**实验十报告**

学号 2017K8009929044

2017K8009929025

姓名 李昊宸 李颖彦

箱子号 33

一、实验任务（10%）

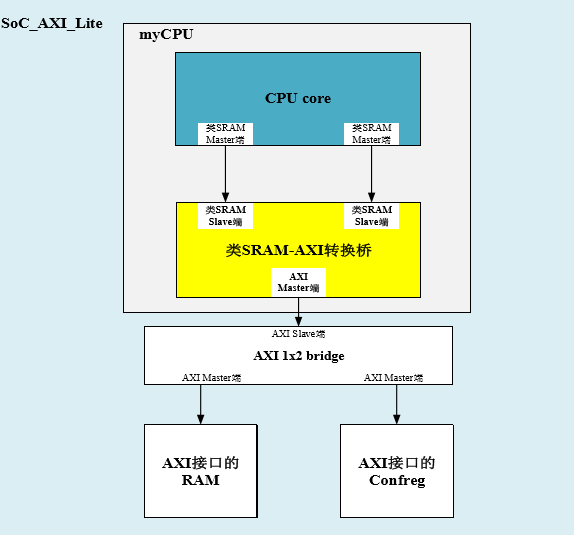
1. 掌握 AXI协议的知识。

2. 学会设计类 SRAM接口到 AXI接口的转换转换桥。

二、实验设计（40%）

（一）总体设计思路

CPU与内存的交互通过转接桥来完成控制信号的转换和数据的交互：



图一 类SRAM-AXI转接桥与CPU、RAM的关系

CPU中有两个内存模块的读写控制器：指令ram和数据ram，这两个端口设置为类SRAM的Master端，Slave端在转接桥内实现；AXI的Master端在转接桥内实现，AXI Ix2 bridge设置为Slave端，从而实现从两个类SRAM到一个AXI RAM的映射。

转接桥中的重要部分是AXI端口的设计。AXI接口由五个通道构成：读请求通道，读响应通道、写请求通道、写响应通道、写返回通道。上述五个通道分别完成了读、写的功能。由于存在写后读相关、读写碰撞等需要让另一个功能被阻塞的情况，构建两个状态机分别描述读通道和写通道的状态是合适的。

（二）重要模块1设计：封装

1. 工作原理

转接桥的设计需要封装输入端口和输出端口。其中Slave端口为类SRAM的两个Slave端口，分别实现指令ram和数据ram的功能；Master端口为AXI的Master端口。

1. 功能描述

input clk,

input resetn,

//inst sram-like (master: cpu, slave: interface)

input inst\_req ,// master -> slave

input inst\_wr ,// master -> slave

input [ 1:0] inst\_size ,// master -> slave

input [31:0] inst\_addr ,// master -> slave

input [31:0] inst\_wdata ,// master -> slave

output reg [31:0] inst\_rdata ,// slave -> master

output inst\_addr\_ok ,// slave -> master

output inst\_data\_ok ,// slave -> master

//data sram-like (master: cpu, slave: interface)

input data\_req ,// master -> slave

input data\_wr ,// master -> slave

input [ 1:0] data\_size ,// master -> slave

input [31:0] data\_addr ,// master -> slave

input [31:0] data\_wdata ,// master -> slave

output reg [31:0] data\_rdata ,// slave -> master

output data\_addr\_ok ,// slave -> master

output data\_data\_ok ,// slave -> master

//axi (master: interface, slave: axi)

