具有汉明码矫正的 8 通道动态仲裁的同步 FIFO 设计

By: YunTing-K
https://github.com/YunTing-k/SyncFIFO

目录

具有汉明码矫正的 8 通道动态仲裁的同步 FIFO 设计

1. 设计要求与设计思路

1.1 设计要求概述

本项目要求实现一个同步 FIFO，其写入通过标准 APB3 协议的 APB Slave 进行，至少需要维护 FIFO 状态以及 FIFO 写入数据这两个寄存器。对于读出，通过 8-通道的仲裁器实现，其读出具有 8 个独立通道读出。对于 FIFO 的读写指针以及存储的内容，都需要进行汉明码的校验与矫正，对于读出，8 个通道具有动态可修改的权重和地址(Valid-Ready 握手)，读出内容依据 8 路通道对应的最大权重的地址决定，对于读出内容，至少可以读出 FIFO 数据或者读写指针数据以及汉明码校验信息等，如果读出了 FIFO 数据，FIFO 需要 POP，如果没有读出其他数据 FIFO 不需要 POP。

本节将会概述性的描述设计需求以及本项目的设计思路，不对设计进行具体的阐述。详细内容如时序，状态转移图等，详见**系统设计描述**小节。

1.2 FIFO 设计思路

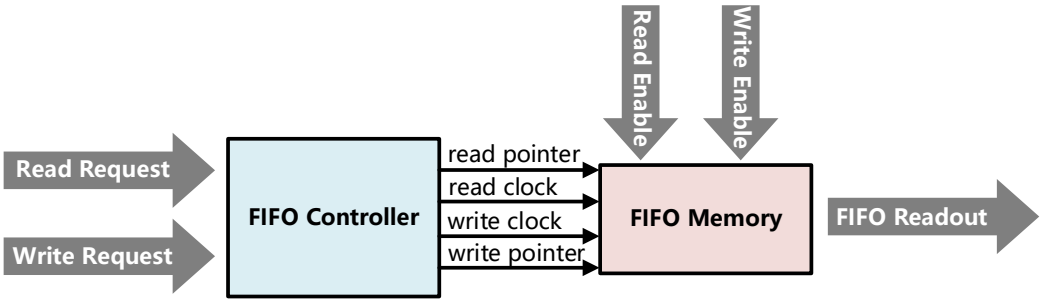


图 1. FIFO 外部交互与内部实现框图

图 1 给出了本项目的 FIFO 大致设计思路，由于读写分离(即写入请求是 APB 给出，读出请求是仲裁器 Arbiter 给出，读写请求来源不一致)，因此可先考虑实现 FIFO。

1.2.1 FIFO Controller 实现思路

FIFO 结构大致可由两部分组成，首先是 FIFO Controller，该模块控制 FIFO Memory 的读写，对于 FIFO Controller，其需要管理的参数有：FIFO 状态标志、读时钟、读时针、写时钟和写时针，对于同步 FIFO 设计，读写时钟一般同频同相是源自于同一个时钟信号。

读写指针具有以下行为：

- 每写入一次数据，写地址指针会加 1，每读取一次数据，读地址指针会加 1
- 当读地址指针追上写地址指针，FIFO 便是读空状态
- 当写地址指针再次追上读地址指针，FIFO 便是写满状态
- 空状态的读出(POP)是非法的，不能够影响读指针
- 满状态的写入(PUSH)是非法的，不能够影响写指针和栈顶的数据

- 当读写指针相同时，证明 FIFO 空或者满

因此 FIFO Controller 其需要管理 Read pointer 与 Write pointer 以正确对应 FIFO 的特性，即空 FIFO 不读，满 FIFO 不写，并且能够给出相应的信号指示 FIFO 目前的状态。为了使得维护简单，引入一个 FIFO 计数器(FIFO Counter)的方式来管理 FIFO 访存行为：

- 当只有写操作时，FIFO Counter 加 1
- 只有读操作，FIFO Counter 减 1
- 其他情况下，FIFO Counter 保持不变
- 当 FIFO Counter 为 0 时，说明 FIFO 为空
- 当 FIFO Counter 等于 FIFO 深度时，说明 FIFO 已满

此外为了更细致的指示 FIFO 状态，还划分了 1/4 空，2/4 空，3/4 空，全空等状态，这些状态的判断实现都是通过 FIFO Counter 与其本身深度大小比较获得的。

1.2.2 FIFO Memory 实现思路

基于前面 **1.2.1** 小节的 FIFO Controller 实现思路，FIFO Memory 被实现为一个双端口的 RAM，可以同时进行数据读出和写入。对于 FIFO 行为控制，还需要额外的读写使能信号，read/write enable 信号来完成。对于本项目来说，由于读写分离，来自于不同的设备，因此 read enable 信号来自于 Arbiter，write enable 信号来自于 APB Slave。

1.3 汉明码编解码实现思路

对于原始宽度$M\ bit$的待加密信息，汉明码编码需要$r\ bit$位冗余位宽，需要满足：

$$2^r \geq M + r + 1$$

以$M = 10\ bit$的原始信息为例，汉明码需要$r = 4\ bit$冗余位宽，因此编码后的数据位宽是$14\ bit$。汉明码是非破坏性的编码：即原数据可以在编码后的数据逐一找到且对应，冗余位是通过插入原始数据实现的(**图 2**)。

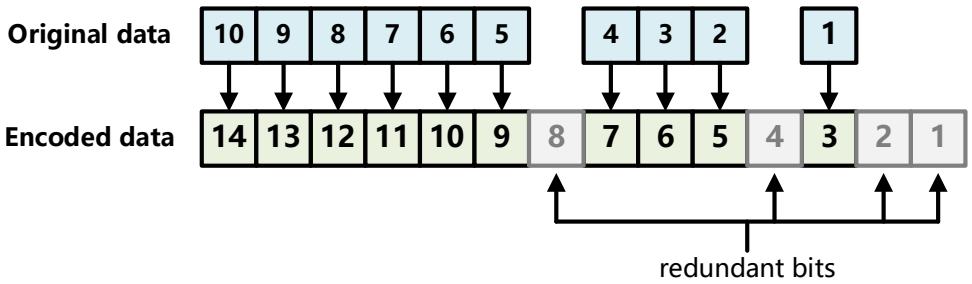


图 2. 10bit 为例的汉明码编码构成

而冗余位需要通过对原始数据进行分组奇偶校验得到：对于$r\ bit$冗余位宽的编码数据，一共有r个奇偶校验组，对于第i组(其中i满足$r + 1 > i > 0$)，其成员是编码后的数据(Encoded data)索引二进制的第i个下标为“1”的所有元素。

因此对于图 X 中的例子，具有以下 4 组：

- 第 1，3，5，7，9，11，13 元素 (第一组)
- 第 2，3，6，7，10，11，14 元素 (第二组)
- 第 4，5，6，7，12，13，14 元素 (第三组)
- 第 8，9，10，11，12，13，14 元素 (第四组)

由于每组通过码型设计只有一个冗余位，因此只需要该组元素满足奇校验或者偶校验就可得到冗余位数据。在本项目中，处于逻辑上的简单性，我们考虑偶校验。对于汉明码的解

码，只需要同样按照分组检验奇偶校验，如果某一组不满足奇偶校验，则标记该组，最后将所有组的标记情况转换为二进制数，该数即错误码的位置，通过翻转即可矫正。如对于以上情况，假设只有第一组出现了错误，标记情况为$0001_2 = 1$，说明第 1 个数错误。如果所有的组都没有错误，则标记情况为$0000_2 = 0$，说明第 0 个数据错误，但是由于不存在第 0 个数据错误，实际上表示数据通过了汉明码校验。

由于汉明码码型分组比较复杂，很难在 Verilog 中通过 generate 与 genvar 配合循环完成对任意长度的汉明码编码和解码，因此考虑使用硬编码完成汉明码的编码和解码。为了方便硬编码的编程，我们首先在 MATLAB 实现了三个函数和一个脚本来辅助我们的 Verilog 设计。源代码请见**附录**，详细内容请见 **2.4** 小节。

1.4 APB3 Slave 设计思路

对于 AMBA 高级处理器总线架构，不同的速率要求构成了高性能 SOC 设计的通信标准(没有列出 AXI 协议)：

1. AHB 先进高性能总线：主要是针对高速率、高频宽及快速系统模块所设计的总线，可以连接在如微处理器、芯片上或芯片外的内存模块和 DMA 等高效率总线
2. APB 先进外围总线：主要应用在低速且低频率的外围设备，可针对外围设备做功率消耗及复杂接口的最佳化

