**Lab 7报告**

学号：2020K8009929034、2020K8009929019、2020K8009929037

姓名：胡康、李子恒、吕星宇

箱子号：05

一、实验任务（10%）

为设计的CPU增加AXI总线接口，更好地实现CPU与系统中的内存、外设进行交互。整体任务分为三个阶段：

1. 将原有CPU访问SRAM的接口调整为类SRAM总线接口，在原有基础上增加握手信号。
2. 设计类SRAM-AXI转接桥，将原有的类SRAM接口转接为AXI接口，CPU核对外提供AXI信号端。
3. 完成整体总线设计对随机种子的验证。

二、实验设计（40%）

（一）总体设计思路

1. 设计类SRAM总线：增加CPU的端口信号，利用req、addr\_ok、data\_ok等信号实现握手机制（具体端口信号的增加见重要模块1）。同时。为了保证流水线的正常运作，更改部分ready\_go、allowin信号；最后考虑异常清空流水线、转移计算未完成等特殊情况。

2. 设计类SRAM-AXI转接桥：增加AXI接口信号，一方面需要与类SRAM信号对接，另一方面需要设计状态机，对外通过握手信号完成交互。

（二）重要模块1设计：类SRAM总线接口信号

1. 工作原理

作为类SRAM总线接口信号，实现握手机制，与内存、外设进行交互。由于内存可以抽象为inst\_sram与data\_sram，故接口信号也分为inst信号与data信号。两组信号的接口定义见表1，由于整体功能较为相似，故在表1中一并展示。

CPU与内存的交互主要分为三种：取指、load类访存、store类访存。对于取指操作，CPU发出读请求（inst\_sram\_req），内存接收请求并返回接收成功信号（inst\_sram\_addr\_ok）与指令（inst\_sram\_rdata）；对于load类访存，CPU发出读请求（data\_sram\_req）并用字节数（data\_sram\_size）表示请求的字节数，内存接收请求并返回接收成功信号（data\_sram\_addr\_ok）与数据（data\_sram\_rdata）；对于store类指令，CPU发出写请求（data\_sram\_req）以及字节数（data\_sram\_size）、字节写使能（data\_sram\_wstrb）、写数据（data\_sram\_wdata），内存接收请求并返回成功信号（data\_sram\_addr\_ok）表示写入完成。

1. 接口定义

表1：类SRAM总线接口信号表

| **名称** | **位宽** | **方向** | **功能描述** |
| --- | --- | --- | --- |
| clk | 1 | input | 时钟信号 |
| req | 1 | output | 请求信号 |
| wr | 1 | output | 1表示写请求，0表示读请求 |
| size | 2 | output | 传输的字节数 |
| addr | 32 | output | 请求的地址 |
| wstrb | 4 | output | 写请求的字节写使能 |
| wdata | 32 | output | 写请求的写数据 |
| addr\_ok | 1 | input | 该次请求的地址已被接收 |
| data\_ok | 1 | input | 该次请求的数据已传输 |
| rdata | 32 | input | 该次请求返回的读数据 |

1. 功能描述

在SRAM的基础上增加了握手机制，更加贴近真实内存的访问情况。此外，握手机制意味着指令、数据不会在下一拍立刻返回，因此需要考虑流水线新增的阻塞情况，引发了许多需要考虑的复杂问题，需要在原有基础上进行改进。

1. 重要模块2设计：类SRAM-AXI转接桥
2. 工作原理

与原有的类SRAM接口连接，同时对外提供AXI接口。

由于AXI也需要处理对外握手过程，因此需要设计状态机，完成状态转移的描述。其中，读状态机设计如下：

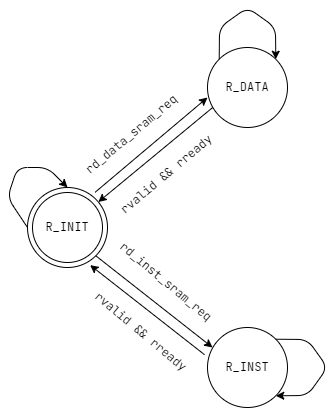


图1：读状态机状态转移示意图

初始时，读状态机停留在R\_INIT阶段，当读数据请求来临时，读状态机转移到等待数据的R\_DATA阶段；当读指令请求来临时，读状态机转移到等待指令的R\_INST阶段。进入R\_DATA或R\_INST阶段后，状态机状态不变，直到读响应完成，即rvalid与rready完成握手。因此，这两个阶段实际上是从读请求开始直到读响应结束，将读请求状态机和读响应状态机整合简化为一台读状态机。

