中　興　大　學
電　機　工　程　學　系
專　題　報　告

FPGA 平台
十六位元嵌入式
微處理器設計開發

指導教授:范志鵬

學生:黃琨驊

摘要

FPGA 強大之處在於可輕易的編輯電路能力，可依使用者需求自訂所需求的電路，對於要客製化硬體時，是一個十分實用且強大的工具。

然而通常我們在製作各種硬體元件行為時,都需要重新從邏輯電路上重新設計起，如果有錯誤的部分，便需要重新查找，甚至重新構建新的邏輯電路，這樣的行為十分浪費時間，如果需要在設計非常複雜的操作行為時，需要改動的部分龐大，在開發上顯然十分沒有效率。

因此，若能為 FPGA 開發一顆嵌入式微處理器，並使用這顆微處理器，操控周邊硬體,未來只需要定義一次與微處理器的溝通協議,僅僅透過寫程式的方式,便可以輕易的操控周邊元件。

本文即為採用 HKH-II 十六位元精簡指令集架構(本人設計)，所設計的一款十六位元微處理器，這顆微處理器具有六級流水線架構，具有 8KB 內部程式記憶體，及具有兩組十六位元計數器以及一個外部中斷源。

為了效能上的提升，此微處理器的包含 Data Hazard 的前饋(forwarding)處理以暫存器的前半 clock 寫入後半 clock 讀出處理、JUMP 指令的二位元跳躍分支預測(2-bit prediction)以及管線凍結、LOAD 指令的暫停、IP CORE 的 Clock wizard 超頻使用等等。

中斷介面採用硬體中斷，內部具有計時器，可利用計時器進行中斷，也可利用外部中斷源進行中斷，此外部中斷方式採用邊緣觸發中斷，中斷向量須先由軟體進行設定。

目錄

摘要.....1

目錄.....2

圖目錄.....5

第一章 緒論.....8

1.1 研究動機.....8

1.2 研究目的與方法.....9

第二章 Instruction 設計.....10

2.1 Instruction 設計考量.....10

2.2 平台介紹.....10

2.3 各種硬體元件考量.....11

2.4 各種硬體元件考量結論.....12

2.5 各種 Operation code 設計.....12

2.6 內部 RAM 空間設計.....12

2.7 狀態暫存器設計.....13

2.8 各種 Operation code 指令區塊設計.....15

2.9 ALU 指令設計.....16

2.10 LOG 指令設計.....21

2.11 SHF 指令設計.....25

2.12 SETFLAG 指令設計.....28

2.13 LOAD 指令設計.....29

2.14 SET 指令設計.....30

2.15 MOVE 指令設計.....31

2.16 PUSH 指令設計.....32

2.17 JUMP 指令設計.....33

2.18 IN、OUT 指令設計.....34
2.19 CALL 指令設計.....35
2.20 NULL 指令設計.....36
2.21 HALT 指令設計.....36
第三章 設計開發 Microprocessor.....37
3.1 微處理器架構設計.....37
3.2 微處理器 RAM 設計.....38
3.3 微處理器管線設計.....39
3.4 微處理器資料障礙處理.....40
3.5 微處理器跳躍障礙處理.....41
3.6 微處理器跳躍改良.....42
3.7 微處理器頂層模組架構.....43
3.8 微處理器 RAM 模塊介紹.....44
3.9 微處理器 BLOCK 1 模塊介紹.....45
3.10 微處理器 Register 模塊介紹.....46
3.11 微處理器 BLOCK 2 模塊介紹.....47
3.12 微處理器 OUT 模塊介紹.....49
第四章 設計開發 Assembler.....50
4.1 Assembler 開發設計前言.....50
4.2 Assembler 開發設計考量.....50
4.3 COE 檔匯入 RAM 中.....51
4.4 利用 C 語言建構 COE 檔流程.....54
4.5 利用 C 語言建構 COE 檔方法.....55
4.6 完成組譯器開發.....56

第五章 設計周邊元件控制.....59
5.1 周邊元件控制設計前言.....59
5.2 周邊元件控制方法.....59
5.3 周邊元件溝通控制介面結構.....60
5.4 周邊元件 VGA 顯示介面建構範例.....61
第六章 成果展示.....62
6.1 前言.....62
6.2 周邊元件使用介紹.....62
6.3 成果展示.....63
結語.....66

圖目錄

圖 1-2-1.....9
圖 2-2-1.....10
圖 2-9-1.....16
圖 2-9-2.....16
圖 2-10-1.....21
圖 2-10-2.....21
圖 2-11-1.....25
圖 2-11-2.....25
圖 2-12-1.....28
圖 2-12-2.....28
圖 2-13-1.....29
圖 2-13-2.....29
圖 2-14-1.....30
圖 2-14-2.....30
圖 2-15-1.....31
圖 2-15-2.....31
圖 2-15-3.....31
圖 2-15-4.....31
圖 2-15-5.....31
圖 2-15-6.....32
圖 2-15-7.....32
圖 2-16-1.....32
圖 2-16-2.....32
圖 2-17-1.....33

圖 2-17-2.....33
圖 2-18-1.....34
圖 2-18-2.....34
圖 2-19-1.....35
圖 2-20-1.....36
圖 2-21-1.....36
圖 3-1-1 CPU 內部資料流程.....37
圖 3-2-1 Ture Dual Port Ram 模塊圖.....38
圖 3-2-2 RAM 時序圖.....38
圖 3-3-1 微處理器管線設計.....39
圖 3-4-1 微處理器資料障礙處理.....40
圖 3-5-1 微處理器跳躍障礙處理.....41
圖 3-6-1 跳躍分支預測改良圖示.....42
圖 3-7-1.....43
圖 3-8-1.....44
圖 3-9-1.....45
圖 3-10-1.....46
圖 3-11-1.....47
圖 3-12-1.....49
圖 4-2-1.....50
圖 4-3-1.....51
圖 4-3-2.....51
圖 4-3-3.....52
圖 4-3-4.....52
圖 4-3-5.....53

圖 4-4-1.....54
圖 4-6-1.....56
圖 4-6-2.....57
圖 4-6-3.....57
圖 4-6-4.....58
圖 5-2-1.....59
圖 5-3-1.....60
圖 5-4-1.....61
圖 6-2-1.....62
圖 6-3-1.....63
圖 6-3-2.....63
圖 6-3-3.....64
圖 6-3-4.....64
圖 6-3-5.....65
圖 6-3-6.....65

第一章 緒論

1.1 研究動機:

FPGA 平台在實作各種硬體的時候，可以輕易地操作各種平台上的元件，譬如說，在訂立 LED 燈的各種行為時，或是利用 SWITCH 開關操縱各個元件的行為等等，透過 VERILOG 設計各種的電路，使用邏輯電路控制各個元件的行為，對於要客製化硬體時，是一個十分實用且強大的工具。

然而通常我們在製作各種硬體元件行為時,都需要重新從邏輯電路上重新設計起，如果有錯誤的部分，便需要重新查找，甚至重新構建新的邏輯電路，這樣的行為十分浪費時間,如果需要在設計非常複雜的操作行為時，需要改動的部分龐大，在開發上顯然十分沒有效率。

因此，如果能盡量不要在硬體上面下手，只需要改動程式記憶體的部分，將需要操縱的硬體元件行為模式，寫在記憶體中，如此一來，不需要改寫龐大的硬體，僅需要透過改動記憶體的方式，便可以使 FPGA 做出需要的操作行為。

也就是說，若是能在 FPGA 平台上，設計並建立一個微處理器，透過這顆微處理器讀取記憶體中的資料，便能夠操控 FPGA 周邊的元件，未來僅僅只需要對記憶體中的資料作改動，並不需要對硬體做改動，便可以輕易的製作出各種硬體元件行為，這樣在開發上能更有效率，甚至可以製作更加龐大的設計。

1.2 研究目的與方法:

本研究希望可以發展出一套適用於 FPGA 平台的微處理器，透過這顆微處理器，支援各種 FPGA 周邊元件的控制，以及更加複雜的操作，同時也希望未來能在 FPGA 平台上建立起一套系統架構。

目標如下(如圖 1-2-1):

1. 第一步，考量 FPGA 平台的各種周邊元件，設計出具有實用性、有效率的 Instruction Format(指令格式)。
2. 第二步，設計具有效率的 Microprocessor(微處理器)，為提升 Microprocessor 的效率，將 Pipeline 架構加入設計中。
3. 第三步，發展一套 Assembler(組譯器)，將組合語言轉譯成 Machine Language(機械語言)，方便開發使用。
4. 第四步，建立微處理器與周邊元件溝通控制介面。

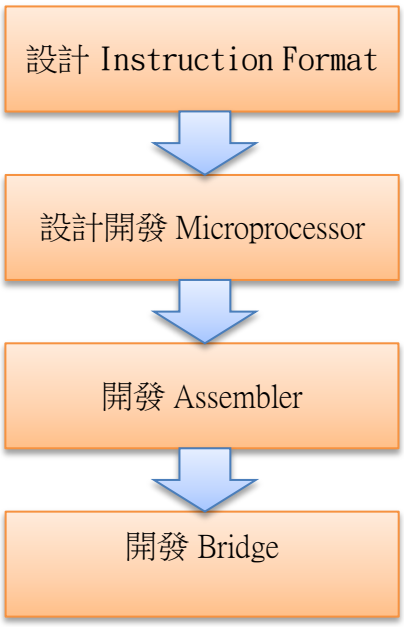

圖 1-2-1

第二章 Instruction 設計

2.1 Instruction Format 設計考量:

在設計平台的時候，需要先對 Instruction Format 做設計，從 Instruction Format 開始設計完成後，才開始進行開發硬體以及軟體部分，因此 Instruction Format 需要先評估各個元件的特性，從以適合操控各個元件的先決條件中去設計 Instruction Format，此 Instruction Format 被命名為 HKH-II，

2.2 平台介紹:

圖 2-2-1

如圖2-2-1，板子上的晶片為***Xilinx XC3S2000-FG676***，具有LCDM 16/2, Switch, Button, Rotary push-button Switch, KeyPad, User Pins, 7_Seg, 5x7 Dot Matrix, LED, RS232, VGA, PS/2, AUDIO以及256 x 16bit High Speed Static RAM和512 SDRAM。

2.3 各種硬體元件考量:

1. 256 x 16bit High Speed Static RAM(SRAM):

   SRAM具有18根Address腳位，與16根DATA腳位，這意味著，每一個CLOCK都可以讀取與寫入一筆SRAM的16-bits資料，同時相當於256k-16bits的儲存空間。
2. SDRAM:

   同樣的，每次讀取與寫入都是16-bits資料。
3. LED:

   LED共24顆，Red,Green,Blue三色LED燈各8顆，如果要控制LED燈，可以分開來控制，因此最少需要8-bits。
4. LCDM 16*2:

   控制每個字元為8-bits，所以需要有8-bits的資料空間。
5. SWITCH:

   SWITCH總共為16根腳位，可以為16-bits的資料空間，或是分開兩個來讀取，各為8-bits的資料空間。
6. 7-segment:

   每個7-segment元件都需要8-bits的資料空間。
7. KEY-PAD:

   KEY-PAD有12顆鍵位，僅需要4-bits的資料空間便可以表達其行為。
8. Rotary push-button Switch:

   同樣有12個檔位，僅需要4-bits的資料空間便可以表達其行為。
9. PS/2(Keyboard):

   輸入字元使用，需要8-bits的資料空間。
10. GPIO:

    其腳位從7至20共14根，需要10- bits的資料空間。
11. VGA:

    因無VGA DA晶片，無法擁有高色彩表現，只可表現出八色， 每一個像素需要3-bits的資料空間。
12. 5x7 Dot Matrix:

    需要建立Array群，用以控制每行或每列的燈號，需要另外設計其顯示規則，故現階段無法判定需要多少資料空間，暫定為7-bits。
13. AUDIO:

    僅需一個bit。
14. RS232:

    串列發送與接收資料，依照常規的通訊協定，需要8-bits的資料空間。

2.4 各種硬體元件考量結論:

比照各種的硬體周邊元件發現,如果以16-bit去做設計,剛好可以符合從記憶體讀出的數據量,並且對於各個硬體元件來說,各種的資料空間都可以得到滿足,同時16位元的設計,較之8051等微處理器,能有較多的指令個數,以及更高的數值計算,可以得到較高效能,且適用於各種的硬體元件端。

2.5 各種 Operation code 設計:

為滿足最基本的操控設計,必須包含如下的指令:

1. ADD 加法指令
2. SUB 減法指令
3. AND 基本邏輯指令
4. OR 基本邏輯指令
5. XOR 基本邏輯指令
6. NOT 基本邏輯指令
7. LOAD 讀取指令(從記憶體中讀取特定資料)
8. STORE寫入指令(將特定資料寫入記憶體中)
9. JUMP 跳躍指令(需要在特定條件的時候進行指令位置跳躍)
10. IN、OUT 輸入輸出指令

這幾項指令為微處理器中最基礎的設計,如果缺少了其中一種指令,微處理器便無法正常地進行其運作,也無法發展出其必要有的控制、及資料運算表現。

因為決定這個顆微處理器為16-bits的資料架構,可以有充足的Instruction的指令空間來定義各種的Operation Code,因此可以對基本的指令進行擴充,令指令可以支持較為複雜的計算以及控制。

2.6 內部 RAM 空間設計:

考量晶片內部所具有的RAM容量,此平台具備約七十三萬位元容量,然而考量到此十六位元嵌入式微處理器,是為FPGA平台開發的一套通用性微處理器,因此在最初的容量考量設計上,採用此Spartan-3系列,最低限度可提供的RAM之容量容量共4個BLOCK,73728位元的記憶容量,所以此十六位元嵌入式微處理器總共擁有4096行的指令空間,相當於4K-word。

2.7 狀態暫存器設計:

為了能夠進行快速的計算,於是設計了16個16-bits暫存器,分別為R0~R15,以及為了能夠進行各種數值判斷、分析或是資料接收、傳遞或是中斷指令等等,所以加入了Status register(狀態暫存器) 或是可以稱呼為Flag register(旗標暫存器)。

狀態暫存器設計以下表格:

CY	OF	SF	Z	IF	OF	EQ	BT	ST	ITR	IEN	IT1	IT0	X	X	X

CY(Carry Flag進位旗標):

用來判斷運算後,第十六位元的運算是否有進位。若有、則CY為 1,若無、則為0。

OF(Overflow Flag益位旗標):

因此微處理器目前規劃為無號數,因此尚未加入設計之中,可供未來擴充開發使用。

SF(Sign Flag 符號旗標):

用來判斷運算後,第十六位元的運算是否有負值。若有、則SF為 1,若無、則SF為0。

Z(Zero Flag 零值旗標):

因跳躍指令採用直接判斷暫存器內部是否為零值,無須零值旗標,但可供未來擴充開發使用。

IF(Input Flag 輸入旗標):

在最初始版本的微處理器設計上,需要透過輸入旗標判斷是否有數值確切傳入,但為了增加指令運算速度以及減少微處理器的複雜程度等原因,並未加入後期的微處理器設計上,但可供未來擴充開發使用。

OF(Output Flag 輸入旗標):

在最初始版本的微處理器設計上,需要透過輸出旗標判斷是否有數值確切傳出,但為了增加指令運算速度以及減少微處理器的複雜程度等原因,並未加入後期的微處理器設計上,但可供未來擴充開發使用。

EQ(Equal Flag 相等判斷旗標):

透過特定的判斷指令集,若兩值相等、則EQ為1,若兩值不相等、則EQ為0。

BT(Bigger Than Flag 大於判斷旗標):

透過特定的判斷指令集,若左側值大於右側值、BT為1,若左側值不大於右側值、則BT為0。

ST(Smaller Than Flag 小於判斷旗標):

透過特定的判斷指令集,若左側值大於右側值、BT為1,若左側值不大於右側值、則BT為0。

ITR(Interrupt Request Flag 外部中斷旗標):

用來判斷是否有外部中斷要求，若外部有中斷要求、則ITR為1，若無外部有中斷要求、則ITR為0。

IEN(Interrupt Enable Flag 中斷允許旗標):

用來判斷是否可以執行中斷，若IEN為1時、允許中斷，若IEN為0時、不允許中斷。

IT1(Interrupt Timer 1 Request Flag 內部時鐘1中斷旗標):

用來判斷是否有Timer Counter 1中斷要求，Timer Counter為獨立分開的計數單位，具有16-bit位元的儲存寬度，Timer Address為Timer Counter發生中斷時的挑躍位置，具有12-bit位元的儲存寬度。若Timer Counter 1的內部數值大於零值，則每一個CPU clock Cycle會將內部數值減一。Timer Counter 1減至為0時、IT1為1；Timer Counter 1恆為0時、IT1為0。若內部計時器中斷發生，則從Timer Address 1讀取跳躍絕對位置，可跳躍至RAM任一處。

IT0(Interrupt Timer 0 Request Flag 內部時鐘1中斷旗標):

用來判斷是否有Timer Counter 0中斷要求，若Timer Counter 0的內部數值大於零值，則每一個CPU clock Cycle會將內部數值減一。Timer Counter 0減至為0時、IT0為1；Timer Counter 0恆為0時、IT0為0。若內部計時器中斷發生，則從Interrupt Timer Address 0讀取跳躍絕對位置，可跳躍至RAM任一處。

X(未指定功能旗標0):

尚未指定其功能旗標，供未來擴充使用。

2.8 各種 Operation code 指令區塊設計:

擴充指令部分可以分為數個區塊設計，如下:

NULL指令:空白指令，無任何操作，可避免程式誤讀錯誤指令。

ALU指令:供基本運算使用，譬如加法指令、減法指令等等。

LOG指令:供邏輯運算使用，譬如AND、NAND、NOR等等。

SHF指令:供位移位元使用，譬如Shift、Rotate。

SETF指令:供設定Flag旗標使用，將旗標設為1或0。

LOAD指令:可使資料寫入記憶體中或是將資料從記憶體中讀出。

SET指令:將需要寫入的數值，放置入暫存器中。

MOVE指令:搬移指令，包括搬移計數值進入計時器中，以及中斷時需要跳躍的位址。

PUSH指令:將數值放置入堆疊暫存器中堆疊，數值會後進先出的方式讀出。

JUMP指令:判斷各種情況做跳躍,最大跳躍距離為向上或向下各255行指令間距。

IN、OUT指令:對外部裝置進行控制以及接收外部裝置資料。

CALL指令:遠程跳躍位置，可以進行較遠程跳躍，呼叫副程式，並記錄初始跳躍位置，回到主程式中。

HALT指令:程式中斷、休眠，須透過外部中斷或Timer中斷訊號回復運行。

2.9 ALU 指令設計:

ALU 用在各種運算使用，其各種指令需要有共通性，如此設計可以在電路輸入與輸出上節省較多線路也較有效率。

考慮到FPGA平台上並不需要過於複雜的計算,以及為了顧全指令的多樣性，盡可能利用 4-bit 的 OPCODE 指令寬度，在這個前提下，在 ALU 的運算上，並沒有採用有號數以及浮點數的設計。指令格式設計如下圖 2-9-1:

4-bits	3-bits	3-bits	3-bits	3-bits
OPCODE	目標暫存器(IN2)	來源暫存器 1(IN1)	來源暫存器 0(IN0)	MODE

圖 2-9-1

為了增加指令個數,所以將來源暫存器與目標暫存器使用的位元數減一個位元，意味著在原本設計使用的 16 個暫存器 R0~R15 只用了其中的 R0~R7。所以在使用 ALU 的運算要注意的地方是運算只會使用到 R0~R7，若需要 R8~R15 的話，便需要使用 MOVE 搬移指令,將數值從 R8~R15 搬移至 R0~R7 中使用或者將數值從 R0~R7 搬移至 R8~R15 中使用。

MODE 則是為了擴充指令數使用。如下圖 2-9-2:

OPCODE(4-bits)	MODE(3-bits)	INSTRUCTION(指令)
0001	000	ADD
0001	001	SUB
0001	010	MUXA
0001	011	MUXB
0001	100	BTD
0001	101	INC
0001	110	DEC
0001	111	CLR

圖 2-9-2

ADD 指令組合語言詳述:

Instruction Set:　ADD R(IN2) R(IN1) R(IN0)

Operation:　R(IN2) = R(IN1) + R(IN0)

整個系統上並未加入有號數的設計,所以在設計上目前並未加入溢位旗標,當然在目前的設計上已保留溢位旗標的設計,供未來擴充指令以及修改計算方式使用。

如果兩數相加第 16 位元進位,則 CY(進位旗標)會成為 1,反之若無則為 0。

兩數相加後 R(IN2)為零,則 Z(零值旗標)為 1,反之為 0。

舉例:

ADD R0 R1 R2　//運算行為: R0 = R1 + R2

Opcode : 0001_000_001_010_000

若 R1= 4'hffff, R2=4'h0001,兩數相加後 R0 為 0,Z = 1,CY = 1。

若 R1= 4'hfffe, R2=4'h0001,則 R0=4'hffff,Z = 0,CY = 0。

SUB 指令組合語言詳述:

Instruction Set:　SUB R(IN2) R(IN1) R(IN0)

Operation:　R(IN2) = R(IN1) - R(IN0)

整個系統上並未加入有號數的設計,所以在設計上目前並未加入溢位旗標,也因為沒有加入有號數的設計,所以兩數相減將得到的值將以絕對值表示,若是負值發生,則 SF(符號旗標)為 1。同樣兩數相減後 R(IN2)為零,則 Z(零值旗標)為 1,反之為 0。

舉例:

SUB R2 R1 R0　//運算行為: R2 = R1 - R0

Opcode : 0001_010_001_000_001

若 R1= 4'h0005, R0=4'h0002,則 R2=4'h0003,Z = 0,SF = 0。

若 R1= 4'h0002, R0=4'h0005，則 R2=4'h0003，Z = 0，SF = 1。

MUX 指令組合語言詳述:

Instruction Set:　MUXA(or MUXB) R(IN2) R(IN1) R(IN0)

Operation:　{R(MUXA), R(MUXB)} = R(IN1) * R(IN0)

乘數與被乘數各16位元進行相乘所得到的數值需要32位元的寬度，如果完整進行計算後，要得到完整的數值，需要分開來兩個暫存器進行儲存，為了減少硬體上的設計難度以及不影響指令格式的情況下，將 MUX 指令分為 MUXA 以及 MUXB，較高位元的部分由 MUXA 負責儲存，低位元的部分由 MUXB 進行儲存。

舉例:

MUXA R7 R6 R5　//運算行為: {R7, 4'hxxxx} = R6 * R5

Opcode : 0001_111_110_101_010

若 R6= 4'h8000, R5=4'h8000，則 R7=4'h4000，Z = 0。

MUXB R7 R6 R5　//運算行為: {4'hxxxx ,R7} = R6 * R5

Opcode : 0001_111_110_101_011

若 R6= 4'h8000, R5=4'h8000，則 R7=4'h0000，Z = 1。

BTD 指令組合語言詳述:

Instruction Set:　BTD R(IN2) R(IN1)

Operation:　R(IN2)= 二進制轉十進制(R(IN1))

二進制數值轉十進制數值，包括進位旗標的數值表示範圍為 0~19999。

舉例:

BTD R6 R5　//運算行為: R6 = Binery_To_Decimal(R5)

Opcode : 0001_110_101_000_100

若 R6= 4'h4369, R5=4'h1111。

INC 指令組合語言詳述:

Instruction Set: INC R(IN2)

Operation: R(IN2)= R(IN2) +1

原數值+1，此功能為了方便寫組合語言使用，也可以較容易避免 Data Hazard 的發生。

舉例:

INC R1 //運算行為: R1 <= R1 + 1

Opcode : 0001_001_001_000_101

若 R1= 4'hffff，下一個 CLOCK，R1= 4'h0000，Z = 1，CY = 1。

若 R1= 4'h0001，下一個 CLOCK，R1= 4'h0002，Z = 0，CY = 0。

DEC 指令組合語言詳述:

Instruction Set: INC R(IN2)

Operation: R(IN2)= R(IN2) -1

原數值-1，此功能為了方便寫組合語言使用，也可以較容易避免 Data Hazard 的發生，因為了避免 R1 為 0 時，若以絕對值得方式進行計算，如果不斷進行 DEC 指令時，會造成 R1 的值在 0 與 1 之間互換，所以在設計 DEC 指令時還是使用補數的方式。

舉例:

INC R1 //運算行為: R1 <= R1 + 1

Opcode : 0001_001_001_000_110

若 R1= 4'hffff，下一個 CLOCK，R1= 4'hfffe，Z = 0。

若 R1= 4'h0001，下一個 CLOCK，R1= 4'h0000，Z = 1。

若 R1= 4'h0000，下一個 CLOCK，R1= 4'hffff，Z = 0。

CLR 指令組合語言詳述:

Instruction Set: CLR R(IN2)

Operation: R(IN2)= 0

某數值清零，此功能為了方便寫組合語言使用，也可以較容易避免 Data Hazard 的發生，同時旗標 CY、V、SF、Z 同時清零

舉例:

INC R2 //運算行為: R2 <= 4'h0000

Opcode : 0001_010_010_000_111

若 R1= 4'hf56a，下一個 CLOCK，R1= 4'h0000，CY = 0，SF = 0，Z = 0。

2.10 LOG 指令設計:

LOG 用在各種邏輯運算使用，其中還加入大於、等於、小於的數值比較指令，只會更改旗標數值。而邏輯指令中並沒有加入更改旗標數值設計。指令格式設計如下圖 2-10-1:

4-bits	3-bits	3-bits	3-bits	3-bits
OPCODE	目標暫存器(IN2)	來源暫存器 1(IN1)	來源暫存器 0(IN0)	MODE

圖 2-10-1

為了增加指令個數，所以將來源暫存器與目標暫存器使用的位元數減一個位元，意味著在原本設計使用的 16 個暫存器 R0~R15 只用了其中的 R0~R7。所以在使用 LOG 的邏輯運算要注意的地方是運算只會使用到 R0~R7，若需要 R8~R15 的話，便需要使用 MOVE 搬移指令，將數值從 R8~R15 搬移至 R0~R7 中使用或者將數值從 R0~R7 搬移至 R8~R15 中使用。

MODE 是為了擴充指令數使用。當 MODE = 3'b111 的時候，進行數值比較，不需要目標暫存器，因為存放目標為旗標，所以將目標暫存器使用的三個位元作為擴充比較指令的個數。其中有 EQ(Equal Than):相等於、NE(Not Equal):不等於、GT(Greater Than):大於、LE(Less or Equal):小於等於、LT(Less Than):小於、GE(Greater or Equal):大於等於。指令設計如下圖 2-10-2:

OPCODE(4-bits)	IN2(3-bits)	MODE(3-bits)	INSTRUCTION(指令)
0010	XXX(任意)	000	NOT
0010	XXX(任意)	001	AND
0010	XXX(任意)	010	NAND
0010	XXX(任意)	011	OR
0010	XXX(任意)	100	NOR
0010	XXX(任意)	101	XOR