对于要实现的 APB 协议，其主要特点是其不是流水操作，不是全双工通讯，两个周期完成一次读或写的操作；APB 支持最大 32-bit 的数据位宽；APB 有两个独立的数据通道：读通道和写通道。由于 APB 的两个通道没有自己的握手信号(valid / ready)，因此两个通道不会同时使用，即不支持读写并行操作；APB 3 根据 PREADY 信号是否拉高，决定传输是否具有等待周期。在**表 1** 中给出了具体的信号列表，以及其描述

表 1. APB3 信号列表与描述

信号	来源	描述
PCLK	时钟源	时钟信号
PRESETN	系统总线	APB 接口异步复位信号，但必须等待 PCLK 的上升沿
PADDR	APB Bridge	地址线
PWRITE	APB Bridge	读写操作线，为 1 时写，为 0 时读
PSELx	APB Bridge 的译码器	APB 外围片选信号，抬高后进入 SETUP 状态，检查地址和控制信号，至少保持 2 个时钟周期
PENABLE	APB Bridge	允许读写操作信号线
PWDATA	APB Bridge	master 通过 PWDATA 线将数据写到 slave，数据最大支持 32bit
PRDATA	APB Slave	master 通过 PRDATA 线将数据从 slave 读取回来，数据最大支持 32bit
PREADY	APB Slave	APB slave 的响应信号，使能信号 PENABLE,会 在 ACCESS 状态中置位
PSLVERR	APB Slave	若拉高，则表示 APB 数据传输失败。 APB 的外围设备可以不使用这根线

在本设计中，由于 **top_wrapper.v** 模块没有预留 PSLVERR 信号，因此不考虑该信号以

及相关功能的实现。针对本项目的要求，APB Slave 主要进行以下两个关键寄存器的访存：

1. FIFO 写入寄存器的写入和读出访存

2. FIFO 状态的读出访存(FIFO 状态显然是只读的)

除此之外之外，基于 **1.4** 小节的描述，APB Slave 还应该具有控制功能，在真正需要写入 FIFO 的时候对 FIFO 模块的关键信号 **write enable** 信号进行调控，需根据 FIFO 的状态进行 FIFO 功能的维护：

1. 如果 FIFO 满，需要挂起该请求，直到 FIFO 不满才进行数据的写入，控制 **write enable** 信号进行 FIFO 写入

2. 如果 FIFO 不满，则直接控制 **write enable** 信号进行 FIFO 写入

3. 此外，由于需要满足标准的 APB3 时序，因此还需要具有如下特性：

4. 能够完成连续写入数据以及有等待的写入数据

5. 能够完成连续读出数据以及有等待的写入数据

6. (当访问寄存器越界时，能够正确处理)

为了满足以上要求，可以通过 FIFO 模块的 FIFO status 输入到 APB 模块中，根据该信号判断 FIFO 是否满，进而控制状态机决定是立即控制写入还是等待 FIFO 不满。状态机拟采用三段式状态机设计，以方便维护。具体设计细节请见 **2.3** 小节。

1.5 仲裁器设计思路

仲裁器是一个具有八通道输出的，动态权重以及数据读出可通过 Valid-Ready 数据握手通讯的读出设备。因此仲裁器首先需要满足如下设计特性：

1. 每一路都有单独的 Valid-Ready 握手，用以配置目标读出地址，通道此次请求的优先级权重 priority

2. 对于同一时刻，同时请求的通道，优先响应权重最高的请求

3. 对于读出 FIFO 数据的，需要正确控制 FIFO 的 POP 操作

4. 对于不是读出 FIFO 数据的，需要正确控制 FIFO 只读，不进行 POP 操作

基于 **1.2** 小节的描述，FIFO 控制信号的读使能信号是由 Arbiter 给出的，因此对应通道的被仲裁选择后，还需要进行 FIFO 控制。

1.5.1 最值查找器件设计思路

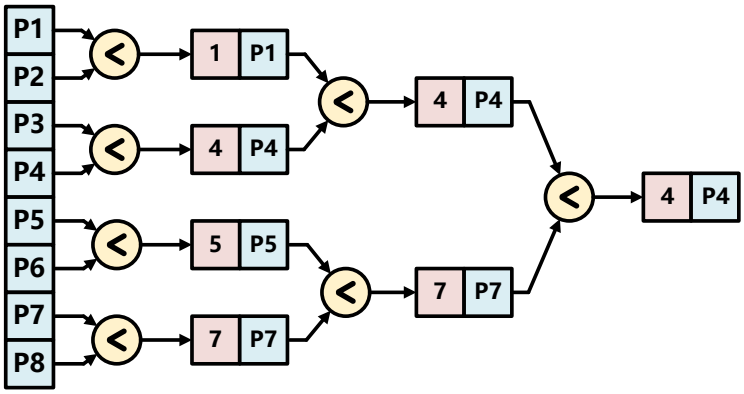


图 3. 最值查找设计框图以 P4 最大为例

由于仲裁需要优先响应最大优先级的通道，需要对优先级进行最值查找，获得最大优先级的通道对应的下标。**图 3** 展示了本项目中的最值查找器的设计框图，对于 8 通道的最值查

找，我们需要进行三级比较，通过逐级两两比较，输出最大值对应的权重以及下标到下一级，最终通过三级比较获得优先级最大的通道对应的下标。该设计可以仅仅通过组合逻辑实现，且组合逻辑较简单，不需要较多的时钟周期开销。

1.5.2 仲裁的设计思路

由于八路仲裁权重动态变化，为了实现无竞争，保证所有的请求都必须被响应，设计思路如下：

1. 对于每一路通道的优先级，通过 Valid-Ready 握手更新后，将更新结果同步到寄存器中，并且输出到 **1.5.1** 小节中的最值查找器中得到所有通道的最优先通道的下标

2. 基于 **1** 中得到的下标，选通对应通道，如果一个通道没有被选通则对应以下情况：

- 通道没有读出请求，因此没有被选通是符合仲裁器的设计逻辑的
- 通道有读出请求，但是因为在当前还有比它更优先的通道，因此没有被选通

3. 对于选通的通道，进行访存操作，如果读取 FIFO 数据，则对 FIFO 进行 POP，否则不进行 POP

4. 完成访存后，该通道的优先级寄存器置零，以便将控制权移交给其他的通道

因此，要实现无竞争的通道仲裁，还需要保证具有请求的通道的 priority 不是 0，0-priority 是仲裁器保留的标识与置位符号。此外，如果仲裁器的某个通道试图在 FIFO 空的时候读出(POP)，按照 **1.2** 小节的描述，不会操作读指针，但是会返回最后一次读出的值。通过该设计思路，保证了在通道被仲裁时满足如下特性：

1. 最值比较模块保证了——最高优先级通道最先响应

2. 优先级寄存器置零保证了——所有请求一定会被响应，而不会被覆盖

3. 控制 FIFO 信号只能由当前被仲裁的通道控制发出保证了——对于 FIFO 以及 Reg 的访存数据，一定会送入对应的通道中，不会因为过于邻近的请求(如同一时刻多个同样优先级仲裁，或者每隔一个时钟周期，后来者拥有比前一个更高的优先级导致的抢占)导致数据发送混淆

2. 系统设计描述

2.1 顶层架构与关键信号描述

接下来根据**设计要求与设计思路**小节描述的设计思想，分别对系统的模块设计进行详细阐述。**图 4** 展示了本项目所实现的系统的顶层架构，本项目的顶层模块为给定的 **top_wrapper** 模块，并且在其中例化了三个模块，分别是 APB Slave 模块、FIFO Wrapper 模块以及 Arbiter 模块。

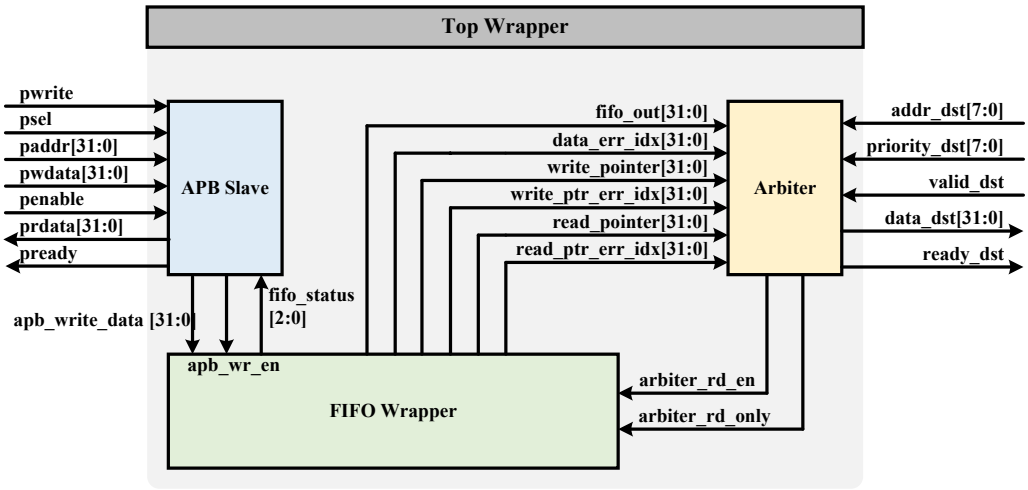


图 4. 系统顶层架构 (8 路读出系统绘图化简为 1 路，实际实现为 8 路)

