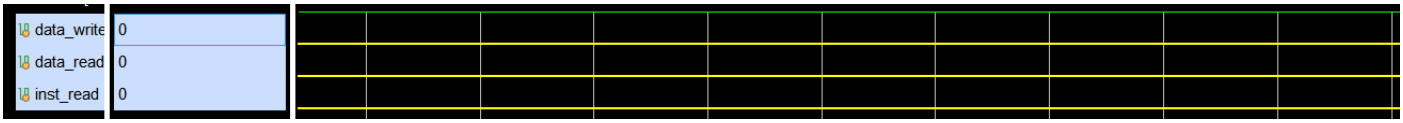
3、错误 3：在读指令时仿真一直无法停下

（1）错误现象

仿真时在读取指令数据时一直卡住，没有任何返回信息。

（2）分析定位过程

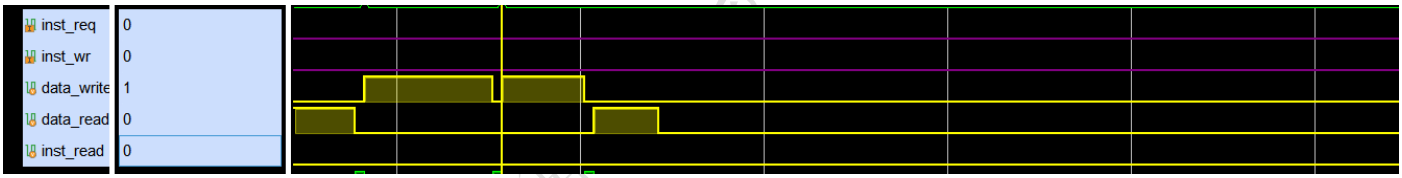
观察波形可以发现，设置的一个用于表示当前是否正在读指令的信号 inst_read 一直是 0，如下图所示：



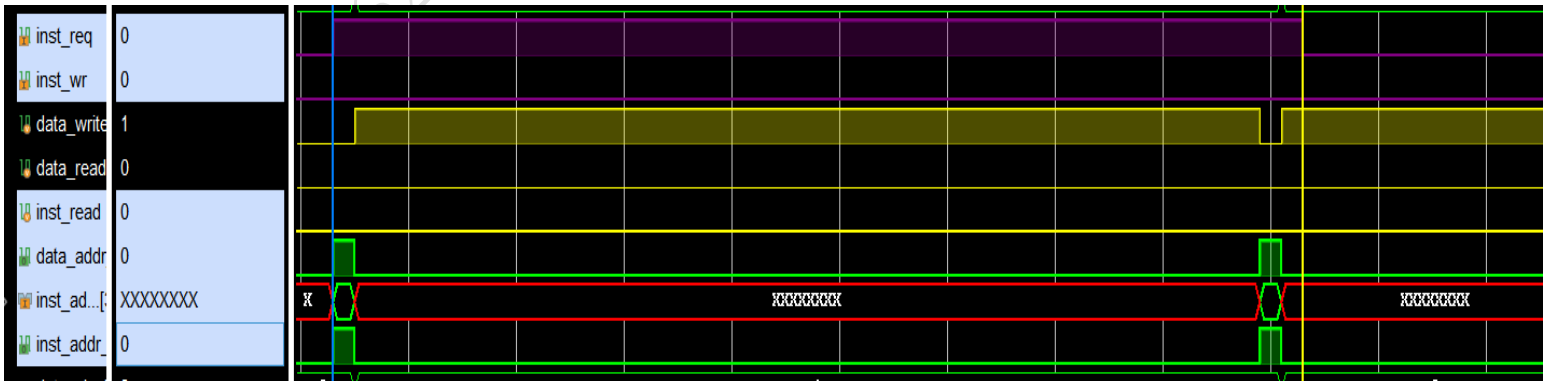

```
always@(posedge clk) begin
    if(~resetn) begin
        inst_read <= 1'b0;
    end
    else if(~inst_read && inst_req && ~inst_wr && ~data_req && ~data_read) begin
        inst_read <= 1'b1;
    end
    else if(inst_read && rready && rvalid) begin
        inst_read <= 1'b0;
    end
end
```

从 inst_read 的赋值逻辑可以看出，inst_read 信号被拉高的条件是当前 inst_read 为低，并且确实有读指令请求(inst_req & ~inst_wr) 以及 当前没有 data_read 正在进行，没有 data_req 的请求。

观察波形图：




可以发现在很长的一段时间内，inst_req 的请求都是 0，这也正是导致 inst_read 信号一直为低的原因。但是，inst_req 是输入信号，类 SRAM，AXI 转接桥本身不能对其进行直接控制。所以应该是某些转接桥的输出信号导致了 CPU 发生误判，所以输入的 inst_req 一直为 0 了。



