同步 FIFO 的 SystemVerilog 验证框架搭建

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

目录

同步 FIFO 的 SystemVerilog 验证框架搭建

1 HDL 设计更改

在进行 SystemVerilog 验证框架的搭建之前，对 HDL 设计进行修改和更正使其更符合预期行为。

1.1 0-Priority 支持

在之前的 HDL 实现中，读出通道的权重无法设置为 0 权重，因为 0 权重被保留用于表示没有请求的通道状态。该特性显然是一种功能缺陷，因此修改了读出通道的权重逻辑，拓宽了配置后的权重位宽，从 8 位拓展至 9 位：

```
CONFIG: //config priority and addr
begin
    if (busy == 1'b0) begin // if not busy, we can config priority
and take control
        // configured pri = input pri + 1
        priority <= {1'b0, priority_cfg} + 1'b1;
        ...
```

将配置的权重映射为输入权重加 1，这样使得最小的配置权重实际为 1，在该通道操作完毕后重置为 0 避免持续占用通道，因此 find max 模块也需要拓宽比较位宽来选择响应的通道。在维持原有功能的情况下实现了对通道 0-priority 的支持。

1.2 无竞争的读出

为了实现真正无竞争的读出，保证所有的请求都能够被相应，并且所有请求的输出都是对应于该请求的正确结果(即各个通道之间没有干扰和混淆)，在原有的机制上额外增加了 busy-block 信号机制，如**图 1**。

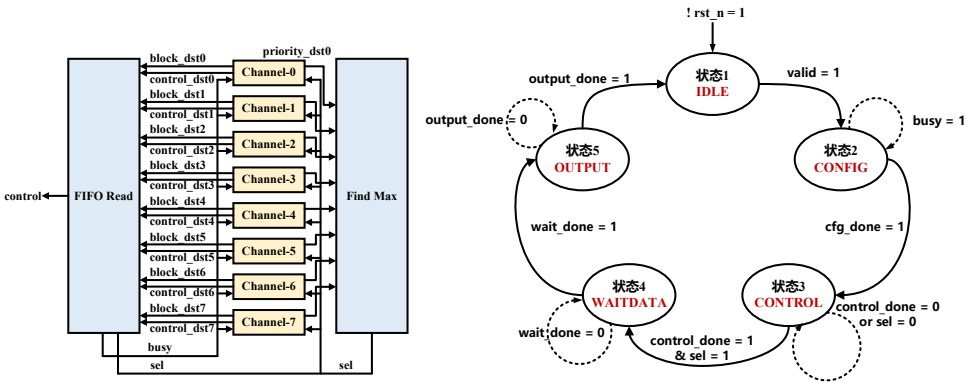

图 1. 仲裁器框图与读出通道状态机

每一个读出通道都会将配置好的权重送入 Find Max 模块，Find Max 模块会返回一个独热码信号，用于指示被选中的通道。同时每一个读出通道还会送出一个 block 信号，用于抢

占 arbiter，当任意一个通道的 block 信号为高时，arbiter 的状态为忙碌，测试 FIFO Read 模块的 busy 信号置高。同时，基于给出的 sel 信号，FIFO Read 模块选通对应通道的 control 信号，发送到 FIFO 模块中进行实际的读取时序控制。Block-Busy 信号的加入用于更好的避免通道间的竞争。为了更好阐述，结合如**图 1** 所示状态机，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。当 busy 信号为高时，不进行配置，当 busy 信号置低，说明 arbiter 不再忙碌，此时进行权重配置，并且将该通道的 block 信号置高抢占 arbiter 模块，进入输出控制 CONTROL 状态
- **CONTROL：**在该状态对 FIFO 读取信号进行控制，如果 sel 信号不为高，说明该通道抢占失败，控制权在别的通道，需要等待 sel 信号变高即控制权回到该通道时进行控制信号输出
- **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。同时由于控制信号已经发出，只需要等待数据传输即可，故此时 block 信号置低，不再对 arbiter 进行抢占尝试
- **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)

通过如上设计，保证了不论在何种情况下，arbiter 都能够正常运行：即所有的请求都会被响应，响应的数据一定属于响应请求发出的通道。

2 验证平台框架与实现

2.1 验证平台框架

本实验所实现的 DUT 框图与 SystemVerilog 验证框架如**图 2** 所示：

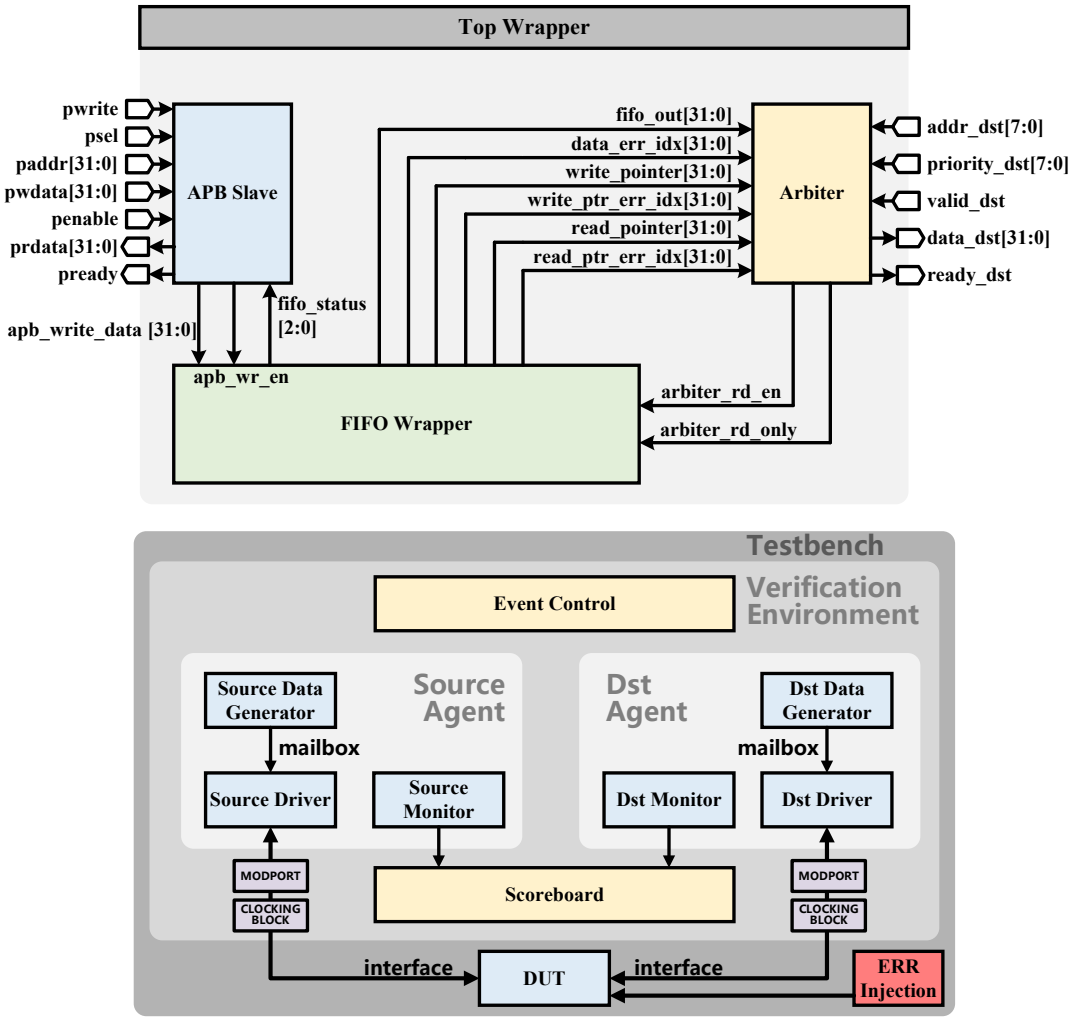

图 2. DUT 架构与 SystemVerilog 验证平台框架

DUT 与外交互的接口为 APB3 Slave 模块与 Arbiter 模块，APB3 模块对 FIFO 写入数据寄存器进行读写，对 FIFO 状态寄存器进行读取。Arbiter 模块进行 priority 的配置以及仲裁后的结果输出。显然，FIFO 数据的来源是 APB3 Slave，而数据最终去向是 Arbiter，因此在验证框架中分别用 source 和 dst 来构建 APB 和 Arbiter 的具体行为。

在验证框架中，最顶层为 Testbench，实际仿真的最顶层为名为 **testbench_top** 的 module，所有的仿真都是在这个 module 下完成的，在 testbench 中，同时包含验证环境即 verification environment 以及待测模块即使 Design Under Test (DUT)以及错误注入模块 ERR Injection。DUT 与验证环境的接口连接都是通过 interface 实现的，interface 分别定义了接口与 DUT 连接和与 Testbench 连接的 MODPORT，包括输入输出信号方向，同时还添加了 CLOCKING BLOCK 进行时序调整。DUT 的实现仅仅通过例化 **top_wrapper** 模块即可，而对于 source 和

dst 的 interface，显然 APB 只有一端的 IO 需求，而 Arbiter 需要至少 8 端的 IO 需求。因此 source 与 dst 分别需要 1 个和 8 个 interface。同时 ERR Injection 模块实现了随机数据，随机时刻的错误注入测试。