0010	XXX(任意)	110	NXOR
0010	000	111	EQ
0010	001	111	NE
0010	010	111	GT
0010	011	111	LE
0010	100	111	LT
0010	101	111	GE

圖 2-10-2

NOT 指令組合語言詳述:

Instruction Set: NOT R(IN2) R(IN1)

Operation: R(IN2) = ~R(IN1)

將 R(IN1) 的數值通過 NOT 閘放置至 R(IN2)。

AND 指令組合語言詳述:

Instruction Set: AND R(IN2) R(IN1) R(IN0)

Operation: R(IN2) = R(IN1) & R(IN0)

將 R(IN1) 與 R(IN0)的數值通過 AND 閘放置至 R(IN2)。

NAND 指令組合語言詳述:

Instruction Set: NAND R(IN2) R(IN1) R(IN0)

Operation: R(IN2) = ~(R(IN1) & R(IN0))

將 R(IN1) 與 R(IN0)的數值通過 NAND 閘放置至 R(IN2)。

OR 指令組合語言詳述:

Instruction Set: OR R(IN2) R(IN1) R(IN0)

Operation: R(IN2) = R(IN1) | R(IN0)

將 R(IN1) 與 R(IN0)的數值通過 OR 閘放置至 R(IN2)。

NOR 指令組合語言詳述:

Instruction Set:　NOR R(IN2) R(IN1) R(IN0)

Operation:　R(IN2) = ~(R(IN1) | R(IN0))

將 R(IN1) 與 R(IN0)的數值通過 NOR 閘放置至 R(IN2)。

XOR 指令組合語言詳述:

Instruction Set:　XOR R(IN2) R(IN1) R(IN0)

Operation:　R(IN2) = R(IN1) ^ R(IN0)

將 R(IN1) 與 R(IN0)的數值通過 XOR 閘放置至 R(IN2)。

NXOR 指令組合語言詳述:

Instruction Set:　NXOR R(IN2) R(IN1) R(IN0)

Operation:　R(IN2) = R(IN1) ~^ R(IN0)

將 R(IN1) 與 R(IN0)的數值通過 NXOR 閘放置至 R(IN2)。

EQ 指令組合語言詳述:

Instruction Set:　EQ R(IN1) R(IN0)

Operation:　if(R(IN1) == R(IN0))

將 R(IN1) 與 R(IN0)的數值相互比較，如果兩數相等，則相等判斷旗標(EQ)為 1，不相等則(EQ)為 0。

NE 指令組合語言詳述:

Instruction Set:　NEQ R(IN1) R(IN0)

Operation:　if(R(IN1) != R(IN0))

將 R(IN1) 與 R(IN0)的數值相互比較，如果兩數不相等，則相等判斷旗標(EQ)為 0，不相等則相等判斷旗標(EQ)為 1。

GT 指令組合語言詳述:

Instruction Set:　GT R(IN1) R(IN0)

Operation:　if(R(IN1) > R(IN0))

將 R(IN1) 與 R(IN0)的數值相互比較，若 R(IN1)大於 R(IN0)，則大於判斷旗標(BT)為 1；若 R(IN1)不大於 R(IN0)，則大於判斷旗標(BT)為 0。

LE 指令組合語言詳述:

Instruction Set: LE R(IN1) R(IN0)

Operation: if(R(IN1) <= R(IN0))

將 R(IN1) 與 R(IN0)的數值相互比較，若 R(IN1)小於等於 R(IN0)，則小於判斷旗標(ST)以及相等判斷旗標(EQ)皆為 1；若 R(IN1)不小於等於 R(IN0)，則小於判斷旗標(ST)以及相等判斷旗標(EQ)皆為 0。

LT 指令組合語言詳述:

Instruction Set: LT R(IN1) R(IN0)

Operation: if(R(IN1) < R(IN0))

將 R(IN1) 與 R(IN0)的數值相互比較，若 R(IN1)小於 R(IN0)，則小於判斷旗標(ST)為 1；若 R(IN1)不小於 R(IN0)，則小於判斷旗標(ST)為 0。

GE 指令組合語言詳述:

Instruction Set: GE R(IN1) R(IN0)

Operation: if(R(IN1) <= R(IN0))

將 R(IN1) 與 R(IN0)的數值相互比較，若 R(IN1)大於等於 R(IN0)，則大於判斷旗標(BT)以及相等判斷旗標(EQ)皆為 1；若 R(IN1)不大於等於 R(IN0)，則大於判斷旗標(BT)以及相等判斷旗標(EQ)皆為 0。

2.11 SHF 指令設計:

SHF 用在各種位移位元使用,共設計了五種位移方式,又分為左移以及右移,所以總共有十種位移方式,需要 4-bits 的指令空間。其中,因為暫存器設計為 16 位元,故若要一次完成任意位元都能達到任意指定位置,則需要移動 15 次,故需要使用 4-bits 的指令格式。包含 OPCODE 共有 12-bits,因此使用的暫存器可以增至 16 個。指令格式設計如下圖 2-11-1:

4-bits	4-bits	4-bits	4-bits
OPCODE	目標以及來源暫存器(IN2&IN1)	移位位元數	MODE

圖 2-11-1

指令設計如下圖 2-11-2:

OPCODE(4-bits)	移位位元數(4-bits)	MODE(4-bits)	INSTRUCTION(指令)
0011	XXXX(任意)	0000	SHL(左移)
0011	XXXX(任意)	0001	SHR(右移)
0011	XXXX(任意)	0010	SCL(含進位左移)
0011	XXXX(任意)	0011	SCR(含進位右移)
0011	Don't care	0100	SAL(算術左移)
0011	Don't care	0101	SAR(算術右移)
0011	XXXX(任意)	0110	ROL(循環左移)
0011	XXXX(任意)	0111	ROR(循環右移)
0011	XXXX(任意)	1000	RCL(含進位循環左移)
0011	XXXX(任意)	1001	RCR(含進位循環右移)

圖 2-11-2

SHL 指令組合語言詳述:

Instruction Set: SHL R(IN1 & IN2) SHF_TIMES(移位次數)

Operation: R(IN1 & IN2) = R(IN1 & IN2) << SHF_TIMES

將暫存器內的值向左移動 SHF_TIMES(Shift Times)次，空白位元補零值。

SHR 指令組合語言詳述:

Instruction Set: SHR R(IN1 & IN2) SHF_TIMES(移位次數)

Operation: R(IN1 & IN2) = R(IN1 & IN2) >> SHF_TIMES

將暫存器內的值向右移動 SHF_TIMES(Shift Times)次，空白位元補零值。

SCL 指令組合語言詳述:

Instruction Set: SCL R(IN1 & IN2) SHF_TIMES(移位次數)

Operation: R(IN1 & IN2) = R(IN1 & IN2) << SHF_TIMES

將暫存器內的值包含進位旗標向左移動 SHF_TIMES(Shift Times)次，空白位元補零值。

SCR 指令組合語言詳述:

Instruction Set: SCR R(IN1 & IN2) SHF_TIMES(移位次數)

Operation: R(IN1 & IN2) = R(IN1 & IN2) << SHF_TIMES

將暫存器內的值包含進位旗標向右移動 SHF_TIMES(Shift Times)次，空白位元補零值。

SAL 指令組合語言詳述:

Instruction Set: SAL R(IN1 & IN2)

Operation: {SF,R(IN1 & IN2)} = {SF,R(IN1 & IN2)} <<< 1

將暫存器內的值包含負值旗標進行算術左移，僅一次算術左移。

SAR 指令組合語言詳述:

Instruction Set: SAR R(IN1 & IN2)

Operation: {SF,R(IN1 & IN2)} = {SF,R(IN1 & IN2)} >>> 1

將暫存器內的值包含負值旗標進行算術右移，僅一次算術右移。

ROL 指令組合語言詳述:

Instruction Set: ROL R(IN1 & IN2)

Operation: R(IN1 & IN2)內的值進行循環左移

將暫存器內的值向左循環移動 SHF_TIMES(Shift Times)次。

ROR 指令組合語言詳述:

Instruction Set: ROR R(IN1 & IN2)

Operation: R(IN1 & IN2)內的值進行循環右移

將暫存器內的值向右循環移動 SHF_TIMES(Shift Times)次。

RCL 指令組合語言詳述:

Instruction Set: RCL R(IN1 & IN2)

Operation: {CY,R(IN1 & IN2)}進行循環左移

將暫存器內的值包含進位旗標向左循環移動 SHF_TIMES(Shift Times)次。

RCR 指令組合語言詳述:

Instruction Set: RCR R(IN1 & IN2)

Operation: {CY,R(IN1 & IN2)}進行循環右移

將暫存器內的值包含進位旗標向右循環移動 SHF_TIMES(Shift Times)次。

2.12 SETFLAG 指令設計:

SETFLAG 用在各種設定各種旗標數值使用，目前一共設計了 18 種更改旗標指令，使用的指令格數僅占 5-bits，加上 OPCODE 指令格數，僅 9-bits，故未填滿 16 位元，另外此處可供未來指令擴充使用。指令格式設計如下圖 2-12-1:

4-bits	7-bits	5-bits
OPCODE	Don't care	MODE

圖 2-12-1

指令設計如下圖 2-12-2:

圖 2-12-2

OPCODE(4-bits)	MODE(5-bits)	INSTRUCTION(指令)	Operation
0100	00000	SETCY	CY = 1
0100	00001	CLRCY	CY = 0
0100	00010	SETSF	SF =1
0100	00011	CLRSF	SF =0
0100	00100	SETEQ	EQ =1
0100	00101	CLREQ	EQ =0
0100	00110	SETBT	BT =1
0100	00111	CLRBT	BT =0
0100	01000	SETST	ST =1
0100	01001	CLRST	ST =0
0100	01010	SETIEN	IEN =1
0100	01011	CLRIEN	IEN =0
0100	01100	SETIT1	IT1 =1

0100	01101	CLRIT1	IT1 =0
0100	01110	SETIT0	IT0 =1
0100	01111	CLRIT0	IT0 =0
0100	10000	SETITR	ITR =1
0100	10001	CLRITR	ITR =0

2.13 LOAD 指令設計:

LOAD 用在各種設定存放數值進入記憶體的資料儲存區中，在此平台上，可以說是將數值寫入 RAM 記憶體中，或從 RAM 記憶體中讀出需要的資料。指令格式設計如下圖 2-13-1:

4-bits	4-bits	4-bits	4-bits
OPCODE	目標或來源暫存器(IN2)	RAM 儲存區尋址位置(IN1)	MODE

圖 2-13-1

指令設計如下圖 2-13-2:

OPCODE	MODE	INSTRUCTION	Operation
0101	0000	LOAD	R(IN2) = MEM(R(IN1))
0101	1111	STORE	MEM(R(IN1)) = R(IN2)

圖 2-13-1

2.14 SET 指令設計:

SET 指令是為了能夠將 16 位元的數值放入暫存器中，然而 OPCODE 及目標暫存器的指令空間，需要耗費 8-bits，因此無法完全輸入 16 個位元的數值，所以此指令需要占用兩個 Address 的儲存空間，設計方式有兩種:一種是使用剩餘的 8 個位元,要完全儲存一筆資料需要花費兩個 SET 指令;另一種是分成兩條指令，第一條為 SET 的 OPCODE 指令以及目標暫存器，然後第二條，完全記錄 16 位元的數值。 考慮到硬體的設計，為了避免在 PIPELINE 上遭遇資料障礙，影響整體的效能，因此決定使用第二種的設計方式。其指令格式設計如下圖 2-14-1:

4-bits	4-bits	8-bits
OPCODE	目標暫存器(IN2)	Don't care
16-bits 數值		

圖 2-14-1

指令設計如下圖 2-14-2:

OPCODE	INSTRUCTION	Operation
0110	SET	R(IN2) = 16-bits value

圖2-14-2

2.15 MOVE指令設計:

MOVE 用以搬移各種暫存器數值使用，同時因設置了 Interrupt Timer(內部中斷時鐘)，所以 MOVE 也用來放置 Timer Counter(以下將簡寫為 TC)的數值以及 Timer Address(以下將簡寫為 TA)的記憶體絕對位置。指令格式設計如圖 2-15-1:

4-bits	4-bits	4-bits	4-bits
OPCODE	目標或來源暫存器(IN2)	目標或來源暫存器(IN1)	MODE

圖 2-15-1

指令設計如下圖:

組合語言:MOVE(圖 2-15-2)

OPCODE	IN2	IN1	MODE	Operation
0111	R(IN2)	R(IN1)	0000	R(IN2) = R(IN1)

圖 2-15-2

組合語言:MOVEIT0(圖 2-15-3)

OPCODE	IN2	IN1	MODE	Operation
0111	R(IN2)	R(IN1)	0001	TC0<= R(IN2)，TA0<=R(IN1)

圖 2-15-3

組合語言:MOVEIT1(圖 2-15-4)

OPCODE	IN2	IN1	MODE	Operation
0111	R(IN2)	R(IN1)	0010	TC1<= R(IN2)，TA1<=R(IN1)

圖 2-15-4

組合語言:MOVEITR(圖 2-15-5)

OPCODE	IN2	IN1	MODE	Operation
0111	Don't care	R(IN1)	0011	TA_ITR<=R(IN1)

圖 2-15-5

組合語言:MOVEFS(圖 2-15-6)

OPCODE	IN2	IN1	MODE	Operation
0111	Don't care	Don't care	0100	TEMP_FLAG <={CY,SF,EQ,BT,ST}