//ar: acquire reading channels

output reg [ 3:0] arid ,// master -> slave

output reg [31:0] araddr ,// master -> slave

output [ 7:0] arlen ,// master -> slave, fixed at 8'b0

output reg [ 2:0] arsize ,// master -> slave

output [ 1:0] arburst ,// master -> slave, fixed at 2'b1

output [ 1:0] arlock ,// master -> slave, fixed at 2'b0

output [ 3:0] arcache ,// master -> slave, fixed at 4'b0

output [ 2:0] arprot ,// master -> slave, fixed at 3'b0

output reg arvalid ,// master -> slave

input arready ,// slave -> master

//r: reading response channels

input [ 3:0] rid ,// slave -> master

input [31:0] rdata ,// slave -> master

input [ 1:0] rresp ,// slave -> master, ignore

input rlast ,// slave -> master, ignore

input rvalid ,// slave -> master

output reg rready ,// master -> slave

//aw: acquire writing channels

output [ 3:0] awid ,// master -> slave, fixed at 4'b1

output reg [31:0] awaddr ,// master -> slave

output [ 7:0] awlen ,// master -> slave, fixed at 8'b0

output reg [ 2:0] awsize ,// master -> slave

output [ 1:0] awburst ,// master -> slave, fixed at 2'b1

output [ 1:0] awlock ,// master -> slave, fixed at 2'b0

output [ 3:0] awcache ,// master -> slave, fixed at 4'b0

output [ 2:0] awprot ,// master -> slave, fixed at 3'b0

output reg awvalid ,// master -> slave

input awready ,// slave -> master

//w: write data channels

output [ 3:0] wid ,// master -> slave, fixed at 4'b1

output reg [31:0] wdata ,// master -> slave

output reg [ 3:0] wstrb ,// master -> slave

output wlast ,// master -> slave, fixed at 1'b1

output reg wvalid ,// master -> slave

input wready ,// slave -> master

//b: writing response channels

input [ 3:0] bid ,// slave -> master, ignore

input [ 1:0] bresp ,// slave -> master, ignore

input bvalid ,// slave -> master

