同步 FIFO 的 SystemVerilog 断言

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

目录

1 HDL 设计与验证模块更改 2
1.1 命名混淆修改 2
1.2 APB3 时序修改 2
1.3 APB 背靠背传输 2
1.4 ARBITER 时序修改 2
2 验证平台框架与断言实现 3
2.1 验证平台框架与断言 3
2.2 APB 断言实现 3
2.3 ARBITER 断言实现 5
2.4 FIFO 断言实现 6
3 验证结果与分析 8
3.1 激励选择 8
3.2 APB3 协议断言情况 9
3.3 ARBITER 断言情况 9
3.4 FIFO 断言情况 10
4 总结 11

同步 FIFO 的 SystemVerilog 断言

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

1 HDL 设计与验证模块更改

在进行项目的 SystemVerilog 断言之前，对 HDL 设计以及验证模块进行修改和更正以便符合设计规格与验证需要。

1.1 命名混淆修改

在之前的 HDL 与验证实现中，复位信号为 **reset_n** 与 **rst_n** 混用，在 HDL 设计的模块中，统一使用的是 **rst_n** 标识，在 **top_warpper** 与验证模块的 DUT 与 INTF 中使用的是 **reset_n** 标识。在进行断言时发现，混用的复位信号使得顶层的复位信号激励无法传递到部分层级的模块中，使得验证出现异常。因此统一将所有的 **reset_n** 标识更改为 **rst_n**。

1.2 APB3 时序修改

在 APB3 协议中，不论读写，对于 **addr**、**pwrite** 与 **pwdata** 信号应该在下一个请求前保持不变，减少电平翻转降低功耗。因此将 SRC_AGENT 的读写 task 最后的 **access** 后到 **idle** 状态阶段，不对 **addr**、**pwrite** 与 **pwdata** 信号进行修改。

1.3 APB 背靠背传输

在之前的验证设计中，没有实现 APB3 协议的背靠背的请求时序，为了完整覆盖 APB3 的功能，加入了背靠背的读写操作。对于背靠背的连续传输与通讯的操作，可分为三个阶段

1. 背靠背传输中第一次传输时，在 **access** 阶段，**sel** 信号不会拉低而是保持高
2. 背靠背传输的第二次以后与最后一次传输之前，**sel** 信号依旧保持高，并且交替进入 **config** 与 **access** 阶段，从而跳过 **idle** 状态进行快速传输
3. 背靠背传输的最后一次传输，在 **access** 阶段后进入 **idle** 状态，**sel** 信号拉低

因此具体实现背靠背传输的 task 时，读取传输分为 **data_read_head**、**data_read_body** 与 **data_read_tail** 三个不同的 task 分别对应以上三个阶段。对于写入传输，也类似分为 **data_write_head**、**data_write_body** 与 **data_write_tail**，三个独立的 task 实现。