圖 2-15-6

組合語言:MOVEFL(圖 2-15-7)

OPCODE	IN2	IN1	MODE	Operation
0111	Don't care	Don't care	0101	{CY,SF,EQ,BT,ST}<= TEMP_FLAG

圖2-15-7

2.16 PUSH指令設計:

PUSH 指令用以快速儲存資料使用，放置在暫時的堆疊暫存器中，PUSH 為堆疊資料、POP 為取出資料，其資料讀取有著後進先出的特性，其儲存空間可以儲存 128 筆 16 位元的資料。指令格式設計如下圖 2-16-1:

4-bits	4-bits	8-bits
OPCODE	目標或來源暫存器(IN2)	MODE

圖 2-16-1

指令設計如下圖 2-16-2:

OPCODE	MODE	INSTRUCTION	Operation
1000	00000000	PUSH	STACK(COUNT)=R(IN2), COUNT<= COUNT+1
1000	11111111	POP	R(IN2)=STACK(COUNT-1),COUNT<= COUNT-1

圖2-16-2

2.17 JUMP指令設計:

JUMP 指令用以判斷是否要跳躍至某指令位置，其指令跳躍方式使用相對位置跳躍，以 JUMP 指令為中心點向上或向下跳躍多少指令位置個數，可跳躍 -255~255 之間的位置個數。若判斷式不成立則無跳躍發生。指令格式設計如下圖 2-17-1:

4-bits	3-bits	1-bits	8-bits
OPCODE	來源或旗標暫存器(IN2 or MODE)	向上(1)或向下(0)	相對位置跳躍

圖 2-17-1

指令設計如下圖 2-17-2:

圖 2-17-2

OPCODE	IN2 or MODE	INSTRUCTION	Operation
1001	R0~R7	JZ	If(R(IN2) == 0)JUMP
1010	R0~R7	JNZ	If(R(IN2) != 0)JUMP
1011	000	JMP	JUMP
1011	001	JCY	If(CY == 1)JUMP
1011	010	JNCY	If(CY == 0)JUMP
1011	011	JSF	If(SF == 1)JUMP
1011	100	JNSF	If(SF == 0)JUMP
1011	101	JEQ	If(EQ == 1)JUMP
1011	110	JNEQ	If(EQ == 0)JUMP
1011	111	JBT	If(BT == 1)JUMP
1100	000	JNBT	If(BT == 0)JUMP
1100	001	JST	If(ST == 1)JUMP

1100	010	JNST	If(ST == 0)JUMP
1100	011	JIT1	If(IT1 == 1)JUMP
1100	100	JNIT1	If(IT1 == 0)JUMP
1100	101	JIT0	If(IT0 == 1)JUMP
1100	110	JNIT0	If(IT0 == 0)JUMP
1100	111	JIEN	If(IEN == 1)JUMP

2.18 IN、OUT指令設計:

IN、OUT 指令用以溝通 Bridge 的外部裝置使用，可以取得或發送需要的資料給 Bridge 操控外部裝置，考量到裝置的多樣性以及為了能夠傳遞足夠的資訊量，因此 OUT 指令所需要的數值以及操控碼，皆來自暫存器中，而 IN 指令僅僅為單方面接收，需要由 OUT 指令先呼叫該輸入裝置，其裝置才會將其所需要的數值放入暫存器中。

IN 指令格式設計如下圖 2-18-1:

4-bits	4-bits	4-bits	4-bits
OPCODE	Don't care	目標暫存器(IN2)	MODE
1101	Don't care	R0~R15	0000

圖 2-18-1

OUT 指令格式設計如下圖 2-18-2:

4-bits	4-bits	4-bits	4-bits
OPCODE	裝置種類	裝置數值	MODE
1101	來源暫存器(IN1)	來源暫存器(IN0)	1111

圖2-18-2

(其詳細內容將放置在硬體設計規劃上，以及其設計概念，在此不便作詳述。)

2.19 CALL指令設計:

CALL 指令用以進行遠距離的指令位置跳躍，其指令跳躍方式使用絕對位置跳躍，為了能夠跳躍至 RAM 記憶體中的任一個位置。為考量相容於絕大多數 FPGA 平台環境，RAM 記憶體區塊大小緣故，設計為 12 位元絕對位置跳躍，

同時跳躍至副程式後，需要進行返回動作，則交由 RET 指令，跳回至原本 CALL 指令所在位置的下一行，為了能跳回 CALL 指令的位置，另外設計了一個暫存空間，此空間與 PUSH 指令使用的空間相似，可儲存需要返回的位置資訊，有後進先出的特性，此空間可儲存 12 位元的空間位置資訊，可儲存量為 32 行位置資訊。

RET 指令的設計為絕對位置為 12 個 1 時，因跳躍不可能跳躍至最後一行指令，因此將此位置供 RET 指令作為返回使用。

指令格式設計如下圖 2-19-1:

4-bits	12-bits
OPCODE	12bit 位置
1110	12bit 位置
1110	111111111111

圖 2-19-1

2.20 NULL指令設計:

NULL 指令單純為空白指令，只要 OPCODE 為 0 則為 NULL 指令，NULL 指令的存在是為了避免程式錯誤，防止程式誤讀不需要的指令。

NULL 指令格式設計如下圖 2-20-1:

4-bits	12-bits
OPCODE	Don't care
0000	Don't care

圖2-20-1

2.21 HALT指令設計:

HALT 指令為休眠指令，使 CPU 進入休眠，不進行任何運算，若中斷發生則解除休眠，可以從外部中斷(ITR)，以及內部時鐘中斷(IT1、IT0)。

指令格式設計如下圖 2-21-1:

4-bits	12-bits
OPCODE	Don't care
1111	Don't care

圖2-21-1

第三章 設計開發 Microprocessor

3.1 微處理器架構設計:

採用 HKH-II 的十六位元精簡指令集架構，為追求較高效能表現，採用五級流水線架構，如圖 3-1-1。

由 PC(Program Counter)程式計數器，指向 RAM 位置，取得指向位置所儲存的指令後，將其指令放置第一層暫存器中，第二層為控制 PC 值且取得計算所需要的數值，第三層為將取得的數值放入運算元中進行計算。第四層為將結果暫時儲存，第五層則為寫回暫存器群中。

若遇到跳躍指令，則需要將第三層運算所得到的結果回傳至第二層，使第二層判斷是否要跳躍或不跳躍。

若運算時遇到資料障礙，則第四層的結果暫存，會返回值給第三層的運算，使運算不需要受到資料障礙而暫停。

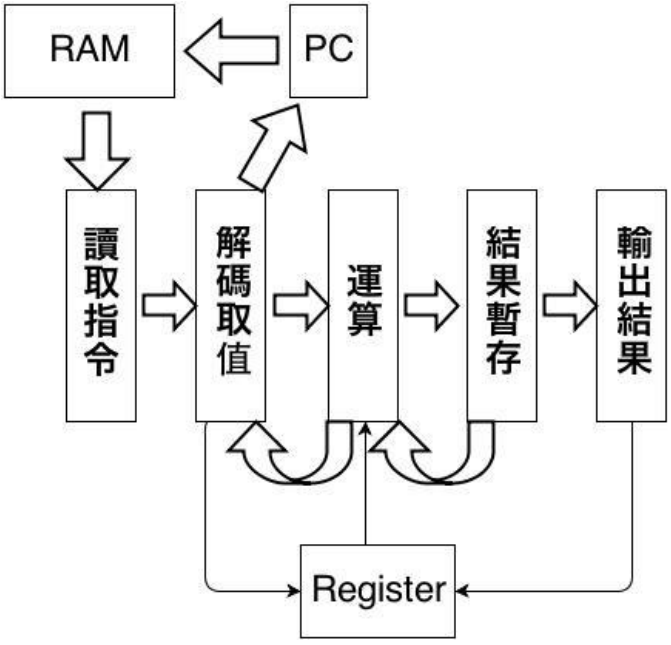

圖3-1-1 CPU內部資料流程

3.2 微處理器 RAM 設計:

為了提供順暢的流水線作業,比如說遇到SET指令、LOAD指令以及STORE指令,需要將資料寫入RAM中或取出資料,為了盡量避免管線中斷或停止,因此採用Ture Dual Port Ram,可在不影響讀取指令的時候進行寫入資料以及讀取資料。

4K-word的可程式記憶體採用由IP core生成的Ture Dual Port Ram,如圖3-2-1。並使用Read First Mode,其時序時脈圖如圖3-2-2。

在負源觸發時指向指令位置,並在下一個負源發生時讀取指令,讀取指令的行為會延遲一個時脈。

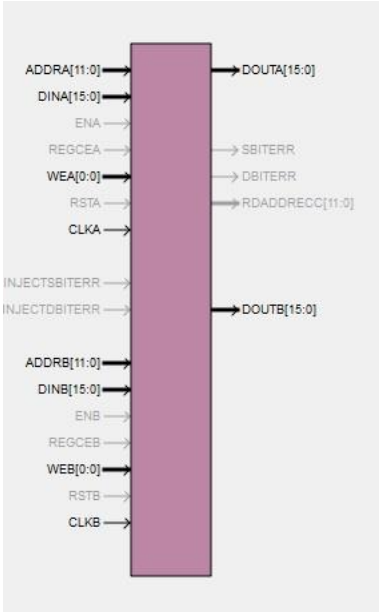

圖3-2-1 Ture Dual Port Ram模塊圖

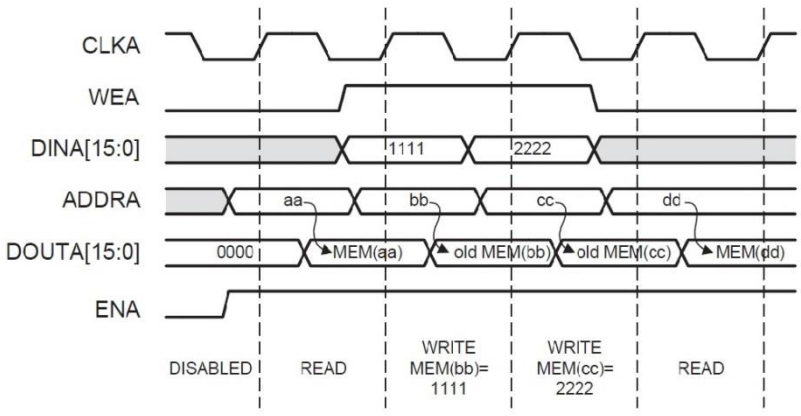

圖3-2-2 RAM時序圖

3.3 微處理器管線設計:

如圖 3-3-1 為微處理器管線設計，編號 1 至 5 皆為管線使用的暫存器，N 為負源觸發、P 為正源觸發，分別分為，讀取指令階段、解碼階段、取值階段、計算階段、結果暫存階段、輸出階段，一共六個階段。

第一階段為讀取指令，因為 RAM 從位置設定到數值輸出需一個 CLOCK，為了取得確切、穩定的數值，需要經過一次緩衝，所以在負源觸發時，擷取 RAM 的數值，然後存入編號 1 號暫存器。

第二階段為解碼，在負源時觸發，因不同指令的指令格式稍許不同，需要透過解碼，判斷指令功能，且決定 PC 計數器的位置指向，並選擇指令所需的 R0~R15 暫存器群(編號 2 號暫存器)。

第三階段為取值階段，在正源觸發時，將第二階段所選擇的暫存器的數值取出並提供給第四階段使用。

第四階段為計算階段，在負源觸發時，將數值提供給計算元使用，或者執行一些特殊指令，比如判斷是否跳躍、對 RAM 的讀取、數值堆疊等等。

第五階段為結果暫存階段，在正源時觸發，將計算所取得的結果數值暫存，提供給下一個指令計算所需，避免資料障礙發生。同時作為輸出緩衝，用來判斷是否將數值寫入暫存器、將數值做輸出或是將數值寫入 RAM 中等等地判斷。

第六階段為結果輸出，在負源時觸發，將輸出寫入 RAM 中、寫入暫存器群中或是對輸出暫存器作控制。

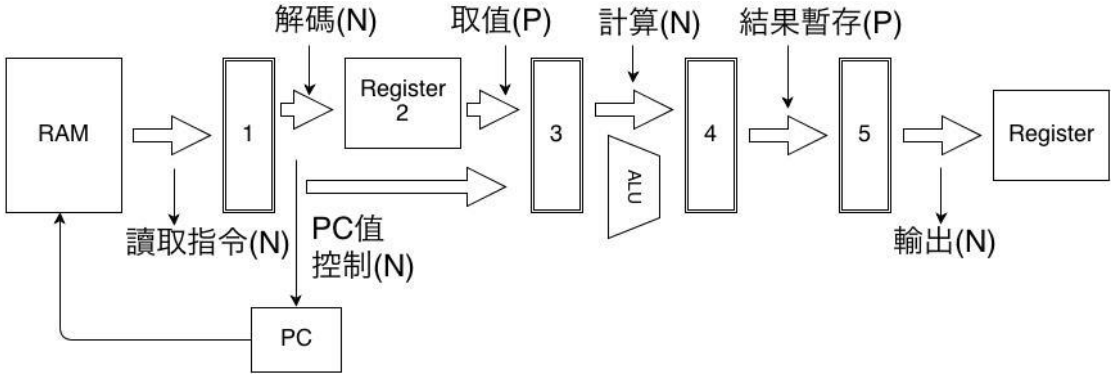

圖3-3-1 微處理器管線設計

3.4 微處理器資料障礙處理:

如圖 3-4-1 ，當連續兩個指令所需的值有相關時，比如:

第一行指令 R1 <= R2 + R3

第二行指令 R4 <= R1 + R5

下一行指令需要上一行所計算出來的數值,若是第二行指令無法正確取得第一行指令時，便會造成資料障礙。

解決資料障礙的方法，在於第一行指令在負源觸發時進行計算，正源觸發時進行數值儲存，當第二行指令排入編號3時，與第一行指令相互比較，如果第二行指令輸出所需暫存器與第一行指令輸出暫存器相同，則將編號4所儲存的結果值提供給編號3的第二行指令使用，如此不用進行暫停動作，便可以繼續進行計算。

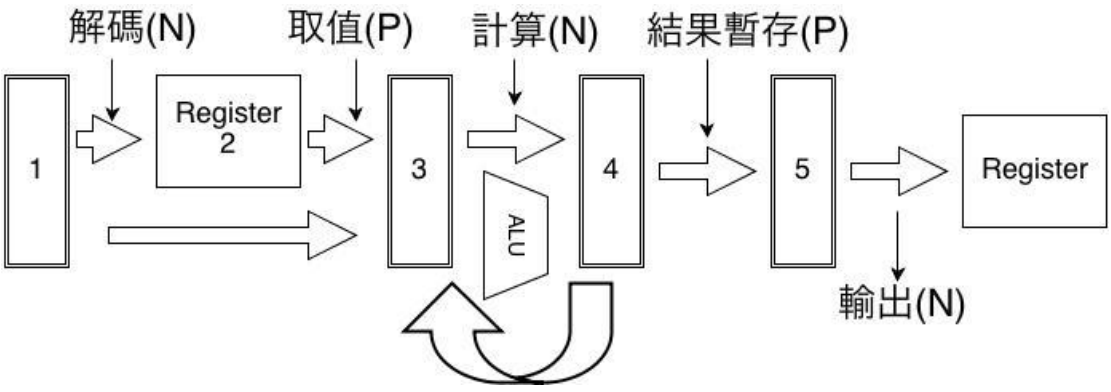

圖3-4-1 微處理器資料障礙處理

3.5 微處理器跳躍障礙處理:

圖3-5-1為跳躍障礙處理圖示。當跳躍指令進入編號1的暫存器中時,PC停止計數，並指向下一行程式位置，同時跳躍指令停留在編號1中，但其指令依然排入管線之中，之後經過兩個CLOCK後，跳躍指令排入編號3暫存器中，開始進行跳躍判斷，判斷是否進行跳躍，向編號1暫存器傳回跳躍判斷旗標。

若是判斷不跳躍則繼續執行下一行指令，若是判斷需要跳躍，則將編號1的暫存器排入空指令，並更改PC值指向跳躍位置。

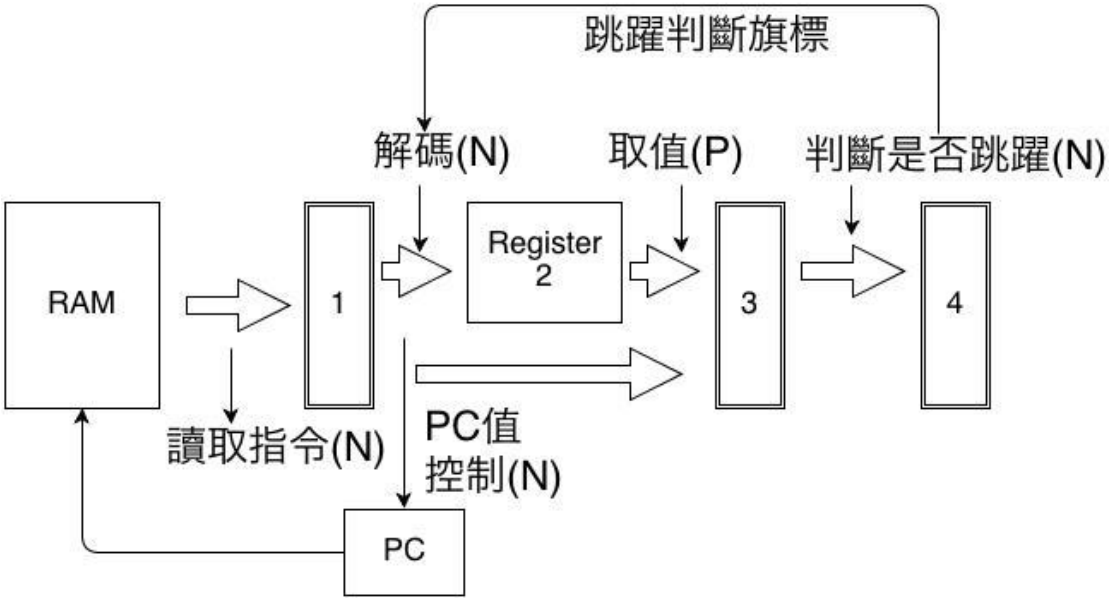

圖3-5-1 微處理器跳躍障礙處理

3.6 微處理器跳躍改良:

圖3-6-1為跳躍分支預測改良圖示，為了減少跳躍失敗所造成的失誤代價，並增加跳躍成功機率，所以採用2位元的分支預測，00以及01為猜測不跳躍，10以及11為猜測跳躍。

若是猜測不跳躍，而且猜測成功，則沿著細線方向行進，同樣的，若猜測跳躍，然而猜測失敗，同樣會沿著細線方向行進。

若是猜測跳躍，而且猜測成功，則沿著粗方向線行進，同樣的，若猜測不跳躍，然而猜測失敗，同樣會沿著粗方向線行進。

在程式中常出現,FOR迴圈或是WHILE迴圈等等，若是能大量增加跳躍成功機率，可增進微處理器效能。

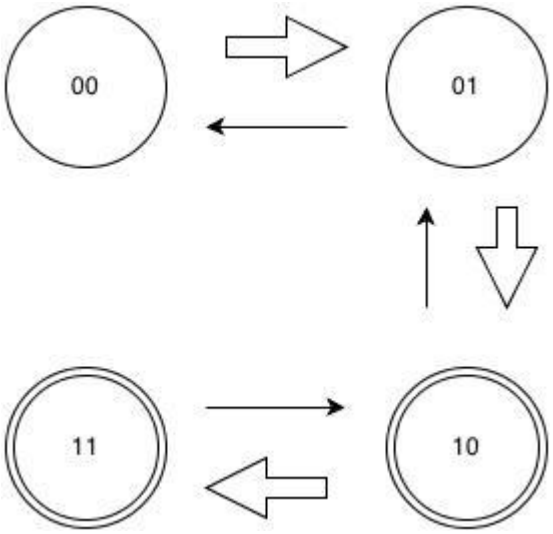

圖3-6-1 跳躍分支預測改良圖示。

3.7 微處理器頂層模組架構

微處理器本身的架構複雜，尤其加入管線設計後，更不易開發設計，為了避免電路撰寫以及除錯上的困難，於是建立頂層模塊，如圖3-7-1。

在設計上，若是將所有級數管線分開成一個一個的模塊，會大量的增加複雜程度，於是在設計頂層模塊的時候，將其分為五個部分：RAM、BLOCK 1、Register、BLOCK 2、OUT。

RAM為指令以及資料儲存的位置，BLOCK 1包含PC控制、第一級以及第二級管線，Register為暫存器群，BLOCK 2包含第三級、第四級、第五級管線，OUT則是處理輸出資料使用。

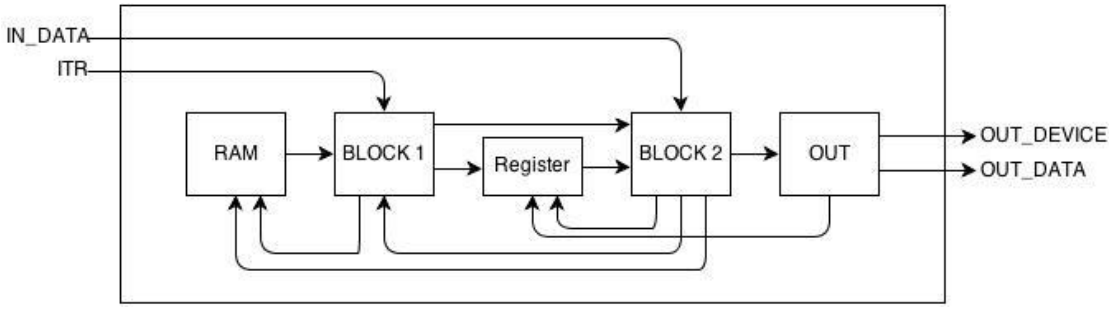

圖 3-7-1

3.8 微處理器 RAM 模塊介紹

此微處理器的RAM是用以儲存資料以及指令使用，並無明顯區分指令以及資料儲存位置，為了盡量減少管線暫停，因此採用DUAL PORT RAM，如圖3-8-1。

指令讀取一律交由RAM的上半部做處理，而下半部則是用以處理存取資料使用，可避免需要存取時，而暫停讀取指令。

上半部為clka、wea、addra、dina、douta。其中clka為時序控制線，其時序與微處理器相同。wea為是否寫入資料，但此部分的功用以讀取指令為主，所以並沒有設計可寫入狀態，因此wea以及dina皆為接地。addra為指令位置控制線，連接PC使用。douta為指令輸出，可將addra位置的值輸出，連接BLOCK 1的opcode_in腳位。

下半部為clkb、web、addrb、dinb、doutb。clkb時序與微處理器相同。Web為控制是否寫入，由BLOCK 2做控制，受STORE指令控制，在計算階段將值存入。addrb為資料位置控制線，同樣由BLOCK 2做控制，受STORE、LOAD、SET指令控制。doutb為資料輸出，將指令所需位置值讀出，在輸出階段直接寫入暫存器中。為避免資料障礙發生，因此LOAD、SET指令將使管線暫停一個時脈週期。

