上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

# System Verilog 实验报告


学院 电子信息与电气工程学院

班级 电院 24M091

学号 124039910018

姓名 汪子尧


2025 年 1 月 28 日

# 目录

**1 实验内容** **4**

**2 DUT 设计** **6**

2.1 ICB 从机模块 ( icb_slave ) . . . 7
2.2 加密模块 ( encrypt ) . . . 9
2.3 fifo 模块 ( fifo ) . . . 10
2.4 APB 主机模块 ( apb_master ) . . . 11
2.5 DUT：使能 DES . . . 16

**3 验证平台设计** **17**

3.1 Icb Agent . . . 17
3.1.1 数据生成器（icb_generator） . . . 17
3.1.2 驱动器（icb_driver） . . . 17
3.1.3 监视器（icb_monitor） . . . 18
3.1.4 代理顶层（icb_agent） . . . 20
3.2 Apb Agent . . . 20
3.2.1 驱动器（apb_driver） . . . 20
3.2.2 代理顶层（apb_agent） . . . 21
3.3 Scoreboard . . . 22
3.3.1 Host-To-Device 行为级验证 . . . 22
3.3.2 Device-To-Host 数据级验证 . . . 23
3.4 仿真环境 env 与验证顶层 . . . 24
3.5 使能 DES：完整的验证平台 . . . 26
3.5.1 ICB 主机端：原始激励加密 . . . 26
3.5.2 验证端：加密激励解密 . . . 28
3.6 验证结果及分析 . . . 29
3.6.1 ICB 端总线时序验证 . . . 29
3.6.2 APB 端总线时序验证 . . . 29
3.6.3 数据流 LOOPBACK 验证 . . . 31
3.6.4 基于随机化测试的 golden model 验证 . . . 32

**4 SVA 断言设计** **34**

4.1 ICB 端断言检查 . . . 34
4.1.1 X 态检查 . . . 34
4.1.2 稳定性检查 . . . 34
4.1.3 握手检查 . . . 35
4.2 APB 端断言检查 . . . 35

4.3 FIFO 断言检查 . . . . . 36
4.3.1 空满信号检查 . . . . . 36
4.3.2 写入、读出功能检查 . . . . . 37
4.3.3 读写指针变化检查 . . . . . 37
4.4 SVA：启用 SVA 与 bindfile 设计 . . . . . 38

**5 重构验证平台：基于 UVM 的验证平台设计 39**
5.1 UVM 树状结构 . . . . . 40
5.2 factory 机制 . . . . . 41
5.3 uvm_phase 与 objection 机制 . . . . . 41
5.4 组件间通信 . . . . . 43
5.5 sequence 机制 . . . . . 44

**6 Coverage 覆盖率设计 45**
6.1 功能覆盖率 . . . . . 45
6.2 代码覆盖率 . . . . . 46

**7 DUT 综合及分析 49**
7.1 资源分析 . . . . . 49
7.2 时序分析 . . . . . 50

**8 总结 51**

# 1 实验内容

在 Lab1 中我们完成了具备加解密功能的一主四从总线桥的 DUT 设计，在 Lab2 中我们完成仿真平台的搭建以及使用仿真平台对 DUT 进行测试验证，在 Lab3 中我们对 DUT 模块添加断言模块，对一些经典数据交互进行断言。在 Final Project 中，我们对 DUT 与验证平台进行了系统性的优化与完善，基于制定的验证计划，在系统增加了 coverage 覆盖率检测设计。**此外，我们基于 UVM 对验证平台进行了重构，实现一套额外的 UVM 验证平台**。最终工程的框架图如 图 1 所示：

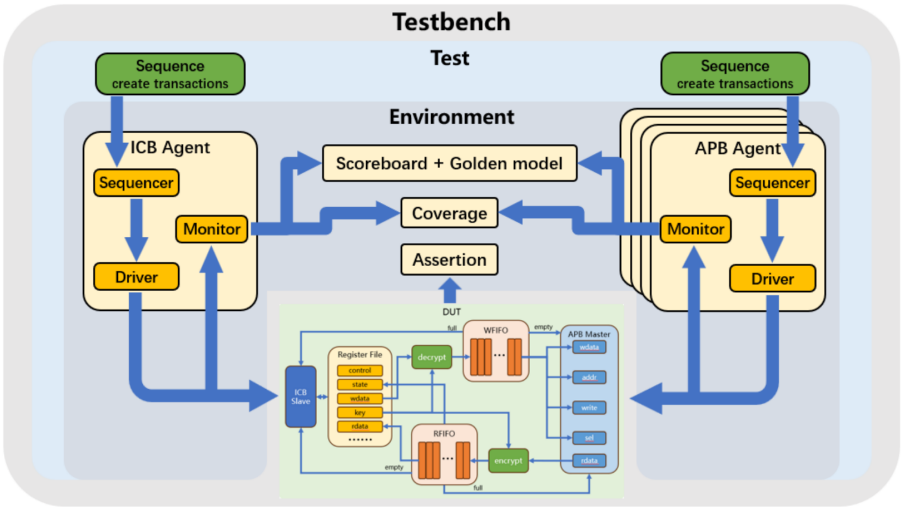


图 1: 仿真平台基本框架

UVM 验证框架如图 2所示：

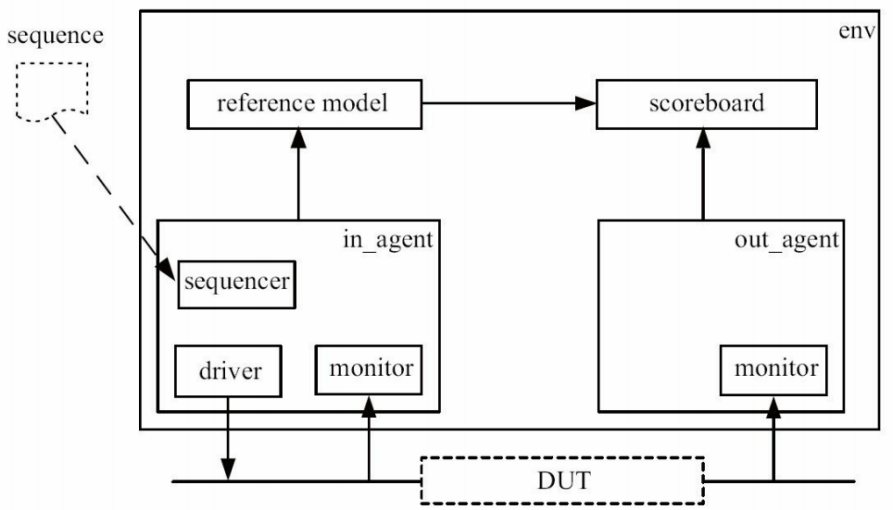


图 2: UVM 验证框架

本次 Project 完成情况如 表 1 所示：

表 1: Projects 完成度情况

| 类型 | 项目 | 完成度 |
|---|---|---|
| 验证计划制定 | 制定验证计划，包括功能覆盖及代码覆盖目标 | 完成 |
| ICB 驱动设计 | 验证平台通过 interface 连接 DUT，并拥有 ICB Agent 来驱动 DUT 进行工作：<br>（1）完成 ICB 读操作<br>（2）完成 ICB 写操作 | 完成 |
| APB 驱动设计 | 验证平台通过 interface 链接 DUT，并拥有 APB Agent 来响应 DUT 的指令进行工作：<br>（1）完成 APB 读操作<br>（2）完成 APB 写操作 | 完成 |
| 功能验证设计 | （1）完成 ICB Agent 和 APB Agent 的 monitor 模块，采集数据，包括 ICB 总线和 APB 总线上的读写数据等。<br>（2）设计 Scoreboard 和 golden model 模块，结合 monitor 采集的数据，完成功能的验证 | 完成 |
| 断言设计 | 根据验证计划，设计适当的 assertion 来检查 ICB 总线与 APB 总线时序的正确性：<br>（1）信号的 X 态检查<br>（2）信号传输时的稳定性检查<br>（3）ICB 信号的时序关系检查，如未握手的保持，握手的出现，CMD 通道发送指令则 RSP 通道必会返回等<br>（4）APB 信号的时序关系检查，如 PSEL，PENABLE 的拉高，PENABLE 握手后拉低等 | 完成 |
| 覆盖率设计 | （1）对 DUT 中所有有效寄存器进行功能覆盖率统计：无论寄存器能否读写，均对其进行读写检查<br>（2）对 DUT 中 wfifo 和 rfifo 的空满状态进行功能覆盖率统计<br>（3）对 APB 总线、ICB 总线的读写操作进行功能覆盖率统计<br>（4）对 DES 加解密功能正确性的覆盖<br>（5）给出 DUT 的代码覆盖率报告并分析 | 完成 |
| 随机化测试 | 采用随机化完成测试，要求如下：<br>（1）配置随机化<br>（2）操作发起时间随机化<br>（3）读写操作随机化<br>（4）数据随机化：读写数据、读写地址等 | 完成 |
| DUT 设计 | 实现 DES 加解密算法 | 完成 |
| DUT 综合 | DUT 在 vivado 内完成综合（资源、时序等） | 完成 |

# 2 DUT 设计

总线桥（bus bridge）是计算机系统中用于连接两种不同总线的硬件组件。总线桥的主要作用是实现不同总线之间的数据传输和协议转换，以确保多个设备之间的兼容性和通信。在 SoC 系统中，处理器常会与多个设备连接，因而总线桥常为一主多从或多主多从的实现形式。在某些涉及信息安全的场景下，主设备传输的数据可能是具有特定格式的加密数据，当从设备不具备解密功能时，则需额外的硬件电路实现加解密。本次实验验证的 DUT 部分设计如 图 3 所示：

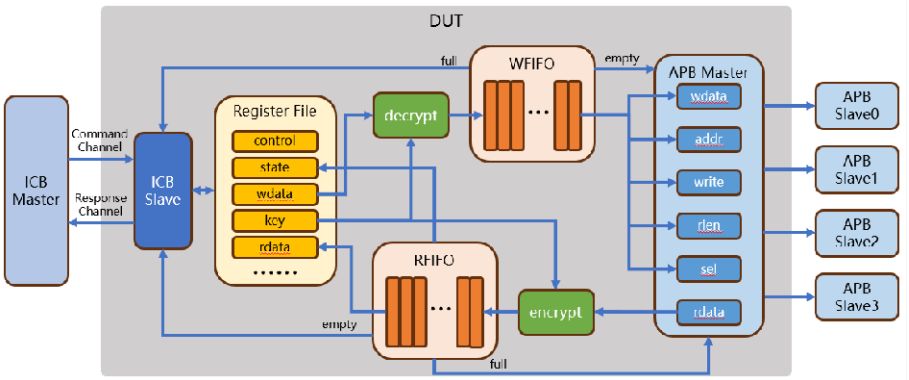


图 3: DUT 框架图

各模块功能设计如下所示：

(1) ICB 从机模块：

* 实现满足标准 ICB 时序的从机端口，数据位宽为 64bit，地址位宽为 32bit。
* 维护特定的寄存器，包括 CONTROL、STATE、WDATA、RDATA 和 KEY。

(2) 加密 (encrypt) 解密 (decrypt) 模块：

* 使用配置的密钥，在数据写入 FIFO 前完成加密和解密。
* 使用数据与密钥异或的方式进行加密、解密。

(3) FIFO 模块：

* 设计 FIFO 的基本功能，控制数据的写入和读出。
* 探索扩展 FIFO 的实现方式，如读写指针使用格雷码变换、读写时钟异步等。

(4) APB 主机模块：

* 实现满足标准 APB3 时序的主机端口，数据位宽为 32bit，地址位宽为 32bit。
* 对从 WFIFO 中获取的数据包进行解码，驱动 APB Master 执行相应操作。
* APB 从机返回的读数据，在 [63:32] 位补零后，送到加密模块加密，再送入 RFIFO。

## 2.1 ICB 从机模块（ icb_slave ）

ICB 总线主要包含 2 个通道：命令通道（cmd）与响应通道（rsp）。

表 2: ICB 总线信号

| 通道 | 方向 | 宽度 | 信号名 | 介绍 |
|---|---|---|---|---|
| Command Channel | Input | 1 | icb_cmd_valid | 主设备发送读写请求信号 |
| Command Channel | Output | 1 | icb_cmd_ready | 从设备返回读写接受信号 |
| Command Channel | Input | 32 | icb_cmd_addr | 读写地址 |
| Command Channel | Input | 1 | icb_cmd_read | 读或是写操作的指示 |
| Command Channel | Input | 32 | icb_cmd_wdata | 写操作的数据 |
| Command Channel | Input | 4 | icb_cmd_wmask | 写操作的字节掩码 |
| Response Channel | Output | 1 | icb_rsp_valid | 从设备发送读写反馈请求信号 |
| Response Channel | Input | 1 | icb_rsp_ready | 主设备返回读写反馈接受信号 |
| Response Channel | Output | 32 | icb_rsp_rdata | 读反馈的数据 |
| Response Channel | Output | 1 | icb_rsp_err | 读或者写反馈的错误标志 |

控制通道中，为了提高 icb 从机对主机的响应速度，一旦检测到主机发送的读写请求信号，立即对其做出响应，因此采用组合逻辑：

```
always_comb begin : icb_cmd_ready
    if ( icb_cmd_valid ) begin
        if ( !icb_cmd_read && icb_cmd_addr == ICB_SLAVE_WDATA &&
            full ) begin
            icb_cmd_ready = 1'b0;
        end
        else begin
            icb_cmd_ready = 1'b1;
        end
    end
    else begin
        icb_cmd_ready = 1'b0;
    end
end
```

**需要注意的是**，如果 wfifo 满，且 icb 主机发送写请求，数据继续写入可能会导致 wfifo 内数据被覆写，因此在这种情况下，从机 icb_cmd_ready 信号拉低不对主机读请求响应。

响应通道中，同样为了提高 icb 总线传输效率，在设计中，从机在检测到控制通道完成握手后下一周期，拉高读写反馈接受信号（rsp_valid），直到检测到主机读写反馈