对于 Testbench 中的验证环境部分，其主要实现两个功能：仿真的功能控制(根据输入的指令，进行对应的验证激励)，检查 FIFO 错误情况以及数据完整性(Data integrity validation)。验证环境主要包含 source agent 与 dst agent 组件，数据完整性的检查主要通过对比 source agent 与 dst agent 的 monitor 的数据一致性完成的。而仿真功能控制主要是通过根据不同指令完成对 source agent 与 dst agent 的相关对象/函数的构建/调用完成的。

对于 Source agent 和 Dst agent，结构类似，其都包含一个 data generator 用来生成激励数据或相关信号的随机化生成，并通过 mailbox 内建 class 传递数据到 source/dst driver 中，同时 monitor 也会对 driver 进行监测，并将信息记录以便后续的 scoreboard 检查需要。source/dst driver 根据 mailbox 传递的信息，根据时序以及相关 flag 信号进行 DUT 中的 APB 或者 Arbiter 模块的控制。

2.2 验证平台对象/功能描述

在阐述实际的验证平台的功能实现方式之前，基于 **2.1** 小节给定的框架，首先给出验证平台的对象/功能的描述。**表 1** 给出了本验证平台的文件以及对应的功能描述：

表 1. 验证平台的文件以及对应的功能描述

文件名称(模块名称.sv)	功能描述
TESTBENCH.sv	验证平台顶层文件，DUT 和 TB 例化、连接以及错误注入
DUT.sv	待测模块例化，以及对应的 interface 连接
ENV.sv	验证环境定义，仿真事件的行为定义与数据完整性 scoreboard
SRC_AGENT.sv	APB agent，包含数据生成随机化与 APB 驱动定义
DST_AGENT.sv	Arbiter agent，包含数据生成随机化与 Arbiter 驱动定义
INTF.sv	DUT 与 TB 之间的 Interface 定义，包括 MODPORT 与 CLOKCING BLOCK

接下来按照自顶向下(Top Down)的顺序分别对每个文件涉及到的类、对象、函数和任务进行功能描述。对于 TESTBENCH 文件，**表 2** 给出了要素列表，在**表 3** 给出了功能列表：

表 2. TESTBENCH.sv 的要素列表

要素名称	要素类型	要素描述
testbench_top	module	定义 DUT 与 TB 以及连接的顶层模块
testbench	program	TB 的具体实现，例化 src/dst agent 与错误注入实现

表 3. TESTBENCH.sv 的功能列表

功能名称	功能类型	功能描述
N/A	forever loop	FIFO 写指针的随机错误注入
N/A	forever loop	FIFO 读指针的随机错误注入
N/A	forever loop	FIFO 读出数据的随机错误注入

对于 DUT 文件，**表 4** 给出了要素列表：

表 4. DUT.sv 的要素列表

要素名称	要素类型	要素描述
dut	module	定义 DUT，top_wrapper 的例化以及 interface 连接

对于 ENV 文件，**表 5** 给出了要素列表，在**表 6** 给出了功能列表：

表 5. ENV.sv 的要素列表

要素名称	要素类型	要素描述
env	package	封装有类 env_ctrl 以及 src/dst_agent_main package 的 package
env_ctrl	class	仿真事件控制的类，封装了相关函数与类成员

表 6. ENV.sv 的功能列表

功能名称	功能类型	功能描述
new	function(构造)	类 env_ctrl 的构造函数，对 class 里的成员，即例化的类 src_agent 与 dst_agent 执行对应的构造函数
integrity_valid	function	验证数据一致性的 scoreboard 函数，对成员类 src_agent 与 dst_agent 的 monitor 数据进行对比验证数据是否错误与完整
set_interface	function	将传入的 interface 分别分配给对应类 src_agent 与 dst_agent 的 active channel 中
run	task	根据传入的字符串 state 执行对应的仿真操作

对于 SRC_AGENT 文件，**表 7** 给出了要素列表，在**表 8** 给出了功能列表：

表 7. SRC_AGENT 的要素列表

要素名称	要素类型	要素描述
src_agent_objects	package	封装有 src_randomgen_datapkg 类，src_generator 类与 src_driver 类的 package
src_randomgen_datapkg	class	被使用于 APB 数据生成与驱动的数据类的定义
src_generator	class	APB 数据生成的具体实现类
src_driver	class	结合 interface，进行时序驱动的类定义
src_agent_main	package	封装有类 src_agent 以及 src_agent_objects package 的 package
src_agent	class	Source driver 的调用层

表 8. SRC_AGENT 的功能列表

功能名称	功能类型	功能描述
src_generator.new	function(构造)	类 src_generator 的构造函数，将传入的 mailbox 作为类成员
src_generator.data_gen	task	生成 APB 驱动所需要的数据以及序列随机化实现
src_driver.new	function(构造)	类 src_driver 的构造函数，将传入的 mailbox 作为类成员

src_driver.set_interface	function	将输入的 interface 作为类成员
src_driver.data_read	task	根据生成的数据进行 APB 数据读取时序定义
src_driver.data_write	task	根据生成的数据进行 APB 数据写入时序定义
src_agent.new	function(构造)	类 src_agent 的构造函数，初始化 monitor 数据并定义一个新的 mailbox 并传入对应的类成员的构造函数
src_agent.set_interface	function	将输入的 interface 传入对应的类成员的 set_interface 函数
src_agent.single_tran	task	data_read 与 data_write 函数的调用封装

对于 DST_AGENT 文件，给出了要素列表，在给出了功能列表：

表 9. DST_AGENT 的要素列表

要素名称	要素类型	要素描述
dst_agent_objects	package	封装有 dst_randomgen_datapkg 类，dst_generator 类与 dst_driver 类的 package
dst_randomgen_datapkg	class	被用于 Arbiter 数据生成与驱动的数据类的定义
dst_generator	class	Arbiter 数据生成的具体实现类
dst_driver	class	结合 interface，进行时序驱动的类定义
dst_agent_main	package	封装有类 dst_agent 以及 dst_agent_objects package 的 package
dst_agent	class	Dst driver 的调用层

表 10. DST_AGENT 的功能列表

功能名称	功能类型	功能描述
dst_generator.new	function(构造)	类 dst_generator 的构造函数，将传入的 mailbox 作为类成员
dst_generator.data_gen	task	生成 Arbiter 驱动所需要的数据以及序列随机化实现
dst_driver.new	function(构造)	类 dst_driver 的构造函数，将传入的 mailbox 作为类成员
src_driver.set_interface	function	将输入的 interface 作为类成员
src_driver.data_read	task	根据生成的数据进行 Arbiter 数据读取时序定义
dst_agent.new	function(构造)	类 dst_agent 的构造函数，初始化 monitor 数据并定义一个新的 mailbox 并传入对应的类成员的构造函数
dst_agent.set_interface	function	将输入的 interface 传入对应的类成员的 set_interface 函数
dst_agent.single_tran	task	data_read 与 data_write 函数的调用封装

对于 INTF 文件，**表 11** 给出了要素列表：

表 11. INTF 的要素列表

要素名称	要素类型	要素描述
duttb_intf_srcchannel	interface	APB 的 interface 定义
cb_src	clocking block	APB 的 interface clocking block 定义
DUTconnect	modport	连接 DUT 方向的 MODPORT 定义
TBconnect	modport	连接 TB 方向的 MODPORT 定义
duttb_intf_dstchannel	interface	Arbiter 的 interface 定义
cb_dst	clocking block	Arbiter 的 interface clocking block 定义
DUTconnect	modport	连接 DUT 方向的 MODPORT 定义
TBconnect	modport	连接 TB 方向的 MODPORT 定义

2.3 APB Driver 实现

APB3 协议的读写支持无等待的读写和有等待的读写，无等待方式实现较为复杂需要定义连续读写头尾和中部控制的，他们所需的时序不同，因此没有在 APB Driver 里实现支持连续读写的驱动，只实现了单次读写驱动。

在具体实现中，驱动的方法定义在类 **src_driver** 的 task 成员中，以数据写入为例，在 **set_interface** 之后，首先获取类 **src_driver** 的 mailbox 中的信息：

```
task data_read();
    // get data from mailbox to random_data_get
    src_randomgen_datapkg random_data_get;
    this.gen2drv.get(random_data_get);
```

然后根据 interface 里的 clk 按照协议定义的传输时序对信号进行相应控制：

```
// APB config
@(posedge this.active_channel.clk)
    this.active_channel.cb_src.channel_pwrite <= 1'b0;
    this.active_channel.cb_src.channel_psel <= 1'b1;
    this.active_channel.cb_src.channel_paddr <=
random_data_get.addr;
    this.active_channel.cb_src.channel_pwdata <=
random_data_get.data;
    this.active_channel.cb_src.channel_penable <= 1'b0;
// APB access
@(posedge this.active_channel.clk)
    // this.active_channel.channel_penable = 1'b1;
    this.active_channel.cb_src.channel_penable <= 1'b1;
// Wait for APB slave ready
// wait(this.active_channel.channel_pready)
wait(this.active_channel.cb_src.channel_pready == 1'b1)
    // to APB idle
    @(posedge this.active_channel.clk)
        this.active_channel.cb_src.channel_pwrite <= 1'b0;
```

```
        this.active_channel.cb_src.channel_psel <= 1'b0;
        this.active_channel.cb_src.channel_paddr <= 32'h0000_0000;
        this.active_channel.cb_src.channel_pwdata <= 32'h0000_0000;
        this.active_channel.cb_src.channel_penable <= 1'b0;
```