写请求与写数据状态机设计如下：

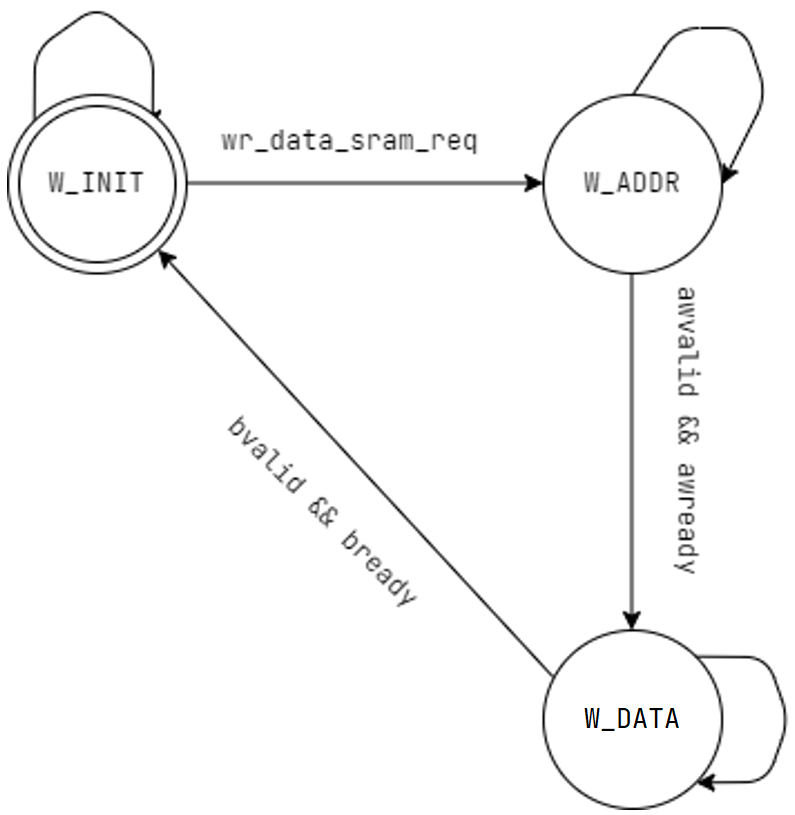


图2：写请求与写数据状态机状态转移示意图

初始时，写请求与写数据状态机停留在W\_INIT阶段。当写数据请求来临时，状态机转移到W\_ADDR阶段，表示正在发送写请求地址。写请求awvalid与awready握手成功，即表示地址已发送，这时转移到W\_DATA阶段，表示正在发送写数据。写响应bvalid与bready握手成功，即表示写数据已完成，本轮请求全部完成，转移到W\_INIT阶段等待下轮请求。

写响应状态机如下：

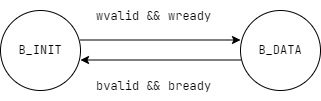


图3：写响应状态机状态转移示意图

写响应其实就是对从方是否写完数据的检查，保证主方不会在从方未完成数据写入的时候就匆忙发送读请求，读出旧值。初始时，写响应状态机位于B\_INIT阶段。写数据wvalid与wready握手，说明已经开始写数据了，这时要转移到B\_DATA阶段，等待从方将数据写完并发送写响应。当写响应bvalid与bready握手时，写数据已经完成，这时回到B\_INIT阶段，等待下轮请求。

2、各通道接口信号定义

（1）读请求通道

只讨论核心的几个接口信号。读请求来临时，需要在初始阶段需要准备好arid, araddr和arsize，同时将读请求arvalid置为有效。以arid为例，代码如下：

always @(posedge aclk) begin

if(!aresetn)

arid\_r <= 4'd0;

else if(ar\_current\_state == R\_INIT && rd\_data\_sram\_req)

arid\_r <= 4'd1; // 取数据置1（优先级高）

else if(ar\_current\_state == R\_INIT && rd\_inst\_sram\_req)

arid\_r <= 4'd0; // 取指令置0

end

（2）读数据通道

发送读请求后，主方就可以开始准备读数据了，因此这时可以将rready拉高，等读数据完成（即下一个阶段恰好为R\_INIT时）将rready拉低，等待下一轮请求。在这里我采用的是valid before ready handshake，ready信号置有效依赖于valid信号。

