RSA-256

摘要

本專案旨在實現一個 256-bit 的 RSA 硬體加解密器，並在 PYNQ-Z2 開發板上進行測試和驗證。RSA 演算法是現代加密技術的重要基石，其安全性依賴於大數的質因數分解難題。為了提高計算效率，我們設計並實現了一個三層 256-bit 的跳過進位加法器(carry skip adder)，其中每層包含 8-bit 的漣波進位加法器（ripple carry adder)。此設計在 PYNQ-Z2 FPGA 上佔用 3592 個查找表(LUT)與 1885 個正反器(FF)，運行頻率為 18.52 MHz，板上總功耗為 11mW。此外，所提出的 RSA 電路設計以位元方式模組化，確保了設計的靈活性和可擴展性。此篇會對 RSA 演算法進行了詳細的數學分析，並推導出其硬體實現方案。接著，設計了加密和解密過程中的關鍵模塊，包括模數運算、密鑰生成、以及加法器的優化設計。通過前模擬和後模擬驗證，並確保了設計的正確性和穩定性。

RSA 加密系統 (RSA Cryptosystem)

1. RSA 演算法概述

RSA（Rivest-Shamir-Adleman）是目前廣泛應用的一種非對稱加密演算法，由 Ron Rivest、Adi Shamir 和 Leonard Adleman 於 1977 年發明。非對稱加密演算法使用一對密鑰：公開密鑰（public key）和私有密鑰（private key）。公開密鑰用於加密數據，而私有密鑰則用於解密數據。這種密鑰分離的特性使 RSA 特別適合於需要高安全性的場景，如數據加密、數字簽名等。

2. 公鑰和私鑰生成

RSA 演算法的安全性基於大數質因數分解的困難性。公鑰和私鑰的生成過程如下：

I. 質數選擇: 選擇兩個大質數 p 和 q。

II. 模數計算: 計算模數n，其中 $n = p \times q$。

III. 歐拉函數計算: 計算 $\varphi(n)$，其中$\varphi(n) = (p-1)(q-1)$。這是因為 n 是由兩個質數相乘所得，而質數歐拉函數為其自身減去 1。

IV. 公開密鑰選擇: 選擇一個小於 $\varphi(n)$ 整數e，使得 e 與 $\varphi(n)$ 互質。這是為了保證有一個整數 d 使得 $e \times d \equiv 1 \pmod{\varphi(n)}$。

V. 私有密鑰計算: 計算 d，使得 d 滿足$e \times d \equiv 1 \pmod{\varphi(n)}$

利用擴展歐幾里得算法，可以找到 e 和 $\varphi(n)$ 的反元素 d。這使得 e 和 d 互為模 $\varphi(n)$ 的乘法逆元。

3. 加密與解密過程

RSA 加密和解密過程如下：

I. 加密過程:

- 將明文 M（需轉換為數字形式例如:ASCII)加密成密文 C。
- 使用公開密鑰 (e,n) 進行加密，計算$C = M^2 \ mod \ n$。
- 加密過程中，明文 M 的數值必須小於 n。

II. 解密過程:

- 將密文 C 解密回明文 M。
- 使用私有密鑰 (d,n) 進行解密，計算$M = C^d \ mod \ n$。

4. 加密與解密的正確性證明

證明解密過程的正確性，需要證明：

$$M = (M^e \ mod \ n)^d \ mod \ n$$

根據模指數運算的性質：

$$M^{ed} \equiv M \ (mod \ n)$$

$$\because e \times d \equiv 1 \left(mod \ \varphi(n)\right) \Rightarrow \text{there is exist k let } e \times d = 1 + k \times \varphi(n)$$

$$\therefore \mathrm{M^{ed}} = M^{1+k\varphi(n)} = M \times \left(M^{\varphi(n)}\right)^k$$

根據歐拉定理，對任何與 n 互質的整數 M 有：

$$\mathrm{M^{\varphi(n)} \equiv 1 \ (mod \ n)} \Rightarrow \mathrm{M} \times \left(M^{\varphi(n)}\right)^k \equiv M \times 1^k \equiv M \ (mod \ \ n)$$

這證明了解密過程的正確性。

硬體設計 (Hardware Design)

1. 硬體實現演算法分析

RSA 公鑰演算法的本質是模幕運算，根據下面次方轉換，可以把模幕運算轉為模乘運算。

$$\mathrm{a^b} = \begin{cases} 1 & if\ b = 0 \\ \left(a^{\frac{b}{2}}\right)^2 & if\ b > 0\ and\ b\ is\ even \\ a \cdot a^{b-1} & if\ b > 0\ and\ b\ is\ odd \end{cases}$$

則 RSA 模幕運算

$$C = M^e\ mod\ N$$

方法 1：e 從 LSB 到 MSB 的計算

- 初始化: $P = M, C = 1$
- 對於每一個 i 從 0 到 $k - 1$做:
  - 如果$e[i] = 1$，則$C = P \times C\ mod\ N$
  - $P = P \times P\ mod\ N$

這種方法的優點在於 C 和 P 的計算可以並行進行，因為它們之間沒有數據依賴性。

方法 2：e 從 MSB 到 LSB 的計算

- 初始化: $C = M$
- 對於每一個 i 從$k - 2$ 到 0 做:
  - $C = C \times C\ mod\ N$

- 如果$e[i]=1$，則$C=C\times M\ mod\ N$

此方法不需要暫存P，但因數據依賴性，兩次模組化乘法無法並行運行。

儘管方法 1 和方法 2 在計算速度上沒有本質區別,但由於方法 2 的循序計算特性，更適合於硬體實現中節省面積的要求，因此採用方法 2。

2. 模乘法運算硬體演算法

I. 乘法後除法（Multiply-and-Divide）:

乘法後除法方法是傳統的模乘運算方法。它分為兩個步驟:首先進行乘法運算，然後進行模除法運算。這種方法的優點是概念簡單，易於理解和實現。具體實現過程如下：

- 初始化: $\mathrm{DB}=\mathrm{A},\ \mathrm{C}=0$
- 對於每一個 i 從 0 到 255 做 (先乘法)
  - $C=C+A\times B[i]\cdot 2^{i}$
- 對於每一個 i 從 255 到 0 做 (後除法)
  - $q[i]=Q_{estimate}(C,N)\leftarrow q[i]=0,1$ (要減或不減)
  - $C=C-q[i]\times N\times 2^{i}$