在上面的代码片段中，我们将 **random_data_get** 里的数据作为 interface 中信号线的驱动数据。同时，依赖 interface 里的 **pready** 信号作为触发到 IDLE 的事件，对这些信号线进行重置。

2.4 APB Data 数据随机化实现

APB Data 的生成是通过类 **src_generator** 的成员 **data_gen task** 实现的。APB 进行随机访存对应两种情况：随机写与随机读。随机生成是否启用取决于调用该 task 时输入的 flag 信号 **random** 是否为 1。对于随机写，由于写的目的寄存器只有一个，因此不考虑地址随机化，只进行写入数据的随机化，写入 32 位随机数通过调用系统函数**$urandom()**实现：

```
else begin // random write access
    random_val = $urandom();
    tran_data.addr = `FIFO_WRITE_DATA;     // get APB access addr
    tran_data.data = random_val;          // get APB access data
end
```

对于 APB 读取数据的随机化，通过一个生成的随机数与其生成范围的中间值(对于随机 32 位无符号数，其位 0x7FFFFFFF)作比较，随机访问写数据寄存器或者 FIFO 状态寄存器：

```
random_val = $urandom();
if (random_val > 32'h7FFF_FFFF) begin
    tran_data.addr = `FIFO_WRITE_DATA;  // get APB access addr
    tran_data.data = data;            // get APB access data
end
else begin
    tran_data.addr = `FIFO_STATUS;     // get APB access addr
    tran_data.data = data;            // get APB access data
end
```

2.5 Arbiter Driver 实现

Arbiter 的驱动的方法定义在类 **dst_driver** 的 task 成员中，在 **set_interface** 之后，首先获取类 **dst_driver** 的 mailbox 中的信息，然后按照与 APB Driver 类似的方式根据取得的数据进行驱动和时序生成。与 APB Driver 不同，由于 Arbiter 具有 8 个独立的通道，因此在具体调用时需要例化 8 个独立的 **dst_driver** instances：

```
dst_driver                    dst_driver[8];   // dst driver
instance
function new(); // constuctor of dst agent
    this.popped_count = 13'd0;
    this.mailbox_gen2drv = new(16);              // create a 16
size mailbox
```

```
    this.dst_generator   = new(mailbox_gen2drv);     // pass mailbox
to data gen
    this.dst_driver[0]     = new(mailbox_gen2drv);  // pass mailbox
to data drive[0]
    this.dst_driver[1]     = new(mailbox_gen2drv);  // pass mailbox
to data drive[1]
    this.dst_driver[2]     = new(mailbox_gen2drv);  // pass mailbox
to data drive[2]
    this.dst_driver[3]     = new(mailbox_gen2drv);  // pass mailbox
to data drive[3]
    this.dst_driver[4]     = new(mailbox_gen2drv);  // pass mailbox
to data drive[4]
    this.dst_driver[5]     = new(mailbox_gen2drv);  // pass mailbox
to data drive[5]
    this.dst_driver[6]     = new(mailbox_gen2drv);  // pass mailbox
to data drive[6]
    this.dst_driver[7]     = new(mailbox_gen2drv);  // pass mailbox
to data drive[7]
endfunction
```

以及对应 8 个独立的 interface：

```
function void set_interface(
    virtual duttb_intf_dstchannel.TBconnect dst_0,
    virtual duttb_intf_dstchannel.TBconnect dst_1,
    virtual duttb_intf_dstchannel.TBconnect dst_2,
    virtual duttb_intf_dstchannel.TBconnect dst_3,
    virtual duttb_intf_dstchannel.TBconnect dst_4,
    virtual duttb_intf_dstchannel.TBconnect dst_5,
    virtual duttb_intf_dstchannel.TBconnect dst_6,
    virtual duttb_intf_dstchannel.TBconnect dst_7
);
    // connect to dst
    this.dst_driver[0].set_interface(dst_0);
    this.dst_driver[1].set_interface(dst_1);
    this.dst_driver[2].set_interface(dst_2);
    this.dst_driver[3].set_interface(dst_3);
    this.dst_driver[4].set_interface(dst_4);
    this.dst_driver[5].set_interface(dst_5);
    this.dst_driver[6].set_interface(dst_6);
    this.dst_driver[7].set_interface(dst_7);
endfunction
```

2.6 Arbiter Data 数据随机化实现

Arbiter Data 的生成是通过类 **dst_generator** 的成员 **data_gen task** 实现的。Arbiter 进行随机访存只有一种情况，即随机读。因此随机生成访存地址。此外，由于 arbiter 的读出通道的权重也需要被配置，因此也一并随机化，随机生成是否启用取决于调用该 task 时输入的 flag 信号 **random_addr** 与 **random_priority** 是否为 1 来分别决定是否随机化生成地址和权重。对于读取地址随机化，通过调用系统函数**$urandom()**并对地址范围取模实现：

```
if (random_addr == 1'b0) begin  // not a random addr access
    tran_data.addr = addr;      // get Arbiter access addr
end
else begin // random addr access
    _random_addr = ($urandom() % 6);
    tran_data.addr = _random_addr;
end
```

对于权重的随机化，依然采用调用系统函数**$urandom()**并对权重范围取模实现：

```
if (random_priority == 1'b0) begin  // not a random priority
     tran_data.ch_priority = ch_priority;
 end
 else begin // random priority
     _random_priority = ($urandom() % 256);
     tran_data.ch_priority = _random_priority;
 end
```

2.7 Monitor 实现

Monitor 实现思路为在 src/dst agent 中插入一个数组 data 和一个计数器 counter。对于 source 的 monitor 实现，修改 source driver 里的 **data_write** 的 task 即可，data 记录所有写入 FIFO 的数据以及元素总数，注意这里通过 ref 实现了传递引用：

```
task data_write(
    ref bit [31:0] data [8192],
    ref bit [12:0] pushed_count
);
```

每有一个元素被写入则更新 data 和 counter，因此在 pready 信号拉高时才更新数据以确保数据完全写入：

```
wait(this.active_channel.cb_src.channel_pready == 1'b1)
    data[pushed_count] = random_data_get.data;
    pushed_count = pushed_count + 1'b1;
    $display("[SRC AGENT] @%0d ns Data pushed into FIFO: %h, total
pushed: %d ",
             $time, random_data_get.data, pushed_count);
```

对于 dst 的 monitor 实现，修改 dst driver 里的 **data_read** 的 task 即可，data 记录所有读出 FIFO 的数据以及元素总数，注意这里也通过 ref 实现了传递引用：

```
task data_read(
    input [2:0] channel,
```

```
    ref bit [31:0] data [8192],
    ref bit [12:0] popped_count
    );
```

每有一个元素被读出则更新 data 和 counter，因此在通道的 ready 信号拉高并且读取地址是 FIFO 数据时才更新数据以确保数据完全读出：

```
wait(this.active_channel.cb_dst.ready_dst == 1'b1)
    if (random_data_get.addr == `FIFO_OUT_DATA) begin
        data[popped_count] = active_channel.cb_dst.data_dst;
        popped_count = popped_count + 1'b1;
        $display("[DST AGENT] @%0d ns Data popped in ch-[%d] from
FIFO: %h, total popped: %d",
                 $time, channel, this.active_channel.cb_dst.data_dst,
popped_count);
    end
```

2.8 Scoreboard 实现

实现了 APB 与 Arbiter 的 Monitor 后，只需要在 ENV 中额外定义一个 task，并比较 **env_ctrl** 类中，子类 **src_agent** 与 **dst_agent** 的 **data** 是否一一对应完全一致即可：

```
function void integrity_valid();
    int push_count, pop_count;
    int correct_count;
    push_count = this.src_agent.pushed_count;
    pop_count = this.dst_agent.popped_count;
    $display("[ENV] FIFO data total pushed: %0d, total
popped: %0d", push_count, pop_count);
    if (pop_count > push_count) begin
        $display("[ENV] Exist attemp to pop an emoty FIFO!");
    end
    else begin
        correct_count = 0;
        for (int i = 0; i < pop_count; i++) begin
            if (this.dst_agent.data[i] == this.src_agent.data[i])
begin
                correct_count = correct_count + 1;
            end
        end
        $display("[ENV] %0d elements checked, %0d passed with %0d
failed, ratio = %4f %%",
                 pop_count, correct_count, (pop_count -
correct_count), 100 * (correct_count / pop_count));
        if (correct_count == pop_count) begin
```

```
            $display("[ENV] Data intergrity validation PASSED!");
        end
        else begin
            $display("[ENV] Data intergrity validation FAILED!");
        end
    end
endfunction
```