（3）写请求与写数据通道

与读请求类似，写请求来临时，需要设置好awaddr与awsize，并置起awvalid信号。这里写请求与写数据共用状态机，需要一并把数据wdata和wstrb设置好，同样置起wvalid信号。

（4）写响应通道

只需在写数据完成握手时，将bready拉高，并在完成握手，即bvalid有效时，再将bready拉低即可。

3、类SRAM信号定义

需要定义转接桥向类SRAM方发送的信号，即数据RAM与指令RAM的addr\_ok, data\_ok与rdata信号。这里设计的addr\_ok对应于读/写请求有效（注意指令RAM不可写），data\_ok对应于读/写响应有效。对于rdata，只需在R\_DATA或R\_INST阶段把读到的数据（AXI接口上的rdata）放入缓冲区，然后交给相应的指令或数据RAM的rdata端口即可。

三、实验过程（50%）

（一）实验流水账

2022.10.27 19：00——23：00 阅读讲义8.1、8.2；

2022.10.28 20：00——24：00 exp14 debug

2022.10.29 19：30——23：24 完成exp14仿真及上板

2022.10.30 8：00——10：30 完成exp14的实验报告

2022.11.05 10：00——20：25 阅读讲义8.3、8.4，并完成exp15 debug

2022.11.06 9：30——14：00 完成exp15仿真，修复之前的bug， 完成上板测试

2022.11.12 20：00——20：20 完成exp16上板测试

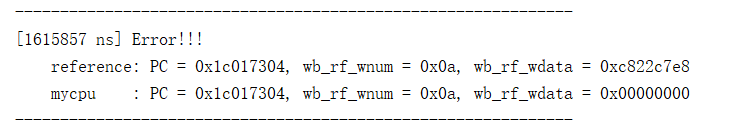
（二）错误记录

1、错误1：EXE阶段未控制addr\_ok有效时再进入下一阶段

（1）错误现象

如图4，某指令写回数据发生错误

图4：错误1出错情况截图



（2）分析定位过程

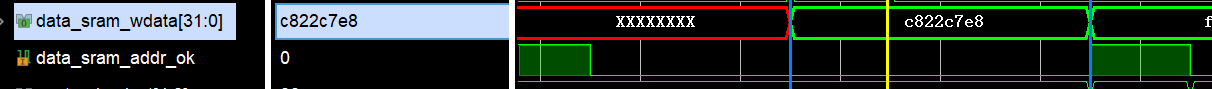
查询汇编文件可以看到这是一条load指令，且波形图显示在WB阶段拿到数据时就已经出错，于是推测可能是之前的store指令存入该内存的数据错误。从该时刻向前找写入该地址的store指令，找到了不久前刚向该地址写入数据的某store指令。检查该指令EXE阶段与MEM阶段的访存行为并重点锁定data\_sram\_wdata与data\_sram\_addr\_ok两个信号后可以发现，该数据有效期间data\_sram\_addr\_ok始终为0（如图5），说明正确的数据没有写入。

图5：错误1分析定位过程

（3）错误原因

未控制该store指令在EXE阶段看到addr\_ok为1才能进入MEM阶段，导致数据没有正确写入，进而导致后续load指令读出错误数据。

（4）修正效果

按照讲义P195的要点讲解，修改es\_ready\_go使得访存指令的req与addr均为1后才能进入下一级

（5）归纳总结

由于本阶段增加了握手机制，很多原本一拍内一定能完成的操作都变为未知，因此要让流水线在某些情况下“暂停”，而实现的方式便是通过ready\_go信号。

2、错误2：清空流水线时没有修改fs\_valid为0

（1）错误现象

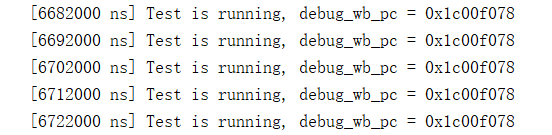
仿真跑到660000ns后在0x1c00f078处进入死循环（如图6）

图6：错误2出错情况截图

（2）分析定位过程

查询汇编文件可以看到0x1c00f078处指令是一条ertn指令，波形图显示该指令执行后，next\_pc取到了ERA寄存器内的地址，但该地址无法进入IF级，导致流水线“断流”。这说明错误原因是pre-IF级与IF级的交互存在问题。进一步检查相关信号后，发现是由于fs\_valid始终为1，导致该地址无法进入IF级。