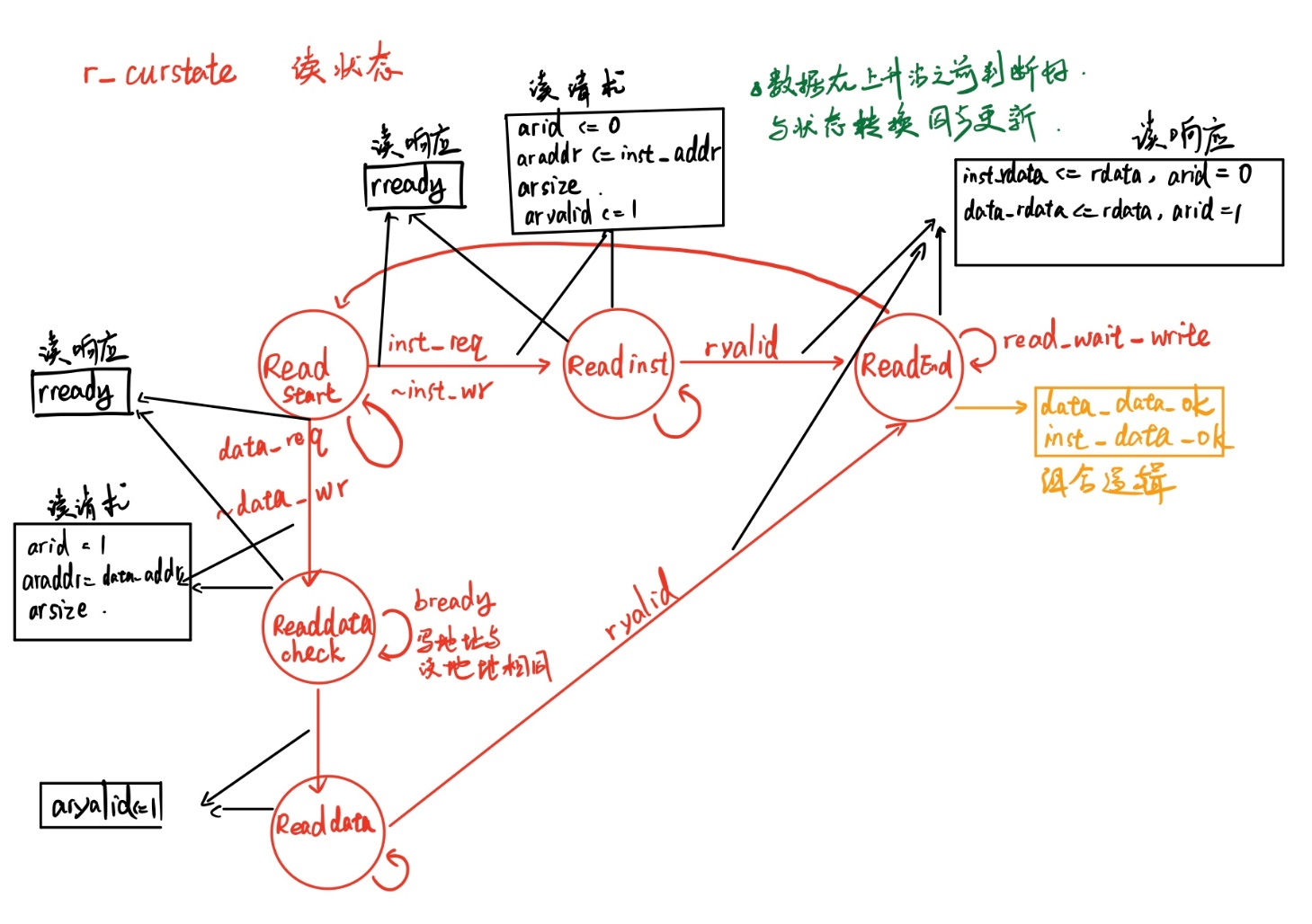
output reg bready // master -> slave

（三）重要模块2设计：读状态机

1. 工作原理

引入读状态机来描述读请求通道和读返回通道的状态，便于对时序信号的判断和处理。

1. 功能描述



图二 读状态机的状态转移图，其中对于信号的两个箭头，前一个箭头表示信号生成时间，后一个箭头表示信号更新时间

1. 具体代码实现

读状态机：

*always@(\*)*

*begin*

*case(r\_curstate)*

*ReadStart:*

*begin*

*if(inst\_req && ~inst\_wr)*

*r\_nxtstate = Readinst;*

*else if(data\_req && ~data\_wr)*

*r\_nxtstate = Read\_data\_check;*

*else*

*r\_nxtstate = r\_curstate;*

*end*

*Readinst, Readdata:*

*begin*

*if(rvalid)*

*r\_nxtstate = ReadEnd;*

*else*

*r\_nxtstate = r\_curstate;*

*end*

*Read\_data\_check:*

*begin*

*if(bready && awaddr\_t[31:2] == araddr[31:2])*

*r\_nxtstate = r\_curstate;*

*else*

*r\_nxtstate = Readdata;*

*end*

*ReadEnd:*

*begin*

*if(read\_wait\_write)*

*r\_nxtstate = r\_curstate;*

*else*

*r\_nxtstate = ReadStart;*

*end*

*default:*

*r\_nxtstate = ReadStart;*

*endcase*

*end*

相关读信号：

*always@(posedge clk)*

*begin*

*if(r\_curstate == ReadStart && r\_nxtstate == Readinst) begin*

*arid <= 4'd0;*

*araddr <= inst\_addr;*

*arsize <= !inst\_size ? 3'd1 : {inst\_size, 1'b0};*

*end else if(r\_curstate == ReadStart && r\_nxtstate == Read\_data\_check) begin*

*arid <= 4'd1;*

*araddr <= {data\_addr[31:2], 2'd0};*

*arsize <= !data\_size ? 3'd1 : {data\_size, 1'b0};*

*end else if(r\_curstate == ReadEnd) begin*

*araddr <= 32'd0;*

*end*

*end*

*assign inst\_addr\_ok = r\_curstate == ReadStart && r\_nxtstate == Readinst || w\_curstate == WriteStart && w\_nxtstate == Writeinst;*

*assign data\_addr\_ok = r\_curstate == ReadStart && r\_nxtstate == Read\_data\_check || w\_curstate == WriteStart && w\_nxtstate == Writedata;*

对于读指令ram，arid、araddr与arsize信号在inst\_req和~inst\_wr拉高时进行赋值；对于读数据ram，在data\_req和~data\_wr拉高时进行赋值。另外，在读响应通道返回值握手成功后，将araddr的值修改为0。