優點：容易理解，面積較小。

缺點: 因為金鑰長度十分長例如 1024 位元,不可能直接用乘法或除法器直接運算，所以必須要用快速長整數加法器像進位保

存加法器（Carry-Save Adder）、前瞻加法器（Carry Look-Ahead Adder），且在運算時中間變數必須為輸入長度的兩倍例如 2048。

II. 交織的乘除法（Interleaved Multiplication and Division）：

交織的乘除法方法是對傳統乘法後除法方法的改進。它在進行乘法運算的同時，交織進行模除法，減少運算時間。具體實現過程如下：

$$C = A \times B \ mod \ N$$

$$\sum_{i=0}^{255} \{[(A \cdot 2^i) \ mod \ N] \cdot B[i]\} \ mod \ N$$

$$\Rightarrow \begin{cases} DB = A \cdot 2^i \ mod \ N \\ C \sum_{i=0}^{255} (DB \cdot B[i]) \ mod \ N \end{cases}$$

- 初始化: $\mathrm{DB} = \mathrm{A},\ \mathrm{C} = 0$
- 對於每一個 i 從1 到 255 同時做:
  - 當 $(2 \times DB - N \geq 0) \ \Rightarrow \ DB = 2 \times DB - N$

    不然 $DB = 2 \times DB$
  - 當 $(B[i] = 1) \ \Rightarrow \ C = C + DB$

    不然如果 $(C > 0) \ \Rightarrow \ C = C - k \times N$
- 把 C 適配到$0 < \mathrm{C} < \mathrm{N}$
  - 當 $(C \geq 0)$ 一直做 $C = C - k \times N$

- 當 $(C < 0)$ 一直做 $C = C + N$

優點： 算法簡單容易實現，且只要中間變數為輸入長度在加一 (因為乘 2)。

缺點： 一樣不可能直接用乘法或除法器直接運算，所以必須要用快速長整數加法器像進位保存加法器（Carry-Save Adder）。

III. 蒙哥馬利算法（Montgomery Multiplication）：

蒙哥馬利算法是一種高效的模乘算法，它避免了直接的模除法操作，通過一種位移站存器方法來進行模乘運算。具體實現過程如下：

- 將輸入轉換為蒙哥馬利表示形式

$$S = A \times B \times 2^{-n} \; mod \; N$$

- 在蒙哥馬利表示形式下進行乘法運算，並在每次乘法後進行一次蒙哥馬利約簡（Montgomery Reduction）。
- 最後將結果從蒙哥馬利表示形式轉換回標準表示形式。

優點： 避免了直接的模除法操作，提高了運算速度，特別適合於大數運算。

缺點： 需要額外的轉換步驟需多出 2^{-2n} 週期。

綜合考慮上述結論能容易模組化、規律性、擴展性，交織的乘除法運算為最佳選擇，不僅設計容易且當擴展至更大金鑰時能夠不影響程式碼結構。

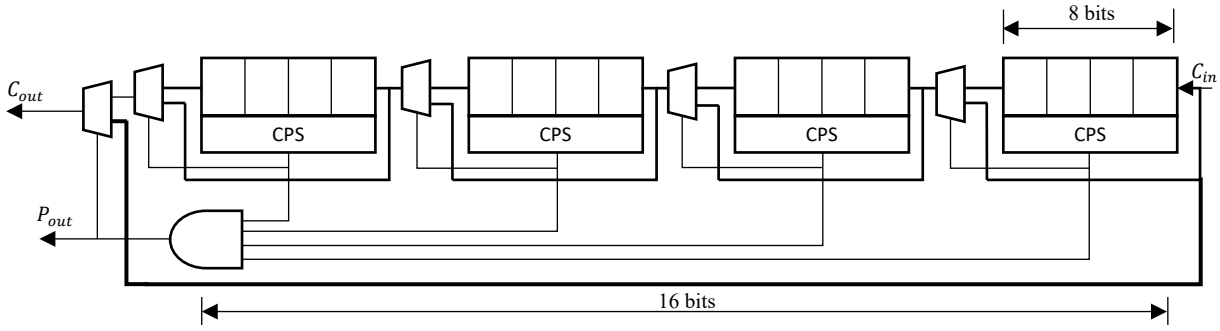
3. 加法器硬體設計

在上述方法中，長整數的加法和減法操作是核心運算。由於 RSA 演算法中涉及的數字通常非常大，因此高效的長整數加法器是整體性能的關鍵。

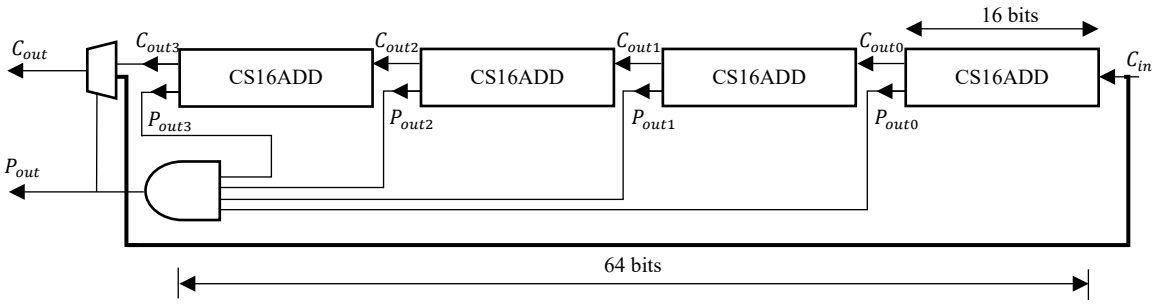
I. 進位保存加法器（Carry-Save Adder）：

將 256 位元設計成三層加法器，第一層為 4 個 4 位元前瞻進位加法器，第二層為 4 個 16 位元前瞻進位加法器，最後一層為 4 個 64 位元前瞻進位加法器。

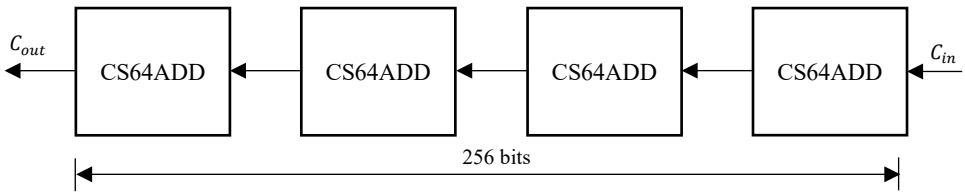
第一層:



第二層:

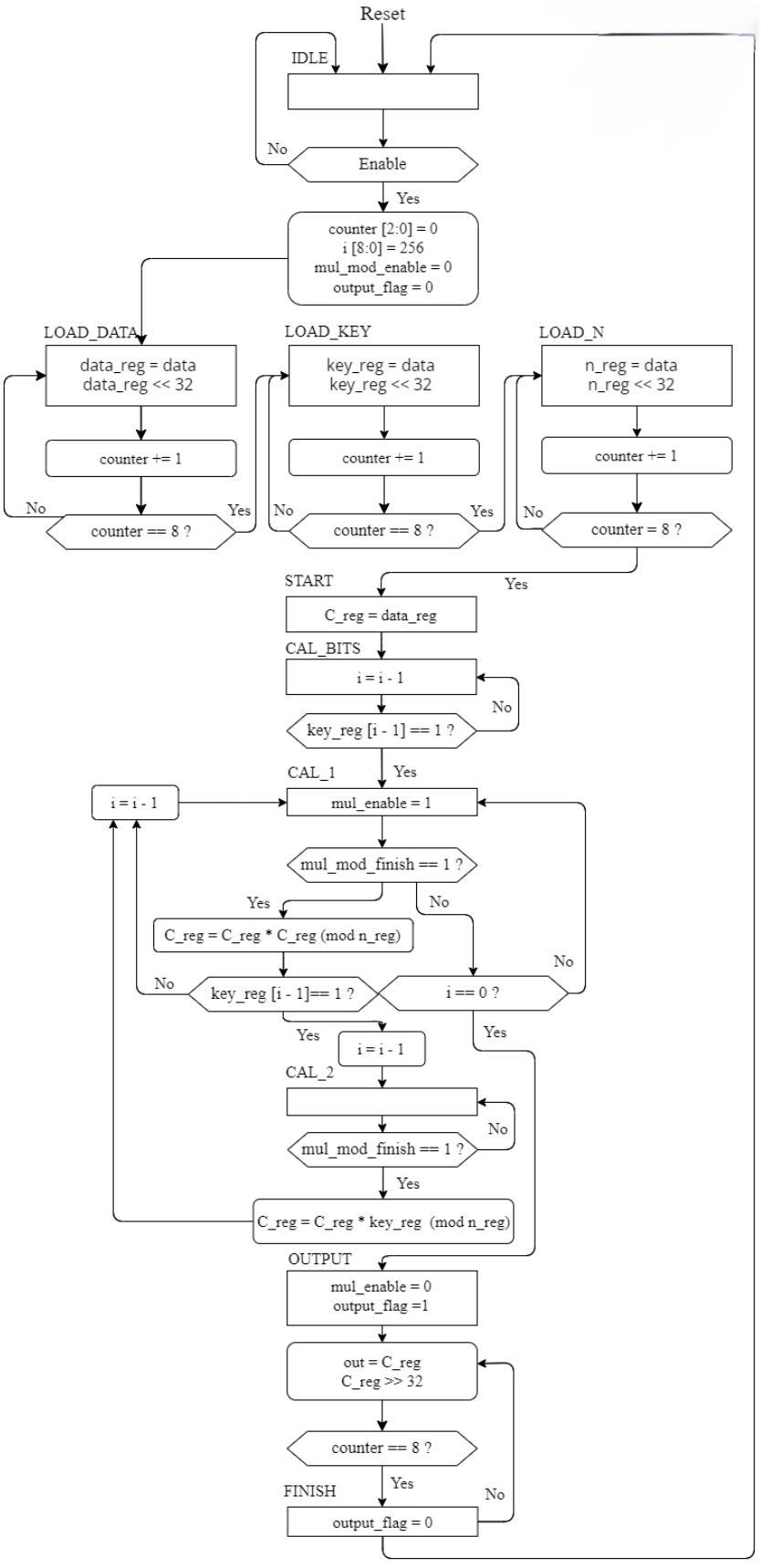


第三層:



實現細節 (Implementation Details)

1. 模幕運算狀態機



- 初始化:

模幕運算一開始會先等待外部 enable 訊號當作開始訊號（此訊號要一直保持到運算結束），當一拉起訊號外部就接著在負緣開始提供資料，先從要加解密的訊息開始傳入，緊接著傳入要使用的金鑰，最後傳入金鑰中模幕運算的模數，之所以在負緣提供資料式確保內能在正緣時正確得到資料，並在傳入格式把 256 位元資料劃分為 8 組 32 位元資料，從最高一組 32 位元做連續傳入動作。

- 計算最高位元:

CAL_BITS 狀態是為了處理在模幕運算轉為模乘運算中，運算位元是金鑰長度 $k-2$ 到 0，如果輸入金鑰長度不是完整 256 位元，則必須忽略前面為 0 的位元，直到遇到第一個 1 的位元為止。

- 計算模乘法運算第一部分:

CAL_1 狀態為計算 $C = C \times C \ mod \ N$，並等待模乘法運算模塊傳回完成訊號。當完成運算時判斷金鑰 $i-1$ 位數（因為計算位元是 $k-2$ 而在當下狀態 $i = k-1$ 所以要在減 1），如果金鑰 $i-1$ 位數為 1 則進入 CAL_2，除外回到 CAL_1 繼續計算下一個位元。

