逢 甲 大 學

資訊工程學系專題報告

Performance Analysis of RSA IPs for SOC Design

學　　生：　張家維 (四甲)
　　楊勝吉 (四甲)
　　蕭詣懋 (四丙)

指導教授 ： 王益文

中華民國九十二年十二月

目　　錄

圖表目錄 IV
第一章　前言 1
1.1　動機 1
1.2　目的 1
1.3　工作分配 2
第二章　RSA 簡介 3
2.1　Introduction 3
2.2　RSA 在網路上之應用 4
第三章　工具程式：InterNios 6
3.1　程式寫作需求： 6
3.2　程式基本原理 7
3.2.1　基本做法 7
3.2.2　Java Communications API 7
3.2.3　SREC 檔介紹 7
3.3　安裝流程 8
3.4　程式設計 10
3.4.1　主程式 UML 模型 10
3.4.2　UartFrame 物件 UML 模型 11
3.4.3　RSA 程式執行流程 12
3.4.4　資料 I/O 流程 13
3.4.5　RsaKey 產生流程 15
3.5　程式執行畫面: 16
3.5.1　功能說明 16
3.6　RSA 執行流程 18
3.7　程式功能 RAM I/O 使用 21
3.8　程式問題解決 22
3.8.1　SREC 的 checksum 22
3.8.2　Uart 傳輸不穩定的問題 22
3.8.2.1　1K 限制 22
3.8.2.2　250K 限制 23
3.8.2.3　Nios Build 無法內嵌到 Java 程式內 23
第四章　架構一：使用 RSA.C 24
4.1　Introduction 24
4.2　RSA C CODE 做法 24

4.2.1 資料結構 24
4.2.2 比較大小（compare） 25
4.2.3 移位運算（shift） 26
4.2.3.1 左移 26
4.2.3.1 右移 26
4.2.4 加法 27
4.2.4 減法 28
4.2.5 MOD 29
4.2.6 模乘法運算 30
4.2.7 模指數運算 31
4.3 System Design Framework of RSA.C 32
4.3.1 RSA.C 簡介 33
4.4 架構一的內部比較 34
第五章 使用 Custom Instruction 增加執行效率（架構二） 38
5.1 Introduction 38
5.1.1 Altera Nios Embedded Processor VS. Custom Instruction 38
5.1.2 Interface of hardware 40
5.1.3 Interface of software 42
5.2 Percent of Time Spent 分析 43
5.3 MOD 的做法 44
5.3.1 利用長除法求 C = A mod B 44
5.3.1.1 原理 44
5.3.1.2 模擬結果 45
5.3.1.3 Leonardo Spectrum 預估 RSA-WRAPPER 資料 46
5.3.1.4 優缺點 46
5.3.2 利用加速長除法求 C = A mod B。 46
5.3.2.1 原理 46
5.3.2.2 模擬結果 48
5.3.2.3 Leonardo Spectrum 預估 49
5.3.2.4 優缺點 49
5.3.3 利用超級加速長除法求 C = A mod B 49
5.3.3.1 原理 49
5.3.3.2 模擬結果 51
5.3.3.3 Leonardo Spectrum 預估 52
5.3.3.4 優點 52
5.3.4 利用暴力法求 C = A mod B 52
5.3.4.1 原理 52

5.3.4.2 模擬結果 55
5.3.4.3 Leonardo Spectrum 預估 55
5.3.4.4 優缺點 55
5.3.5 討論 56
5.4 System Design Framework of CI 58
5.4.1 RSA.C 簡介 59
5.5 架構二比較 60
5.5-1 使用 CUSTOM INSTRUCTION 的 MOD 與 MUL 60
5.5-2 使用 CUSTOM INSTRUCTION 的 MOD 與 C 的 MUL 60
5.5-3 使用 C 的 MOD 與 CUSTOM INSTRUCTION 的 MUL 60
5.5-4 使用 C 的 MOD 與 MUL 60
第六章 以硬體元件實現 RSA 系統架構（架構 3） 61
6.1 Introduction 61
6.1.1 Polling & Interrupt 簡介 61
6.1.2 Interrupt Service Routines（ISRs） and IRQ 62
6.1.3 Exception Handling 流程 62
6.2 System Design Framework 3 64
6.2.1 RSA-WRAPPER Framework 65
6.2.2 RSA-WRITE framework 67
6.2.3 RSA-READ Framework 69
6.3 RUN.C 簡介 71
6.3.1 Polling C 簡介 71
6.3.2 Interrupt C 簡介 73
6.4 Leonardo Spectrum 預估 RSA-WRAPPER 資料 74
6.5 Interrupt 與 Polling 速度比較 74
第七章 架構比較 75
第八章 心得 77
附錄一 Nios Development tool kit 簡介 79
附錄 1.1 Introduction 79
附錄 1.2 Sopc Builder 簡介 81
附錄 1.3 SDK (Software Development Kit)簡介 81
附錄 1.3.1 GNUPro Tools 81
附錄 1.3.2 Nios On-Chip Instrumentation (OCI) Debug Module 81
附錄 1.3.3 Nios OCI Debug Console 82
附錄 1.3.4 Nios SDK Shell 82
附錄 1.3.5 Nios SDK(Nios Software Development Kit) 82
附錄 1.4 Nios SDK Shell Commands 簡介 83
附錄二 硬體環境介紹 84

參考資料 86

圖表目錄

圖 2.2-1 網路狀態圖 4
圖 2.2-2 加 RSA 加解密器的網路狀態圖 5
圖 3.3-1 安裝畫面 1 8
圖 3.3-2 安裝畫面 2 9
圖 3.3-3 安裝結果 9
圖 3.4-1 主程式 UML 圖 10
圖 3.4-2 RsaSend 物件 UML 圖 11
圖 3.4-3 RSA 程式流程圖 12
圖 3.4-4 資料 Input 流程 13
圖 3.4-5 資料 Output 流程 14
圖 3.4-6 資料燒錄流程 15
圖 3.5-1 程式執行畫面 16
圖 3.6-1 選擇欲加密的檔案 18
圖 3.6-2 輸入所要產生的 RSA KEY 大小 18
圖 3.6-3 輸入記憶體起始位置 19
圖 3.6-4 程式執行結果 19
圖 3.6-5 加解密完的資料輸出到檔案，用編輯器開啟直接顯示 20
圖 3.7-1 輸出記憶體資料 21
圖 4.2-1 比較大小流程圖 25
圖 4.2-2 左移說明圖 26
圖 4.2-3 右移說明圖 26
圖 4.2-4 加法說明圖 27
圖 4.2-5 減法說明圖 28
圖 4.2-6 MOD 流程圖 29
圖 4.2-7 模乘法運算流程圖 30
圖 4.2-8 模指數運算流程圖 31
圖 4.3-1 架構一的 System design framework 32

圖 4.3-2 RSA.C 程式流程圖 33
表 4.4-1 Encode 花費時間表 34
表 4.4-2 Decode 花費時間表 34
圖 4.4-1 Encode I cache 比較圖 35
圖 4.4-2 Decode I cache 比較圖 36
圖 4.4-3 Encode D cache 比較圖 37
圖 4.4-4 Decode D cache 比較圖 37
圖 5.1-1 Custom Logic And Nios ALU 39
圖 5.1-2 Custom Instructions Tab 40
圖 5.1-3 Custom Logic 介面圖 41
表 5.1-1 User Opcode, Type & Format 的例子 42
表 5.2-1 架構 1 的 Analysis Data 43
圖 5.3-1 第一種 MOD 做法架構設計圖 44
圖 5.3-2 第一種 MOD 做法 FSM 圖 45
圖 5.3-3 第一種 MOD 做法模擬圖 45
圖 5.3-4 第二種 MOD 做法架構設計圖 47
圖 5.3-5 第二種 MOD 做法 FSM 圖 48
圖 5.3-6 第二種 MOD 做法模擬圖 48
圖 5.3-7 第三種 MOD 做法架構設計圖 50
圖 5.3-8 第三種 MOD 做法 FSM 圖 51
圖 5.3-9 第三種 MOD 做法模擬圖 51
圖 5.3-10 第四種 MOD 做法架構設計圖 53
圖 5.3-11 第四種 MOD 做法 FSM 圖 54
圖 5.3-12 第四種 MOD 做法模擬圖 55
圖 5.3-13 四種 MOD 所需 LOGIC ELEMENTS 比較 56
圖 5.3-14 四種 MOD 速度比較 56
圖 5.3-15 四種 MOD 花費時間比較 56
表 5.3-1 四種 MOD 做法比較 57
圖 5.4-1 CI 的 System design framework 58
圖 5.4-2 RSA.C 程式流程圖 59
圖 6.1-1 Exception Handling 流程圖 62
圖 6.2-1 System Design Framework 3 64
圖 6.2-2 WRAPPER Framework 65
圖 6.2-3 WRAPPER 元件設計圖 66
圖 6.2-4 RSA-WRITE 流程圖 67
圖 6.2-5 RSA-WRITE FSM 設計圖 68
圖 6.2-6 RSA-READ 流程圖 69

圖 6.2-7 RSA-READ 架構圖 70
圖 6.2-8 RSA-READ FSM 設計圖 70
圖 6.3-1 POLLING C 程式流程圖 71
圖 6.3-2 INTERRUPT C 程式流程圖 73
表 7-1 架構時間比較表 75
表 7-2 單位資料大小 76
表附錄 1.4-1 Command Description 83
圖附錄 2-1 Nios Development Board Components 85
圖附錄 2-2 Stratix 實體圖 85

第一章　前言

1.1　動機

通常為了解決一個應用上的需求，不論是用何種方法，都會有利有弊，在尋求一個問題的解答時，衡量效益取其最大成效就是最重要的問題，電腦上的應用，如果說純軟體解跟硬體解均可以達成目的，那又該那如何選擇呢？在成本跟效能之間，是要選擇軟體還是硬體？亦或者介於中間呢？

1.2　目的

分析軟硬體之間效能跟成本，為了能徹底分析之間的差異，也必須熟習embedded system的架構，藉由改變架構、改變CPU、增加元件，測量數據，並且瞭解其中差異。

1.3 工作分配

	張家維	楊勝吉	蕭詣懋
實作架構一			●
RSA程式改寫	●		●
架構二設計	●	●	
架構二控制程式	●	●	
架構三設計	●	●	
架構三控制程式		●	
JAVA程式構想	●	●	
JAVA程式撰寫	●		
報告撰寫	●	●	●

表1.3-1　工作分配表

1.4 未來展望

RSA因為是利用在網路加解密，所以我們要把他改成透過網路線做加解密，一來傳送資料速度比較快，再來直接在版子上實現一個web sever，使用者可以直接從網路跟版子做連結取得版子的資料，而資料皆經由RSA做加解密，最後證明硬體加密的速度足以應付大量的client端的資料要求。

第二章　RSA簡介

2.1　Introduction

公開金鑰密碼系統：

有別於傳統式的密碼系統收發雙方需將加解密金鑰秘密保存的方式，公開金鑰密碼系統(public-key cryptosystem)加密所使用的金鑰可以公開給所有使用者知道而不會影響到系統的安全性，當然解密金鑰一定要秘密保存。

在公開金鑰的密碼系統中，每位使用者的加密金鑰是公開給系統的其他所有使用者的，因此破密者所能掌握的資訊也相對的比起傳統式的密碼系統為多。

RSA 公開金鑰密碼系統為 1978 年美國麻省理工學院三位學者 Rivest、Shamir 以及 Adleman 等三位學者所研究發展出來的一套密碼系統，它是以因數分解為基礎來設計的一套密碼系統。

而金鑰要如何產生呢？

隨機找出兩個夠大的質數 p、q。

1.計算 p、q 的乘積 n；(p-1)、(q-1) 的乘積 $\phi(n)$。

(n=p*q；ϕ (n) = (p-1) * (q-1) 。)

2.隨機找出一個滿足 gcd(e, ϕ(n))=1 之整數 e。

3.計算 d，d 滿足 ed≡1(mod ϕ(n))。

4.以 e、n 為公開金鑰，d 為秘密金鑰。

有了金鑰之後要如何加解密呢？

若欲送一資訊給某一使用者，必須取得該使用者之公開金鑰 e 與 n，接著將資訊分成數個區段，每個區段之長度比公開金鑰 n 之長度略小以確保明文及密文範圍都在[0,n-1]之內。

加密公式：C=E(M)≡M^e (mod n)，0≦M＜n。

當使用者收到密文之後，使用秘密金鑰（d）透過下面公式解密。

解密公式：M=D(C)≡C^d (mod n)，0≦C＜n。

2.2 RSA 在網路上之應用

如圖 2.2-1，當 PC1 想跟 PC2 拿某資料 DATA a 的時候，他可以在跟 PC2 要求資料的時候送出 public key（當然 private key 由他自己保管）跟 N，而 PC2 收到 PC1 要求以後，會將 DATA a 用 public key 做模乘法產生出被加密的密文，由於單只有 public key 跟 N 是很難反推出 private key 的（只要 N 夠大），如此一來就可以保證在網路上傳送的資料別人無法得知內容。

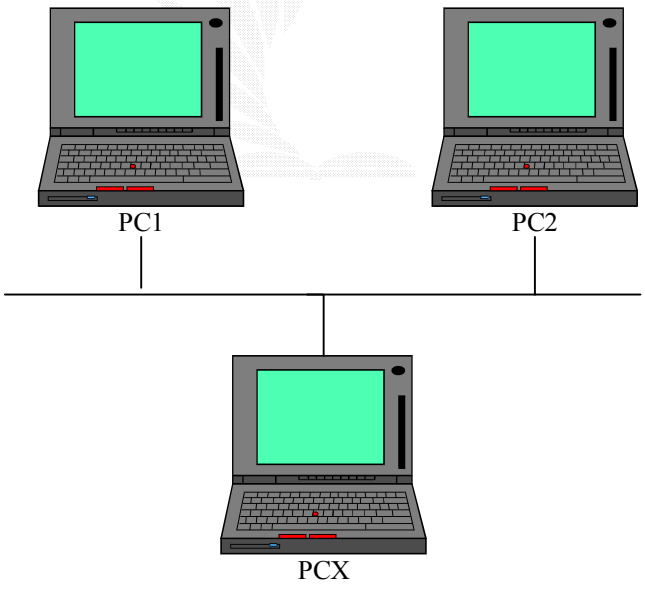


圖 2.2-1　網路狀態圖

為何需要硬體？

RSA 加解密演算法的精髓在於模指數運算，一般模指數運算均是利用連續乘法完成，而乘法在 compiler 下面組譯成 assembly code 的時候，是採用移位相加的方式完成，在 32bit width 的

CPU 下面，內建有 32bit 的加法跟移位器，但是超過 32bit 的話，就必須程式去做自動處理，每次均只能以 32bit 為單位作運算，而 RSA 就是要求數值越大越好，光是 64bit 相加在 C 裡面就必須至少花掉兩倍的 32bit 加法時間還有中間處理 carry 的時間，指令變長成好幾倍，何況還要做指數運算，速度嚴重不足，造成 RSA 無法在網路上做即時加解密的應用。

如圖2.2-2，由於RSA加解密演算法花費大量的時間，所以需要硬體來幫忙做加解密，硬體理論上的速度應該至少達到軟體的百倍效率，所以我們希望藉由這速度提升，能使RSA真正使用在網路加解密上。

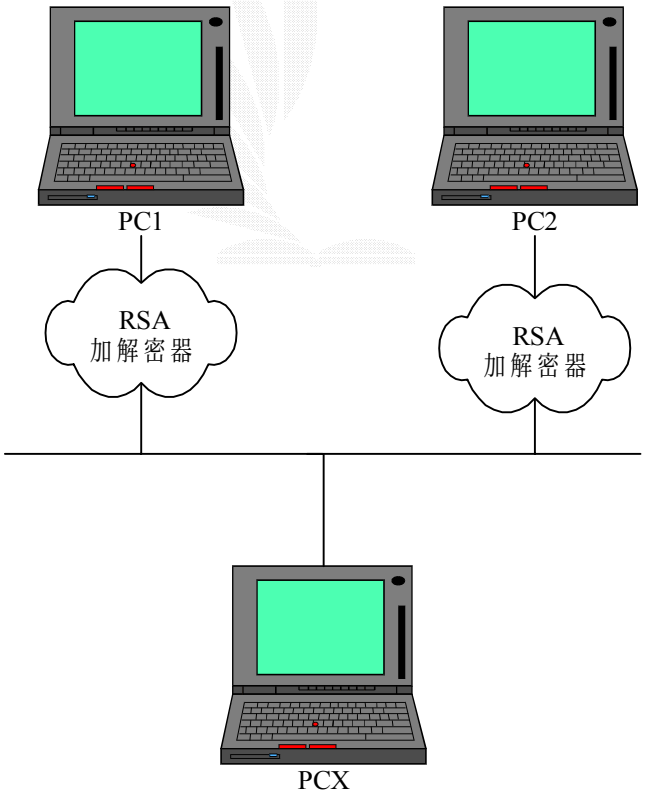


圖 2.2-2　加 RSA 加解密器的網路狀態圖

第三章　工具程式：InterNios

3.1　程式寫作需求：

a．正規的做法太過繁雜

b．檔案讀寫問題

由於 Nios SDK 對於 STRATIX 這塊版子的支援並不方便，全部控制都在 command mode 下面進行，指令複雜，並不容易上手，再來它是在版子上的系統執行 C Code，也就是說限制在系統能用的資源，在沒有硬碟的情況下，無法做出檔案讀寫的動作，沒有硬碟，那就只能把記憶體當作儲存媒介了，（FLASH 對版子來說也是當作記體體使用），在 C 程式裡面可以藉由記憶體指標位置來對記憶體做寫入。(這是 NIOS 的 C COMPILER 所針對版子所做的特殊記體處理方式，一般的 C 不能這樣做)　。

如果想對一整個區塊的記憶體做寫入值的動作，就是非常麻煩的事情，在文件教學裡面，我們必須先寫出一個記憶體位置&值的檔案(.mif 檔，一個記憶體位置對應一個值)，然後再經由 nios-covert 這道轉換指令，把檔案轉換成可以燒錄到版子的 srec 檔，接著在 nios-terminal 模式下決定你所要燒進去的記憶體起始位置，然後才能進行燒錄，而且燒錄完還要把記憶體起始位置跳回程式起始的記憶體位置，這樣才能開始你的程式，再來，即使對記憶體寫入成功，也無法將記憶體的結果轉存到電腦上，只能在螢幕上觀看，這對開發硬體來說十分不便。

3.2 程式基本原理

3.2.1 基本做法

由於 Stratix 是透過 RS232 介面跟 PC 做溝通的，如果知道中間的傳輸格式就可以對版子做出控制，顯示版子的訊息，模擬出類似 nios-sdk 的功能，而且在程式碼可以自己控制的情況下，可以對記憶體做直接寫入的動作，或者把版子的資料取出到 PC 上。

3.2.2 Java Communications API

由於在 windows 保護模式的關係，想直接控制 rs232 並不簡單，如果要直接從低階 I/O 來控制 Serial port 的話,又會涉及到 interrupt 的問題,在 Windows 處理 interrupt 也是相當麻煩的，還好有 Java communications API 可以專門用來解決串列通訊的問題。Java communications API 主要是用來在各獨立平台上寫通訊程式用的，它包含了 RS232 serial ports 跟 IEEE 1284 parallel ports 部分，是這個程式的通訊埠的介面部分，這個程式主要就是利用 Java Communications API 取代掉原本 Nios SDK Shell 應該做的事情，雖然聽起來有點不可能，但是實作起來也沒有說困難到哪裡去，熟知他的傳輸格式大概就可以猜到應該送什麼訊息給版子。