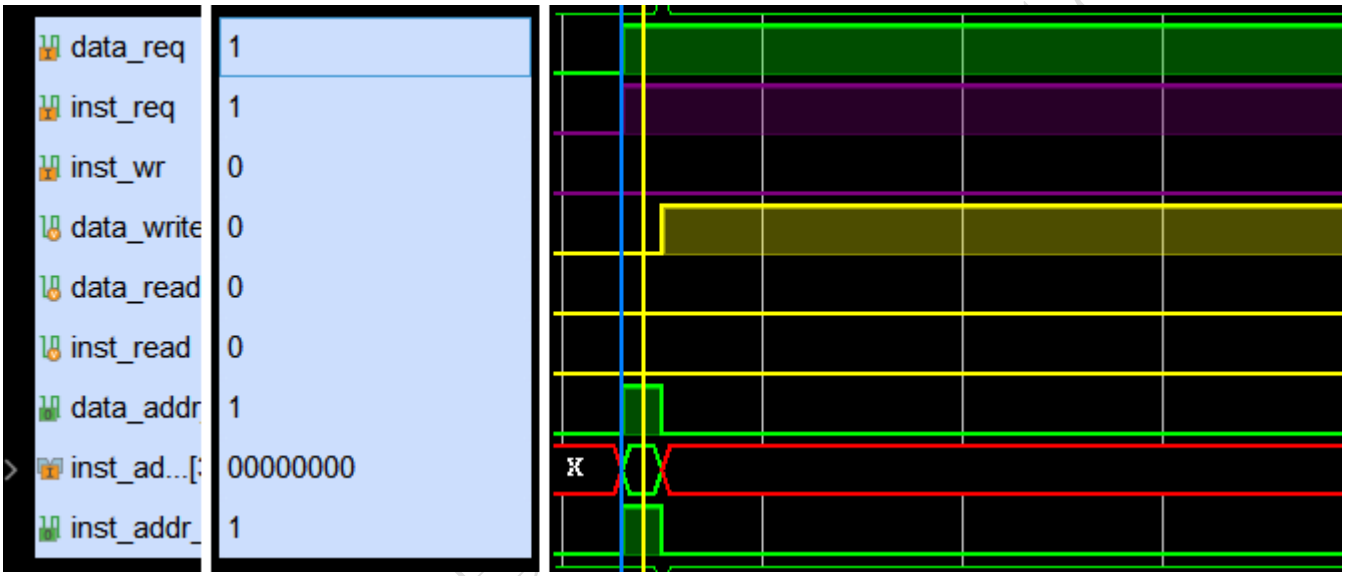

通过观察 inst_req 信号的变化可以看出，在 inst_req 请求为高的情况下，并没有进行读指令的操作(inst_read)为低。但是 inst_addr_ok 信号却意外地拉高了。从而导致此时的 inst_addr 变为有效了。结合本次实验是通过五次读取

数据，指令来判断的规则。可以猜测，virtual CPU 中是通过一个计数器来记录进行了多少次总线事务，而无故拉高的 addr_ok 信号显然改变了这个计数器的值，从而影响到了 inst_req 的输出。因此，出错的根本原因是在当前事务并不是 inst_read 的情况下就让信号拉高了。

观察 inst_addr_ok 信号的设置，如下所示：

```
assign inst_addr_ok  =  ~inst_read & ~data_write & ~data_read & inst_req & ~inst_wr ;
```

这是在老师提供的实例代码的基础上修改过来的，意思是说，当前没有读数据(对应于~has_inst_req)，没有正在进行 data_write，data_read 的操作，因为我们的设计是当总线完成一个事务之后再进行下一个事务。并且当前 inst_req 为 1，inst_wr 为 0，说明已经发出了读指令的请求，但是正在等待握手。




从波形图上来看，确实这样的逻辑就会得到这样的结果。不过这是忽略了 data 通道的请求的情况下，如果此时正好有 data_req，那么下一个事务一定不会是 inst_read, 此时拉高 inst_addr_ok 毫无意义。

（3）错误原因

未能正确设置 inst_addr_ok 的赋值逻辑

（4）修正效果

inst_addr_ok 的组合逻辑设置时还需要看此时是否有 data_req 的请求，如果有，说明下一个总线事务就不会是 inst_read 了，因为在设计中，每一个总线事务都需要完全完成之后才会进行下一个。即使是 inst_read 和 data_write 之间也是这样的。所以需要在 inst_addr_ok 的设置中最后加上一个与~data_req。