如果依次比较读出 FIFO 的数据与写入 FIFO 的数据完全一致，则说明校验通过。

2.9 错误注入实现

由于需要通过 force 与 release 语句进行强制赋值与释放实现错误注入，因此错误注入一定需要在能够直接可视 DUT 的地方定义，因此错误其被定义在 TESTBENCH.sv 中。对于错误注入分 src_agent 的错误注入和 dst_agent 的错误注入两种情况，错误注入与否通过对应 agent 的中常量成员是否为 1 决定，如 **err_wr_addr，err_rd_data，err_rd_addr** 分别为是否给写指针、读指针读出数据进行错误注入。其具体实现是在对应的错误的值更新时，首先通过一个变量暂存这个值，再通过随机系统函数随机翻转比特位然后通过 force 语句强制赋值，以写指针的随机错误注入为例：

```
initial begin
    if (envctrl.src_agent.err_wr_addr == 1'b1)
        $display("[TB-SYS] error injection of write pointer");
    forever @(posedge
dut.top_wrapper.fifo_wrapper_inst.fifo_ctrl_inst.wr_en) begin
        // trigger at the posedge of wr_en
        wr_addr_flip_index = $urandom() % 14;
        wr_addr_temp =
dut.top_wrapper.fifo_wrapper_inst.ptr_encode_instB.enc_data;
        wr_addr_temp[wr_addr_flip_index] =
~wr_addr_temp[wr_addr_flip_index];
        if ((envctrl.src_agent.err_wr_addr == 1'b1) && ($urandom() %
2 == 1'b1)) begin
            // inject error during source request (write pointer)
            force
dut.top_wrapper.fifo_wrapper_inst.ptr_encode_instB.enc_data =
wr_addr_temp;
            @(negedge
dut.top_wrapper.fifo_wrapper_inst.fifo_ctrl_inst.wr_en) begin
            // release at negedge
            release
dut.top_wrapper.fifo_wrapper_inst.ptr_encode_instB.enc_data;
            end
        end
```

```
    end
end
```

在这里，通过**$urandom()**取模 2 来随机决定是否错误注入，并通过**$urandom()**模下标范围来决定具体翻转哪一位进行错误注入。随后在合适的时机进行释放恢复原始驱动的值。

2.10 验证场景划分

在给出了最后所划分的验证场景与对应的验证行为：

表 12. 验证场景划分与验证行为描述

验证场景名称	描述
APB Single Write	APB 进行单次，规律的 FIFO 写入，直到写满 FIFO
APB IO Random Access	APB 进行随机的读写，读地址随机，写数据随机，直到写满 FIFO
Arbiter Read Single	(先填满 FIFO)Arbiter 进行单个通道的，规律的读出，直到 FIFO 为空
Arbiter Read Simultaneous	(先填满 FIFO)Arbiter 进行八个通道的同时请求读出，直到 FIFO 为空
Arbiter Read Race	(先填满FIFO)Arbiter进行八个通道的依次错位一个时钟周期，但是后一个通道的权重大于前一个请求的通道的权重进行抢占，直到 FIFO 为空
Arbiter Random Access	(先填满 FIFO)Arbiter 进行八个通道的，一共 128 次的同时的随机读出，读出地址随机，请求发生在同一时刻
Random APB/Arbiter Access	APB 和 Arbiter 的联合随机测试，先填一半的 FIFO 空间，，然后在随机时刻，同时进行随机的 APB 读/写，Arbiter 的八个通道在随机时刻，随机通道，随机地址的读取请求
Error Injection	错误注入测试，先填满 FIFO，再读出 FIFO，中间穿插随机错误注入

3 验证结果与分析

3.1 APB 固定写入测试

首先进行 APB 写入测试，依次写入 1~1024 写满 FIFO，并且尝试在 FIFO 满之后再次写入数据。**图 3** 展示了时序结果，其中亮蓝色是 APB Slave 模块的信号，白色是 FIFO 控制器以及存储器的信号。随着数据的不断写入 **fifo_status** 逐渐从 0(空)到 1(容量大于 3/4 但小于 4/4)到 2(容量大于 2/4 但小于 3/4)到 3(容量大于 1/4 但小于 2/4)到 4(容量大于 0/4 但小于 1/4)到 5(满)。可见 FIFO 工作如预期，APB 协议也正确。

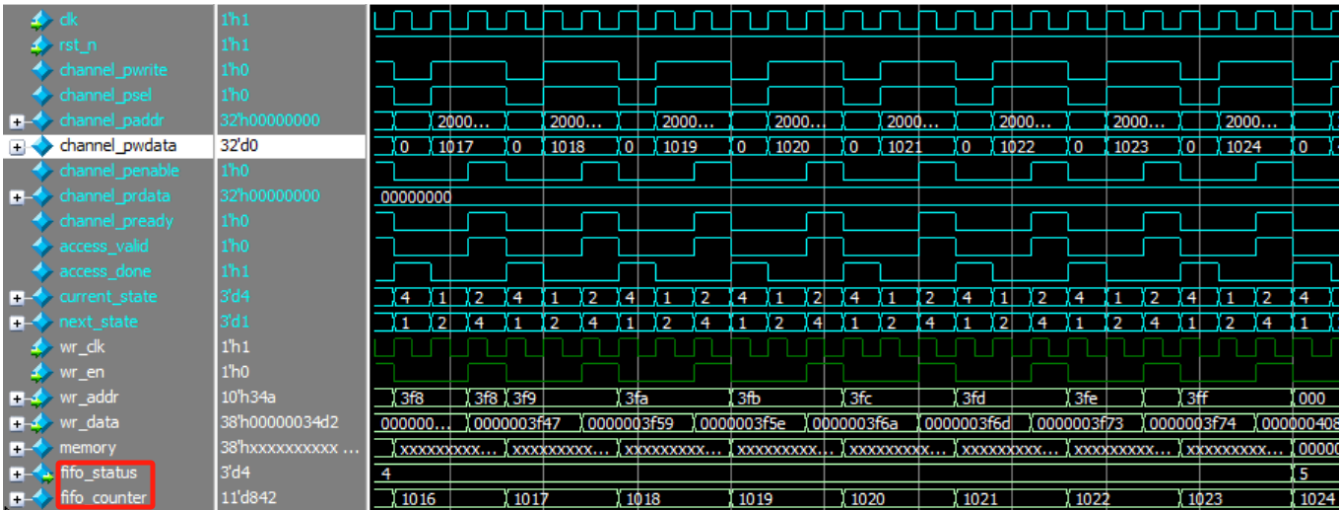

图 3. APB 固定数据写入直到 FIFO 满

随后进行在 FIFO 满的情况下的数据写入测试，如**图 4**。可见虽然 APB 进行了写入请求，但是 **pready** 信号没有进行响应，同时写指针 **wr_addr** 不被改变，**memory** 里存储内容也不变，证明 FIFO 工作如预期。

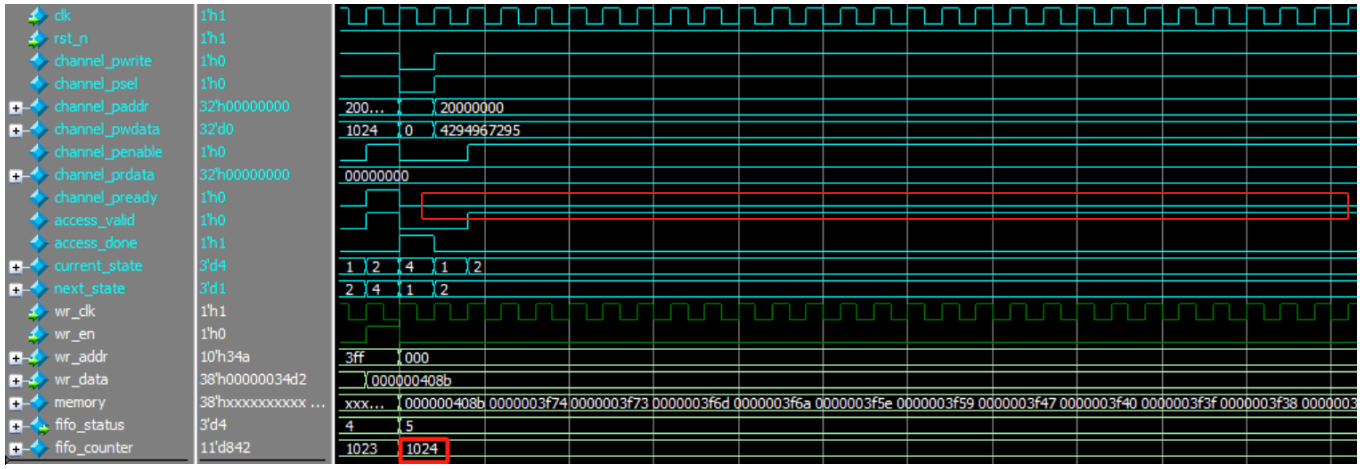

图 4. APB 在 FIFO 满时继续写入

同时 MONITOR 与 SCOREBOARD 在 CONSOLE 中输出了对应的 FIFO 数据写入结果：

```
# [SRC AGENT] @25 ns Data pushed into FIFO: 00000001, total pushed:      1
…
```

```
…
# [SRC AGENT] @6133 ns Data pushed into FIFO: 000003fb, total pushed: 1019
# [SRC AGENT] @6139 ns Data pushed into FIFO: 000003fc, total pushed: 1020
# [SRC AGENT] @6145 ns Data pushed into FIFO: 000003fd, total pushed: 1021
# [SRC AGENT] @6151 ns Data pushed into FIFO: 000003fe, total pushed: 1022
# [SRC AGENT] @6157 ns Data pushed into FIFO: 000003ff, total pushed: 1023
# [SRC AGENT] @6163 ns Data pushed into FIFO: 00000400, total pushed: 1024
# [ENV] time out !
# [ENV] FIFO data total pushed: 1024, total popped: 0
# [ENV] 0 elements checked, 0 passed with 0 failed, ratio = 0.000000 %
# [ENV] Data integrity validation PASSED!
```