3.2.3 SREC 檔介紹

SREC全名叫做S-Record File Format，內容主要是將資料轉換HEX格式方便連續燒錄到版子上，所以一般大量資料傳輸會藉由

SREC檔。而nios-build Compiler程式也是產生SREC檔來燒到記憶體做執行動作。

格式如下：

S<*type*><*length*><*address*><*data*><*checksum*>

<*type*>代表0~9的數字，至於各數字代表的意義主要是跟記憶體address長度有關。

<*length*>是兩個 hex字元，代表這一串S-Record的長度。

<*address*>長度不一，主要依照<type>的定義決定，有4、6 or 8個字元長度

<*data*> 資料區，每hex字元組為單位放在這裡。

<*checksum*> 8bit的檢查碼。

3.3 安裝流程

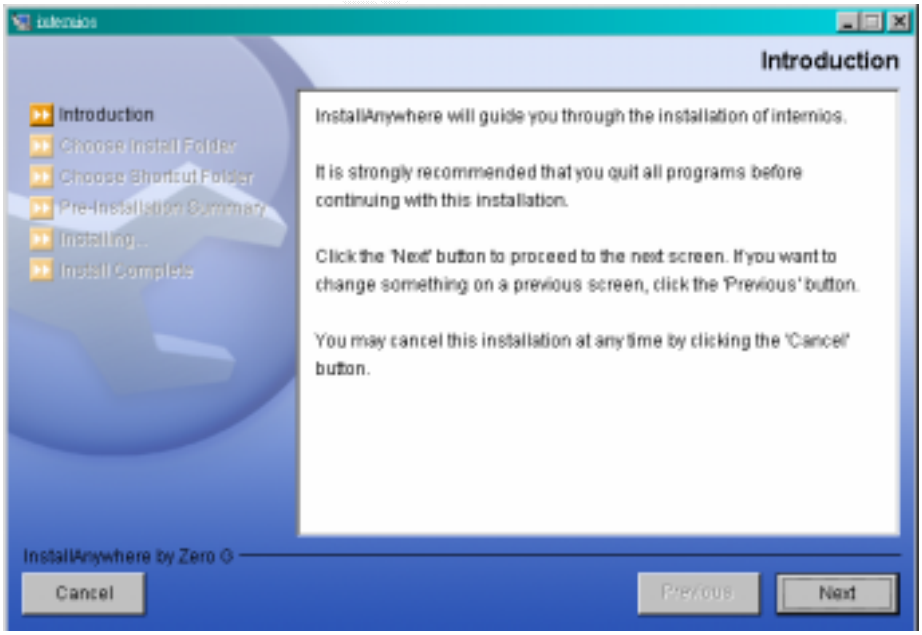


圖 3.3-1 安裝畫面 1

進入安裝畫面以後只需要一直按 Next，如果之前有安裝過，會自動覆蓋。

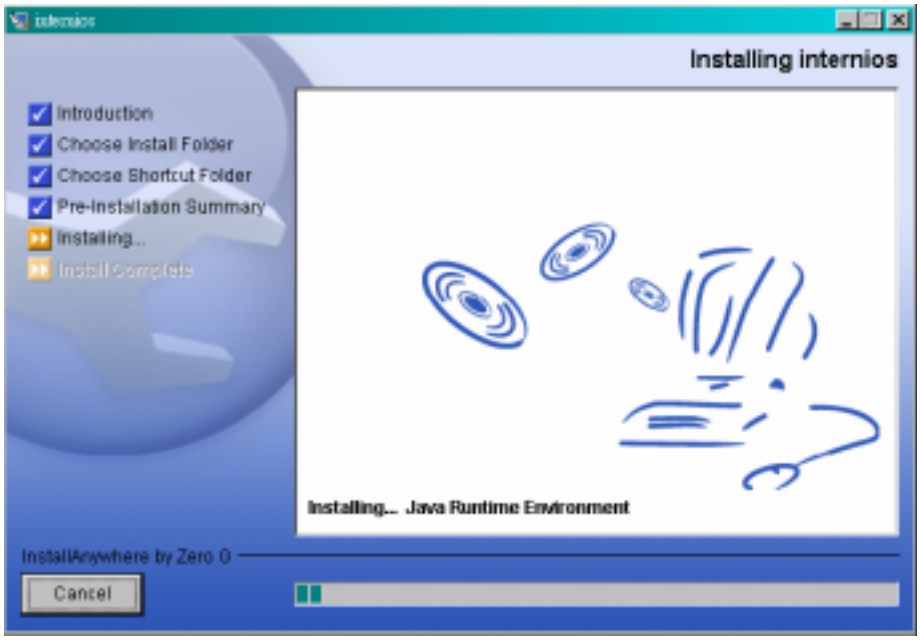


圖 3.3-2　安裝畫面 2

如果系統沒有安裝 Java Virtual Machine，安裝程式會自動安裝。

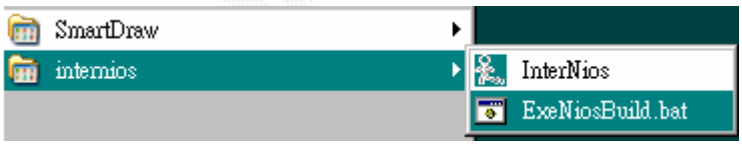


圖 3.3-3　安裝結果

安裝完成會在程式目錄出現 InterNios 資料夾

3.4 程式設計

3.4.1 主程式UML模型

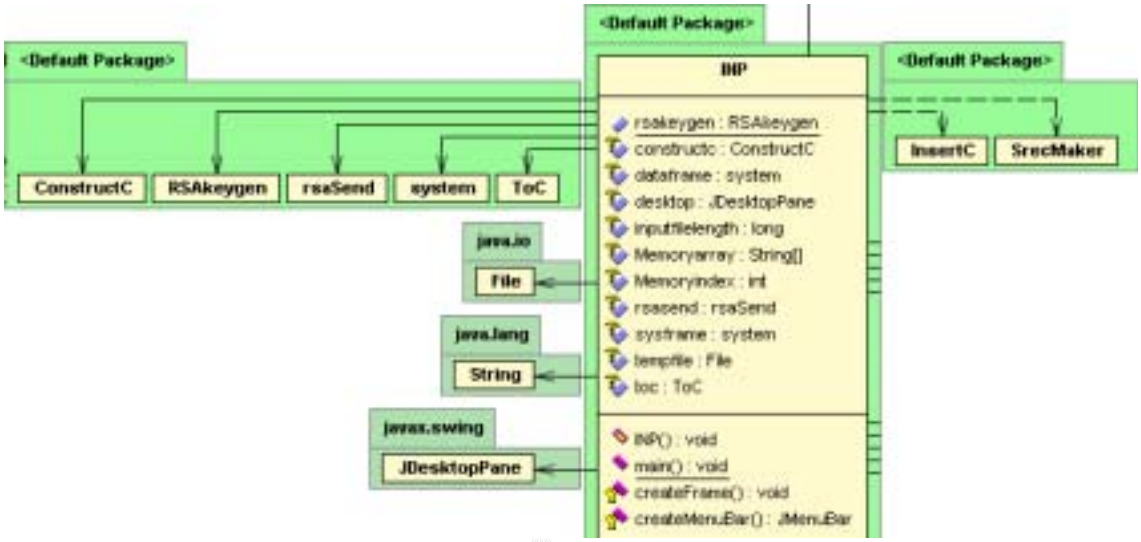


圖3.4-1 主程式UML圖

主程式主要負責user interface，也負責連結各物件。

3.4.2 UartFrame物件UML模型

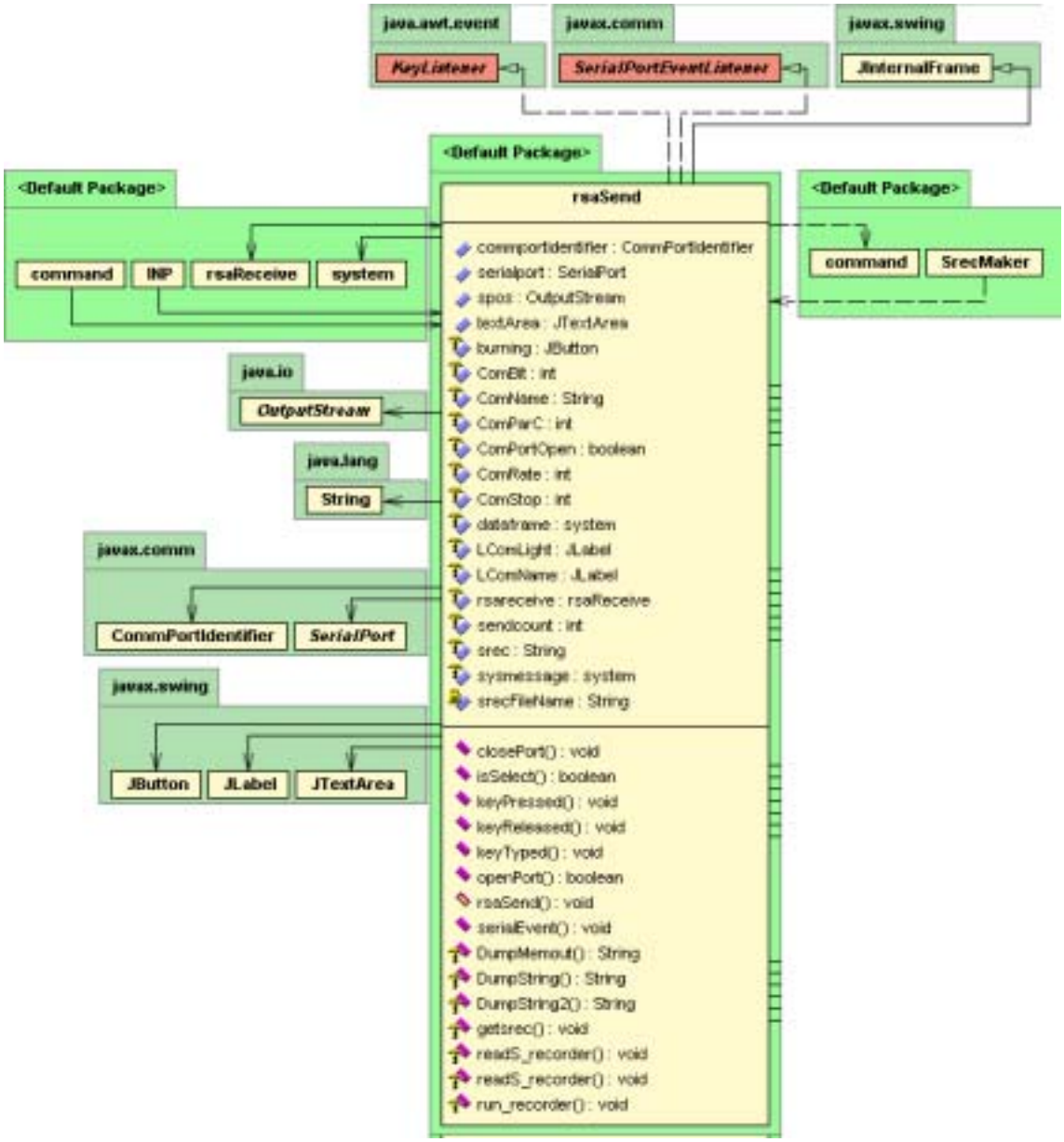


圖3.4-2 RsaSend物件UML圖

RsaSend這個物件是這個程式的重心所在，我把所有關於RS232傳輸的工作全部都交他處理，所以他是繼承java.comm 下來的一個物件，本身實做thread的runable方法，所以他跟主程式是同時在執行的，不斷的監聽serial port是否有資料進來，另外這物件也是需要同時跟許多物件做通訊，所以跟其他物件的關係也

是最複雜，光從上面UML圖跟其他package的連結就可以知道。

3.4.3 RSA程式執行流程

這程式提供的功能主要分成兩個部分，一個是設計給 RSA 使用，一個是用來對版子做資料 I/O 用的。

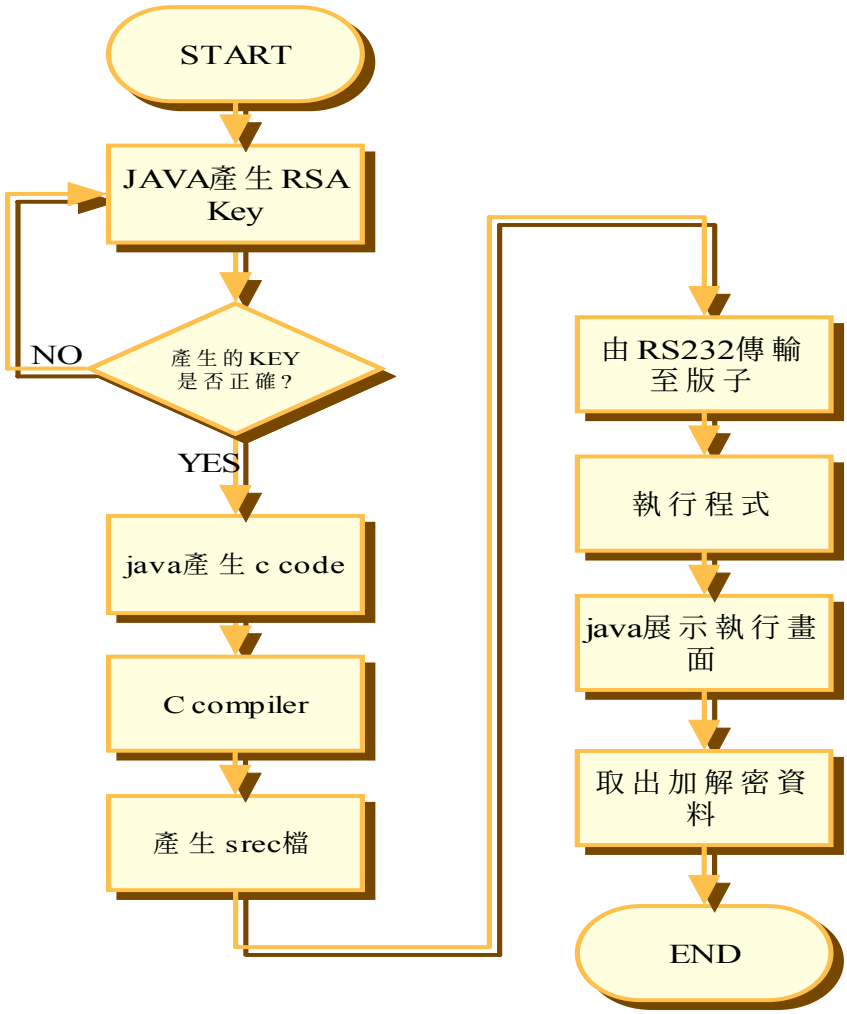


圖 3.4-3 RSA 程式流程圖

RSA 程式流程是專門設計給這三個架構使用的，幫助將加解密資料放入跟取出，並且提供我們快速測試加解密的正確性。

3.4.4 資料I/O流程

這功能專門加強版子對 PC 的 I/O。

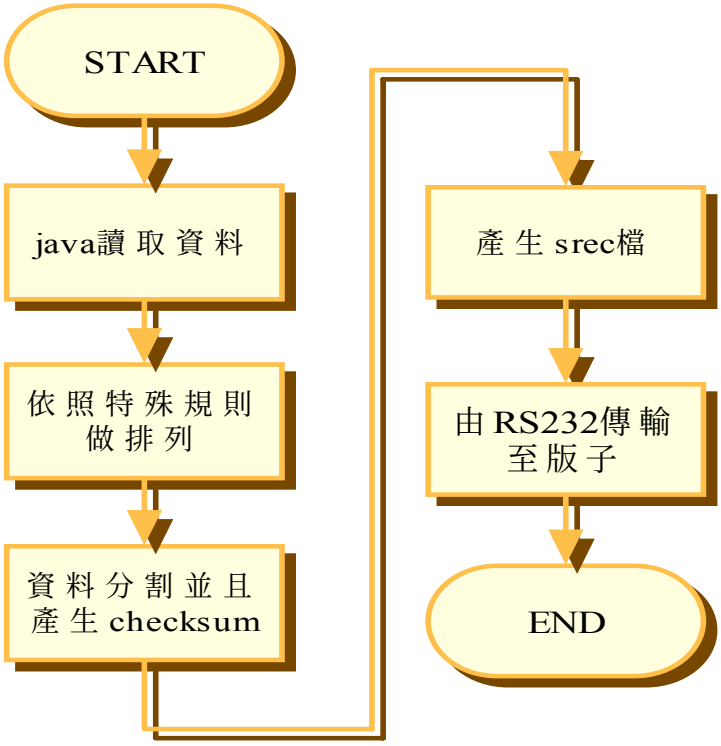


圖 3.4-4　資料 Input 流程

InterNios 可以幫使用者把檔案轉換成 SREC 檔然後傳輸到版子上，而使用者只需要用滑鼠點幾個步驟即可，但仍然有幾個要注意的地方，第一個是 Nios 內的記憶體排列方式不是依序排列，第二個是資料分割後必須產生 checksum，知道這些以後才不會燒資料後卻產生跟預期不合的情形。

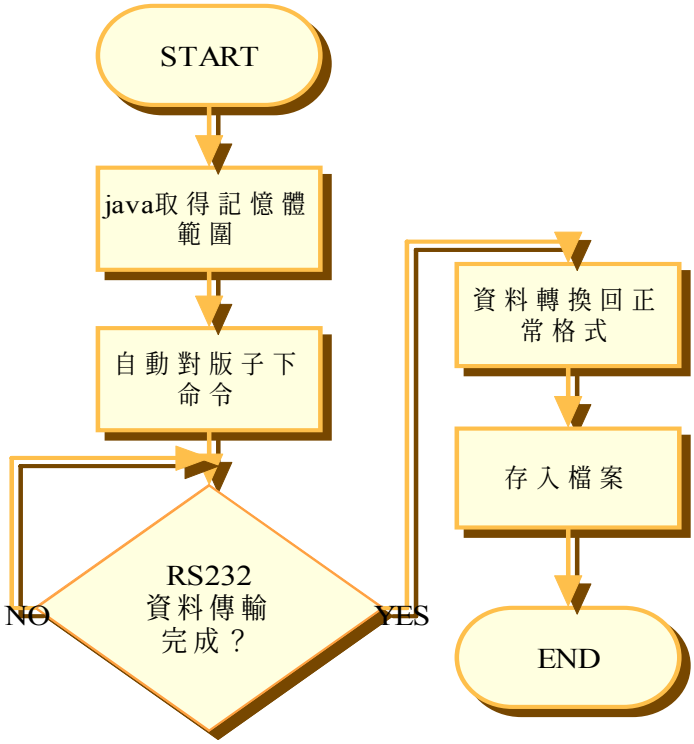


圖 3.4-5　資料 Output 流程

資料 Output 對使用者來說只要輸入所要輸出的 address 範圍即可，InterNios 會自動將記憶體的值輸出到檔案。

3.4.5 RsaKey產生流程

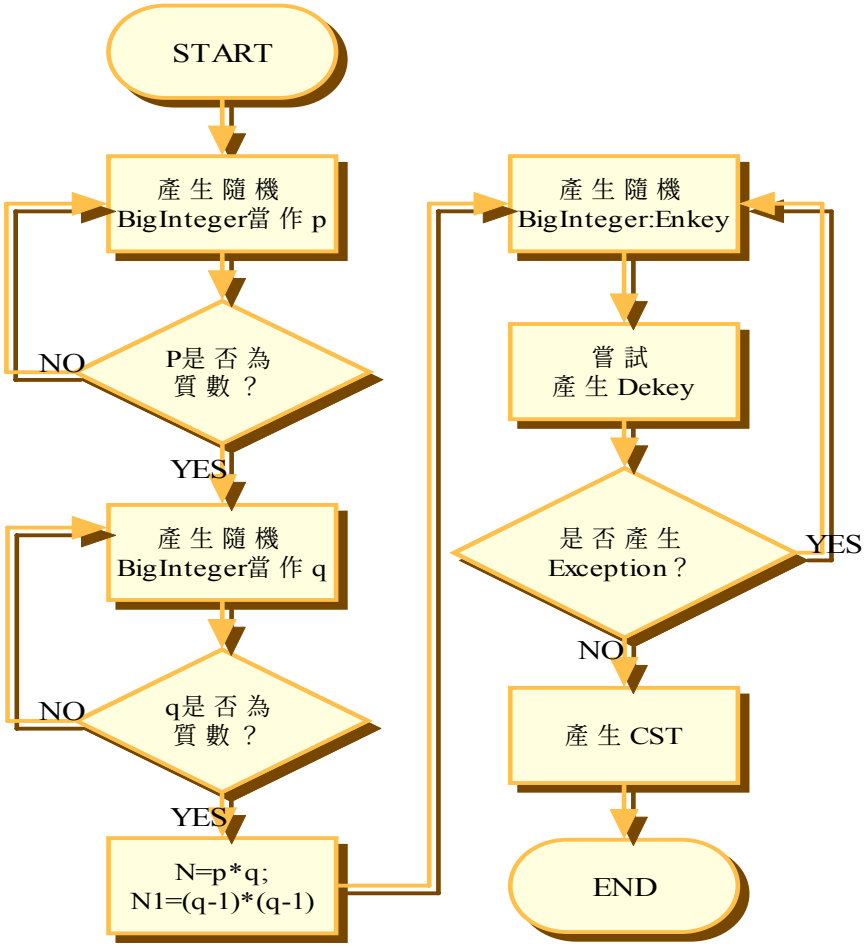


圖 3.4-6 資料燒錄流程

這是有關於 RSA 演算法的 KEY，基本上程式只是依照 RSA KEY 的產生過程，改成用程式碼產生。

3.5 程式執行畫面:

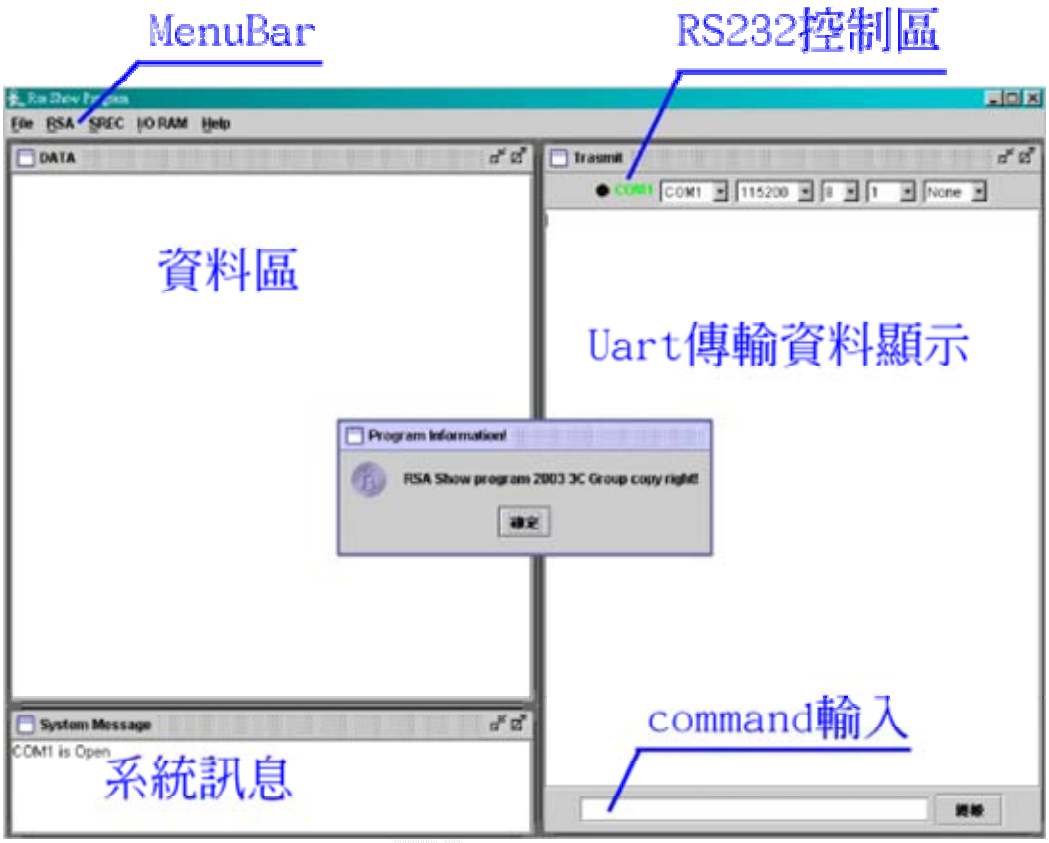


圖 3.5-1 程式執行畫面

3.5.1 功能說明

<Menu>File 下：

INPUT	：用來選擇輸入檔案。
Construct C	：呼叫程式去修改 C 程式碼的 RSA key 值。
Save Encode	：把加密過的資料輸出到檔案。
Save Decode	：把解密完的資料輸出到檔案。
Direct Link to C	：直接把輸入檔案的內容放到 C 程式碼，建議不用。
Exit	：離開程式。

<Menu>RSA 下：

RSA Auto Run ：自動展示 RSA 程式流程。

Generate RSA keys ：隨機產生一組 RSA key 值供程式使用。

<Menu>SREC 下：

Choice SREC File ：選擇 SREC 檔以供燒錄

<Menu>RAM I/O 下：

RAM Input ：直接對版子上某段 RAM 做寫入動作。

RAM Output ：直接把版子上某段記憶體位置的值 DUMP 到程式暫存區。

Output File ：把程式暫存區的值存到檔案。

3.6 RSA 執行流程

程式有設定 auto run 的功能，直接執行 auto run 即可，首先選擇要做加密的檔案。

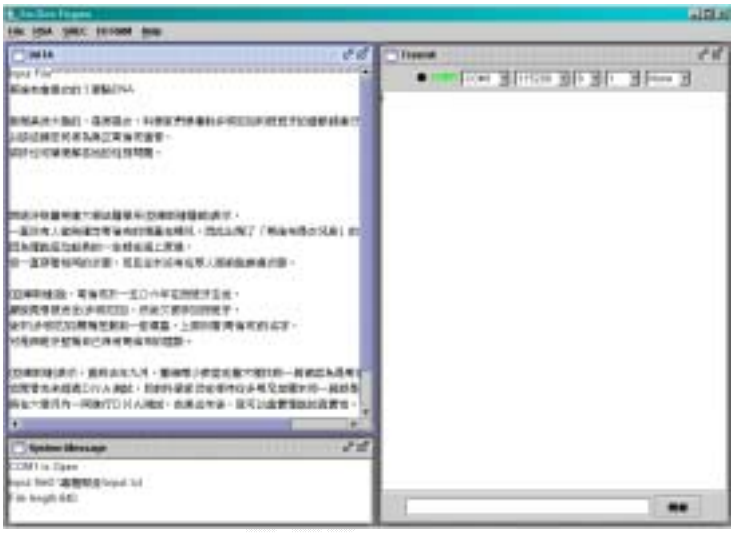


圖 3.6-1 選擇欲加密的檔案

完成之後，輸入你所要產生的 RSA KEY 大小

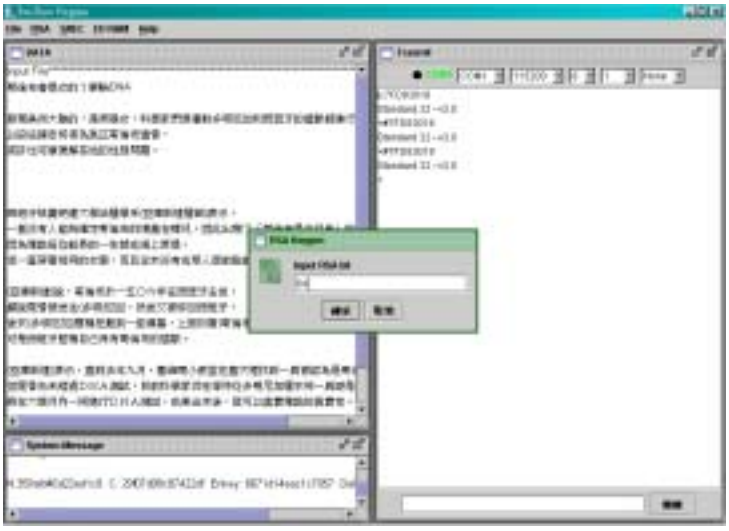


圖 3.6-2 輸入所要產生的 RSA KEY 大小

接著把剛才 INPUT 的資料放到記憶體內，輸入記憶體起始位置

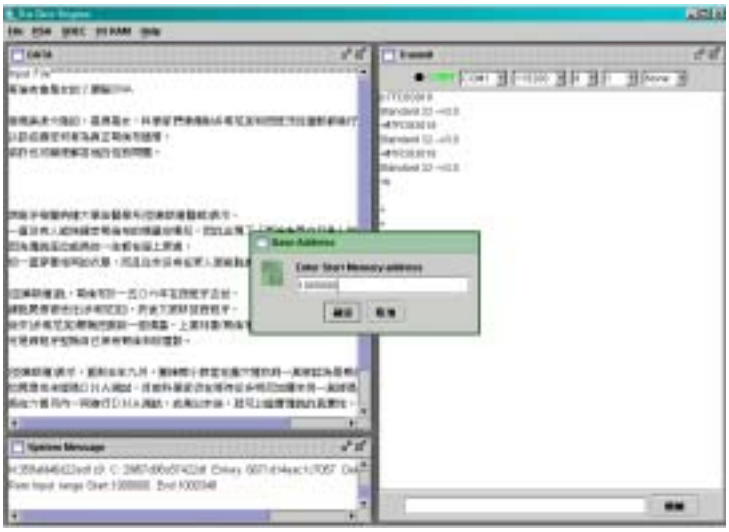


圖 3.6-3　輸入記憶體起始位置

然後選擇程式所要執行的 C code，程式將自動將 RSA KEY 跟檔案資訊加到 C 檔案內，接著就可以執行 ExeNiosBuild 了，執行完後可以按燒錄，就可以看到程式結果。

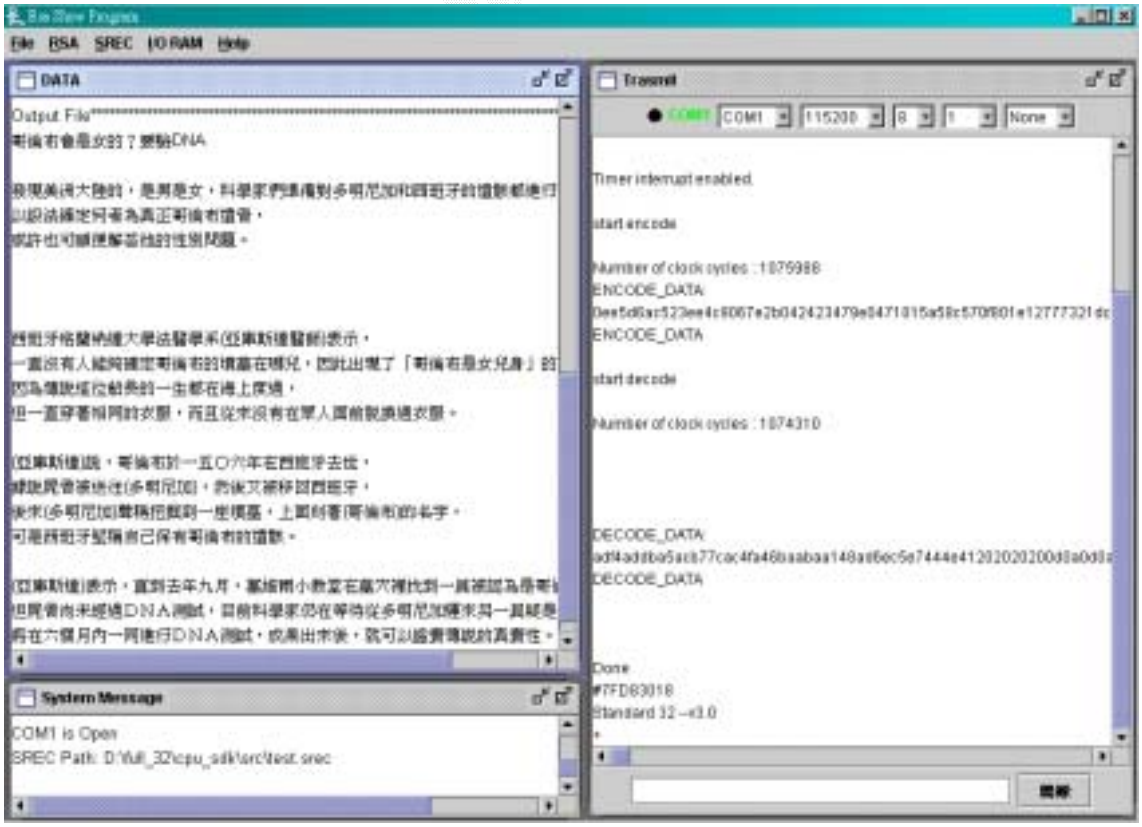


圖 3.6-4　程式執行結果

Transmit 欄會顯示程式輸出畫面，並且顯示耗費時間。DATA 欄會顯示 INPUT 跟 OUTPUT 結果 (以文字表現)，接著把 Encode 跟 Decode 資料輸出到檔案。

