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

2024 年 12 月 11 日

目录

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，在本次实验中，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。仿真平台的基本框架图如 图 1 所示。

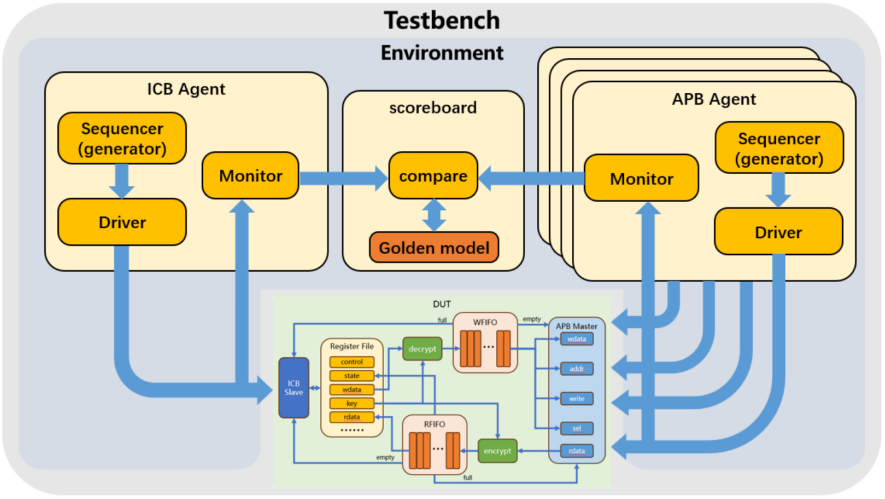

图 1: 仿真平台基本框架

基于搭建的仿真平台，对 DUT 进行了完整的功能验证，本次实验内容完成情况如 表 1 所示：

表 1: 实验内容及完成情况

要求	项目	完成度
必做	ICB 端： （1）ICB 总线时序正确性检查 （2）所有寄存器的读写测试 （3）随机地址读写测试	完成
必做	APB 端： （1）APB 总线时序正确性检查 （2）任意地址的读写测试 （3）数据包解码正确性检查 （4）4 个通道读写覆盖	完成
必做	WFIFO 与 RFIFO：配合 ICB 端和 APB 端读写完成两个 FIFO 空满状态测试覆盖	完成
必做	Encrypt/Decrypt：完成加/解密正确性检查	完成
必做	数据流：ICB 端与 APB 端数据正确传输检查	完成
必做	Interface 中加入 clocking block 来对时序进行调整	完成
选做	随机化测试： （1）测试数据，如 ICB 的 wdata、address、请求类型，APB 的 rdata 等 （2）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	完成
选做	完成一个完整的 testbench 结构： （1）搭建 monitor 对象：采集 ICB 端和 APB 端的数据 （2）搭建 golden model 对象：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 （3）搭建 scoreboard 对象（golden model 可以内嵌在这个对象里），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进行打印。	完成
选做	实现 DUT 使用 DES 进行加解密的功能并完成相关测试	完成

2 模块设计简述

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 2 所示：

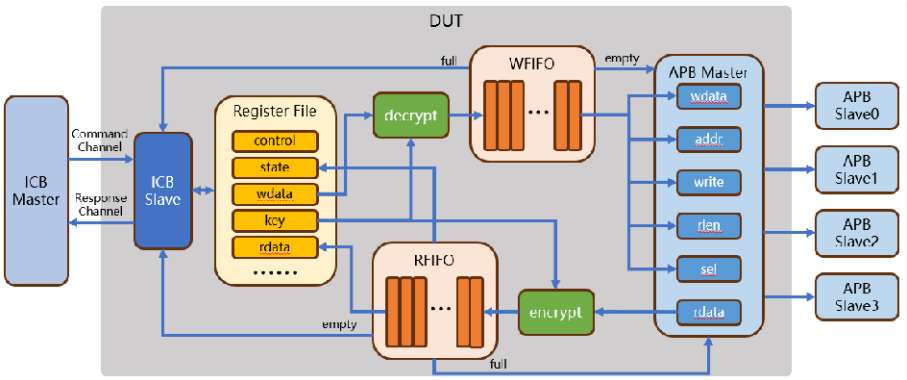

图 2: DUT 设计

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。

3 验证平台搭建

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。为了简化以上行为级验证过程，我们暂时将 DUT 加解密的 DES 模块 disable，使 scoreboard 无需对 ICB 使用 DES 加密事务包再进行一层解密才能获得初始事务包。