接受信号（rsp_valid）后拉低，表示完成一次读写过程。下面分别考虑读写两种情况：

**主机读请求**：在拉高读写反馈接受信号（rsp_valid）同周期给出响应数据即可。但是，如果 icb 主机请求的数据为 RDATA 寄存器，由于 icb 主机请求数据需要从 rfifo 中读出，而从机发出 rsp_valid 信号同周期需要给出数据，因此 icb 从机需要在响应提前一个周期向 rfifo 发出读请求（rd_en）信号更新 RDATA ，即在控制通道握手周期判断是否读寄存器 RDATA ，如果是则向 rfifo 请求数据。

```
assign rdata_en = icb_cmd_ready && icb_cmd_read && icb_cmd_addr ==
    ICB_SLAVE_RDATA;
```

同时，由于 rfifo 在接收到读数据使能信号，下一个周期才能返回 fifo 数据，而在前面我们提到，从机一旦检测到主机发送的读写请求信号，需要立即对其做出响应，下一周期即返回结果，因此这里我们必须采用组合逻辑对 icb_rsp_rdata 赋值，而不能是时序逻辑。同时需要增加 mux 选择数据来源是 icb 从机的 csr 寄存器还是 rfifo，选通逻辑为 rfifo 读数据有效信号。

```
assign icb_bus.icb_rsp_rdata = fifo_data_vld ? rdata : icb_rsp_rdata_reg;
```

**主机写请求**：写逻辑较为简单，控制通道握手时序逻辑在下一周期更新相应寄存器即可，这里不多做赘述，各寄存器读写逻辑仅在地址判断上存在区别。

需要注意的是，由于在控制通道已经对 wfifo 能否写入（full）进行判断，且在 fifo 的实现中，读写均采用时序逻辑，将 fifo 本身视为一个多 bit 移位寄存器，因此 RDATA 寄存器与 WDATA 寄存器的实现并不必要。在设计中，这里直接将这两个寄存器映射到 wfifo 的写接口与 rfifo 的读接口，wfifo 需要额外提供写使能逻辑。

```
always_ff @(posedge clk or negedge rst_n) begin : wdata_vld
    if ( !rst_n ) begin
        wdata_vld <= 1'b0;
    end
    else begin
        if( icb_cmd_ready && !icb_cmd_read && icb_cmd_addr ==
            ICB_SLAVE_WDATA ) begin
            wdata_vld <= 1'b1;
        end else begin
            wdata_vld <= 1'b0; // 1 cycle pulse
        end
    end
end
```

## 2.2 加密模块（encrypt）

DES（Data Encryption Standard）是一种对称加密算法，用于数据的加密和解密。它是在 1970 年代末期开发的，并在 1980 年代成为美国联邦政府的标准加密算法。DES 算法的特点是明文按 64 位进行分组，密钥长 64 位，但实际上只有 56 位参与 DES 运算，其余 8 位用于奇偶校验。DES 加密过程包括多个步骤，如 IP 置换、F 轮函数、密钥生成等，最终通过 16 轮迭代产生密文。算法流程图如图 4所示：

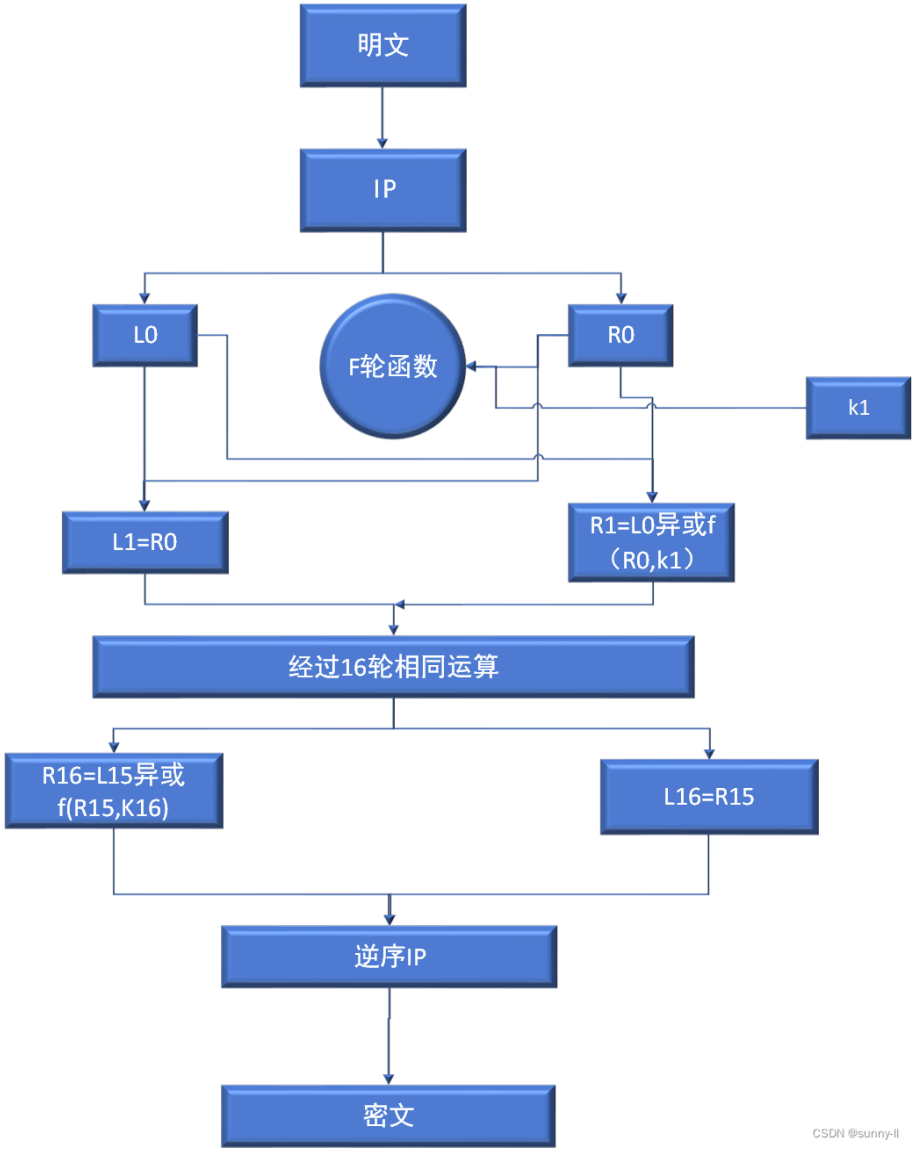


图 4: DES 算法流程图

在硬件实现 DES 算法时，由于基于查找表运算与异或运算，算法本身实现并不复杂，但由于需要考虑到算法的效率和速度，因此对 F 轮函数采用流水线化技术来优化性能。这里不对算法的代码实现进行展开介绍，仅基于波形图对算法的正确性进行验证说明：

在 testbench 中给予 DES 加解密算法模块输入激励，为了便于调试与验证，测试向

量的选取参考以下文章，对每一步输出结果进行验证：

- 算法科普：神秘的 DES 加密算法: https://cloud.tencent.com/developer/article/1497864

测试波形图如图 5所示：

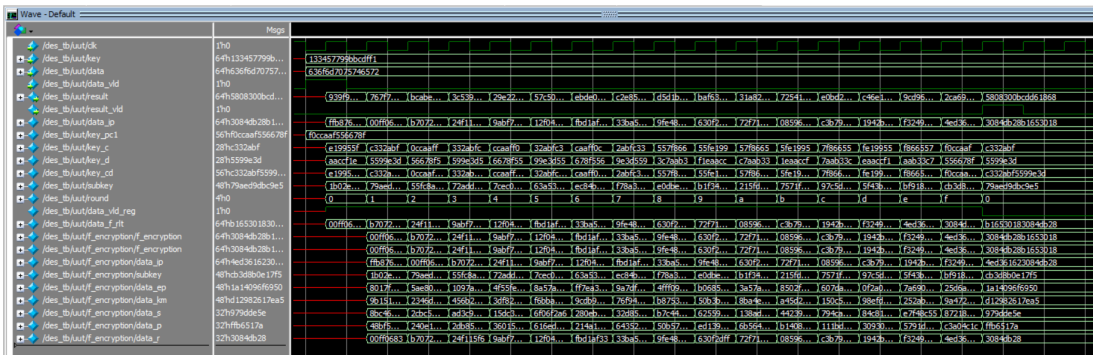

图 5: DES 测试波形图

将 16 轮 F 函数的输出结果以及最终的 result 输出与参考文章进行对比，容易验证 DES 算法的正确性！从波形图中我们也能看到算法的 Latency 为 16 个 clock cycle。

## 2.3 fifo 模块（fifo）

由于读写时钟域不同，对于 wfifo ，写端口在 icb 总线时钟域下，而读端口在 apb 时钟域下，因此设计中采用异步 fifo 的设计方法。具体到代码实现中，异步 fifo 的读写逻辑与同步 fifo 相似，唯一需要注意的是读写操作的时钟需要区分。与同步的 fifo 设计中不同的是，由于写指针 wptr 与读指针 rptr 也处于不同时钟域下，需要对时钟域进行同步进行空满判断。

当一个信号从一个时钟域传递到另一个不同时钟域时，由于时钟频率和相位的差异，信号可能会经历亚稳态。如果直接使用这个信号，可能会导致系统不稳定。因此这里使用打两拍（两个寄存器级）来同步指针。

```
always_ff @(posedge rclk or negedge rst_n) begin
    if (!rst_n) begin
        wptr_gray_sync1 <= 0;
        wptr_gray_sync2 <= 0;
    end else begin
        wptr_gray_sync1 <= wptr_gray;
        wptr_gray_sync2 <= wptr_gray_sync1;
    end
end
```

这里两拍同步的并非原始的读写指针，而是对读写指针进行二进制到格雷码的转换后的指针。格雷码是一种二进制编码，其中连续的数值之间的编码只有一位发生变化。这意味着在格雷码中，指针值的变化（如从 0110 到 0111）只涉及一位的跳变，而不是多位。这显著减少了由于时钟域不同步而可能发生的亚稳态情况。

```
function logic [ADDR_WIDTH:0] bin2gray(input logic [ADDR_WIDTH:0] bin);
    return (bin >> 1) ^ bin;
endfunction

assign wptr_gray = bin2gray(wptr);
assign rptr_gray = bin2gray(rptr);
```

对同步后的指针进行空满逻辑判断，空判断逻辑与同步 fifo 一致，由于格雷码的特性，满的判断需要进行修改。格雷码判满：最高位和次高位都不等。

```
//格雷码判空：读写指针相等
assign empty = (rptr_gray == wptr_gray_sync2);
//格雷码判满：最高位和次高位都不等
assign full = (wptr_gray ==
    {~rptr_gray_sync2[ADDR_WIDTH:ADDR_WIDTH-1],
    rptr_gray_sync2[ADDR_WIDTH-2:0]});
```

## 2.4 APB 主机模块（apb_master）

apb 协议的实现较为简单，但是，在该设计中，由于 apb 主机需要的数据信息与控制信息均以数据包的格式从 wfifo 中获取，完成一次 apb 读需要从 wfifo 中读取两个数据包（一个包含地址的控制信息，一个包含需要发送的数据），完成一次 apb 写需要从 wfifo 中读取包含控制信息的数据包，并对数据包解码获得控制信息（与数据信息），因此需要状态机来进行控制。数据包格式如表 3所示：

表 3: DUT 数据包格式

| **[31:8]** | **[7:2]** | **[1]** | **[0]** |
|---|---|---|---|
| addr | select | cmd | flag |
| apb 地址 | 000001：APB0<br>000010：APB1<br>000100：APB2<br>001000：APB3 | 0：读<br>1：写 | 0：控制 |
| data[30:0] | | | 1: 数据 |