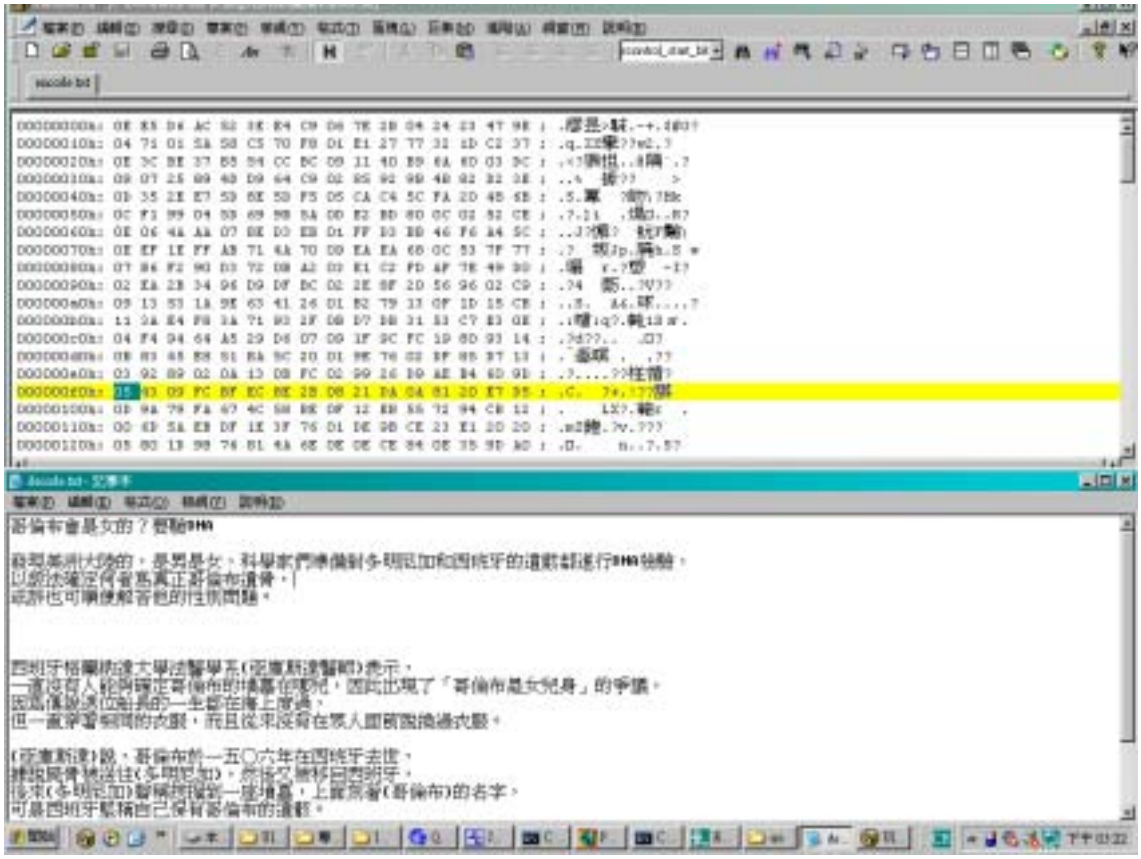


圖 3.6-5　加解密完的資料輸出到檔案，用編輯器開啟直接顯示

比較 INPUT 跟 OUTPUT 檔案的 HEX 碼，結果顯示經由 RSA public key 加密後的檔案由 RSA private key 解密回來是一樣的。

3.7 程式功能 RAM I/O 使用

這功能是專門為能跟 NiosShellSdk 溝通的版子所製作的，幫助使用者快速在記憶體內放入跟取出測試資料。輸入資料的方法在前面簡介過了，而輸出資料只要輸入你要的範圍就可以輸出了

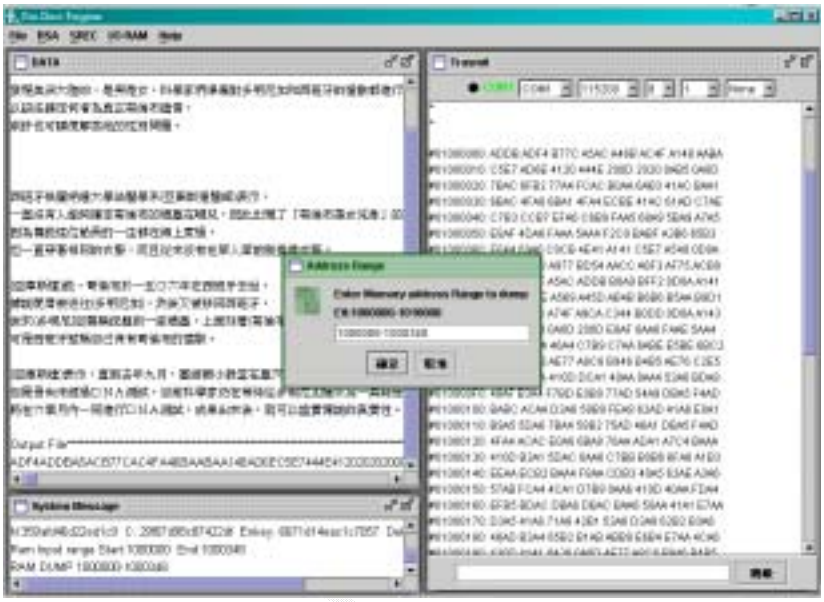


圖 3.7-1 輸出記憶體資料

3.8 程式問題解決

3.8.1 SREC 的 checksum

這是產生 SREC 檔的時候的難題，因為每產生一行就必須產生 checksum，document 內並沒有講解怎麼產生 checksum，一開始猜測既然是 sum，就把全部都加在一起，可惜並不對，最後是去找 compiler 內的 source code 才知道。Checksum 是扣除掉檔頭，然後把其他的資料全部加在一起，最後再把值反相。

例如:

S11800000100020003000400050006000700080009000A000BA5

18 長度　　checksum+=0x18;

15 address 位置　　checksum+=0x15;

checksum+=0x0c,0x0d,0x0e,0x0f,0x10,0x0f,0x0e,0x0d,0x0c,0x0b

最後 checksum=255 - (checksum & 255)

3.8.2 Uart 傳輸不穩定的問題

3.8.2.1 1K 限制

一開始程式只要接收資料量超過大約 1K 就會出錯，後面的資料會有一段錯誤，沒多久又正常的情形，最後發現是關於 pc receiver buffer 太小的問題，一開始是很單純一直加大 buffer，後來想到更好的做法，就是針對 receiver buffer overflow 做 exception handler，當觸發條件的時候，馬上呼叫 buffer. flush 方法，他會把 buffer 清空以後才繼續動作。

3.8.2.2 250K 限制

解決 1K 限制以後，又發現傳送超過 250k 左右資料量，會導致整個傳輸停擺，由於後來沒什麼時間去測試了，所以這問題並沒有去解決，但已經推測到原因，也想出解決的辦法，就是除了 pc 端的 buffer 會 overflow 外，on board 上的 buffer 只有 1024byte，即使硬體運算速度很快，可能也趕不上 Uart 115200 的傳輸速率，所以檔案過大依然會 overflow，解決方法就是在 PC 端的程式做限制，每傳送多少 bit 出去給 board 就讓執行序 sleep 一下再繼續傳。

3.8.2.3 Nios Build 無法內嵌到 Java 程式內

整合 Nios Build 這個 compiler 程式一直是我很想做的事情，雖然 Java 可以用 Runtime.getRuntime().exec()的方式執行外部程式，但是 Nios Build 其實是一個 script 檔，必須在 cygwin 的模擬 Linux 下面執行，這就是最困難的地方，不是單純的一個外部程序就很難鑲進 Java 裡面，把 system.io 導向到 java 並不困難，但是導向 cygwin 的 output 就很困難，對於這個問題，花了很多時間也還是沒解決，目前為止僅能用變通的方法暫時解決，不過我希望以後能解決這問題。

第四章　架構一：使用 RSA.C

4.1　Introduction

當我們在開發程式時會先寫好一個 C 檔然後透過 compiler(例如 turbo c)把程式編譯成機器碼然後交給 CPU 去執行這是一般以 pc 為平台的方式使用 X86 架構的 CPU。

如今我們使用Nios Development Board, Stratix Edition作為開發平台建構出一個電腦系統並且完成一個RSA加解密的job。

4.2　RSA C CODE做法

4.2.1　資料結構

由於 c 語言沒有支援大數（big integer）運算，所以必須自定資料結構用來做大數的運算。

程式是做 64bit 的整數運算所以定義的下面的資料結構：

typedef unsigned long LINT[3]。

例如：

n=0x33F9604D853CEE57，則

LINT n；

n[0]=2；

n[1]=0x853CEE57；

n[2]=0x33F9604D；

其中 n[0]=2； 表示有兩個 32bit 的資料，也就是要做 64bit 的運算。

4.2.2 比較大小（compare）

要比較兩個大數的大小先比較長度，長度大的就比較大。當兩數的長度一樣時，就依次比較最高權位的數字大小。

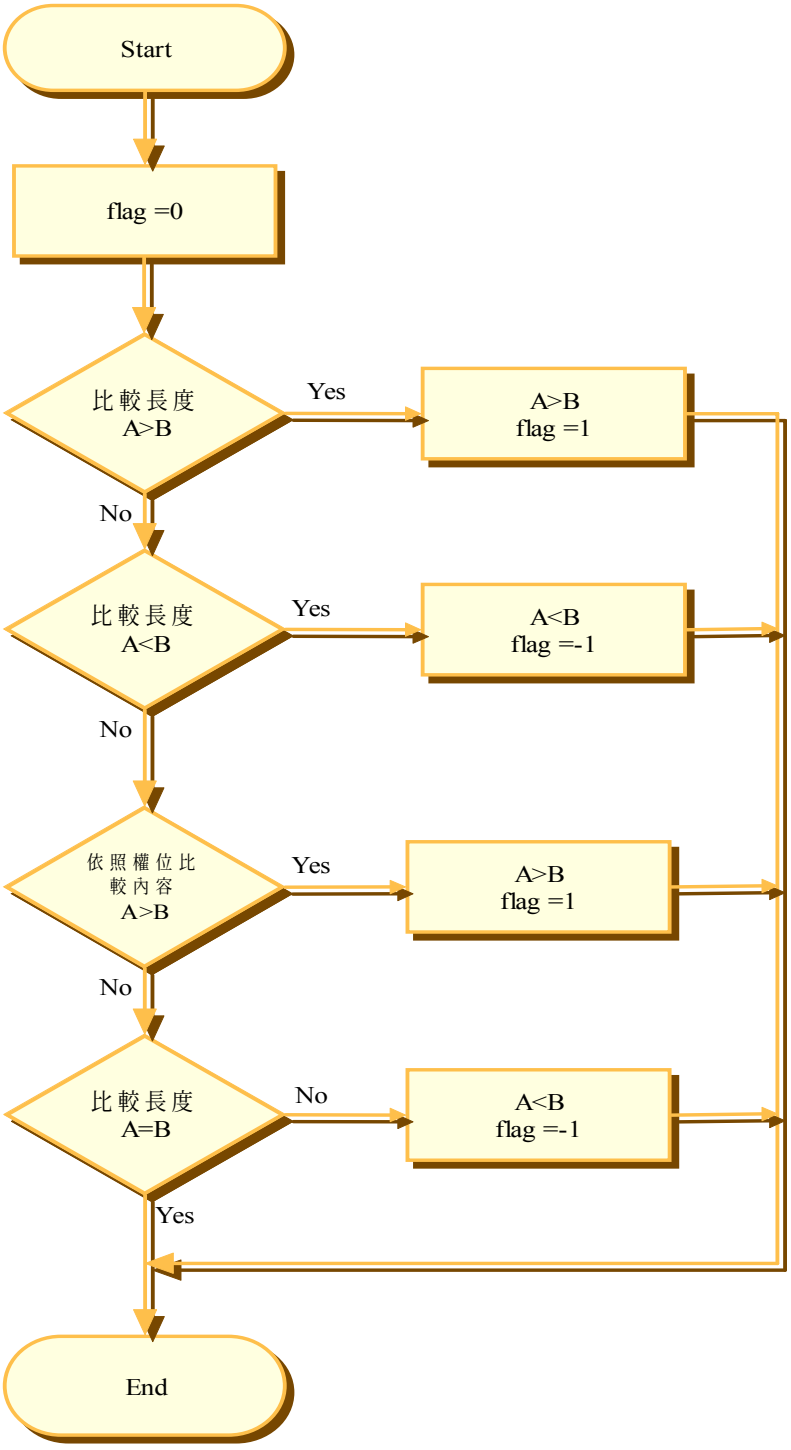


圖 4.2-1 比較大小流程圖

4.2.3 移位運算（shift）

4.2.3.1 左移

將每一個大整數中的每一長度整數組成份子由最低位元開始皆向左移一個位元。由於整數是由兩個 long 組成，所以在做 shift 時 n[1]的最高 bit 要移到 n[2]的最低 bit。

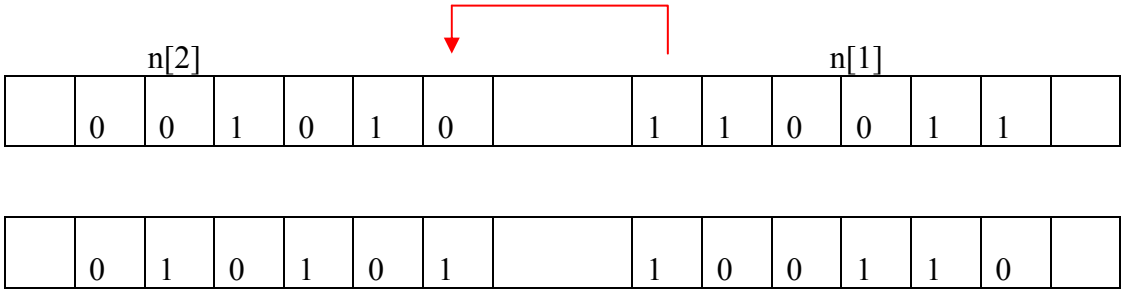


圖 4.2-2 左移說明圖

4.2.3.1 右移

將每一個大整數中的每一長度整數組成份子由最低位元開始皆向右移一個位元。由於整數是由兩個 long 組成所以在做 shift 時 n[1]的最低 bit 要移到 n[2]的最高 bit。