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 3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 000100：APB2 001000：APB3	0：读 1：写	0：控制
数据			1：数据

图 3: icb 事务包

任务 behavior_verify 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 3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 golden model 进行传输结果的正确性判断，并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进行打印。golden model 的判断逻辑如下所示：

```
if( ctrl_packet[1] == 1 ) begin
    if( apb_data.addr == {8'b0,ctrl_packet[31:8]} && apb_data.wdata ==
        {1'b0,data_packet[31:1]} ) begin
        $display("| APB Write Success ! |");
        this.pass_cnt++;
        this.total_cnt++;
    end else begin
        $display("| APB Write Failed ! |");
        this.total_cnt++;
```

```
        end
    end else begin
        if( apb_data.addr == {8'b0,ctrl_packet[31:8]} ) begin
            $display("| APB Read Success ! |");
            this.pass_cnt++;
            this.total_cnt++;
        end else begin
            $display("| APB Read Failed ! |");
            this.total_cnt++;
        end
    end

    $display("| Pass / Total : %d / %d |" , this.pass_cnt,this.total_cnt);
    $display("| Pass Rate : %f%% |", this.pass_cnt/this.total_cnt * 100);
```

3.4 仿真 env 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，在终端调试信息打印中加入时仿真时间的打印。

4 DUT 功能验证及分析

4.1 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 4所示：

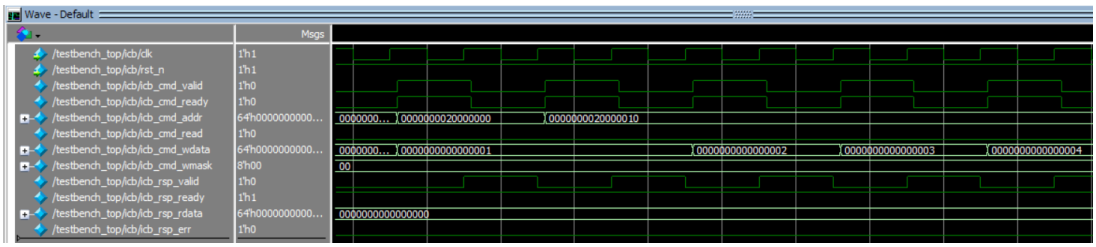

图 4: ICB 总线写时序验证

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 5所示：

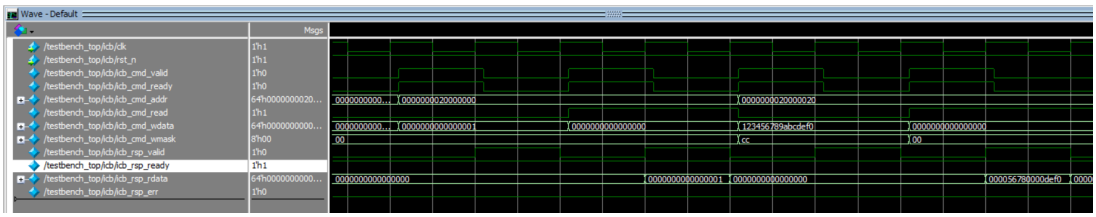

图 5: ICB 总线读写时序验证

同时我们将波形图与终端打印的调试信息图 6进行对比：

```
# ================================================================
# [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 6: RAW_TEST 终端打印信息

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！

4.2 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 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 7，图 8所示：