1.4 Arbiter 时序修改

依据与 APB3 类似的低功耗要求，减少电平翻转，对于 Arbiter 读出通道的地址与权重配置信息都遵循在下一次传输之前保持不变。

2 验证平台框架与断言实现

2.1 验证平台框架与断言

本实验所实现 SystemVerilog 验证框架如**图 1** 所示：

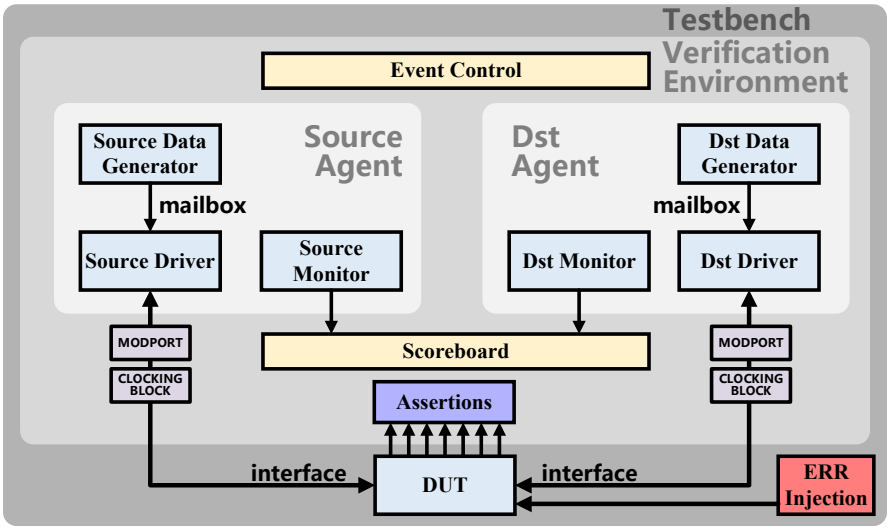

图 1. SystemVerilog 验证平台框架

与之前的验证框架相比，增加了 Assertions，通过额外定义一个 DUT_ASSERTIONS 的 module 通过 bind 进行设计模块/实例与验证模块的绑定，实现设计与验证分立，通过降低设计验证的耦合度方便进行设计与验证的独立调整。

本项目实现的 Assertions 分为三类:

1. **APB3 断言：**针对 APB3 协议的各种信号，如地址、读/写数据、请求/响应信号进行断言，进行功能正确性验证以及低功耗标准的验证
2. **Arbiter 断言：**针对 Arbiter 的 Valid-Ready 协议进行各种信号，如地址、读出数据、权重、请求/响应信号的断言，进行功能正确性验证以及低功耗标准的验证
3. **FIFO 断言：**针对 FIFO 的功能，对各种信号，如读写使能、读写地址进行功能性正确的验证，并对 FIFO 状态信号进行正确性验证

2.2 APB 断言实现

对于 APB 断言，在阐述实现方式之前，首先给出针对 APB3 协议的 19 个断言覆盖的项描述，如**表 1** 所示：

表 1. APB3 协议断言覆盖项描述

序号	断言描述
1	在 **psel** 信号置高后 **paddr** 信号不为 X/Z
2	在 **psel** 信号置高后 **paddr** 信号不超出可访存范围
3	在传输的 **SETUP** 阶段 **paddr** 信号应该保持不变
4	为支持低功耗，**paddr** 信号在下一次传输前保持不变
5	在复位后，**penable** 信号不为 X/Z

6	**psel** 信号上升后 **penable** 信号需要下一周期置高
7	直到 **pready** 信号拉起，**penable** 信号拉起后则保持不变
8	在 **pready** 拉高后，**penable** 应在下一周期拉低
9	在复位后 **psel** 信号不为 X/Z
10	在 **pready** 拉高之前 **psel** 应该保持稳定
11	**psel** 拉高后 **pwdata** 不应该为 X/Z
12	在 **ACCESS** 状态之前 **pwdata** 应该保持稳定
13	为支持低功耗，**pwdata** 信号在下一次传输前保持不变
14	**pwrite** 信号在复位后不为 X/Z
15	**pwrite** 在 **pready** 拉高之前保持稳定
16	在 **pready** 拉高后 **prdata** 不应该为 X/Z
17	为支持低功耗，**prdata** 信号在下一次传输前保持不变
18	**pready** 必须与 **penable** 一同拉低
19	**pready** 必须在拉高后一个周期下降

对于以上断言，可分为以下 4 类典型情况：

1. 复位后或者某信号拉起后不为 X/Z
2. 信号值不超出某给定范围
3. 信号在某信号拉高之前保持稳定
4. 信号在下一个传输前保持稳定

对于第一类情况，以 APB3 断言项“1”在 **psel** 信号置高后 **paddr** 信号不为 X/Z 为例，断言描述如下：

```
property _paddr_no_x_psel_high;
       // after psel set high, paddr shouldn't be X
       @(posedge clk) $rose(psel) |-> not ($isunknown(paddr));
endproperty
```

通过系统函数**$isunknown** 来辅助判断是否为 X/Z 等未知值，而对于复位后某信号不为 X/Z，则以 APB3 断言项“5”为例：

```
property _penable_no_x;
    // penable shouldn't be x after reset
    @(posedge clk) disable iff(~rst_n)
       not $isunknown(penable);
endproperty
```

通过 **disable iff** 描述，在复位时禁用该 property，复位结束后，则在每个上升沿判断相应信号是否是未知值。

对于第二类的情况，以 APB3 断言项“1”在 **psel** 信号置高后 **paddr** 信号不超出可访存范围为例，断言描述如下：

```
property _paddr_valid;
    // after psel set high, paddr shouldn't be out of access range
    @(posedge clk) $rose(psel) |-> ((paddr == `FIFO_WRITE_DATA) ||
(paddr == `FIFO_STATUS));
endproperty
```

