

數位電路實驗：Lab2 — RSA256 解密機

組別：第七組

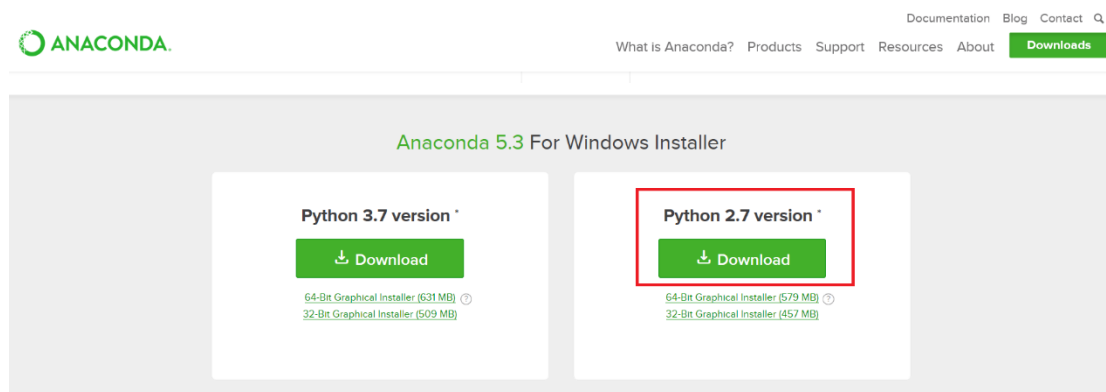
成員：李旻芳、洪鈺萌、劉力仁

一、使用說明

1. Python2 下載:

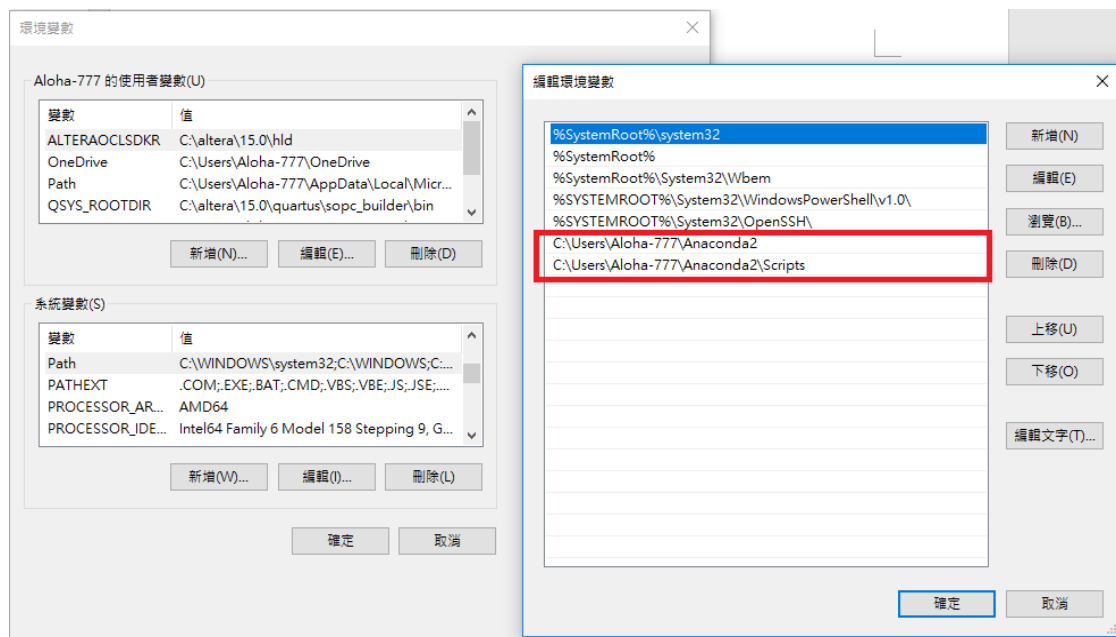
如果是 windows 10 系統，到 Anaconda 網站

(<https://www.anaconda.com/download/#windows>) 下載 Anaconda2

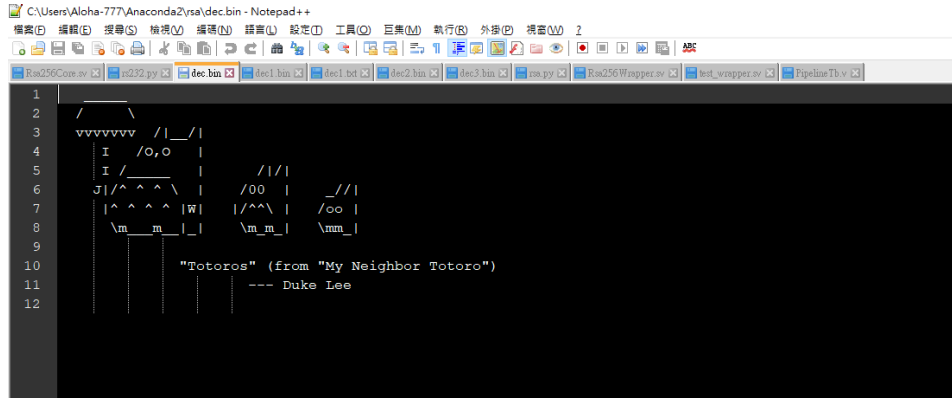


2. 修改環境變數:

壓 window 鍵，打“編輯系統環境變數”，點選環境變數，在系統變數的欄位中，點 Path，新增紅框中兩個環境變數至 path，壓確定。(如此在命令列打 python 才搜尋的到 python2，也才可以用 pip install，如果命令列找不到 python，請重新開機)



3. 下載 pyserial 套件



二、教學說明

1. 實驗目的

透過以硬體實作 RSA256 解碼演算法，了解不同運算對於硬體效率的影響，進而體會硬體加速的重要性，並學習利用數學方式簡化需消耗較多硬體資源之乘法及模運算；接著，利用 Qsys 套用 Altera 提供的其他 IP；此外，通過 Avalon MM 介面來實作 RS232 的溝通架構以理解模組溝通的基礎模式，可為日後進一步邁入 SoC 相關領域的重要概念建設。

2. RSA Cryptosystem

RSA 演算法為非對稱加密演算法，即以可為大眾取得的公鑰（public key）進行加密，並以不公開的私鑰（private key）對被相對應公鑰加密的資料進行解密，此方法可確保資料不被第三者得知。

所謂「鑰」由 N 、 e 、 d 三個數字構成。 N 為兩個極大的質數乘積，而 e 與 d 滿足： $(e \cdot d) \bmod N \equiv 1$ 。

