

# 《密码工程基础》

## 实验报告

--密码 2101 闵翔宇 U202114221

(报告中只出现核心代码, 完整代码在压缩包里)

### 实验一:

#### 一、基本原理及计算过程

本实验侧重于 PC 与 FPGA (Field-Programmable Gate Array) 的综合应用, 使用 UART (Universal Asynchronous Receiver/Transmitter)、FIFO (First-In, First-Out) 等技术。这些技术在网络空间安全领域中非常重要, 特别是在加密算法的硬件实现中。

UART (通用异步接收/发送器): 它是用于 PC 和 FPGA 之间数据通信的关键组件。选择和配置正确的波特率对于确保数据传输的准确性和效率至关重要。这一技术允许两个硬件设备通过串行通信端口高效地交换数据。

FIFO (先进先出): FIFO 作为一个临时存储区, 用于保持数据的顺序性和完整性。它在处理不同速率的数据生产者和消费者时尤其重要, 例如, 当 FPGA 以不同的速率生成数据时, FIFO 可以确保数据在发送给 PC 之前保持有序和完整。

ILA (集成逻辑分析仪): 在 FPGA 的调试阶段, ILA 是一个重要的工具。它用于捕获和分析 FPGA 内部的数字信号, 从而可以发现和解决潜在的 bug。这对于确保 FPGA 正确执行其加密算法等功能至关重要。

#### 二、整体架构设计

设计目的:

建立一个高效、可靠的 PC-FPGA 通信系统, 能够在两者之间快速、准确地传输数据, 并能够通过模拟和分析来优化和调试系统。

设计架构:

上位机: 通过 Python 编写, 负责打开 UART 并与 FPGA 进行通信。

UART 控制器: 在 PC 和 FPGA 之间实现数据通信, 并监控 FIFO 的数据计数。

运算单元: 当使能信号被置高时, 从接收 FIFO 中获取数据, 处理后写入发送 FIFO。

### 三、主要模块仿真

仿真步骤：

Vivado 工程构建：包括添加设计文件（.v），约束文件（.xdc），以及仿真文件（tb\_Top.v）。

IP 核的应用：如 Clocking Wizard, FIFO Generator, ILA 等。

模块仿真：包括 UART、FIFO、运算单元等模块的独立仿真，以及整体系统的仿真。

输入反向仿真结果如下：



可见，输入是 0、1、2、3，输出是 3、2、1、0，完成了既定功能。

联合测试：结合 Python 脚本和 ILA 进行 PC-FPGA 的联合测试和调试。

### 四、上板联调联测

Python 进行 ttl 串口实测：

```
▷ #发送数据
data_to_send = b'\x00\x11\x22\x33\x44\x55\x66\x77\x88\x99\xaa\xbb\xcc\xdd\xee\xff'
write_len = ser.write(data_to_send)
print(data_to_send.hex())
print(write_len)
[79] ✓ 0.0s

... 00112233445566778899aabbccddeeff
16
+ 代码 + Markdown

▷ #接收数据
data_to_read = ser.readline() # 读取一行数据
print(data_to_read.hex())
[80] ✓ 1.0s

... 3322110077665544bbaa9988ffeeddcc3322110077665544bbaa9988ffeeddcc
```

可见串口输入 00112233，输出 33221100，完成了既定目标。

## 五、总结

对 FPGA 和密码工程的理解

FPGA (Field-Programmable Gate Array)：作为一种高度灵活的可编程硬件，FPGA 在密码工程中扮演着重要的角色。它允许工程师实现快速原型设计和对加密算法的高效硬件级优化，特别是在需要高安全性和快速数据处理的应用中。

密码工程：集成了计算机科学、数学和电子工程的知识，重点在于开发和分析安全通信系统。在密码工程中，FPGA 可以用来快速实现和测试复杂的加密算法。

从课程学到的内容

对硬件描述语言（如 VHDL 或 Verilog）的深入理解，这是设计和实现 FPGA 项目的基础。掌握了密码学的基本原理，包括对称和非对称加密、散列函数、数字签名等，以及它们在现代通信系统中的应用。结合了前两门课程的知识，专注于密码算法在硬件上的实现。特别是如何利用 FPGA 实现高效、安全的密码系统。