其中，这里的 **FIFO_WRITE_DATA** 与 **FIFO_STATUS** 为自定义的 APB3 维护的寄存器地址的宏定义，在 **psel** 拉高后，检查 **paddr** 信号是否处于该范围。

对于第三类情况，以 APB3 断言项“7”直到 **pready** 信号拉起，**penable** 信号拉起后则保持不变为例，断言描述如下：

```
property _penable_stable;
    // when pready dosen't rise, penable should be stable after it
rises
    @(posedge clk) disable iff(~rst_n)
        $rose(penable) |-> penable[*1:$] ##0 $rose(pready);
endproperty
```

以 **penable** 拉起作为触发，并且紧跟带不确定周期的 consecutive repetition，即 **penable[*1:$]**，并且最终直到 **pready** 拉高。此外，除了使用 consecutive repetition 的实现，在第四类情况中，以 APB3 断言项“4”为支持低功耗，**paddr** 信号在下一次传输前保持不变为例，还可以通过比较前后目标信号值是否相等实现：

```
property _paddr_stable_until_next_trans;
    // paddr should be stable until next transmission
    logic[31:0] addr1, addr2;
    // avoid multiple matched sequences (multiple addr1 and addr2
that may not in a nearby request)
    @(posedge clk) first_match(($rose(penable),addr1=paddr) ##[1:$]
((psel && !penable),addr2=$past(paddr))) |-> addr1 == addr2;
endproperty
```

通过 **fisrt_match** 避免匹配不相邻请求的 **addr** 判断，并用临时变量 **addr1** 与 **addr2** 结合赋值与**$past** 赋值进行比较。与通过 consecutive repetition 进行稳定性判断相比，该方法开销更小，但是无法避免传输中间发送了数据变动的可能，如果要进行严格判断应该使用 consecutive repetition 进行断言。

2.3 Arbiter 断言实现

对于 Arbiter 的断言，首先给出针对 Arbiter 的 Valid-Ready 握手以及整体功能性验证的 15 个断言覆盖的项描述，如**表 2** 所示：

表 2. Arbiter 断言覆盖项描述

序号	断言描述
1	在 **valid** 信号置高后 **addr** 信号不为 X/Z
2	在 **valid** 信号置高后 **addr** 信号不超出可访存范围
3	在传输阶段 **addr** 信号应该保持不变
4	为支持低功耗，**addr** 信号在下一次传输前保持不变
5	在 **valid** 信号置高后 **priority** 信号不为 X/Z
6	在传输阶段 **priority** 信号应该保持不变
7	为支持低功耗，**priority** 信号在下一次传输前保持不变
8	在复位后 **valid** 信号不为 X/Z

9	在 **ready** 拉高之前 **valid** 应该保持稳定
10	**valid** 信号应该在 **ready** 信号拉高一周期后拉低
11	在 **ready** 拉高后，**data** 不为 X/Z
12	为支持低功耗，**data** 信号在下一次传输前保持不变
13	在复位后 **ready** 信号不为 X/Z
14	**ready** 信号必须与 **valid** 一起拉低
15	在 **ready** 信号拉高之后必须在一个周期后拉低