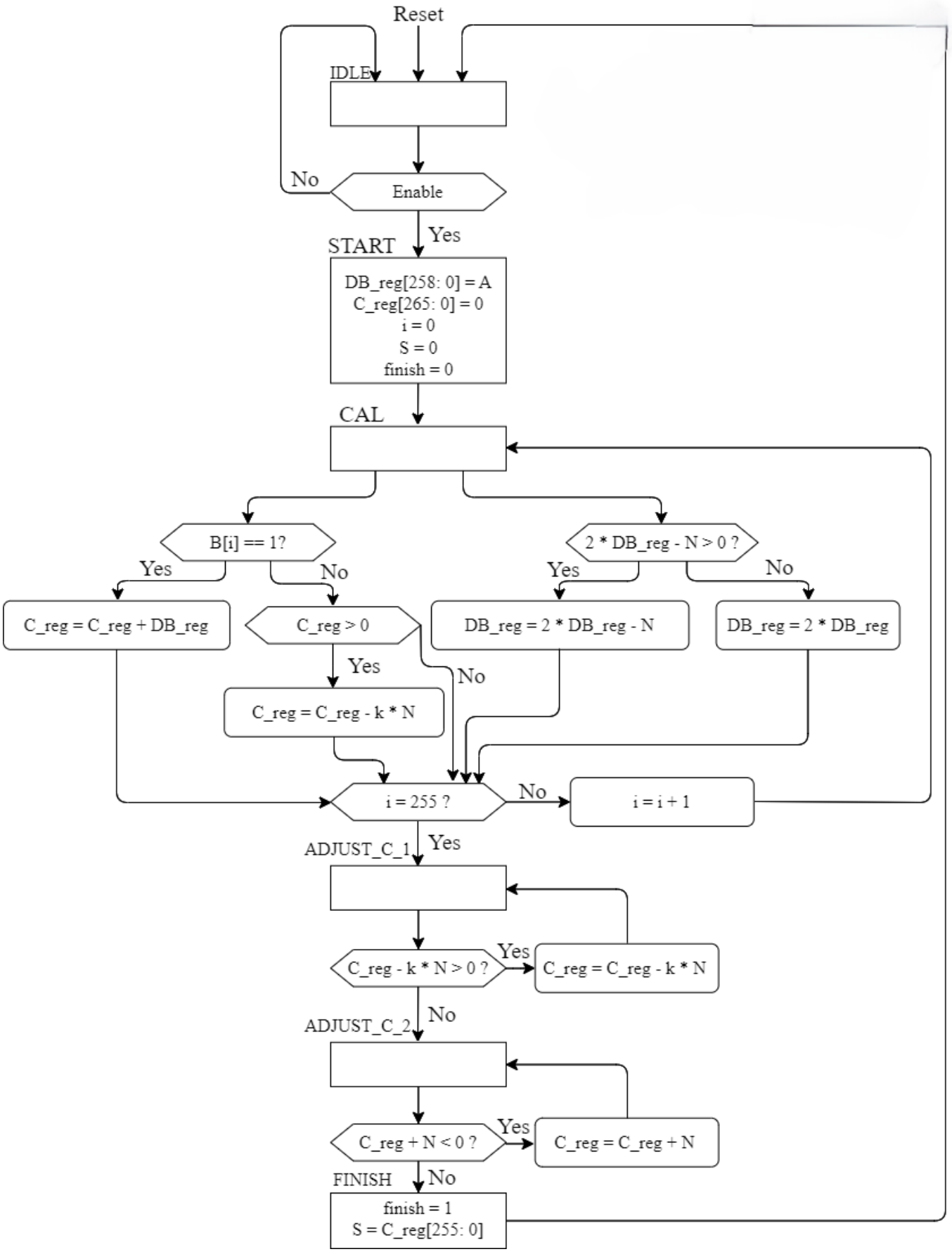
- 計算模乘法運算第二部分:

CAL_2 狀態為計算 $C = C \times M \ mod \ N$，並等待模乘法運算模塊傳回完成訊號，當完成訊號跳回 CAL_1 繼續計算下一位元。

- 回傳資料:

結束條件為當在 CAL_1 狀態中判斷到 $i = 0$，代表已經運算結束。緊接著在輸出狀態中做連續輸出，先從最高一組 32 位元做輸出，並拉起輸出訊號。在外部當讀到輸出訊號就在負半周作讀取動作以確保資料完整性。

2. 模乘運算狀態機


Reset
IDLE
No
Enable
Yes
START
DB_reg[258: 0] = A
C_reg[265: 0] = 0
i = 0
S = 0
finish = 0
CAL
B[i] == 1?
Yes
No
C_reg = C_reg + DB_reg
C_reg > 0
Yes
No
C_reg = C_reg - k * N
2 * DB_reg - N > 0 ?
Yes
No
DB_reg = 2 * DB_reg - N
DB_reg = 2 * DB_reg
i = 255 ?
No
i = i + 1
Yes
ADJUST_C_1
C_reg - k * N > 0 ?
Yes
C_reg = C_reg - k * N
No
ADJUST_C_2
C_reg + N < 0 ?
Yes
C_reg = C_reg + N
FINISH
No
finish = 1
S = C_reg[255: 0]

- 初始化:

模乘運算模塊會先等外部輸入 enable 訊號，並且當如果外部把 enable 訊號再運行之中拉低成 0，則內部狀態會直接跳回 IDLE。DB_reg 訊號因為 $DB = 2 \times DB - N$ ，會有兩倍或出現負數，會比 256 位元多出兩位元。C_reg 訊號為當$B[i] = 1$ 則 $C = C + DB$，所以最差的狀況是 $C + DB$ 加了 256 次（2^8），則 C_reg 位元數為 256 加上最差狀況 8 位元和一位元有號數。

- 平行計算 DB_reg、C_reg（CAL）:

因為 C_reg 資料要拿到前一次 DB_reg 的資料，所以 i 從 0 開始計算，讓 DB_reg 先計算出答案。在判斷式中全部都由位元運算達成，計算完資料才會判斷是否要結束計算，條件為 $i = 255$。

- 適配 $0 <$ C_reg $<$ N:

為了讓答案輸出是在 $mod\ N$ 範圍內，當狀態在 ADJUST_C_1 裡先粗略的把 C_reg 先用 $k \times N$ 減為負數，接著在詳細的慢慢加 N，直到數值是在$0 <$ C_reg $<$ N 為止。

- 參數k:

此參數是利用位移運算，使得在做除法 $mod\ N$ 時可以快速先減去 k 倍 $mod\ N$ $(N \times 2^k)$，避免當 $B[i]$ 有很多 1 時 C_reg 一直再加 DB_reg，導致 C_reg 變大，所以入參數 k 達到加速運算效果（避免減太慢）。

前模擬驗證 (Pre-simulation Verification)

1. 模擬環境設置

為了驗證設計的 RSA 加解密器的功能，我們使用 Verilog 語言編寫了一個測試平台 (testbench) 進行模擬。測試平台中設計了一個用於金鑰加解密功能的 task，這樣可以更方便地進行測試。測試平台會自動讀取外部 txt 文件中的金鑰資料並加載到內部。測試的流程如下：