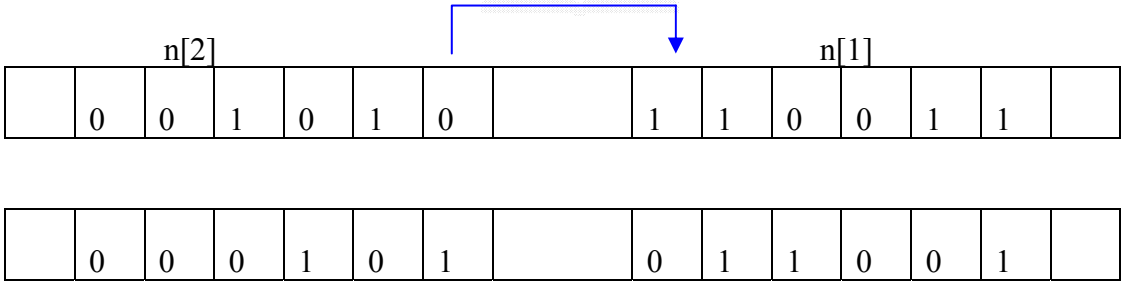


圖 4.2-3 右移說明圖

4.2.4 加法

如同一般的加法先從低權位(n[1])開始加，並且注意進位的問題。當有進位時要在高權位加 1 也就是 n[2]+1。

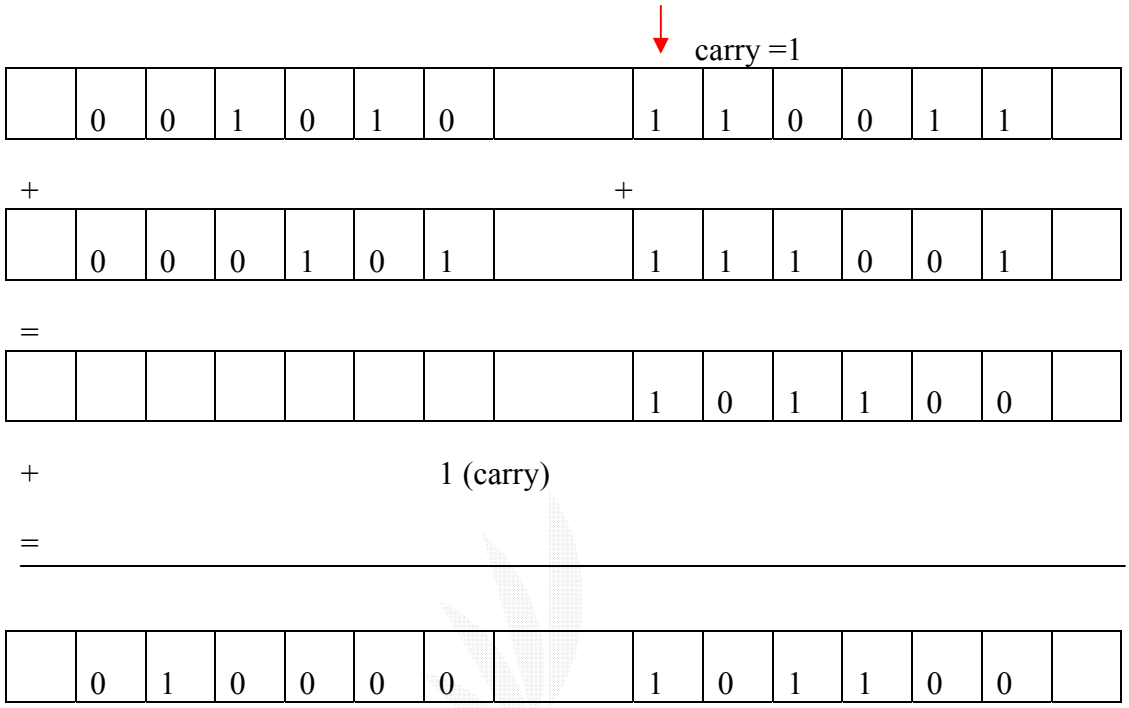


圖 4.2-4 加法說明圖

4.2.4 減法

如同一般的減法先從低權位（n[1]）開始相減，並且注意借位的問題。當有借位時要在高權位減 1 也就是 n[2]-1。

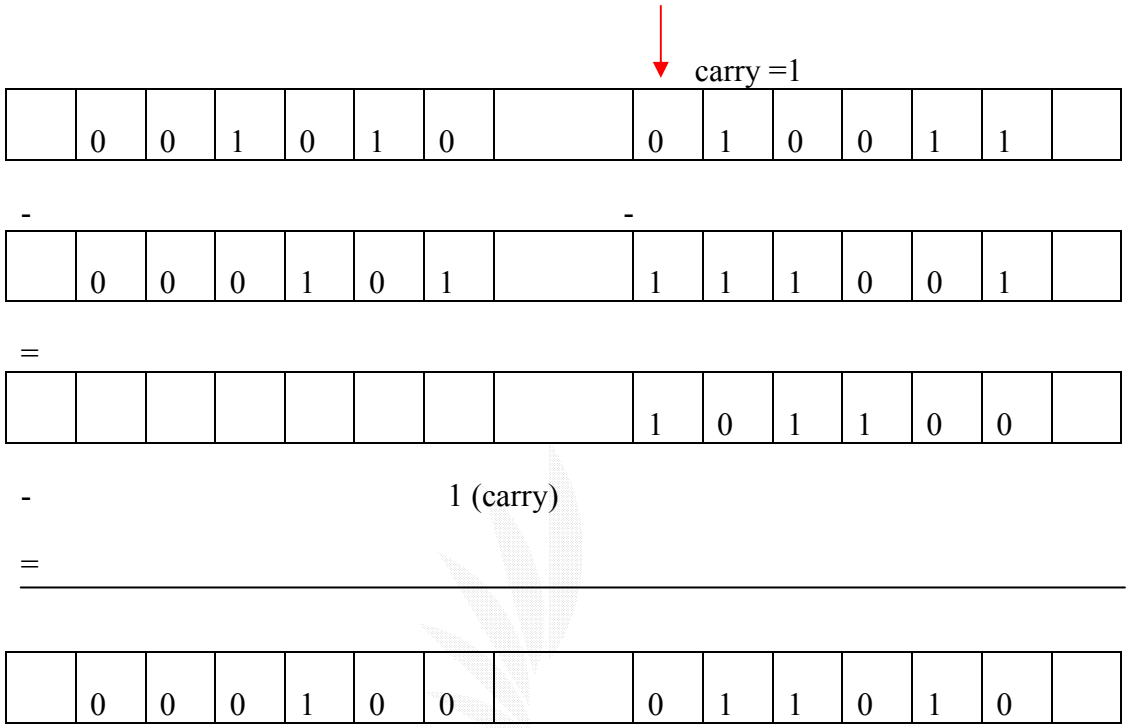


圖 4.2-5 減法說明圖

4.2.5 MOD

利用長除法求餘數。Mod (a,b) , a = a mod b。

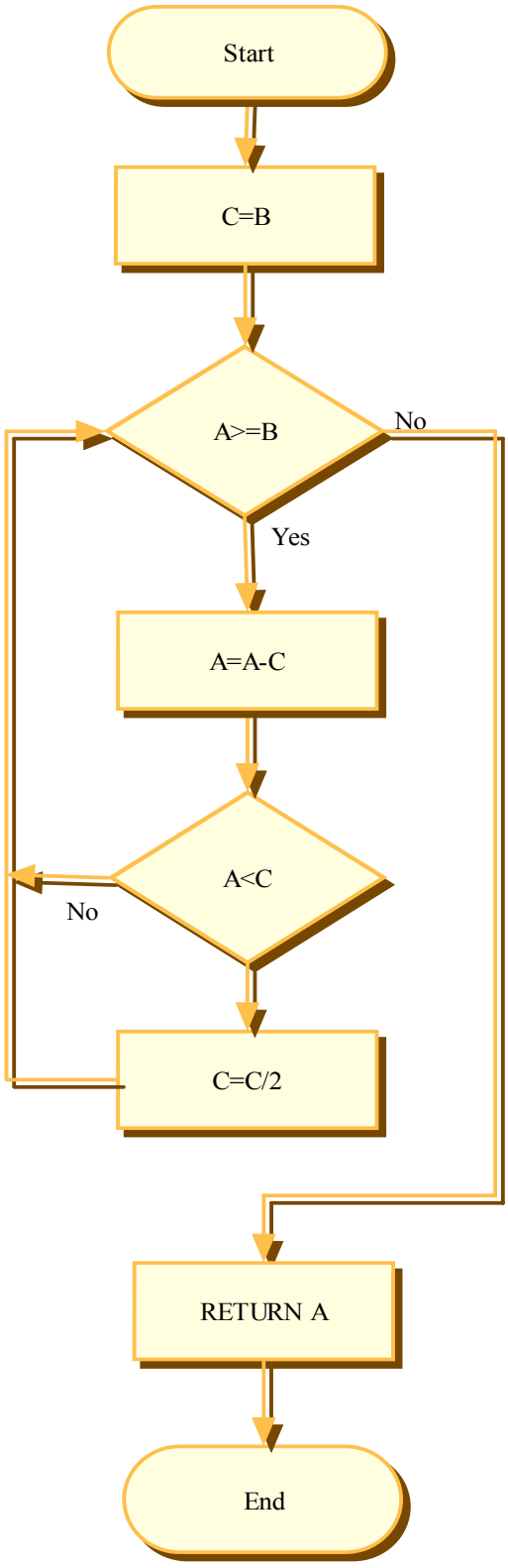


圖 4.2-6 MOD 流程圖

4.2.6 模乘法運算

使用移位和加法來做模乘法運算。Modmul (X,Y,N,Z) ，Z= (XXY) mod N。

把 Y 二進位表示<YL-1,YL-2,…..Y1,Y0>，L 為 Y 的總 bit 數 i=L-1。

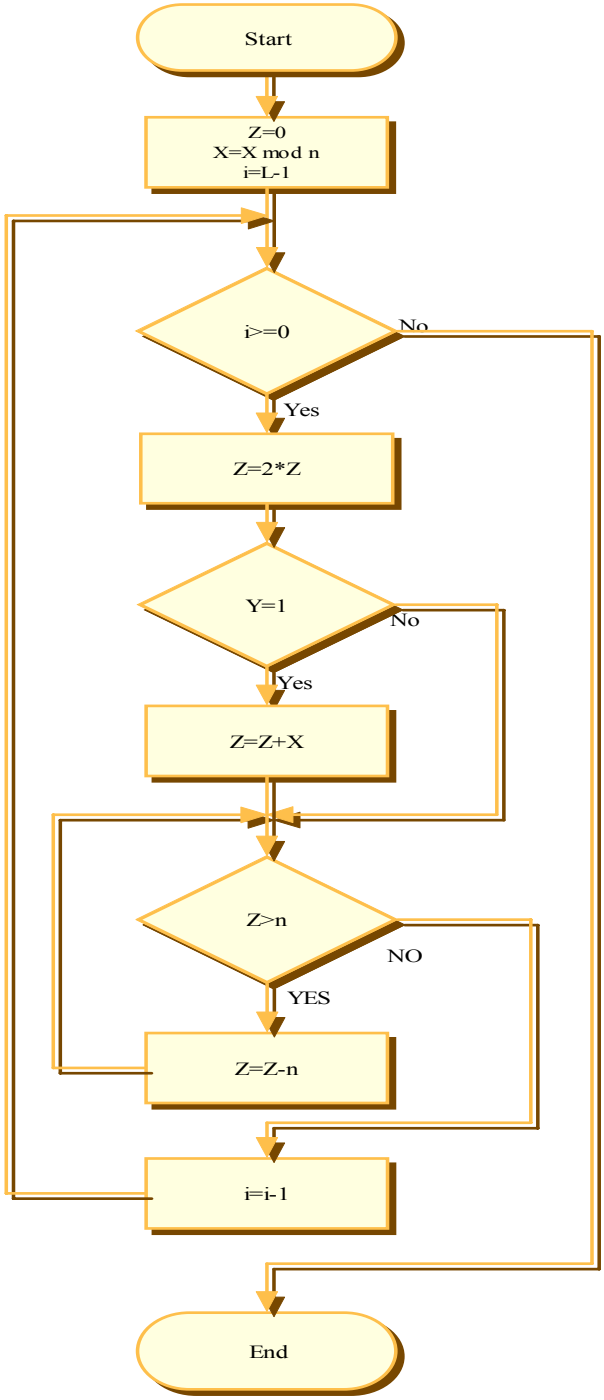


圖 4.2-7 模乘法運算流程圖

4.2.7 模指數運算

使用二進位法模指數運算。modexp(a,b,c,d)這個副程式是在做 $d=a^b \bmod c$。

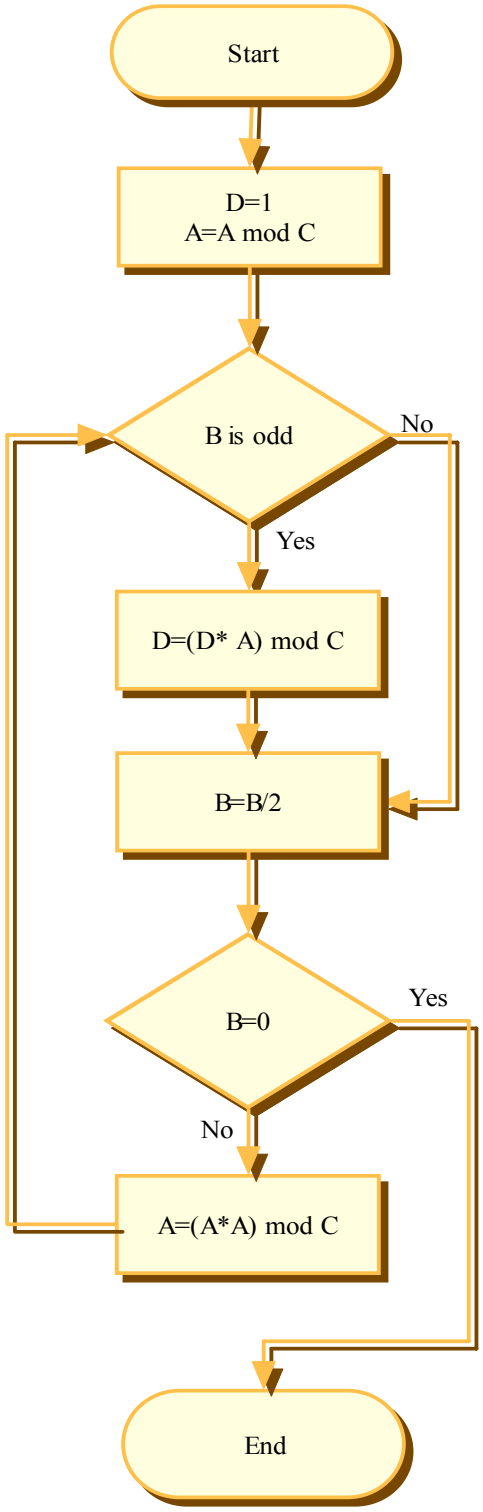


圖 4.2-8 模指數運算流程圖

4.3 System Design Framework of RSA.C

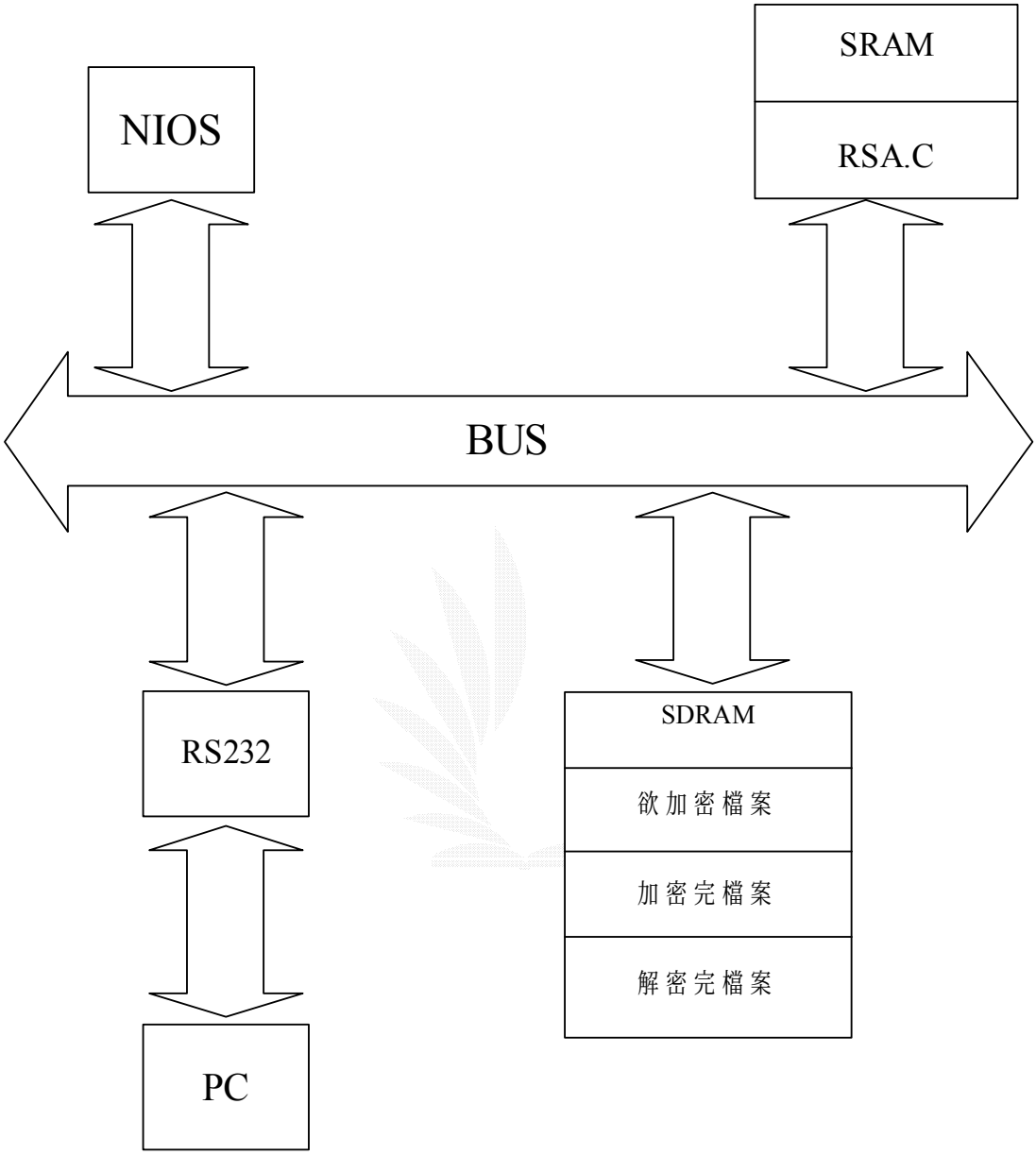


圖4.3-1 架構一的System design framework

這個架構最簡單，只要有RSA的C CODE之後，放進SRAM，然後把資料放進SDRAM之後就可以做加解密了。

4.3.1 RSA.C 簡介

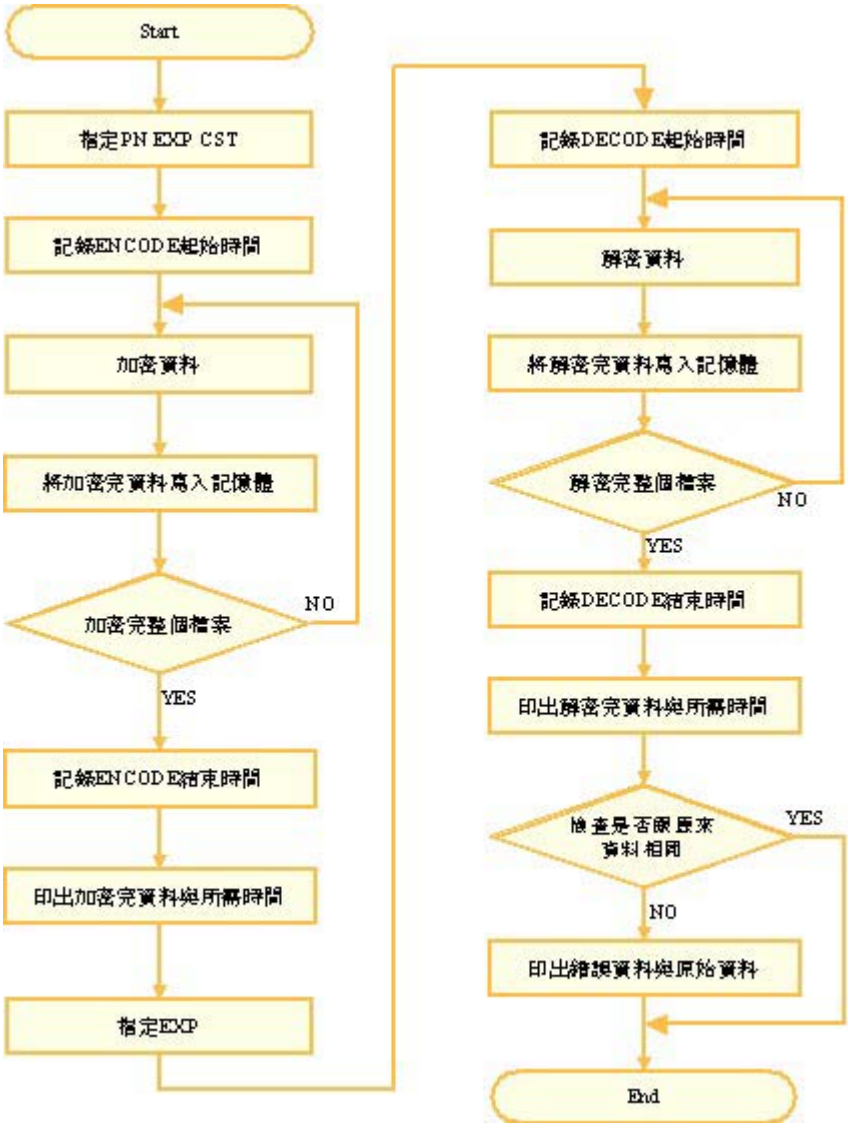


圖 4.3-2 RSA.C 程式流程圖

因為 RSA 運算的限制（會 OVERFLOW），所以這個程式的公、私鑰都是 64bit，而加密資料一次是 32bit（其實最大可到約 58 bit，但是使用 32bit 資料會比較好處理，所以採用 32bit。），加密完 32bit 的資料後，會產生 64bit 資料，所以加密完的資料便會是原來資料的兩倍大；而解密的時候，不需考慮 OVERFLOW，所以直接把 64bit 資料丟回去解密就可以，而解密完後的資料大小跟原來的資料大小一樣。

4.4 架構一的內部比較

Encode						
D \ I	0	1	2	4	8	16
0	599034545	489098476	489098461	489098465	489098448	489098490
1	572474452	470901962	470901946	470901937	470901946	470901961
2	572474404	470901861	490713471	470901865	480034008	—
4	572474327	465267526	490713405	470901824	480033941	—
8	572474339	440253122	490713425	470901803	480033967	—
16	572474345	455332387	490713420	436838764	—	—

表 4.4-1 Encode 花費時間表

Decode						
D \ I	0	1	2	4	8	16
0	549069550	448349336	448333388	448333294	448333303	448333289
1	524710571	431661955	431647600	431647507	431647509	431647496
2	524706643	431658321	443690730	431643623	486096688	—
4	524705868	438984134	443689938	431642831	486095880	—
8	524705868	454970189	443689938	431642831	486095880	—
16	524705868	426109310	443689938	516755715	—	—

表4.4-2 Decode花費時間表

Sopc builder 提供 INSTRUCTION and DATA cache 的調整所以我們針對 RSA.C 作分析並紀錄下來當把資料加密後所花的 clock cycles 的數目。如表 4.4-1。

一、固定DATA cache（簡稱為D cache）的值觀察

INSTRUCTION cache（簡稱為I cache）的變化及其影響程式執行的clock cycle 的數目 。如圖4.4-1我們觀察到當I cache由0變成1時clock cycle 的數目明顯的下降、程式的執行效能提升。但是當I cache再往上增加時clock cycle 的數目沒有明顯的改變。因為當I cache從0改變為1時，在cpu裡I cache是從無變有，因此clock cycle 的數目明顯的下降，由於大部分的運算都是固定的順序的指令，最常見的地方就是連續一大串的乘法指令(因為指數運算)，並不會有很多的不同的指令，所以有1Kbyte的I catch就足夠了，多餘的並沒有用。

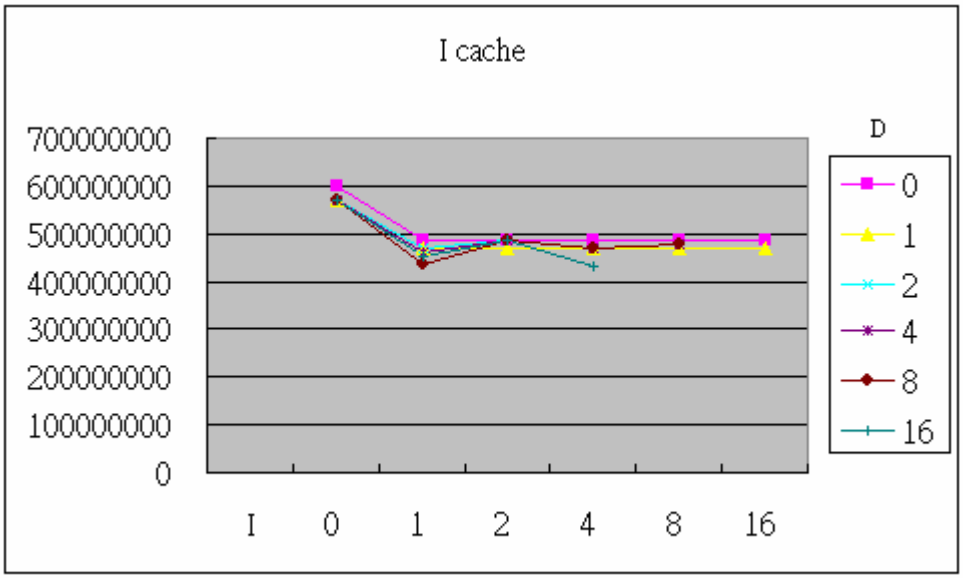


圖 4.4-1　Encode I cache 比較圖

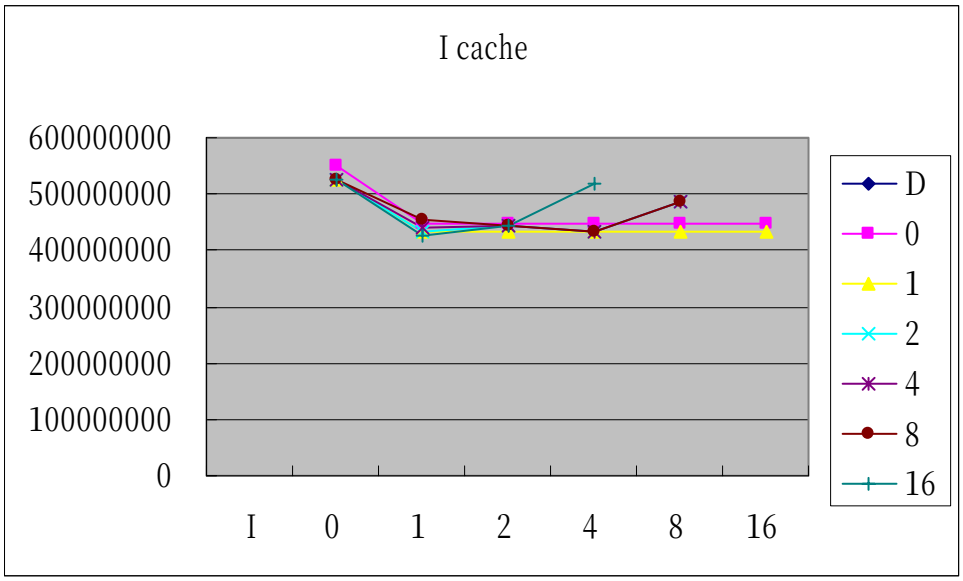


圖4.4-2　Decode I cache比較圖

二、固定 INSTRUCTION cache（簡稱為 I cache）的值觀察 DATA cache（簡稱為 D cache）的變化及其影響程式執行的 clock cycle 的數目。如圖 4.4-2 我們觀察到當當 D cache 再往上增加時 clock cycle 的數目沒有非常明顯的改變。因為我們的程式在作加密時是一次讀進 64bit 的資料作加密後寫到記憶體裡然後在作一次 64bit 資料的加密，用完就丟，因此 DATA cache miss 的機會很高。

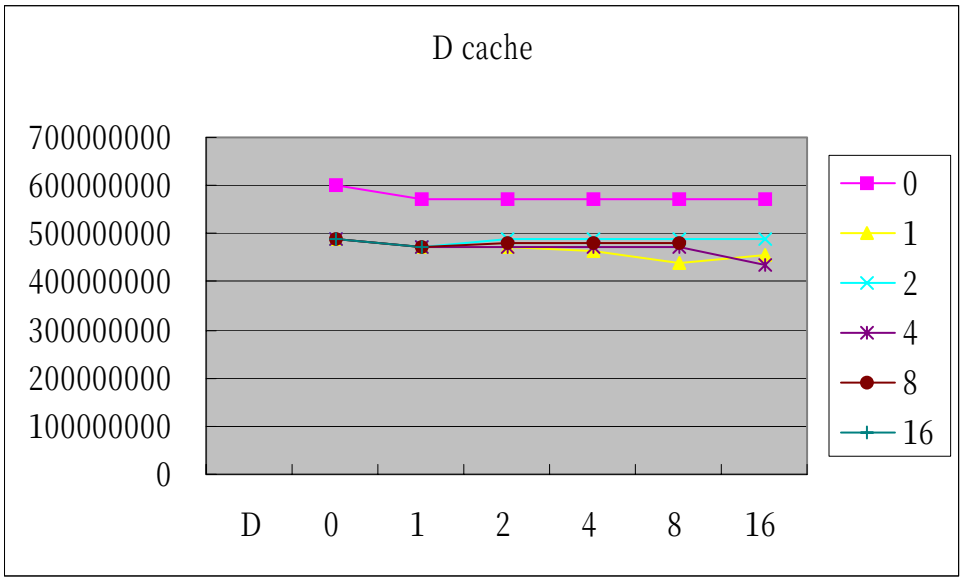


圖4.4-3　Encode D cache比較圖

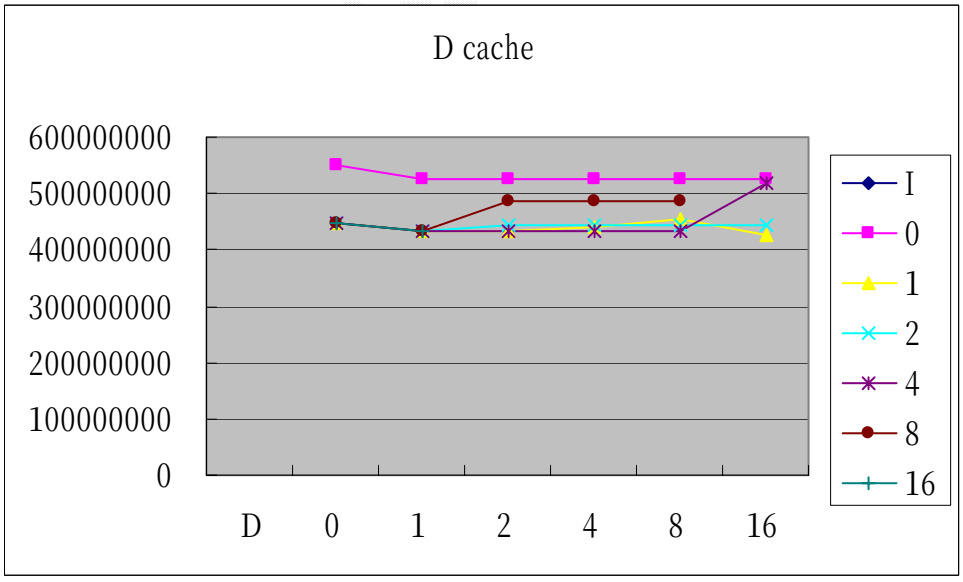


圖4.4-4　Decode D cache比較圖

第五章　使用 Custom Instruction 增加執行效率（架構二）

5.1　Introduction

對於CPU來說，常常一道 C 指令被轉譯成一堆Assembly Code，更或者，System is designed for special purposes，這樣一來，就形成有一連串指令不斷的被重複，如果把這一連串的Assembly Code或指令結合成一個步驟，就可以加快執行。

利用這種特性，我們可以把這個東西應用在很多地方，例如（DSP）為了執行影像加解壓，可能有個loop不斷的在內部執行，這時候就會把這個常常用到的地方抓出來做成硬體來達到加速的目的。

5.1.1　Altera Nios Embedded Processor VS. Custom Instruction

Altera Nios Embedded Processor支援使用者在Nios Instruction Set內增加Custom Instructions來加快效能，所以我們可以把複雜的工作簡化成一個動作，另外我們可以在Custom Instructions內對Nios System外的memory做存取，也可以跟外部的logic unit溝通。使用者可以根據所要的功能設計Combinational電路或者Sequential電路，然後在Nios Processor’s Arithmetic Logic Unit (ALU)跟Instruction Set內增加你設計的Custom Instructions。

Custom instructions 主要包含兩個部分：

1. 硬體設計部份：*Custom logic block*：主要就是使用者自訂的logic blocks，然後Nios把這個block整合到Nios microprocessor ’ s ALU內。
2. 軟體支援部份：*Software Macro*：允許使用者在software code內去使用Custom Logic。

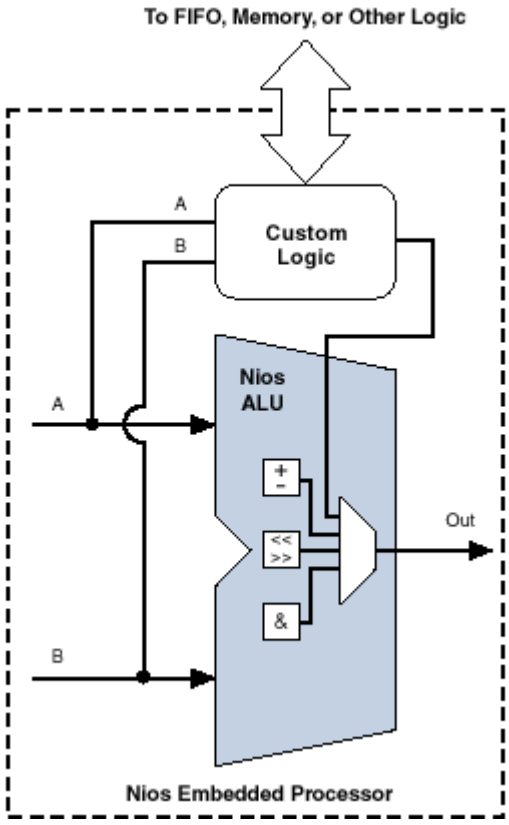


圖5.1-1　Custom Logic And Nios ALU

Custom Logic Block 執行使用者定義的logic的時候，是把兩個暫存器內的值(Ra and Rb)讀出來，然後把result存回Ra去，example：Ra <- Ra op Rb；或者是對其他位置的資料做存取 example：Ra <- Ra op %r0 （%r0代表暫存器，可以是ADDRESS或者暫存器。）所以只要是符合這些規定的任何logic block均可放到Nios ALU內。

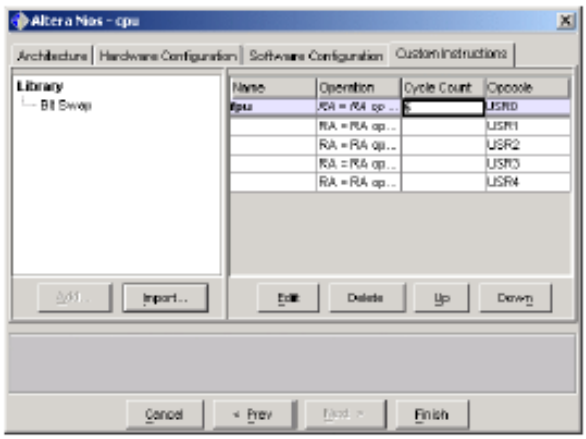


圖5.1-2　Custom Instructions Tab

我們可以在Nios Embedded Processor設定內增加我們要的Custom Logic，Nios Configuration 內我們可以建立Software Marcos（c/c++ or Assembly）來提供Software Code可以存取Custom Logic Block。如圖示，Nios CPU預留了5個Custom Instruction的位置，所以使用者可以根據所要的功能設計Combinational電路或者Sequential電路，接著在SOPC Builder的UI下面直接加入即可，我們可以定義Custom Instructions的Name還有動作，跟cycle count，如果是Combinational電路，那clock cycles就是1，如果是Sequential電路，那就依照你所測量出來的clock cycles填上去，注意clock cycles如果不正確，那也別指望答案會正確。

5.1.2　Interface of hardware

設計Custom Instruction的硬體，我們可以用好幾種格式來定義Custom Logic Block，如:Verilog HDL、VHDL、EDIF net list file、Quartus II Block Design File （.bdf）、Verilog Quartus Mapping File (.vqm)等等，SOPC Builder可以直接辨識

