數位電路實驗

Lab2 – RSA Decoder

B04901060 黃文璁、B04901080 戴靖軒、B04901048 陳則宇

零、摘要

這份教學手冊前半部分原理和狀態圖設計，和其他手冊範例大同小異，可以交互參考。後半部分則包含了我們如何製作出額外的功能（連續傳檔案、切換 128~1024bit）等，以及在測試的過程中常遇到哪些 bug，還有在測試額外功能時遇到的一些問題，而又可以朝哪些方向解決，希望可以給將來同學遇到類似問題時作為參考。

一、實驗目的

1. 了解 RSA 加密演算法的基本原理。
2. 了解如何利用數學方法，避開對硬體資源要求較多的乘法運算和模運算。
3. 練習利用 Qsys 套用 Altera 提供的其他 IP，並藉由 Avalon MM 介面進行 RS-232 傳輸。

二、原理

（一）RSA 演算法

RSA 演算法是一種非對稱加密演算法，也就是使用公鑰、私鑰兩個金鑰分別進行加密、解密。若有公鑰 (N,e)、私鑰 (N,d)，則其他人可將訊息 m 分段後利用公鑰 (N,e) 對訊息段 n 進行加密，得到加密後的訊息段 c，方法為：

$$n^e \equiv c\ (mod\ N)$$

利用私鑰 (N,d) 可以將訊息 c 解密：

$$c^d \equiv n\ (mod\ N)$$

詳細的數學原理可以參考其他網路資料或書籍。

比如說維基百科：https://zh.m.wikipedia.org/wiki/RSA%E5%8A%A0%E5%AF%86%E6%BC%94%E7%AE%97%E6%B3%95

（二）快速冪演算法、Montgomery 演算法

在將 RSA 演算法實作到 FPGA 上之前，要先了解如何設計演算法，來加速冪運算以及乘法運算，以及如何避免模運算。

1. 快速冪演算法

計算 y^d 時，最基本的想法是將 y 乘上 d 次，便是一個 $O(d)$ 的演算法。但對於高達 256bit 的 d 時，這個算法顯然不適用，

但我們可以觀察到在計算模運算時，y^d 有以下關係：

$$\begin{cases} y^d = y^{\frac{d}{2}} \cdot y^{\frac{d}{2}} & d\ even \\ y^d = y^{\lfloor \frac{d}{2} \rfloor} \cdot y^{\lfloor \frac{d}{2} \rfloor} \cdot y & d\ odd \end{cases}$$

換句話說，在計算 y^d 時，可以將問題轉換為計算 $y^{\frac{d}{2}}$ 後再乘起來。

也就是可以藉由 d 的 bit pattern 來計算 y^d。舉例來說，計算 y^{25} 時，注意到 $25 = (11001)_2$，故可以將問題轉換成：

$$y^{25} = 1 \cdot y^{16} + 1 \cdot y^8 + 0 \cdot y^4 + 0 \cdot y^2 + 1 \cdot y^1$$

注意到 y^{2^k} 的計算只要在每次迴圈中進行 $y \leftarrow y \cdot y$ 即可。

完整演算法如下：

Algorithm 1. Fast Power Algorithm

Input: positive integers y, d
Output: y^d

```
1:  function FastPow(y, d)
2:      t ← y
3:      ret ← 1
4:      for i ← 0 to ⌊log2 d⌋ do
5:          if i-th bit of d is 1 then
6:              ret ← ret · t
7:          end if
8:          t ← t · t
9:      end for
10:     return ret
11: end function
```

注意到在 RSA 演算法中，所有運算都是在 mod N 下進行，考慮到 mod 在乘法下的性質：$a\ (mod\ N) \times b\ (mod\ N) = ab (mod\ N)$，可以將上述演算法改寫為適用於金鑰長度為 256bit 的 RSA 版本：

Algorithm 2. Fast Power Algorithm (for RSA 256bit)

Input: 256bit integers y, d, N

Output: $y^d \ (mod\ N)$

```
1:  function FastPow(y, d, N)
2:      t ← y
3:      ret ← 1
4:      for i ← 0 to 255 do
5:          if i-th bit of d is 1 then
6:              ret ← ret · t (mod N)
7:          end if
8:          t ← t · t (mod N)
9:      end for
10:     return ret
11: end function
```