RSA 加密：傳送端以接收端之公鑰 (N, e) 對訊息段 m 進行加密

加密訊息 $m_e = m^e \bmod(N)$ 。

RSA 解密：接收端以自己的私鑰 (N, d) 對加密訊息段 m_e 進行解密

解密訊息 $m = (m_e)^d \bmod(N)$ 。

更詳細有關 RSA 加密方式的推導可上網搜尋或參考其他書籍。

在了解 RSA Cryptosystem 後，再來對 RSA 演算法進行修改，希望能以最小硬體資源計算 $y^d \bmod(N)$ 。當計算 y^d 時，最簡單的方法是直接視為 d 個 y 的連乘，然而考量連乘所需要的 word length 且 mod 對極大數字運算非常昂貴，因此，運用 mod 的特性來得到更有效率的次方算法：

首先，將 y^d 的 d 以二進位制表示，二進制會使得每個乘數之間是平方關係， $y^8 = (y^4)^2 = ((y^2)^2)^2$ 。以 $d = 12 = (1100)_2$ 為例，

$$y^d = y^{1100} = (1 \cdot y^8) \cdot (1 \cdot y^4) \cdot (0 \cdot y^2) \cdot (0 \cdot y^1)。$$

接著，將餘數運算拆在連乘過程中執行，即是把 $y^d \bmod(N)$ 之運算轉換為多次的 $ab \bmod(N)$ ，由於在每一個 iteration 中均取 mod，因此運算結果不會如直接做次方一般有指數性的成長。簡要證明可假設 $a = c \cdot N + r$, $c, N, r \in \mathbb{Z}$ ，則

$$((a \bmod(N)) \cdot a) \bmod(N) \dots a) \bmod(N) = ((r) \cdot a) \bmod(N) \dots a) \bmod(N) \\ = r^d \bmod(N) \equiv a^d \bmod(N) ,$$

所以可得如下之演算法

Algorithm 1. Exponentiation by Squaring Algorithm (mod N)