*always@(posedge clk)*

*begin*

*if(~resetn)*

*arvalid <= 1'b0;*

*else if(r\_curstate == ReadStart && r\_nxtstate == Readinst || r\_curstate == Read\_data\_check && r\_nxtstate == Readdata)*

*arvalid <= 1'b1;*

*else if(arready)*

*arvalid <= 1'b0;*

*end*

当aresetn有效期间，arvalid的值须为0。对于指令ram，读请求地址有效信号与araddr值的更新同步发出；对于数据ram，由于可能存在写通道正在操作数据ram中的同一地址，此时读通道需要被阻塞在Read\_data\_check状态，直到写操作结束时，更新arvalid的值。需要提醒的是，arvalid的置1不能与同通道的ready信号（arready）相关。另外，为保证握手的正确性，当arready返回值为1时，需要将arvalid置为0，中断握手。

*always@(posedge clk)*

*begin*

*if(~resetn)*

*rready <= 1'b1;*

*else if(r\_nxtstate == Readinst || r\_nxtstate == Read\_data\_check)*

*rready <= 1'b1;*

*else if(rvalid)*

*rready <= 1'b0;*

*end*

当aresetn有效期间，rready的值不能为不定值，此处设置为1。当检测到状态到达Readinst或Read\_data\_check时，说明读地址已经准备就绪，此时可以发出rready信号。当rvalid握手成功后，需要将rready置0。需要提醒的是，rready信号不能依赖于当拍的rvalid信号，但是此处是依赖于上一拍的rready信号，所以是合理的。

*always@(posedge clk)*

*begin*

*if(rvalid && arid == 4'd0)*

*inst\_rdata <= rdata;*

*if(rvalid && arid == 4'd1)*

*data\_rdata <= rdata;*

*end*

当读响应通道握手成功后，把读出数据送回到类SRAM Master端，在下一个时钟上升沿更新。

*assign inst\_data\_ok = r\_curstate == ReadEnd && arid == 4'd0;*

*assign data\_data\_ok = r\_curstate == ReadEnd && r\_nxtstate == ReadStart && arid == 4'd1*

*|| w\_curstate == WriteEnd && w\_nxtstate == WriteStart || rvalid\_t;*

*always@(posedge clk)*

*begin*

*if(~resetn)*

*rvalid\_t <= 1'b0;*

*else if(r\_curstate == ReadEnd && r\_nxtstate == ReadStart && arid == 4'd1*

*&& w\_curstate == WriteEnd && w\_nxtstate == WriteStart)*

*rvalid\_t <= 1'b1;*

*else*

*rvalid\_t <= 1'b0;*

*end*

rvalid\_t信号就是当读和写同时开始，同时结束时，先让写发出ok信号，再到下一拍发出读ok，从而避免明明同时读写，最后只有读/写发出data\_ok信号。

*always@(posedge clk)*

*begin*

*if(~resetn)*

*read\_wait\_write <= 1'b0;*

*else if(r\_curstate == ReadStart && r\_nxtstate == Read\_data\_check && bready && ~bvalid) read\_wait\_write <= 1'b1;*