对于以上断言，与 APB3 协议的断言类似，依然可分为以下 4 类典型情况：

1. 复位后或者某信号拉起后不为 X/Z
2. 信号值不超出某给定范围
3. 信号在某信号拉高之前保持稳定
4. 信号在下一个传输前保持稳定

以上断言的实现与 **2.2** 小节几乎完全一致，这里不再赘述。

2.4 FIFO 断言实现

对于 FIFO 的断言，首先给出针对 FIFO Controller 的 I/O 与整体功能性验证的 17 个断言覆盖的项描述，如**表 3** 所示：

表 3. FIFO Controller 断言覆盖项描述

序号	断言描述
1	在复位后 **wr_en** 信号不为 X/Z
2	在复位后 **rd_en** 与 **rd_only** 信号都不为 X/Z
3	在 **wr_en** 信号置高后 **wr_addr** 信号不为 X/Z
4	在 FIFO 有数据送入且 FIFO 不满，**wr_addr** 应该自增 1
5	在 FIFO 满的时候，即使 **wr_en** 为高 **wr_addr** 也不变
6	在 **wr_en** 为低的时候，**wr_addr** 不变
7	在 **rd_en** 信号置高后 **rd_addr** 信号不为 X/Z
8	在 FIFO 有数据读出且 FIFO 不空，**rd_addr** 应该自增 1
9	在 FIFO 空的时候，即使 **rd_en** 为高 **rd_addr** 也不变
10	在 FIFO 只读(**rd_only** 高)的时候，即使 **rd_en** 为高 **rd_addr** 也不变
11	在 **rd_en** 为低的时候，**rd_addr** 不变
12	在复位后 **fifo_status** 信号不为 X/Z
13	在复位后 **fifo_status** 信号任何时候都要在指示范围内
14	当 FIFO 有数据送入且无数据读出，**fifo_status** 应从 0 变为 1
15	当 FIFO 有数据读出且无数据送入，**fifo_status** 应从 5 变为 4
16	在 **wr_en** 或 **rd_en** 低时，**fifo_status** 应保持不变
17	当 FIFO 同时有数据送入与读出时，**fifo_status** 应保持不变

其中由于要实现只读的 FIFO 功能，相比于读使能信号 **rd_en**，还添加了一个只读信号 **rd_only**，在 **rd_en** 与 **rd_only** 均为高的时候，只会读出挡墙 **rd_addr** 指向的 FIFO 内容而不会增加 **rd_addr**。相应的，如果 **rd_en** 为高，**rd_only** 为低，则正常读出并且 **rd_addr** 自增，而如果 **rd_en** 为低，则无数据读出。

对于以上断言，可分为以下 3 类典型情况：

1. 复位后或者某信号拉起后不为 X/Z
2. 满足某条件时，信号自增 1/自减 1
3. 满足某条件时，信号不变

对于第一类情况，与 APB3 断言以及 Arbiter 断言类似，不再赘述。

对于第二种情况，以 FIFO 断言项“4”在 FIFO 有数据送入且 FIFO 不满，**wr_addr** 应该自增 1 为例，断言描述如下：

```
property _wr_addr_inc_when_push;
    // wr_addr should increase when FIFO push data
    logic[ADDR - 1:0] addr;
    @(posedge clk) (wr_en && fifo_status != 3'd5, addr=wr_addr) |=>
(wr_addr == (addr + 1'b1));
endproperty
```

定义一个局部变量 addr 存储在有数据送入时的写地址，并将其与自增 1 后的写地址进行相等比较完成断言。以 FIFO 断言项“15”当 FIFO 有数据读出且无数据送入，**fifo_status** 应从 5 变为 4 为例，断言描述如下：

```
property _fifo_status_dec;
    // fifo_status should decrease from 5 to 4 when FIFO only pop
data
    @(posedge clk) ($rose(rd_en) && fifo_status == 3'd5 && !rd_only
&& !wr_en) |=> (fifo_status == 3'd4);
endproperty
```

在满足 **fifo_status** 为 5(FIFO 满)，并且无写入(**wr_en** 为 0)，有读出(**rd_en** 拉高并且只读信号 **rd_only** 为低)时，下一周期后，**fifo_status** 变为 4。

对于第三类情况，以 FIFO 断言项“17”当 FIFO 同时有数据送入与读出时，**fifo_status** 应保持不变为例，断言描述如下：

```
property _fifo_status_stable_io;
    // fifo_status should be stable when no FIFO both push and pop
    @(posedge clk) ($rose(rd_en) && !rd_only) |-> ($rose(wr_en))
|=> $stable(fifo_status);
endproperty
```

当 **rd_en** 拉高并且只读信号 **rd_only** 为低，说明有读取发生，立即检查 **wr_en** 是否拉高，如果依然满足说明此时 FIFO 同时有数据送入与读出，则 **fifo_status** 应该保持稳定，利用系统函数**$stable** 以判定。

3 验证结果与分析

3.1 激励选择

为了更好的进行验证的覆盖，在这里修改了激励情况，结合在 **1.3** 小节所描述的新实现的背靠背 APB 传输，对 DUT 激励为首先进行 256 次以单次 APB 传输为传输方式的 FIFO 数据随机写入，再进行 256 次以背靠背 APB 传输为传输方式的 FIFO 数据随机写入。具体实现如下所示：

```
for (int i = 0;i < 256;i++) begin // push 256 data into the fifo
with single tran
    src_agent.single_tran(1, 0, `FIFO_WRITE_DATA, i + 1);
end
for (int i = 256;i < 512;i++) begin // push 256 data into the fifo
with b2b tran
    if (i == 256)
        src_agent.b2b_tran_head(1, 0, `FIFO_WRITE_DATA, i + 1);
    else if(i != 511)
        src_agent.b2b_tran_body(1, 0, `FIFO_WRITE_DATA, i + 1);
    else
        src_agent.b2b_tran_tail(1, 0, `FIFO_WRITE_DATA, i + 1);
end
```