Input: positive integers y, d, N

Output: $y^d \bmod N$

```

1: function Exp ( $y, d, N$ )
2:    $t \leftarrow y$ 
3:    $ret \leftarrow 1$ 
4:   for  $i \leftarrow 0$  to  $\lfloor \log_2 d \rfloor$  do
5:     if  $i$ -th bit of  $d$  is 1 then
6:        $ret \leftarrow ret \cdot t \bmod N$ 
7:     end if
8:      $t \leftarrow t \cdot t \bmod N$ 
9:   end for
10:  return  $ret$ 
11: end function
```

同樣，可將乘法換成加法。假設 $a = 12$ ，

$$ab \bmod(N) = 12 \cdot b \bmod(N) = 1100 \cdot b \bmod(N) = (8b + 4b) \bmod(N)。$$

並且把 mod 換成減法，故可得如下之演算法

Algorithm 2. Modulo of Products

Input: positive integers a, b, N

Output: $ab \bmod N$

```

1: function MOP ( $a, b, N$ )
2:    $t \leftarrow a$ 
3:    $ret \leftarrow 0$ 
4:   for  $i \leftarrow 0$  to  $\lfloor \log_2 b \rfloor$  do
5:     if  $i$ -th bit of  $b$  is 1 then
6:       if  $ret + t \geq N$  then
7:          $ret \leftarrow ret + t - N$ 
8:       else
9:          $ret \leftarrow ret + t$ 
10:    end if
11:  end if
12:  if  $t + t \geq N$  then
13:     $t \leftarrow t + t - N$ 
14:  else
```

```

15:          $t \leftarrow t + t$ 
16:     end if
17: end for
18: return  $ret$ 
19: end function

```

然而，Algorithm 2 為了確保不會發生 overflow，在每個 iteration 均須與 N 比較大小來判斷是否進行減法運算，是故考慮以 $ab \cdot 2^{-i} \bmod(N)$ 來取代 $ab \bmod(N)$ ，即每個 iteration 均乘以 2^{-1} 來避免 overflow，所以不需要每個 iteration 皆做餘數運算。假設 $a = 4$ ， $i = 4$ ，則

$$ab \cdot 2^{-i} = 4'b1100 \cdot b \cdot 2^{-4} = (((b \cdot 2^{-1} + b) \cdot 2^{-1} + 0) \cdot 2^{-1} + 0) \cdot 2^{-1},$$

注意，此處的 $(b \cdot 2^{-1} + b)$ 仍須 5 bits 來儲存以避免 overflow。再來，考慮 $i = 1$ 的情形，

$$a \cdot 2^{-i} \equiv b \cdot 2^{-4} \bmod(N),$$

所以，if (a is even)， $b = \frac{a}{2}$ ，else if (a is odd)， $b = \frac{a+N}{2}$ 。於是，可以得到如下的 Montgomery Algorithm

Algorithm 3. Montgomery Algorithm

Input: positive integers a, b, N

Output: $ab \cdot 2^{-(\lfloor \log_2 b \rfloor + 1)} \bmod(N)$

```

1: function Mont( $a, b, N$ )
2:      $ret \leftarrow 0$ 
3:     for  $i \leftarrow 0$  to  $\lfloor \log_2 b \rfloor$  do
4:         if  $i$ -th bit of  $b$  is 1 then
5:              $ret \leftarrow ret + a$ 
6:         end if
7:         if  $ret$  is odd then
8:              $ret \leftarrow ret + N$ 
9:         end if
10:     $ret \leftarrow ret/2$ 
11: end for
12: if  $ret \geq N$  then
13:      $ret \leftarrow ret - N$ 
14: end if
15: return  $ret$ 
16: end function