## 附录

```
STA_ROUND      : begin
    out_rcv_rd_en    <= #TCQ 1'b0;
    tmp              <= #TCQ {rcv_dout[7:0], rcv_dout[15:8], rcv_dout[23:16],
end
```

核心部分如上：

## 实验二

### 一、基本原理及计算过程

AES 是一种广泛使用的对称加密算法，被广泛应用于数据加密和保护。AES 的工作原理是通过一系列的替换和置换操作，将明文转换为密文。这个过程需要一个密钥，且加密和解密使用同一个密钥。AES 支持多种长度的密钥，通常为 128、192 或 256 位。

AES 加密的基本单位是 16 字节的数据块。这些数据块在加密过程中经过多个轮次的处理。每一轮的处理包括四个主要步骤：SubBytes（字节替换）、ShiftRows（行移位）、MixColumns（列混淆）和 AddRoundKey（轮密钥加）。

SubBytes（字节替换）：这一步使用一个称为 S-box 的查找表来替换数据块中的每个字节。S-box 是基于有限域上的逆元运算构造的，目的是为了提供非线性性。

ShiftRows（行移位）：在这一步中，数据块中的每一行将会进行位移操作。例如，第一行不移位，第二行向左移动一个位置，依此类推。这一步的目的是为了提高数据块中字节之间的依赖性。

MixColumns（列混淆）：这一步将每一列视为一个多项式，然后使用特定的多项式运算进行混淆。这一步的目的是为了进一步增加数据块中字节之间的复杂性。

AddRoundKey（轮密钥加）：这一步是将每一轮的密钥与数据块进行 XOR 操作。轮密钥是从原始密钥通过密钥扩展算法产生的。

每个步骤的目的都是为了增加加密的复杂度和安全性。整个加密过程包括多个这样的轮次，其中 128 位密钥的 AES 有 10 轮，192 位的有 12 轮，256 位的有 14 轮。

### 二、整体架构设计

#### 1. 工程建立与模块化设计

建立工程：首先在 Verilog 环境中创建一个新工程。

添加文件：包括必要的源代码文件、库文件和配置文件。

模块化设计：将整个 AES 算法分解成多个模块，如控制单元、SubBytes 模块、ShiftRows 模块、MixColumns 模块、AddRoundKey 模块等。

## 2. IP 核的例化与集成

在 IP Catalog 中例化：选择并配置所需的 IP 核，如 ILA（Integrated Logic Analyzer）核、UART（通用异步接收/发送）核等。

