

# 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）与指令（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。

#### 2、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	该次请求的地址传输 ok
data_ok	1	input	该次请求的数据传输 ok
rdata	32	input	该次请求返回的读数据

### 3、功能描述

在 SRAM 的基础上增加了握手机制，更加贴近真实情况。

## （三）重要模块 2 设计：类 SRAM-AXI 转接桥

### 1、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：

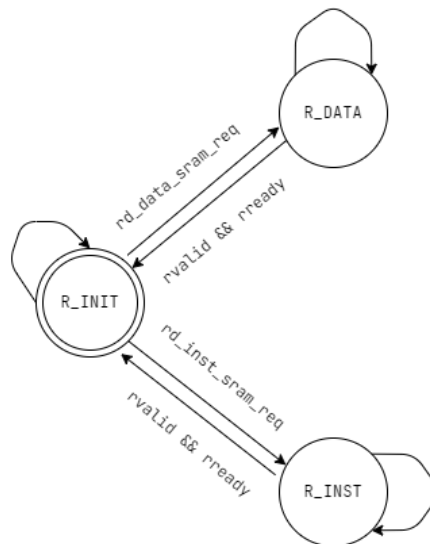


图 1：读状态机状态转移示意图

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：

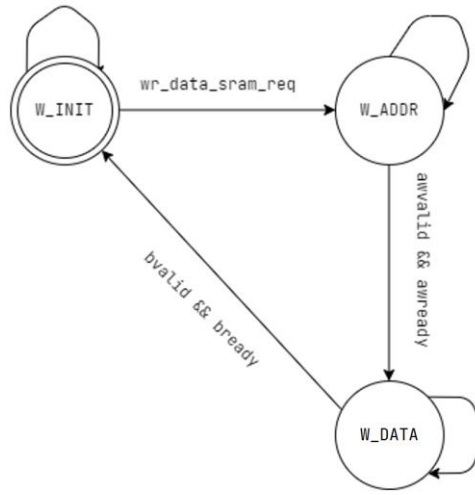


图 2：写请求与写数据状态机状态转移示意图

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：

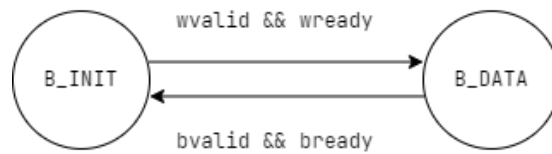


图 3：写响应状态机状态转移示意图

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
2022.11.13	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

```
-----
[1615857 ns] Error!!!
reference: PC = 0x1c017304, wb_rf_wnum = 0x0a, wb_rf_wdata = 0xc822c7e8
mycpu    : PC = 0x1c017304, wb_rf_wnum = 0x0a, wb_rf_wdata = 0x00000000
-----
```

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）

```
[6682000 ns] Test is running, debug_wb_pc = 0x1c00f078
[6692000 ns] Test is running, debug_wb_pc = 0x1c00f078
[6702000 ns] Test is running, debug_wb_pc = 0x1c00f078
[6712000 ns] Test is running, debug_wb_pc = 0x1c00f078
[6722000 ns] Test is running, debug_wb_pc = 0x1c00f078
```

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

```
always @(posedge clk) begin
    if (reset) begin
        fs_valid <= 1'b0;
    end
    else if (fs_allowin) begin
        fs_valid <= to_fs_valid;
    end
    else if (br_taken_cancel)
        fs_valid <= 1'b0;
end
```

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。

```
always @(posedge clk) begin
    if (reset) begin
        fs_valid <= 1'b0;
    end
    else if (fs_allowin) begin
        fs_valid <= to_fs_valid;
    end
    else if (br_taken_cancel || ws_reflush_fs)
        fs_valid <= 1'b0;
end
```

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。

## 四、实验总结（可选）

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。