以上幾種格式。

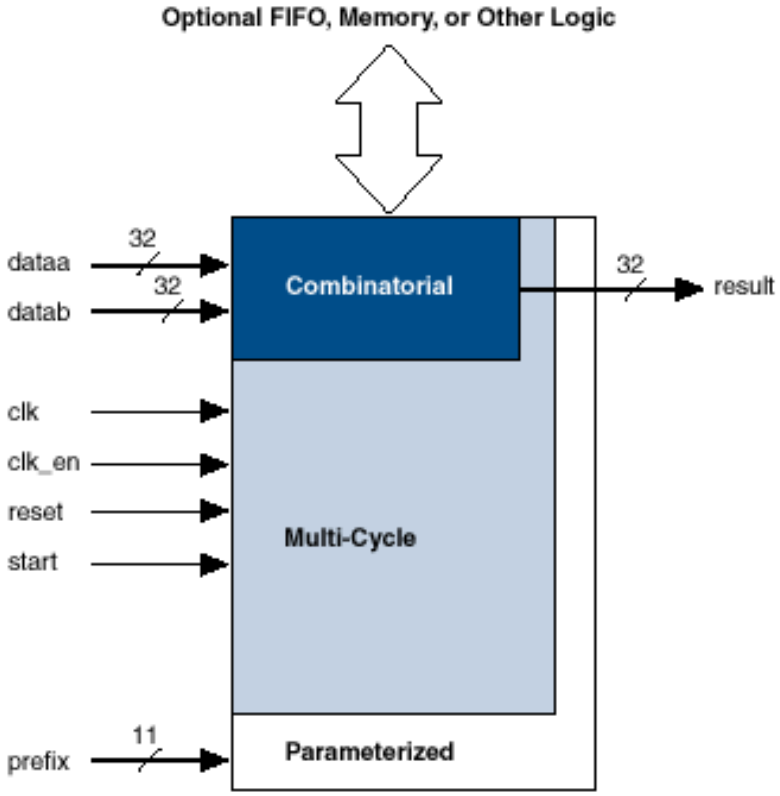


圖5.1-3　Custom Logic 介面圖

由於block要跟Nios ALU做配合，所以必須提供一個介面，而此介面必須符合Nios ALU的規定，port跟names都要依照規格來定義，這樣Nios才能夠對你所設計的custom logic找到需要的port並且連接正確。

架構如圖5.1-3所示，dataa，datab跟result的bit數是跟CPU的width相同的，而prefix這個port是把暫存器K(11bit)內的值取出來當作參數一樣使用，這port通常用來做function選擇用的，其他像clk、clken、reset、start等即使沒有使用也必須定義出來，否則Nios會無法正確啟動Custom Logic。

5.1.3 Interface of software

軟體介面主要就是提供類似Assembly 的opcode的東西，能提供software去使用Custom Logic，Nios內可以支援到最多5個Custom Instructions，也就是說opcode最多5個，除了定義opcode的Name外，還有要Type跟Format的格式，如表X.1-1所示，USER0 opcode 的Type是RR，RR表示是暫存器Ra跟Rb所能產生的任何結果，Format則表示這個Opcode的主要表示格式，像USER1的Type是RW，這表示Rb這時候代表的是%r0 的暫存器。

Opcode	Type	Format
USR0	RR	Ra <- Ra op Rb
USR1	Rw	Ra <- Ra op %r0
USR2	Rw	Ra <- Ra op %r0
USR3	Rw	Ra <- Ra op %r0
USR4	Rw	Ra <- Ra op %r0

表5.1-1 User Opcode, Type & Format的例子

5.2 Percent of Time Spent分析

為了知道哪幾個指令是最耗費時間的，我們必須針對硬體上的實際執行情形來分析，從分析資料決定需要加速的指令，表5.2-1是架構1的Analysis Data。

Index	Time	Self	Children	Name
1	50.2	178342.00	0.00	internal_mcount
2	10.4	36939.00	0.00	sub
3	9.2	32570.00	0.00	_vfprintf_r
4	7.9	28086.0	0.00	compare
5	5.7	20276.00	0.00	modmul
6	5.1	18298.00	0.00	shiftleft
7	4.9	17589.00	0.00	add
8	4.4	15616.00	0.00	mod

表5.2-1 架構1的Analysis Data

上面分析結果顯示internal_mcount耗費最多，不過這不是RSA必要的副程式，所以捨棄掉，sub佔第二高，但是sub本身就是CPU指令了，也捨棄掉，_vfprintf_r是印出結果的，跟RSA也無關，compare其實只是mod裡面的一小部分而已，最後我們發現把modmul跟mod做成CI可以加快最多的速度。後來在設計上的考量，就決定做mod跟mul這兩個instructions。

5.3 MOD 的做法

5.3.1 利用長除法求 C = A mod B

5.3.1.1 原理

這個求餘數的做法是利用長除法做的，每次都用 B×2（B << 1）來跟 A 比較，如果當 A 大於 B 的時候，B 就回到先前的狀態，然後 A=A-B；再來 B 又回到最初狀態，依照剛剛的方法做，直到 A 比原來的 B 小為止，此時，減完的 A 就是 A mod B 的結果。

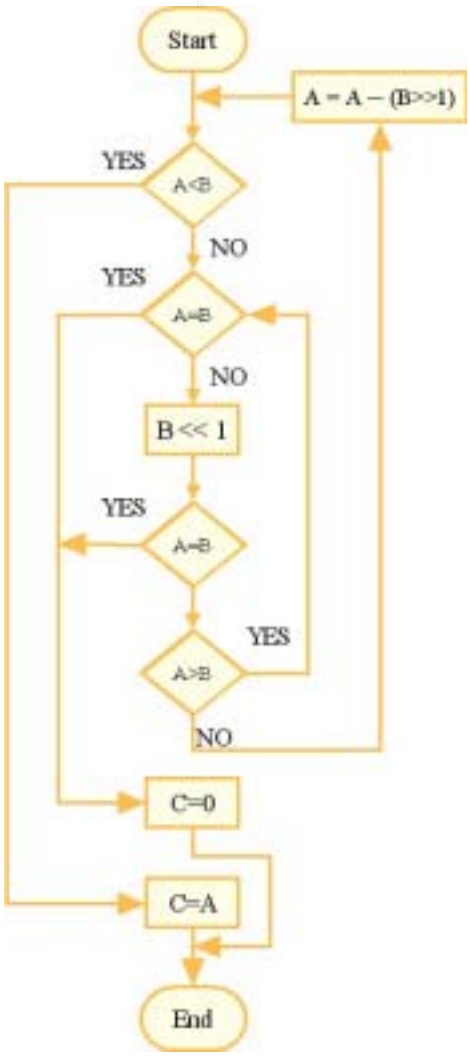


圖 5.3-1　第一種 MOD 做法架構設計圖

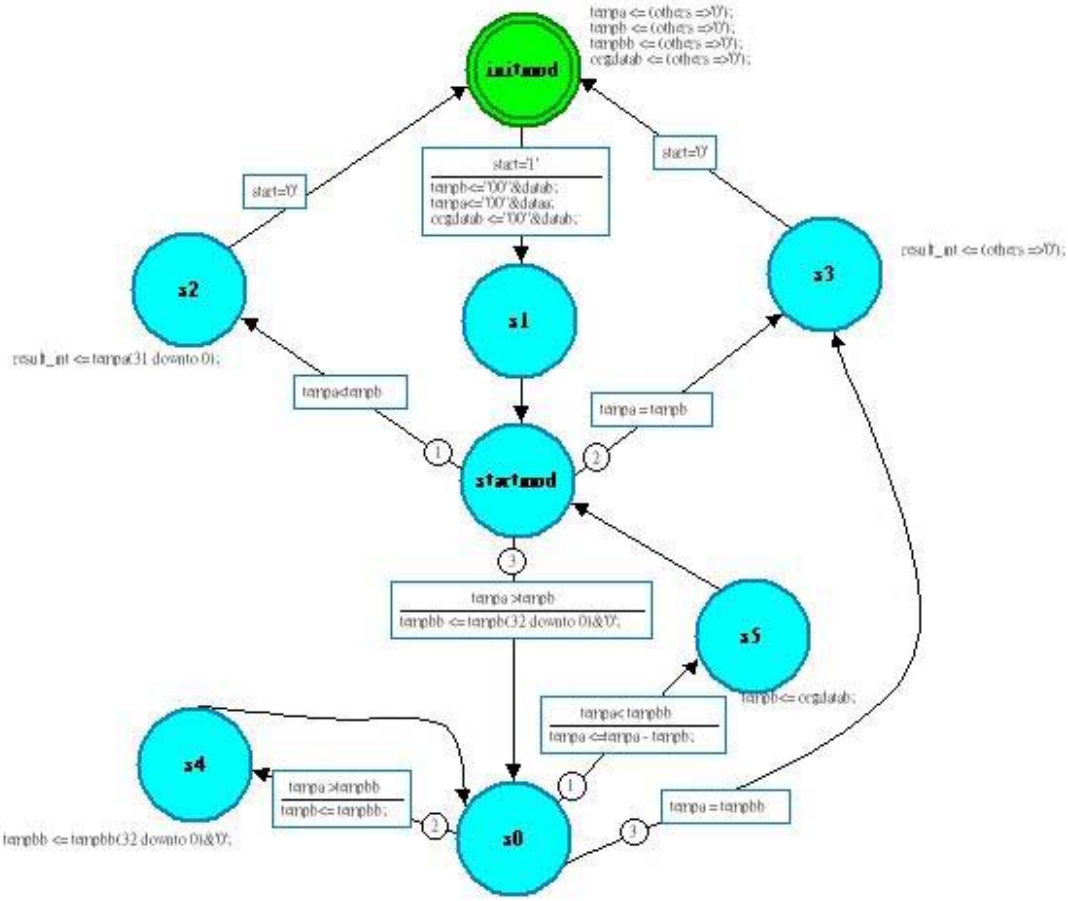


圖 5.3-2　第一種 MOD 做法 FSM 圖

5.3.1.2　模擬結果

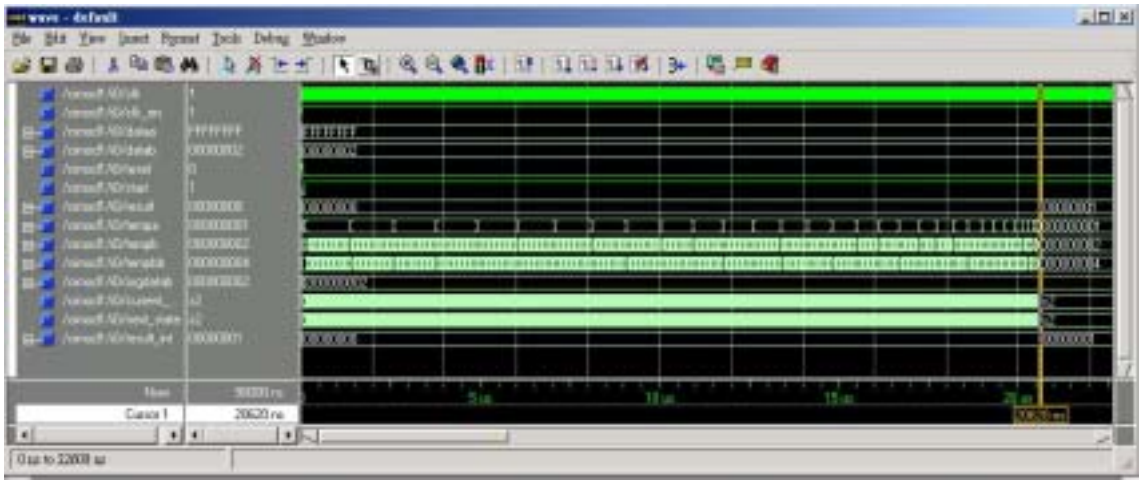


圖 5.3-3　第一種 MOD 做法模擬圖

設定 A=0xFFFFFFFF，B=2，求最長執行時間。求出來的最長時間是 20620ns，我們設定一個 clock cycle=20ns，所以他最久需要 1031 個 clock cycles 做完一個 MOD 指令。

5.3.1.3 Leonardo Spectrum 預估 RSA-WRAPPER 資料

單一元件執行速度：174.8 MHz。
單一元件所需邏輯單元：427。

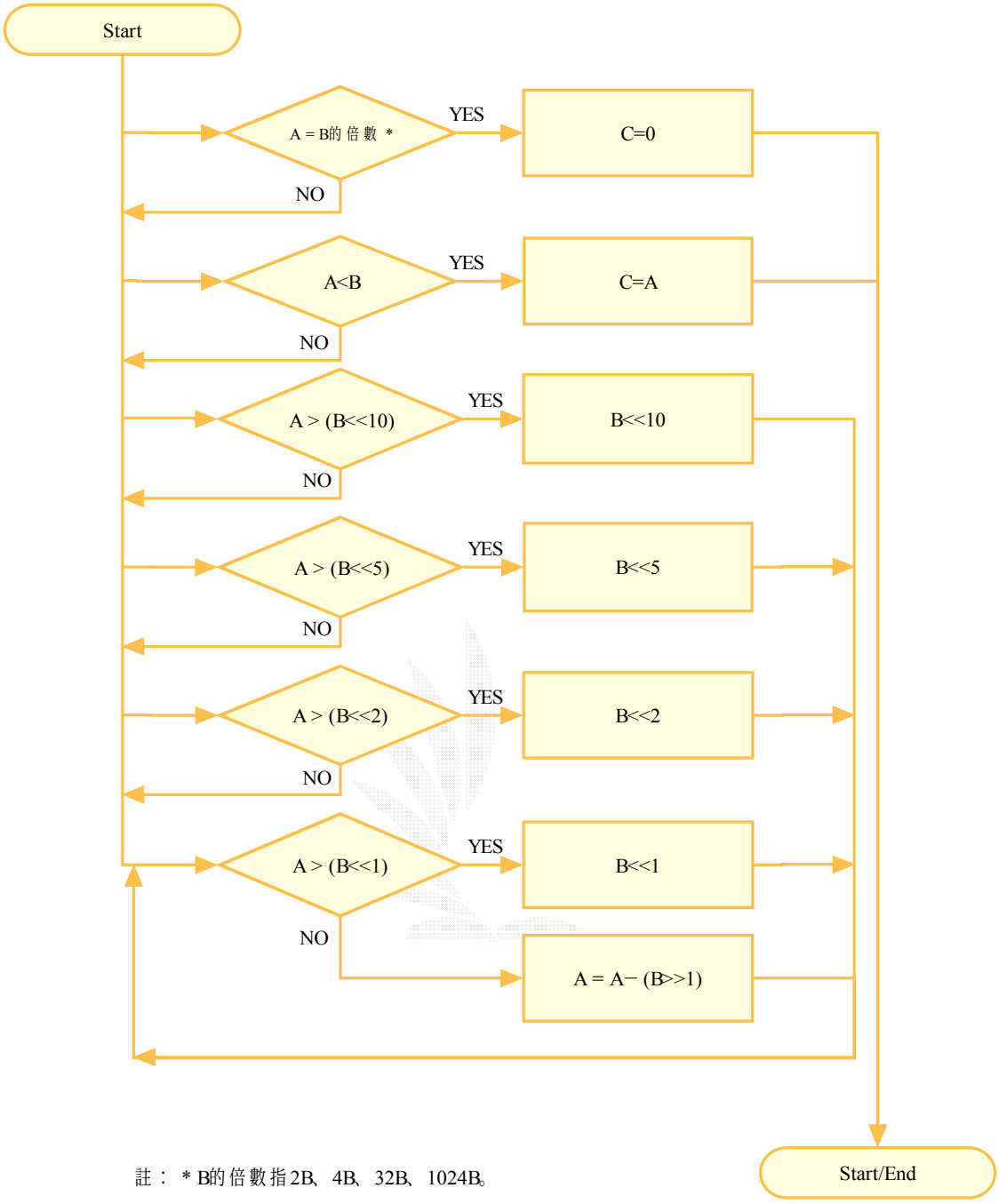
5.3.1.4 優缺點

由最基本的長除法做出來的，所以架構簡單，因此所花費的邏輯單元最少。花費時間最多，甚至比使用軟體更沒有效率，大部分所花費的時間都是在做 BX2（B << 1）以及比較 A、B 大小。

5.3.2 利用加速長除法求 C = A mod B。

5.3.2.1 原理

類似原理一，但是同時做 B<<10、B<<5、B<<2、B<<1，加速找 A 的最高位元。



註： * B的 倍 數 指 2B、 4B、 32B、 1024B。

圖 5.3-4　第二種 MOD 做法架構設計圖

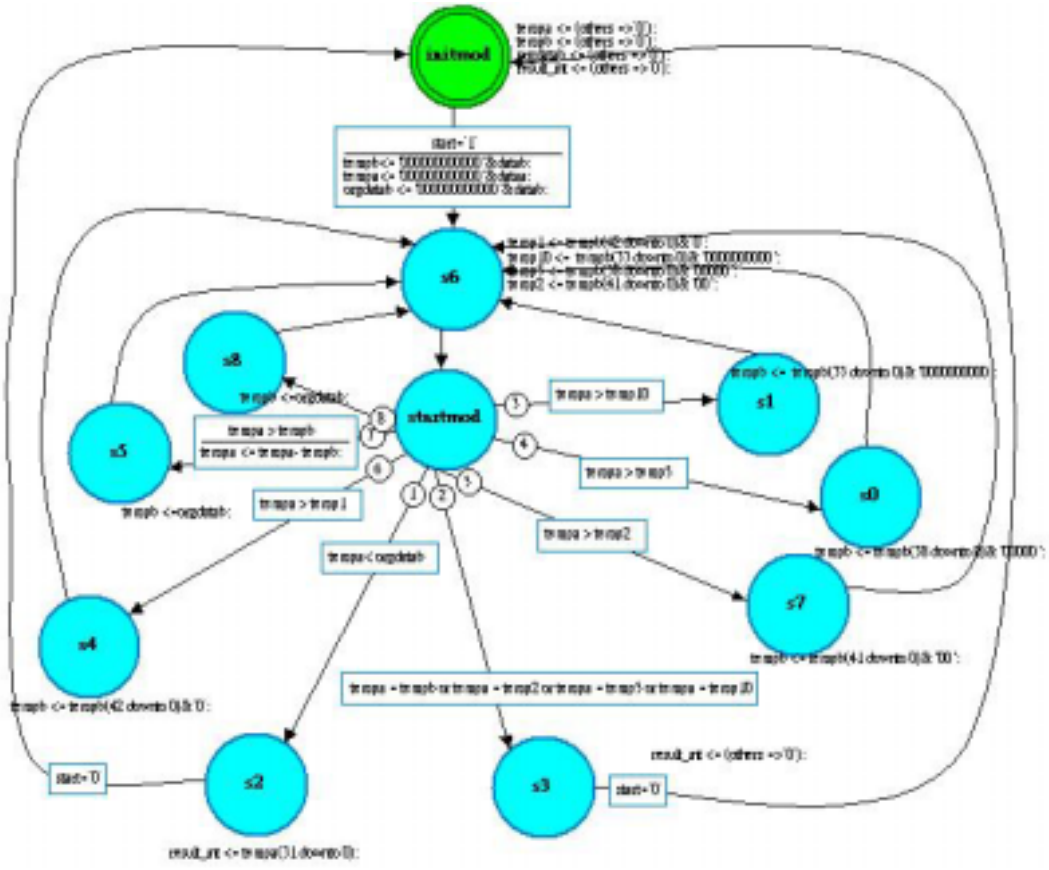


圖 5.3-5　第二種 MOD 做法 FSM 圖

5.3.2.2　模擬結果

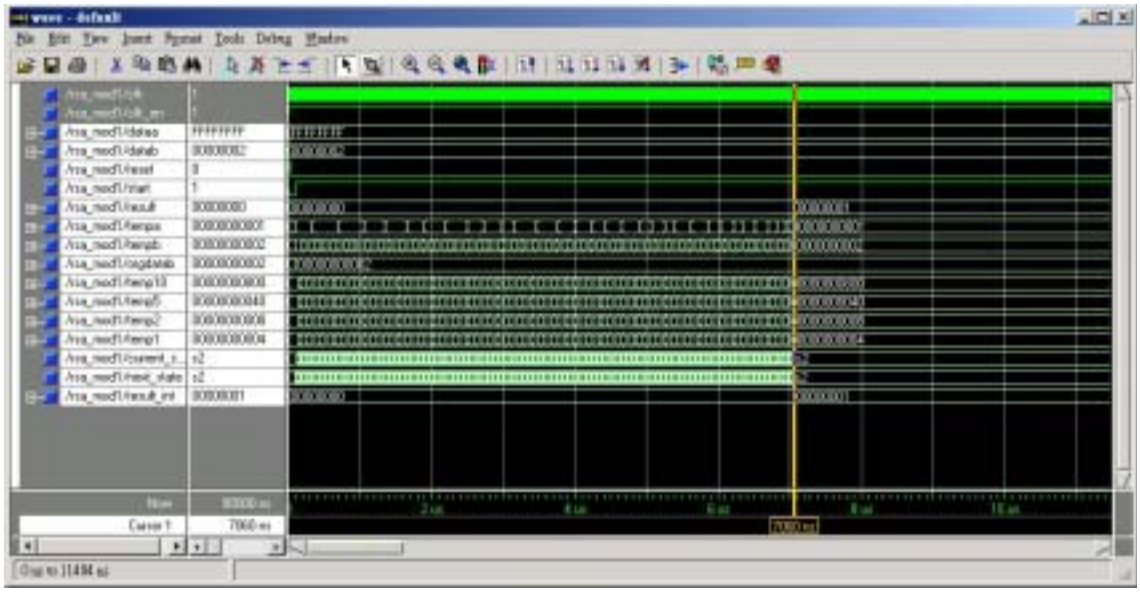


圖 5.3-6　第二種 MOD 做法模擬圖

設定 A=0xFFFFFFFF，B=2，求最長執行時間。求出來的最長時間是 7060ns，我們設定一個 clock cycle=20ns，所以他最久需要 353 個 clock cycles 做完一個 MOD 指令。