集成不同模块：将 AES 的各个组件（如 FIFO 缓冲、UART 通信模块、加密核心模块等）集成到一个统一的架构中。

## 3. 仿真与调试

单独模块仿真：对每个模块进行单独仿真，以确保它们的独立功能正确无误。

整体架构仿真：在单个模块通过仿真测试后，进行整体架构的仿真，以确保所有组件在一起工作时的正确性和性能。

## 三、主要模块仿真

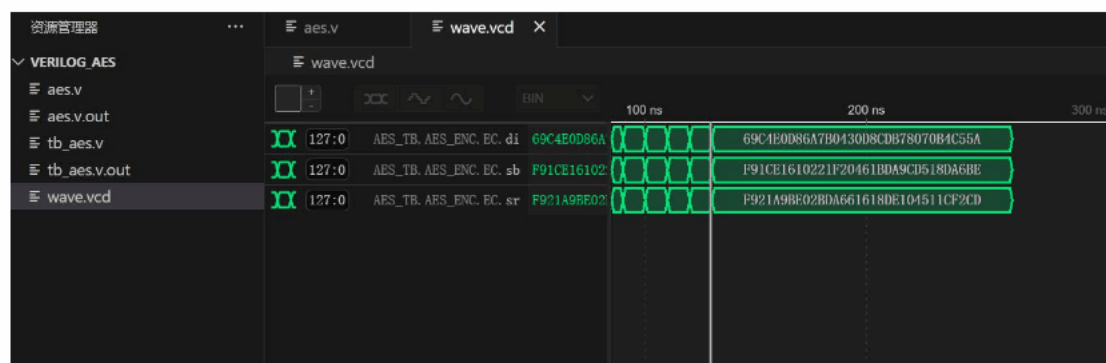
InstComputation（计算模块）仿真：

对 AES 的 SubBytes、ShiftRows、MixColumns 和 AddRoundKey 操作进行验证。

使用测试向量检查加密和解密的正确性。

下面对行变换进行仿真：

（我先进行了 s 盒代换，然后行移位，接着列混合）



其中，di 是输入，sb 是 s 盒代换之后的结果，sr 是行变换之后的结果，可见行变换正确。

## 四、上板联调联测

```
data_to_send = bytes.fromhex('69C4E0D86A7B0430D8CDB78070B4C55A')
write_len = ser.write(data_to_send)
print(data_to_send.hex())

[3] ✓ 0.0s Python
... 69C4E0D86A7B0430D8CDB78070B4C55A

#接收数据
data_to_read = ser.readline() # 读取一行数据
print(data_to_read.hex())

[4] ✓ 1.0s Python
... F921A9BE02BDA661618DE104511CF2CD
```

可见串口输入 69C4E0D86A7B0430D8CDB78070B4C55A，输出 F921A9BE02BDA661618DE104511CF2CD，完成了既定目标。

## 五、总结

这次实验不仅提高了对 AES 算法的理解，还提升了在实际应用中设计、仿真和调试复杂系统的能力。

实验过程中遇到的困难和挑战，如调试中的问题、理解复杂的硬件逻辑等，都是宝贵的学习经验。

实验也展示了理论与实践相结合的重要性。理论知识是基础，但将其应用于实践中，解决实际问题，才能真正掌握和深入理解这些知识。

## 附录

核心部分如下：

```
SubBytes SB3 (di[127:96], sb[127:96]);
SubBytes SB2 (di[ 95:64], sb[ 95:64]);
SubBytes SB1 (di[ 63:32], sb[ 63:32]);
SubBytes SB0 (di[ 31: 0], sb[ 31: 0]);

assign sr = {sb[127:120], sb[ 87: 80], sb[ 47: 40], sb[ 7: 0],
             sb[ 95: 88], sb[ 55: 48], sb[ 15: 8], sb[103: 96],
             sb[ 63: 56], sb[ 23: 16], sb[111:104], sb[ 71: 64],
             sb[ 31: 24], sb[119:112], sb[ 79: 72], sb[ 39: 32]};

MixColumns MX3 (sr[127:96], mx[127:96]);
MixColumns MX2 (sr[ 95:64], mx[ 95:64]);
MixColumns MX1 (sr[ 63:32], mx[ 63:32]);
MixColumns MX0 (sr[ 31: 0], mx[ 31: 0]);
```

## 实验三

### 一、基本原理及计算过程

#### 一、基本原理

SHA-2（安全散列算法 2）是一种密码散列函数集合，广泛用于网络安全和数据完整性校验中。它将输入数据（如文本或二进制文件）转换成固定长度的散列值，这个过程是单向且不可逆的，确保了数据的唯一性和安全性。SHA-2 相比于其前身 SHA-1，提供了更高的安全性。

#### 二、计算过程

数据预处理：将原始数据进行填充，确保数据长度符合处理要求。通常包括填充一些特定的位和长度字段。

初始化哈希缓冲区：设置初始哈希值，这些值是算法的一部分，通常是固定的。

数据分块处理：将预处理后的数据分为固定大小的块（例如 512 位或 1024 位），每个块单独处理。

循环压缩：对每个数据块，执行一系列复杂的函数和操作，包括位运算（如与、或、异或、非）和模运算。这一步骤涉及到多个循环和子函数。

输出最终的哈希值：经过多轮循环压缩后，合并所有数据块的处理结果，得到最终的哈希值。

### 二、整体架构设计

项目建立与模块化设计：

工程建立：首先建立整个项目的工程框架，包括所有必要的文件和模块。

模块化：整个系统被划分为多个模块，例如 InstClkWiz（时钟向导模块）、InstUartTx（UART 发送模块）、InstUartRx（UART 接收模块）、InstRcvFifo（接收 FIFO 模块）、InstSndFifo（发送 FIFO 模块）和 InstComputation（计算模块）。