这里将 apb 主机行为分为两个过程：数据准备与驱动 apb 数据传输。其中在数据

准备中包括从 wfifo 中读取数据包与解码数据包两个过程，模块的状态机如图 6所示：

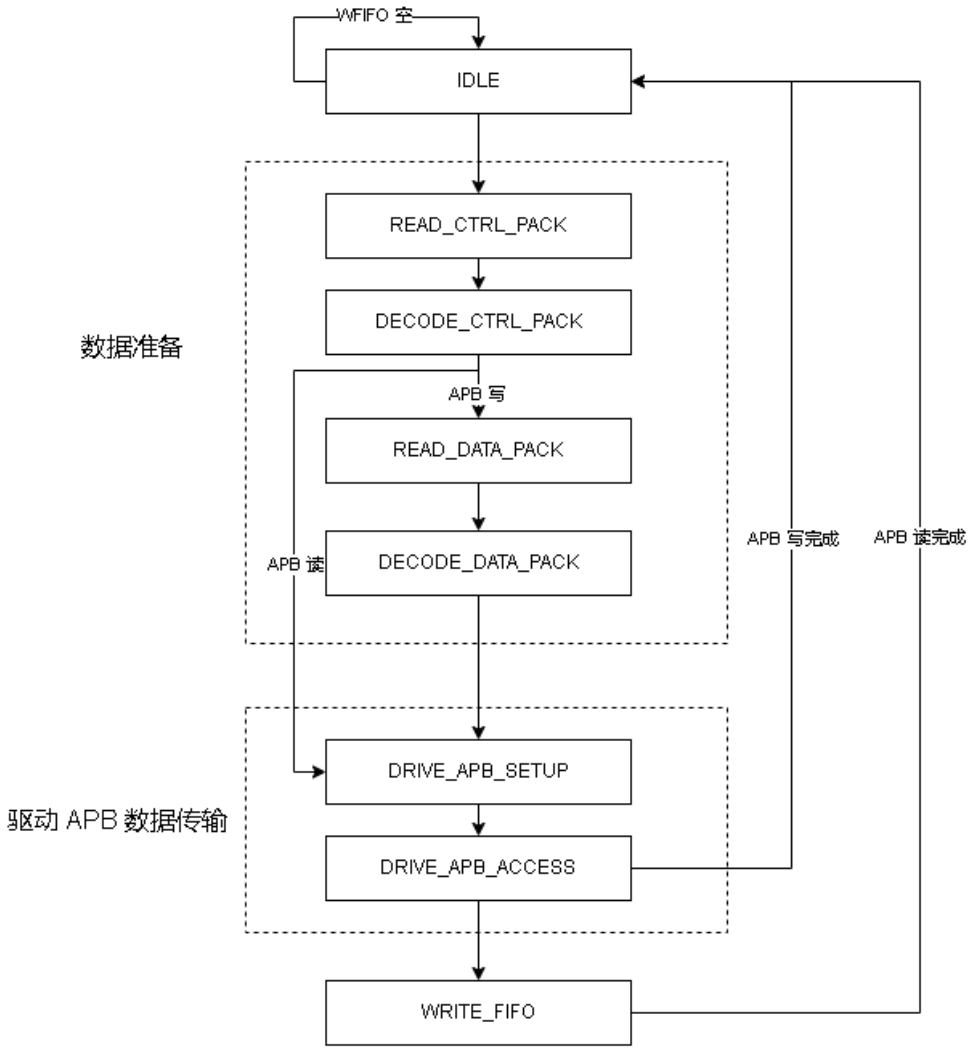


图 6: fsm

具体到代码实现如下：

```
always_comb begin
    case ( state )
        IDLE: begin
            if ( !empty ) begin
                next_state = READ_CTRL_PACK;
            end else begin
                next_state = IDLE;
            end
        end
        READ_CTRL_PACK: begin
            next_state = DECODE_CTRL_PACK;
```

```
            end
            DECODE_CTRL_PACK: begin
                if ( pack_write ) begin // apb_write
                    if ( !empty ) begin
                        next_state = READ_DATA_PACK; // wait for fifo not
                            empty
                    end else begin
                        next_state = DECODE_CTRL_PACK;
                    end
                end else begin // apb_read
                    next_state = DRIVE_APB_SETUP;
                end
            end
            READ_DATA_PACK: begin
                next_state = DECODE_DATA_PACK;
            end
            DECODE_DATA_PACK: begin
                next_state = DRIVE_APB_SETUP;
            end
            DRIVE_APB_SETUP: begin
                next_state = DRIVE_APB_ACCESS;
            end
            DRIVE_APB_ACCESS: begin
                if (apb_pready) begin
                    if ( pack_write_reg ) begin // apb_write
                        next_state = IDLE;
                    end else begin
                        next_state = WRITE_FIFO; // apb_read
                    end
                end else begin
                    next_state = DRIVE_APB_ACCESS;
                end
            end
            WRITE_FIFO: begin
                if ( !full ) begin
                    next_state = IDLE;
                end else begin
```

```
                next_state = WRITE_FIFO;
            end
        end
        default: begin
            next_state = IDLE;
        end
    endcase
end
```

**下面对各个状态进行分析：**

**IDLE**：状态机初始状态或无 apb 读写任务的默认状态，一旦检测到 wfifo 非空，即存在数据包，则启动 apb 读写任务。

**READ_CTRL_PACK**：这里依据数据包最后一位的不同，将其分为 CTRL_PACK 控制信息包与 DATA_PACK 数据包。这里要求 icb 主机完成一次 apb 从机写操作需要先发送一个控制信息包，再发送一个数据包，完成一次 apb 从机读操作则只需要发送一个控制信息包。因此由 IDLE 状态进入 apb 读写任务时，首先需要读 wfifo 获得控制信息包，wfifo 接收到读使能信号后下一周期发送数据。

```
assign rdata_en = (state == READ_CTRL_PACK ) || (state ==
    READ_DATA_PACK) ;
```

**DECODE_CTRL_PACK**：对数据包进行解码，同时判断进行主机需要对 apb 从机读还是写：apb 读直接驱动 apb 总线，apb 写则需要继续读取 wfifo 内的数据包解码获得写数据。由于完成一次 apb 写操作需要读两次 wfifo ，且无法同时从 wfifo 中读取控制信息与数据信息，因此需要对控制信息进行寄存器寄存，以便等控制信息与数据信息对齐后再驱动 apb 总线写。

```
always_ff @(posedge clk or negedge rst_n) begin
    if ( !rst_n ) begin
        pack_sel_reg <= 6'b0;
        pack_addr_reg <= 24'b0;
        pack_write_reg <= 1'b0;
    end
    else begin
        if ( pack_valid & ( pack_flag == 0 ) ) begin // ctrl pack
            pack_sel_reg <= pack_sel;
            pack_addr_reg <= pack_addr;
            pack_write_reg <= pack_write;
```

```
            end
        end
    end
```

**READ_DATA_PACK**：读 wfifo 数据包。

**DECODE_DATA_PACK**：解码数据包，寄存数据，下一周期驱动 apb setup。

**DRIVE_APB_SETUP**：从寄存控制信息与数据的寄存器中读出驱动 apb 总线需要的控制信息与数据信息，同时拉高总线 apb_setup 信号。

```
always_comb begin
    if ( apb_setup && (pack_sel_reg == 6'b000001) ) begin
        apb_bus_0.pwrite = pack_write_reg;
        apb_bus_0.psel = 1'b1;
        apb_bus_0.paddr = pack_addr_reg;
        apb_bus_0.pwdata = pack_wdata_reg;
    end
    else begin
        apb_bus_0.pwrite = 1'b0;
        apb_bus_0.psel = 1'b0;
        apb_bus_0.paddr = 32'b0;
        apb_bus_0.pwdata = 32'b0;
    end
end

assign apb_setup = (state == DRIVE_APB_SETUP) || (state ==
    DRIVE_APB_ACCESS); // apb setup and access
```

**DRIVE_APB_ACCESS**：维持控制信息与数据信息，apb_setup 信号，拉高 apb_access 信号。等待 apb 从机响应，一旦检测到从机的 pready 信号，完成本次 apb 读写操作。同时如果本次操作是 apb 读，则寄存读取到的数据，进入 WRITE_FIFO 状态，将数据准备写入 rfifo。

**WRITE_FIFO**：当 rfifo 不满时，将寄存的 apb 读数据写入 rfifo，下一周期回到 IDLE。

```
assign wdata_vld = (state == WRITE_FIFO) && !full ;
```

## 2.5 DUT：使能 DES

DES 为对称加解密算法，但是由于加密与解密在实现时，子密钥需要逆序，因此通过定义 parameter DES_TYPE 来实现对加密解密的不同例化。同时，使用 ‘define 来控制 DUT 的 encrypt 是否启用 DES：

```
# define.sv
    ‘define DES

# dut.sv
    ‘ifdef DES
        // DES algorithm
        ...
    ‘else
        // no encrypt
        assign result = data;
        assign result_vld = data_vld;
    ‘endif
```

# 3 验证平台设计

## 3.1 Icb Agent

在 ICB 总线代理层，需要完成主机数据的产生并驱动 ICB 总线向 DUT 发送数据。ICB AGENT 的设计包括数据生成器（Generator）、驱动器（Driver）和监视器（Monitor）三个主要组件，它们协同工作以实现对 ICB 总线的数据传输和监控。以下是对 ICB AGENT 各个模块的详细介绍：

### 3.1.1 数据生成器（icb_generator）

generator 主要负责生成用于传输的事务数据。通过 data_gen 任务，根据输入参数（读/写标志、数据掩码、数据值和地址）创建一个新的事务数据对象，并将其发送到驱动器。同时通过使用 mailbox 机制（gen2drv）与驱动器通信，确保数据的有序传输。

### 3.1.2 驱动器（icb_driver）

driver 通过 set_intf 函数设置与 ICB 总线的接口，并初始化端口状态。data_trans 任务从 mailbox 中获取数据，将接收到的数据包转换为 ICB 协议格式设置 ICB 总线控制信号，等待握手完成，然后结束事务。这里我们在主机接口中加入 clocking block 来对时序进行调整，用于仿真中模拟实际的信号传播延迟，clocking block 如下所示：

```
clocking mst_cb @(posedge clk);
    default input #1 output #1;
    output icb_cmd_valid, icb_cmd_read, icb_cmd_addr, icb_cmd_wdata,
        icb_cmd_wmask, icb_rsp_ready;
    input icb_cmd_ready, icb_rsp_valid, icb_rsp_rdata, icb_rsp_err;
endclocking
```

在 clocking block 时钟域下，驱动 ICB 总线的过程如下所示：

```
task automatic data_trans();
    icb_trans get_trans;

    // get the input data and address from mailbox
    this.gen2drv.get(get_trans);

    // setup the transaction
    @(this.active_channel.mst_cb)
    this.active_channel.mst_cb.icb_cmd_valid <= 1'b1;
    this.active_channel.mst_cb.icb_cmd_read <= get_trans.read;
```

```
    this.active_channel.mst_cb.icb_cmd_wmask <= get_trans.mask;
    this.active_channel.mst_cb.icb_cmd_wdata <= get_trans.wdata;
    this.active_channel.mst_cb.icb_cmd_addr <= get_trans.addr;
    this.active_channel.mst_cb.icb_rsp_ready <= 1'b1;

    // wait until the handshake finished
    while(!this.active_channel.icb_cmd_ready) begin
        @(this.active_channel.mst_cb);
    end

    // end the transaction
    this.active_channel.mst_cb.icb_cmd_valid <= 1'b0;
endtask //automatic
```

驱动总线数据后，等待 DUT 发送 icb_cmd_ready 信号完成命令通道握手后拉低 icb_cmd_valid 信号。

### 3.1.3 监视器（icb_monitor）

monitor 收集 ICB 总线上的数据，并将其转换为数据包，以便与得分板（scoreboard）进行比较。其中 mst_monitor 和 slv_monitor 任务分别监控主设备和从设备的信号，记录读/写操作和数据，并将部分信息打印在测试终端便于测试观察。另外，monitor2scoreboard 任务将监视到的数据发送到得分板，用于验证测试结果。

由于 monitor 只需要收集监控 ICB 总线信息与数据，因此在 interface 定义中，创建了 monitor 的 modport，并将所有 ICB 总线信号全部定义为输入信号。类似于 driver，这里同样定义了 moniter 的 clocking block。

monitor 主要分为主机监控（mst_monitor）与从机监控（slv_monitor）两个独立的模块，分别对主机行为与从机响应过程进行监控。主机监控与从机监控行为分别如以下代码所示：

```
task automatic mst_monitor(ref bit is_read);

    @(this.monitor_channel.mnt_cb)
    while(!this.monitor_channel.icb_cmd_ready) begin
        @(this.monitor_channel.mnt_cb);
    end
    this.monitor_trans.read = this.monitor_channel.icb_cmd_read;
    this.monitor_trans.mask = this.monitor_channel.icb_cmd_wmask;
    this.monitor_trans.wdata = this.monitor_channel.icb_cmd_wdata;
```

```
    this.monitor_trans.addr = this.monitor_channel.icb_cmd_addr;

    is_read = this.monitor_trans.read;

    if(is_read) begin
        $display("ICB Master Read : Addr=%h", this.monitor_trans.addr);
    end else begin
        $display("ICB Master Write : Addr=%h, WData=%h",
            this.monitor_trans.addr, this.monitor_trans.wdata);
    end
endtask

task automatic slv_monitor(ref bit is_read);

    @(this.monitor_channel.mnt_cb)
    while(!this.monitor_channel.icb_rsp_valid) begin
        @(this.monitor_channel.mnt_cb);
    end

    this.monitor_trans.rdata = this.monitor_channel.icb_rsp_rdata;

    if(is_read) begin
        $display("ICB Master Response : RData=%h ",this.monitor_trans.rdata);
    end
endtask
```

主机监控等待从机发送 icb_cmd_ready 信号，即命令通道握手成功后，进行主机信号采样 read、mask、wdata、addr 数据。从机监控等待从机发送 icb_rsp_valid 信号，即此时从机返回数据有效时，进行从机信号采样 rdata 数据。为了优化最终调试信息打印的输出，我们这里在顶层中还定义了 is_read 信号记录主机信号采样中的 read 信号，即 ICB 主机对 DUT 的是完成读还是写任务。如果是读任务，则在主机监控中无需打印 wdata 数据，在从机监控中需要打印 rdata 数据；如果是写任务，则反之。

此外，在 icb_monitor 类中，我们定义了区别于数据生成与驱动模块里的 mailbox（gen2drv），这里额外定义了一个 mailbox（icb_monitor_data），并通过任务 monitor2scoreboard 将主从机监控中采样到的数据发送至 scoreboard 进程。

#### 3.1.4 代理顶层（icb_agent）

agent 作为顶层类，连接生成器、驱动器和监视器，在 new 函数中初始化各个组件，并设置它们之间的通信 mailbox。single_tran 任务并行地调用生成器、驱动器和监视器的相关任务，以执行数据传输事务。

```
fork
    begin
        this.icb_generator.data_gen(read, mask, data, addr);
        this.icb_driver.data_trans();
        this.icb_monitor.monitor2scoreboard();
    end

    this.icb_monitor.mst_monitor( this.is_read );
    this.icb_monitor.slv_monitor( this.is_read );
join_any
```

这里需要注意的是，为了模拟仿真的真实性，主从机数据监控与数据的驱动过程应该是完全并行，因此使用 fork join_any 而不直接串行执行。

### 3.2 Apb Agent

在 APB 总线代理层，需要 APB 从机响应 DUT 的 APB 总线请求，并完成相应的读写任务。APB AGENT 的设计包括数据生成器（Generator）、驱动器（Driver）和监视器（Monitor）三个主要组件，它们协同工作以实现对 APB 总线的数据传输和监控。由于 APB Agent 的数据生成器与监视器模块与 ICB Agent 基本一致，这里不做过多赘述，仅对 apb_driver 与代理顶层进行说明：