APB Slave 通过接受 APB Master 的控制，进行 FIFO 数据、FIFO 状态寄存器的访存和维护，并且根据指令控制以及 **fifo_status** 状态信号通过产生写入数据 **apb_write_data** 信号与写使能信号 **apb_wr_en** 信号控制 FIFO。

Arbiter 通过 Ready-Valid 进行动态仲裁，通过 **rd_en** 信号与 **rd_only** 信号对 FIFO 进行 POP 的读取操作或者非 POP(nPOP，即只读取不读出)操作。并将对应访存寄存器数据经过仲裁与获取，送至 **data_dst** 至外部。

FIFO Wrapper 模块通过接受 APB Slave 与 Arbiter 模块的控制实现 FIFO 功能正常读写，同时通过汉明码编码解码，将解码后的数据、读写指针以及数据、读写指针对应的错误下标送入寄存器以便被 Arbiter 正常访问并送出。

表 2 给出了以上关键信号的属性：

表 2. 系统关键信号属性与描述

信号名	位宽	Direction	描述
clk	1	IN	系统时钟
reset_n	1	IN	系统复位信号，低有效
pwrite	1	IN (to APB Slave)	APB 读写标志
psel	1	IN (to APB Slave)	APB 片选信号
paddr	32	IN (to APB Slave)	APB 地址信号
pwdata	32	IN (to APB Slave)	APB 写入数据
penable	1	IN (to APB Slave)	APB 使能信号

prdata	32	OUT (from APB Slave)	APB 读出信号
pready	1	OUT (from APB Slave)	APB Slave 完成信号
apb_write_data	32	APB Slave to FIFO Wrapper	送入 FIFO 数据
apb_wr_en	1	APB Slave to FIFO Wrapper	FIFO 写使能
fifo_status	3	FIFO Wrapper to APB Slave	FIFO 状态信号
addr_dst	8	IN (to Arbiter)	仲裁器通道读出地址
priority_dst	8	IN (to Arbiter)	仲裁器优先级
valid_dst	1	IN (to Arbiter)	仲裁器通道 valid 信号
data_dst	32	OUT (from Arbiter)	仲裁器通道读出信号
ready_dst	1	OUT (from Arbiter)	仲裁器通道完成信号
fifo_out	32	FIFO Wrapper to Arbiter	FIFO 读出解码后的数据
data_err_idx	6	FIFO Wrapper to Arbiter	数据误码位置
write_pointer	10	FIFO Wrapper to Arbiter	解码后的写指针
write_ptr_err_idx	4	FIFO Wrapper to Arbiter	写指针误码位置
read_pointer	10	FIFO Wrapper to Arbiter	解码后的读指针
read_ptr_err_idx	4	FIFO Wrapper to Arbiter	读指针误码位置
arbiter_rd_en	1	Arbiter to FIFO Wrapper	FIFO 读使能
arbiter_rd_only	1	Arbiter to FIFO Wrapper	FIFO 只读信号

2.2 设计文件与例化关系

本项目设计文件包含提供的 **top_wrapper.v** 在内，一共有 14 个文件。**表 3** 给出了本文件的设计文件极其功能描述：

表 3. 设计文件与功能描述

文件名称(模块名称.v)	功能描述	例化层级
top_wrapper.v	最顶层模块，例化了 FIFO，APB Slave 和 Arbiter	0
fifo_wrapper.v	FIFO 顶层模块，例化了控制器，存储器与汉明编解码	1
apb_slave.v	APB Slave 模块，完成 FIFO 写入与寄存器访存	1
arbiter.v	Arbiter 模块，例化了读出通道，FIFO 读控制与最值查找	1
fifo_ctrl.v	FIFO 控制模块，容量计数与读写指针维护	2
fifo_mem.v	FIFO 存储器，一共同步双端口 RAM	2
ptr_encode.v	读写指针的汉明编码模块	2
ptr_decode.v	读写指针的汉明解码模块，矫正并给出错误位下标	2
data_encode.v	存储数据的汉明编码模块	2
data_decode.v	存储数据的汉明解码模块，矫正并给出错误位下标	2
fifo_reg.v	将 Arbiter 要访存的数据寄存，避免组合逻辑输出的毛刺	2
find_max.v	最值查找器，对配置的优先级寄存器排序，仲裁最优	2
read_channel.v	读出通道的 Valid-Ready 握手与 FIFO 访存控制	2
fifo_read.v	FIFO 读出信号片选，保证只有仲裁通道才可控制	2

同时在**图 5** 中给出了本项目设计文件的具体例化关系。

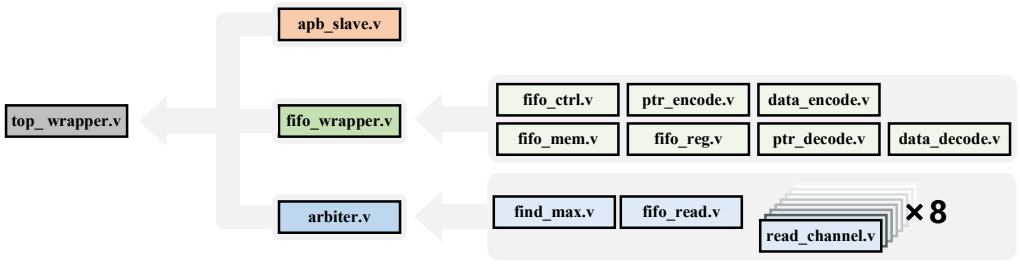


图 5. 设计文件例化关系

2.3 APB Slave 模块设计

APB Salve 模块需要满足的时序如**图 6** 所示，对于写入：

- 当进入 access 状态时，如果 **PREADY** 信号为高，那么传输就不会等待，会立即在 **PENABLE** 拉高的周期内完成传输。
- 写数据线(**PWDATA**)是同写信号线变化(**PWRITE**)的，而不是同 **PENABLE** 信号线变化(区别于读操作)
- **PENABLE** 在写操作周期结束后，会同 PSEL 一同拉低；

如果需要进行连续写入，**PSEL** 会在 **PREADY** 下降时保持为高，此时 APB Master 需要更新新的数据到 **PWDATA** 与 **PADDR** 中，此时 APB Slave 不再进入 IDLE 状态。

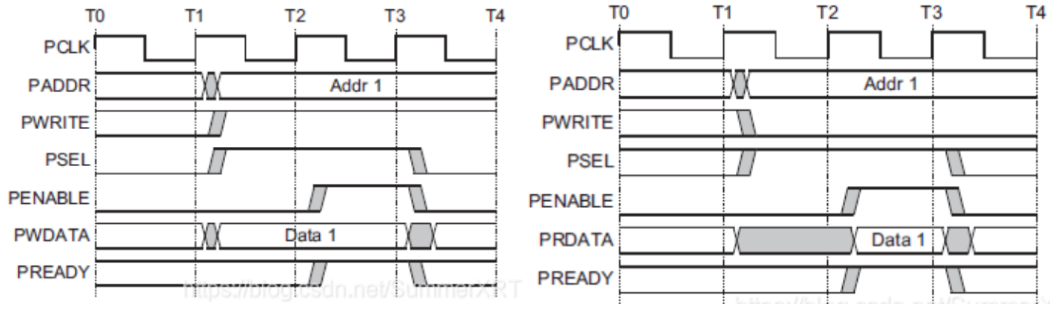


图 6. APB3 无等待写入与读取时序

对于读取：

- 当总线为 ACCESS 状态时，如果 **PREADY** 保持为高，传输会立即进行，在一个时钟周期内完成传输

如果需要进行连续的读取，同样 **PSEL** 会在 **PREADY** 下降时保持为高，此时 APB Mater 需要准备新的数据到 **PADDR** 中。

在**图 7** 中展示了所设计的状态机时序，各个状态具体功能描述如下:

- **IDLE：**当复位后，系统状态处于空闲态，直到输入信号 **PSEL** 为高时将会使得系统进入 SETUP 状态，否则停留在 IDLE 状态
- **CONFIG：**在该状态，对根据系统的输入的地址进行验证，如果超过了访存范围，则认为访存非法，**access_valid** 置 0 返回到 IDLE 状态。如果访存合法，则等待 **PREADY** 置高，**PREADY** 置高取决于 FIFO 是否是满的，如果 FIFO 满且 APB 要写入数据，则 **PREADY** 仅仅会在 FIFO 不满时置高；其余情况 **PREADY** 会置高进入 ACCESS 状态
- **ACCESS：**在该状态，如果访存未完成会停留在此状态，如果访存完成，且 **PSEL=0**