模块间的接口与协调：设计模块间的接口，确保数据能够在各个模块之间正确传输。同时，协调各模块之间的工作，以确保整个系统的流畅运行。

### 三、主要模块仿真



InstComputation 仿真：

目的：核心模块，执行 SHA-2 算法的计算。

方法：

编写针对 SHA-2 算法的测试台，包括数据准备和预期散列值。

模拟数据输入，观察和验证输出散列值是否与预期一致。

特别注意测试不同的数据边界情况和异常情况。

测试台（Testbench）编写：

每个模块都需要一个测试台来模拟实际运行环境。

测试台应包括模拟输入信号、监视输出信号、并能检测和报告任何不符合预期的行为。

波形观察与分析：

使用仿真软件观察各个模块的波形输出。

通过波形分析，可以更直观地理解模块间的交互以及数据流动。

迭代和调整：

如果在仿真中发现问题，需要对 Verilog 代码进行调整，并重新仿真。

这个过程可能需要多次迭代，直到每个模块都能正确地执行其功能。

核心部分如下：

```
always @(posedge clk)
begin
    if(index>0 && index <65)
    begin
        sum1 = {e[5:0],e[31:6]}^e[10:0],e[31:11]}^e[24:0],e[31:25]};
        sum0 = {a[1:0],a[31:2]}^a[12:0],a[31:13]}^a[21:0],a[31:22]};
        Ch = (e&f)^((~e)&g);
        Maj = (a&b)^(b&c)^(c&a);
        T1 = h + sum1 + Ch +K[index-1] + W[index-1];
        T2 = sum0 + Maj;
        h <= g;
        g <= f;
        f <= e;
        e <= d + h + sum1 + Ch + K[index-1]+ W[index-1];
        d <= c;
        c <= b;
        b <= a;
        a <= T1 + T2;
```

下面是核心部分各个阶段结果的仿真结果：



my_file.txt				170 s	180 s	190 s	200 s	210 s	220 s	230 s	240 s	250 s	260 s	270 s	280 s
sha_engine.v	31:0	sha256.UUT_Eng.sum0	FE721699	FDCE308D	0F962422	46773400	FE721699	26941875	7236809E						
sha_engine.v.out	31:0	sha256.UUT_Eng.sum1	60CE3C2F	5101B536	F063C823	140F07FB	60CE3C2F	95F3BA71	1E7F94AB						
sha256.v	31:0	tb_sha256.UUT_Eng.Ch	EAB18396	DA1F6C6	DAE44689	2FD0C8A8	EAB18396	D330E29F	F38E067F						
tb_sha256.v.out	31:0	tb_sha256.UUT_Eng.Maj	A267473C	300F9517	0B47059F	25675798	A267473C	62631F75	200257B5						
tb_sha256.v	31:0	tb_sha256.UUT_Eng.T1	C066E1B2	6E0670FA	798521AF	365138C8	C066E1B2	8F082FCB	01346D10						
tb_sha256.v.out	31:0	tb_sha256.UUT_Eng.T2	A00932F5	360000A4	CC00F9C1	6B098C75	A00932F5	88F737EA	92390553						
wave.vcd															