*else if(bvalid)*

*read\_wait\_write <= 1'b0;*

*end*

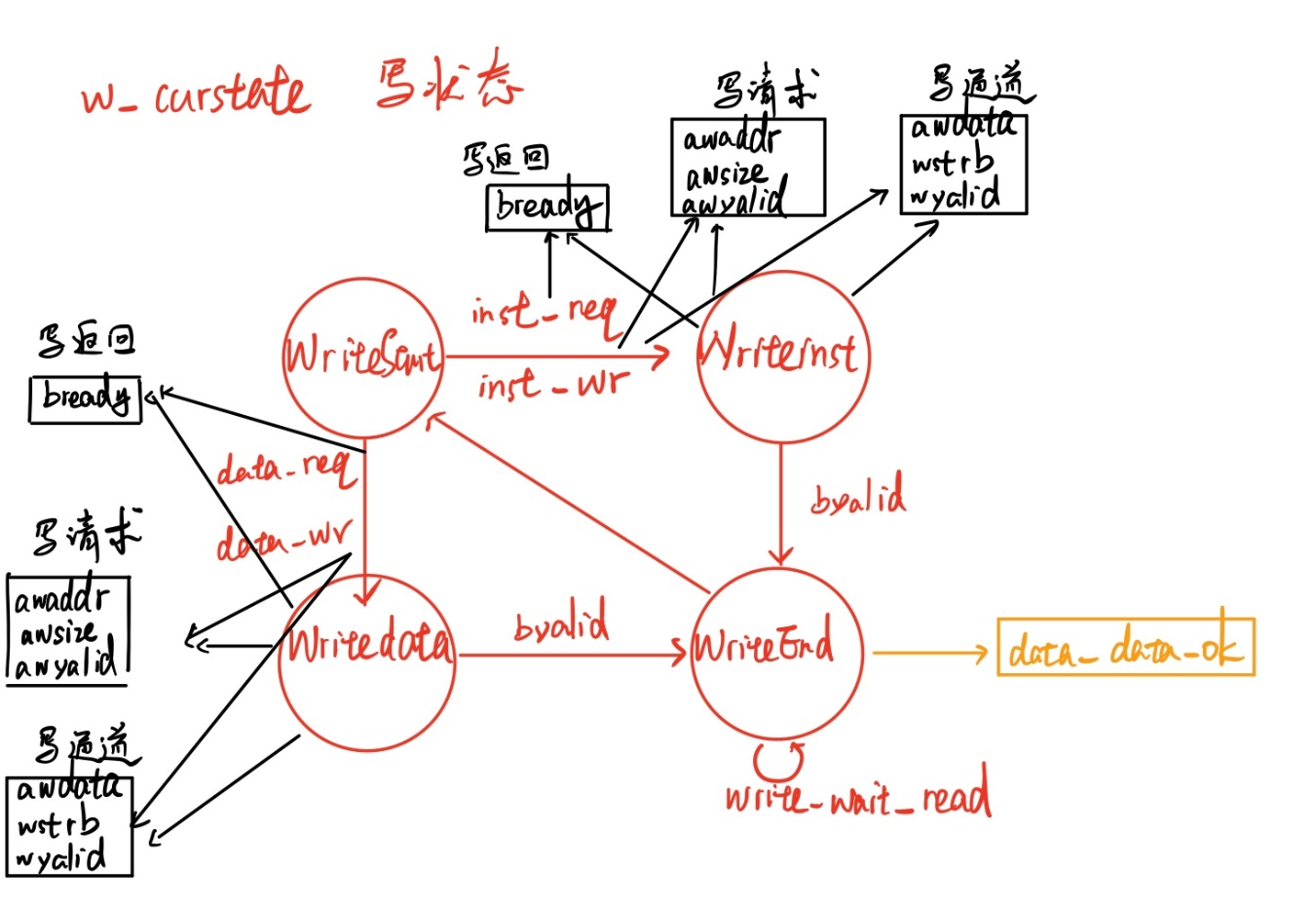
read\_wait\_write表示在读开始时，如果发现写通道较早开始，则需要拉高，等到按顺序在写通道结束之后才能结束读操作，即拉低该信号。