说明没有连续访存事件，则进入 IDLE 状态，如果 **PSEL** 等于 1 并且访存完成，已经正常读出或写入数据，则说明有连续写入/读取事件，进入 SETUP 状态

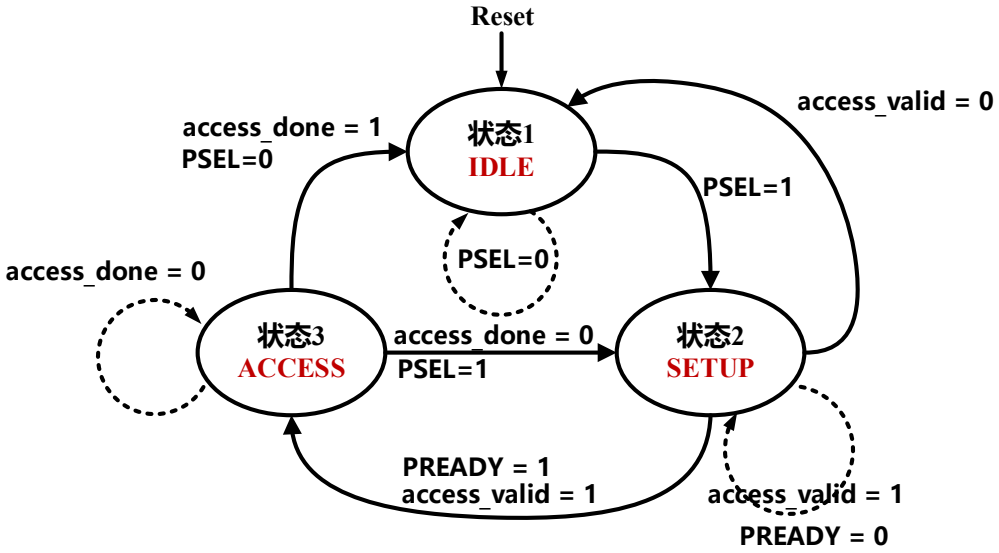


图 7. APB Slave 状态机

表 4 给出了寄存器地址偏移定义：

表 4. APB Slave 寄存器地址定义

寄存器/地址名称	地址
FIFO_BASE_ADDR	(基地址) 32'h2000 0000
FIFO_WRITE_DATA	(偏移) 32'h00
FIFO_STATUS	(偏移) 32'h04

2.4 汉明编码解码模块设计

通过在**附录**中的 MATLAB 代码，实现了汉明码分组的获取、汉明码编码与汉明码解码。通过利用该 MATLAB 程序，辅助我们进行硬编码实现汉明码编解码。

通过运行脚本：

```
[redundacy_bit, group, index] = gen_group(10)
```

得到输出：

```
redundacy_bit =
     4
group =
  3×4 cell 数组
    {[1  3  5  7  9  11  13]}    {[2  3  6  7  10  11  14]}    {[4  5  6  7  12  13
14]}    {[8  9  10  11  12  13  14]}
    {[  3  5  7  9  11  13]}    {[  3  6  7  10  11  14]}    {[  5  6  7  12  13
14]}    {[  9  10  11  12  13  14]}
```

```
    {[    1  2  4  5  7  9]}    {[    1  3  4  6  7  10]}    {[    2  3  4  8  9  10]}
{[      5  6  7  8  9  10]}
index =
  1×2 cell 数组
    {[3  5  6  7  9  10  11  12  13  14]}    {[1  2  3  4  5  6  7  8  9  10]}
```

说明对于长度为 10 的读写指针(本设计 FIFO 深度为 1024)，冗余位为 4，并且分组情况也一并得到。对于 group：第一行，包括了冗余位，不同的组在编码后数据的索引，第二行，不包括冗余位，不同的组在编码后数据的索引，第三行，原始数据从低到高对应在编码后的数据的索引。对于 index：第一行，编码后，非冗余位在编码后数据的索引，第二行，编码后非冗余位对应的编码前的数据的索引。(下面给出了冗余位的实现描述)

```
assign parity_0 = (raw_data[0] + raw_data[1]) + (raw_data[3] +
raw_data[4]) + (raw_data[6] + raw_data[8]); // raw 0 1 3 4 6 8 ==>
redundancy bit 0
assign parity_1 = (raw_data[0] + raw_data[2]) + (raw_data[3] +
raw_data[5]) + (raw_data[6] + raw_data[9]); // raw 0 2 3 5 6 9 ==>
redundancy bit 1
assign parity_2 = (raw_data[1] + raw_data[2]) + (raw_data[3] +
raw_data[7]) + (raw_data[8] + raw_data[9]); // raw 1 2 3 7 8 9 ==>
redundancy bit 2
assign parity_3 = (raw_data[4] + raw_data[5]) + (raw_data[6] +
raw_data[7]) + (raw_data[8] + raw_data[9]); // raw 4 5 6 7 8 9 ==>
redundancy bit 3
```

通过 MATLAB 脚本，对于长度为 32 的数据(FIFO 数据宽度为 32)进行类似计算，帮助我们硬编码设计。

2.5 FIFO 模块设计

对于 FIFO 模块，以读指针的维护为例子：

```
// generate read address
always @(posedge clk or negedge rst_n) begin
    if (rst_n == 1'b0) begin
        rd_addr <= {ADDR{1'b0}};
    end
    else if ((!empty_flag) && fifo_pop) begin // this moment, addr
is the current pointer
        rd_addr <= rd_addr + 1'b1;
    end
    else begin
        rd_addr <= rd_addr;
    end
end
```

只有当 FIFO 非空和 FIFO 确定要 POP 时，才修改读取指针，而对于是否 POP，信号定

义如下：

```
assign fifo_pop = (rd_en && (!rd_only)) ? 1'b1:1'b0;
```

只有在读取使能并且只读时才 POP。对于写入指针维护同理。最后通过维护 **fifo_counter** 实现 FIFO 状态的估计：

```
// update fifo data amount
always @(posedge clk or negedge rst_n) begin
    if (rst_n == 1'b0) begin
        fifo_counter <= {(ADDR+1){1'b0}};
    end
    else if ((!full_flag) && wr_en && (!fifo_pop)) begin // not
full, enable write, no read >> data increase
        fifo_counter <= fifo_counter + 1'b1;
    end
    else if (!empty_flag && (!wr_en) && fifo_pop) begin // not
empty, no write, enbale read >> data decrease
        fifo_counter <= fifo_counter - 1'b1;
    end
    else begin // other case: amount keep the same
        fifo_counter <= fifo_counter;
    end
end
```

2.6 Arbiter 模块设计

对于 Arbiter 模块，关键模块是最值查找和读取通道的设计，对于最值查找，按照 **1.5.2** 节的思路，通过 Assign 结合三目运算符，可以实现三级比较完成最值查找，最终得到索引以及独热码表示的选择信号 **select**：

```
// parallel compare [first layer priority and index gen]
assign pri_l1_0 = (priority_0 >= priority_1) ?
priority_0:priority_1;
assign idx_l1_0 = (priority_0 >= priority_1) ? 3'd0:3'd1;
...
// parallel compare [second layer priority and index gen]
assign pri_l2_0 = (pri_l1_0 >= pri_l1_1) ? pri_l1_0:pri_l1_1;
assign idx_l2_0 = (pri_l1_0 >= pri_l1_1) ? idx_l1_0:idx_l1_1;
...
// third layer index gen
assign idx_l3 =  (pri_l2_0 >= pri_l2_1) ? idx_l2_0:idx_l2_1;
// one-hot output selction signal gen
assign select = (8'b0000_0001 << idx_l3);
```

对于 Read channel 模块，其需要实现 Valid-Ready 握手通讯，**图 8** 展示了其时序需求。在本项目中，需要对 Channel 的优先级进行配置，并且还需要根据优先级仲裁并得到输出结

果，因此采用有等待的 **READY** 时序：只有通道被配置好并且请求响应完成/数据备好才会拉高 **READY** 信号，否则该请求会被一直挂起，直到被响应或者数据可以读出。

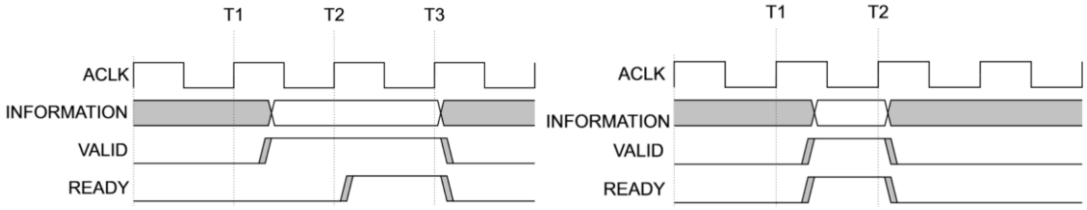


图 8. Valid-Ready 握手时序 (有等待响应和同时响应)