#### 3.2.1 驱动器（apb_driver）

APB 从机不会主动发起事务请求，只需要对 DUT 的 APB 请求进行相应响应即可，由于在 Lab 设计中无需对 APB 从机的完整读写任务进行实现，因此，响应只体现在 APB 总线的驱动中，驱动的代码如下：

```
task automatic data_trans();
    apb_trans get_trans;

    // get the input data and address from mailbox
    this.gen2drv.get(get_trans);

    // wait until apb access
```

```
    while(!(this.active_channel.psel && this.active_channel.penable)) begin
        @(this.active_channel.slv_cb);
    end

    this.active_channel.slv_cb.pready <= 1'b1;
    this.active_channel.slv_cb.prdata <= this.active_channel.pwrite? 32'b0 :
        get_trans.rdata;

    // end the transaction
    @(this.active_channel.slv_cb)
    this.active_channel.slv_cb.pready <= 1'b0;
endtask //automatic
endclass //apb_driver
```

依据 APB 总线的时序要求，从机等待 psel 与 penable 信号都拉高后进行响应，拉高 pready 信号并判断 APB 主机的读写类型，如果读则返回 generator 产生的 rdata 数据，否则返回 32'b0。延迟一个时钟周期后拉低 pready 信号完成数据传输。

### 3.2.2 代理顶层（apb_agent）

apb 的代理顶层需要区别于 icb 的代理顶层，由于 icb 是主机主动发起，因此需要在 testbench 主动调用 icb agent 中的相关任务发起请求，而 apb 从机并不具备这种主动的行为逻辑，因此在仿真中，需要时刻检测 apb 总线信号并自发响应 apb 主机的驱动。因此这里我们在事务顶层定义了任务 single_channel_agent，通过 while(1) 循环不断保持被动响应过程，同时在每个 while(1) 循环的结束需要 #1 防止仿真时在一个时间片一直调用而陷入死循环。另外，从机并不具备实际功能，写入的 wdata 数据由于不会被实际记录，因此在返回数据时，通过随机数产生 rdata 返回而不需要顶层调用手动指定。

```
    while(1) begin

        void'(random_trans.randomize());
        fork
            begin
                this.apb_generator.data_gen(random_trans.rdata);
                this.apb_driver.data_trans();
                this.apb_monitor.monitor2scoreboard();
            end
            this.apb_monitor.mst_monitor(this.channel_id, this.is_read);
```

```
            this.apb_monitor.slv_monitor(this.channel_id, this.is_read);
        join
        #1;
    end
```

在从机响应中，我们还做了让仿真调试与终端打印更加人性化的优化，在 apb agent 类中定义了成员变量 channel_id，在例化时可以指定 channel_id，并在 monitor 打印中添加 channel_id 的信息打印以区分多个 apb 从机。

## 3.3 Scoreboard

scoreboard 类通过比较 ICB 和 APB 事务数据包，根据 ICB 和 APB 端输入输出及编解码结果判断传输结果的正确性，验证 DUT 作为 ICB 到 APB 桥接器的行为是否符合预期。为了简化行为级验证过程，我们首先 disable DUT 加解密模块的 DES 加密算法。区别于传统 UVM 框架下的验证模型，本次 DUT 的设计中，APB 端会接收到从机返回的读数据，并通过 rfifo，最终写入 RDATA 寄存器。**因此，在验证平台中，scoreboard 不仅需要完成 Host-To-Device 的行为级验证，即在 ICB 端产生有效激励，经总线桥处理后被译码为正确的 APB 激励发送，同时需要完成 Device-To-Host 的数据级验证，即 APB 端返回的数据经总线桥后写入 RDATA 寄存器的数据有效且正确，这一行为我们将通过 ICB 端主机额外发起一次 RDATA 寄存器的读请求获得该数据来实现正确性判断。**

### 3.3.1 Host-To-Device 行为级验证

scoreboard 类的顶层任务为 verify_top，与 apb agent 相似的是，由于 scoreboard 应在仿真全过程中始终处于等待响应状态，而不需要每次主动调用，因此顶层任务也采用 while(1) 循环。循环中，scoreboard 每个仿真时间片都会对 icb agent 的 mailbox 进行轮询，一旦检测到有效数据后，对其进行判断，如果 icb 写地址 32'h2000_0010（寄存器 WDATA），即表示会对 apb 从机进行相关读写事务请求，调用子任务 behavior_verify 进入验证流程。

在任务 behavior_verify 中，会对 icb 事务包进行解码，获得相应控制信息与数据。icb 事务包相关信息对应关系如图 7所示：

| [31:8] | [7:2] | [1] | [0] |
|---|---|---|---|
| addr | select | cmd | flag |
| 写到APB的地址，基地址为0x20000000 | 000001：APB0 000010：APB1<br>000100：APB2 001000：APB3 | 0：读<br>1：写 | 0：控制 |
| 数据 | | | 1：数据 |

图 7: icb 事务包

在 golden model 中，我们分别完成对 icb 控制包与 icb 数据包的解码，依据控制包的解码结果，如果解码获得的 apb channel 不在图 7的索引范围中，则打印"Invalid Channel ID , SCOREBOARD ERROR" 信息，如果 channel 存在则到相应 apb channel 的 agent 中获取 mailbox 中的数据信息，调用任务 behavior_verify 进行传输结果的正确性判断。behavior_verify 的判断逻辑如下所示：

```
if( ctrl_packet[1] == 1 ) begin
    if( apb_dri.addr == {8'b0,ctrl_packet[31:8]} && apb_dri.wdata ==
        {1'b0,data_packet[31:1]} ) begin
        $display("[Golden Model] Behavior Verify : APB Write Success !");
        this.behavior_pass = 1;
    end else begin
        $display("[Golden Model] Behavior Verify : APB Write Failed !");
        this.behavior_pass = 0;
    end
end else begin
    if( apb_dri.addr == {8'b0,ctrl_packet[31:8]} ) begin
        $display("[Golden Model] Behavior Verify : APB Read Success !");
        this.behavior_pass = 1;
    end else begin
        $display("[Golden Model] Behavior Verify : APB Read Failed !");
        this.behavior_pass = 0;
    end
end
```

其中，behavior_pass 可以理解为 bool 类型的变量，用于记录是否通过 Host-To-Device 验证，下一小节中的 data_pass 类似。这两个 bool 类型的变量将最终用于 Scoreboard 计分板的输出结果打印。

### 3.3.2 Device-To-Host 数据级验证

Device-To-Host 数据级验证的验证思路较为简单，只需要暂存 apb_monitor 中监听到的 rdata 数据，同时监听 ICB 端发起的 RDATA 寄存器读请求时由 DUT 返回的 ICB 总线的 rdata 数据。当两者数据均获得后，进行比较即可。data_verify 的判断逻辑如下所示：

```
// start to verify
if(icb_rdata.addr == 32'h2000_0018 && icb_rdata.read == 1 &&
```

```
        icb_rdata.rdata ==
        des_encrypt({32'b0,apb_rdata},64'h1234_5678_9abc_def0)) begin
        $display("[Golden Model] Data Verify : Read Data Right !");
        this.data_pass = 1;
    end else begin
        $display("[Golden Model] Data Verify : Read Data Wrong !");
        this.data_pass = 0;
    end
```

## 3.4 仿真环境 env 与验证顶层

在仿真环境 env 顶层，完成各接口的例化以及类的实例。在主任务 run 中，接收 testbench 发送的测试参数 state，并依据 state 进行不同的仿真任务，需要注意的是，仿真任务的进行与各 channel 的 apb agent 以及 scoreboard 均为并行发生，这一点在 3.2，3.3 节已经进行过详细的解释，因此主任务同样采用 fork join。

根据测试参数 state 的不同，run 将仿真任务分为"ICB Write Test"、"ICB RAW Test"、"APB Write"、"APB Read"、"LOOPBACK Test"、"RANDOM Test"、"Time_Run"。下面对各任务进行简单描述：

1. "ICB Write Test"：主机对 DUT 的所有可写寄存器的写测试。
2. "ICB RAW Test"：主机对 DUT 的所有可读可写寄存器进行 RAW（写后读）测试。
3. "APB Write"：主机向 DUT 的 WDATA 寄存器发送控制包与数据包完成 APB 写请求测试。
4. "APB Read"：主机向 DUT 的 WDATA 寄存器发送控制包完成 APB 读请求测试。
5. "LOOPBACK Test"：基于"APB Write" 与"APB Read" 任务后，主机读取 DUT 的 RDATA 寄存器获取 APB Read 返回的数据测试。
6. "RANDOM Test"：主机随机发送有效的 APB 读写请求事务包，通过 scoreboard 完成每次行为的正确性判断。
7. "Time_Run"：用于检测仿真系统是否超时。

下面对 RANDOM Test 任务进行详细解释：

```
task random_test();

// Randomization of test data
```

```
icb_trans ctrl_packet;
icb_trans data_packet;
bit request_type;
bit [5:0] channel_sel;
int case_cnt = 0;

ctrl_packet = new();
data_packet = new();

repeat (10) begin // Repeat the random test for 10 times

    #($urandom_range(20, 100) * 10); // Random delay between 200ns to 1000ns

    channel_sel = 6'b010000;
    channel_sel >>= $urandom_range(1, 4); // Random channel selection

    void'(ctrl_packet.randomize());
    void'(data_packet.randomize());

    request_type = ctrl_packet.wdata[1] ; // 0 for read, 1 for write

    // Drive ICB master with randomized data
    if (request_type) begin
        $display("================================
            Random Write ================================");
        $display("time : @ %t ns", $realtime/1000);
        this.icb_agent.single_tran(1'b0, 8'h00, {32'b0, ctrl_packet.wdata[31:8],
            channel_sel, 1'b1, 1'b0}, WDATA_ADDR); // apb bus0 write addr
            0000004
        this.icb_agent.single_tran(1'b0, 8'h00, {32'b0, data_packet.wdata[31:1],
            1'b1}, WDATA_ADDR); // data 8
    end else begin
        $display("================================
            Random Read
            ================================");
        $display("time : @ %t ns", $realtime/1000);
        this.icb_agent.single_tran(1'b0, 8'h00, {32'b0, ctrl_packet.wdata[31:8],
```

```
                channel_sel, 1'b0, 1'b0}, WDATA_ADDR); // apb bus0 read addr
                    0000004 // data 8
            #200; // 由于异步时钟设计打了两拍，数据写入后 empty
                信号等两周期才会拉低
            this.icb_agent.single_tran(1'b1, 8'h00, 64'h0000_0000_0000_0000,
                RDATA_ADDR); // icb read rdata
        end
    end

    #200; // Wait for the last transaction to complete
    $display("============================ Random Test Finish !
        ============================");
endtask
```

在 RANDOW TEST 随机化测试中，完成了测试请求的随机化，即在随机的时刻驱动 ICB master 发起指令，并据情况接收 APB 端的结果以及测试数据，以及 ICB 的 wdata、address、请求类型的随机化。需要注意的是，这里的随机化测试是主机通过 ICB-APB 总线桥对 APB 从机的随机化读写测试，因此 ICB 发送的数据包与控制包并不是完全随机，例如对其他未定义的地址读写以及未定义的 channel 的读写是不可行的。并且，由于 DUT 数据的传输以及解码存在一定的 latency，因此在完成 ICB 事务包的发送之后，不能立即读取 RDATA 寄存器获取 APB 返回的数据。

此外，我们在终端调试信息打印中加入时仿真时间的打印。

## 3.5 使能 DES：完整的验证平台

下面，我们在 encrypt 模块与 decrypt 模块中启用 DES 对称加密算法，同时对验证平台进行相应的完善优化，进行更加完整可靠的系统级验证。完善工作我们简化如图 8所示：

### 3.5.1 ICB 主机端：原始激励加密

在前面几小节中，原始激励中的 addr 与 wdata 均由 randomize() 随机函数产生，但是**值得注意的是**，需要额外对控制包与数据包的格式以及 channel_sel 字段进行控制，即产生满足 DUT 译码格式的控制包以及数据包，才可以进行 icb tansaction 任务。但是，在 DUT 使能 DES 模块后，发送的原始激励经过 DUT 内部的一轮 DES 解密计算后才会送入译码模块进行译码，而经过一轮解密计算后的控制包与数据包不再具备可被识别并译码的格式，因此会造成 DUT 工作异常甚至崩溃。因此，使能 DUT 内部的 DES 模块后，发送的激励不可将满足译码格式的包直接发送，而需要先经过一轮 DES 加密计算，而这里的加密计算，应该由 ICB 主机端来模拟完成。这里我们额外实现了 C