最后 a 到 h 的结果如下：

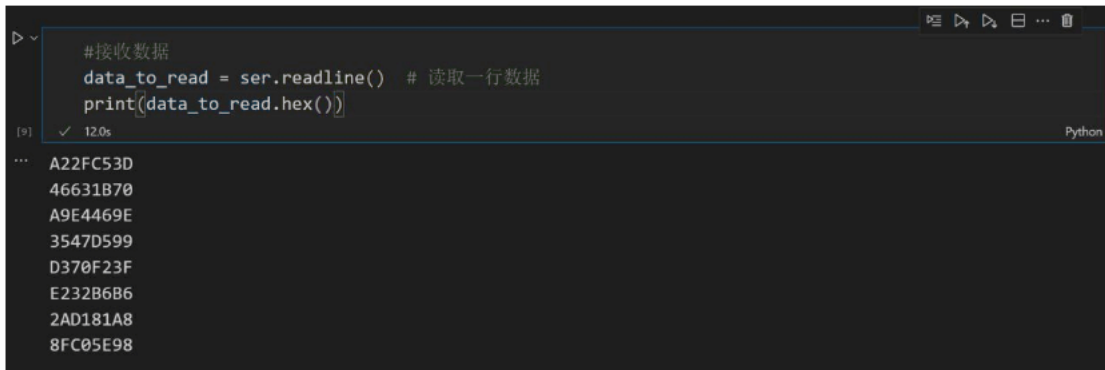
31:0	tb_sha256.UUT_Eng.a	A22FC53D	A9E4469E	46631B70	A22FC53D	61403FC7
31:0	tb_sha256.UUT_Eng.b	46631B70	3547D599	A9E4469E	46631B70	A22FC53D
31:0	tb_sha256.UUT_Eng.c	A9E4469E	9D1FB977	3547D599	A9E4469E	46631B70
31:0	tb_sha256.UUT_Eng.d	3547D599	68AD9507	9D1FB977	3547D599	A9E4469E
31:0	tb_sha256.UUT_Eng.e	D370F23F	2AD181A8	E232B6B6	D370F23F	F5AEB76B
31:0	tb_sha256.UUT_Eng.f	E232B6B6	8FC05E98	2AD181A8	E232B6B6	D370F23F
31:0	tb_sha256.UUT_Eng.g	2AD181A8	DA35C6A1	8FC05E98	2AD181A8	E232B6B6
31:0	tb_sha256.UUT_Eng.h	8FC05E98	DD9F8A5E	DA35C6A1	8FC05E98	2AD181A8

根据 sha2 的计算过程，我们截取了所有的核中间计算结果，如 Ch，Maj，sum0，sum1，T1，T2 等等，并仿真了最后的 a 到 h 的结果，与 python 脚本比对，发现完全正确。

可见已经完成了实验目标。

## 四、上板联调联测

可见串口输出 A22FC53D 46631B70 A9E4469E 3547D599 D370F23F E232B6B6  
2AD181A8 8FC05E98，完成了既定目标。



```
#接收数据
data_to_read = ser.readline() # 读取一行数据
print(data_to_read.hex())

... A22FC53D
46631B70
A9E4469E
3547D599
D370F23F
E232B6B6
2AD181A8
8FC05E98
```

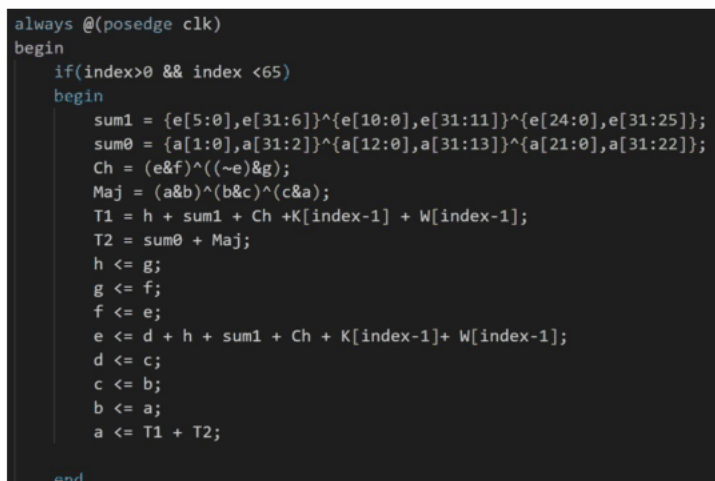
## 五、总结

我们通过一系列详细的步骤探索了 SHA-2 算法的实现和仿真。从项目的初步设立，到模块化设计，再到各个关键模块的独立仿真，整个过程不仅加深了对 SHA-2 算法的理解，也锻炼了使用 Verilog 进行硬件编程和仿真的技能。

最终，通过这个实验，不仅强调了理论知识与实践技能的结合，还突出了在现代网络安全领域中，硬件级别安全措施的重要性。整个过程不仅是对学生技能的考验，也是对他们解决复杂问题能力的培养，为他们未来在网络空间安全领域的职业生涯打下了坚实的基础。