状态机如**图 9** 所示，各个状态具体功能描述如下:

- **IDLE**：当复位后，系统状态处于空闲态，直到输入信号 **valid** 为高时将会使得系统进入 SETUP 状态，否则停留在 IDLE 状态
- **CONFIG**：在该状态，对根据系统的输入的地址以及优先级进行配置，配置完成如果还是被选中**(sel = 1)**，说明仲裁成果，进入输出控制 CONTROL 状态
- **CONTROL**：在该状态对 FIFO 读取信号进行控制
- **WAITDATA**：在该状态，等待信号从地址输入，FIFO 读出再通过汉明码解码输出到寄存器更新完毕，避免读出错误数据
- **OUTPUT**：这一阶段对获取的地址编码，输出对应寄存器的内容，对于超出地址范围的访存，默认是 POP 出 FIFO 的内容(对读指针有修改的 FIFO 读出)

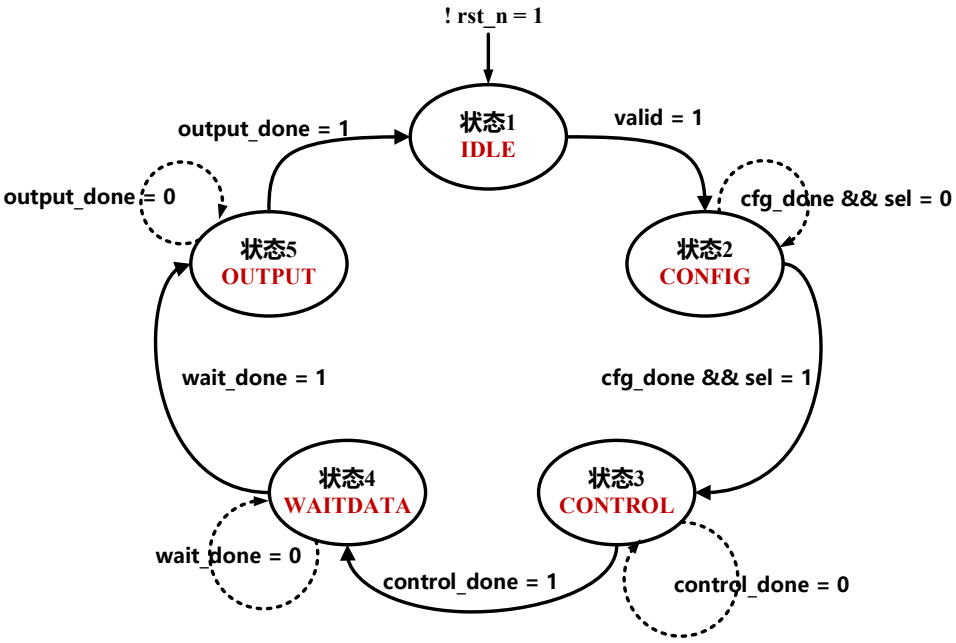


图 9. 读出通道状态机

同时，**表 5** 给出了读出寄存器的地址定义：

表 5. Arbiter 读出地址寄存器定义

寄存器名称	地址
FIFO_OUT_DATA	0
DATA_ERR_IDX	1

WRITE_PTR	2
WRITE_PTR_ERR_IDX	3
READ_PTR	4
READ_PTR_ERR_IDX	5

3. 设计验证与结果分析

3.1 Testbench 概述与文件描述

本项目最终编写了 5 个不同的 testbench 文件，作用如**表 6** 所示：

表 6. Testbench 文件与描述

文件名称(模块名称.v)	功能描述
apb_io_tb.v	APB 读取的测试
arbiter_read_tb.v	仲裁器件读取的测试
data_read_tb.v	FIFO 数据读出测试
data_write_tb.v	FIFO 数据写入测试
hanming_tb.v	汉明码编解码测试

对 design files 的 14 个文件以及编写的 5 个 testbench 在 ModelSim 中进行编译，最后编译结果如**图 10** 所示。

```
# Loading project SyncFIFO_IMPL
# Compile of apb_slave.v was successful.
# Compile of arbiter.v was successful.
# Compile of data_decode.v was successful.
# Compile of data_encode.v was successful.
# Compile of fifo_ctrl.v was successful.
# Compile of fifo_mem.v was successful.
# Compile of fifo_read.v was successful.
# Compile of fifo_reg.v was successful.
# Compile of fifo_wrapper.v was successful.
# Compile of find_max.v was successful.
# Compile of ptr_decode.v was successful.
# Compile of ptr_encode.v was successful.
# Compile of read_channel.v was successful.
# Compile of top_wrapper.v was successful.
# Compile of data_write_tb.v was successful.
# Compile of apb_io_tb.v was successful.
# Compile of data_read_tb.v was successful.
# Compile of arbiter_read_tb.v was successful.
# Compile of hanming_tb.v was successful.
# 19 compiles, 0 failed with no errors.
```

图 10. 设计文件与测试文件编译结果

3.2 汉明码编解码测试

图 11 展示了汉明码编解码测试的时序图，首先根据编码解码 0xFF0000FF，最后编码与解码均正确，此时没有引入错误。随后我们进行对第 5 位利用 **force** 语句进行翻转，最后的 **err_index** 显示值 4 (即 5-1)，并且解码内容正确，说明成功纠错。对 10bit 同样进行编码译码与错误校验，同样通过，都会返回错误下标-1 以及纠正后的数据(**图 12**)。

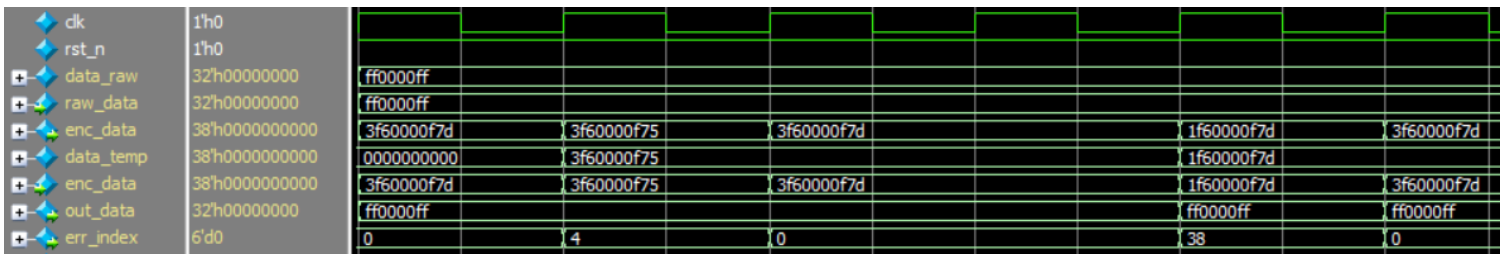
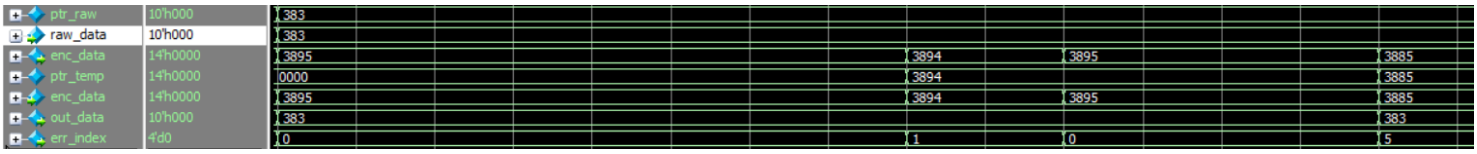