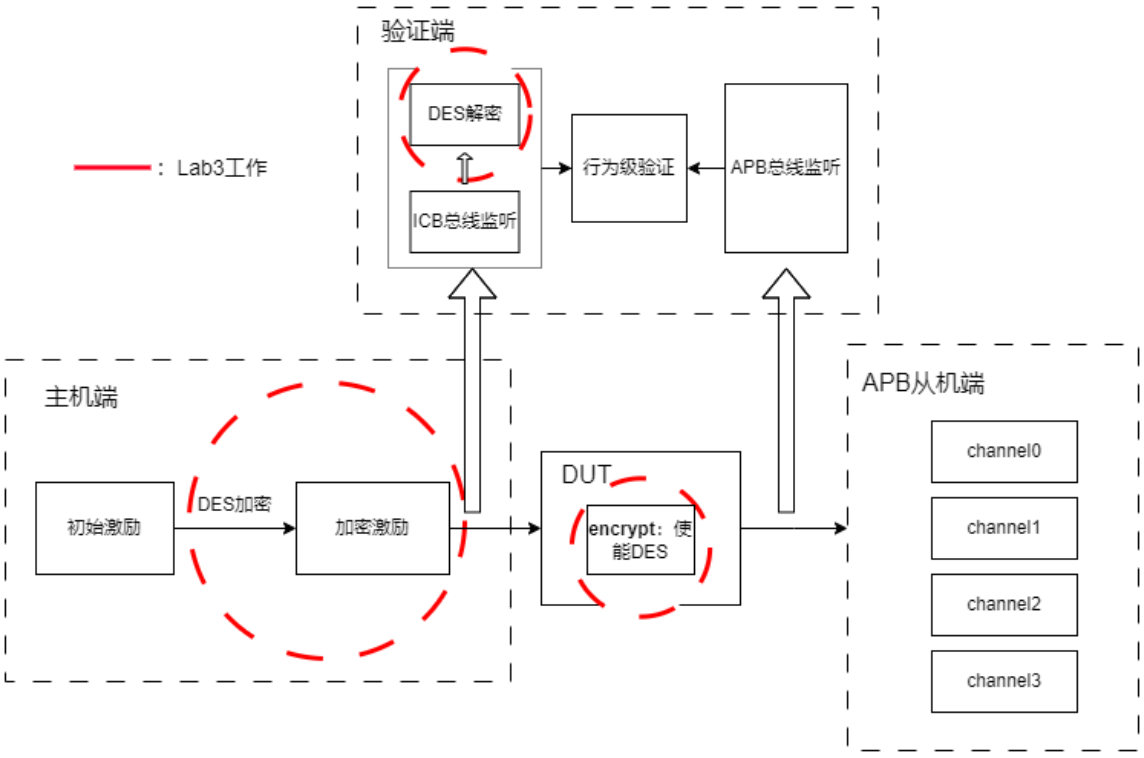


图 8: 验证框架

逻辑下的 DES 加解密 function 部分，将原始满足译码格式的包先加密后再发送。

```
void'(input_stimulus.randomize());
request_type = input_stimulus.write;

ctrl_packet_raw = {32'b0, input_stimulus.addr, input_stimulus.channel_sel,
    input_stimulus.write, 1'b0};
data_packet_raw = {32'b0, input_stimulus.wdata, 1'b1};

`ifdef DES
    ctrl_packet_true =
        des_encrypt(ctrl_packet_raw,64'h1234_5678_9abc_def0);
    data_packet_true =
        des_encrypt(data_packet_raw,64'h1234_5678_9abc_def0);
`else
    ctrl_packet_true = ctrl_packet_raw;
    data_packet_true = data_packet_raw;
`endif
```

这里的 ctrl_packet_raw 与 ctrl_packet_true 分别为原始满足 DUT 译码格式的控制包与经过 ICB 主机端 DES 加密后的控制包。与 DUT 相对应的，我们同样使用 `define

来保证验证平台与 DUT 的加密/不加密的行为一致性。

### 3.5.2 验证端：加密激励解密

类似地，验证端对 moniter 采集到的 ICB 总线数据，需要对其进行一轮 DES 解密，才能得到原始的控制包与数据包。从另外一个角度考虑，这里的验证端需要模拟 DUT 中的 DES decrypt 模块，才能进行 4.5 节中的 golden model 验证。

```
// decrypt ctrl packet
‘ifdef DES
    ctrl_packet_decrypt =
        des_decrypt(ctrl_packet_icb,64’h1234_5678_9abc_def0);
‘else
    ctrl_packet_decrypt = ctrl_packet_icb;
‘endif
ctrl_packet_decrypt_32 = ctrl_packet_decrypt[32:63];
...

// decrypt data packet
‘ifdef DES
    data_packet_decrypt =
        des_decrypt(data_packet_icb,64’h1234_5678_9abc_def0);
‘else
    data_packet_decrypt = data_packet_icb;
‘endif
data_packet_decrypt_32 = data_packet_decrypt[32:63];
...
```

**值得注意的是**，这里我们在调用 C 逻辑下的 DES 加解密函数时，指定的第二个参数 64’h1234_5678_9abc_def0 实际上是 DES 中的 key，而 DUT 中的 key，则是我们在测试开始时，首先需要调用 icb.single_tran 驱动向 ICB Agent 中寄存器 KEY 写入。

```
$display(”[TB− ENV ] Write KEY register.”);
this.icb_agent.single_tran(1’b0, 8’h00, 64’h1234_5678_9abc_def0, KEY_ADDR);
```

我们可以看到，这里写入的 key 与调用 C 逻辑的 DES 函数的 KEY 参数值一致。

## 3.6 验证结果及分析

### 3.6.1 ICB 端总线时序验证

对 ICB 端总线时序的验证，这里我们在 testbench 顶层调用任务 ICB WRITE TEST 与 ICB RAW TEST 进行波形图分析与说明。

在 ICB WRITE TEST 任务中，我们依次对 DUT 的 CTRL、WDATA、KEY 寄存器进行写操作。同时，由于 WDATA 被映射到 WFIFO 中，因此这里我们对其连续进行次数等于 WFIFO 深度的连续写入，以验证 WFIFO 的空满信号是否能及时拉高以阻塞后续对 WFIFO 的写入。ICB 总线的波形图如图 9所示：

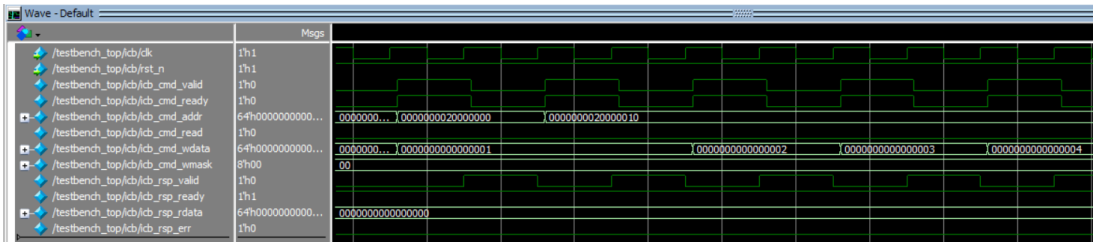


图 9: ICB 总线写时序验证

在 ICB RAW TEST 任务中，我们分别对 DUT 的 CTRL、KEY 寄存器进行读后写操作，ICB 总线的波形图如图 10所示：

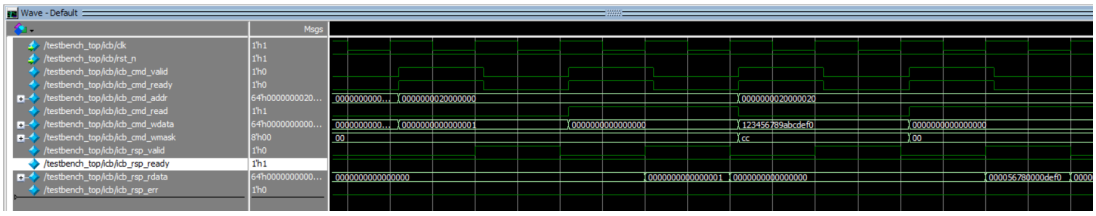


图 10: ICB 总线读写时序验证

同时我们将波形图与终端打印的调试信息图 11进行对比：

需要注意是，我们在向 KEY 寄存器写入数据时指定了 mask 为 8'hcc，因此只有 1、2、5、6 字节被写入有效数据，从读出的数据中可以验证这一点。波形图与输出信息打印同样验证 ICB 总线在 RAW 测试下的正确性！

### 3.6.2 APB 端总线时序验证

APB 端总线时序的验证较为特殊，由于 DUT 作为 APB 主机不会自发生成 APB 总线上的读写激励，而是由 ICB 发送的数据包经解码后获得相关的控制信息或 APB 的写入数据，因此这里在验证时，我们先将 DES 加解密模块 disable，使得在主机端发送的数据包可以直接经由 DUT 译码，这也方便我们进行调试与 debug。

APB 端总线时序的验证主要分为两个部分，分别为 APB 读测试与 APB 写测试，这里我们使用的测试向量参考 Lab1 中 APB 子模块独立 testbench 的测试向量。在 APB

```
# ================================================================
# [TB- ENV ] Start work : ICB Read !
# [TB- ENV ] Write CTRL register.
# ICB Master Write : Addr=20000000, WData=0000000000000001
# [TB- ENV ] Read CTRL register.
# ICB Master Read : Addr=20000000
# [TB- ENV ] Write KEY register.
# ICB Master Response : RData=0000000000000001
# ICB Master Write : Addr=20000020, WData=123456789abcdef0
# [TB- ENV ] Read KEY register.
# ICB Master Read : Addr=20000020
# ICB Master Response : RData=000056780000def0
```

图 11: RAW_TEST 终端打印信息

读测试中，发送读控制包，对 APB 从机 channel 0 的偏移地址 24'h000004 进行读；在 APB 写测试中，依次发送写控制包以及写数据包，对 APB 从机的 channel 0 的偏移地址 24'h000004 进行写数据 32'h8。

由于在 4.3 节中我们对数据流进行了完整的 LOOPBACK 验证，LOOPBACK 测试本身即包含 APB 的写，APB 的读，因此这一部分更为详细的验证见 4.3。这里列出 APB 读写的波形图如图 12，图 13所示：

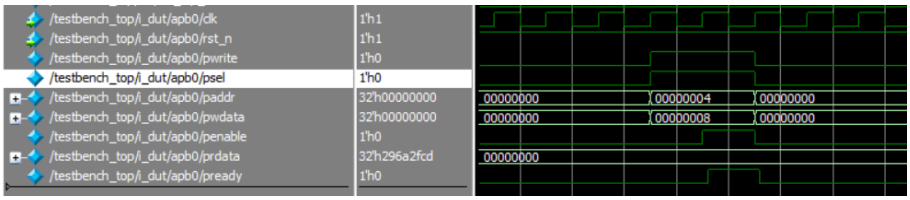


图 12: APB 写波形

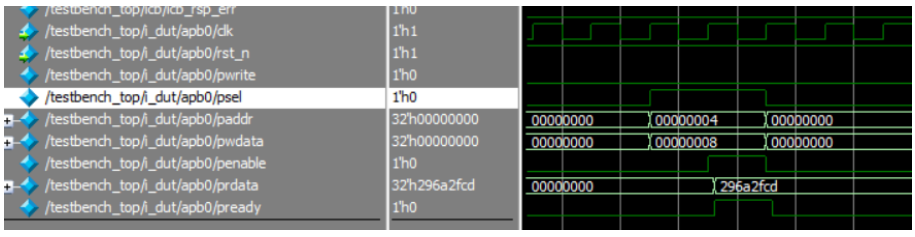


图 13: APB 读波形

### 3.6.3 数据流 LOOPBACK 验证

在数据流的 LOOPBACK 验证中，我们主要完成了一整套 ICB 主机到 APB 从机的写测试以及读测试，而不局限于某一端的验证，LOOPBACK 测试流程图如图 14所示：

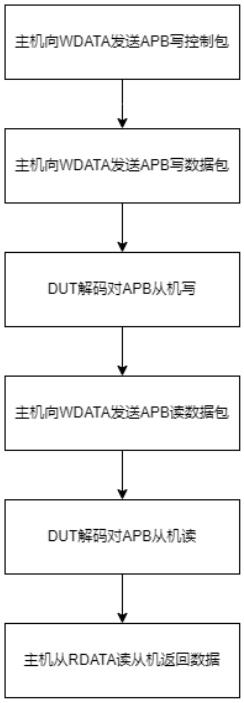


图 14: LOOPBACK 流程图

通过终端监控信息的打印，我们很容易验证完整数据流的正确性。终端信息打印如图 15所示：

```
[TB- SYS ] running
============================================================
[TB- ENV ] Start work : LOOPBACK Test !
ICB Master Write : Addr=20000010, WData=0000000000000406
ICB Master Write : Addr=20000010, WData=0000000000000011
APB Deocode : APB Master channel_0 Write: Addr=00000004, WData=00000008
----------------------golden model----------------------
|     APB Write Success !                              |
|     Pass / Total :          1 /          1           |
|     Pass Rate : 100.000000%                          |
-------------------------------------------------------
ICB Master Write : Addr=20000010, WData=0000000000000404
APB Deocode : APB Master channel_0 Read: Addr=00000004
APB Slave channel_0 Response : RData=296a2fcd
----------------------golden model----------------------
|     APB Read Success !                               |
|     Pass / Total :          2 /          2           |
|     Pass Rate : 100.000000%                          |
-------------------------------------------------------
ICB Master Read : Addr=20000018
ICB Master Response : RData=00000000296a2fcd
```

图 15: LOOPBACK 测试终端打印

这里我们可以看到 APB 从机返回数据 Rdata=296a2fcd，被主机通过 RDATA 寄存

器（ADDR=20000018）读取，得到数据仍为 296a2fcd 被高 32 位补 0 得到的数据。此外，终端同样对每一次 APB 读写操作的正确性进行了 golden model 自动化验证，这一部分在 4.5 节有更加详细的解释。

#### 3.6.4 基于随机化测试的 golden model 验证

运行 RANDOM TEST 任务，连续进行若干次 APB 的随机读写任务，读写次数可以通过修改任务的 repeat(x) 中的 x 参数进行修改。由于我们已经实现了 golden model 根据 ICB 和 APB 端输入输出及编解码结果判断加解密和传输结果的正确性进行自动化判断，而无需测试人员通过观察波形或者 monitor 的打印信息自行分析验证，因此我们通过直接观察 golden model 的输出即可进行系统验证！golden model 的输出如图 16所示：

```
#                    golden model
# |     APB Read Success !                              |
# |     Pass / Total :           7 /           7        |
# |     Pass Rate : 100.000000%                         |
# -----------------------------------------------------
# ICB Master Read : Addr=20000018
# ICB Master Response : RData=0000000053d86f6a
# ============================== Random Write ==============================
# time : @                 6575 ns
# ICB Master Write : Addr=20000010, WData=000000009181b622
# ICB Master Write : Addr=20000010, WData=000000002fb08dbf
# APB Deocode : APB Master channel_3 Write: Addr=009181b6, WData=17d846df
# --------------------golden model---------------------
# |     APB Write Success !                             |
# |     Pass / Total :           8 /           8        |
# |     Pass Rate : 100.000000%                         |
# -----------------------------------------------------
# ============================== Random Read ==============================
# time : @                 7265 ns
# ICB Master Write : Addr=20000010, WData=0000000083c52120
# APB Deocode : APB Master channel_3 Read: Addr=0083c521
# APB Slave channel_3 Response : RData=7211b293
# --------------------golden model---------------------
# |     APB Read Success !                              |
# |     Pass / Total :           9 /           9        |
# |     Pass Rate : 100.000000%                         |
# -----------------------------------------------------
# ICB Master Read : Addr=20000018
# ICB Master Response : RData=000000007211b293
# ============================== Random Read ==============================
# time : @                 8405 ns
# ICB Master Write : Addr=20000010, WData=0000000046296608
# APB Deocode : APB Master channel_1 Read: Addr=00462966
# APB Slave channel_1 Response : RData=c250f978
# --------------------golden model---------------------
# |     APB Read Success !                              |
# |     Pass / Total :          10 /          10        |
# |     Pass Rate : 100.000000%                         |
# -----------------------------------------------------
# ICB Master Read : Addr=20000018
# ICB Master Response : RData=00000000c250f978
# ========================== Random Test Finish ! ==========================
# Break key hit
```