（四）重要模块3设计：写状态机

1. 工作原理

引入读状态机来描述读请求通道和读返回通道的状态，便于对时序信号的判断和处理。

1. 功能描述



图三 写状态机的状态转移图，其中对于信号的两个箭头，前一个箭头表示信号生成时间，后一个箭头表示信号更新时间

1. 具体代码实现

写状态机：

*always@(\*)*

*begin*

*case (w\_curstate)*

*WriteStart:*

*begin*

*if(inst\_req && inst\_wr)*

*w\_nxtstate = Writeinst;*

*else if(data\_req && data\_wr)*

*w\_nxtstate = Writedata;*

*else*

*w\_nxtstate = w\_curstate;*

*end*

*Writeinst, Writedata:*

*begin*

*if(bvalid)*

*w\_nxtstate = WriteEnd;*

*else*

*w\_nxtstate = w\_curstate;*

*end*

*WriteEnd:*

*begin*

*if(write\_wait\_read)*

*w\_nxtstate = w\_curstate;*

*else*

*w\_nxtstate = WriteStart;*

*end*

*default:*

*w\_nxtstate = WriteStart;*

*endcase*

*end*

相关写信号：

*always@(posedge clk)*

*begin*

*if(w\_curstate == WriteStart && w\_nxtstate == Writeinst) begin*

*awaddr <= inst\_addr;*

*awsize <= !inst\_size ? 3'd1 : {inst\_size, 1'b0};*

*end else if(w\_curstate == WriteStart && w\_nxtstate == Writedata) begin*

*awaddr <= {data\_addr[31:2], 2'd0};*

*awsize <= !data\_size ? 3'd1 : {data\_size, 1'b0};*

*end*

*end*

对于写数据ram，awaddr与awsize信号在data\_req和data\_wr拉高时进行赋值。

*always@(posedge clk)*

*begin*

*if(~resetn)*

*awvalid <= 1'b0;*

*else if(w\_curstate == WriteStart && (w\_nxtstate == Writeinst || w\_nxtstate == Writedata))*

*awvalid <= 1'b1;*

*else if(awready)*

*awvalid <= 1'b0;*

*end*

当aresetn有效期间，awvalid的值须为0。对于数据ram，写请求地址有效信号与awaddr值的更新同步发出。需要提醒的是，awvalid的置1不能与同通道的ready信号（awready）相关。另外，为保证握手的正确性，当awready返回值为1时，需要将awvalid置为0，中断握手。

*always@(posedge clk)*

*begin*

*if(~resetn)*

*wvalid <= 1'b0;*

*else if(w\_curstate == WriteStart && (w\_nxtstate == Writeinst || w\_nxtstate == Writedata))*

*wvalid <= 1'b1;*

*else if(wready)*

*wvalid <= 1'b0;*

*end*

当aresetn有效期间，wvalid的值须为0。对于数据ram，写请求地址有效信号与wdata值的更新同步发出。需要提醒的是，wvalid的置1不能与同通道的ready信号（wready）相关。另外，为保证握手的正确性，当wready返回值为1时，需要将wvalid置为0，中断握手。