3.2 APB 随机访存测试

进行 APB 进行随机访存测试，先写后读，读地址随机，写数据随机，直到写满 FIFO。**图 5** 展示了时序结果，其中亮蓝色是 APB Slave 模块的信号，白色是 FIFO 控制器以及存储器的信号。在第一个请求随机写入数据，在第二个请求读出 **fifo_status** 为 1，在第三个请求随机写入数据 1003859694，在第四个请求读出了写入寄存器的值 1003859694。可见功能与预期一致。

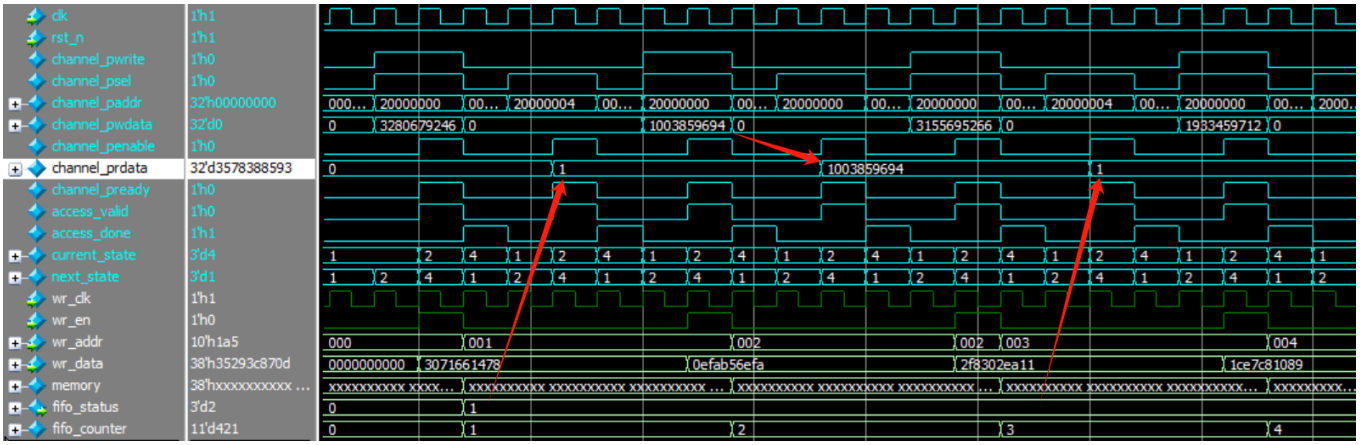

图 5. APB 进行随机访存测试

同时 MONITOR 与 SCOREBOARD 在 CONSOLE 中输出了对应的 FIFO 数据写入结果：

```
# [SRC AGENT] @25 ns Data pushed into FIFO: c38b314e, total pushed:      1
# [SRC AGENT] @37 ns Data pushed into FIFO: 3bd5aeee, total pushed:      2
…
…
# [SRC AGENT] @12241 ns Data pushed into FIFO: 7b2a83c3, total pushed: 1019
# [SRC AGENT] @12253 ns Data pushed into FIFO: 494f9cb2, total pushed: 1020
# [SRC AGENT] @12265 ns Data pushed into FIFO: ddb8d78b, total pushed: 1021
# [SRC AGENT] @12277 ns Data pushed into FIFO: f3369036, total pushed: 1022
# [SRC AGENT] @12289 ns Data pushed into FIFO: d3302d9b, total pushed: 1023
# [SRC AGENT] @12301 ns Data pushed into FIFO: 6c171b27, total pushed: 1024
```

```
# [ENV] time out !
# [ENV] FIFO data total pushed: 1024, total popped: 0
# [ENV] 0 elements checked, 0 passed with 0 failed, ratio = 0.000000 %
# [ENV] Data integrity validation PASSED!
```

可见随机数据被成功写入并且 FIFO 最终被写满。

3.3 单通道读出测试

进行 Arbiter 的单通道读出测试，首先固定写入 1~1024 到 FIFO 中，然后再通过 Arbiter 单个通道读出，同时权重配置为当前读取次数模 256，即权重为 0~255 进行循环。**图 6** 展示了时序结果，其中棕黄色是 Arbiter 第 0 通道读出模块的信号，白色是 FIFO 控制器的信号。可见最后成功依次读出了 1~1024 的数据，并且 **fifo_status** 也从 5 变为 0 全空，功能与预期一致。

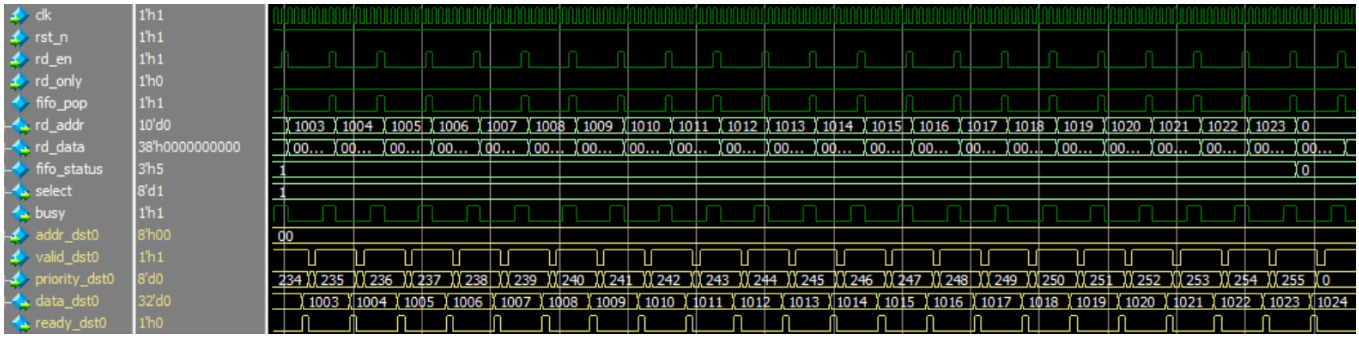

图 6. 单通道读出测试

随后进行在 FIFO 空的情况下的数据读出测试，如**图 7**。可见虽然 Arbiter 进行了写入请求，但是读指针 **rd_addr** 没有改变，证明 FIFO 工作如预期。

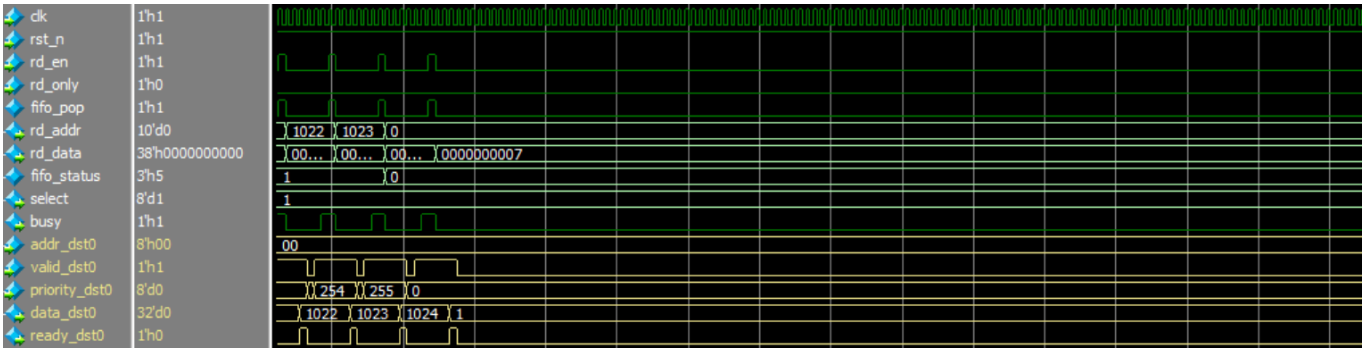

图 7. 单通道读出空 FIFO

同时 MONITOR 与 SCOREBOARD 在 CONSOLE 中输出了对应的 FIFO 数据读出结果：

```
…
# [DST AGENT] @20429 ns Data popped in ch-[0] from FIFO: 000003fb, total popped: 1019
# [DST AGENT] @20443 ns Data popped in ch-[0] from FIFO: 000003fc, total popped: 1020
# [DST AGENT] @20457 ns Data popped in ch-[0] from FIFO: 000003fd, total popped: 1021
# [DST AGENT] @20471 ns Data popped in ch-[0] from FIFO: 000003fe, total popped: 1022
# [DST AGENT] @20485 ns Data popped in ch-[0] from FIFO: 000003ff, total popped: 1023
# [DST AGENT] @20499 ns Data popped in ch-[0] from FIFO: 00000400, total popped: 1024
```

```
# [DST AGENT] @20513 ns Data popped in ch-[0] from FIFO: 00000001, total popped: 1025
# [ENV] finish work : Arbiter Read Single!
# [ENV] FIFO data total pushed: 1024, total popped: 1025
# [ENV] Exist attempt to pop an empty FIFO!
```