I. 使用公鑰對測試資料進行加密。

II. 使用私鑰對加密後的資料進行解密。

III. 將解密後的結果與原始測試資料進行比對，如果相同則說明 RSA IP 功能正確。

2. 測試用例:

測試資料的格式使用 ASCII 碼進行加解密，每個字元佔用 1 字節 (8 位元)。由於此 RSA IP 最大能輸入 256 位元的資料，所以加密資料的字串長度最多為 32 個字元。在測試中，我們使用了最大狀況下的測試資料，即公私鑰均為 256 位元，加密資料為 32 字元。

測試金鑰是使用 Python 的 sympy 函式庫生成的大質數。當需要生成質數大於 64 位元時，可能會出現偽質數的情況，導致金鑰錯誤。下面為具體金鑰生成過程：

I. 生成兩個 128 位元的質數 P 和 q。

II. 計算模數 N 作為金鑰的一部分，其中 $N = p \times q$ 為 256 位元。

III. 生成一個 256 位元的質數作為公鑰 e。常見的公鑰數值為 0x10001 (65537)，但為了測試功能的完整性，我們選擇生成一個 256 位元的質數。

IV. 使用擴展歐幾里得算法計算私鑰 d，其中 d 是公鑰 e 的逆元素。

3. 測試結果與分析

我對設計的 RSA 加解密器進行了多次模擬測試，並記錄了以下測試結果：

- 測試資料的輸入狀況為最大位元情況，公私鑰均為 256 位元，加密資料為 32 字元。
- 在進行了 20 筆金鑰資料測試後，結果均顯示功能完全正確。
- 由於此 RSA IP 最大支持 256 位元的運算，且 RSA 加解密公能都一樣為模幕運算，因此可以向下支持任意位元的運算。
- 額外測試了 128 位元和 64 位元的情況，結果同樣顯示功能正確。
- 紀錄執行時脈數為加解密的總執行時脈，可以看見時脈數從 256 位元 (200202) 到 128 位元 (103049) 最後 64 位元 (51563)，時脈數跟位元數相差成正比都為兩倍。

- 256 位元範例:

```
RSA Testbench

Key data:
 e = 5b3d4907e8adc60572fb9710b40d8c53a472bfbdb169b6934043ec7ccaae00df
 d = 2f1fbc65d0f5636c4a1bb7507efd924a0e081ba92c367cd29ab63532460a1f9f
 N = 9892fad5470c15b673ede6d83e6bffa3dd08af313b1d401685363046fafd04e7

Testing data (expect): RSA-256 passed congragulations:)
=> ASCII 5253412d323536207061737365642063 6f6e67726167756c6174696f6e733a29

Encrypted data: 32ad6577d14830cd0bcaf4674a1172lae9ddcc6a9d3993ef9710c2aa7f8b30fd
Decrypted data (ASCII): 5253412d323536207061737365642063 6f6e67726167756c6174696f6e733a29
Decrypted string: RSA-256 passed congragulations:)
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                      congratulation you pass RSA testbench                 //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////
Cycle Count:      200202
```

- 128 位元範例:

```
RSA Testbench

Key data:
 e = 00000000000000000000000000000000fd3ed3f384d654f586b5f04686e75a73
 d = 00000000000000000000000000000000 48cf6df789561415da976eb6959ec86b
 N = 00000000000000000000000000000000a4619978011c4698c68799a75009223f

Testing data (expect):                Hello world! -_-
=> ASCII 00000000000000000000000000000000 48656c6c6f20776f726c6421202d5f2d

Encrypted data: 00000000000000000000000000000004aa98706ad2d1e9d8636ce3a9da9ea26
Decrypted data (ASCII): 00000000000000000000000000000000 48656c6c6f20776f726c6421202d5f2d
Decrypted string:                Hello world! -_-
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                      congratulation you pass RSA testbench                 //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////
Cycle Count:      103049
```

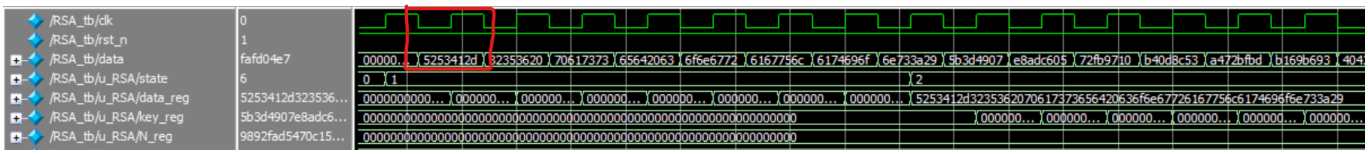
- 64 位元範本:

```
RSA Testbench

Key data:
 e = 0000000000000000000000000000000000000000000000072c07fd1fcbd1cb7
 d = 0000000000000000000000000000000000000000000000b50dd5ef0ca39987
 N = 0000000000000000000000000000000000000000000000beb26c163b30dc97

Testing data (expect):                          12345678
=> ASCII 0000000000000000000000000000000000000000000000003132333435363738

Encrypted data: 0000000000000000000000000000000000000000000008b8ca2d2b74c05cf
Decrypted data (ASCII): 0000000000000000000000000000000000000000000000003132333435363738
Decrypted string:                          12345678
////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                      congratulation you pass RSA testbench                 //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////
Cycle Count:      51563
```

測試過程中的每一筆資料均成功加密並解密，解密後的結果與原始資料完全一致，驗證了 RSA IP 的功能正確性和穩定性。

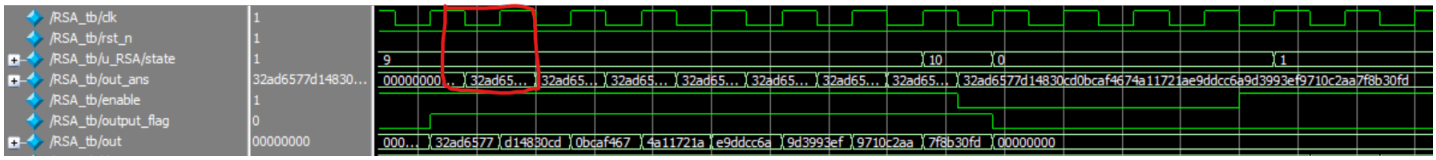
4. 傳輸資料波形分析:

- 寫入資料波形:



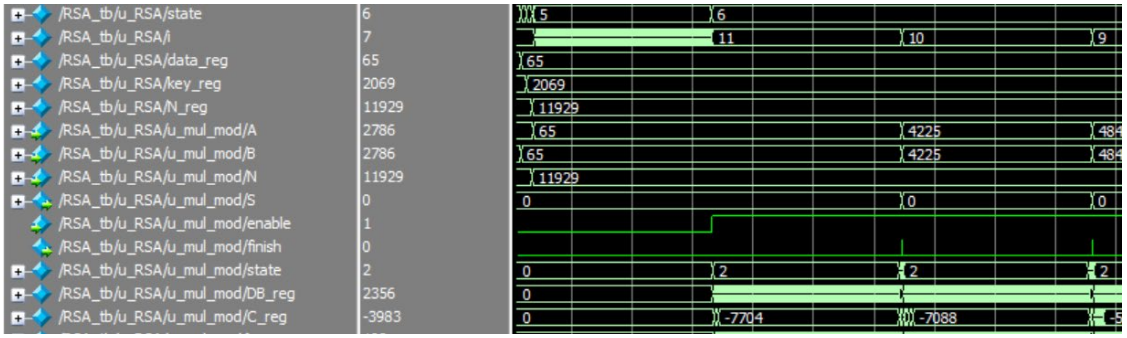
當在 state 為 1 時代表外部傳入要加解密資料，並且是從負緣開始提供資料以確保 IP 內部可以保證在正緣得到資料，並且資料為連續寫入，從最高 32 位元組做傳入動作。寫入完整加解密資料與金鑰總共花費 24 Cycles。

- 讀出資料波形:



讀出資料方式為連續讀出，當 state 為 9 代表輸出資料，並把 output_flag 拉起給外部訊號，外部讀取資料時一樣在負緣讀取，並且花費 8 Cycles 從最高 32 位元組做傳出動作。

5. 模幕與模乘運算波形



從上圖可以看出當 u_RSA/state 為 5 時代表正在找尋最高位元為 1 的位數，u_RSA/state 為 6 時代表已經找到第 i 個最高一位元，換模乘運算開始工作，目前這個波型正在計算$65^{2069}\ mod\ 11929$ (加密字元 A)，在模乘運算中先計算$A \times B\ mod\ N$ (A, B 都為輸入資料 65)，當計算完成後回傳 4425 是為正確答案。

後模擬驗證 (Post-simulation Verification)

1. FPGA 實現

在完成前模擬驗證後，我們使用 Vivado 2020.2 進行了 FPGA 合成模擬。所使用的 FPGA 板子為 Zynq-7000 系列，具體型號為 xc7z020clg400-2。在實際合成和布局過程中，我們設定了實現條件為 Performance NetDelay low，這有助於有效地降低 setup time 問題，從而減少最差負面裕量 (Worst Negative Slack, WNS)。

2. 實驗設置

實驗設置如下：

- FPGA 型號：Zynq-7000 系列 xc7z020clg400-2
- 合成工具：Vivado 2020.2
- 合成條件：Performance NetDelay low，其餘 default
- 時脈頻率：18.519 MHz（54 ns）
- 輸入腳位時間限制: max 3.8 ns, min 3.2 ns
- 輸出腳位時間限制: max 3.8 ns, min 3.2 ns

在這些設置下，進行了以下驗證模擬：

- Post synthesis functional simulation
- Post synthesis timing simulation
- Post implementation functional simulation
- Post implementation timing simulation

3. 測試結果與分析

- 合成與實現資源分析:
  - Post-Synthesis:

Resource	Estimation	Available	Utilization %
LUT	3497	53200	6.57
FF	1885	106400	1.77
IO	68	125	54.40
BUFG	1	32	3.13

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Bonded IOB (125)	BUFGCTRL (32)
N RSA	3497	1885	68	32	68	1
u_mul_mod (Mul_mod)	2926	794	68	32	0	0

  - Post-Implementation:

Resource	Utilization	Available	Utilization %
LUT	3592	53200	6.75
FF	1885	106400	1.77
IO	68	125	54.40
BUFG	1	32	3.13

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	Bonded IOB (125)	BUFGCTRL (32)
N RSA	3592	1885	68	32	1047	3592	68	1
u_mul_mod (Mul_mod)	3027	794	68	32	909	3027	0	0

- LUT（查找表）：LUT 是 FPGA 中的基本邏輯單元，可以實現任意的邏輯函數。每個 LUT 有固定的輸入和輸出。
- Slice Registers（片寄存器，FF）：用來存儲數據的單元。每個 Slice 通常包含若干個寄存器。
- F7/F8 Muxes（多工器）：多工器是用來選擇多個輸入中的一個作為輸出。F7 和 F8 Muxes 是 FPGA 中的特定多工器，用於更複雜的邏輯實現。
- Slices（邏輯單元片）：Slice 是 FPGA 的基本構建模塊，每個 Slice 包含若干個 LUT 和寄存器。
- Bonded IOB（綁定的 I/O 塊）：用於信號的輸入輸出。
- BUFGCTRL（全局時鐘緩衝控制）：BUFGCTRL 是用來控制全局時鐘信號的緩衝器。

- 合成與實現時序分析:
  - Post-Synthesis:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.921 ns	Worst Hold Slack (WHS):	0.132 ns	Worst Pulse Width Slack (WPWS):	26.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5128	Total Number of Endpoints:	5128	Total Number of Endpoints:	1886

All user specified timing constraints are met.

Setup/Hold for Input Buses - Clocked by clk - data

Source	Setup	Setup Edge	Setup Process Corner	Hold	Hold Edge	Hold Process Corner	Setup Slack	Hold Slack	Source Offset to Center
data[0]	0.140	Rise	FAST	1.019	Rise	SLOW	50.360	2.181	24.089
data[1]	0.141	Rise	FAST	1.003	Rise	SLOW	50.359	2.197	24.081

Worst Case Data Window: 1.160 ns

Ideal Clock Offset to Actual Clock: 0.439 ns

Max/Min Delays for Output Buses - Clocked by clk - out

Pad	Max Delay	Max Edge	Max Process Corner	Min Delay	Min Edge	Min Process Corner	Edge Skew
out[0]	5.814	Rise	SLOW	2.298	Rise	FAST	0.000
out[1]	5.814	Rise	SLOW	2.298	Rise	FAST	0.000

Bus Skew: 0.000 ns

*輸入與輸出接腳時間符合時間規範並使用 FDCE 為有致能接腳的異步清除 D 型正反器。

  - Post-Implementation:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.065 ns	Worst Hold Slack (WHS):	0.033 ns	Worst Pulse Width Slack (WPWS):	26.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5128	Total Number of Endpoints:	5128	Total Number of Endpoints:	1886

All user specified timing constraints are met.