（3）错误原因

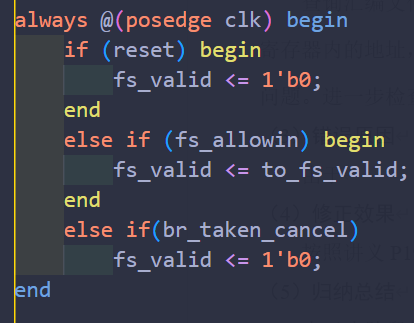
图7展示了原有的fs\_valid的赋值逻辑。可以看到，当清空流水线（ws\_reflush\_fs为1）时，三个条件都不符合。若此时fs\_valid为1，那么接下来fs\_valid将一直为1，导致next\_pc的地址永远无法进入IF级。

图7：错误2出错代码截图

（4）修正效果

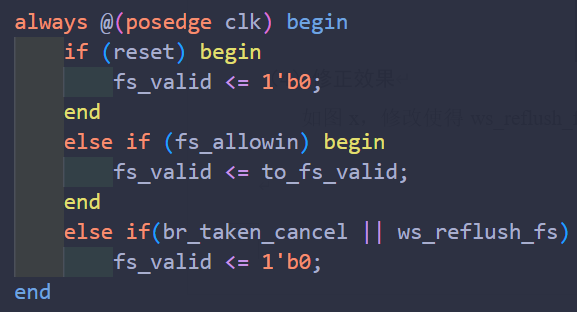
如图8，修改使得ws\_reflush\_fs为1时，fs\_valid变为0。这样便可以让ERA寄存器内的地址及时进入IF级，使流水线重新流起来。

图7：错误2修正后代码截图

（5）归纳总结

在exp13的设计中，由于IF阶段一定可以拿到ERA寄存器的地址，因此ertn指令清空流水线的操作可以只将其余阶段valid改为0，而保留fs\_valid为1；但在exp14中，由于握手机制的存在，next\_pc的地址不确定何时能进入IF阶段，因此需要将fs\_valid也及时改为0。可见随着后续功能的增加，之前实验的设计也要考虑是否相应地改变。

3、错误3：ld写后读相关未有效阻塞

（1）错误现象

如图8，仿真显示写回数据错误

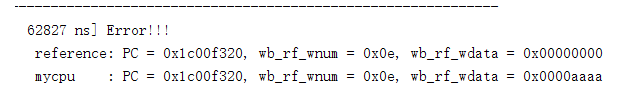


图8：错误3出错情况截图

（2）分析定位过程

首先查看汇编指令序列，如图9所示，发现出错指令是xor指令，其上一条指令是ld指令，且之间存在写后读相关。错误显示xor写回的数据出现问题，很大可能是ld指令前递的数据存在问题。



图9：错误3汇编指令序列

于是查看出错时刻附近两条指令的波形图，发现xor指令在ID阶段判断出写后读相关后，会发生阻塞，但阻塞到ld指令离开EXE阶段的下一拍就取消，而ld指令下一拍并没有取到正确的数据，因此前递的数据错误，导致xor得到的数据有误。

（3）错误原因

在之前的设计中，ld指令在MEM阶段只会停留一拍，因此在离开EXE阶段的下一拍就一定能得到正确的数据并前递给下一条指令，因此原有的ld前递逻辑是没有问题的；但在加入握手机制后，ld在MEM阶段可能很久才能得到正确的数据，因此一拍内无法得到正确的前递数据，会导致后面一条指令得到的操作数有误。

（4）修正效果

如图10，新增一个MEM阶段到ID阶段的信号，表明此时存在ld写后读，且ld指令在MEM阶段尚未取到正确的数据，需要让下一条指令在ID阶段继续阻塞。这样便可让xor指令在ID阶段阻塞直至获得正确的前递数据。

图10：错误3修正的主要内容

四、实验总结

还是得认真、反复读讲义。第一遍读讲义的时候有些说法看不明白，但是之后自己调代码遇到难以解决的bug时，往往重新翻阅一下讲义会有收获，甚至会直接得到解决方式。

另外，上板随机测试debug实在让人头疼……经常遇到某种子上板偶尔出错、但仿真仍然正确的问题，也没有合适、便捷的调试方法……感觉在本课程之后的实验设计中，总线随机验证需要后续改进，例如讲解Chipscope等上板调试的方法