這時候有了新的問題，演算法中使用了兩次乘法和模運算，兩種操作對硬體來說都是不小的負擔，於是我們可以利用和快速冪類似的概念，利用加法進行乘法運算。舉例來說，計算 $y\times25$ 時，可以將他拆成：

$$y\times25 = 1\cdot16y + 1\cdot8y + 0\cdot4y + 0\cdot2y + 1\cdot y$$

同樣的，$2^k y$ 的計算只要在每次迴圈中進行 $y \leftarrow y + y$ 即可。

Algorithm 3. Multiplication using addition

Input: 256bit integers a, b, N

Output: y^d

```
1:  function Mul(a, b, N)
2:      t ← a
3:      ret ← 0
4:      for i ← 0 to 255 do
5:          if i-th bit of b is 1 then
6:              ret ← ret + t (mod N)
7:          end if
8:          t ← t + t (mod N)
9:      end for
10:     return ret
11: end function
```

注意到上述演算法會使用模運算，但可以注意到計算過程中 ret 和 t 都會小於 N，故相加後會小於 $2N$，可以直接判斷相加後是否大於 N 後，用減法取代模運算：

Algorithm 4. Multiplication using addition (without mod operation)

Input: 256bit integers a, b, N
Output: $ab \bmod N$

```
1:  function Mul(a, b, N)
2:      t ← a
3:      ret ← 0
4:      for i ← 0 to 255 do
5:          if i-th bit of b is 1 then
6:              if ret + t ≥ N then
7:                  ret ← ret + t − N
8:              else
9:                  ret ← ret + t
10:             end if
11:         end if
12:         if t + t ≥ N then
13:             t ← t + t − N
14:         else
15:             t ← t + t
16:         end if
17:     end for
18:     return ret
19: end function
```

若將這個演算法套用到前面的快速冪演算法中，基本上已經是硬體上可行的程式了，不過注意到上面的乘法運算中，用了兩個比較運算，要針對這方面繼續進行優化的話，可以使用接下來要介紹的 Montgomery 演算法。

2. Montgomery 演算法

Montgomery 演算法是用來計算 $ab(mod\ N)$ 的演算法，先將 a, b 轉換成 $aR(mod\ N)$ 和 $bR(mod\ N)$，其中 R 是與 N 有關的數字，接著此演算法可以計算出 $abR(mod\ N)$，最後再轉換回$ab(mod\ N)$。

本實驗中使用此演算法計算 256bit 的乘法，而 R 的值為 2^{256}，原理如下。

對 256bit 的數字 a, b 而言，假設已知 $A \equiv a \cdot 2^{256}\ (mod\ N)$ 和 $B \equiv b \cdot 2^{256}\ (mod\ N)$，則有：

$$ab \cdot 2^{256} \equiv AB \cdot 2^{-256} \ (mod\ N)$$

注意 2^{-1} 在模運算下稱為「模反元素」，定義為： $a \cdot a^{-1} \equiv 1\ (mod\ N)$。而 2^{-256} 代表的是 $(2^{-1})^{256}$。

假設我們將 A 寫成：

$$A = \sum_{i=0}^{255} A_i\, 2^i$$

則有：

$$AB \cdot 2^{-256} = \left(\sum_{i=0}^{255} A_i\, 2^i\right) B \cdot 2^{-256} = \sum_{i=0}^{255} A_i\, B \cdot 2^{i-256}$$
$$= \Big(\big((A_0 B \cdot 2^{-1} + A_1 B) \cdot 2^{-1} + A_2 B\big) \cdot 2^{-1} + \cdots + A_{255} B\Big) \cdot 2^{-1}$$

可以發現原先演算法中的 $ret \leftarrow ret + t$ 可以改為 $ret \leftarrow (ret + A_i B) \cdot 2^{-1}$ 這時候可以利用一個關係，將乘上 2^{-1} 的動作轉換成除以 2：

$$k \cdot 2^{-1} \equiv \begin{cases} \dfrac{k}{2} & k\ even \\ \dfrac{(k+N)}{2} & k\ odd \end{cases} \ (mod\ N)$$

注意到將乘法轉換成除以 2 的過程中，我們可以保證 $(ret + A_i B) \cdot 2^{-1}$ 的值一定維持在 $[0,\ 2N)$ 中，故可以避免迴圈中的比較運算，結果如下：