```

最後結合上述，修改 RSA 演算法如下

Algorithm 4. RSA Algorithm

Input: positive integers y, d, N

Output: $y^d \pmod N$

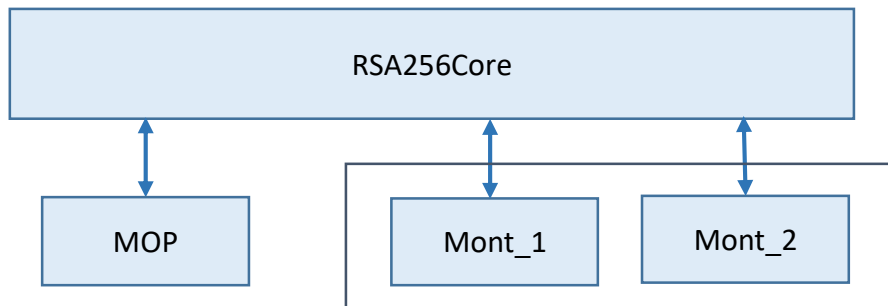
```

1: function RSA ( $y, d, N$ )
2:    $t \leftarrow \text{MOP} (y, 2^{\lfloor \log_2 d \rfloor + 1}, N)$ 
3:    $m \leftarrow 1$ 
4:   for  $i \leftarrow 0$  to  $\lfloor \log_2 d \rfloor$  do
5:     if  $i$ -th bit of  $d$  is 1 then
6:        $m \leftarrow \text{Mont} (m, t, N)$ 
7:     end if
8:      $t \leftarrow \text{Mont} (t, t, N)$ 
9:   end for
10:  return  $m$ 
11: end function

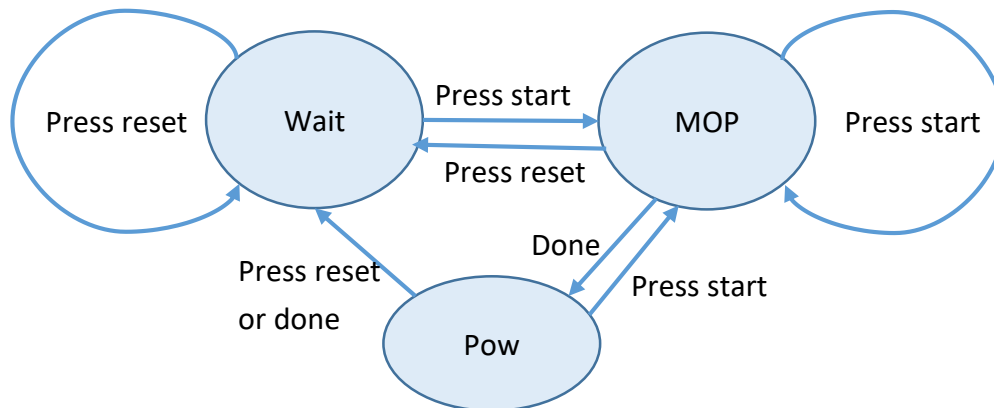
```

3. 撰寫 RSA256Core.sv

可以用 start 與 done 來控制 states，block diagram 如下圖：



整合 Mont_1 和 Mont_2 並命名為 Pow，參考 Algorithm 4 則 Core 的 finite state machine 可設計如下：



其中，Wait 時會等待訊號輸入；MOP 輸出 $ab \pmod N$ ，並把 2^{256} 代入 b ，詳見 Algorithm 3；Pow 輸出最終結果。而要使用 Rsa256Core Testbench 時，輸入 `ncverilog +access+rb tb.sv Rsa256Core.sv`。

4. 撰寫 Rsa256Wrapper.sv

Wrapper 負責的工作主要有兩個：

- (1) 在恰當的時候讀資料 (key & encoded data)。

(2) 在恰當的時候寫資料 (decoded data)。

在撰寫 Rsa256Wrapper.sv 前要先了解 Altera 提供的 UART 模組使用的 Avalon 介面要如何接收與傳送，詳細內容可參考官方文件。

Avalon Memory-Mapped Interface 提供許多功能，但此次實驗使用的 RS232 UART 模組僅會用到以下訊號：

Signal	Type	Description
avm_clk	input	clock
avm_rst	input	Reset
avm_read	output	當 read data 時，avm_read = 1，其餘時間為 0
avm_write	output	當 write data 時，avm_write = 1，其餘時間為 0
avm_address	output	讀寫資料時的資料位置
avm_waitrequest	input	當 avm_waitrequest = 0 時，才能進行讀寫的動作
avm_readdata	input	讀進來的資料
avm_writedata	output	寫出去的資料