图 16: golden model 系统级验证

# 4 SVA 断言设计

断言（System Verilog Assertion 简称 SVA）可以被放在 RTL 设计或验证平台中，方便在仿真时查看异常情况。在本小节中，我们对 DUT 的关键模块，包括 ICB 总线端口、APB 总线端口、异步 FIFO 进行了 SVA 检查。在仿真平台运行下，对相关信号的逻辑正确性进行判断验证。

## 4.1 ICB 端断言检查

### 4.1.1 X 态检查

使用 $isunknown 对 ICB 总线信号在其有效/使用时的 X 态检查。例如:

```
always_ff @(icb.clk) begin
    if (icb.icb_cmd_valid && icb.icb_cmd_ready)
        cmd <= icb.icb_cmd_read;
    else
        cmd <= cmd;
end

property icb_rsp_rdata_no_x_check;
    @(posedge icb.clk) disable iff(!icb.rst_n)
    (icb.icb_rsp_valid && cmd) |-> (not ($isunknown(icb.icb_rsp_rdata)));
endproperty
```

这里首先定义 cmd 变量对最近一次 cmd 通道握手时，主机发起的读写行为类型进行记录。对于 icb_rsp_data 的 X 态检查，需要在需要在 icb_rsp_valid 的时候去查询最近一次 cmd 通道握手时的 read 的高低，如果是主机读请求，则对 icb_rsp_rdata 应当有效，使用 SVA 蕴含操作符 |-> 并调用 $isunknown 对其进行 X 态检查。其他信号同理。

### 4.1.2 稳定性检查

在 cmd 通道主机发起请求而从机未及时响应，或 rsp 通道从机发起请求而主机未及时响应时，总线信号应当保持稳定，使用 $stable 对其稳定性进行检查。例如:

```
property icb_rsp_rdata_keep_check;
    @(posedge icb.clk) disable iff(!icb.rst_n)
    (icb.icb_rsp_valid && !icb.icb_rsp_ready && cmd) |=>
        $stable(icb.icb_rsp_rdata);
endproperty
```

这里沿用在 5.1.1 节定义的 cmd 变量。rsp 通道从机发起请求但主机未及时响应，即 (icb.icb_rsp_valid && !icb.icb_rsp_ready) 时，查询最近一次 cmd 通道握手的 read 的高低，如果是主机读请求，则此时 icb_rsp_rdata 存在有效数据且在主机响应前必须保持稳定。其他信号同理。

### 4.1.3 握手检查

握手检查确保 ICB 总线的响应信号在正确的时间内完成握手过程。例如命令通道握手检查：

```
property icb_cmd_handshake_check;
    @(posedge icb.clk) disable iff(!icb.rst_n)
    (($rose(icb.icb_cmd_valid)) or ($past(icb.icb_cmd_valid &&
        icb.icb_cmd_ready) && icb.icb_cmd_valid)) |-> ##[0:$]
        icb.icb_cmd_valid && icb.icb_cmd_ready;
endproperty
```

## 4.2 APB 端断言检查

由于 APB 总线的设计目标是简单、高效且低功耗，适用于低带宽的外设连接，因此 APB 总线的时序相较于 ICB 端更为简单。X 态检查、稳定性检查与握手检查与 ICB 端基本同理。此外，基于 APB 总线的时序图，我们对其进行额外若干信号的时序检查。APB 总线时序图如图 17所示：

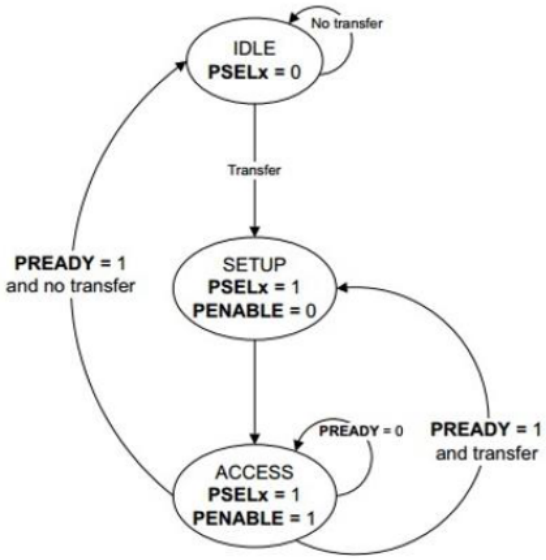


图 17: APBtiming

从时序图中我们可以看到：

1. penable 与 pready 握手后一周期必须拉低。

2. penable 拉高的前一周期，psel 必须为高。

3. psel 拉高的下一周期，penable 必须为高。

相应地，使用 |=> 与 $past 函数即可完成相关信号的验证：

```
// penable and pready must pull low after handshake
    property penable_after_handshake_check;
        @(posedge apb.clk) disable iff(!apb.rst_n)
        (apb.penable && apb.pready) |=> (!apb.penable);
    endproperty

// psel must be high one cycle before penable is pulled high
    property psel_before_penable_check;
        @(posedge apb.clk) disable iff(!apb.rst_n)
        $rose(apb.penable) |-> $past(apb.psel);
    endproperty

// penable must be high one cycle after psel is pulled high
    property penable_after_psel_check;
        @(posedge apb.clk) disable iff(!apb.rst_n)
        $rose(apb.psel) |=> apb.penable;
    endproperty
```

## 4.3 FIFO 断言检查

5.1 节 ICB 端断言检查与 5.2 节 APB 端断言检查更面向于对总线时序的 SVA 检查，在本小节中，我们对异步 FIFO 内部的相关控制信号进行更加逻辑上的 SVA 检查。

### 4.3.1 空满信号检查

由于异步 FIFO 的设计中，在跨时钟的读写指针比较上我们采用了打两拍的操作，因此这里需要引入**弱判断条件**的概念：当 FIFO 为空时,empty 信号一定拉高，但 empty 信号拉高时，FIFO 不一定为空，可能有异步的 delay；当 FIFO 为满时,full 信号一定拉高，但 full 信号拉高时，FIFO 不一定为满，可能有异步的 delay。因此，empty 与 full 信号的检查 SVA 实现如下：

```
    property fifo_empty_check;
        @(posedge rclk) disable iff(!rst_n)
```

```
    rptr == wptr |-> empty;
endproperty

property fifo_full_check;
    @(posedge wclk) disable iff(!rst_n)
    ((wptr[2:0] == rptr[2:0]) && (wptr[3]!=rptr[3])) |-> full;
endproperty
```

### 4.3.2 写入、读出功能检查

写入、读出功能的检查通过判断读写行为发生后指针变化来实现。在时钟上升沿检测到读写有效信号时，判断读写指针相较于上一周期是否完成自增即可。

```
property fifo_rd_function_check;
    @(posedge rclk) disable iff(!rst_n)
    rdata_en |=> rptr == ($past(rptr)+1);
endproperty
```

### 4.3.3 读写指针变化检查

由于在异步 FIFO 的设计中，使用格雷码对读写指针变换后才进行空满信号的判断，因此这里我们额外对读写指针的格雷码变换的行为进行正确性检查。同样的，我们首先定义了 bin2gray 的函数，我们依次在指针发送变换时（ptr!=($past(ptr)）对其进行检查：

```
property fifo_rptr_gray_code_check;
    @(posedge rclk) disable iff(!rst_n)
    (rptr!=($past(rptr))) |-> (rptr_gray == bin2gray(rptr));
endproperty

property fifo_rptr_gray_code_sync1_check;
    @(posedge rclk) disable iff(!rst_n)
    (rptr_gray!=($past(rptr_gray))) |=> (rptr_gray_sync1 ==
        ($past(rptr_gray)));
endproperty

property fifo_rptr_gray_code_sync2_check;
    @(posedge rclk) disable iff(!rst_n)
    (rptr_gray_sync1!=($past(rptr_gray_sync1))) |=> (rptr_gray_sync2 ==
        ($past(rptr_gray_sync1)));
```

```
    endproperty
```

## 4.4 SVA：启用 SVA 与 bindfile 设计

为了更好地实现 SVA 代码与设计代码分离，我们使用 bindfile 文件来分离设计验证代码与 SVA 代码。bindfile 文件通过 bind 绑定的方式将断言模块与设计顶层模块进行连接，从而实现了验证代码的模块化和可重用性。在 bindfile 文件中，我们定义了多个绑定语句，每个绑定语句将一个断言模块与设计中的相应接口进行连接。icb_assertion 模块被绑定到 dut_top 模块的 icb.monitor 接口上，apb_assertion 模块被实例化多次，分别绑定到不同的 APB 通道上，以检查各个通道的行为是否符合预期，fifo_assertion 模块被绑定到 dut_top.rfifo 与 dut_top.wfifo，通过参数化的方式支持不同类型的 FIFO（RFIFO 和 WFIFO）的验证。这种模块化的绑定方式不仅提高了代码的可读性和可维护性，还使得断言模块可以在不同的设计中进行重用，提高了验证工作的效率。

同时，使用 ‘define 来控制是否启用 SVA：

```
# define.sv
    `define SVA


# dut.sv
    `ifdef SVA
        binding_module i_binding_module();
    `endif
```

# 5 重构验证平台：基于 UVM 的验证平台设计

UVM（Universal Verification Methodology）作为一种广受认可的验证方法学，提供了一套标准化的验证组件和流程，极大地提升了验证的可重用性和可扩展性。

在之前的章节中，我们已经基于 System Verilog 构建了一个验证平台。然而，随着项目规模的扩大和复杂度的提升，该平台逐渐暴露出一些局限性，例如可拓展性不足、组件管理复杂以及部分功能实现较为繁琐等。为了进一步提高验证效率和质量，我们决定基于 UVM 对验证平台进行重构。通过引入 UVM，我们能够更有效地管理验证环境中的各个组件，并实现模块化的验证设计，从而显著提升验证的效率和质量。

UVM 验证环境主要由 Agent、Scoreboard、Environment 和 Test 等组件构成。Agent 负责与 DUT 进行交互，包含驱动器（driver）、监视器（monitor）和序列器（sequencer）。**将 UVM 验证环境与我们现有的验证平台进行对比，我们可以发现二者的框架在逻辑上完全一致**。典型的 UVM 验证平台框图如图 18所示：

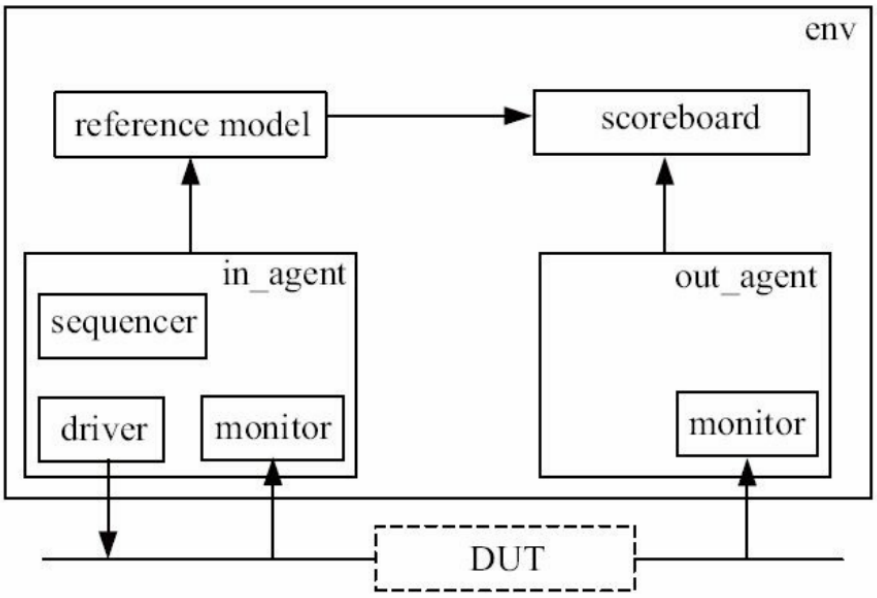


图 18: 典型 UVM 验证平台框图

本次验证平台的设计区别于上图，由于 DUT 需要接收 APB 端从机返回的数据，而对于该数据的正确性我们同样需要验证。但同样地，我们在之前的验证平台也考虑过这个问题，即可以理解为 DUT 反向，APB 端作为输入端，ICB 端作为输出端即可。因此，我们在重构验证平台时并未遇到太多问题，主要是在逻辑上参考现有的验证平台进行移植。在本节中，我们将重点关注 UVM 的特性机制与对 uvm_pkg 使用过程中的一些问题进行说明。

## 5.1 UVM 树状结构

UVM 树是 UVM 验证环境中组件层次结构的可视化表示，它清晰地展示了各个组件之间的关系和组织结构。在基于 UVM 的验证平台中，UVM 树的合理构建对于验证环境的可维护性和可扩展性至关重要。通过 UVM 树，我们可以直观地了解验证环境中各个组件的层次关系，便于进行模块化设计和功能扩展。在我们的验证平台中，UVM 树的构建遵循了 UVM 的标准架构，主要包括以下几个层次：