5.3.2.3 Leonardo Spectrum 預估

單一元件執行速度：135.6 MHz。
單一元件所需邏輯單元：741。

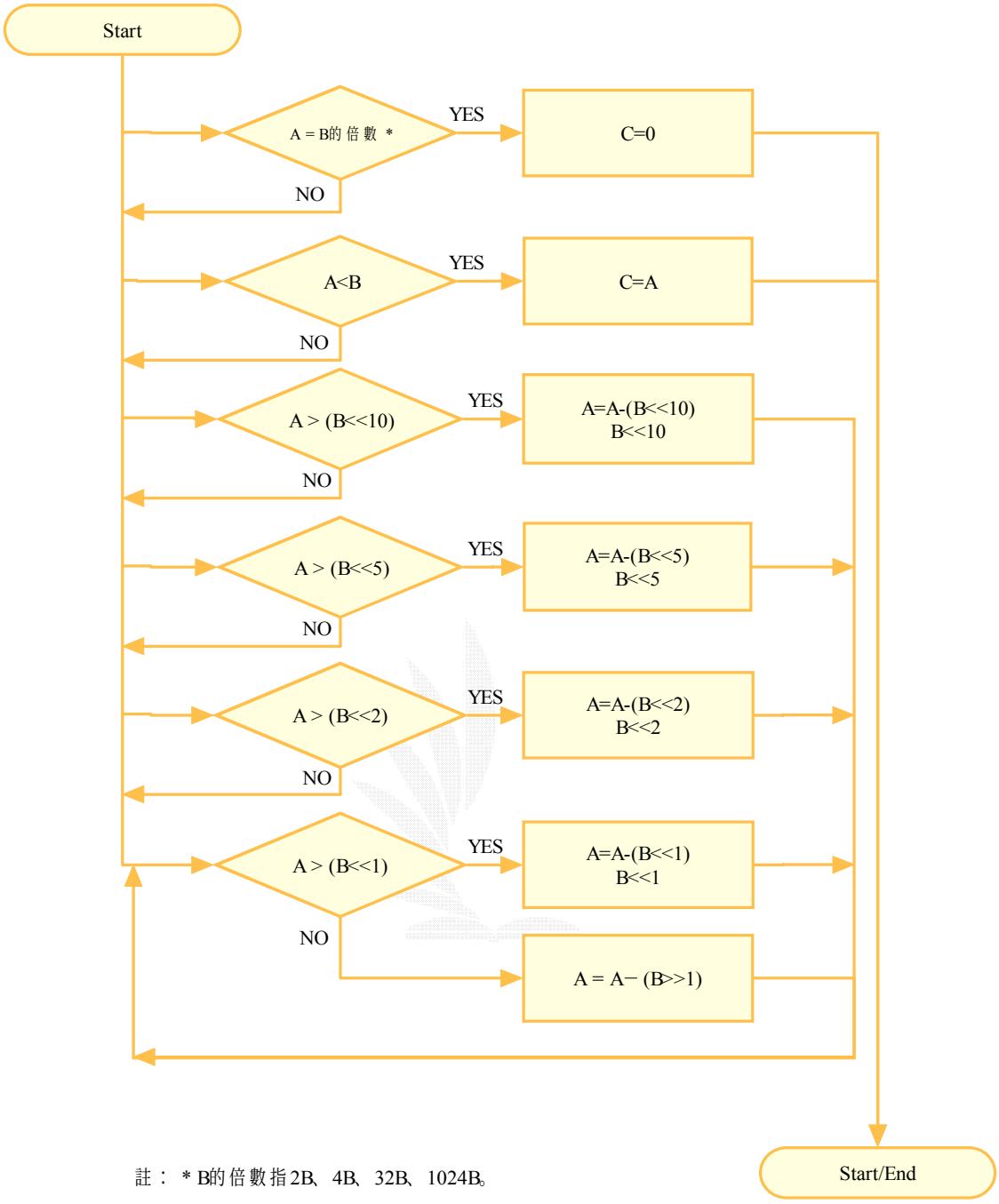
5.3.2.4 優缺點

速度比第一個做法快。架構比較複雜，而且做完 SHIFT 的資料沒好好利用，有點浪費；所需的空間也比第一個做法大，但是這種做法依然比軟體速度來的慢。

5.3.3 利用超級加速長除法求 C = A mod B

5.3.3.1 原理

類似第二個做法，不過每次比較完大小之後就減掉該值。



註： * B的 倍 數 指 2B、 4B、 32B、 1024B。

圖 5.3-7　第三種 MOD 做法架構設計圖

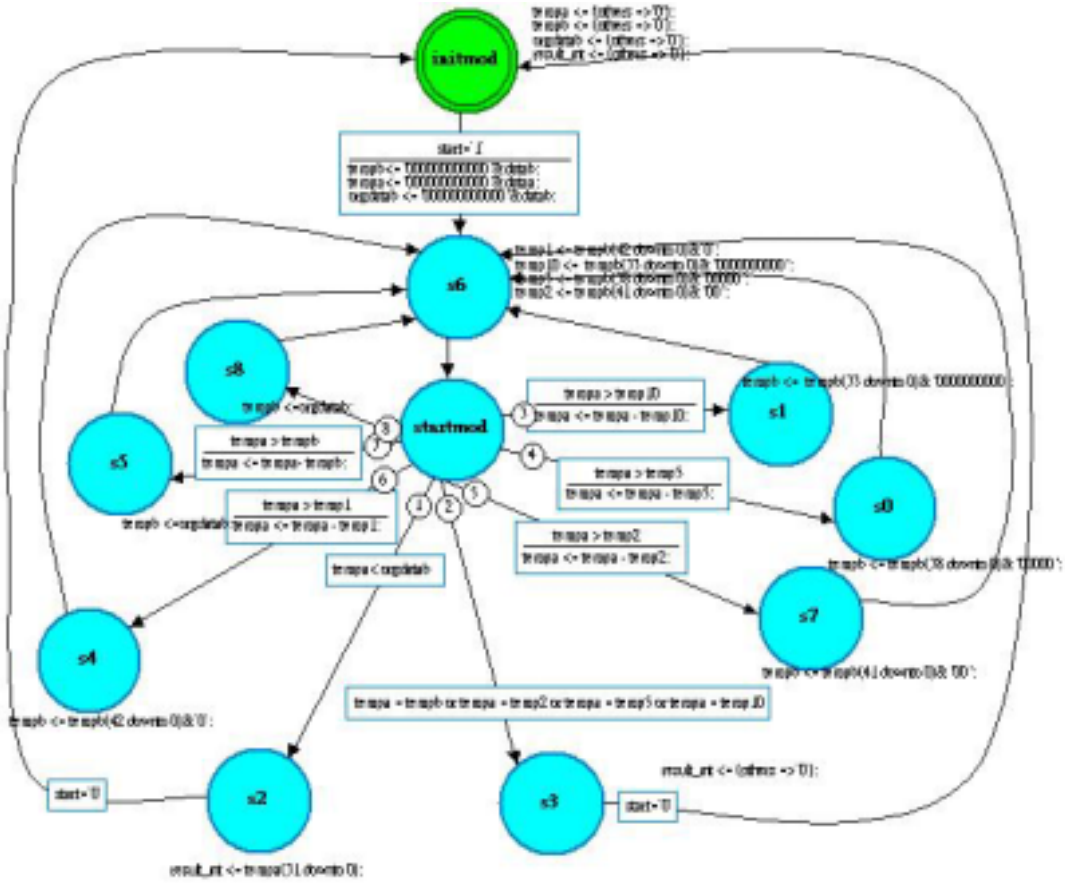


圖 5.3-8　第三種 MOD 做法 FSM 圖

5.3.3.2　模擬結果

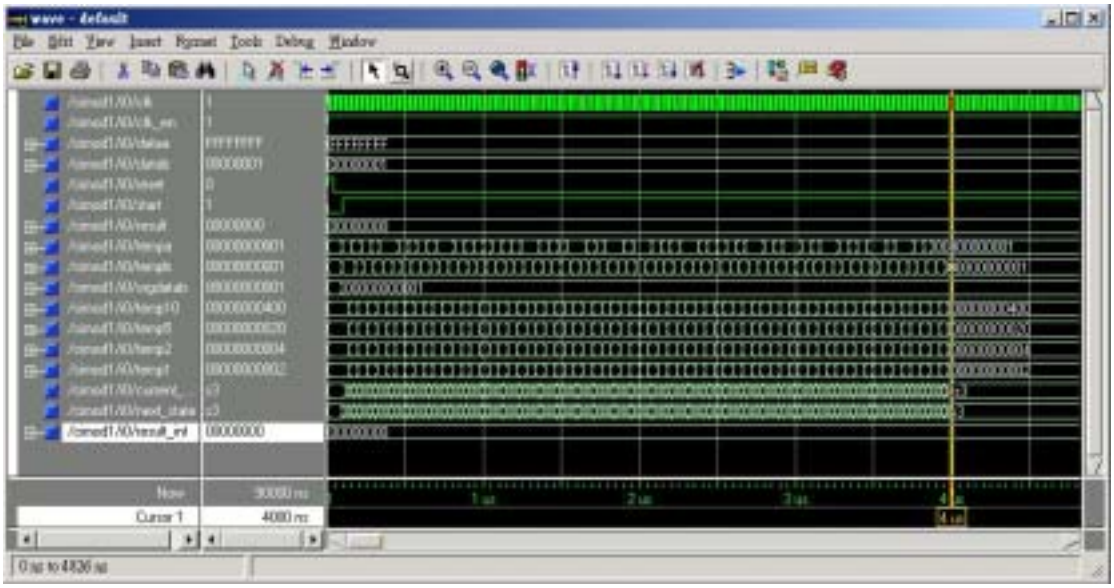


圖 5.3-9　第三種 MOD 做法模擬圖

設定 A=0xFFFFFFFF，B=2，求最長執行時間。求出來的最長時間是 4000ns，我們設定一個 clock cycle=20ns，所以他最久需要 200 個 clock cycles 做完一個 MOD 指令。

5.3.3.3 Leonardo Spectrum 預估

單一元件執行速度：82.1 MHz。
單一元件所需邏輯單元：894。

5.3.3.4 優點

執行速度比第二個快，較能利用剛剛所做過的動作。所要的邏輯單元次多，次佔空間，架構次複雜。

5.3.4 利用暴力法求 C = A mod B

5.3.4.1 原理

這是用第三個架構改來的，第三個架構只針對某幾個倍數做比較，而這個做法是比較所有的倍數，所以速度快上很多，但是所花費的 AREA 也多上好幾倍。

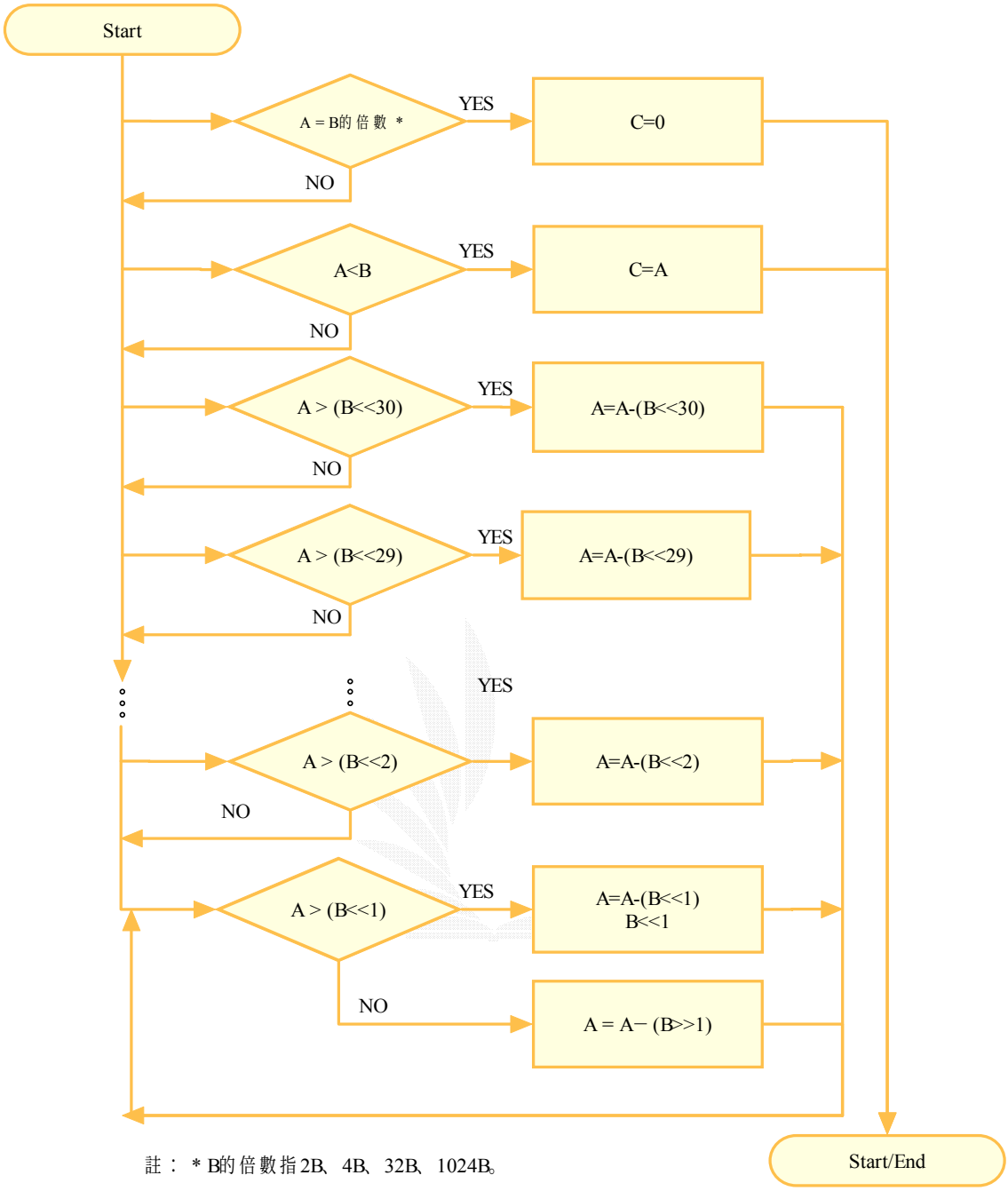


圖 5.3-10　第四種 MOD 做法架構設計圖

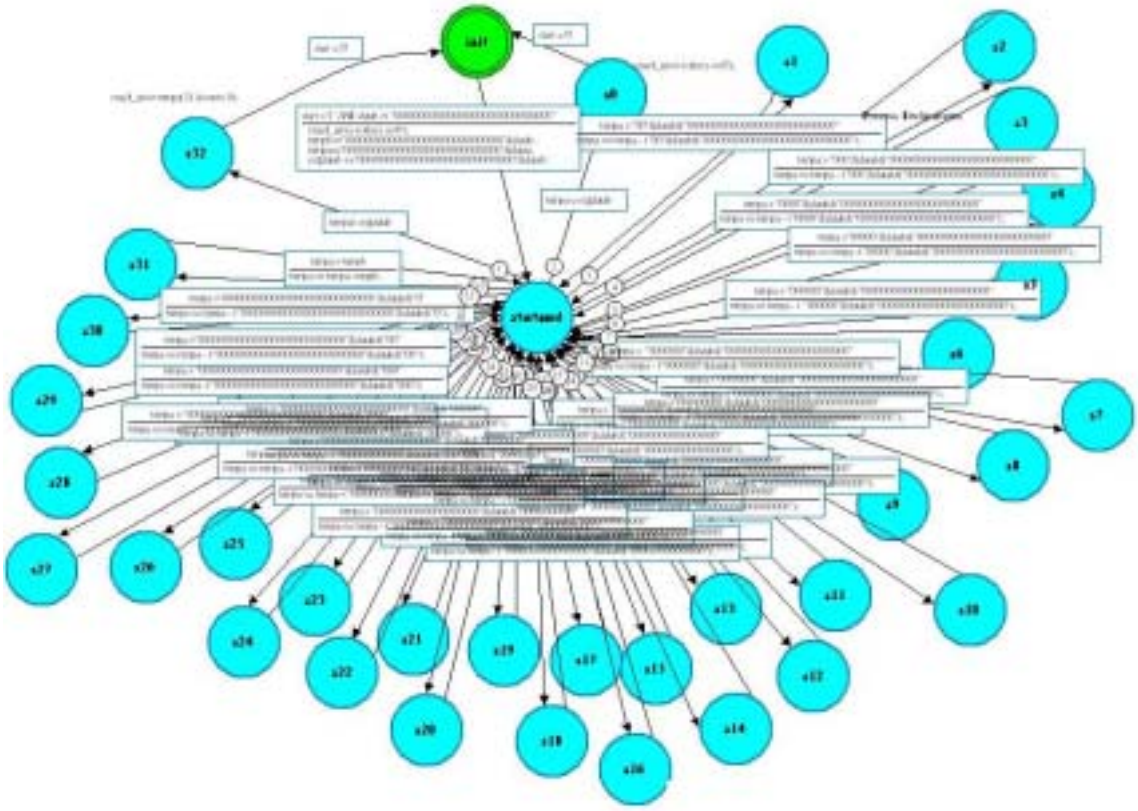


圖 5.3-11　第四種 MOD 做法 FSM 圖

5.3.4.2 模擬結果

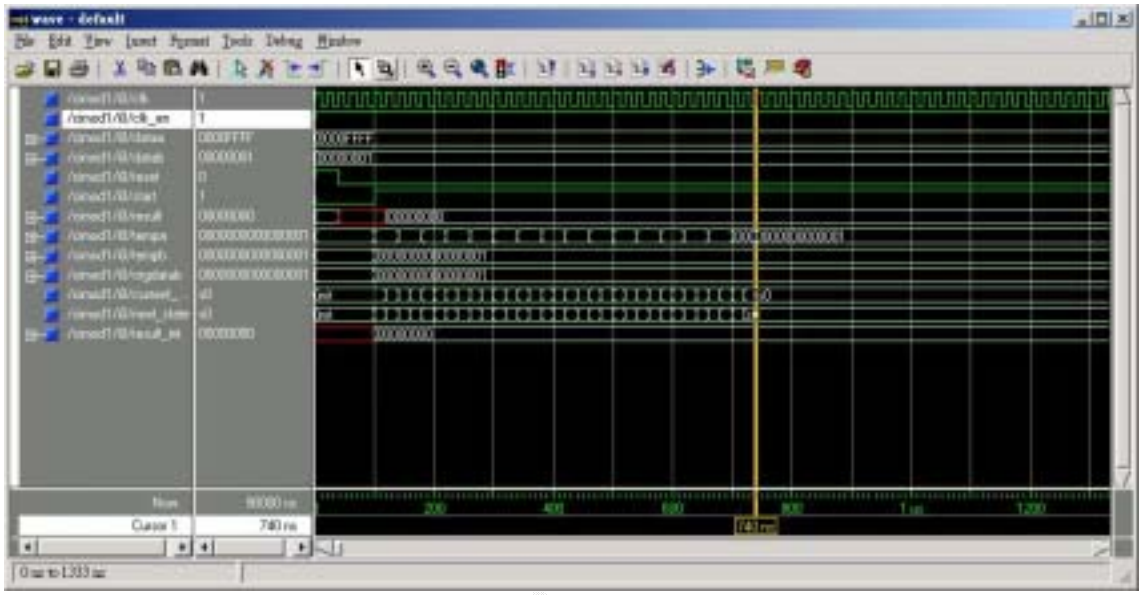


圖 5.3-12　第四種 MOD 做法模擬圖

設定 A=0xFFFFFFFF，B=2，求最長執行時間。求出來的最長時間是 640ns，我們設定一個 clock cycle=20ns，所以他最久需要 32 個 clock cycles 做完一個 MOD 指令。

5.3.4.3 Leonardo Spectrum 預估

單一元件執行速度：58.6 MHz。

單一元件所需邏輯單元：3078。

5.3.4.4 優缺點

速度最快，但所需 AREA 最大，架構最亂。

5.3.5 討論

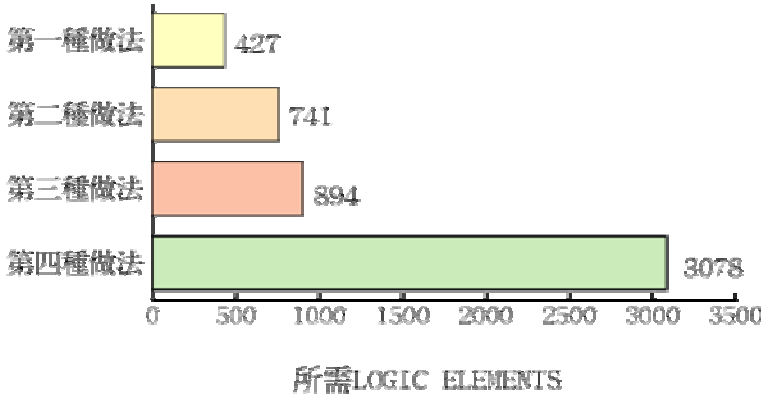


圖 5.3-13　四種 MOD 所需 LOGIC ELEMENTS 比較

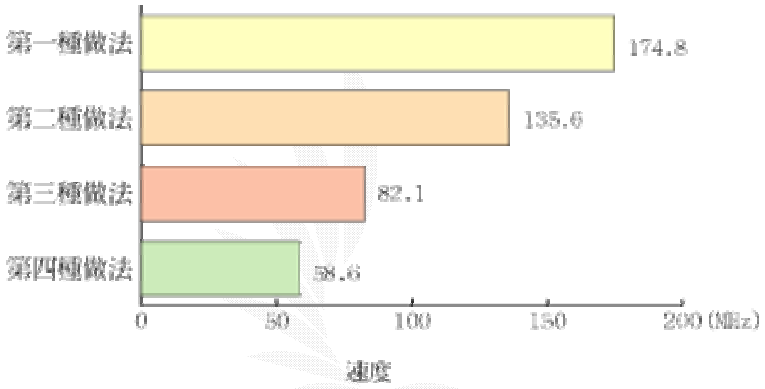


圖 5.3-14　四種 MOD 速度比較

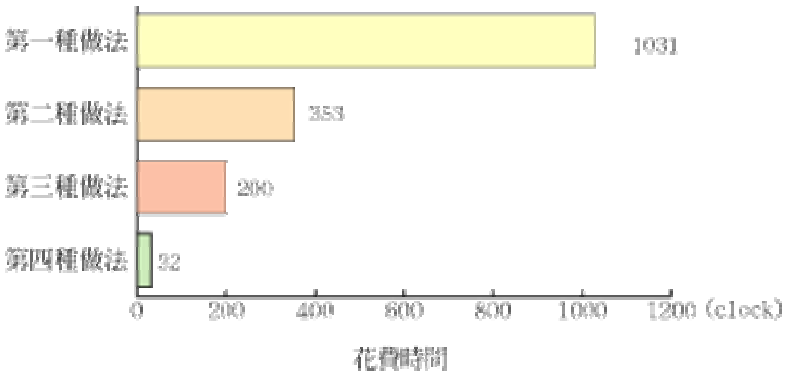


圖 5.3-15　四種 MOD 花費時間比較

	所需 L.E.	速度/MHz	花費時間/clk
第一種做法	427	174.8	1031
第二種做法	741	135.6	353
第三種做法	894	82.1	200
第四種做法	3078	58.6	32

表 5.3-1　四種 MOD 做法比較

為什麼用 Leonardo Spectrum 測出來的速度會跟用 MODELSIM 所模擬出來的結果剛好像反？Leonardo Spectrum 會根據程式找出 Critical Path，然後根據 Critical Path 來計算出速度，實際上沒有真正放數值去跑過；而 MODELSIM 是我們設定數值讓他模擬，所以跑出來的結果會比較接近真正跑的結果；把這四個 MOD 分別放到架構二裡面實作時，也試第四種 MOD 做的比較快。

5.4 System Design Framework of CI

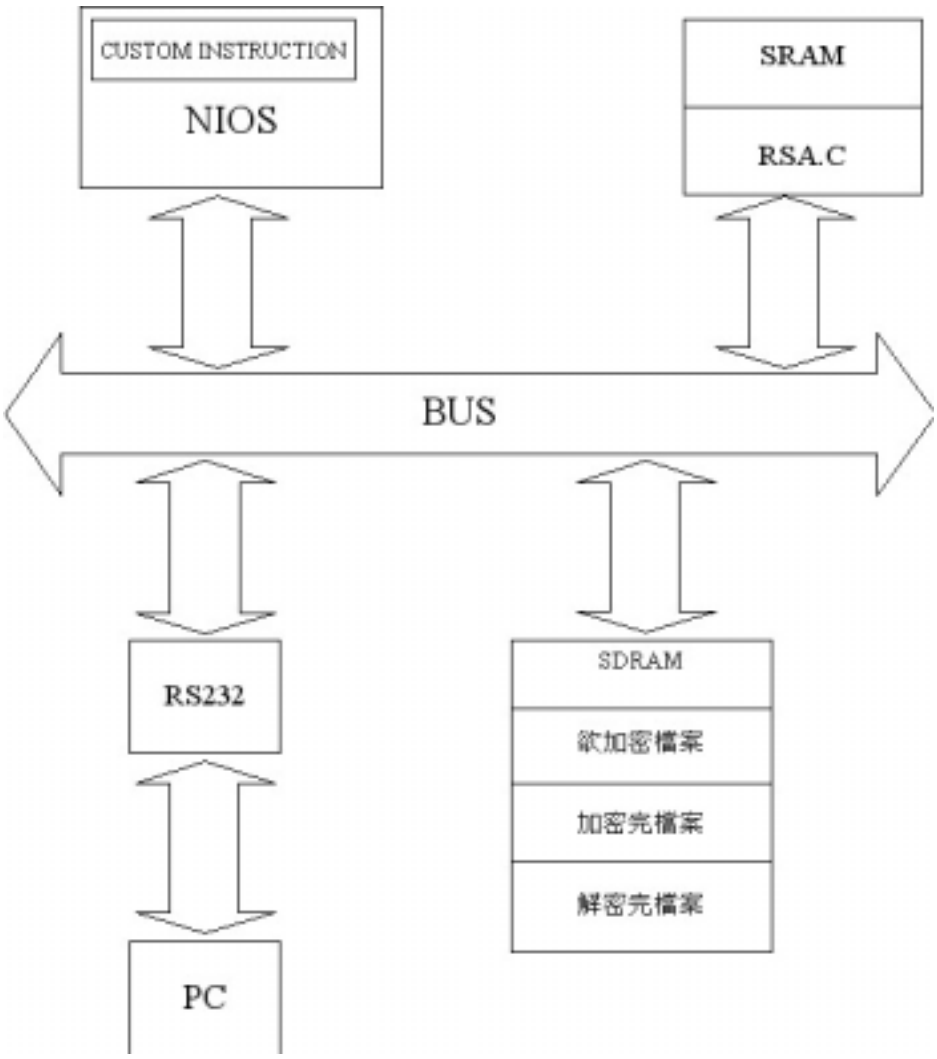


圖 5.4-1 CI 的 System design framework

我們在 Nios CPU 裡增加兩個 Instruction：mod 與 mul，把 RSA.C 放進 SRAM 裡面，執行完一次 C 程式之後就做完加密檔案、解密檔案以及比對解密完檔案是否跟原來檔案相同， SDRAM 分割 3 部分，存放原始資料，加密完的資料，跟解密完的資料。

5.4.1 RSA.C 簡介

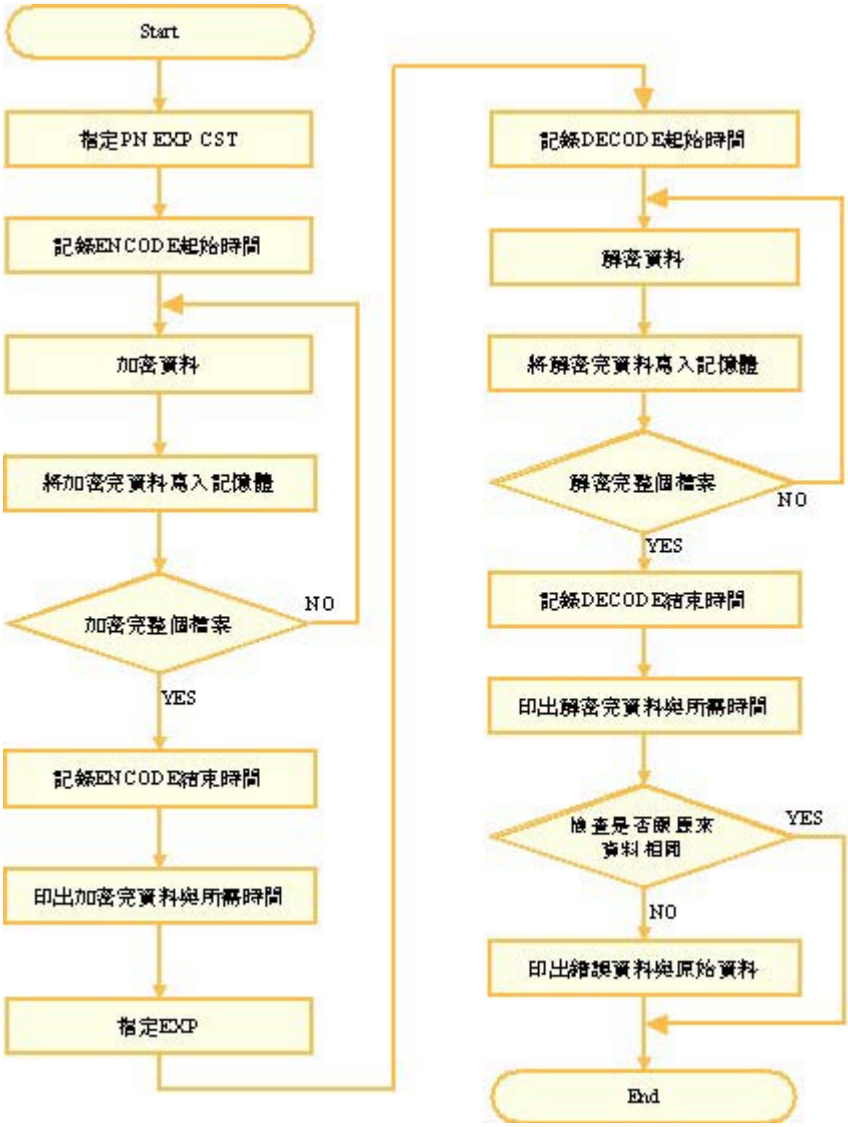


圖 5.4-2 RSA.C 程式流程圖

為了方便起見 Custom Instruction 只做 32bit，因為超過 32bit 就必須使用多次 input 跟 output，難度蠻高的，所以這部分我們並沒有去完成他，又因為 RSA 運算會出現 OVERFLOW 的因素，所以我們的公鑰跟私鑰都只用 16bit；而又限制於公鑰跟私鑰的關係，所以我們每次加解密的資料只有 8bit。雖然感覺上很沒有效率，但是因為運算相對比較簡單，單一執行時間縮短，所