最后进行 1024 次联合测试(写入与读出并行进行)，具体实现为同时进行 APB 的 FIFO 在随机时刻，随机地址，随机写入内容的访存与 8 个读出通道的独立的，随机时刻，随机地址，随机权重的读出访存，具体实现如下所示：

```
for (int i = 0;i < 1024;i++) begin
// fifo random access and arbiter random access
fork
  begin
  // random apb write
  #(2 * ($urandom() % 5)) src_agent.single_tran(1, 1,
`FIFO_WRITE_DATA, 32'h0000_0000);
  // random read
  #(2 * ($urandom() % 5)) src_agent.single_tran(0, 1,
`FIFO_WRITE_DATA, 32'h0000_0000);
  end
  // random arbiter read
  begin #(2 * ($urandom() % 10)) dst_agent.single_tran(0, 1, 1, 0,
0); end
  begin #(2 * ($urandom() % 10)) dst_agent.single_tran(1, 1, 1, 0,
0); end
```

```
  begin #(2 * ($urandom() % 10)) dst_agent.single_tran(2, 1, 1, 0,
0); end
  begin #(2 * ($urandom() % 10)) dst_agent.single_tran(3, 1, 1, 0,
0); end
  begin #(2 * ($urandom() % 10)) dst_agent.single_tran(4, 1, 1, 0,
0); end
  begin #(2 * ($urandom() % 10)) dst_agent.single_tran(5, 1, 1, 0,
0); end
  begin #(2 * ($urandom() % 10)) dst_agent.single_tran(6, 1, 1, 0,
0); end
  begin #(2 * ($urandom() % 10)) dst_agent.single_tran(7, 1, 1, 0,
0); end
join
end
```

3.2 APB3 协议断言情况

图 2 展示了 APB3 协议的断言情况，可以看见全部的 19 个断言全部通过，没有断言失败的情况。

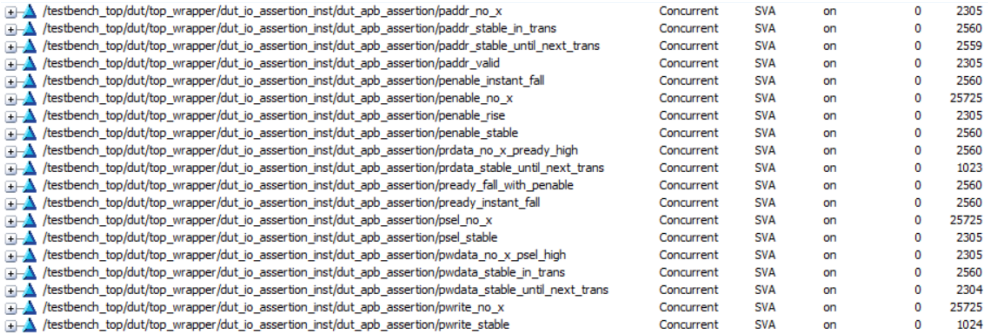

图 2. APB3 协议的全部断言情况

如**图 3** 所示，以 **paddr** 在下一个周期前稳定的断言为例，展示了其波形与断言情况，选择了单次 APB 传输与背靠背传输的分界点(红色箭头)，可看见，不论是单次传输还是背靠背传输，断言都正确触发捕获并且断言正确。