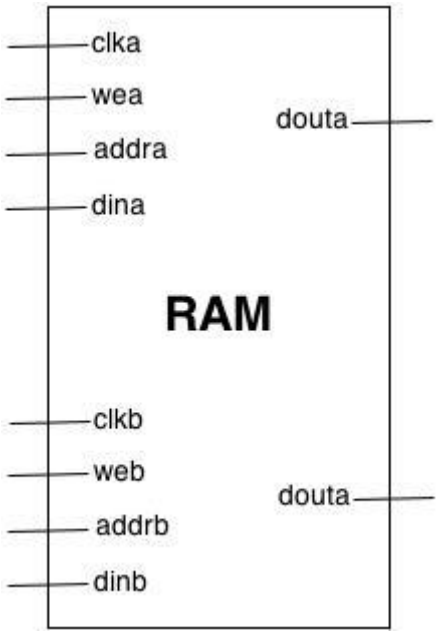

圖 3-8-1

3.9 微處理器 BLOCK 1 模塊介紹

如圖3-9-1為BLOCK 1模塊，BLOCK 1為第一級以及第二級管線所組成，此部分包含讀取指令、PC值控制、中斷控制、並將指令排入下一級管線。

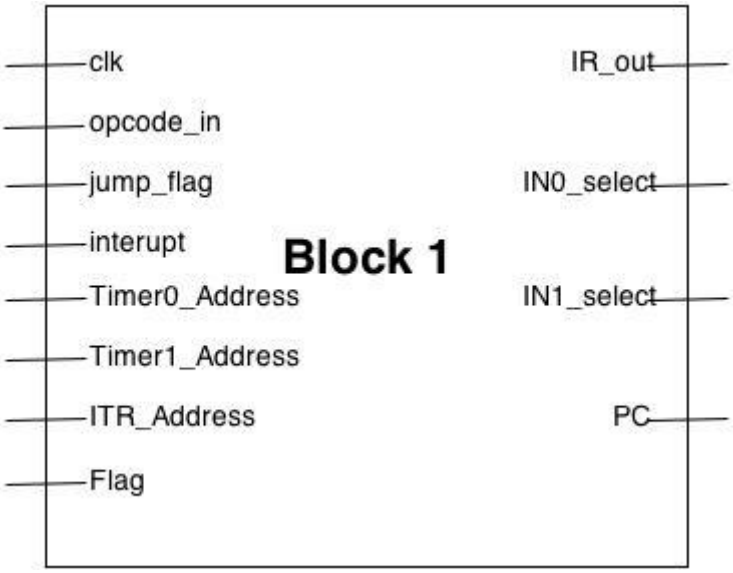

圖 3-9-1

opcode_in腳位連接douta，負責接收指令，為輸入端。

jump_flag腳位連接BLOCK 2的jump_flag_out，用以判斷是否跳躍，

Interrupt為休眠喚醒使用，HALT指令為休眠指令，進入眠後由外部中斷或內部計時中斷進行喚醒，喚醒後執行下一行指令。

Timer0_Address、Timer1_Address皆為中斷跳躍位置輸入，當內部計時中斷發生時，將跳躍至中斷跳躍所設定的位置。

ITR_Address為中斷跳躍位置輸入，當外部中斷發生，則跳躍至外部中斷跳躍所設定的位置。

Flag，旗標值輸入，用以判斷是否中斷。

IR_out，將指令排入下一級管線。

IN0_select、IN1_select為暫存器選擇，用於取值使用。

PC用以指向指令位置。

3.10 微處理器 Register 模塊介紹

如圖3-10-1為Register模塊，其中包含暫存器群，其功能在於可指定兩個暫存器值輸出，以及將值寫入暫存器的功能。

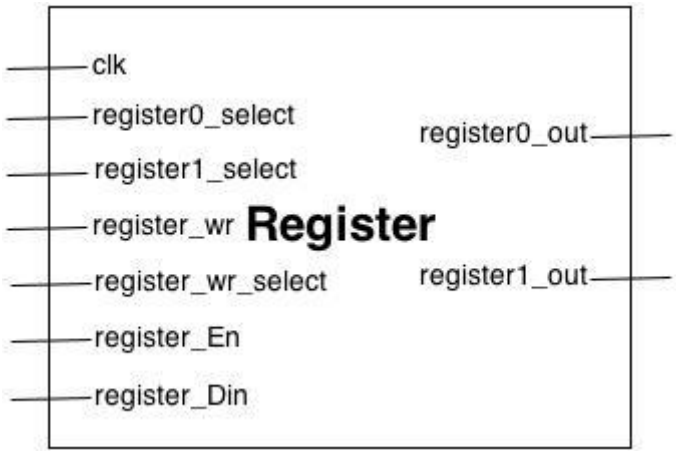

圖 3-10-1

Register0_select為暫存器選擇控制，選擇後將值輸出至Register0_out。

Register1_select為暫存器選擇控制，選擇後將值輸出至Register1_out。

Register_wr、Register_En皆為暫存器寫入控制線。

Register_Din為暫存器寫入資料線

Register_wr_select為暫存器寫入選擇控制，若為可寫入狀態時，則將Register_Din值寫入所選擇的暫存器中。

3.11 微處理器 BLOCK 2 模塊介紹

如圖3-11-1為BLOCK 2模塊，其中包含第三級、第四級、第五級管線，主要用於計算、處理跳躍、存取數值、旗標數值管理、結果暫存。

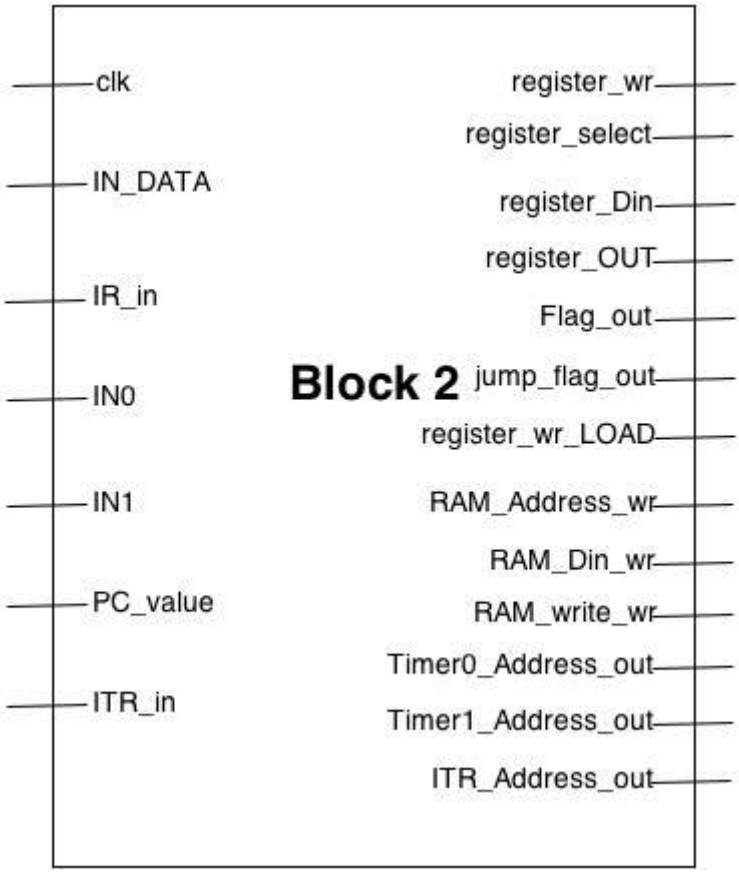

圖 3-11-1

IN_DATA為外部資料輸入線,在此微處理器的設計上,若需讀取外部資料時，需先向外部裝置發出請求訊息，然後將值傳入IN_DATA中，因此需主動接收資料，由IN指令進行控制。

IR_in為接收指令使用，此輸入端連接BLOCK 1的IR_out輸出端。

IN0、IN1為暫存器值輸入，通過IN0_select、IN1_select選擇暫存器後，連接Register0_out、Register1_out輸出端，接收暫存器數值。

PC_value用於取得指令PC值，連接PC，用於資料讀取使用。

ITR_in為外部中斷輸入，在中斷設計上，需由旗標值判定是否需要中斷，外

部中斷發生時，將由ITR_in控制線輸入訊號，並更改ITR旗標值。

Register_wr為暫存器寫入控制旗標，將計算後的數值放置於暫存器中。與Register_wr_LOAD暫存器寫入控制旗標不同，Register_wr_LOAD是將RAM中值讀出並放置暫存器中，兩者控制旗標皆為打開狀態，則為OUT指令的輸出控制，將IN0、IN1數值分別排入Register_Din以及Register_OUT中，並作為OUT_DATA以及OUT_DEVICE微處理器的輸出使用。

Register_select為暫存器寫入選擇控制。

Register_Din為暫存器資料寫入，在OUT指令則為OUT_DATA資料輸出使用

Register_OUT作為輸出裝置控制使用，在OUT指令為OUT_DEVICE裝置輸出。

Flag_out將旗標值輸出，提供給BLOCK 1作為中斷控制。

Jump_flag_out用以判斷是否跳躍，將結果傳回至BLOCK 1。

Register_wr_LOAD為暫存器寫入控制旗標。

RAM_Address_wr用於指向需要讀取或存入RAM的位置。

RAM_Din_wr用於將數值寫入RAM中。

RAM_write_wr用於控制是否將數值寫入RAM中。

Timer0_Address、Timer1_Address此為內部中斷計時發生時，所需跳躍的內部計時中斷程式位置，在設定內部計時中斷時，需先設定內部計時中斷跳躍位置。

ITR_Address_out此為外部中斷發生時，所需跳躍的外部中斷程式位置，置若需要進行外部中斷，則需要先設定外部計時中斷跳躍位置。

3.12 微處理器 OUT 模塊介紹

如圖3-12-1為OUT模塊，包含為第六級管線，其功能主要為輸出。

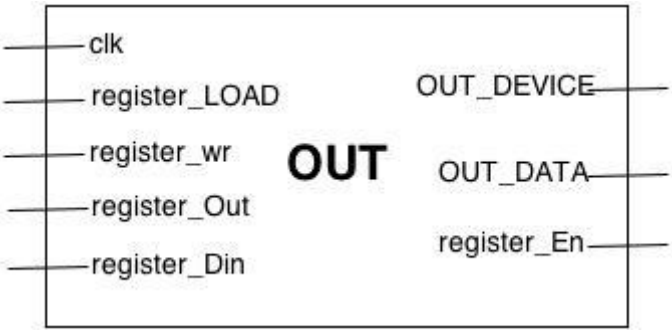

圖3-12-1

Register_LOAD、Register_wr兩者為控制暫存器寫入以及輸出使用，若兩者皆為關時，不進行暫存器寫入以及數值輸出。若Register_wr為開，另一個為關，則將計算數值寫入暫存器中，此時register_En為開。若Register_LOAD為開，另一個為關，則將RAM的douta輸出數值寫入暫存器中，此時register_En為開。若Register_LOAD、Register_wr為開啟狀態，此時register_En為關，並將Register_Din、Register_OUT數值分別寫入OUT_DATA以及OUT_DEVICE暫存器中，作為輸出緩衝，並維持一個時脈週期，之後清空暫存器數值，此動作為避免重複對外部裝置進行控制，並可減少交握方式的設計複雜度。

Register_Din為暫存器資料寫入，在OUT指令則為OUT_DATA資料輸出使用

Register_OUT作為輸出裝置控制使用，在OUT指令為OUT_DEVICE裝置輸出。

OUT_DEVICE為輸出控制，用於指定所需要控制的外部裝置。

OUT_DATA為資料輸出，用於輸出數值提供外部裝置使用。

register_En為暫存器控制輸入開關。

第四章 設計開發 Assembler

4.1 Assembler 開發設計前言

Assembler 在於開發微處理器程式是不可缺少的，許多微處理器的使用比如說 8051，大多數使用者皆是從組合語言開始學起，利用組合語言建構起一個一個的程式，然而微處理器並無法處理組合語言，需要將組合語言轉換成為機械語言，如此微處理器才得以運行。

4.2 Assembler 開發設計考量

這套 Assembler 是提供給 FPGA 平台上的 RAM 使用，因此需要轉換成 COE 檔，並滿足 COE 檔所需要的格式，範例如下:

```
memory_initialization_radix=2;
memory_initialization_vector=
1110000001010000,
0110110000000000;
```