以速度不會慢，只是在實際運用上因為 KEY 值比較小，所以比較容易被破解，但 CI 的目的只是在測試增加指令對效能影響，所以我們重點當然就擺在能加快多少。

5.5 架構二比較

輸入檔案大小：840Byte。

5.5-1 使用 CUSTOM INSTRUCTION 的 MOD 與 MUL

加密所需 clock cycles：2009821。
解密所需 clock cycles：1635214。

5.5-2 使用 CUSTOM INSTRUCTION 的 MOD 與 C 的 MUL

加密所需 clock cycles：2475157。
解密所需 clock cycles：1900655。

5.5-3 使用 C 的 MOD 與 CUSTOM INSTRUCTION 的 MUL

加密所需 clock cycles：8197002。
解密所需 clock cycles：6936640。

5.5-4 使用 C 的 MOD 與 MUL

加密所需 clock cycles：13646013。
解密所需 clock cycles：10743214。

第六章　以硬體元件實現 RSA 系統架構（架構 3）

6.1　Introduction

理論上，純硬體來做 RSA 加解密速度應該是最快，雖然硬體最大的缺點就是 bit 數的限制，bit 增加則 area 直線上升，而 RSA 演算法就是在位元數越高則越不容易被破解，但考慮到及時加密所需求的速度，硬體加密還是最佳選擇。本章將把 RSA.VHD 所組成硬體整合到 Nios System 內，當作純硬體周邊使用。

6.1.1　Polling & Interrupt 簡介

在一般的電腦系統裡，當裝備需要系統來服務時，有二種方法 : Polling 與 Interrupt。

Polling 是由 CPU 一直去問裝備是否需要服務，如果需要時就去服務它。由於周邊的速度太慢，如果 CPU 為了等待一個周邊完成工作就會浪費太多時間等待。

應用 Interrupt 的作法就是將資料丟給周邊以後 CPU 就繼續執行其他工作，直到周邊完成工作以後再發出 IRQ（Interrupt Request）通知 CPU 來處理。這樣可大大減小 CPU 的負擔。一般來說 Interrupt 發生時，CPU 會停止正在進行的工作，先去處理中斷事物。在 Nios CPU 中，每個加入的設備就如同電腦一般，一樣可以使用中斷的方式來通知 CPU 處理。

6.1.2 Interrupt Service Routines（ISRs） and IRQ

當中斷發生時，系統會提供一個 Interrupt Service Routines （ISRs），來專門處理發生 Interrupt 所執行的動作。Nios Development Kits 支援用 C 程式來對自訂的周邊設計個別的 ISRs，而且用了一種蠻特別的方式來支援。

IRQ（Interrupt Request）代表中斷請求，他也代表一組號碼，用來區分每個不同的 ISR，而 ISR 程式的起始位置會放在 Vector Table（中斷向量表）內，當 CPU 收到 IRQ 以後就去查 Vector Table 的對應 ISR 位置，然後執行。

6.1.3 Exception Handling 流程

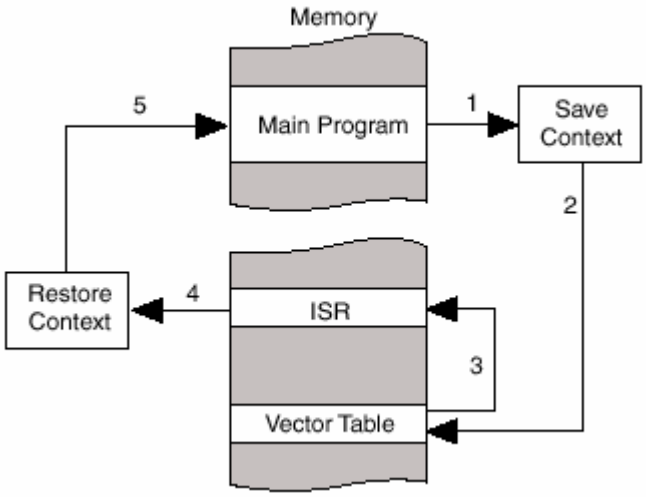


圖 6.1-1 Exception Handling 流程圖

如圖示 6.1-1 所示，就是當發生 Interrupt 時，CPU 進入 Exception Handling 處理流程：

1. 把CPU現在的context儲存起來。
2. 收到IRQ號碼以後去查詢Vector Table來得到ISR的Address。
3. 跳到ISR Routine，然後執行直到完成為止。

4．把之前的儲存的context回覆回去。

5．繼續執行程式。

Nios也支援nested interrupts，也就是說允許有優先權的interrupts。當高優先權的interrupts發生時，就會中斷正在執行的ISR routine，而在高優先權的ISR routine結束前，低優先權的ISR routine就必須一直等待，當中斷連續發生時，就如同nested一樣，一層必須等待一層，最低優先權的ISR routine會等到所有ISR routine都完成後才能繼續執行。

6.2 System Design Framework 3

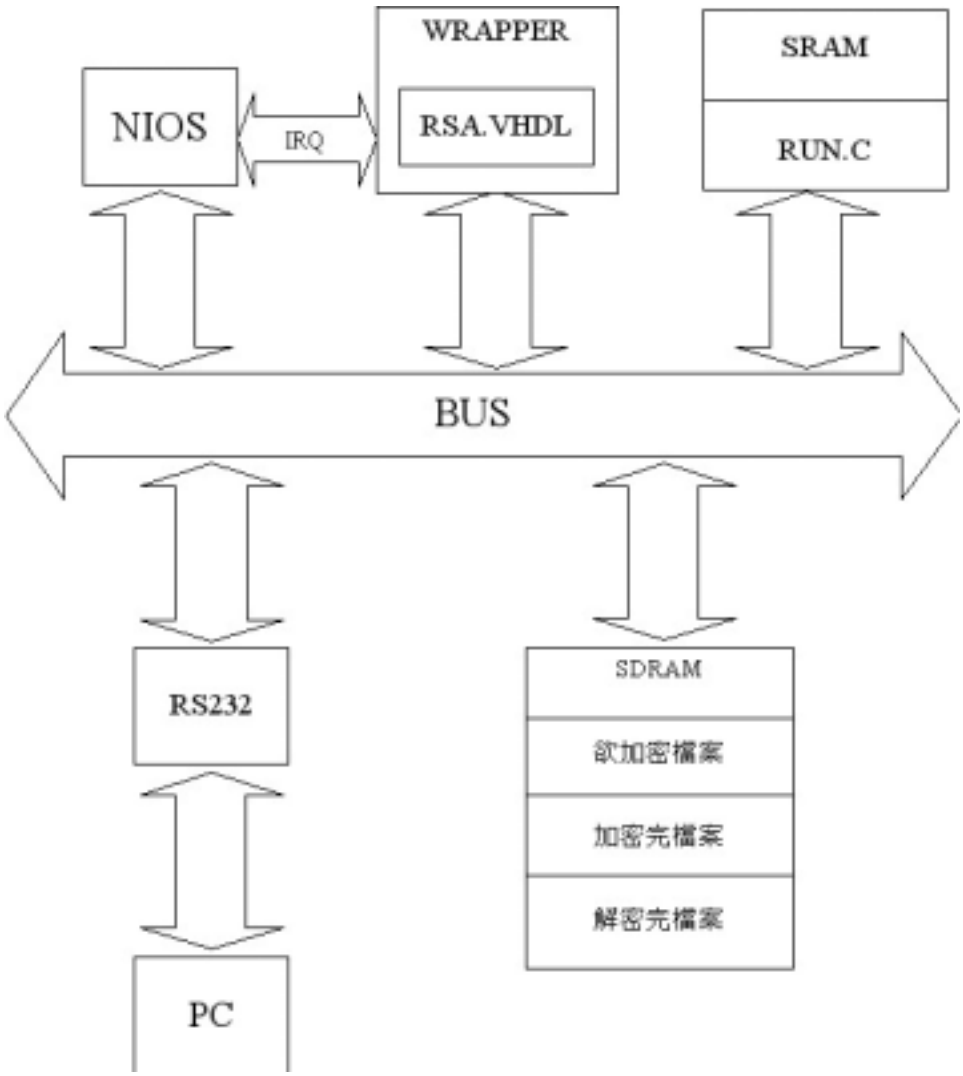


圖 6.2-1 System Design Framework 3

RSA.VHDL 外面的 WRAPPER 是用來包裝 RSA.VHDL 用的，他負責 RSA 與 BUS 上的溝通。由於 BUS 的資料固定為 32bit 而這架構是 64bit，而且必須考慮到資料能被正確讀取所需的時間，所以必須用 WRAPPER 來控制這些動作。

由於我們設計 RSA 這個硬體架構將能同時支援 Polling 跟 IRQ 的方式，所以 WRAPPER 的設計就必須整合這兩種。

主程式 RUN.C 放在 SRAM 裡面，因為同時支援 IRQ 跟

Polling 上，所以 C 的程式也有兩種。

所有的資料都放在 SDRAM 裡面，我們把 SDRAM 分割 3 部分，方便讓我們存放原始資料，加密完的資料，跟解密完的資料，這樣也可以方便我們最後比對加解密的正確性

6.2.1 RSA-WRAPPER Framework

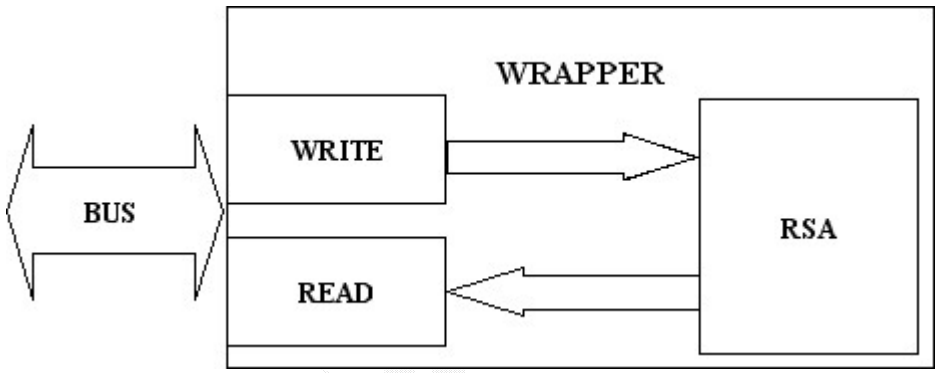


圖 6.2-2 WRAPPER Framework

WRAPPER 主要由三個 COMPONENT 構成，其中 RSA 是就是主要的 RSA 硬體部分，WRITE 是控制從 BUS 上接收資料，經過處理之後再傳給 RSA 處理，READ 是 RSA 加解密完之後，控制資料丟到 BUS 上。

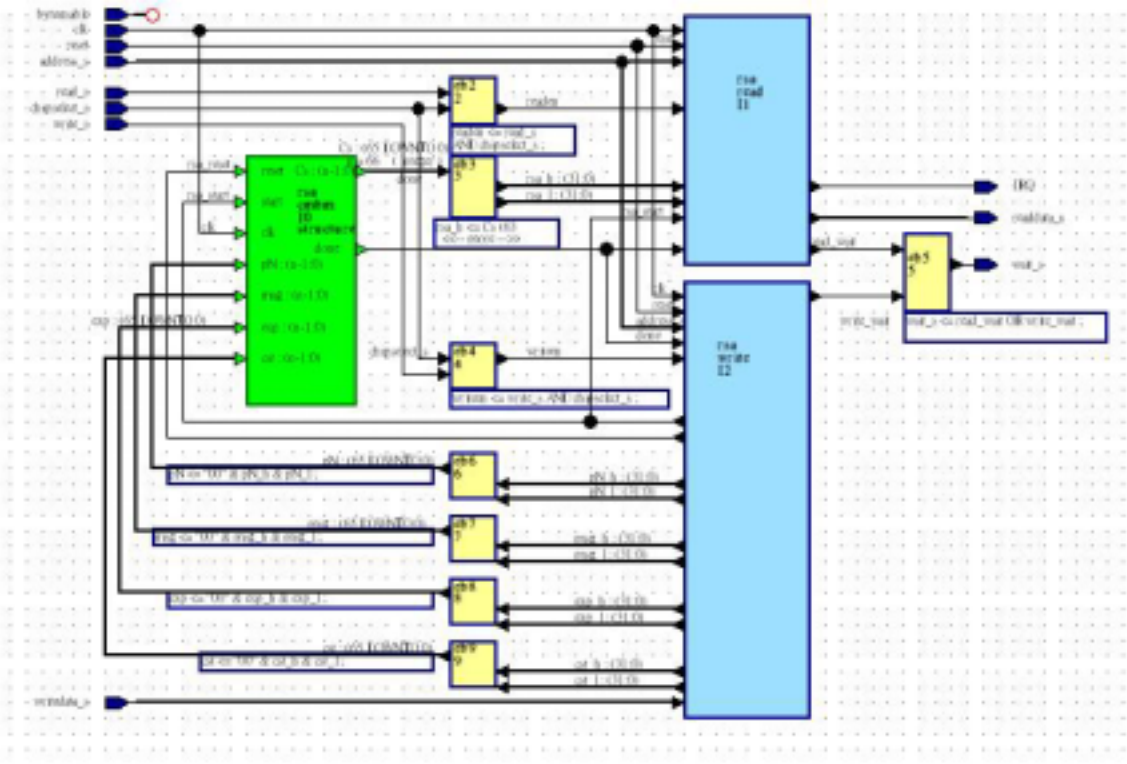


圖 6.2-3　WRAPPER 元件設計圖

6.2.2 RSA-WRITE framework

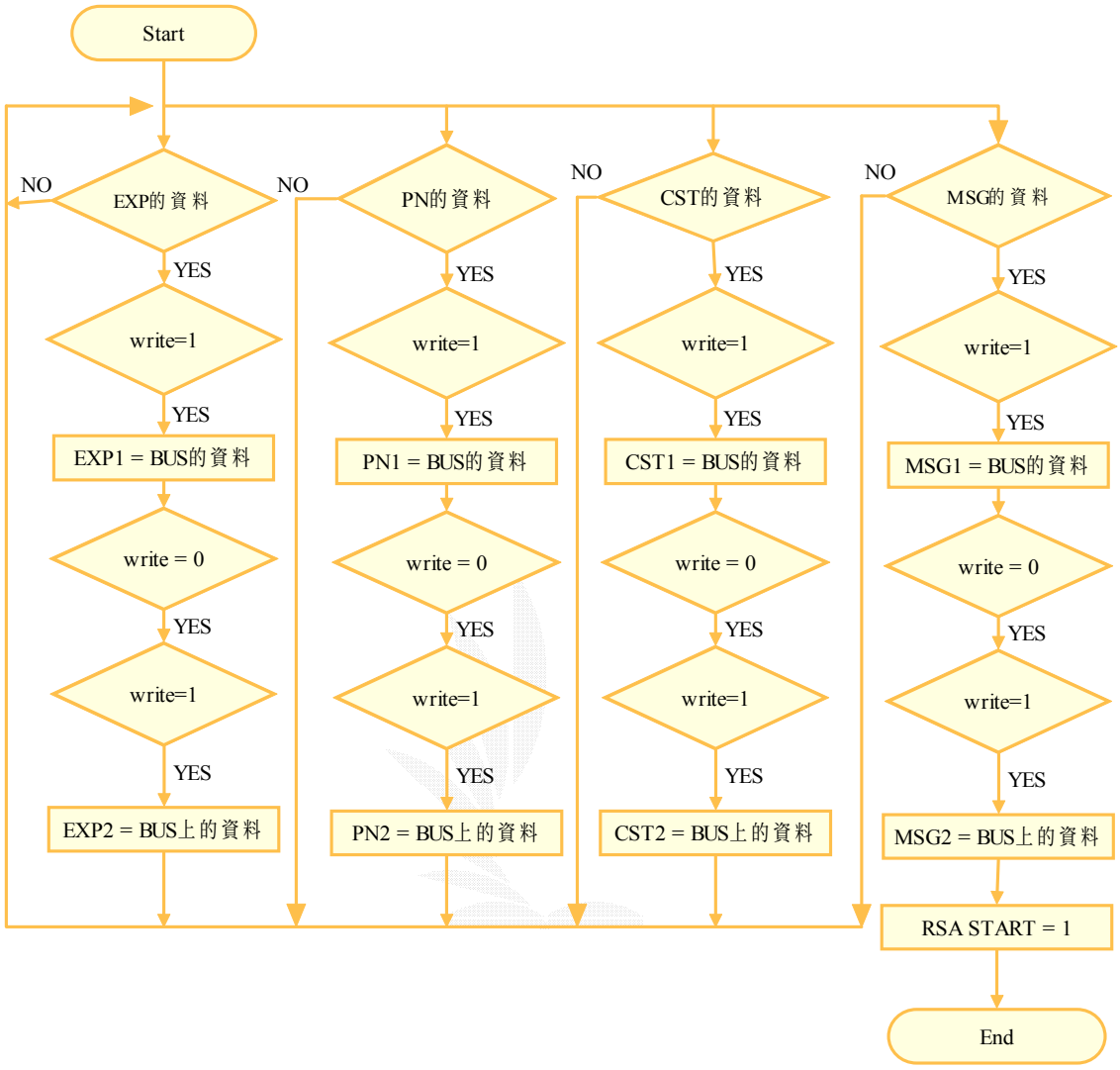


圖 6.2-4 RSA-WRITE 流程圖

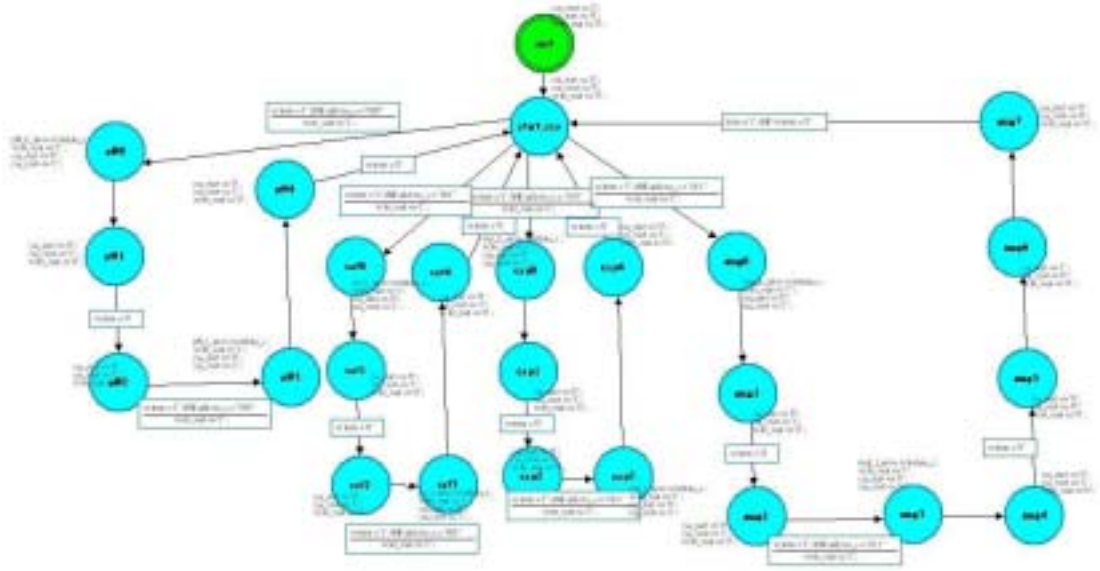


圖 6.2-5　RSA-WRITE FSM 設計圖

WRITE 的工作是控制 BUS 上來的資料，經過處理之後傳給 RSA 做加解密。因為 RSA 所需要的資料有四組，而 BUS 最多只能傳 32 bit 的限制，所以 CPU 要把四組資料要拆成八次送，而 WRITE 便要分辨從 BUS 上收到的資料是屬於哪一種，然後再將收到的兩個 32 bit 資料組合成 64 bit 的資料，再傳給 RSA 做加解密。

WRITE 必須利用 CPU 所傳過來的 ADDRESS 來判別所收到的資料是哪一種，等到四組資料收齊完成，傳送 START 訊號給 RSA，通知 RSA 開始做加解密。

6.2.3 RSA-READ Framework

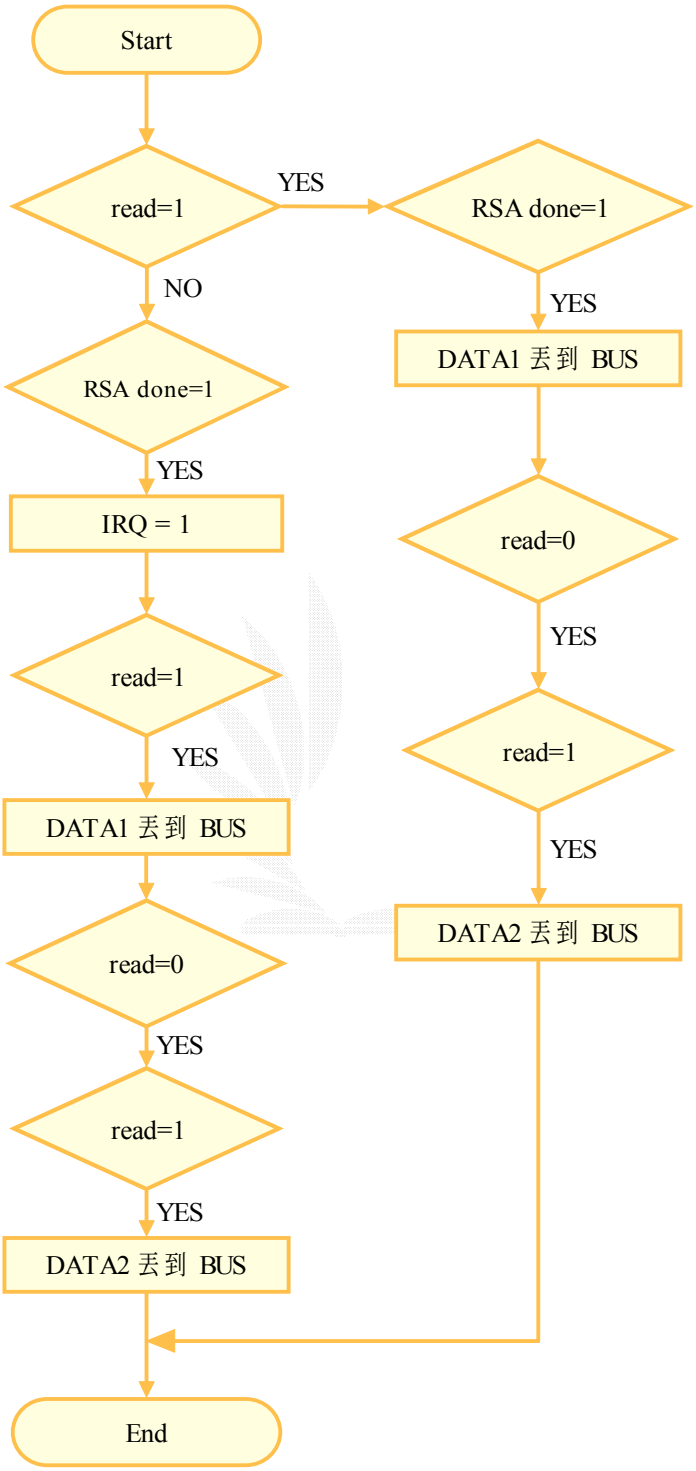


圖 6.2-6 RSA-READ 流程圖

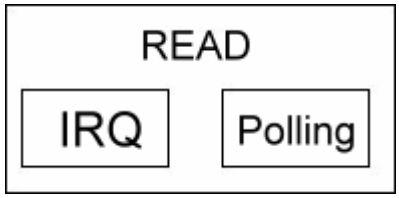


圖 6.2-7　RSA-READ 架構圖

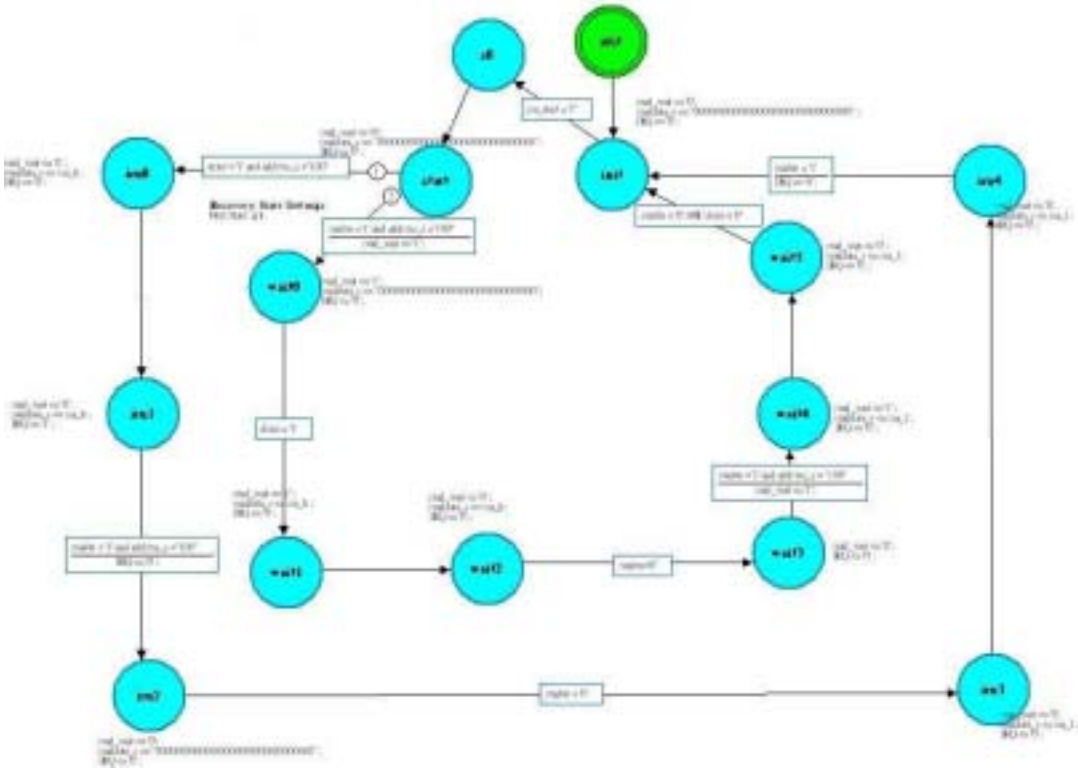


圖 6.2-8　RSA-READ FSM 設計圖

READ 的工作就是把做完 RSA 加解密完的資料丟到 Bus 上。我們用了兩種做法：Polling 與 Interrupt。

Polling 的做法是：當 CPU 發出 READ 訊號時，Polling 便開始動作，當 READ 收到 RSA 已經做完的訊號時，READ 便將 64 bit 資料分兩次丟（因為 BUS 的限制），等到丟完資料後便回到等待狀態，所以，CPU 要不斷的送出 READ 訊號來確認 RSA 是否完成。