（5）归纳总结（可选）

在设置 inst_addr_ok 之前没有仔细考虑清楚影响该信号的所有因素。在真正看到实际的波形之前，自己甚至也还是不清楚这部分的逻辑应该是什么样子的。所以这里有点像为了符合波形而去写代码的感觉了。

## 4、错误 4：读出数据出错

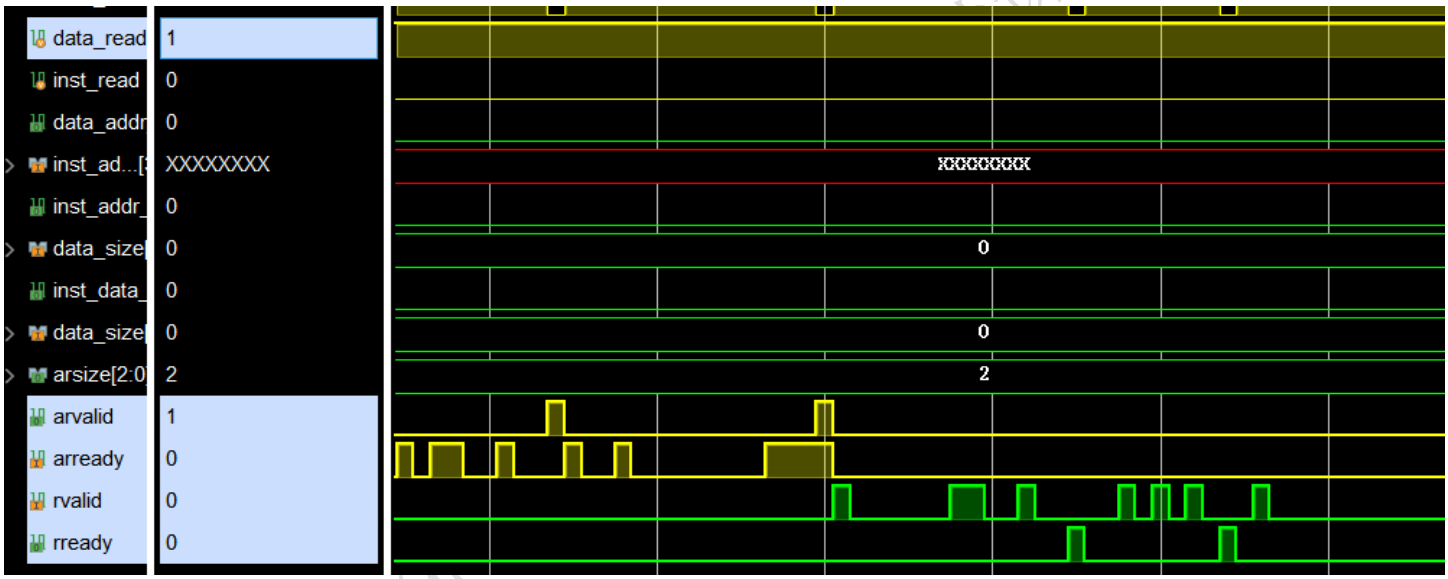
### （1）错误现象

仿真过程中读数据时出现两个错误，如下图所示：

```
[   2875 ns] Fail!!!read from data sram-like, ref_data[31: 0]=32'h22220000, my_data[31: 0]=00000000
[   4975 ns] OK!!!read data 1
[   5215 ns] OK!!!read data 2
[   5815 ns] Fail!!!read from data sram-like, ref_data[31: 0]=32'h11113311, my_data[31: 0]=22220000
```