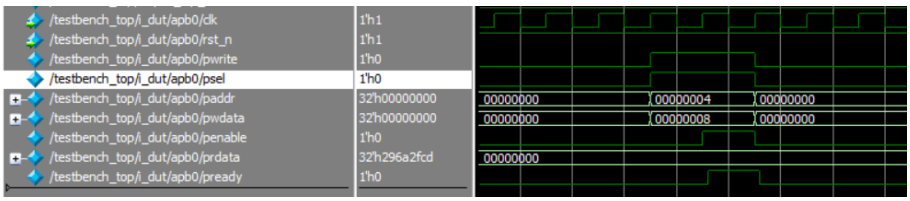

图 7: APB 写波形

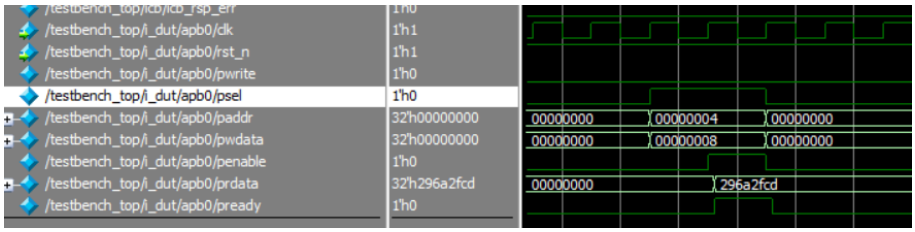

图 8: APB 读波形

4.3 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 9所示：

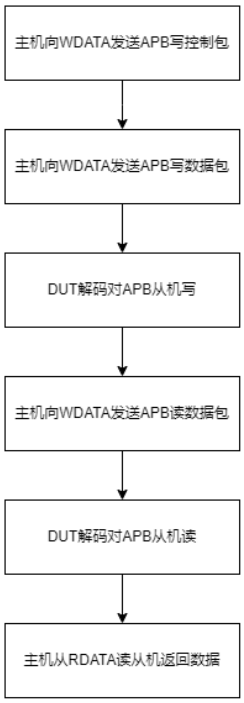

图 9: LOOPBACK 流程图

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 10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 10: 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。

4.4 DES 加解密验证

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 11所示：

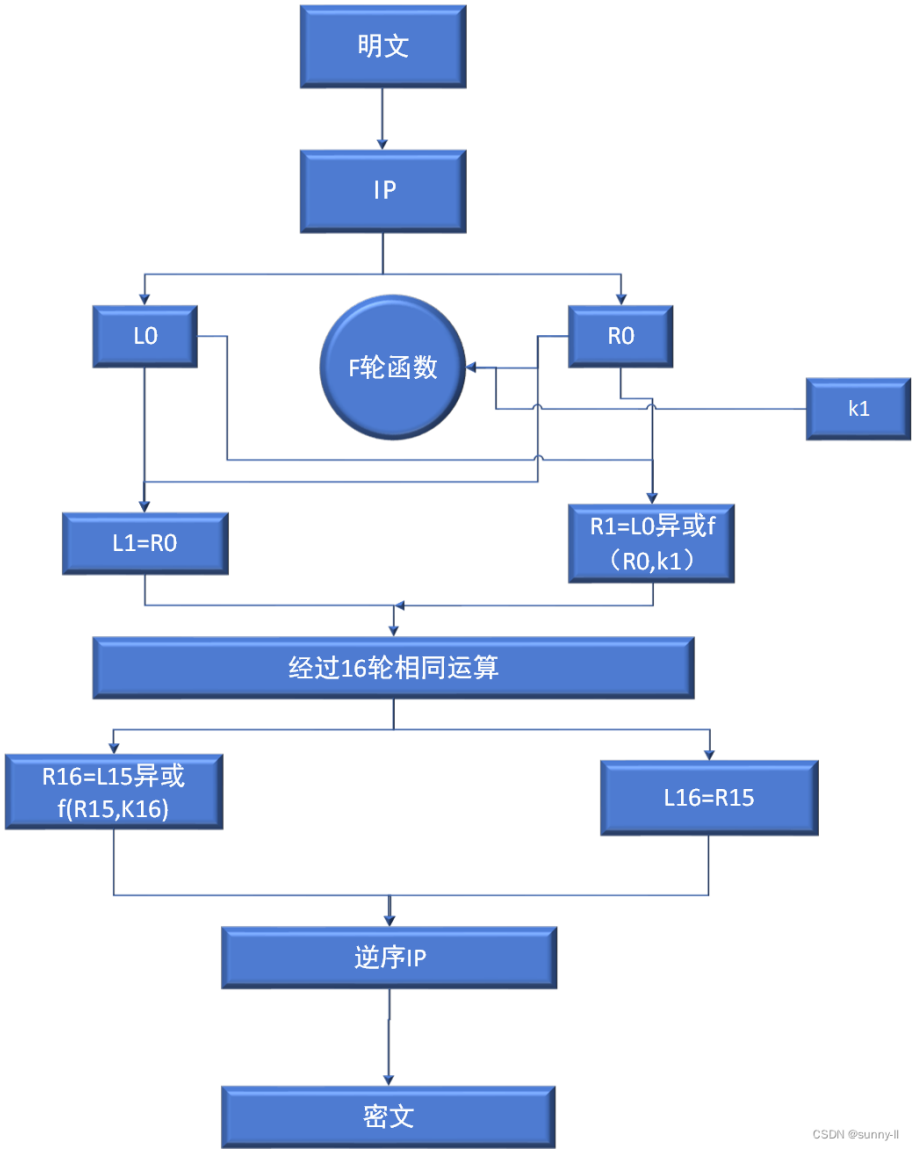

图 11: DES 算法流程图

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 12所示：

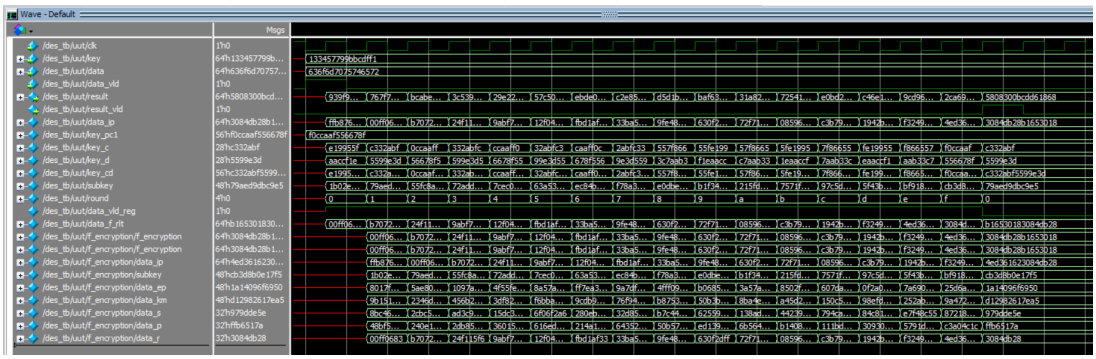

图 12: DES 测试波形图

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。

4.5 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 13所示：

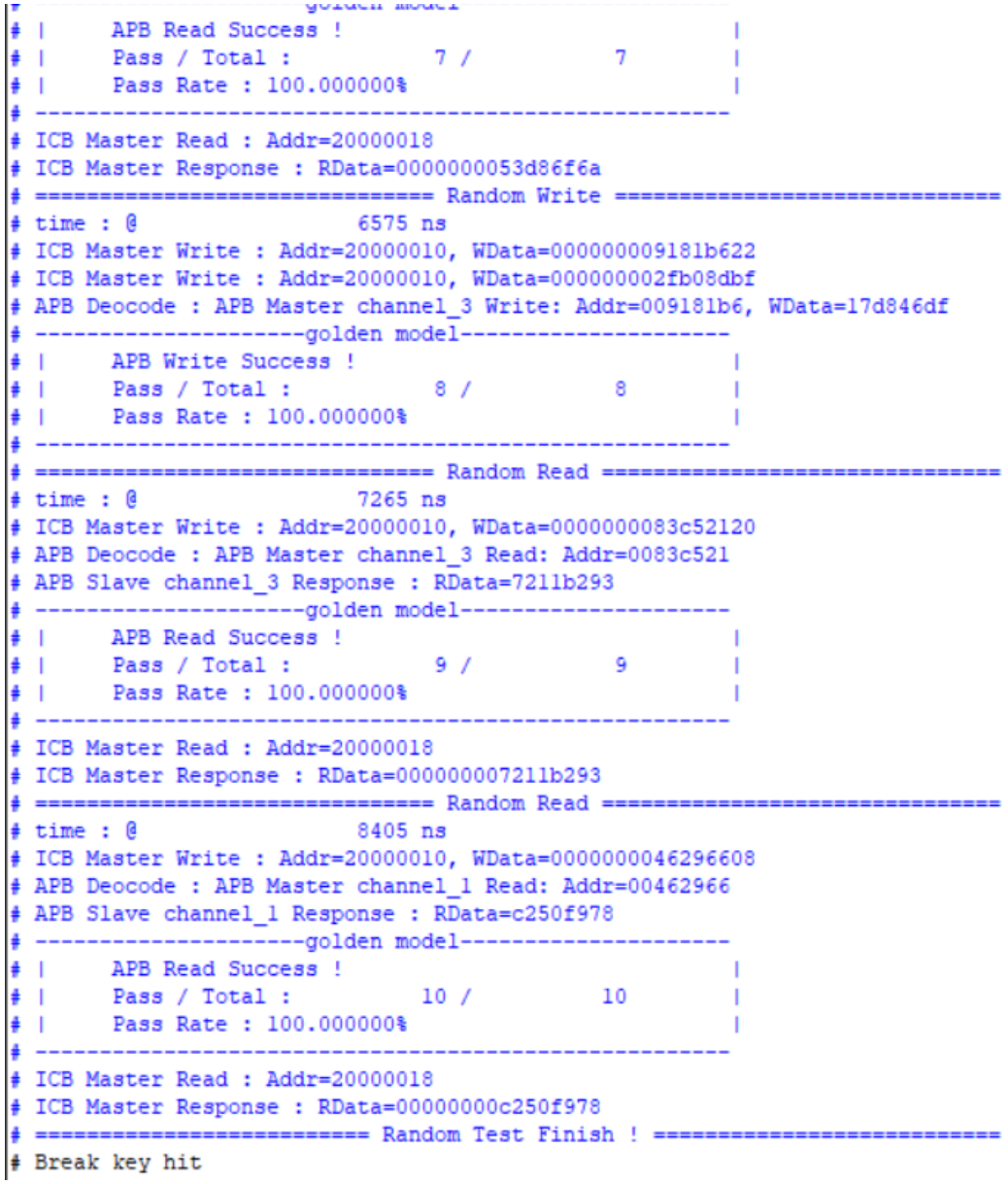

图 13: golden model 系统级验证

5 总结

本次实验完成了对 Lab 1 中 DUT 的仿真平台的搭建以及测试验证。通过在仿真平台中插入有效且合理的终端信息打印，用于监控 DUT 的行为，并且使用 Questa Sim 通过观察波形进行更为严谨的验证，证明了 DUT 设计的正确性。核心关键内容完成如下：