图 11. 汉明码编解码测试 32-bit

图 12. 汉明码编解码测试 10-bit



3.3 FIFO 写入测试

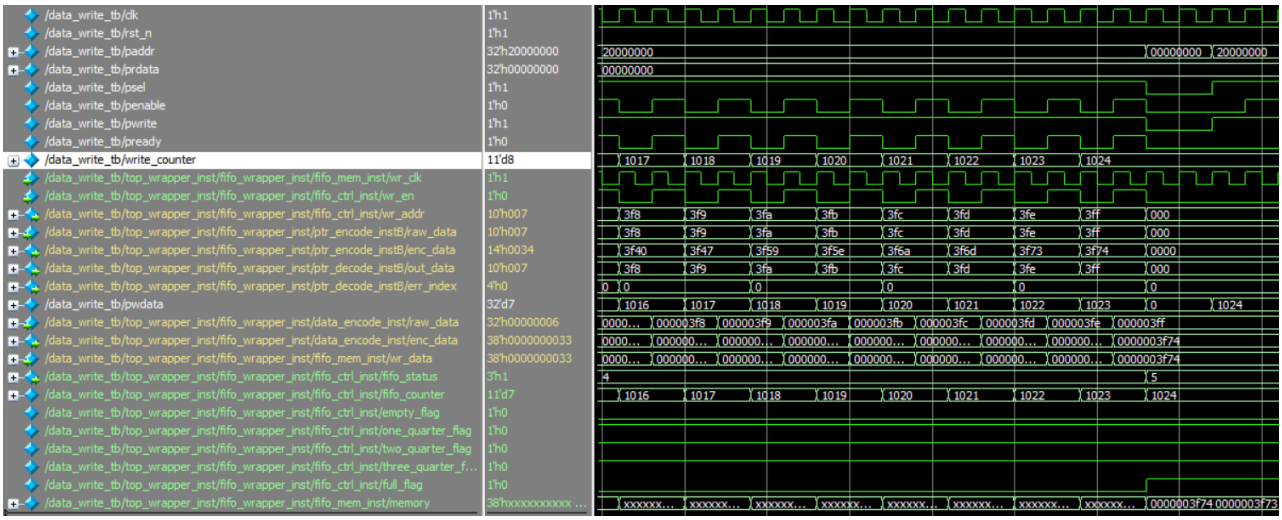


图 13. FIFO 写入测试 APB 连续写入

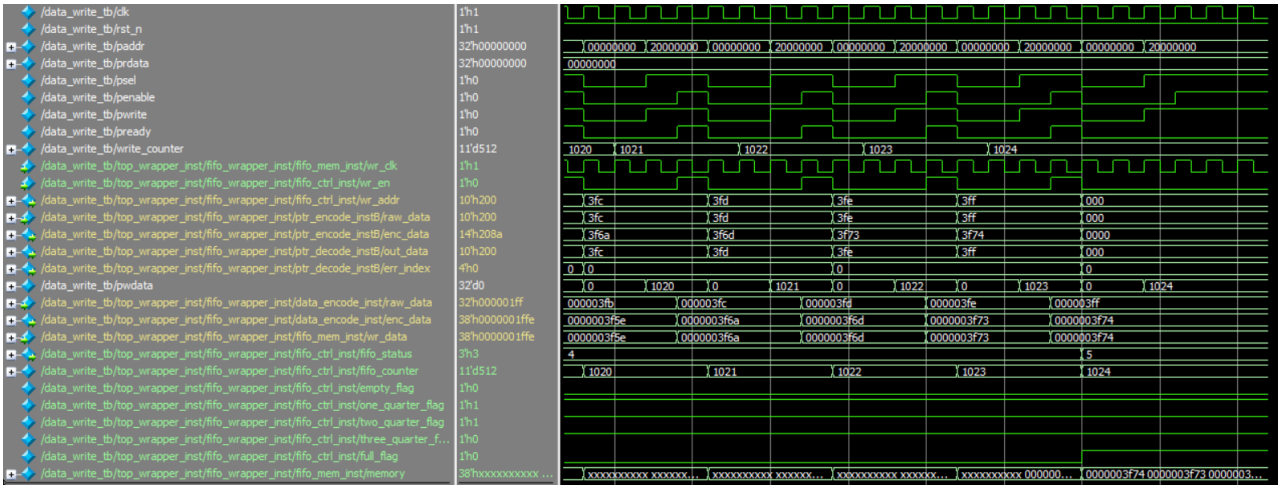


图 14. FIFO 写入测试 APB 非连续写入

通过 APB 进行 FIFO 写入测试(依次写入 0~1023)，只进行写入不进行读出，可从时序图看到，写入 1024 个数据后，再尝试写入数据 APB 请求被挂起，**pready** 不置高响应请求，此时 **fifo_status** 为 5，说明全满。此时 FIFO 没有修改读写指针与存储器内容完成了对 FIFO 的保护。同时还从**图 13** 中看到，写入是连续写入的时序(2 周期一次)，说明连续写入行为正确。同时**图 14** 的结果表明非连续写入也支持良好。

3.4 APB IO 测试

为了测试 APB 的读出和写入，还进行了 APB IO 测试，对于正确的访存可以正常执行(如 FIFO 数据的写入读出，FIFO 状态寄存器的读出)，对于尝试写入只读寄存器 **fifo_status**，无响应，不会改变系统状态如读写指针与存储器内容等。而对于试图访问 reg 外的地址，APB Slave 自动进入 IDLE 状态不再对本次请求进行响应，需要 APB Master 重新发起一共新的请求才会再次响应(如**图 15** 中地址 0x200000FF 越界了，APB Slave 无响应，直到新的请求发出后才再次响应)：

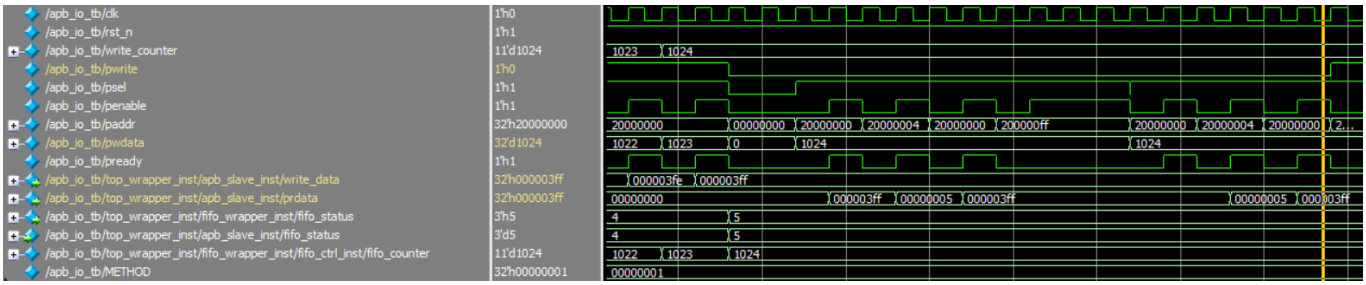


图 15. APB IO 测试

同时从**图 15** 可知，APB 的读出也依然同时支持 APB 的连续读写(两周期一访存)与非连续读写，证明是标准的 APB3 协议的时序。

3.5 FIFO 读出测试

为了测试从 Arbiter 读出 FIFO 数据，先对 FIFO 依次写入 0~1023。然后只选通第 0 个通道，其余通道不发送请求进行读出测试，写满 FIFO 后再逐个读出，先读出非 POP 内容(非 FIFO 存储内容的寄存器，分别是存储数据错误下标，写指针，写指针错误下标，读指针，读指针错误下标)，再读出 POP 内容(是 FIFO 存储内容的寄存器，即 FIFO 数据读出并解码后的输出)。直到 FIFO 空后再次尝试 POP，此时 FIFO 没有修改读写指针与存储器内容，并且读出最后一次 POP 内容(data 读出两次 1023)，完成了对 FIFO 的保护(**图 16**)。

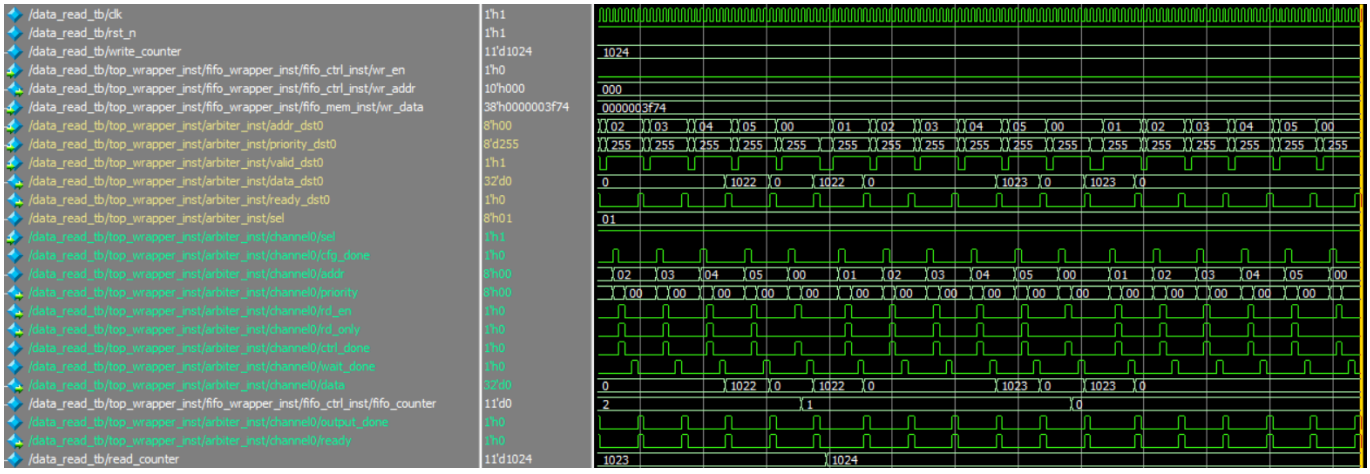


图 16. FIFO 读出测试

3.6 Arbiter 测试

为了测试仲裁模块，首先进行了同一时间同样权重的仲裁测试，从**图 17** 可以看见，由于权重相同，最后的优先顺序按照 channel 的序号排列(这是设计里导致的特性)，并且最终所有的通道都完成了响应，FIFO 成功读出了 8 个数据(0~7)。