memory_initialization_radix 即為設定資料為幾進制。

memory_initialization_vector 則為資料數值輸入，按照順序從第一行到最後一行，若對應到 RAM 儲存空間內，如圖 4-2-1

Address	Value
0x000	1110000001010000
0x001	0110110000000000

圖 4-2-1

4.3 COE 檔匯入 RAM 中

1. 點擊兩次 RAM 的 IP 核，如圖 4-3-1 藍色選取處。

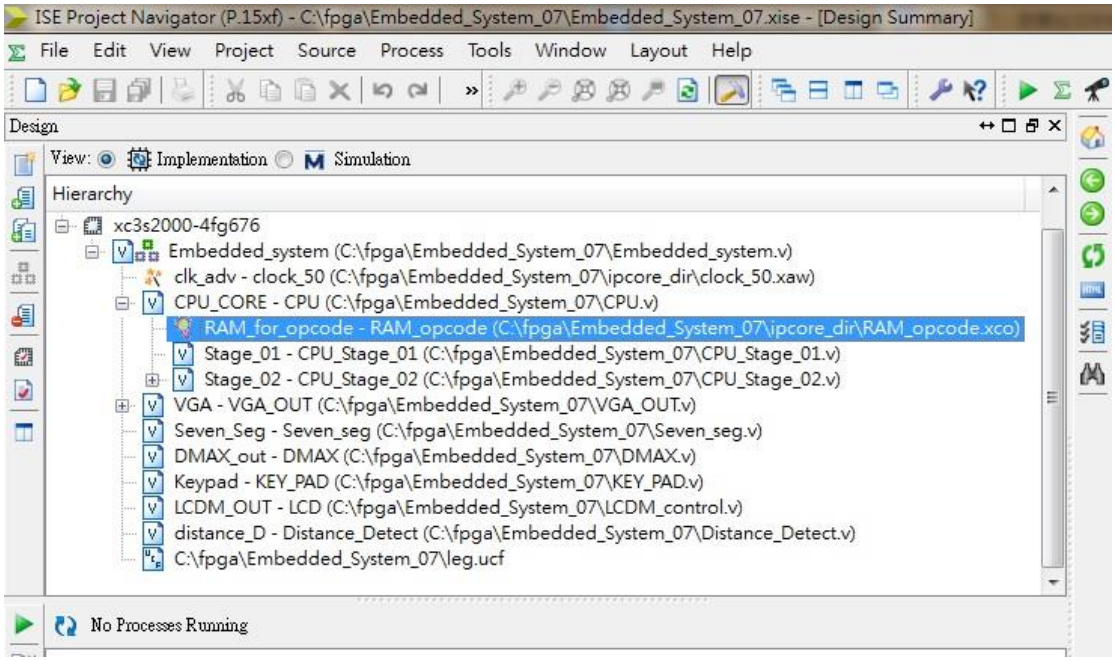

圖 4-3-1

2. 之後設定 RAM 大小，並設定 RAM 行為模式，如圖 4-3-2。

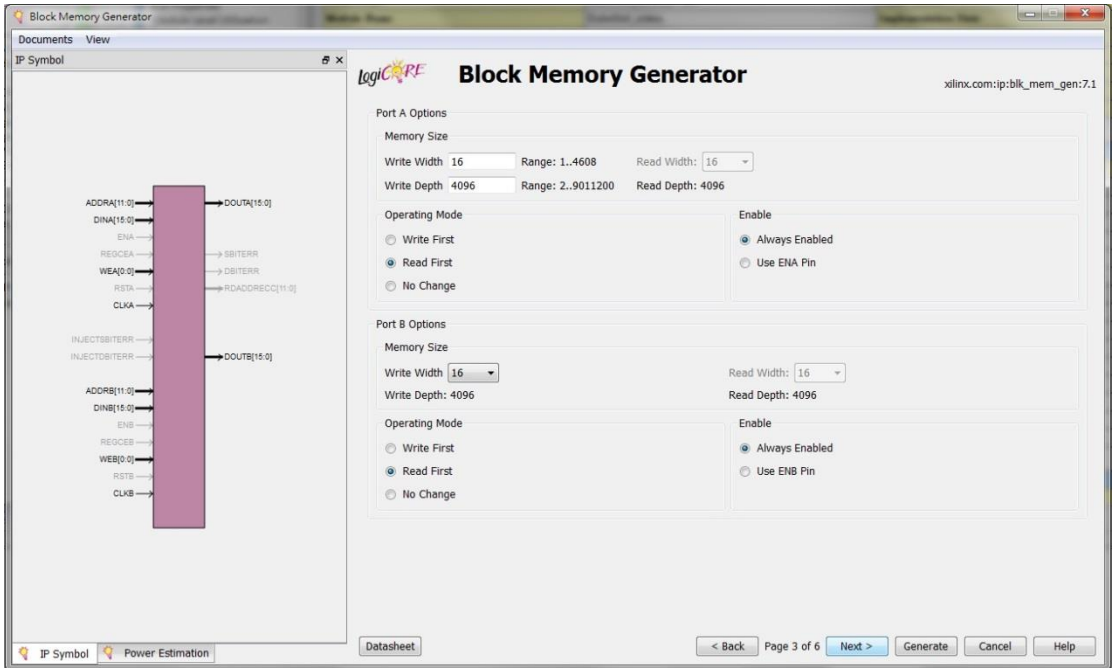

圖 4-3-2

3. 紅色框起部分為匯入 COE 檔，點選 Browse，如圖 4-3-3。

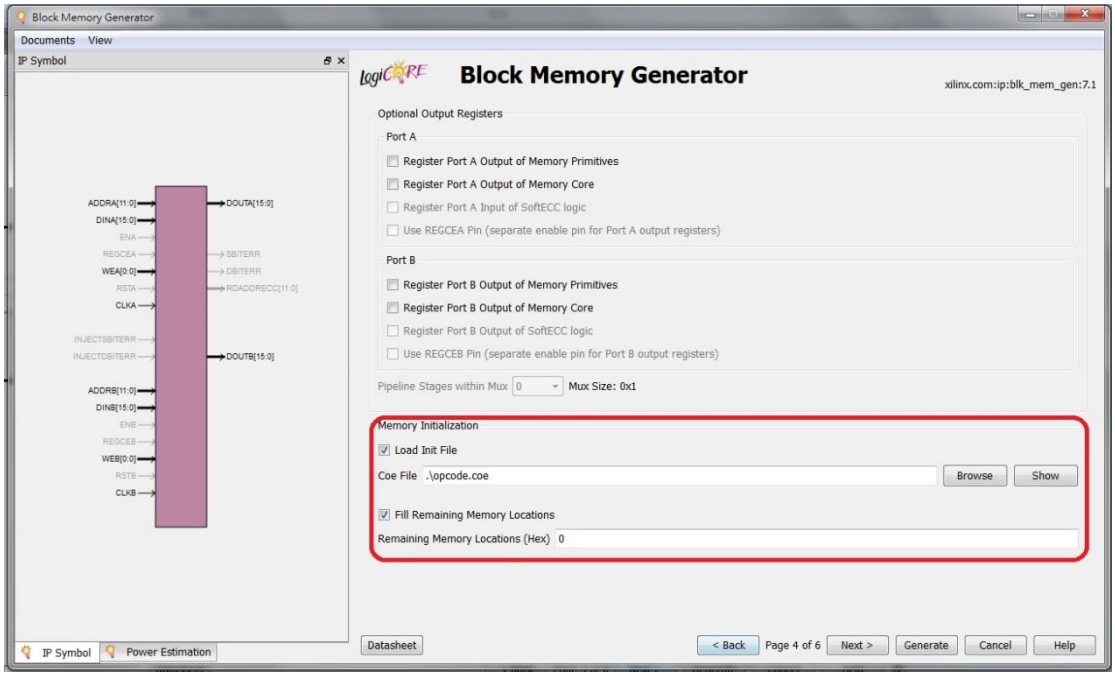

圖 4-3-3

4. 選取要匯入的 COE 檔，如圖 4-3-4。

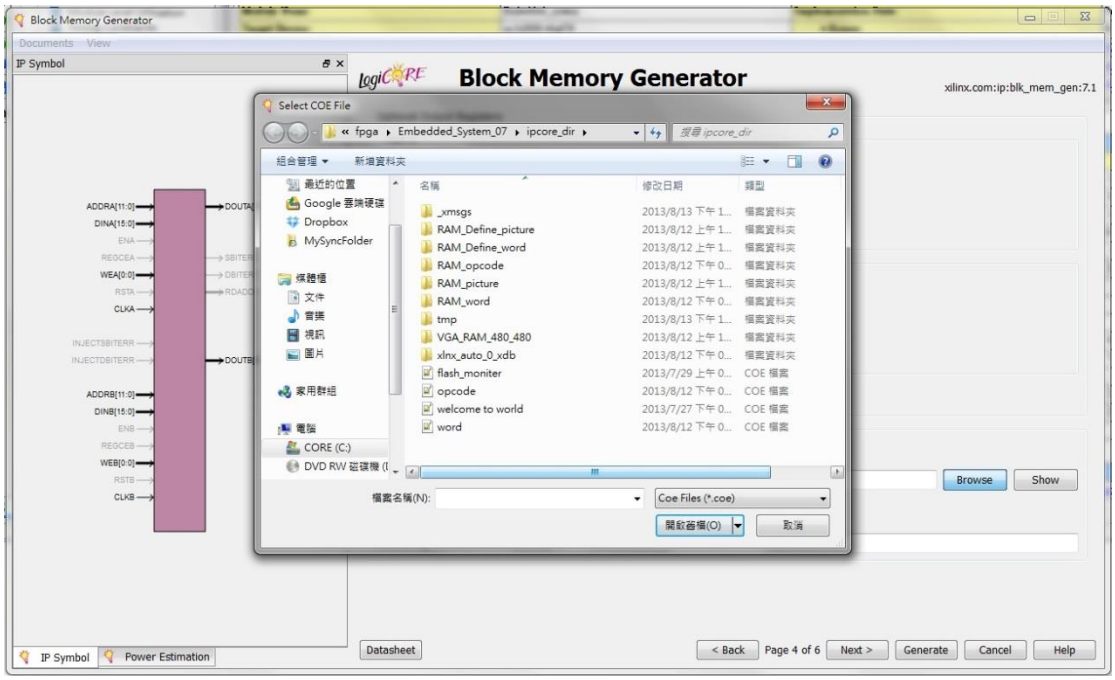

圖 4-3-4

5. 可點選 Show 確認每筆資料對應的 Address 位置，如圖 4-3-5。

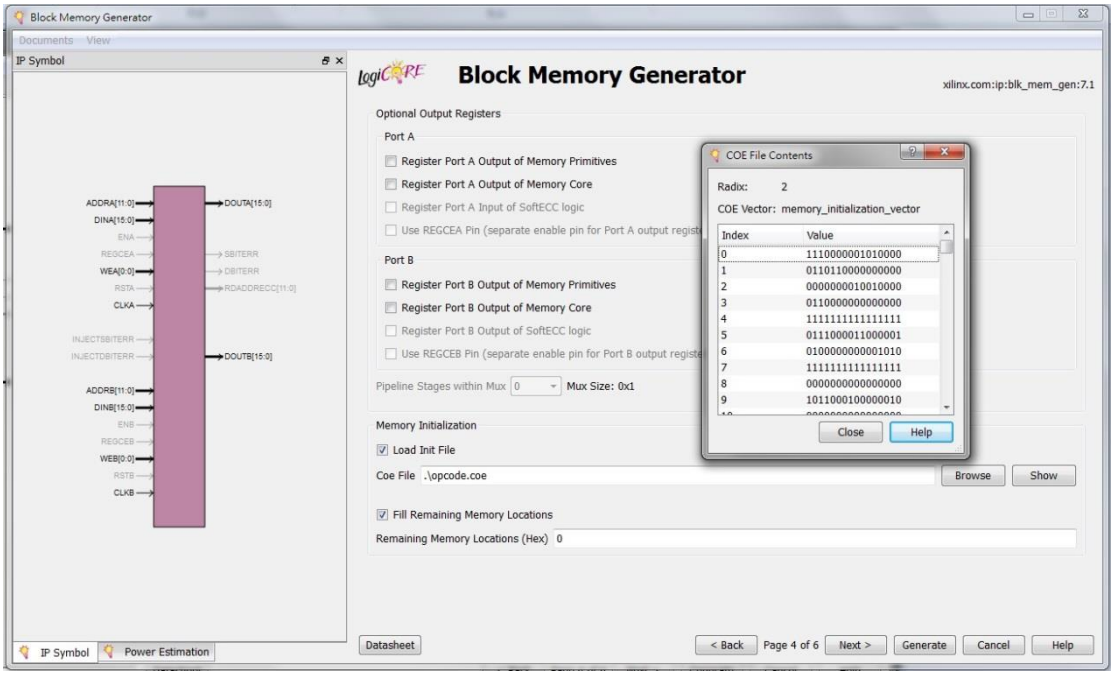

圖 4-3-5

4.4 利用 C 語言建構 COE 檔流程

如圖 4-4-1，此為以 C 語言將組合語言轉為機械語言的流程圖，此組譯器程式除了能夠將組合語言轉譯外，為了避免程式撰寫時的失誤，特別加入偵錯設計。

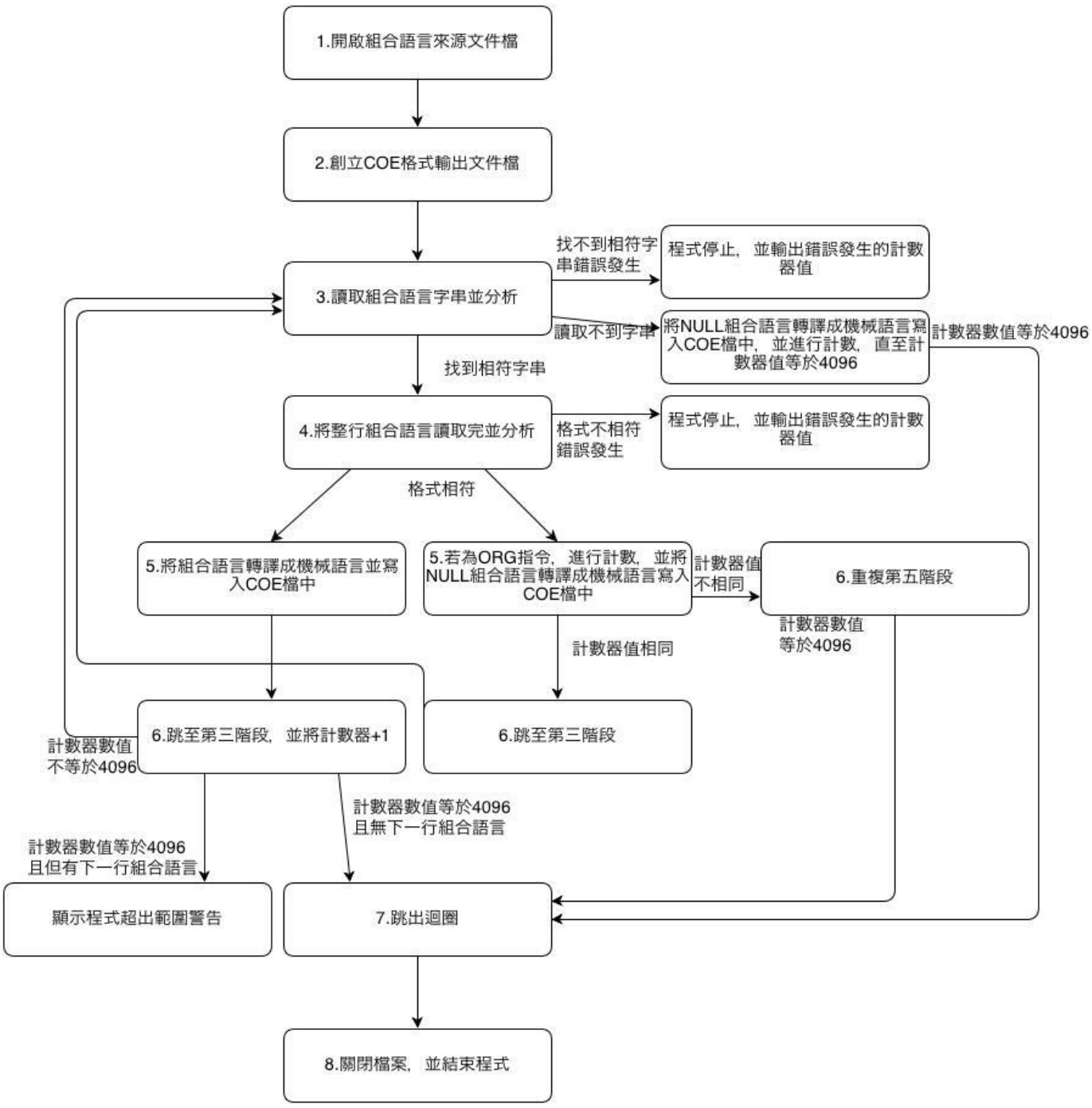

圖 4-4-1

4.5 利用 C 語言建構 COE 檔方法

可利用 C 語言製作 Assemble 組譯器，所需要的 Library 如下:

1. stdio.h
2. stdlib.h
3. string.h

組譯器的目的在於將字串轉換成為 0 與 1 的集合，並輸出 coe 檔。

可利用 fopen 函式讀取組合語言來源檔案，fscanf 函式讀取組合語言，然後利用 strcmp 函式作字串分析，如果正確才開始進行分析，然後再轉換成為需要的字串數值，成功轉換之後，使用 fprintf 函式將轉換後的字串數值打印在目的檔。

4.6 完成組譯器開發

完成後的組譯器，添加除錯機制，避免組合語言格式出錯而未發覺，並且為了提供方便的遠程跳躍、中斷設計等等功能，添加了 ORG 指令，可以設定程式以及資料的起始位置，如圖 4-6-1。

ORG 008 起始位置為 0x008，Address 是從 0x000 開始，因此 CALL 050 指令是從第九行開始，加上前兩個格式設定，剛好出現在第 11 行。