Algorithm 5. Multiplication using Montgomery Algorithm

Input: 256bit integers a, b, N

Output: $ab \cdot 2^{-256}\ mod\ N$

```
1:  function Mont(a, b, N)
2:      ret ← 0
3:      for i ← 0 to 255 do
4:          if i-th bit of b is 1 then
5:              ret ← ret + a
6:          end if
7:          if ret is odd then
8:              ret ← ret + N
9:          end if
10:         ret ← ret/2
11:     end for
12:     if ret ≥ N then
13:         ret ← ret − N
14:     end if
15:     return ret
16: end function
```

3. **結合 Montgomery 演算法和快速冪**

觀察前面的 Algorithm 2，可以發現迴圈中有兩個乘法運算，若我們將 t 初始化為 $y \cdot 2^{256}$ ，則 Mont(t, t, N) 的結果為

$$y^2 \cdot 2^{256},\ y^4 \cdot 2^{256},\ y^8 \cdot 2^{256} \ldots$$

而 Mont(ret, t, N) 則為：

$$ret \cdot y^2, ret \cdot y^4, ret \cdot y^8 \ldots$$

可以發現其中的 2^{256} 被抵銷掉了，故先將 t 初始化為 $y \cdot 2^{256}$ 後便可配合 Montgomery 演算法來進行冪運算，演算法如下：

Algorithm 6. Fast Power Algorithm with Montgomery (for RSA 256bit)

Input: 256bit integers y, d, N

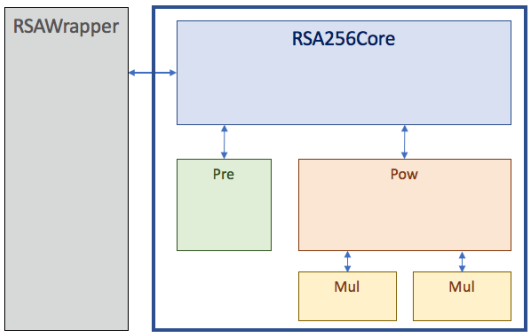
Output: $y^d \ (mod\ N)$

```
1:  function FastPow(y, d, N)
2:      t ← y · 2^256
3:      ret ← 1
4:      for i ← 0 to 255 do
5:          if i-th bit of d is 1 then
6:              ret ← Mont(ret, t, N)
7:          end if
8:          t ← Mont(t, t, N)
9:      end for
10:     return ret
11: end function
```

三、在 FPGA 上實作

（一）撰寫 RSA256Core.sv

RSA256Core 的 Block Diagram 設計如下圖：

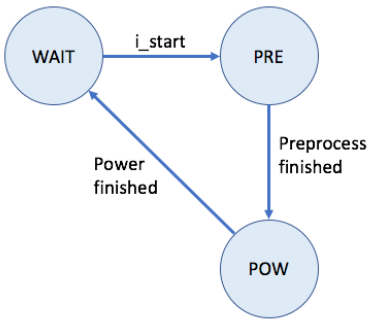


當然若有不同的作法同學也可以自行設計。

狀態圖的部分，首先設計 RSA256Core 本身

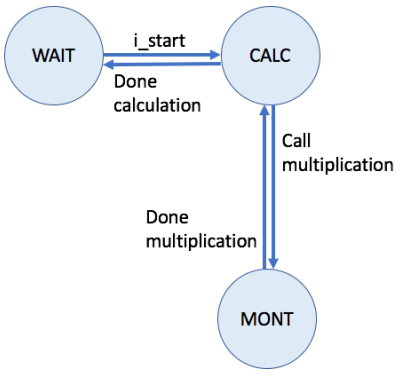
參考實驗原理中的敘述，可以設計出 Core 的三個狀態：

1. WAIT：等待 i_start 訊號
2. PREPROCESS：計算 $a \cdot 2^{256} \ (mod\ n)$，交給 Pre 模組計算
3. POWER：計算 $a^e \ (mod\ n)$，交給 Pow 模組計算



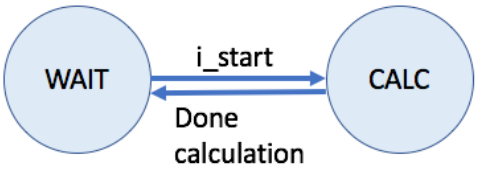
Pow 模組的功能為計算 y^d mod N（Pow 和 Pre 模組也可以合併到 Core 內）