1. **Test 层**：Test 层是验证环境的顶层，负责启动和管理整个验证过程。在我们的验证平台中，Test 层包含了多个测试用例，每个测试用例都通过调用不同的序列（sequence）来驱动 DUT 进行验证。

2. **Environment 层**：Environment 层是验证环境的核心部分，包含了多个 Agent、Scoreboard 和其他辅助组件。在我们的验证平台中，Environment 层负责协调各个 Agent 的行为，收集和处理验证数据，并通过 Scoreboard 进行验证结果的检查。

3. **Agent 层**：Agent 层负责与 DUT 进行交互，包含驱动器（driver）、监视器（monitor）和序列器（sequencer）。在我们的验证平台中，每个 Agent 都针对 DUT 的一个特定接口进行验证，通过驱动器向 DUT 发送激励，通过监视器收集 DUT 的响应，并通过序列器生成随机化的测试序列。

4. **Component 层**：Component 层是 UVM 树的底层，包含了各种基础组件，如寄存器模型、覆盖率收集器等。在我们的验证平台中，Component 层提供了验证环境中所需的各种基础功能，支持了 Agent 和 Environment 层的实现。

UVM 树的结构如图 19所示：

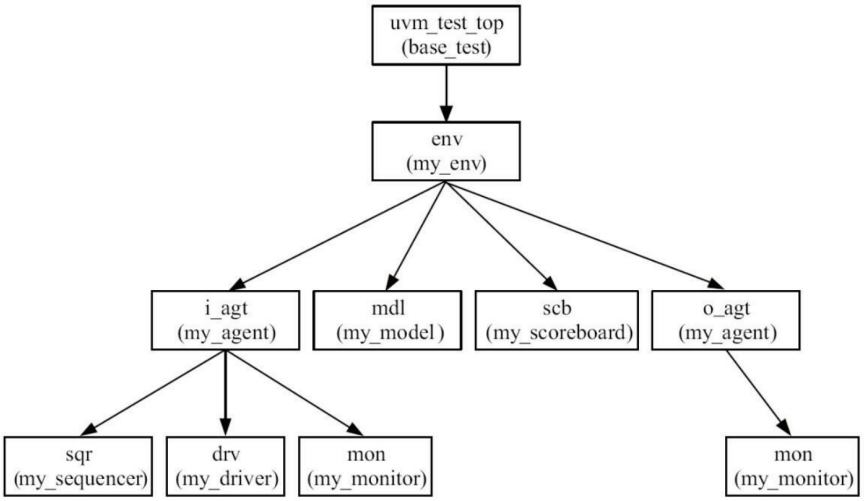


图 19: UVM 树结构图

## 5.2 factory 机制

UVM 的 Factory 机制是基于工厂设计模式的一种机制，它允许自动创建一个类的实例并调用其中的函数（function）和任务（task）。

使用 Factory 机制的第一步是将类注册到工厂。这个工厂是整个全局仿真中存在且唯一的“机构”，所有被注册的类才能使用 Factory 机制。注册通常通过使用 uvm_component_utils 宏来完成。这些宏会为类生成必要的类型信息和工厂注册代码。完成了注册之后，就可以通过 Factory 机制来创建对象实例。所有注册到 Factory 的类均可通过 Factory 独特的方式实例化对象。创建对象时，通常使用 type_id::create 方法。这个方法会调用类的 new 函数来创建实例，并且可以处理类的重载。

因此，在验证平台的移植中，我们使用该种方式替代使用 new() 的方式实现对类的实例化操作，示例如下：

```
//env_pkg.sv
    class my_env extends uvm_env;
        ‘uvm_component_utils(my_env)
    ...
    endclass

//task_pkg.sv
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        env = my_env::type_id::create(”env”, this);
    endfunction
```

## 5.3 uvm_phase 与 objection 机制

在完成 5.1 节中注册到工厂的操作后，引入 phase 机制可以实现类中 task 的自动调用。Phase 机制使得验证环境从组建到连接，再到执行，得以分阶段进行。UVM 中划分了很多个 Phase，这些 Phase 之间有着严格的层次关系，也遵循着先后执行的顺序关系。

UVM 中的 Phase 可以分为两大类：不消耗仿真时间的 Function Phase 和消耗仿真时间的 Task Phase。Function Phase 如 build_phase、connect_phase 等，用于验证环境的构建和连接；Task Phase 如 run_phase、main_phase 等，用于验证的执行。所有 Phase 都会按照自上而下的顺序自动执行。其中，run_phase 和 12 个小的 Phase 并行执行。这 12 个小的 Phase 也称为动态运行（run-time）Phase，其存在的目的是为了实现更加精细化的控制，这些 Phase 的核心是 reset、configure、main、shutdown，用来模拟 DUT 正常的工作模式。在本次 project 中，我们主要用到了 build_phase、connect_phase 与 run_phase。这里我们以 icb_driver 为例：

```
class icb_driver extends uvm_driver #(icb_trans);

    `uvm_component_utils(icb_driver);
    local virtual icb_bus.master active_channel;

    function new(string name = "icb_driver", uvm_component parent = null);
        super.new(name, parent);
    endfunction //new

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if (!uvm_config_db#(virtual icb_bus)::get(this, "", "active_channel",
            active_channel)) begin
            `uvm_fatal("NO_VIF", "Virtual interface must be set for: icb_vi")
        end
    endfunction

    task run_phase(uvm_phase phase);

        while(1) begin
            seq_item_port.get_next_item(req);
            @(this.active_channel.mst_cb)
            this.active_channel.mst_cb.icb_cmd_valid <= 1'b1;
            this.active_channel.mst_cb.icb_cmd_read <= req.read;
            this.active_channel.mst_cb.icb_cmd_wmask <= req.mask;
            this.active_channel.mst_cb.icb_cmd_wdata <= req.wdata;
            this.active_channel.mst_cb.icb_cmd_addr <= req.addr;

            // wait until the handshake finished
            while(!this.active_channel.icb_cmd_ready) begin
                @(this.active_channel.mst_cb);
            end

            // end the transaction
            this.active_channel.mst_cb.icb_cmd_valid <= 1'b0;
            seq_item_port.item_done();
        end
```

```
        endtask //run_phase

    endclass //icb_driver
```

build_phase 主要完成验证环境的构建和初始化工作，在 icb_driver 类中，验证环境运行时，模块会自动获取虚拟接口 active_channel，即驱动器与 DUT 交互的接口。而当验证环境进入 run_phase 阶段时，其他组件同样会进入这个 phase 阶段，而 icb_driver 则在这里完成驱动数据发送的任务。这里我们将其与之前的验证平台对比，可以发现明显的区别。**在之前的验证平台中，我们每次想要驱动 ICB 总线时，都必须显式地调用 single_trans() 函数，而这里，当验证环境进入 run_phase 阶段时，不再需要调用，而是会自动运行。这里使用 while(1) 循环，一旦检测到 sequencer 发送激励向量，则自动驱动 ICB 总线。这也更符合真实应用场景的行为。**

此外，为了对 phase 进行更加精细的控制，UVM 引入了 objection 机制。Objection 机制是 UVM 中控制仿真运行的关键机制。通过 raise_objection 和 drop_objection，用户可以控制仿真是否继续运行。Objection 机制的核心在于，当所有提起的 Objection 都被撤销后，UVM 会关闭当前 Phase，开始进入下一个 Phase。如果某个 Phase 没有提起任何 Objection，UVM 会直接跳过该 Phase。**这里我们只在顶层 test 类的 run_phase 中使用 raise_objection 和 drop_objection，使得当验证环境完成了我的 test 任务后，会自动结束，而在先前的验证平台中，由于 apb_agent 始终后台挂起，导致即使 test 运行完了，仿真仍需手动结束。**

## 5.4 组件间通信

在之前的验证环境中，类与类之间的通信我们采用的时 mailbox 的机制，这种机制在使用时相对较为繁琐，**而重构 UVM 验证平台时，我们采用了一种更为方便，也更容易理解的组件间通信机制 TLM（Transaction Level Modeling）**。TLM 的核心如下所示：

1. **发送**：使用 uvm_analysis_port 进行发送。在 build_phase 中实例化 uvm_analysis_port，在 main_phase 中使用 write 方法发送事务。
2. **接收**：使用 uvm_blocking_get_port 进行接收。在 build_phase 中实例化 uvm_blocking_get_port，在 main_phase 中使用 get 方法接收事务。
3. **连接**：使用 uvm_tlm_analysis_fifo 将发送端和接收端连接起来。在 connect_phase 中，将发送端的 analysis_port 连接到 FIFO 的 analysis_export，将接收端的 blocking_get_port 连接到 FIFO 的 blocking_get_export。

## 5.5 sequence 机制

UVM 的 sequence 机制是验证平台中用于生成测试激励（transactions）的关键组件。它主要由两个部分组成：uvm_sequence 和 uvm_sequencer。uvm_sequence 负责生成和控制 transaction 的发送，而 uvm_sequencer 则负责管理和仲裁多个 sequence 的执行。

Sequence 是 uvm_sequence 类的派生类，用于定义和控制 transaction 的生成和发送，同样使用 sequence 前我们需要将其注册到工厂并定义其构造函数。Sequence 的关键在于其 body 方法，我们在 body 方法中定义 transaction 的生成和发送逻辑。Sequence 的启动通过 start 方法启动，该方法需要指定一个 uvm_sequencer 实例。启动 sequence 的步骤如下：

```
seq.start(i_agt.sqr);
```

其中,i_agt.sqr 是一个 sequencer 实例，是 uvm_sequencer 类的派生类，负责管理和仲裁多个 sequence 的执行。它接收来自 sequence 的 transaction，并将它们发送给 driver。

**由于本节 sequence 测试激励的生成与验证覆盖率关系密切，因此我们将在第六章结合验证覆盖率要求对 sequence 进行说明**。

# 6 Coverage 覆盖率设计

## 6.1 功能覆盖率

基于验证计划，我们将在本章节中完成 DUT 中所有有效寄存器读写操作的功能覆盖率统计、APB 总线、ICB 总线的读写操作的功能覆盖率统计。该部分 Coverage 覆盖率设计，我们基于第五章构建的测试激励与第六章完成的 UVM 验证平台进行拓展实现。

我们基于 UVM 的 sequence 机制，完成测试向量的移植与测试顶层的搭建工作。测试工作主要分为两个重要组成部分：

1. 驱动 ICB 完成 DUT 的有效寄存器读写测试

2. 驱动 ICB，经 DUT 一主四从总线桥加密后完成对 APB 从机的读写测试

为此，我们构建了两组测试 sequence，在 test 顶层通过 start 方法依次驱动。UVM 平台同样为我们提供了更加友好的功能覆盖率统计的组件 uvm_subscriber，基于该组件，我们分别搭建了 ICB 端与 APB 端两组 coverage group。这里我们以 icb_subscriber 为例，进行简要说明：

```
covergroup icb_bus;
    option.per_instance = 1;

    cov_icb_mode: coverpoint read {
        bins read = {1'b1};
        bins write = {1'b0};
    }

    cov_icb_addr: coverpoint addr {
        bins ctrl_read = {32'h2000_0000} iff (read == 1'b1);
        bins state_read = {32'h2000_0008} iff (read == 1'b1);
        bins wdata_read = {32'h2000_0010} iff (read == 1'b1);
        bins rdata_read = {32'h2000_0018} iff (read == 1'b1);
        bins key_read = {32'h2000_0020} iff (read == 1'b1);
        bins wrong_read = {[32'h2000_0021:32'hFFFF_FFFF]} iff (read == 1'b1);

        bins ctrl_write = {32'h2000_0000} iff (read == 1'b0);
        bins state_write = {32'h2000_0008} iff (read == 1'b0);
        bins wdata_write = {32'h2000_0010} iff (read == 1'b0);
        bins rdata_write = {32'h2000_0018} iff (read == 1'b0);
```

```
        bins key_write = {32'h2000_0020} iff (read == 1'b0);
        bins wrong_write = {[32'h2000_0021:32'hFFFF_FFFF]} iff (read == 1'b0);
    }

    cov_icb_start_apb: coverpoint wdata[0] {
        bins start = {1'b1} iff ((read == 1'b0) && (addr == 32'h2000_0000));
        bins stop = {1'b0} iff ((read == 1'b0) && (addr == 32'h2000_0000));
    }
endgroup : icb_bus
```

在上述定义的 coverage group 中，我们对 ICB 总线的读写操作进行功能覆盖率统计此外，我们还对 DUT 中所有有效寄存器进行功能覆盖率统计，无论寄存器能否读写，均对其进行读写检查。

APB 总线读写操作的功能覆盖率统计较为特殊，因为 DUT 实现的一主四从总线桥功能，在单次驱动时，能且仅能驱动一个 channel 的 APB 通道，因此这里我们需要例化四组 apb_subscriber，对每个通道的 APB 总线均需要进行各自的读写覆盖率统计。