## 附录

核心部分如下：



```
always @(posedge clk)
begin
    if(index>0 && index <65)
    begin
        sum1 = {e[5:0],e[31:6]}^{e[10:0],e[31:11]}^{e[24:0],e[31:25]};
        sum0 = {a[1:0],a[31:2]}^{a[12:0],a[31:13]}^{a[21:0],a[31:22]};
        Ch = (e&f)^((~e)&g);
        Maj = (a&b)^(b&c)^(c&a);
        T1 = h + sum1 + Ch +K[index-1] + W[index-1];
        T2 = sum0 + Maj;
        h <= g;
        g <= f;
        f <= e;
        e <= d + h + sum1 + Ch + K[index-1]+ W[index-1];
        d <= c;
        c <= b;
        b <= a;
        a <= T1 + T2;
    end
end
```

## 实验四

### 一、基本原理及计算过程

蒙哥马利模乘是一种在模数运算中广泛使用的方法，特别是在公钥密码学领域，它提供了一种高效率的大数模乘算法。该方法的基本原理是将模乘运算转换为等价的、但计算效率更高的运算。蒙哥马利模乘的核心思想在于避免直接进行模除运算，而是通过一系列的乘法和移位操作来实现。

设有两个大整数  $A$  和  $B$ ，以及一个模数  $N$ ，蒙哥马利模乘旨在计算  $AB \bmod N$  的结果。此外，引入一个与  $N$  互质的数  $R$ ，通常取  $R$  为 2 的幂次，使得  $R$  大于  $N$ 。这样选择  $R$  的目的是为了便于计算机处理，因为 2 的幂次可以简化为位移操作。计算  $AB \bmod N$  的蒙哥马利模乘过程可以分为以下几个步骤：

预处理：计算  $A$  和  $B$  的蒙哥马利表示，即计算  $A' = AR \bmod N$  和  $B' = BR \bmod N$ 。这一步骤是为了将原始数据转换到蒙哥马利域。

蒙哥马利模乘运算：这是核心步骤，计算  $C' = A'B' \bmod N$ 。在这个过程中，利用了  $R$  和  $N$  的特性，通过一系列的乘法、加法和位移操作来代替传统的模除运算。

后处理：将  $C'$  转换回常规表示，即计算  $C = C'R^{-1} \bmod N$ 。这一步使用蒙哥马利域到普通域的转换算法。

蒙哥马利模乘的优势在于它将模除运算转换为更为简单的乘法和位移操作，这对于大数运算尤其有效。在公钥密码学中，尤其是 RSA 加密算法中，蒙哥马利模乘被广泛应用于加密和解密过程，大大提高了计算效率。

### 二、整体架构设计

蒙哥马利模乘是一种在计算机系统和密码学中常用的算法，特别是在模数乘法运算中。它的整体架构设计主要关注于优化大数运算的效率，特别是在模数乘法中，这对于诸如 RSA 加密算法等场景非常关键。

在设计蒙哥马利模乘的整体架构时，核心思想是通过将乘法和取模运算转换为更加适合计算机处理的形式，从而减少传统模乘运算中的计算复杂度。这是通过使用蒙哥马利约减来实现的，它允许模乘运算以不涉及显式模除的方式进行。在实际操作中，这意味着所有的乘法操作都转换成位移和加法操作，这些操作对

计算机而言更为高效。

蒙哥马利模乘的架构通常包括预处理阶段，这一阶段涉及到将正常的数字转换成蒙哥马利形式的数字。接下来，实际的乘法操作以这种转换后的形式进行，最后，再将结果转换回常规的数字形式。这种方法避免了直接的模除操作，从而大大提高了运算效率。

为了实现这一架构，通常需要设计一个高效的数据路径，以处理大量的位移和加法操作，并确保这些操作可以快速且准确地执行。此外，对于用于这些运算的数字的存储和管理也是架构设计中的一个重要方面，需要确保数据在处理过程中的安全和完整性。

总体而言，蒙哥马利模乘的整体架构设计旨在优化大数模乘运算，通过智能的算法设计和高效的计算策略，以满足现代密码学中对速度和安全性双重需求。

### 三、主要模块仿真

输入输出参数如下:

[illegible]