RS232 Qsys Module：

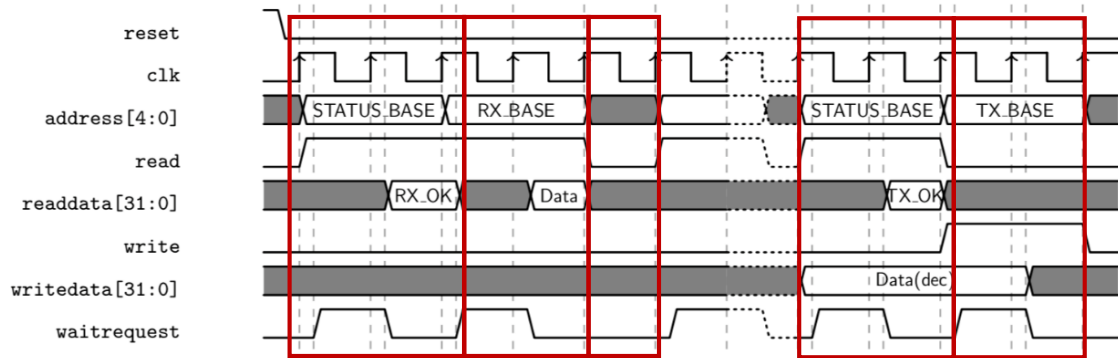
Offset	Register Name	R/W	Description/Register Bits													
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata	RO	Reserved					1	1	Receive Data						
1	txdata	WO	Reserved					1	1	Transmit Data						
2	status2	RW	Reserved	eop	cts	dcts	1	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	control	RW	Reserved	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itm	itoe	iroe	ibrk	ife	ipe
4	divisor3	RW	Baud Rate Divisor													
5	endof-packet3	RW	Reserved					1	1	End-of-Packet Value						

注意：StartRead 時，將 avm_read \leftarrow 1，avm_write \leftarrow 0，avm_address \leftarrow 8；

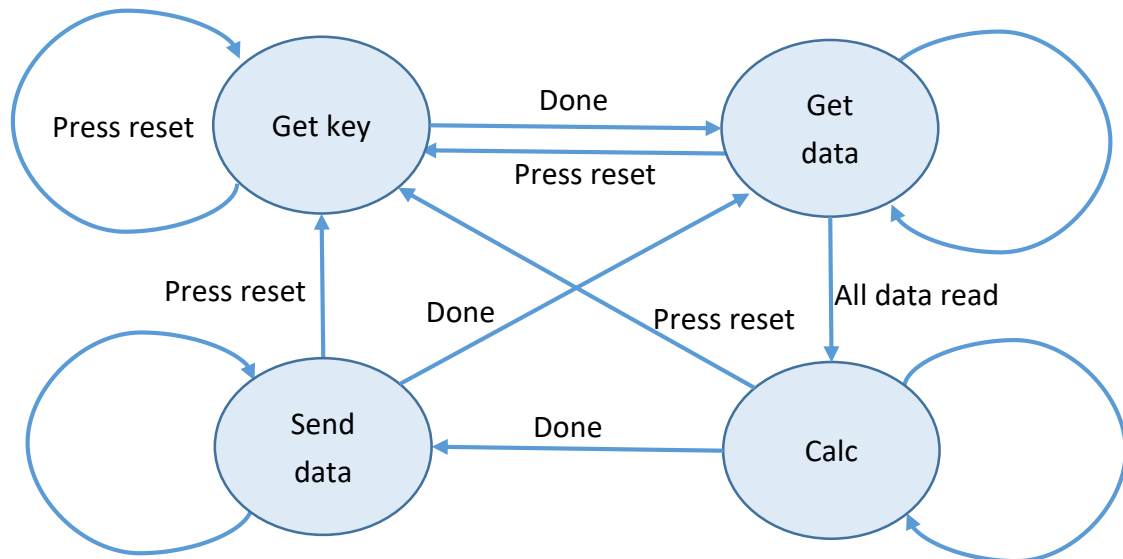
StartWrite 時，將 avm_read \leftarrow 1，avm_write \leftarrow 0，avm_address \leftarrow 0；

SendData 時，將 avm_read \leftarrow 0，avm_write \leftarrow 1，avm_address \leftarrow 4。

令 RX_BASE = 0，TX_BASE = 4，STATUS_BASE = 8，則觀察以下波形圖可以發現，不論讀資料或者寫資料，address 都要先回歸 STATUS_BASE，當 avm_waitrequest = 0 且 RX_OK = 1 時 StartRead，當 avm_waitrequest = 0 且 TX_OK = 1 時 StartWrite。其中，TX_OK 要看 avm_readdata[6] (即 trdy)，RX_OK 則需要看 avm_readdata[7] (即 rrdy)，另外，本實驗送資料的順序為先送 N 再送 e，最後才是 encoded data。