1. WAIT：等待 i_start 訊號
2. CALC：相當於 Algorithm 6 中的迴圈
3. MONT：相當於 Algorithm 6 中的乘法運算，交給 Mul 模組計算



Mul 模組的功能為利用 Montgomery 演算法計算 $ab \cdot 2^{-256} \bmod N$。

1. WAIT：等待 i_start 訊號
2. CALC：相當於 Algorithm 5 的迴圈



Pre 模組基本上和 Mul 模組類似。

（二）Qsys 和使用到的 Altera 模組

1. 建立 Qsys 專案

詳細步驟請參考助教提供的投影片。

基本上就是套用 Altera 提供的模組並利用 Qsys 的 GUI 設定參數、連接模組間的輸出入，最後再利用 Qsys 來生成 HDL。

2. Avalon Memory-Mapped Interface

Avalon MM 整合了 Altera 提供的模組所使用的輸出入介面，詳細的工作方式請參考後文「撰寫 RSAWrapper.sv」的部分。

3. ALTPLL

ALTPLL 是 Altera 提供的 PLL (Phase Locked Loop) 模組。PLL 可以根據輸入的 clock，得到頻率更低的 clock 輸出。本實驗中輸入為 50MHz，輸出為 40MHz。

4. UART (RS232)

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

（三）撰寫 RSAWrapper.sv

撰寫 RSAWrapper.sv 前，要先了解 Altera 提供的 UART 模組使用的 Avalon 介面要如何接受／傳送。詳細內容可參考官方文件《Avalon Interface Specifications》
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf
此外關於 UART 模組的資料可以也參考官方文件：
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf

1. Avalon Memory-Mapped Interface

Avalon Memory-Mapped Interface（以下簡稱 Avalon MM）實際上提供了很多功能，不過這次實驗中使用的 UART 模組只使用了以下輸出入介面：

	方向	功能
clk	input	clock
address	output	需要 read/write 的記憶體位置 （UART 模組對應的記憶體位置參考下圖）
read	output	傳給 Avalon MM Slave 需要讀取資料的訊號
readdata	input	讀取到的資料
write	output	傳給 Avalon MM Slave 需要寫入資料的訊號
writedata	output	要寫入／傳送的資料
waitrequest	input	當 waitrequest 為 1 時，read/write 動作都暫停 （在 waitrequest 回到 0 之前 writedata 要維持不變）

換句話說：

要讀取狀態時，將 read←1、write←0、address←8。
要接受資料時，將 read←1、write←0、address←0。
要傳送資料時，將 read←0、write←1、address←4。

以下是 UART 模組的暫存器內容：

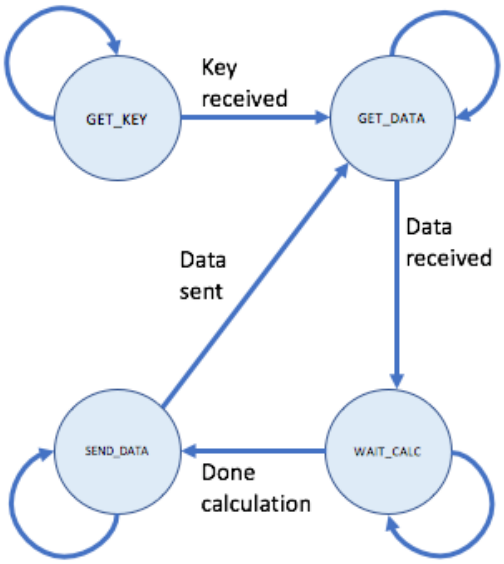
Offset	Register Name	R/W	Description/Register Bits													
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata	RO	Reserved					1	1	Receive Data						
1	txdata	WO	Reserved					1	1	Transmit Data						
2	status 2	RW	Reserved	eop	cts	dcts	1	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	control	RW	Reserved	ieop	rts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

圖中紅色方框內是本實驗會用到的部分。
當 rrdy=1 時代表 RX 已經讀取完畢，可以進行 read 操作。
值得注意的是當進行 read 操作後，rrdy 會自行變回 0。
當 trdy=1 時代表 TX 已經完成傳送資料到 buffer，可以進行 write 操作。