Setup/Hold for Input Buses - Clocked by clk - data

Source	Setup	Setup Edge	Setup Process Corner	Hold	Hold Edge	Hold Process Corner	Setup Slack	Hold Slack	Source Offset to Center
data[0]	1.025	Rise	FAST	2.009	Rise	SLOW	49.475	1.191	24.142
data[1]	0.914	Rise	FAST	2.177	Rise	SLOW	49.586	1.023	24.281
data[2]	0.988	Rise	FAST	2.067	Rise	SLOW	49.512	1.133	24.190

Worst Case Data Window: 3.784 ns

Ideal Clock Offset to Actual Clock: 0.285 ns

Max/Min Delays for Output Buses - Clocked by clk - out

Pad	Max Delay	Max Edge	Max Process Corner	Min Delay	Min Edge	Min Process Corner	Edge Skew
out[0]	8.713	Rise	SLOW	3.221	Rise	FAST	0.013
out[1]	8.724	Rise	SLOW	3.214	Rise	FAST	0.019

Bus Skew: 0.770 ns

所有時間限制都遠小於資料輸入時間，因為外部訊號輸入資料是提前在負緣供給資料，所以可以確保內部資料完全正確。從上圖可以看見實現後時間最壞情況數據窗口（Worst Case Data Window）、理想時鐘偏移到實際時鐘（Idle Clock Offset to Actual Clock）、總線偏斜（Bus Skew）都有稍微增加。

- 時鐘驅動的路徑設置 setup 時間分析:

Intra-Clock Paths - clk - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	1.065	113	260	data_reg_reg[211]/C	u_mul_mod/FSM_onehot_state_reg[5]/D	52.623	12.837	39.786	54.000
Path 2	1.355	113	260	data_reg_reg[211]/C	u_mul_mod/FSM_onehot_state_reg[3]/D	52.490	12.834	39.656	54.000
Path 3	1.718	113	260	data_reg_reg[211]/C	u_mul_mod/FSM_onehot_state_reg[4]/D	52.218	12.834	39.384	54.000
Path 4	1.830	113	260	u_mul_mod/i_reg[0]/C	u_mul_mod/C_reg_reg[263]/D	52.139	12.654	39.485	54.000
Path 5	2.022	113	260	u_mul_mod/i_reg[0]/C	u_mul_mod/C_reg_reg[264]/D	51.981	12.674	39.307	54.000
Path 6	2.353	112	260	data_reg_reg[211]/C	u_mul_mod/C_reg_reg[260]/D	51.576	12.729	38.847	54.000
Path 7	2.389	112	260	data_reg_reg[211]/C	u_mul_mod/C_reg_reg[261]/D	51.579	12.732	38.847	54.000
Path 8	2.587	112	260	data_reg_reg[211]/C	u_mul_mod/C_reg_reg[262]/D	51.344	12.729	38.615	54.000
Path 9	2.594	112	260	data_reg_reg[211]/C	u_mul_mod/C_reg_reg[259]/D	51.377	12.729	38.648	54.000
Path 10	2.795	111	260	data_reg_reg[211]/C	u_mul_mod/C_reg_reg[258]/D	51.178	12.624	38.554	54.000

- Slack：實際時間約束與需求時間之間的差距。如果 Slack 為正值，表示路徑滿足設計要求；如果為負值，則表示設計未能滿足時間約束。
- Levels：路徑中邏輯閘的數量。
- High Fanout：信號驅動的下游信號的數量。

這些路徑的 Slack 都為正值，表示都滿足設計要求。這些路徑的總延遲都在 51.178 ns 到 52.490 ns 之間，邏輯延遲和網路延遲的分佈也相似，並且邏輯延遲約佔總延遲的四分之一，網路延遲約佔總延遲的四分之三，表示信號在連接網路上的傳播延遲是主要的延遲來源。

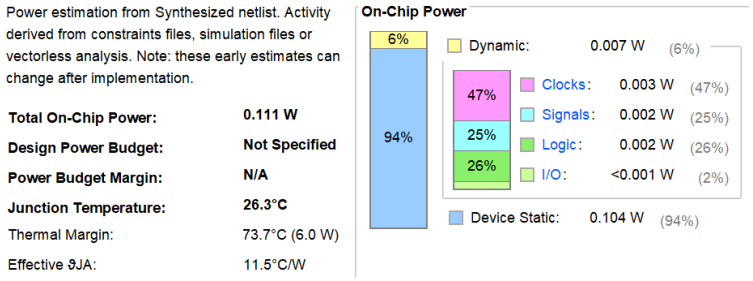
- 時鐘驅動的路徑保持 Hold 時間分析:

Intra-Clock Paths - clk - Hold

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 11	0.033	1	3	data_reg_reg[8]/C	C_reg_reg[8]/D	0.389	0.226	0.163
Path 12	0.089	1	1	u_mul_mod/S_reg[33]/C	C_reg_reg[33]/D	0.445	0.226	0.219
Path 13	0.100	1	1	u_mul_mod/S_reg[26]/C	C_reg_reg[26]/D	0.455	0.209	0.246
Path 14	0.101	1	1	u_mul_mod/S_reg[21]/C	C_reg_reg[21]/D	0.458	0.227	0.231
Path 15	0.111	1	1	u_mul_mod/S_reg[20]/C	C_reg_reg[20]/D	0.468	0.186	0.282
Path 16	0.135	1	6	u_mul_mod/C_reg_reg[47]/C	u_mul_mod/S_reg[47]/D	0.518	0.206	0.312
Path 17	0.135	1	1	u_mul_mod/S_reg[13]/C	C_reg_reg[13]/D	0.491	0.186	0.305
Path 18	0.136	1	1	u_mul_mod/S_reg[3]/C	C_reg_reg[3]/D	0.491	0.226	0.265
Path 19	0.136	1	5	u_mul_mod/C_reg_reg[156]/C	u_mul_mod/S_reg[156]/D	0.492	0.186	0.306
Path 20	0.142	1	2	key_reg_reg[1]/C	key_reg_reg[33]/D	0.286	0.189	0.097