FSM 可設計如下：



其中，每次讀資料僅能讀取 8 bits = 1 byte (avm_readdata[7:0])，而不論 key(N, e) 或 encoded data 都是 256 bits (32 bytes)，因此各自需要 32 次讀資料的過程，並且要注意 key(N, e) 只會讀取一次，而之後的運算則是使用相同的 key 去算出不同 encoded data 的結果。另外，decoded data 為 31 bytes，故需 31 次寫資料的過程，而寫資料的順序為從第 247 bit 開始直到第 0 個 bit 結束，例如以下寫法：

assign avm_writedata = dec_r[247:240] 。

而要使用 Rsa256Wrapper Testbench 時，輸入

```
ncverilog +access+r test_wrapper.sv PipelineCtrl.v \
PipelineTb.v Rsa256Wrapper.sv Rsa256Core.sv
```

5. Qsys 及使用到的 Altera 模組

(1) 建立 Qsys 專案

實際方法可參考助教提供的檔案。基本上只要照著助教投影片一步步設定即可。

(2) Avalon Memory-Mapped Interface

Avalon MM 即為本次實驗所使用的輸出入介面，詳細運作方法已於上方「撰寫 Rsa256Wrapper.sv」說明。

(3) ALTPLL

ALTPLL 為 Altera 所提供的 PLL 模組。PLL 可根據輸入的 CLOCK 頻率，輸出更為低頻的 CLOCK 訊號。本次實驗中輸入為 50MHz，輸出為 40MHz。

(4) UART(RS232)

UART(Universal Asynchronous Receiver/Transmitter)的功能是在 Serial 訊號和 Parallel 訊號之間做轉換，本實驗使用的 Serial 介面為 RS-232。

6. 遇到問題與解決

(1) Algorithm：

一開始看助教提供的有關 RSA256 原理及如何在 FPGA 實現的說明時，我們看了相當久的時間但都遲遲無法理解，因此對於如何寫程式茫無頭緒，後來我們嘗試直接代一些比較小的數字去跑那些演算法的邏輯，在有實際數字運算下，我們才總算搞懂背後的原理也發現助教給的檔案有一些小錯誤並做出修正。

(2) Counter：

- (a) 當使用 counter 來實作 for loop 時，要注意 counter 結束要送出 done 訊號時資料是否算完，否則可能出現測資小時答案正確，但大測資時答案卻錯誤之情形。
- (b) Counter 的結果與想像的不同可能是因為起始值或結束值設定錯誤，或是因為未考慮到 wire 和 register 差了一個 cycle。

(3) Wrapper：

- (a) 和在理解演算法的時候一樣，我們一開始也看不懂助教給的範例波形，究竟甚麼時候該讀資料，甚麼時候該寫資料，也不理解在讀資料前先回到 STATUS_BASE 以及 RX_OK 和 TX_OK 這些接線代表甚麼，硬著頭皮設計 Finite State Machine，常常遇到一些讓自己很混亂的問題(e.g. 前一個 state 當達成哪些條件才會跳到另一個 state)，搞到我們幾乎快崩潰了，然而實際參考官方文件《Avalon Interface Specifications》，並且一方面藉著實際寫程式，另一方面嘗試去摸索、猜測那些波形真正的意思，總算以自己有限的理解找到一組看似合理的實作方法。
- (b) 然而好不容易設計出感覺滿合理的 FSM，但到工作站跑測資卻一直出現「Simulation Abort」的結果，看了 nWave 檢查波形，才發現原來有些訊號我們沒有考慮周到，不小心讓在 avm_read 且 should_keep 為 1 的時候改變 avm_address 的值；此外，在 write 時，我們一開始寫完第一筆資料也忘記將各個狀態回歸到最初的值(也就是 avm_read=1、avm_write=0、avm_address=STATUS_BASE)，這樣會導致波形一直在 write 狀態，程式就永遠都沒辦法跑完(因為要在 avm_read=1 且 avm_address=STATUS_BASE 才能夠更新下一次的狀態)。
- (c) 修改完畢後，結果又從原本的「Simulation Abort」變成了「Simulation Fuck」，一樣開 nWave 檢查讀入與輸出的訊號，並且還好我們之後有

再重新去看實驗室 lab2 的 `readme` 說明文件，才發現原來 `key` 在整個運算中只會給一次，而之後才是一直給不同的 `encoded data` 去計算各種不同的答案，而不像一開始我們所想的每次計算都傳入不同的 `key`，稍微改了一下原本 `FSM` 的架構，總算解決了這個問題。

- (4) 燒到板子上前請善用 `testbench`，切勿直接在板子上 `debug`。
- (5) 將程式燒入 `FPGA` 板，且用 `python` 完成資料傳輸後，要記得打開 `dec.bin` 文件看最終結果，結果並不會顯示在命令提示字元或 `FPGA` 板上。