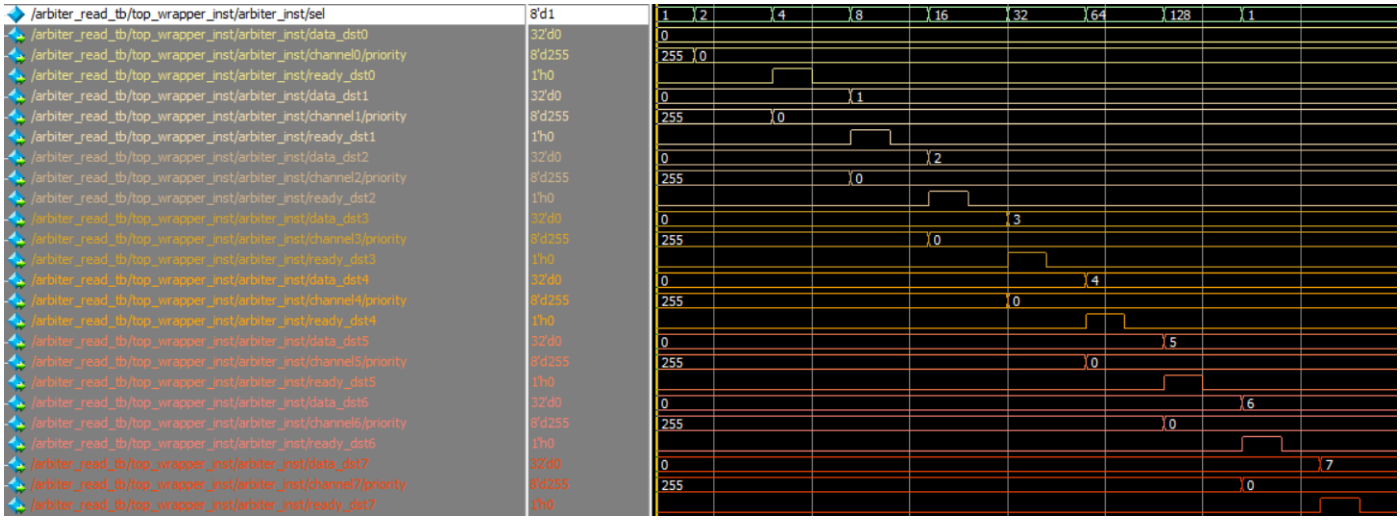


图 17. Arbiter 仲裁结果，8 通道同时请求，优先级相同

然后进行了 8 通道请求同时发送，但是越靠后的优先级越高，即后面的通道会抢占前面的通道控制权。从**图 18** 可以看见，还是仲裁正常，依次从 7~0 通道读出了 9~15 共 8 个数据。

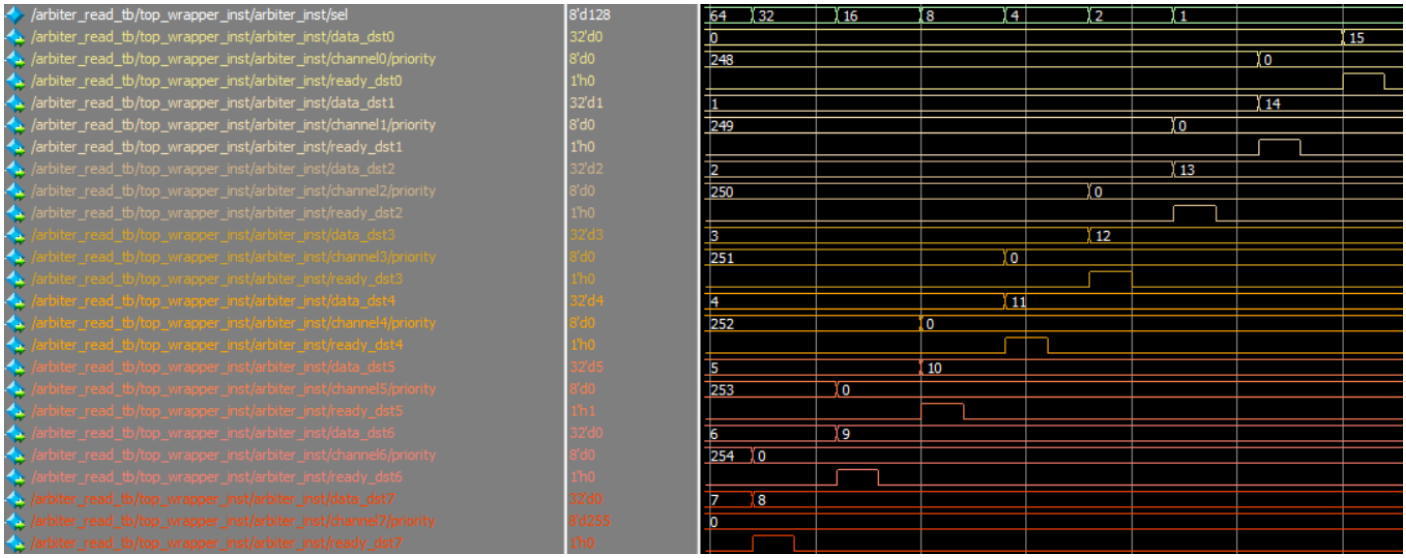


图 18. Arbiter 仲裁结果，8 通道请求同时发送，但是越靠后的优先级越高

然后进行同时读写操作，写 1 次，然后再同时写 1 次并读出 3 次(一共写 2 读 3)。读出时，只有通道 0，1，7 进行 POP 读出，其他都是非 POP 读出。同时后面的通道会抢占前面的通道控制权。最后 FIFO counter 归 0，符合预期，抢占结果符合预期。检查读出内容与 FIFO 存储器对比，结果依次读出 1023，1023，1024 的预期(**图 19**)。

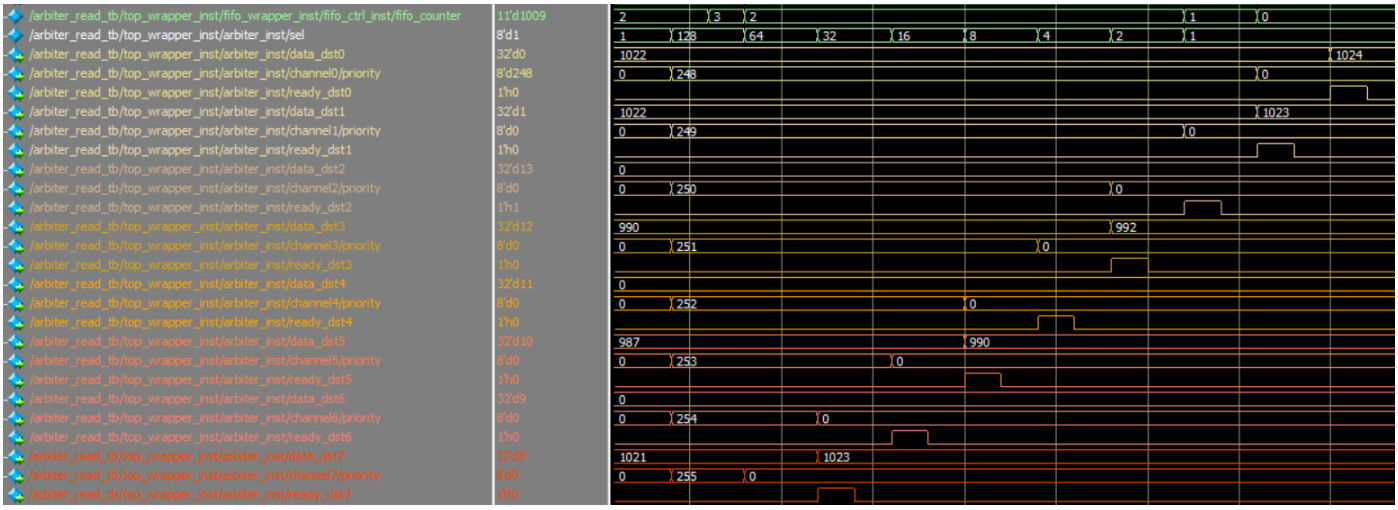


图 19. 写 2 读 3，后面通道抢占前面优先级

4. 总结

本项目成功实现了一共 8 通道动态仲裁的(支持 255 权重级别，0 权重被系统预留，可配置权重为 1~255)支持仲裁抢占(后来的请求可以抢占先来的低优先级的请求)的，32bit 宽度，1024 深度，APB3 协议(支持连续读写，最快 2 周期一次访存)，汉明码编解码矫正的同步 FIFO。对于 Arbiter 的读出，5 周期一访存，未来也可以采取类似 APB 连续读写方式产生连续的数据流加快读出速度。