Path 11 的 Slack 為 0.033，表示滿足時間約束，但有較少的時間餘裕。此路徑有 1 層邏輯，扇出數量為 3。總延遲為 0.389 ns，其中邏輯延遲為 0.226 ns，網路延遲為 0.163 ns。Path 12 到 Path 20，這些路徑的 Slack 值都為正，表示這些路徑都滿足設計要求。這些路徑的總延遲都在 0.286 ns 到 0.518 ns 之間，邏輯延遲和網路延遲的分佈也相似，表示這些路徑的延遲主要由邏輯延遲和網路延遲共同決定。

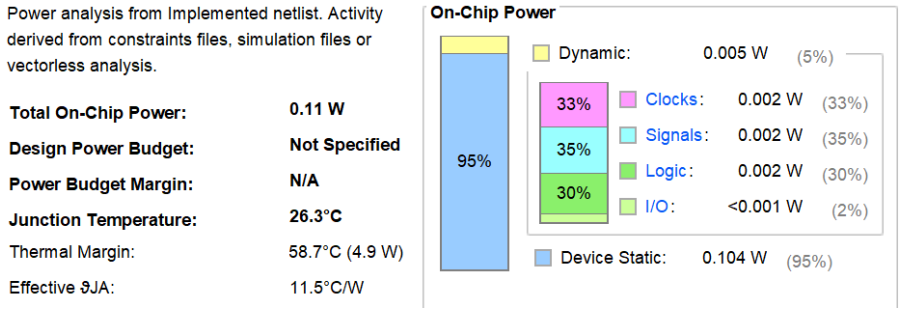
- 功率分析:
  - Post-Synthesis:



Utilization	Name	Clocks (W)	Signals (W)	Data (W)	Clock Enable (W)	Logic (W)	I/O (W)
0.007 W (6% of total)	RSA						
0.004 W (4% of total)	u_mul_mod (Mul_mod)	0.001	0.001	0.001	<0.001	0.001	<0.001
0.004 W (4% of total)	Leaf Cells (4488)						
0.003 W (3% of total)	Leaf Cells (2097)						

RSA-256 IP 動態功耗為 7mW，其中模乘運算為最主要的計算單元，佔動態功耗 57%為 4mW，裡面包含到一個 258 位元和一個 265 位元進位加法器。

  - Post-Implementation:



Utilization	Name	Clocks (W)	Signals (W)	Data (W)	Clock Enable (W)	Logic (W)	I/O (W)
0.005 W (5% of total)	RSA						
0.003 W (3% of total)	u_mul_mod (Mul_mod)	0.001	0.001	0.001	<0.001	0.001	<0.001
0.003 W (3% of total)	Leaf Cells (4488)						
0.002 W (2% of total)	Leaf Cells (2097)						

在經過實現後動態功耗降低至 5 mW，板上總功耗降低至 110 mW，整體功耗下降 25%。

4. 時脈分析:

分別測試資料有 256 位元數全滿的情況，還有 128 位元以及 64 位元情況，且做十筆隨機資料，參數 k 設 0 (N 乘一倍)，由於記錄時脈數為加密與解密的總時脈數所以每筆資料又計算 2 次，最終總共取 20 次平均，成績如下表:

	256-bits	128-bits	64-bits
最高	203099	102263	52088
最低	195670	97290	44228
平均時脈數	200191	98885	43842

可以看見當 256 位元時，平均做一次模幕運算需要 100095 Cycles，而 128 位元為平均 49442 Cycles，64 位元平均 21921 Cycles。以位元差異作分析都差不多相差一倍。

- 評估時脈數:
  - 模幕運算：需分別需計算 256 次的 $C = C \times C \ mod \ N$ 與依照輸入金鑰為 1 的數量要再多做一次 $C = C \times M \ mod \ N$。
  - 模乘運算：需同時做 256 次 DB、C 的乘除運算，且最後把資料是配到 mod N 裡，在 $k = 0$ 狀況下，需多做 2 次運算。

$$Total\ worst\ case = 256 \times 2 \times (256 + 2) = 132096$$

- 平均時脈分析

  因為外部輸入金鑰與資料為隨機資料，所以根據大數法則資料會呈現高斯分佈，所以金鑰為 1 的數量為一半的機率為最大，則平均運算時脈如下。

$$Average\ worst\ case = 256 \times 1.5 \times (256 + 2) = 99072$$

- 可以看出理論值與實際運算數值十分相近。
- 最終，總運算時間:

$$\text{運行時間} = \text{時脈數} \times \text{週期時間} = 99072 \times 54\ \text{ns} = 5.3\ \text{ms}$$

5. 參數 k 值分析:

k 值設定是使得模乘運算可以快速的減去 2^k 倍 N，因為硬體設計是使用位移運算子來做到目的，所以當 $k = 0$ 則是無位移$(-N)$。次參數在輸入金鑰與資料的位元數較平均時只需設定成 0，而當為元數相差較大可以調整成 1 或 2 使得在模乘運算中 C 可以加速適配在$0 < C < N$，以達到加速效果。

結論

在此次專案中，我們成功設計並實現了一個基於 FPGA 的 RSA-256 加解密器。經過前模擬和後模擬驗證，該加解密器在功能和時序上均達到了預期目標。然而，該設計在運行頻率上仍存在一些限制，僅達到了 18.52 MHz，導致在加解密 256 位元長度的金鑰與資料時需要花費約 5.3 毫秒的時間。

造成運行頻率較低的主要原因是 Setup time violation 問題。在設計過程中，我們發現部分路徑過長，導致時序不滿足要求，進而影響整體性能。為了解決這一問題，我們計劃在未來的改進中採用以下措施：

- Pipeline 設計:

  將設計改為流水線結構，以縮短每個計算步驟的延遲時間。這樣可以將整體運算分為多個階段，每個階段的運算時間減少，從而提升整體運行頻率。

- 路徑優化:

  對於較長的路徑，增加暫存器以分割數據傳輸，讓數據在多個時脈週期內完成傳輸，避免單一路徑過長導致的時序違反問題。這可以通過多等待一個時脈週期來實現，以滿足時序要求並提升整體頻率。