2. **狀態圖**

了解 Avalon MM 的工作方式後，可以設計出以下四個狀態：

1. GET_KEY：等待金鑰 (N,d) 傳送完畢
2. GET_DATA：讀取資料直到讀取完 32 byte
3. WAIT_CALC：等待 Core 計算完畢
4. SEND_DATA：等待 31 byte 的解密資料傳送完畢



（四）利用 rs232.py 進行資料傳輸

rs232.py 利用了 pySerial 這個 package 來進行 serial 輸出入，在執行前可能需要先利用 pip 安裝 pyserial。

此外有興趣的同學可以自行更改程式碼來符合自己 FPGA 功能。

四、額外功能

（一）支援連續傳檔案，而不需要手動按下 reset 按鈕

若要讓 FPGA 在接收完一個檔案後可以接受下個檔案，基本上就是要讓 FPGA 端知道是不是已讀到檔案結尾，這裡提供兩種想法：

1. 使用 Escape Character（跳脫字元）的概念

在高階程式語言中若要在字串中表達「換行」的話，可能會寫成：

“Hello world!\n”

但若真的要讓字串的內容為 “Hello world!\n” 的話，則必須寫成：

“Hello world!\\n”

相似的概念也可以用在這次實驗的檔案傳輸上。

這次實驗中設計了兩種跳脫字元，分別為：

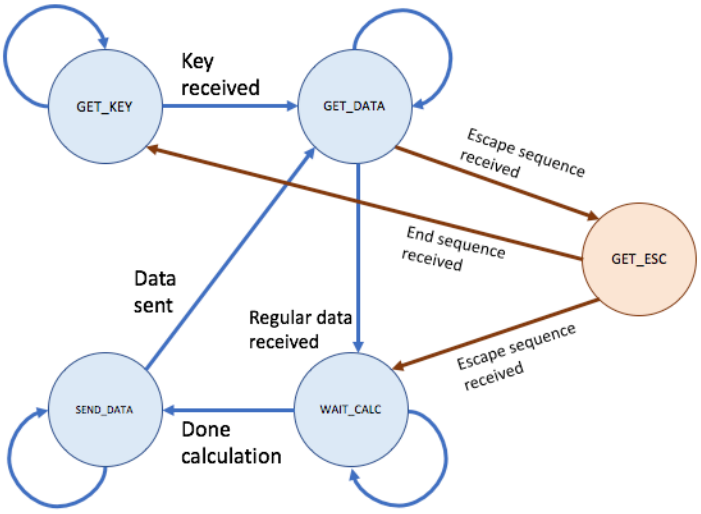
ESC：以 256bit 表示的 0　　（即 0000…0000，共 256bit）

ESC_END：以 256bit 表示的 1　（即 0000…0001，共 256bit）

首先要更改 FPGA 的 FSM 設計，在先前設計的 GET_DATA 狀態中，如果讀取到跳脫字元 ESC，則轉移到狀態 GET_ESC 讀取跳脫字元後的功能。

若在 GET_ESC 狀態中讀取到第二個 ESC，則代表原本要傳輸的資料就是 ESC，可以繼續轉移到 WAIT_CALC 狀態。若在 GET_ESC 狀態中讀取到 ESC_END，則代表讀取到檔案結尾，便轉移到 GET_KEY 狀態，準備讀取下一個檔案。