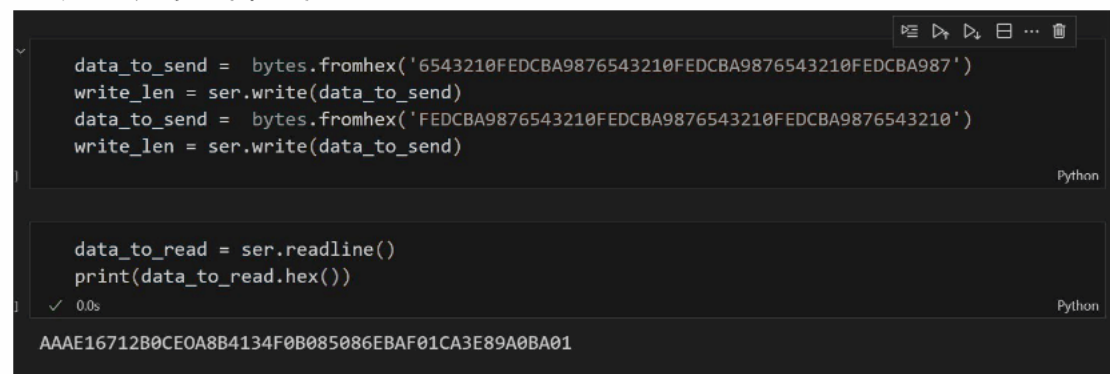
下面是核心部分各个阶段结果的仿真结果:



输入是  $x$  和  $y$ ，参数是之前模块图的参数，输出是  $z$ 。

经过和 python 验证，发现  $x*y \pmod p$  满足设计目标。

#### 四、上板联调联测



```
data_to_send = bytes.fromhex('6543210FEDCBA9876543210FEDCBA9876543210FEDCBA987')
write_len = ser.write(data_to_send)
data_to_send = bytes.fromhex('FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210')
write_len = ser.write(data_to_send)

data_to_read = ser.readline()
print(data_to_read.hex())
```

✓ 0.0s

AAAE16712B0CE0A8B4134F0B085086EBAF01CA3E89A0BA01

可见串口输出 `AAAE16712B0CE0A8B4134F0B085086EBAF01CA3E89A0BA01`，完成了既定目标。

#### 五、总结

蒙哥马利模乘是一种在现代计算机算法中广泛使用的技术，特别是在大数运算和密码学领域。这种方法的核心优势在于它能有效地进行模数乘法运算，特别是当涉及到大数时。在密码学中，如 RSA 算法，蒙哥马利模乘被用来加速模数幂运算，这是公钥加密和数字签名的关键部分。

传统的模乘运算在处理大数时效率较低，因为它涉及到重复的乘法和除法操作。蒙哥马利模乘通过变换操作来避免直接的模数除法，这样不仅提高了计算速度，也简化了硬件实现。它的主要思想是将乘法和模数约减结合起来，进行一次操作，从而避免了繁重的中间步骤。

在实际应用中，蒙哥马利模乘对加密算法的性能提升尤为显著。它通过减少必要的处理步骤，使得在资源受限的环境（如智能卡和嵌入式系统）中的加密操作变得更加高效。此外，蒙哥马利模乘也提高了算法对侧信道攻击的抵抗力，因为它使得操作时间更加一致，更难被外部监测和分析。

## 附录

代码如下:

```
timescale 1ns / 1ps

module mont_mult_modif(
    x, y,
    clk, reset, start,
    z,
    done1
);

parameter m = 192'hffffffffffffffffffffffffffe00000000000000000000000000000000000000000000000000000;
k = 192, logk = 8, zero = { logk {1'b0}},
minus_m = {1'b0, 192'h0000000000000000000000000000000000000000000000000000000000000000},
delay = 8'b01100000, COUNT = 8'b10111111; //(k-1,logk)

parameter S0 = 3'd0, S1 = 3'd1, S2 = 3'd2, S3 = 3'd3, S4 = 3'd4;

input [k-1:0] x;
input [k-1:0] y;
input clk, reset, start;
output [k-1:0] z;
output done1;

reg [logk-1:0] count;
reg [logk-1:0] timer_state;
reg [k:0] pc, psa;
reg [k-1:0] int_x;
wire equal_zero, time_out;
wire [k:0] y_by_xi, half_ac, half_as, half_bc, half_bs, next_pc, next_psa, p, p_minus_m;
wire [k+1:0] ac, as, bc, bs, long_m;
wire xi;
reg load, ce_p, load_timer;
reg [2:0] current_state;
reg [2:0] next_state;
reg done;

assign done1 = done;
genvar i;
generate for(i=0;i<k;i=i+1)
begin:and_gates
    and a(y_by_xi[i],y[i],xi);
end
endgenerate
```

```

assign y_by_xi[k] = 1'b0;
generate for(i=0;i<=k;i=i+1)
    begin:first_csa
        xor x(as[i],pc[i],psa[i],y_by_xi[i]);
        wire w1,w2,w3;
        and a1(w1,pc[i],psa[i]);
        and a2(w2,pc[i],y_by_xi[i]);
        and a3(w3,psa[i],y_by_xi[i]);
        or o(ac[i+1],w1,w2,w3);
    end
endgenerate
assign ac[0] = 1'b0, as[k+1] = 1'b0, long_m = {{2'b00},m};
generate for(i=0;i<=k;i=i+1)
    begin:second_csa
        xor x(bs[i],ac[i],as[i],long_m[i]);
        wire w1,w2,w3;
        and a1(w1,ac[i],as[i]);
        and a2(w2,ac[i],long_m[i]);
        and a3(w3,as[i],long_m[i]);
        or o(bc[i+1],w1,w2,w3);
    end
endgenerate