可见 SCOREBOARD 检测到了存在读出空 FIFO 的尝试。

3.4 八通道同时请求测试

进行 Arbiter 的八通道读出测试，首先固定写入 1~1024 到 FIFO 中，然后再通过 Arbiter 八个通道同时进行读出请求，同时所有通道权重配置为当前读取次数模，即权重为 0~127(FIFO 深度 1024，一次读出 8 个数据，128 次后读空)。**图 8** 展示了时序结果。

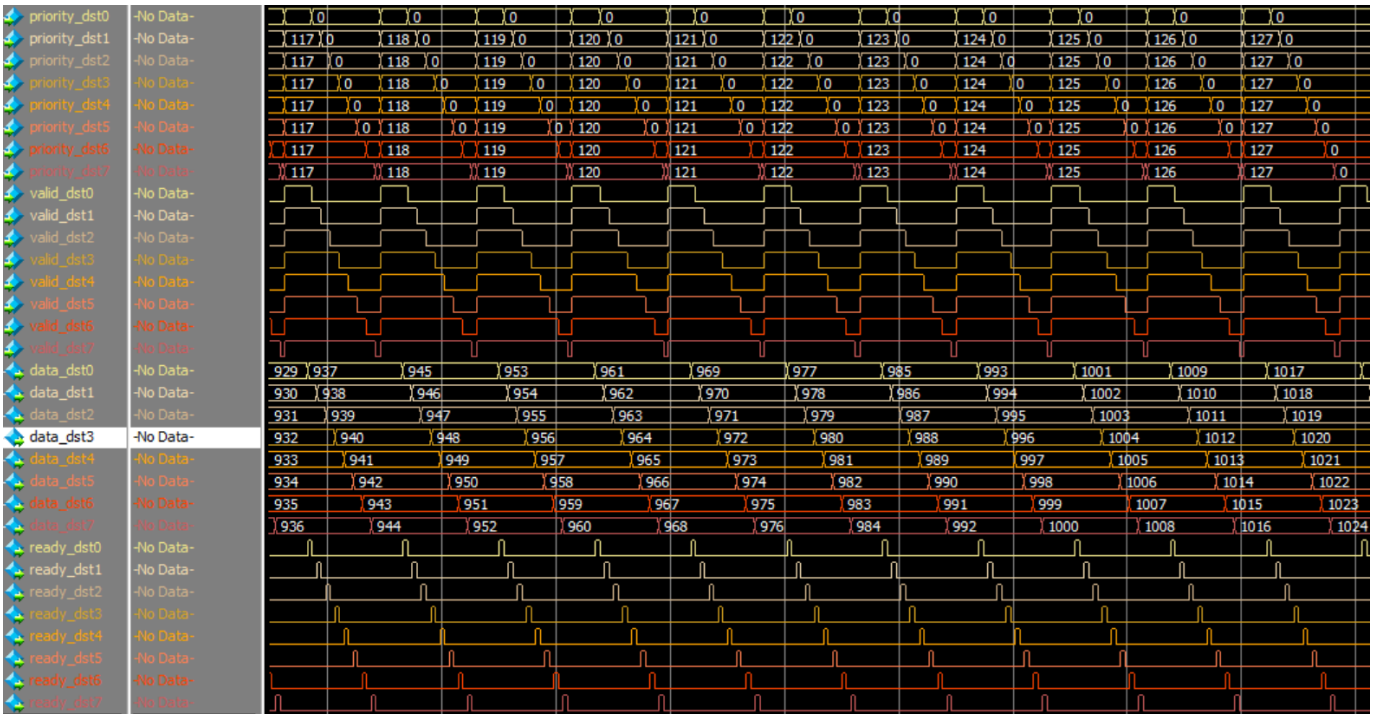

图 8. 八通道 Arbiter 同时读出请求

从时序图可见，请求同时发出，并且优先级相同，最后所有请求都依次被响应，按照通道序号越小，通道优先级越高的顺序逐个对 FIFO 数据进行读出，同时发送不混淆。并且 MONITOR 在 CONSOLE 中的输出也可以验证这一点：

```
…
# [DST AGENT] @11469 ns Data popped in ch-[0] from FIFO: 000003f1, total popped: 1009
# [DST AGENT] @11473 ns Data popped in ch-[1] from FIFO: 000003f2, total popped: 1010
# [DST AGENT] @11477 ns Data popped in ch-[2] from FIFO: 000003f3, total popped: 1011
# [DST AGENT] @11481 ns Data popped in ch-[3] from FIFO: 000003f4, total popped: 1012
# [DST AGENT] @11485 ns Data popped in ch-[4] from FIFO: 000003f5, total popped: 1013
# [DST AGENT] @11489 ns Data popped in ch-[5] from FIFO: 000003f6, total popped: 1014
# [DST AGENT] @11493 ns Data popped in ch-[6] from FIFO: 000003f7, total popped: 1015
# [DST AGENT] @11497 ns Data popped in ch-[7] from FIFO: 000003f8, total popped: 1016
# [DST AGENT] @11511 ns Data popped in ch-[0] from FIFO: 000003f9, total popped: 1017
```

```
# [DST AGENT] @11515 ns Data popped in ch-[1] from FIFO: 000003fa, total popped: 1018
# [DST AGENT] @11519 ns Data popped in ch-[2] from FIFO: 000003fb, total popped: 1019
# [DST AGENT] @11523 ns Data popped in ch-[3] from FIFO: 000003fc, total popped: 1020
# [DST AGENT] @11527 ns Data popped in ch-[4] from FIFO: 000003fd, total popped: 1021
# [DST AGENT] @11531 ns Data popped in ch-[5] from FIFO: 000003fe, total popped: 1022
# [DST AGENT] @11535 ns Data popped in ch-[6] from FIFO: 000003ff, total popped: 1023
# [DST AGENT] @11539 ns Data popped in ch-[7] from FIFO: 00000400, total popped: 1024
```

3.5 八通道竞争抢占测试

进行 Arbiter 的八通道读出测试，首先固定写入 1~1024 到 FIFO 中，然后再通过 Arbiter 八个通道进行读出请求，读出请求先从第 0 通道发起，经过一个时钟周期后再从第 1 通道发起……八个通道的请求相邻并错开一个周期，并且后一个通道的请求的权重大于前一个通道的请求的权重以进行抢占测试。**图 9** 展示了时序结果。

图 9. 八通道竞争抢占测试

从时序结果看出，在第 0 通道进行请求时，由于 Arbiter 空闲，因此直接进行响应，而在第 0 通道控制 FIFO 进行数据读时，恰好第 1，2，3 通道也依次进行请求，但都由于 Arbiter 被抢占被暂时忽略了，在第 0 通道释放 Arbiter 抢占时，同时对第 1，2，3 通道进行权重配置，又由于第 3 通道权重最高，因此先对第 3 通道进行响应而不是最先来到的第 1 通道。此时情况类似于 **3.4 小节**中的同时请求情况。最终直到第 3，2，1 通道依次处理完毕，这个过程中也积累了第 4，5，6，7 通道的请求，在第 1 通道释放抢占状态时，又同时对 4，5，6，7 通道进行权重配置，又由于第 7 通道权重最高因此优先对第 7 通道进行响应，最终依次完成第 7，6，5，4 通道的响应。但最终发送结果也无混淆，结果与预期一致，并且 MONITOR 在 CONSOLE 中的输出也可以验证响应顺序是[第 0 通道→第 3 通道→第 2 通道→第 1 通道→第 7 通道→第 6 通道→第 5 通道→第 4 通道]：

```
# [DST AGENT] @6177 ns Data popped in ch-[0] from FIFO: 00000001, total popped:     1
# [DST AGENT] @6183 ns Data popped in ch-[3] from FIFO: 00000002, total popped:     2
# [DST AGENT] @6187 ns Data popped in ch-[2] from FIFO: 00000003, total popped:     3
# [DST AGENT] @6191 ns Data popped in ch-[1] from FIFO: 00000004, total popped:     4
# [DST AGENT] @6197 ns Data popped in ch-[7] from FIFO: 00000005, total popped:     5
# [DST AGENT] @6201 ns Data popped in ch-[6] from FIFO: 00000006, total popped:     6
# [DST AGENT] @6205 ns Data popped in ch-[5] from FIFO: 00000007, total popped:     7
# [DST AGENT] @6209 ns Data popped in ch-[4] from FIFO: 00000008, total popped:     8
# [DST AGENT] @6223 ns Data popped in ch-[0] from FIFO: 00000009, total popped:     9
# [DST AGENT] @6229 ns Data popped in ch-[3] from FIFO: 0000000a, total popped:    10
# [DST AGENT] @6233 ns Data popped in ch-[2] from FIFO: 0000000b, total popped:    11
# [DST AGENT] @6237 ns Data popped in ch-[1] from FIFO: 0000000c, total popped:    12
# [DST AGENT] @6243 ns Data popped in ch-[7] from FIFO: 0000000d, total popped:    13
# [DST AGENT] @6247 ns Data popped in ch-[6] from FIFO: 0000000e, total popped:    14
# [DST AGENT] @6251 ns Data popped in ch-[5] from FIFO: 0000000f, total popped:    15
# [DST AGENT] @6255 ns Data popped in ch-[4] from FIFO: 00000010, total popped:    16
```