### （2）分析定位过程

在分析第一次做读数据 data_read 操作时发现了一个很奇怪的现象，如下图所示：



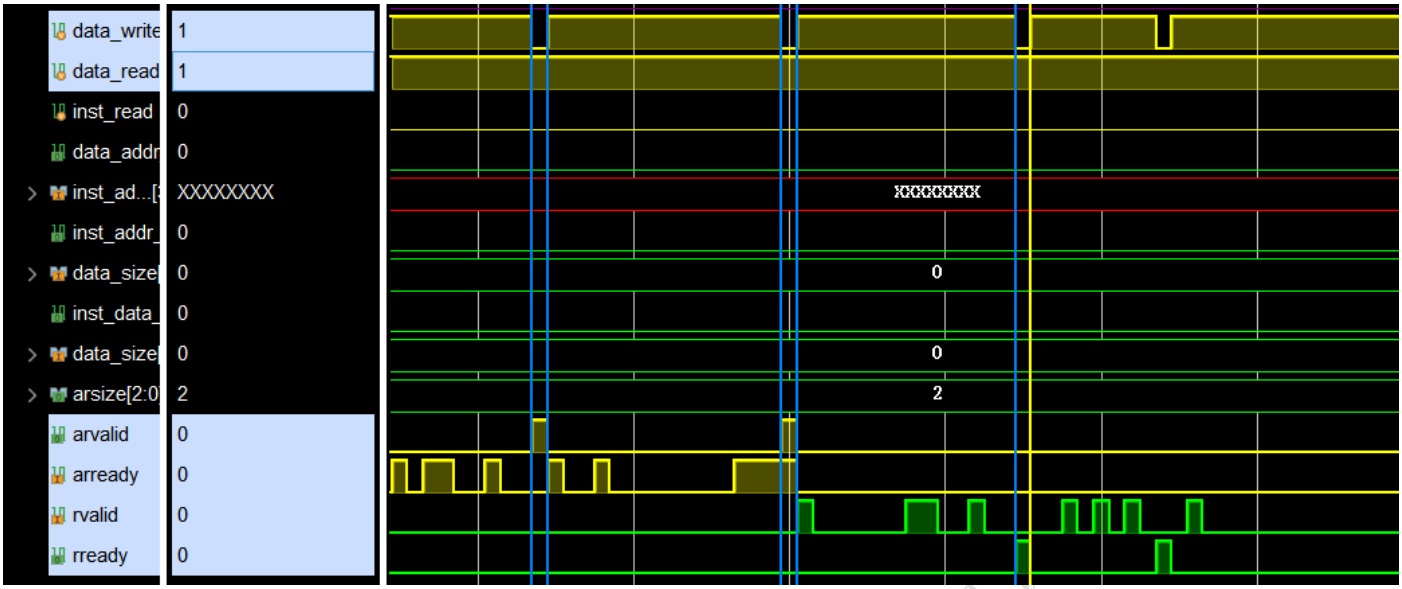

首先是 arvalid 信号，在被拉高了之后，还没有等待握手就自己降了下去，然后就是 rready 信号了，也是在拉高了之后还没等到握手就自己降了下去。这显然是和 AXI 总线协议的相关规定不符合的。

找到 arvalid 和 rready 两个信号的赋值逻辑，如下所示：

```
assign arvalid  =  ( data_read | inst_read ) & ~ar_handshake & ~data_write;
assign rready   =  ( data_read | inst_read ) &  ar_handshake & ~data_write;
```

可以看出，这个赋值的本意是说，当前如果当前的事务正好是 inst_read，data_read 这两个需要读数据的操作，并且 ar 通道还没握手(~ar_handshake)，就将 arvalid 信号拉高，如果已经握上手了，那就把 rready 信号拉高。最后和~data_write 相与是说如果当前没有正在进行 data_write 操作，那么才将这两个信号置为高。这也是因为在设计中我们是采用阻塞的方式来解决数据相关的。就算当前事务是 data_read 或是 inst_read，那么也要等到 data_write 完成之后再做，否则就可能有数据相关的风险。

然而，inst_read, data_read, data_write 这几个信号之间却可能存在“空隙”：




正是因为这些空隙的存在，导致了 arvalid 和 arready 时高时低。从而产生了错误的现象。产生错误的根源在于，这里好像是在同时处理两个事务：即 data_read 和 data_write 都拉高了。这和设计原则是不符合的。观察 data_read 的赋值逻辑：

```
always@(posedge clk) begin
    if(~resetn) begin
        data_read <= 1'b0;
    end
    else if(~data_read && data_req && ~data_wr && ~inst_read )  begin
        data_read <= 1'b1;
    end
    else if(data_read && rready && rvalid) begin
        data_read <= 1'b0;
    end
end
```

我们发现在 data_read 的置一逻辑处有问题，如果当前事务是 data_write 事务，那么就算其他条件满足，也不应该让 data_read 置 1，否则可能会出现读出来的数据是还未写进数据之前的数据的情况。

（3）错误原因

read_data 的置一逻辑有问题，本意是想通过阻塞来解决数据相关问题，但是 data_read 出现了在正在写数据时发起了读请求的情况，从而导致错误。

（4）修正效果

在 data_read 的置一逻辑里面将与~data_write 信号加上，这样能够保证各个总线事务之间独立进行，互不干扰。在这样修改了之后，至少从波形上看出，多个总线事务 data_read, data_write, inst_read 之间是相互隔离开的了。

## 5、错误 5：读写数据同时进行出错

（1）错误现象

在同一拍出现 data_addr_ok 和 data_data_ok,并且读数据错误。

（2）分析定位过程

我们观察波形发现，在这两个信号同时进行了读数据和写数据操作，复用了 data_addr 这一信号，导致错误。

（3）错误原因

同时进行了读数据和写数据操作，没有加以区分，导致部分数据通路的复用。

（4）修正效果

额外增加了两个状态指示信号来判断当前处于读指令还是写指令，让另一条指令等待当前事务的完成。