```

```

assign bc[0] = 1'b0, bs[k+1] = ac[k+1],
    half_as = as[k+1:1], half_ac = ac[k+1:1],
    half_bs = bs[k+1:1], half_bc = bc[k+1:1];

```

```

assign next_pc = (as[0]==1'b0)? half_ac:half_bc;
assign next_psa = (as[0]==1'b0)? half_as:half_bs;

```

```

always@(posedge(clk))
    begin:parallel_register
        if (load==1'b1) begin
            pc = { k+1 {1'b0} }; psa = { k+1 {1'b0} }; end
        else if (ce_p==1'b1) begin
            pc = next_pc; psa = next_psa; end
        end
    end
assign equal_zero = (count==zero)? 1'b1:1'b0;
assign p = psa + pc, p_minus_m = p + minus_m;
assign z = (p_minus_m[k]==1'b0)? p[k-1:0]:p_minus_m[k-1:0];

```

```

always@(posedge(clk))
    begin:shift_register

```



```

    integer i;
    if (load==1'b1) int_x = x;
    else if (ce_p==1'b1) begin
        for(i=0;i<=k-2;i=i+1) int_x[i] = int_x[i+1];
        int_x[k-1] = 1'b0; end
    end
assign xi = int_x[0];

```

```

always@(posedge(clk))
begin:counter
    if (load==1'b1) count <= COUNT;
    else if (ce_p==1'b1) count <= count - 1'b1;
end

always@(clk, current_state) begin
    case(current_state)
        S0: begin ce_p = 1'b0; load = 1'b0; load_timer = 1'b1; done = 1'b1; end
        S1: begin ce_p = 1'b0; load = 1'b0; load_timer = 1'b1; done = 1'b1; end
        S2: begin ce_p = 1'b0; load = 1'b1; load_timer = 1'b1; done = 1'b0; end
        S3: begin ce_p = 1'b1; load = 1'b0; load_timer = 1'b1; done = 1'b0; end
        S4: begin ce_p = 1'b0; load = 1'b0; load_timer = 1'b0; done = 1'b0; end
        default: begin ce_p = 1'b0; load = 1'b0; load_timer = 1'b1; done = 1'b1;
end

    endcase
end

always@(posedge clk) begin
    if(reset) current_state = S0;
    else current_state = next_state;
end

always@(*) begin
    next_state = current_state;
    if (reset==1'b1) next_state = S0;
    else if (clk==1'b1) begin
        case(next_state)
            S0: if(start==1'b0) next_state = S1;
            S1: if(start==1'b1) next_state = S2;
            S2: next_state = S3;
            S3: if(equal_zero==1'b1) next_state = S4;
            S4: if(time_out==1'b1) next_state = S0;
            default: next_state = S0;
        endcase
    end
end

```

```
end
```

```
always@(posedge clk)
```

```
begin:timer
```

```
if (clk==1'b1) begin
```

```
if (load_timer==1'b1) timer_state = delay;
```

```
else timer_state = timer_state - 1'b1;
```

```
end
```

```
end
```

```
assign time_out = (timer_state==zero)? 1'b1:1'b0;
```

```
endmodule
```