*always@(posedge clk)*

*begin*

*if(w\_curstate == WriteStart && w\_nxtstate == Writeinst) begin*

*wdata <= inst\_wdata;*

*wstrb <= (data\_addr[1:0] == 2'b00 && data\_size == 2'b00) ? 4'b0001*

*:(data\_addr[1:0] == 2'b01 && data\_size == 2'b00) ? 4'b0010*

*:(data\_addr[1:0] == 2'b10 && data\_size == 2'b00) ? 4'b0100*

*:(data\_addr[1:0] == 2'b11 && data\_size == 2'b00) ? 4'b1000*

*:(data\_addr[1:0] == 2'b00 && data\_size == 2'b01) ? 4'b0011*

*:(data\_addr[1:0] == 2'b10 && data\_size == 2'b01) ? 4'b1100*

*:(data\_addr[1:0] == 2'b00 && data\_size == 2'b10) ? 4'b1111*

*: 4'b0000;*

*end else if(w\_curstate == WriteStart && w\_nxtstate == Writedata) begin*

*wdata <= data\_wdata;*

*wstrb <= (data\_addr[1:0] == 2'b00 && data\_size == 2'b00) ? 4'b0001*

*:(data\_addr[1:0] == 2'b01 && data\_size == 2'b00) ? 4'b0010*

*:(data\_addr[1:0] == 2'b10 && data\_size == 2'b00) ? 4'b0100*

*:(data\_addr[1:0] == 2'b11 && data\_size == 2'b00) ? 4'b1000*

*:(data\_addr[1:0] == 2'b00 && data\_size == 2'b01) ? 4'b0011*

*:(data\_addr[1:0] == 2'b10 && data\_size == 2'b01) ? 4'b1100*

*:(data\_addr[1:0] == 2'b00 && data\_size == 2'b10) ? 4'b1111*

*: 4'b0000;*

*end*

*end*

wdata和wstrb的更新在req和wr信号发出后进行。

*always@(posedge clk)*

*begin*

*if(~resetn)*

*bready <= 1'b0;*

*else if(w\_nxtstate == Writeinst || w\_nxtstate == Writedata)*

*bready <= 1'b1;*

*else if(bvalid)*

*bready <= 1'b0;*

*end*

当aresetn有效期间，bready的值不能为不定值，此处设置为0。当检测到状态到达Writeinst或Writedata时，说明写地址和写数据已经准备就绪，此时可以发出bready信号。当bvalid握手成功后，需要将bready置0。需要提醒的是，bready信号不能依赖于当拍的bvalid信号，但是此处是依赖于上一拍的bready信号，所以是合理的。

（五）重要模块4设计：其他信号线的设计

1. 工作原理

其他信号线在Lab10中都是可以设置为定值的信号线，故可用wire + assign的组合进行赋值。

1. 功能描述

*// ar*

*assign arlen = 8'd0;*

*assign arburst = 2'b01;*

*assign arlock = 2'd0;*

*assign arcache = 4'd0;*

*assign arprot = 3'd0;*

*//aw*

*assign awid = 4'd1;*

*assign awlen = 8'd0;*

*assign awburst = 2'b01;*

*assign awlock = 2'd0;*

*assign awcache = 4'd0;*

*assign awprot = 3'd0;*

*//w*

*assign wid = 4'd1;*

*assign wlast = 1'b1;*

三、实验过程（50%）

（一）实验流水账

2019/11/11 14:00 – 24:00 状态机设计

2019/11/12 18:00 – 21:00 信号调试

2019/11/14 8:00 – 11:00 书写实验报告

（二）错误记录

1、错误1：连续写操作

（1）错误现象

一个写请求发出后，写过程还未完成就响应了新的写操作，导致数据丢失。

（2）分析定位过程

查看Viture\_cpu，发现信号的发出与data\_ok信号相关。如果有信号提前发出，说明data\_ok信号在写操作完成之前就已经被拉高。查看波形后发现错误确实在data\_ok信号拉高后，新的写请求被发送进入队列，确定问题出在这里。

（3）错误原因

data\_ok判断逻辑不正确。

（4）修正效果

*assign data\_data\_ok = r\_curstate == ReadEnd && r\_nxtstate == ReadStart && arid == 4'd1*

*|| w\_curstate == WriteEnd && w\_nxtstate == WriteStart || rvalid\_t;*

2、错误2： 读返回异常

（1）错误现象

连续的读请求发出后，第一个读请求的数据总是无法返回。

（2）分析定位过程

每次最先返回的都是第二个请求的值，怀疑可能是传参时没有用锁存器保存，在addr\_ok后地址发生了变动，导致前一次取样的地址丢失，新传入的地址代替了原来的地址。查看波形后发现在取样的上升沿，地址发生了变动。

（3）错误原因

未锁存取数地址。

（4）修正效果

*always@(posedge clk)*

*begin*

*if(r\_curstate == ReadStart && r\_nxtstate == Readinst) begin*

*araddr <= inst\_addr;*

*end else if(r\_curstate == ReadStart && r\_nxtstate == Read\_data\_check) begin*

*araddr <= {data\_addr[31:2], 2'd0};*

*end else if(r\_curstate == ReadEnd) begin*

*araddr <= 32'd0;*

*end*

*end*

修改之后数据正常。

四、实验总结（可选）

说实话，对这一次实验的讲义部分表示不太满意（甚至有点生气）。两个部分确实很清楚的介绍了类SRAM和AXI总线的特性和结构，但是这次需要实现的关键部分转接桥却只字不提。（虽然有很多很多的设计建议，但是看起来更像是在有了大框架下需要注意的小细节，这些小细节可不足以作为整个框架的描述）更重要的是，像是【状态机】这么重要的实现方式，还是在看讲义四五遍之后在一句不怎么重要的话中看到被略微提及了一下。总之希望任务和讲义可以结合的更为紧密一些。