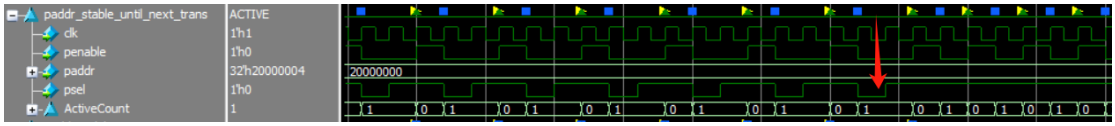

图 3. 在单次传输与背靠背 APB 传输交界点的断言与波形

3.3 Arbiter 断言情况

图 4 展示了第 0 通道的断言情况，可以看见 15 个断言全部通过，并且其余 1~7 共 8 个通道也全部通过了所有的断言，没有断言失败的情况。

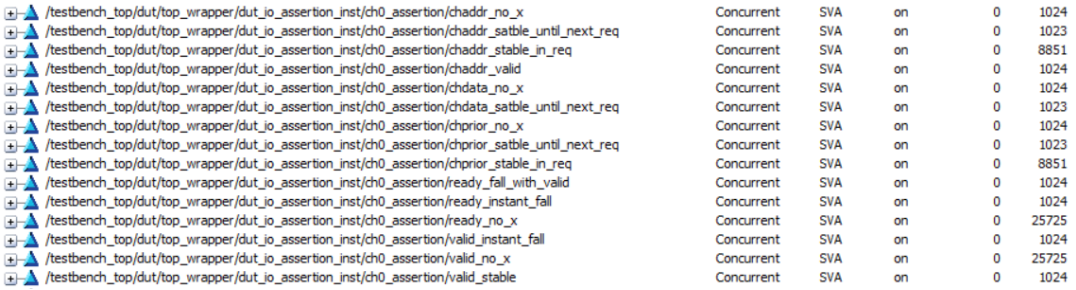

图 4. Arbiter 第 0 通道的断言情况

3.4 FIFO 断言情况

图 5 展示了 FIFO 的断言情况，可以看见 17 个断言全部通过，没有断言失败的情况。而对于没有捕获到断言的项目，其为 FIFO 断言覆盖项目的第 5，9，15 项目：分别为“在 FIFO 满的时候，即使 **wr_en** 为高 **wr_addr** 也不变”、“在 FIFO 空的时候，即使 **rd_en** 为高 **rd_addr** 也不变”、“当 FIFO 有数据读出且无数据送入，**fifo_status** 应从 5 变为 4”。

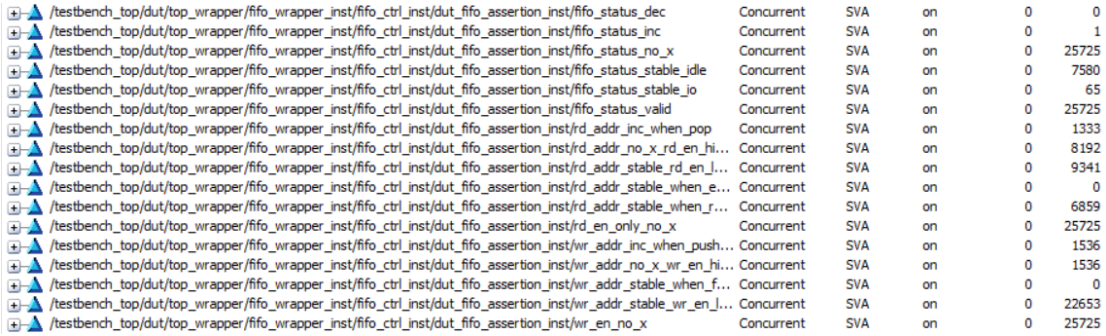

图 5. FIFO 断言情况

如**图 6** 所示，通过额外添加测试，最终第 9，15 未捕获的断言也能够正确断言，无断言失败：

图 6. 添加额外测试的 FIFO 断言情况

而第 5 个项目还是未捕获的原因为：由于 APB Slave 模块的设计，在 FIFO 满的时候，不会给出 **pready** 信号以及不会给 FIFO 的 **wr_en** 信号拉高，因此这样的断言无法被触发，符合设计预期。

4 总结

本次实验在进行 HDL 设计与验证模块修改的基础上，成功完成了同步 FIFO 的 SystemVerilog 的断言，进行了较为详细的验证。针对 APB3 协议的各种信号，如地址、读/写数据、请求/响应信号进行断言，进行功能正确性验证以及低功耗标准的验证；针对 Arbiter 的 Valid-Ready 协议进行各种信号，如地址、读出数据、权重、请求/响应信号的断言，进行功能正确性验证以及低功耗标准的验证；针对 FIFO 的功能，对各种信号，如读写使能、读写地址进行功能性正确的验证，并对 FIFO 状态信号进行正确性验证；由于汉明编码部分实现的为纯组合逻辑，因此没有进行断言。