1. ICB 与 APB 总线的时序正确性检查
2. ICB 端寄存器的读写测试
3. APB 端 4 个通道任意地址读写测试
4. 配合 ICB 与 APB 端完成 FIFO 模块的空满测试
5. DES 解码正确性检查
6. 数据流：数据包解码的正确性检查以及 ICB 端与 APB 端数据正确传输检查
7. Interface 中加入 clocking block 来对时序进行调整
8. 随机化测试
9. golden model 与 scoreboard 搭建

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 代码与测试代码管理在：

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

6 quiz

6.1 package_usage.sv

38 行处 sun 类对象 s 并未被识别，如何改正：在 module 的开头添加:import sky_pkg::sun;

40 行处 sun state 状态并未打印，如何改正：将 apollo 的实例化放到 module 块中或在 module 的开头添加:import sky_pkg::*;

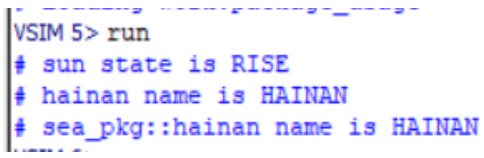

图 14: quiz1

Package 与 Class 的关系： package 可以用来封装一个或多个 class，使得相关的类和功能可以被组织在一起，便于管理和使用。package 提供了一个命名空间，避免了类名和其他标识符之间的冲突。例如，如果在不同的包中定义了同名的类，它们不会冲突。

6.2 interproces_sync.sv

6.2.1 event_use

如何触发事件： 通过-> 操作符被触发，这个操作符用于生成一个事件触发信号，可以解除所有等待该事件的进程的阻塞状态。

@e1 与 e1.triggered() 有什么区别： @e1 是边沿触发的阻塞性等待，进程会等待直到事件被触发。e1.triggered() 是非阻塞性的检查，用于查询事件是否被触发，可以用于非阻塞性等待。