整个系统的关键路径显然在汉明码编码和解码上，可以通过流水线加速该过程提高系统时钟频率，为此，本项目的设计不需要大幅度修改：流水优化后，只需要调整/延长 **read_channel** 模块里的 **wait_counter** 等待时间即可，在设计时预留了调整空间。

整体设计除了汉明码编解码外，设计模块都支持参数化设计，可以调整 FIFO 具体的深度和位宽。对于汉明码编解码模块，也编写了对应的 MATLAB 脚本，在未来，还可以进一步完善脚本，做到给定编码/解码长度和信号名称，直接输出 verilog 设计文件。

5. 附录

5.1 gen_group.m

```
%=======================
% 给定待编码数据长度，计算分组索引
%=======================
function [redundacy_bit, group, index] = gen_group(data_len)
%求得冗余位的位宽
redundacy_bit = 0;
while(2^(redundacy_bit)<(data_len + 1 +redundacy_bit))
    redundacy_bit = redundacy_bit + 1;
end

group = cell(3,redundacy_bit);
%第一行，包括了冗余位，不同的组在编码后数据的索引
%第二行，不包括冗余位，不同的组在编码后数据的索引
%第三行，原始数据从低到高对应在编码后的数据的索引
index = cell(1,2);
%第一行，编码后，非冗余位在编码后数据的索引
%第二行，编码后非冗余位对应的编码前的数据的索引
temp_list1 = [];
temp_list2 = [];
for i = 1:redundacy_bit
    list1 = [];
    list2 = [];
    list3 = [];
    for j = 1:2^redundacy_bit
        if j > (data_len + redundacy_bit)
            break;
        end
        bin_str = fliplr(dec2bin(j,redundacy_bit));
        if strcmp(bin_str(i), '1')
            list1 = [list1, j]; %#ok<*AGROW>
            if ~strcmp(erase(bin_str,'1'), dec2bin(0,redundacy_bit-
1))
                list2 = [list2, j]; %#ok<*AGROW>
                list3 = [list3, j - fix(log2(j)) - 1];
            end
        end
    end
    group{1,i} = list1;
```

```
    group{2,i} = list2;
    group{3,i} = list3;
    temp_list1 = [temp_list1,list2];
    temp_list2 = [temp_list2,list3];
end
[uni_value, uni_tag, ~] = unique(temp_list1);
index{1,1} = uni_value;
index{1,2} = temp_list2(uni_tag);
end
```

5.2 encode.m

```
%%=======================
% 给定待编码数据（字符串，大数左端）返回汉明码编码数据
%=======================
function [out_data] = encode(data,mode)
data_ = fliplr(data); %翻转字符串，得到左侧大端数据格式
data_len = length(data);
[redundacy_bit, group, index] = gen_group(data_len); % 得到索引
out_data = dec2bin(0, redundacy_bit + data_len); % 初始化编码输出

% 填充原始数据段
out_data(index{1,1}) = data_(index{1,2});

for i = 1:redundacy_bit
    % 计算冗余位置段
    sum = int64(0);
    list = group{3,i};
    for j = 1:length(list)
        sum = sum + int64(strcmp(data_(list(j)),'1'));
    end
    if mode == 1 % 奇校验
        out_data(2^(i-1)) = num2str(~(mod(sum,2) == 1));
    elseif mode == 2 % 偶校验
        out_data(2^(i-1)) = num2str(mod(sum,2) == 1);
    else
        error('wrong mode config!')
    end
end
out_data = fliplr(out_data); %翻转字符串，得到左侧大端数据格式
end
```

5.3 decode.m

```
%%=======================
% 给定待解码数据（字符串，大数左端）返回汉明码解码数据，错误位置
%=======================
function [out_data,err_index] = decode(data_in,mode)
%求得冗余位的位宽
redundacy_bit = 0;
full_data_len = length(data_in);
while(2^(redundacy_bit)<(full_data_len + 1))
    redundacy_bit = redundacy_bit + 1;
end
%计算原始数据位宽
data_len = full_data_len - redundacy_bit;
out_data = dec2bin(0,data_len);
%解码判断错误与否与错误位置
data_ = fliplr(data_in);
[~, group, index] = gen_group(data_len);
err_str = dec2bin(0, redundacy_bit);
for i = 1:redundacy_bit
    sum = int64(0);
    list = group{1,i};
    for j = 1:length(list)
        sum = sum + int64(strcmp(data_(list(j)),'1'));
    end
    if mode == 1 % 奇校验
        err_str(i) = num2str(~(mod(sum,2) == 1));
    elseif mode == 2 % 偶校验
        err_str(i) = num2str(mod(sum,2) == 1);
    else
        error('wrong mode config!')
    end
end
err_index = bin2dec(fliplr(err_str));
data_in_ =  fliplr(data_in); % 翻转得到左边大数端
if err_index ~= 0
    data_in_(err_index) =
num2str(~strcmp(data_in_(err_index),'1'));
end
%填充纠错后的数据
out_data(index{1,2}) = data_in_(index{1,1});
out_data = fliplr(out_data); % 翻转得到左边大数端
```

```
end
```

5.4 test.m

```
%=======================
clc
clear
close all
%=======================
% 检验汉明码算法
%=======================
data_len = 10; %测试数据长度
[redundacy_bit, ~, ~] = gen_group(data_len);
simutime = 1000; %检验样本数目
corect_amount = zeros(1,2);
total_amount = simutime * (data_len + redundacy_bit + 1);

for i = 1:simutime %奇校验测试
    raw = dec2bin(randi(2^data_len) - 1,data_len); %随机生成原始数据
    [encoded_data] = encode(raw,1); %基于奇校验的编码
    for j = 1:(data_len + redundacy_bit + 1) %对无错数据与随机翻转数据进
行校验测试
        if j ~= 1 %数据出错
            encoded_data_err = encoded_data;
            flip_index = data_len + redundacy_bit - j + 2;
            encoded_data_err(flip_index) =
num2str(~strcmp(encoded_data(flip_index),'1'));
            [decoded_data,err_index] = decode(encoded_data_err,1);
            if (err_index == (j-1)) && (strcmp(decoded_data,raw))
                corect_amount(1) = corect_amount(1) + 1;
            end
        else % 数据无错
            [decoded_data,err_index] = decode(encoded_data,1);
            if (err_index == 0) && (strcmp(decoded_data,raw))
                corect_amount(1) = corect_amount(1) + 1;
            end
        end
    end
end

for i = 1:simutime %偶校验测试
    raw = dec2bin(randi(2^data_len) - 1,data_len); %随机生成原始数据
```

```
    [encoded_data] = encode(raw,2); %基于奇校验的编码
    for j = 1:(data_len + redundacy_bit + 1) %对无错数据与随机翻转数据进
行校验测试
        if j ~= 1 %数据出错
            encoded_data_err = encoded_data;
            flip_index = data_len + redundacy_bit - j + 2;
            encoded_data_err(flip_index) =
num2str(~strcmp(encoded_data(flip_index),'1'));
            [decoded_data,err_index] = decode(encoded_data_err,2);
            if (err_index == (j-1)) && (strcmp(decoded_data,raw))
                corect_amount(2) = corect_amount(2) + 1;
            end
        else % 数据无错
            [decoded_data,err_index] = decode(encoded_data,2);
            if (err_index == 0) && (strcmp(decoded_data,raw))
                corect_amount(2) = corect_amount(2) + 1;
            end
        end
    end
end
ratio = [100 * corect_amount(1)/total_amount,...
         100 * corect_amount(2)/total_amount,...
         100 * (sum(corect_amount,'all'))/(2 * total_amount)];
fprintf('奇校验汉明码: [%d/%d] = %.4f%%的测试向量通过了验证
\n',corect_amount(1),total_amount,ratio(1))
fprintf('偶校验汉明码: [%d/%d] = %.4f%%的测试向量通过了验证
\n',corect_amount(2),total_amount,ratio(2))
fprintf('总体验证情况: [%d/%d] = %.4f%%的测试向量通过了验证
\n',sum(corect_amount,'all'),2 * total_amount,ratio(3))
if (sum(corect_amount,'all') == (2 * total_amount))
    fprintf('算法验证通过\n')
else
    fprintf('算法验证不通过\n')
end
```