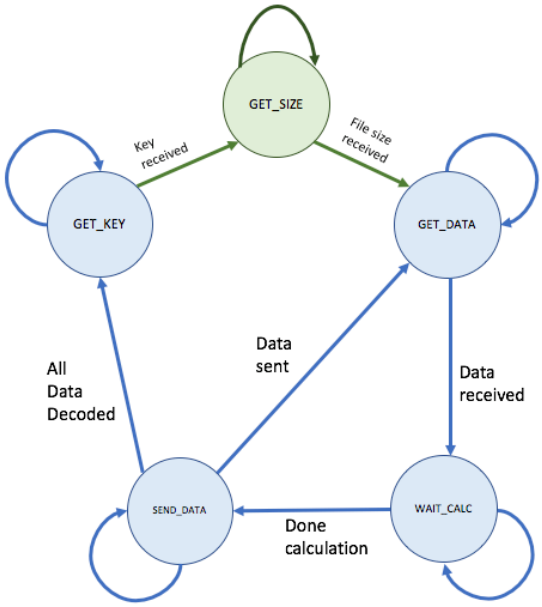
FSM 可以參考下圖：



接著還要更改 rs232.py 以配合 FPGA 的功能變更，Python 的部分同學可以根據助教原先提供的檔案，自行嘗試進行修改以滿足上述需求。

2. **在傳檔案前先傳檔案大小**

另一個做法是在接收完 KEY 之後（或之前）新增一個讀取檔案大小的狀態，這樣就可以知道該讀取多少個 byte 後可以結束，並回到 GET_KEY 狀態。



（二）支援切換不同 bit 數的 RSA key

這個部分看似複雜，但其實相當簡單，觀察前面所述的各個演算法，可以發現將 256 換成其他常見的 bit 數（如 128、512、1024）也是成立的，故其實在設計 Core 的時候可以將 bus 預先開到 1024，再根據所需的 bit 數不同調整 counter 的起始值，同時留意 Wrapper 中輸出的不再是輸出固定的 [247-:8] 而要視情況而定。

若要實作這個功能，則在測試時會需要不同 bit 數的 RSA key，這時候可以利用一些工具幫我們生成 key，相關的工具請參考文末的工具介紹。

除了生成 key 以外，也要自行修改助教提供的 rsa.py，讓他可以支援不同 bit 數的冪運算。

（三）支援顯示目前解密進度

對較大的檔案而言，解密需要的時間可能較久，這時可能會希望有個進度條給我們一點精神上的支持和信心。

先前提到可以在傳檔案前先傳一份檔案大小給 FPGA，這是由於這次實驗使用的 rs232.py 是一次先將整份檔案讀成一個字串，故一開始就能知道整個檔案的大小，這也代表著可以設計一個進度條來顯示目前解密的進度。

當然最基本的就是用 LED 來實現，有興趣的同學也可以用 2x16 螢幕甚至是彩色螢幕來實現，也可以當作螢幕操作的最基本練習。

五、常見問題

（一）我想用 for 迴圈的思路來寫 verilog 可是怎麼寫都錯怎麼辦？

值得注意的是「迴圈」的概念一般是寫在 sequential 的部分，所以常常要意識到所謂的「assignment、<=」實際上要等到下一個 clock 才會統一生效，和高階程式語言有很大的不同。故常常遇到判斷兩個值是否相等時不符預期等情況，或是 counter 差一個 clock 導致多算／少算等情況，都可以先朝這個方向思考。

（二）測試 Core 的時候發現數字小的時候答案正確，但數字大時卻不正確？

遇到這種問題時一般人的直覺大概都是 overflow，不過根據不同的程式邏輯，也可能是其他原因，例如 counter 的起始值設定有誤。

（三）在實作不同 bit 數的解密機時，支援上限 512 的時候電路正常工作，但只是將上限改成 1024 後就不正常了？

注意到當上限調整為 1024 後，整個電路的規模和線路的長度也會跟著變長，故需要的 T_{SU}、T_{COMB} 等時間也會跟著改變，故需要降低 clock 的頻率，這個部分可以藉由 PLL 來達成，有遇到這種問題的時候可以先試著調整 ALTPLL 的參數。

（四）找不到電腦端的 Serial Port 叫什麼名字？

對 Windows 而言，可以進入裝置管理員尋找 RS-232 相關的字眼。

對 Linux 而言，可以使用 dmesg 指令觀察系統的裝置連接紀錄。

六、其他工具

（一）Debugging RSA256Core.sv

利用助教提供的 tb.sv 在工作站上執行 ncverilog 進行測試。

Windows 上可以使用 MobaXTerm。

Mac 和 Linux 則使用 ssh 指令即可連上，傳輸檔案則可以利用 FileZilla。

注意在使用 ncverilog 前要先輸入 tool 2 指令。

（二）生成其他 RSA key

原本覺得這種網頁服務應該很好找，不過還真的不太好找，找了一陣子發現這個 http://www.mobilefish.com/services/rsa_key_generation/rsa_key_generation.php 可以自動生成 16 進位表示的 N, e, d，相當方便，如果有考慮實作不同 bit 數的 RSA 解密的話可以考慮使用這個工具。

（三）以二進位檢視模式閱讀檔案

在這次的實驗中常常會懷疑自己 Core 的結果是不是只是 shift 了幾個 bit 呢？所以會有需要以二進位模式閱讀檔案，這部分其實只要利用 Sublime Text 或 Notepad++ 等常見的文字編輯器即可達成。