最终覆盖率统计结果图 20所示：

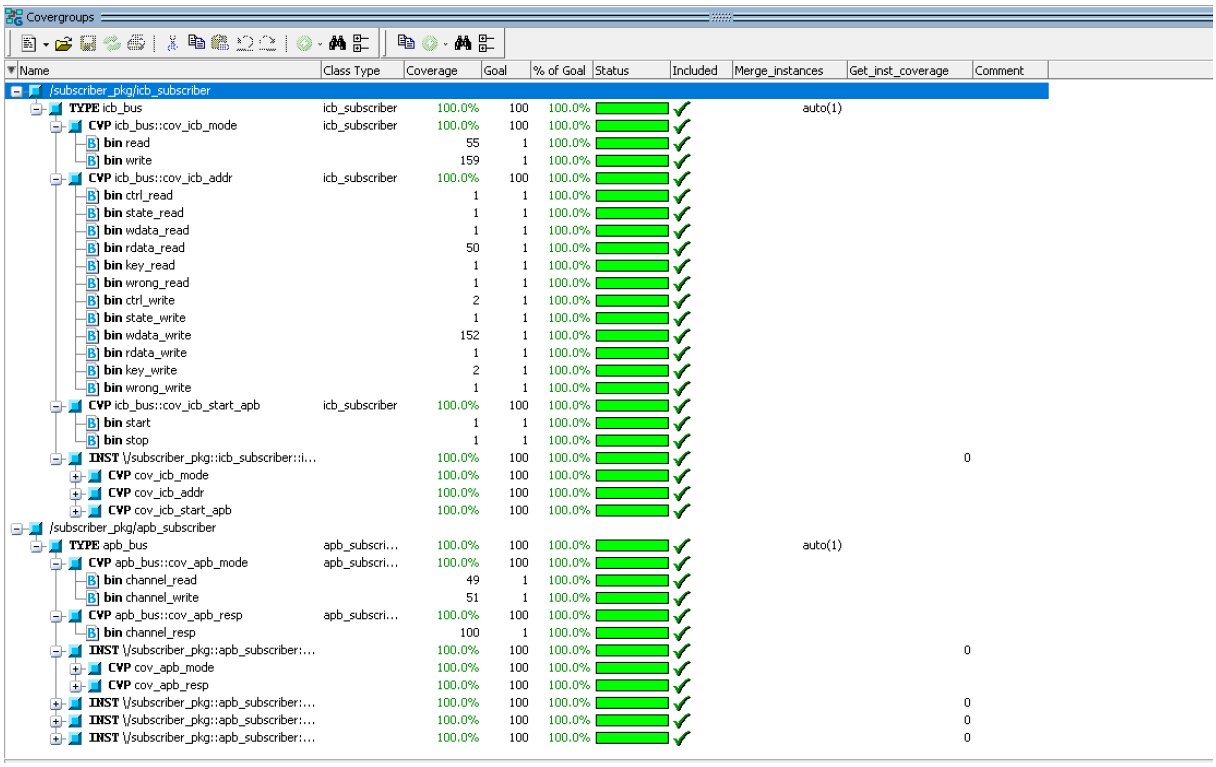


图 20: DUT 功能覆盖率统计

## 6.2 代码覆盖率

代码覆盖率统计结果图 21所示：

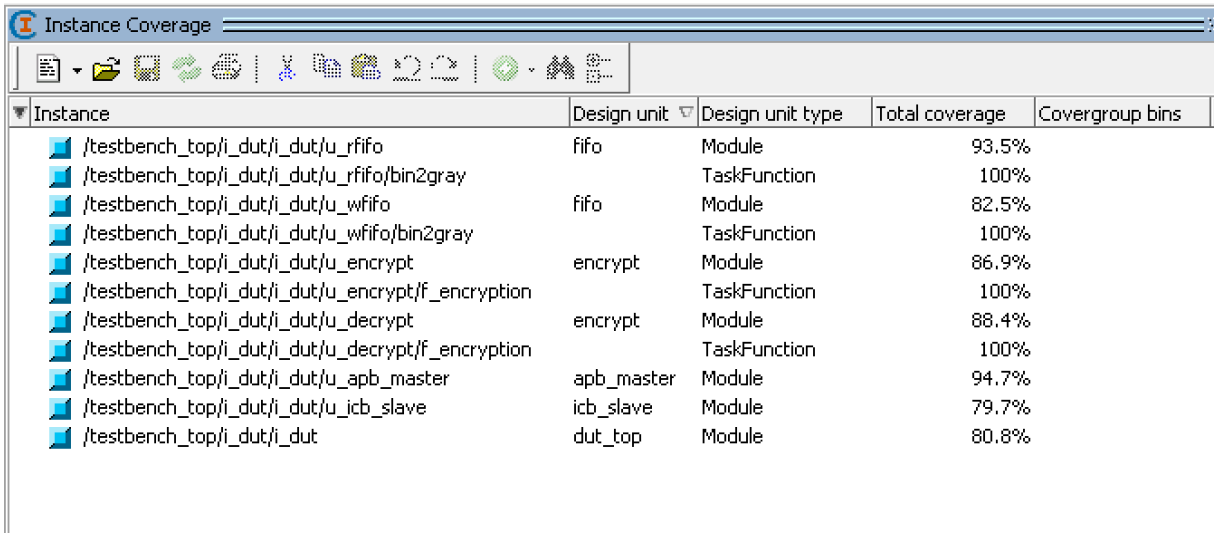


图 21: DUT 功能覆盖率统计

从图中可以看到 **DUT 的代码覆盖率情况相对较好，总体代码覆盖率接近 90%**，其中 icb_slave 与 wfifo 的代码覆盖率相对较低，我们对其进行特别分析：

我们仔细检查了 icb_slave 的每一段代码覆盖率，出现覆盖率缺失的问题段落如图 22所示：

```
108
109  : begin
110  if( icb_bus.icb_cmd_ready && !icb_bus.icb_cmd_read && icb_bus.icb_cmd_addr == ICB_SLAVE_WDATA ) begin
111      if( ~icb_bus.icb_cmd_wmask[0] ) wdata[7:0] <= icb_bus.icb_cmd_wdata[7:0];
112      if( ~icb_bus.icb_cmd_wmask[1] ) wdata[15:8] <= icb_bus.icb_cmd_wdata[15:8];
113      if( ~icb_bus.icb_cmd_wmask[2] ) wdata[23:16] <= icb_bus.icb_cmd_wdata[23:16];
114      if( ~icb_bus.icb_cmd_wmask[3] ) wdata[31:24] <= icb_bus.icb_cmd_wdata[31:24];
115      if( ~icb_bus.icb_cmd_wmask[4] ) wdata[39:32] <= icb_bus.icb_cmd_wdata[39:32];
116      if( ~icb_bus.icb_cmd_wmask[5] ) wdata[47:40] <= icb_bus.icb_cmd_wdata[47:40];
117      if( ~icb_bus.icb_cmd_wmask[6] ) wdata[55:48] <= icb_bus.icb_cmd_wdata[55:48];
118      if( ~icb_bus.icb_cmd_wmask[7] ) wdata[63:56] <= icb_bus.icb_cmd_wdata[63:56];
119  end else begin
120      wdata <= wdata;
121  end
122
123
124
125  f @(posedge clk or negedge rst_n) begin
126  ! !rst_n ) begin
127   key <= 64'b0;
128
129  : begin
130  if( icb_bus.icb_cmd_ready && !icb_bus.icb_cmd_read && icb_bus.icb_cmd_addr == ICB_SLAVE_KEY ) begin
131      if( ~icb_bus.icb_cmd_wmask[0] ) key[7:0] <= icb_bus.icb_cmd_wdata[7:0];
132      if( ~icb_bus.icb_cmd_wmask[1] ) key[15:8] <= icb_bus.icb_cmd_wdata[15:8];
133      if( ~icb_bus.icb_cmd_wmask[2] ) key[23:16] <= icb_bus.icb_cmd_wdata[23:16];
134      if( ~icb_bus.icb_cmd_wmask[3] ) key[31:24] <= icb_bus.icb_cmd_wdata[31:24];
135      if( ~icb_bus.icb_cmd_wmask[4] ) key[39:32] <= icb_bus.icb_cmd_wdata[39:32];
136      if( ~icb_bus.icb_cmd_wmask[5] ) key[47:40] <= icb_bus.icb_cmd_wdata[47:40];
137      if( ~icb_bus.icb_cmd_wmask[6] ) key[55:48] <= icb_bus.icb_cmd_wdata[55:48];
138      if( ~icb_bus.icb_cmd_wmask[7] ) key[63:56] <= icb_bus.icb_cmd_wdata[63:56];
139  end else begin
140      key <= key;
141  end
```

图 22: icb_slave 模块覆盖率问题

恍然大悟的是，由于 WDATA 写入的包必须有效问题，我们在 sequence 中设置的 mask 不可能设置为 8'hff，而排除这一段问题，代码覆盖率达到 93.5%。

而 wfifo 的问题在于由于 encrypt 的存在，我们控制序列写入的速度，导致在有效

包测试中，WDATA 始终不会写满，覆盖率略低于 rfifo，这也是合理的。

# 7 DUT 综合及分析

为了满足验证需求，我们在 ICB 与 APB 总线的 interface 定义中引入了 clocking block 模块。然而，clocking block 模块并不满足可综合设计的要求，因此在进行综合分析时，我们将其注释掉了。接下来，我们将使用 Vivado 工具对 DUT 进行综合分析，并详细报告 DUT 在 Xilinx FPGA（型号：xc7k70tfbg484-1）上的综合结果。

## 7.1 资源分析

DUT 综合的资源报告如图 23所示：

| Name | Slice LUTs (41000) | Slice Registers (82000) | Bonded IOB (285) | BUFGCTRL (32) |
|---|---|---|---|---|
| synthesis_top | 1210 | 1465 | 608 | 1 |
| dut_top (dut_top) | 1210 | 1465 | 0 | 0 |
| u_apb_master (apb_master) | 268 | 98 | 0 | 0 |
| u_decrypt (encrypt__parameterized0) | 144 | 126 | 0 | 0 |
| u_encrypt (encrypt) | 159 | 126 | 0 | 0 |
| u_icb_slave (icb_slave) | 255 | 203 | 0 | 0 |
| u_rfifo (fifo) | 242 | 600 | 0 | 0 |
| u_wfifo (fifo_0) | 142 | 312 | 0 | 0 |

图 23: DUT 综合资源报告

从资源报告中，**我们特别关注了 LUT 和 Register 的消耗量，分别为 1210 和 1465**。LUT 的消耗主要集中在 apb_master、icb_slave 以及 FIFO 模块中，而 Register 的消耗则主要发生在两个 FIFO 模块中。这种资源消耗模式是符合预期的：apb_master 和 icb_slave 模块中包含大量的逻辑判断，用于信号转换和驱动，这导致了较高的 LUT 消耗；而 FIFO 模块由于引入了格雷码，使得逻辑更为复杂，因此 LUT 的消耗量也相对较高。同时，FIFO 模块中开辟了大量的存储资源，这导致了 Register 的消耗量较大。

值得注意的是，我们观察到了一个有趣的现象：尽管 rfifo 和 wfifo 都是从 fifo 模块实例化而来，但它们的资源消耗量却几乎相差一倍。**这一现象的原因在于 ICB 和 APB 之间的数据位宽不匹配。具体来说，ICB 的数据在经过 FIFO 后，被 apb_master 读取时的数据位宽仅为 32 位，而不是写入时的 64 位**。因此，在综合过程中，综合工具经过优化处理，只缓存了 32 位宽的数据，而不是 64 位，这就导致了资源消耗量的显著差异。

这一发现表明，数据位宽的不匹配和综合工具的优化策略对资源消耗有着直接的影响。在设计过程中，我们需要仔细考虑这些因素，以实现资源的最优利用。同时，这也提醒我们在进行模块复用时，要充分考虑不同应用场景下的具体需求，以避免不必要的资源浪费。

总体而言，**由于在 DUT 设计中 DES 加解密模块采用了原地递归的方式进行运算**，因此资源消耗量相对较少，DUT 的整体资源开销小！

## 7.2 时序分析

时序分析是确保设计满足性能要求的关键步骤，它与时钟约束紧密相关。在本项目中，我们最初采用了默认的 2ns 时钟周期，这对应于 500MHz 的时钟频率。然而，如图 24所示，DUT 的综合时序报告显示存在一定的建立时间违例。建立时间违例指的是数据在时钟边沿到达之前没有足够的时间稳定，这可能导致数据错误地被采样。

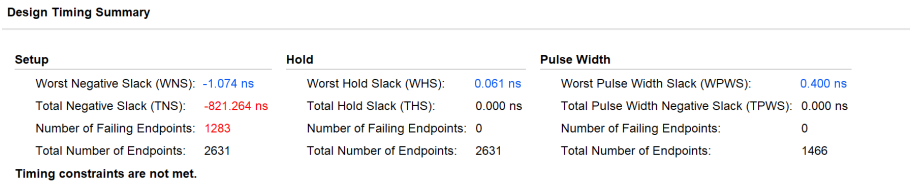


图 24: DUT 综合时序报告（500MHz）

为了解决这一问题，我们采取了降低时钟频率的策略，以提供更多的时间来满足建立时间的要求。最终，我们将时钟周期调整为 3.2ns，这对应于大约 312.5MHz 的时钟频率。通过这种调整，我们成功消除了时序违例，如图 25所示的时序报告。

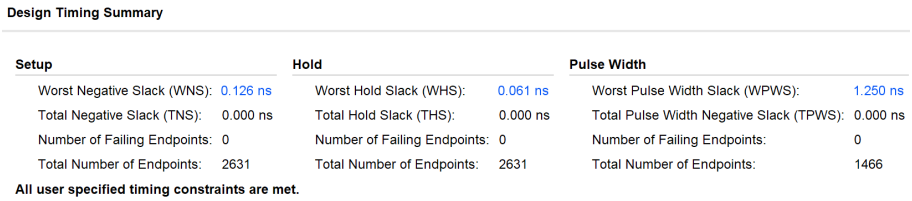


图 25: DUT 综合时序报告（312.5MHz）

总体而言，**由于在 DUT 设计中 DES 加解密模块采用了流水打拍的方式进行运算，将复杂的 DES 分解到 15 个周期完成**，因此关键路径相对较短，DUT 的时钟频率较高！

# 8 总结

在本次 Project 中（with LAB），我们完成了 DUT 的可综合硬件设计，并搭建了验证平台对其进行了完整可靠的验证工作。此外，我们还对 DUT 进行了 SVA 断言验证与 Coverage 覆盖率检测。核心关键内容完成如下：

1. **ICB_APB_CryptoBridge 一主四从加密总线桥的 rtl 实现**
2. **基于 SV 的验证平台**
3. **基于 UVM 的验证平台**
4. **SVA 验证**
5. **覆盖率检测**
6. **DUT 综合结果分析**

其他更为详细的实验内容与总结如表 1所示，这里不再列出。所有 DUT 代码与验证代码管理在：

- Github: https://github.com/WzyNoEmo/ICB_APB_CryptoBridge