3.6 八通道随机权重随机地址访存测试

进行 Arbiter 的八通道读出测试，首先固定写入 1~1024 到 FIFO 中，然后再通过 Arbiter 八个通道同时进行读出请求，但是八个通道的读出请求权重随机，并且读出的地址随机。同时进行 128 次这样的 8 通道随机访存。**图 10** 展示了时序结果。

由于是随机地址，因此一共 1024 次访存，在数学期望上只有$1024 \times 1/6 = 170$的数据被读出，从最后的结果读出到 179 来看，这与预期相符，同时 MONITOR 在 CONSOLE 中的输出也可以进一步验证读出通道是随机的，SCOREBOARD 结果也可以看出 POP 结果与写入 FIFO 的结果是一一对应的：

```
…
# [DST AGENT] @11073 ns Data popped in ch-[1] from FIFO: 000000a5, total popped:   165
# [DST AGENT] @11107 ns Data popped in ch-[0] from FIFO: 000000a6, total popped:   166
# [DST AGENT] @11133 ns Data popped in ch-[1] from FIFO: 000000a7, total popped:   167
# [DST AGENT] @11183 ns Data popped in ch-[7] from FIFO: 000000a8, total popped:   168
# [DST AGENT] @11229 ns Data popped in ch-[5] from FIFO: 000000a9, total popped:   169
# [DST AGENT] @11241 ns Data popped in ch-[2] from FIFO: 000000aa, total popped:   170
# [DST AGENT] @11267 ns Data popped in ch-[6] from FIFO: 000000ab, total popped:   171
# [DST AGENT] @11363 ns Data popped in ch-[1] from FIFO: 000000ac, total popped:   172
# [DST AGENT] @11397 ns Data popped in ch-[7] from FIFO: 000000ad, total popped:   173
# [DST AGENT] @11439 ns Data popped in ch-[1] from FIFO: 000000ae, total popped:   174
# [DST AGENT] @11451 ns Data popped in ch-[4] from FIFO: 000000af, total popped:   175
# [DST AGENT] @11469 ns Data popped in ch-[3] from FIFO: 000000b0, total popped:   176
# [DST AGENT] @11527 ns Data popped in ch-[4] from FIFO: 000000b1, total popped:   177
# [DST AGENT] @11535 ns Data popped in ch-[6] from FIFO: 000000b2, total popped:   178
# [DST AGENT] @11539 ns Data popped in ch-[3] from FIFO: 000000b3, total popped:   179
```

```
# [ENV] finish work : Arbiter Random Access!
# [ENV] FIFO data total pushed: 1024, total popped: 179
# [ENV] 179 elements checked, 179 passed with 0 failed, ratio = 100.000000 %
# [ENV] Data integrity validation PASSED!
```

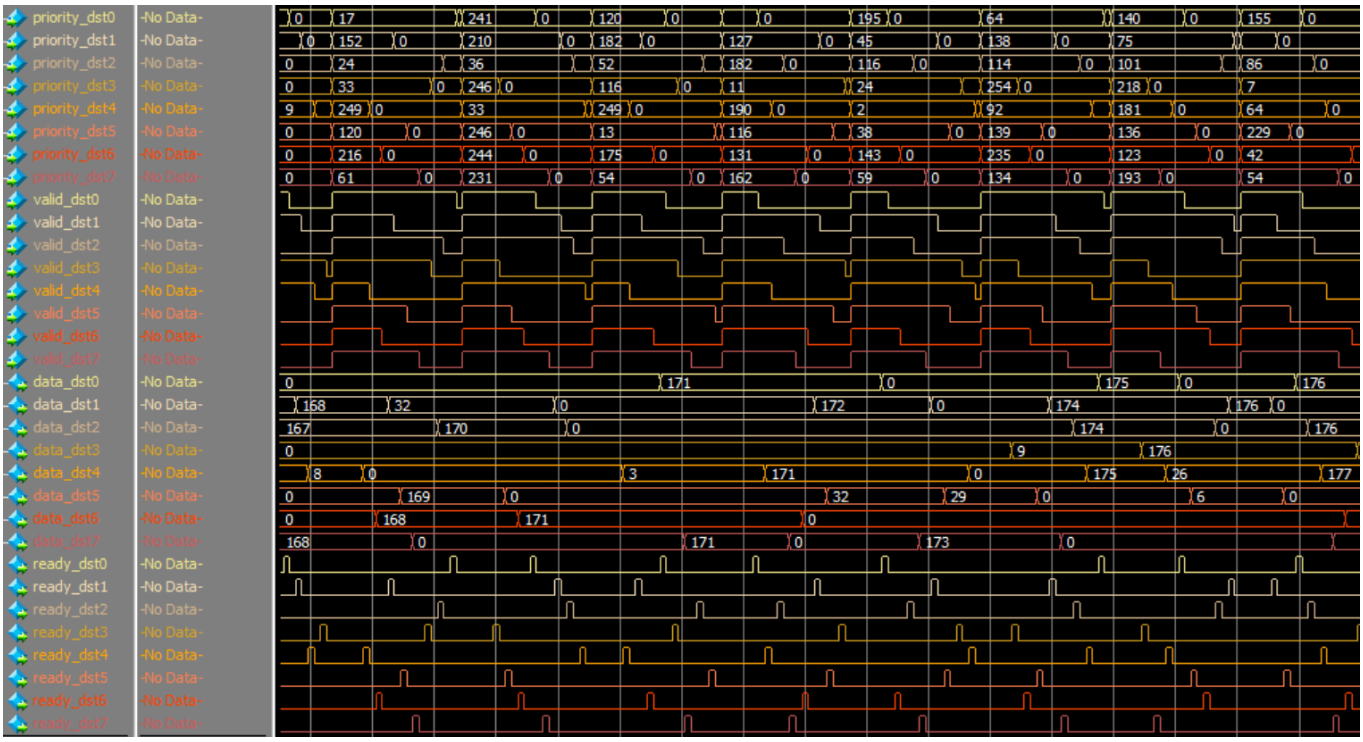

图 10. 八通道随机权重随机地址访存测试

3.7 APB 与 Arbiter 联合随机测试

进行 APB 与 Arbiter 的八通道读出测试，首先固定写入 1~512 共 512 个数据到 FIFO 中使得 FIFO 半满。对于 APB，在随即时刻，随机对 FIFO 写入随机数据，或者随机读出写入寄存器/FIFO 状态寄存器。同时，对于 Arbiter 的八个通道，分别也在随机时刻，随机对某一个通道驱动发出读出请求，并且权重也是随机的。随机的读/写一共进行 1024 次。**图 11** 展示了时序结果。

时序结果中，亮蓝色是 APB 的信号结果，暖色是 Arbiter 的信号结果。从时序结果看出，对于 APB，确实实现了随机时刻的随机读写请求(随机写数据，随机读寄存器)，这一点还可以通过 APB 的 MONITOR 进行验证(在这里，删除了中间夹杂的 Arbiter 的 MONITOR 信息)，每次写入的数据与时刻都不固定：

```
…
# [SRC AGENT] @3149 ns Data pushed into FIFO: 941d20e4, total pushed:   514
# [SRC AGENT] @3191 ns Data pushed into FIFO: 6fd4f441, total pushed:   515
# [SRC AGENT] @3245 ns Data pushed into FIFO: ef4106fa, total pushed:   516
# [SRC AGENT] @3293 ns Data pushed into FIFO: b7ba8fba, total pushed:   517
# [SRC AGENT] @3351 ns Data pushed into FIFO: b2644d14, total pushed:   518
```

```
# [SRC AGENT] @3393 ns Data pushed into FIFO: 6e2cf3b5, total pushed:    519
```