```
VSIM 9> run
# b_event_use process block started
# @0, wait(e2.triggered()) finished
# @10, @e3b finished
# @10, @e3a finished
# @20, @e1 finished
```

图 15: quiz2

6.2.2 mailbox_use

如何向信箱里放入指定类型： 通过 mailbox #(typedef) mb_id 中设置 typedef 指定信箱中元素的类型。

如何使用信箱： mb.put() 将元素放入信箱，mb.get() 取出信箱中的元素。

```
VSIM 11> run
# b_mailbox_use process block started
# box handles array bx content is '{@box@1, @box@2, @box@3, @box@4, @box@5}
# extracting ID and HANDLE from the TWO mailboxes
# ID:0, HANDLE:'{id:0}
# ID:1, HANDLE:'{id:1}
# ID:2, HANDLE:'{id:2}
# ID:3, HANDLE:'{id:3}
# ID:4, HANDLE:'{id:4}
# ID mailbox size is 0
# HANDLE mailbox size is 0
```

图 16: quiz3

6.2.3 mailbox_user_define

运行并观察结果：

```
VSIM 13> run
# b_mailbox_user_define process block started
# box handles array bx content is '{@box@1, @box@2, @box@3, @box@4, @box@5}
# extracting ID and HANDLE from the ONE mailbox
# ID:0, HANDLE:'{id:0}
# ID:1, HANDLE:'{id:1}
# ID:2, HANDLE:'{id:2}
# ID:3, HANDLE:'{id:3}
# ID:4, HANDLE:'{id:4}
# PAIR mailbox size is 0
```

图 17: quiz4

6.3 thread_control.sv

```
VSIM 17> run -all
# b_fork_join process block started
# @0, fork_join_thread entered
# @0, thread id:0 entered
# @0, thread id:1 entered
# @0, thread id:2 entered
# @10, thread id:0 exited
# @20, thread id:1 exited
# @30, thread id:2 exited
# @30, fork_join_thread exited
# @30, box handles array is '{null, null, null}
# b_fork_join_any process block started
# @40, fork_join_any_thread entered
# @40, thread id:0 entered
# @40, thread id:1 entered
# @40, thread id:2 entered
# @50, thread id:0 exited
# @50, fork_join_any_thread exited
# @50, box handles array is '{null, @box@5, @box@6}
# @50, disabled fork_join_any_thread
# @150, box handles array is '{null, @box@5, @box@6}
# b_fork_join_none process block started
# @160, fork_join_none_thread entered
# @160, fork_join_none_thread exited
# @160, box handles array is '{null, null, null}
# @160, thread id:0 entered
# @160, thread id:1 entered
# @160, thread id:2 entered
# @170, thread id:0 exited
# @175, box handles array is '{null, @box@8, @box@9}
# @180, thread id:1 exited
# @190, thread id:2 exited
# @190, fork_join_none_thread's all sub-threads finished
# @190, box handles array is '{null, null, null}
```

图 18: quiz5

fork join, fork join_any, fork join_none 的区别：fork...join：所有线程必须完成，主线程才继续。fork...join_any：任何一个线程完成，主线程就继续，其他线程继续执行。fork...join_none：主线程不等待任何线程完成，直接继续执行。

disable fork 和 wait fork 的作用：wait fork 通常用于确保所有并发操作完成后再继续执行，而 disable fork 用于需要立即停止所有并发操作的场景。