Interrupt 的做法是：CPU 沒發出 READ 訊號，READ 又收到 RSA 已經做完的訊號時，READ 便送出一個 IRQ 訊號，當 CPU 收到 IRQ 時，便去查 Vector Table(中斷向量表)執行 ISR，然後送出

READ 訊號通知 WRAPPER 去送出資料，就跟 Polling 的做法一樣，把 64 bit 的資料分兩次丟到 BUS 上。

6.3 RUN.C 簡介

RUN.C 就是所執行的程式，共兩個 Polling.c and Interrupt.c。

6.3.1 Polling C 簡介

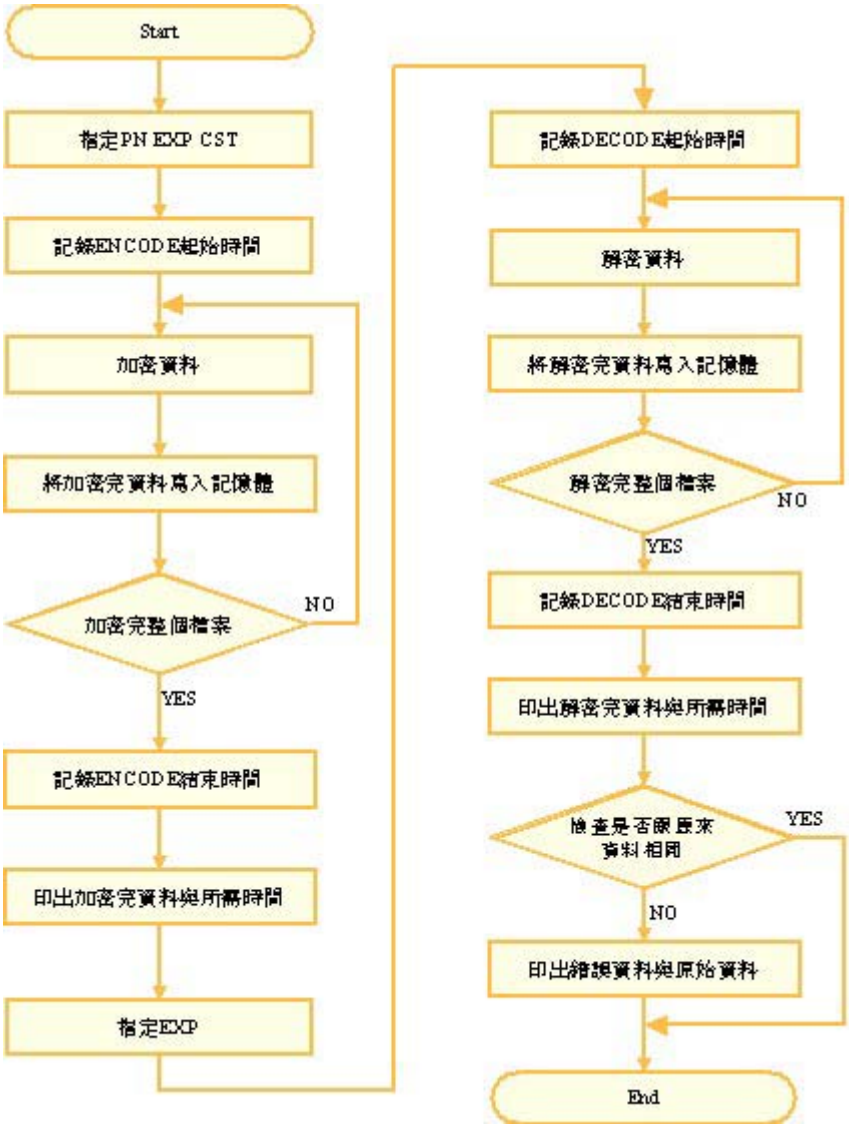


圖 6.3-1 POLLING C 程式流程圖

因為 RSA 運算的限制（會 OVERFLOW），所以這個程式的公、私鑰都是 64bit，而加密資料一次是 32bit（其實最大可到約 58bit，但是使用 32bit 資料會比較好處理，所以採用 32bit），但是 32bit 的資料加密完之後，會產生 64bit 資料，所以加密完的資料便會是原來資料的兩倍大；而解密的時候，不需考慮 OVERFLOW，所以直接把 64bit 資料丟回去解密就可以，而解密完後的資料大小跟原來的資料大小一樣。

這個 C 程式執行前會經過 InterNios 編寫產生檔案大小以及公、私鑰等資料，開始執行時負責一直傳資料給包住 RSA 的 WRAPPER，再傳資料的時候，會先去查表找到要送的資料所對應到的 ADDRESS 之後，便分成 32bit 傳送。

等到一傳送完資料，程式便等待 RSA 把加解密完的資料傳回來，然後寫到記憶體裡面。等到做完加密與解密後，便開始對照解密完資料是否跟原來的資料相同，如果不一樣，便會印出不一樣的資料與所在的位址。

6.3.2 Interrupt C 簡介

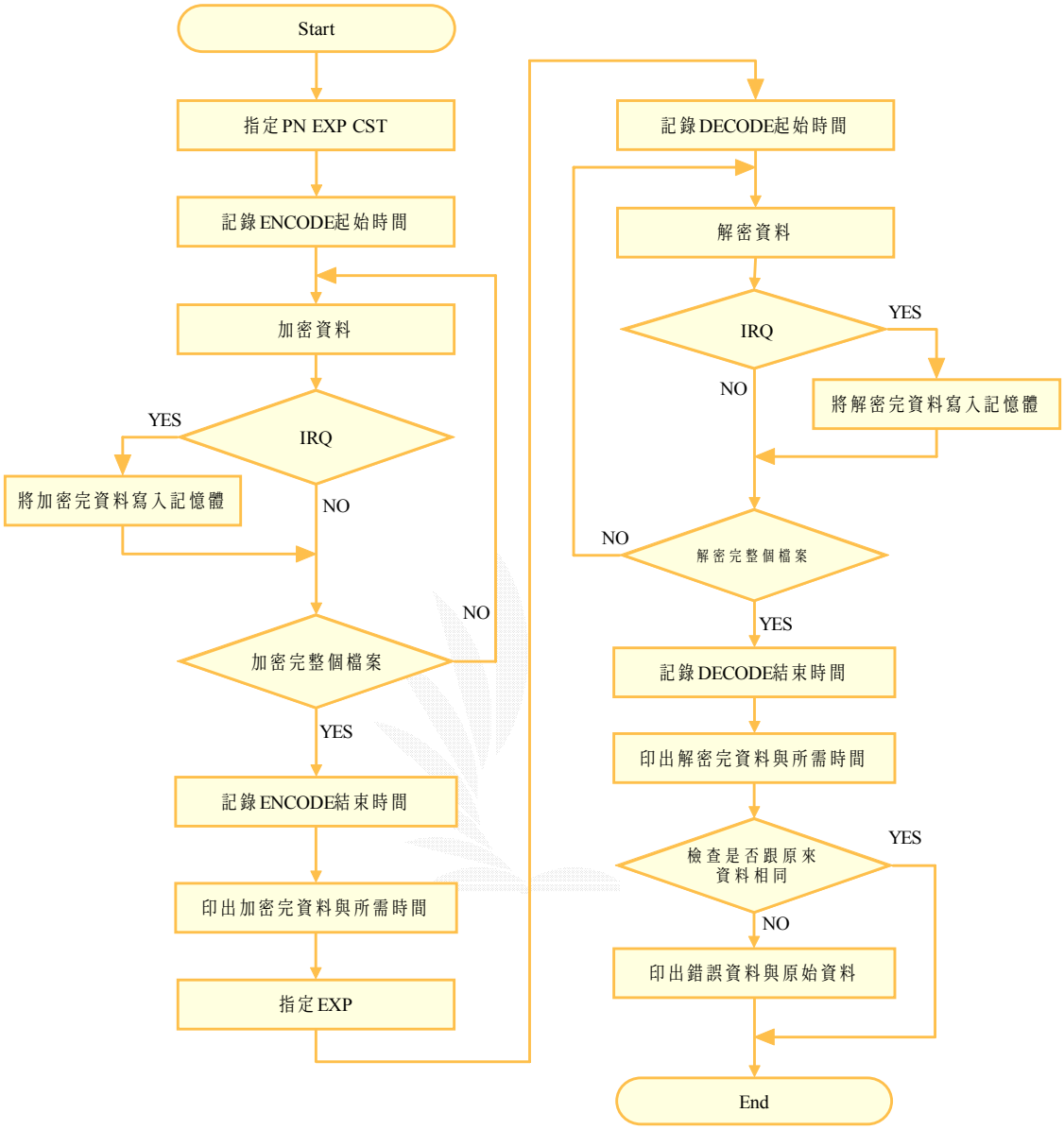


圖 6.3-2 INTERRUPT C 程式流程圖

與 Polling 的 C 程式一樣，因為 RSA 運算的限制，所以這個程式的公、私鑰都是 64bit，而加密資料一次是 32bit。

Interrupt 的 C 程式執行前一樣經過 InterNios 編寫產生檔案大小以及公、私鑰等資料，開始執行的傳送資料方式跟 Polling 相同。與 Polling 的 C 程式不一樣的地方是，Interrupt 的 C 程式不會一丟完資料就等著接收加解密完的資料，也就是 CPU 可以

去做其他事情，RSA 完成工作以後會發出 IRQ，等收到 IRQ 訊號之後，便進入 Exception Handling，根據 IRQ 查詢 Vector Table 找到 ISRs，然後才接收資料；等到接收完資料以後，便回到原來程式中斷的地方繼續執行程式。

6.4 Leonardo Spectrum 預估 RSA-WRAPPER 資料

執行速度：106.0 MHz。

所需邏輯單元：2383。

6.5 Interrupt 與 Polling 速度比較

Polling：

加密：Number of clock cycles: 949464。

解密：Number of clock cycles: 847733。

Interrupt：

加密：Number of clock cycles: 1076226。

解密：Number of clock cycles: 1074289。

Interrupt 的方式會比 Polling 慢的原因是因為 Interrupt 多了 CPU context 存取的動作，加上查詢 Vector Table 的時間，所以 Interrupt 反而花了較多的時間。

第七章　架構比較

架構一單純使用CPU執行C程式，因此架構最簡單，因為速度的關係，所以我們把最耗時間的兩個指令：MOD與MUL做成CUSTOM INSTRUCTION，減少RSA執行的時間。但是利用這兩者還是不夠快，所以我們掛上RSA的硬體元件來做加解密。

用C程式做加解密，pure soft只要系統架構正確，C code也正確就沒問題了，所以架構非常簡單；而利用CUSTOM INSTRUCTION輔助執行的速度，不但速度快、發展的時間也短；利用硬體來做RSA加解密，速度最快，但是架構較複雜，沒有Hardware Debug的工具，處理起來更耗費時間。所以如果要取得發展時間跟速度的平衡，使用CUSTOM INSTRUCTION是個不錯的解。

	ENCODE最短時間	DECODE最短時間	速度比
架構一	436838764	426109310	1
架構二	2009821	1635214	237.7
架構三	949464	847733	480.16

表7-1　架構時間比較表

表7-1示，三種架構的速度比較，硬體的速度理論上可以達到軟體的百倍速度，數據上為480倍，還蠻合理的，問題在於架構二CI的地方，速度居然達到237倍，只比硬體慢一半，這主要是因為mod的地方，硬體裡面是用除法器來求餘數，而CI卻是特製的mod器，用空間換取效能，速度比除法器快約兩倍，大量使用mod的情況就把速度提升了。

	KEY大小(bit)	單位加密資料大小(bit)	單位解密資料大小(bit)
架構一	64	32	64
架構二	16	8	16
架構三	64	32	64

表7-2　單位資料大小

這是這三個架構的基本資料單位大小，基本上是以64bit為原則，bit數越大越不容易被破解，但是架構二由於64bit的指令集增加到32bit的CPU內有一定難度，考量這只是做速度比較，並不是真實應用，所以只使用16bit的架構，減低硬體設計複雜度跟成本。

第八章　心得

蕭詣懋：

還記得大三寒假剛進 VLSI 實驗室跟著學長做訓練，彷彿昨日一班。如今，專題都已經近尾聲了。三年級上學期的時候上了王益文老師數位邏輯設計，讓我對硬體產生了極大的興趣，因此選擇王老師作為專題的指導老師，並且進入 VLSI 實驗室學習。感謝王老師的細心教導，還有江昭磊學長、石博元學長的訓練跟協助是幫助我們這一組專題能完成的大功臣。在這個專題中，讓我有機會能知道整個 nios 內嵌入系統的架構並透過工具來架構整個系統，還有透過專題的三個架構讓我更能體會出使用硬體來加快一件工作的速度，並且有機會認識 RSA 加解密的觀念。此次的專題讓我獲益良多，希望將來有機會能在 VLSI 這個領域能有更多的學習跟成長。

楊勝吉：

在大二上到有關硬體的課程時，便產生了極大的興趣，因此在做專題的時候便想做跟硬體有相關的東西，所以到了VLSI實驗室。有關專題的東西，王益文老師跟學長們給了我們很多寶貴的意見，讓我們的瓶頸可以一一解決，所以真的要感謝他們。

跟張家維在一組也已經三年了，默契也很深，在做這個專題的時候，遇到問題還是一樣兩個人一起DEBUG，不然就是他找不到的錯誤是我DEBUG出來的，我找不到的錯誤是他找出來的，所以在

一起工作的時候不會覺得很累。雖然跟蕭詣懋認識的時間沒比張家維長，但是他在看書的速度快，讓我們減少很多時間讀些理論的東西，而增加許多時間實作。

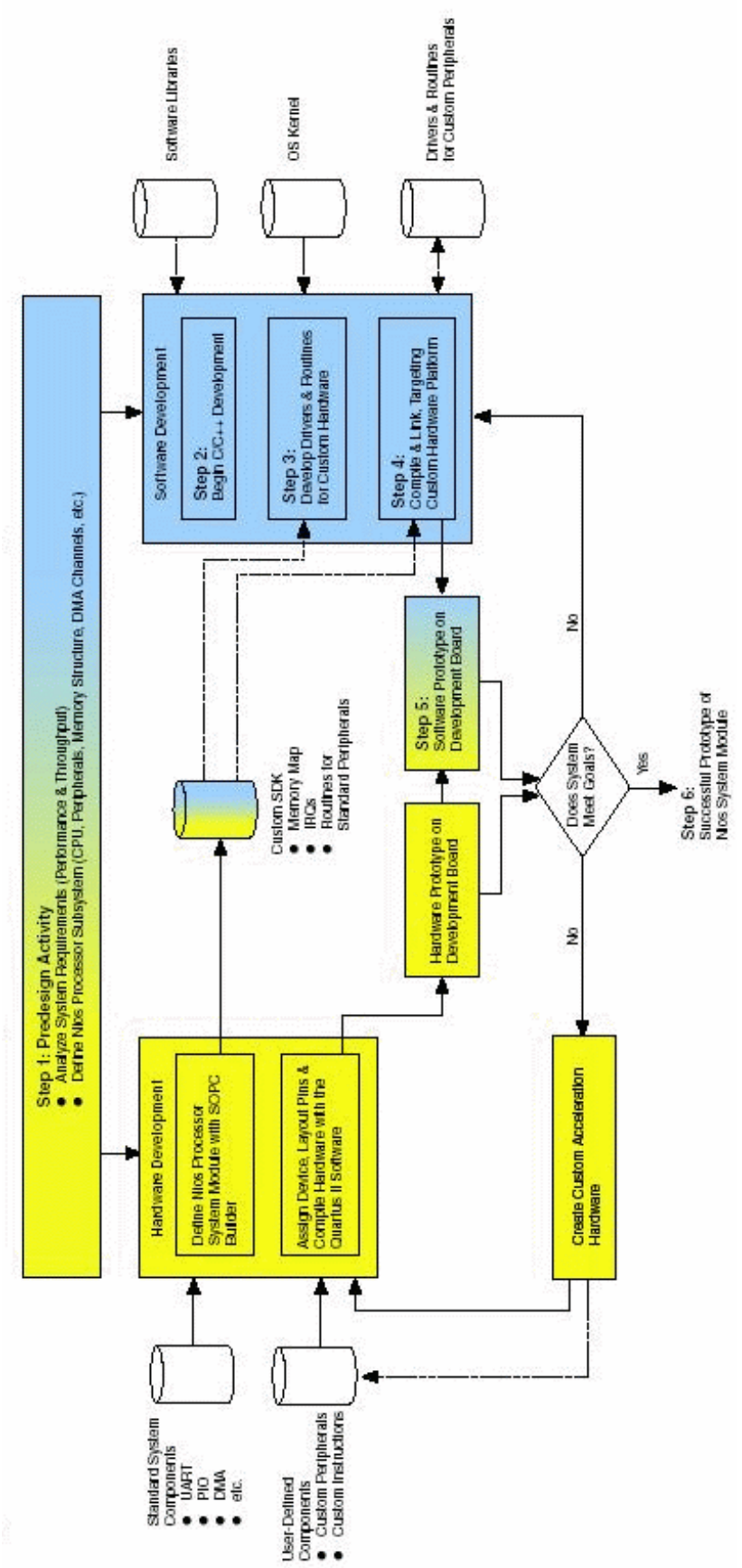
張家維：

Embedded System的確是一個有趣的東西，當瞭解了怎麼去設計以後，就會很想去應用，應用的時候又常常會遇到問題，扯到其他新的東西，於是又回到一開始的拼命瞭解，然後又很想去應用新的東西，就一直這樣下去，對新奇事物的好奇，跟想瞭解的渴望，大概是我做專題最常發生的事情，過程也許很辛苦，甚至困難的地方不見得是技術上，反而是在團體合作上，對整組協調性的考驗，反而是專題最困難的地方。過程中，江昭磊、石博元學長非常辛苦的帶我們研究，幾乎開學前都陪我們到三更半夜，也給我們很多很多建議跟指導，王老師給了許多啟示，遇到問題的時候通常跟老師討論是最快的方法，畢竟我卡了一個禮拜的問題，跟老師討論一下很快就猜到為什麼了。最後非常感謝這些人的辛苦，如果沒有他們的指導，我就不會學到這麼多東西。

附錄一　Nios Development tool kit 簡介

附錄 1.1　Introduction

圖附錄 1.1-1 顯示 Hardware And Software 是怎樣相互作用然後達成我們要求的目標，這圖也是我們這次實作 RSA 的 Design flow，Software And Hardware 即使可以分成許多步驟來完成目標，但是了解基本的流程，可以幫助我們區分 Hardware And Software 各自的需求還有兩者再哪個步驟一樣跟不一樣的地方。



圖附錄 1.1-1　Hardware/Software Development Flow for a Nios Processor System

附錄 1.2　Sopc Builder 簡介

通常當設計者決定了要使用哪些System component，或者設計了一個System component，那麼，設計者可以藉由Sopc Builder來整合整個環境，Sopc Builder會自動產生你所選擇的硬體對應的SDK(Software Development Kit)，如此一來你所選擇或設計的硬體元件，便可以透過SDK來由C CODE來控制，而SDK也整合之後你會用到的software tools。

附錄1.3　SDK (Software Development Kit)簡介

SDK是有關於我們在設計你的硬體的時候所需要的軟體工具集，主要包含了幾個tools。

附錄1.3.1　GNUPro Tools

這主要是包含了一個compiler，還有debugger (command-line GDB and GUI-based Insight)，加上一些software development 工具，像最常使用的Nios-build就是整合在這裡。

附錄1.3.2　Nios On-Chip Instrumentation (OCI) Debug Module

On-chip Debug的方式幫助我們得知硬體的資訊，它提供一個介面跟Nios CPU做溝通，能夠直接做程式流程控制，跟提供memory and register內的值來幫助我們debug，並且可以設定breakpoint。

附錄1.3.3 Nios OCI Debug Console

這主要是用command-line的方式來跟上面的Nios OCI Debug Module做溝通，command內包含了system configuration、emulation control memory access、register access、trace and file downloading、and status indication。

附錄1.3.4 Nios SDK Shell

這是一個base on Unix的環境，主要是由cygwin提供一個UNIX-like environment，Nios SDK Shell提供了上面大部分tools的執行環境，最重要的是，整個程式的執行畫面是在這裡展示的，所以這裡主要功能就是，我們在console畫面，然後下command，而board回應內容到這個Shell下。

附錄1.3.5 Nios SDK(Nios Software Development Kit)

當GNU compilation tools完成compiler後，如果你不想使用SOPC Builder去建立系統，那麼你的硬體必須自己提供一個SDK來讓software去使用它，而設定方法就是依照Nios SDK的格式加入檔案，直接對設定檔做更改，我們如果說SOPC Builder是提供一個GUI介面讓使用者修改系統設定，那Nios SDK就是SOPC Builder背景所更動的設定檔。

附錄1.4　Nios SDK Shell Commands 簡介

Nios SDK Shell Commands	
Command	**Description**
Nios-build	Compiles and links the source code (C and Assembly)
Nios-run	Downloads executable code to a development board and runs it. It is also used as a terminal program to interact with the development board.
Nios-console	Launches the Nios OCI Debug Console.
Nios-debug	Launches the software debugger.
hexout2flash	Converts a hardware design **.hexout** file to a **.flash** file that can be downloaded to a flash device.
srec2flash	Converts a compiled software program S-Record file (.srec) into a flash image that will be executed automatically when the development board is reset.
nios-elf-size	Prints the size of the code, data, and uninitialized storage.
nios-elf-objdump	Creates a .objdump file with disassembly of the **.out** file.
nios-elf-gprof	Creates the execution profile of a C program.

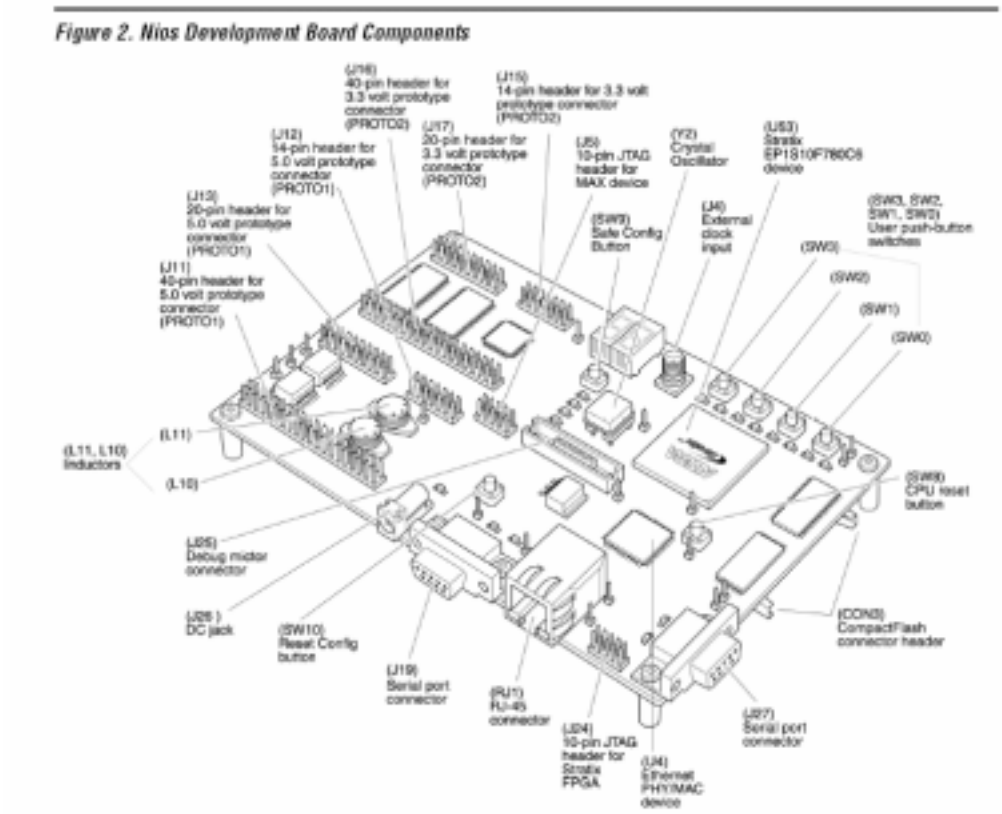
表附錄1.4-1　Command Description

附錄二　硬體環境介紹

Stratix是由Altera公司出產的nios發展工具中的一個版本提供了一個硬體的平台用來發展embedded systems。

Stratix規格如下：

A Stratix EP1S10F780C6 device
_ 8 Mbytes of flash memory
_ 1 Mbyte of static RAM
_ 16 Mbytes of SDRAM
_ On board logic for configuring the Stratix device from flash memory
_ On-board Ethernet MAC/PHY device
_ Two 5-V-tolerant expansion/prototype headers each with access to41 Stratix user I/O pins
_ CompactFlashTM connector header for Type I Compact Flash (CF) cards
_ Mictor connector for hardware and software debug
_ Two RS-232 DB9 serial ports
_ Four push-button switches connected to Stratix user I/O pins
_ Eight LEDs connected to Stratix user I/O pins
_ Dual 7-segment LED display
_ JTAG connectors to Altera devices via Altera download cables
_ 50 MHz Oscillator and zero-skew clock distribution circuitry
_Power-oreset circuitry



圖附錄2-1 Nios Development Board Components

圖附錄2-2 Stratix實體圖

參考資料

[1] 近代密碼學及其應用．ISBN 957-22-1953-7．賴溪松，韓亮，張真誠．松崗電腦．1995.1．12.

[2] 現代密碼學入門與程式設計．ISBN 957-21-1274-0．楊吳泉．全華圖書．1996．1.13

[3] Java 2 Black Book 徹底研究．ISBN 957-527-440-7．Steven Holzner著 張裕益、劉春成譯．博碩文化

[4] JavaTM 2 SDK, Standard Edition Documentation

[5] Altera Nios Tutorial & Nios Documentation