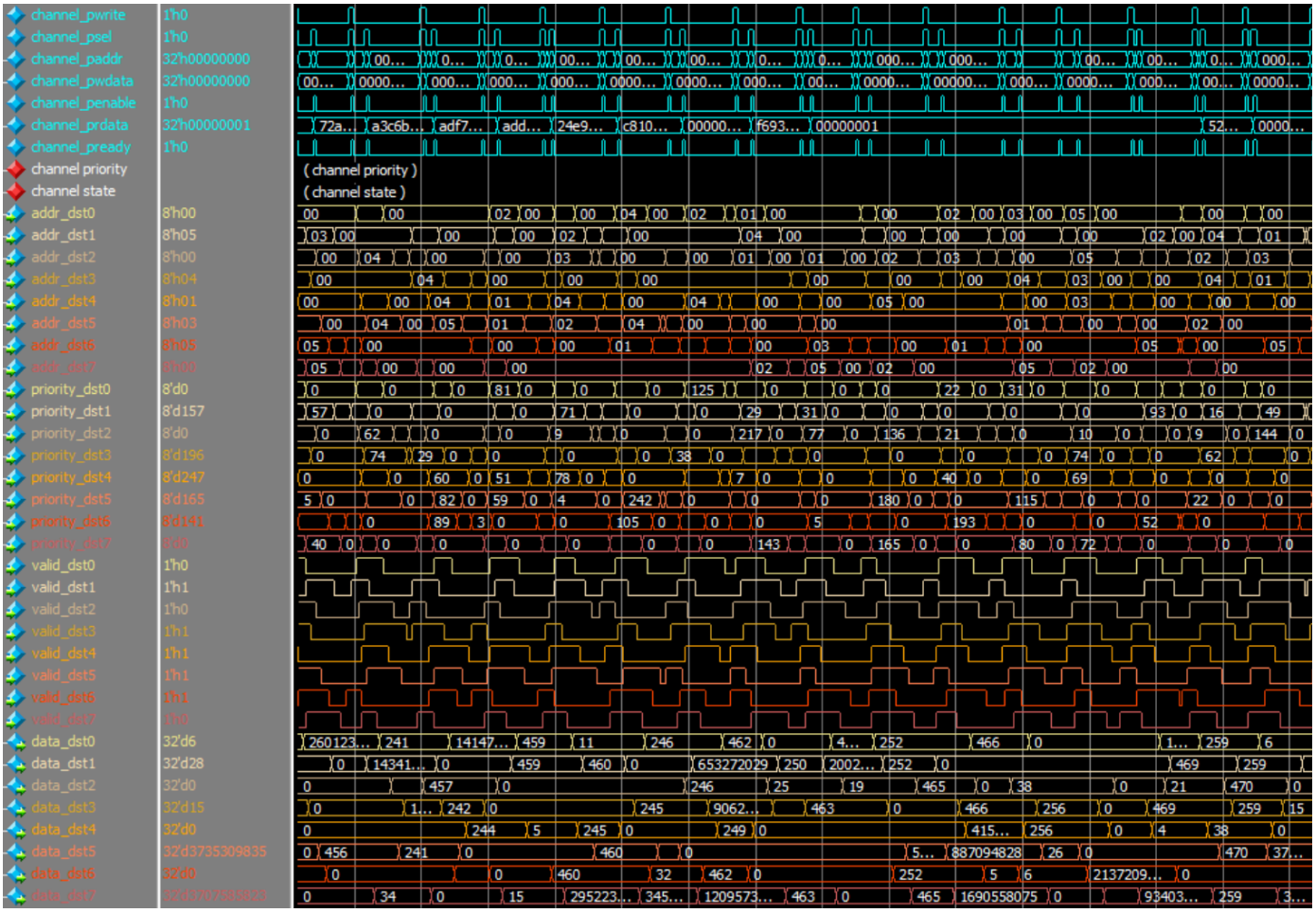

图 11. APB 与 Arbiter 的联合随机测试

对于 Arbiter，确实实现了随机时刻的随机读请求(随机权重，随机读寄存器)，这一点还可以通过 Arbiter 的 MONITOR 进行验证(在这里，删除了中间夹杂的 APB 的 MONITOR 信息)，每次读出的数据的时刻与通道都不固定：

```
# [DST AGENT] @3105 ns Data popped in ch-[3] from FIFO: 00000001, total popped:    1
# [DST AGENT] @3123 ns Data popped in ch-[4] from FIFO: 00000002, total popped:    2
# [DST AGENT] @3155 ns Data popped in ch-[6] from FIFO: 00000003, total popped:    3
# [DST AGENT] @3223 ns Data popped in ch-[1] from FIFO: 00000004, total popped:    4
# [DST AGENT] @3305 ns Data popped in ch-[5] from FIFO: 00000005, total popped:    5
# [DST AGENT] @3323 ns Data popped in ch-[1] from FIFO: 00000006, total popped:    6
# [DST AGENT] @3355 ns Data popped in ch-[4] from FIFO: 00000007, total popped:    7
# [DST AGENT] @3369 ns Data popped in ch-[6] from FIFO: 00000008, total popped:    8
```

对于最后的数据一致性，观察最后的 SCOREBOARD 结果，可以发现随机测试依然通过，数据完整性保持一致：

```
# [ENV] FIFO data total pushed: 1496, total popped: 1285
# [ENV] 1285 elements checked, 1285 passed with 0 failed, ratio = 100.000000 %
# [ENV] Data integrity validation PASSED!
```

```
# [TB-SYS] testbench system has done all the work, exit !
```

3.8 随机错误注入测试

最后进行随机错误注入测试，首先固定写入 1~1024 的数据写满 FIFO，再单通道读出 FIFO 数据，写入和读出的时刻随机进行错误注入。最终时序结果如**图 12** 与**图 13** 所示。可见到写入指针/读出指针/读出数据随机发生单比特翻转，并且在随机时刻进行错误注入，纠错模块同时给出错误比特索引以及纠错结果。

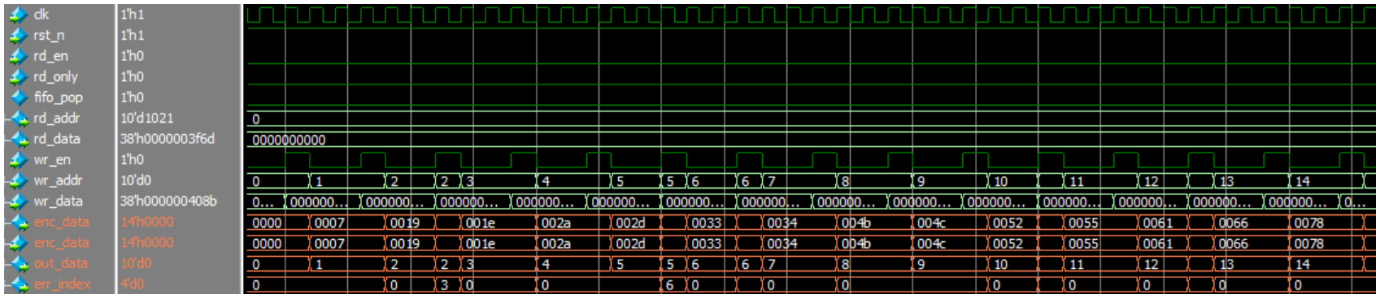

图 12. 写入指针的随机错误注入测试

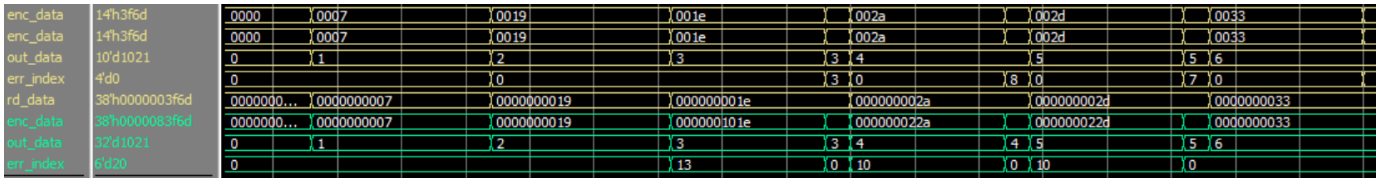

图 13. 读取指针与读取数据的随机错误注入测试

同时，错误注入的结果正确性通过 SCOREBOARD 的输出进行验证，所有的数据均通过了校验：

```
# [ENV] finish work : Error Injection!
# [ENV] FIFO data total pushed: 1024, total popped: 1024
# [ENV] 1024 elements checked, 1024 passed with 0 failed, ratio = 100.000000 %
# [ENV] Data integrity validation PASSED!
# [TB-SYS] testbench system has done all the work, exit !
```

4 总结

本次实验成功完成了同步 FIFO 的 SystemVerilog 的验证框架搭建，进行了初步的验证。完成的验证功能如**表 13** 总结：

表 13. 验证框架功能实现情况

功能描述	完成情况
测试的目标端 DRV 请求必须要包括： （1）每个通道的单一时刻独占测试 （2）多个通道的单一时刻竞争测试，最好能够包括所有通道同时请求的极端情况 （3）包含 FIFO 的空满状态的测试覆盖	完成
（1）单次测试进行 priority 配置，配置后进行对应数据传输测试 （2）priority 配置尽可能覆盖多通道多情况的 priority 竞争情况，如所有通道 priority 值都是 0	完成
完成源端的 DRV 读写请求测试，该测试需要配合完成 FIFO 的空满状态的测试覆盖	完成
进行错误注入，验证模型纠错功能正确性	完成
INTF 中加入 clocking block 来对时序进行调整	完成
为测试引入随机化测试过程： （1）测试数据的随机化，包括源端 data，目标端 priority，addr （2）测试请求的随机化，即在随机的时刻，驱动任意数量的目标端 DRV 来生成请求，同时源端的 DRV 也会在随机时刻生成请求 （3）注错数据的随机化，即在随机的时刻，对随机的数据或指针进行注错	完成
完成一个完整的 testbench 结构，这包括如下部分的完成： （1）搭建 MONITOR 对象，对源端发送和目标端收集的数据进行采集 （2）搭建 GOLDEN_MODEL 对象，根据注错情况结合输入数据对读出的各项结果的正确性进行判断 （3）搭建 SCOREBOARD 对象(GOLDEN_MODEL 可以内嵌这这个对象里)，对上述判断结果进行错误率的统计，统计结果在测试结束时打印	完成