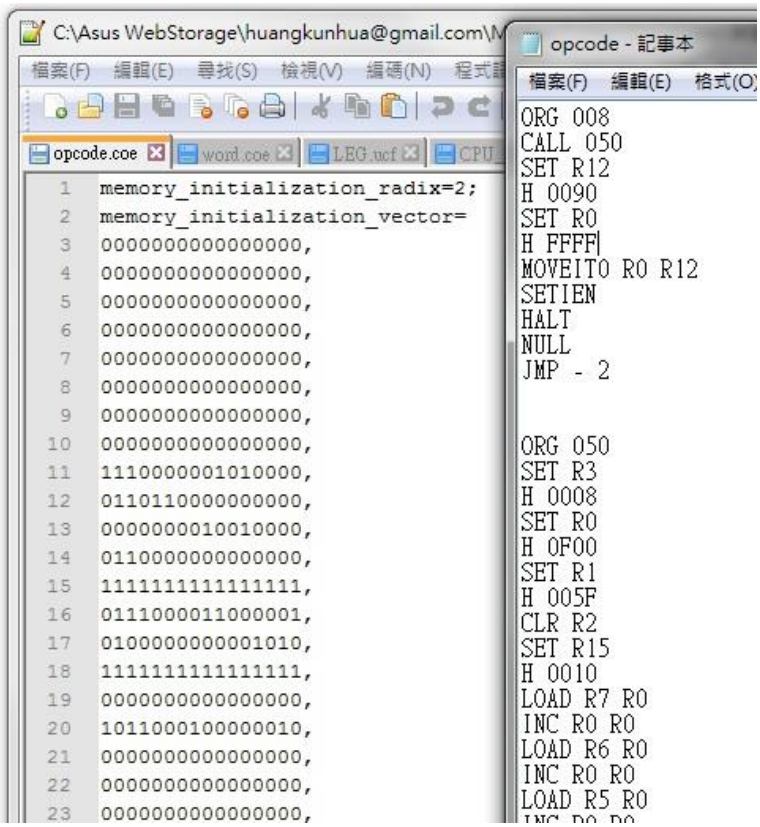

圖 4-6-1

若是程式出現格式出現錯誤，便會指出錯誤是發生在對應的 ADDRESS 第幾行，如圖 4-6-2。比如 opcode 文件中第 11 行的 JMP - 2 指令格式出錯，而此指令所對應的 ADDRESS 剛好為第九行。

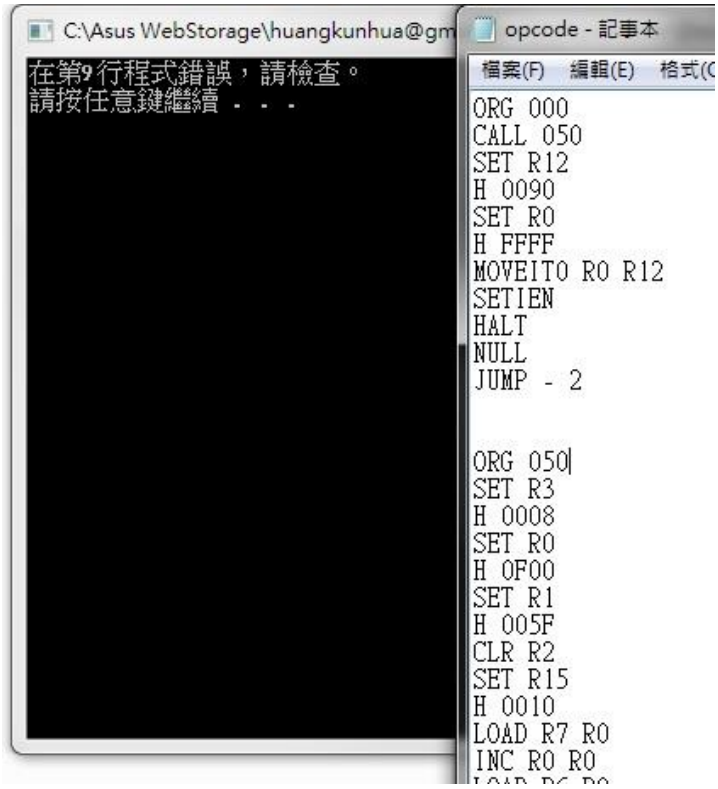

圖 4-6-2

若是程式超出 4096 行，便會跳出程式超出長度的警告，如圖 4-6-3。

圖 4-6-3

最後，程式組譯完成之後，會將程式填滿至最後一行，如圖 4-6-4。

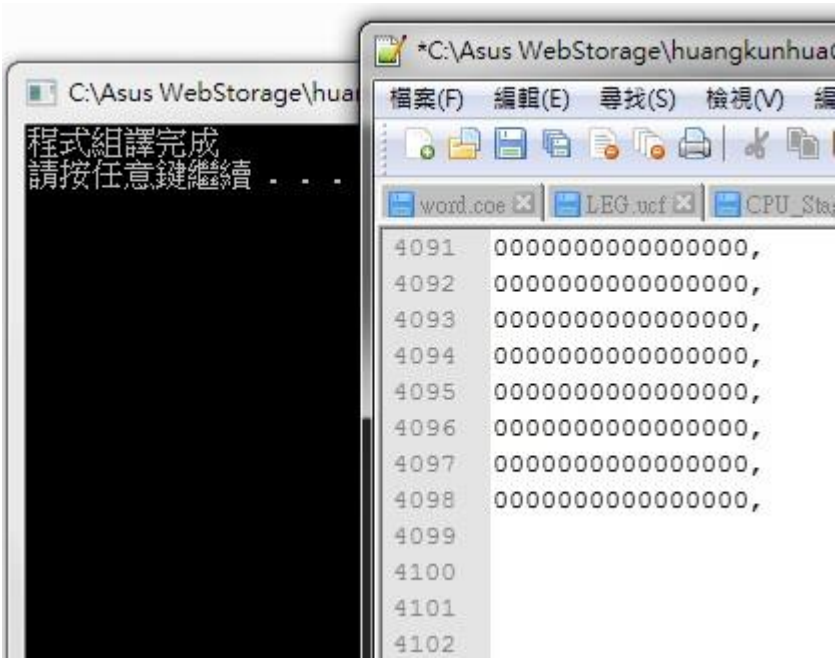

圖 4-6-4

第五章 設計周邊元件控制

5.1 周邊元件控制設計前言

周邊元件控制設計，是為使開發出來的微處理器得以運作、控制周邊元件的方式，在完成微處理器的開發設計後，可以透過各種的協定方式，與周邊元件相互溝通，這便是開發此微處理器的目標，但對於開發出完整的平台，並非本文重點，因此在此會簡述設計流程以及成果。

5.2 周邊元件控制方法

此微處理器的輸出設計，為了免除複雜的設計，因此去除交握電路的設計，同時因為沒有交握電路的設計，在輸出上為了避免產生重複的輸出控制，設計一種解決方法，也就是輸出值只會停留一個時脈週期，經過一個時脈週期之後，便會歸零，透過這種方式，可避免重複控制周邊元件。

如圖 5-2-1，透過元件與微處理器的中間協定，在發生輸出變化的時候，便會擷取輸出的數值，並執行數值所協議的指令模式，然後進行執行這項指令。

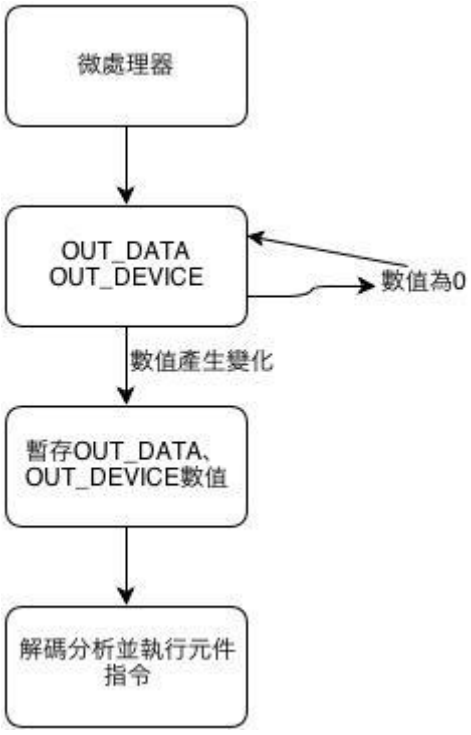

圖 5-2-1

5.3 周邊元件溝通控制介面結構

如圖 5-3-1，此為溝通控制介面結構，使用微處理機作為核心，通過周邊元件溝通控制介面，對周邊元件進行控制。

此種結構相比單片機的操控行為。有明顯的優點，可先自定義各種元件行為模式，然後透過微處理器操控元件，可大幅減少微處理器的計算負擔，並能靈活建構各種複雜行為，這是單片機所無法比擬的。

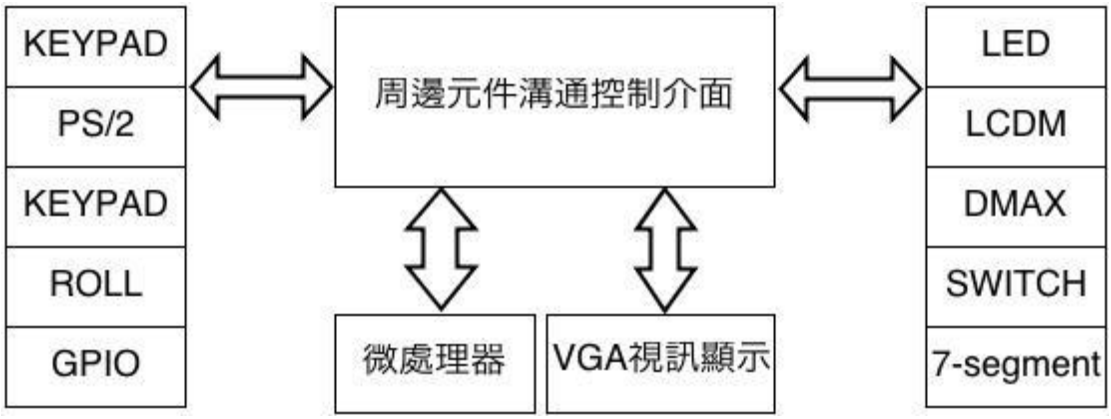

圖 5-3-1

5.4 周邊元件 VGA 顯示介面建構範例

此 FPGA 平台，VGA 輸出共可使用 8 色，因顏色過於稀少，以及考量完善的顯示介面所需的內部記憶體容量，因此以單色進行顯示。顯示上為求方便設計開發，畫面解析度為 480X480，若以每個像素點來對應 RAM，在單色的顯示介面上共需要 230400 位元。

如圖 5-4-1，此顯示介面架構設計目的是為同時顯示文字、圖形，以及可即時繪圖。

Word_posdition:其功能為定義字元的位置、編碼、編號，可同時顯示 1024 個字體。

Word_Graphic:其功能為儲存字元圖形檔，並將設定的字元輸出至顯示暫存做為緩衝

Picture_posdition:其功能為定義圖形的位置、編碼、編號，可同時顯示 1024 個 16x16 點陣圖形。

Picture _Graphic:其功能為儲存圖形 16x16 點陣圖檔，並將設定的圖形輸出至顯示暫存做為緩衝。

480x480_paint:其功能提供即時繪點使用，可對畫面的每一點進行繪圖。

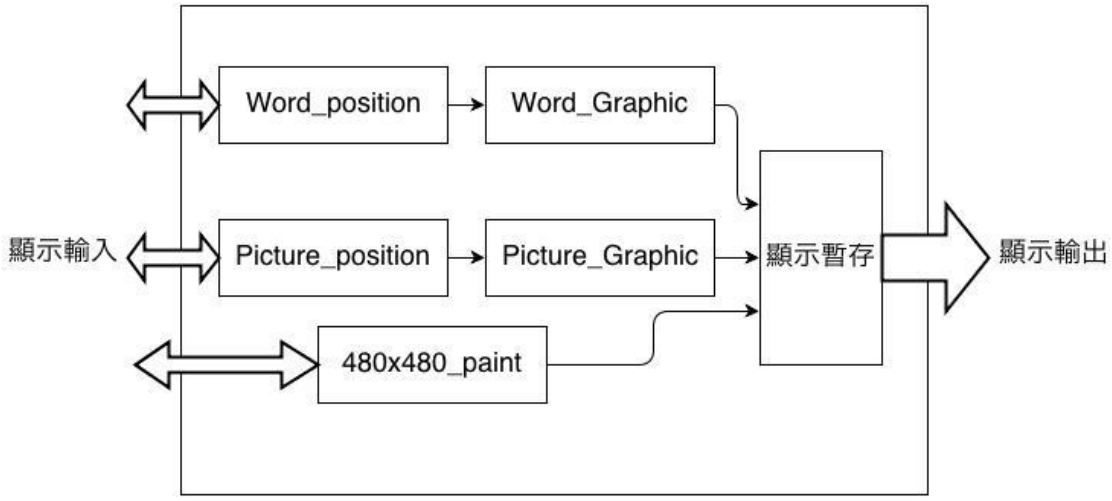

圖 5-4-1

第六章 成果展示

6.1 前言

此成果的目的在於顯示此微處理器的運算能力以及此嵌入式平台的相容性，因此著重於顯示方面，同時為了測試周邊相容性，特別準備超音波測距 HC-SR04 模組，進行距離測量。

利用超音波模組進行測距，利用微處理器取得距離數據後，在七段顯示器上顯示其測量距離，且微處理器將距離轉換成半徑後，在螢幕上對每一點進行判別並即時繪出圓形，並顯示字體字串在螢幕上。

6.2 周邊元件使用介紹

如圖 6-2-1，此程式使用到 LCDM、七段顯示器、超音波測距模組以及 VGA 顯示輸出。此程式利用超音波測距後所取得的數據，將距離顯示在七段顯示器上面，且利用此數據進行即時繪圖。

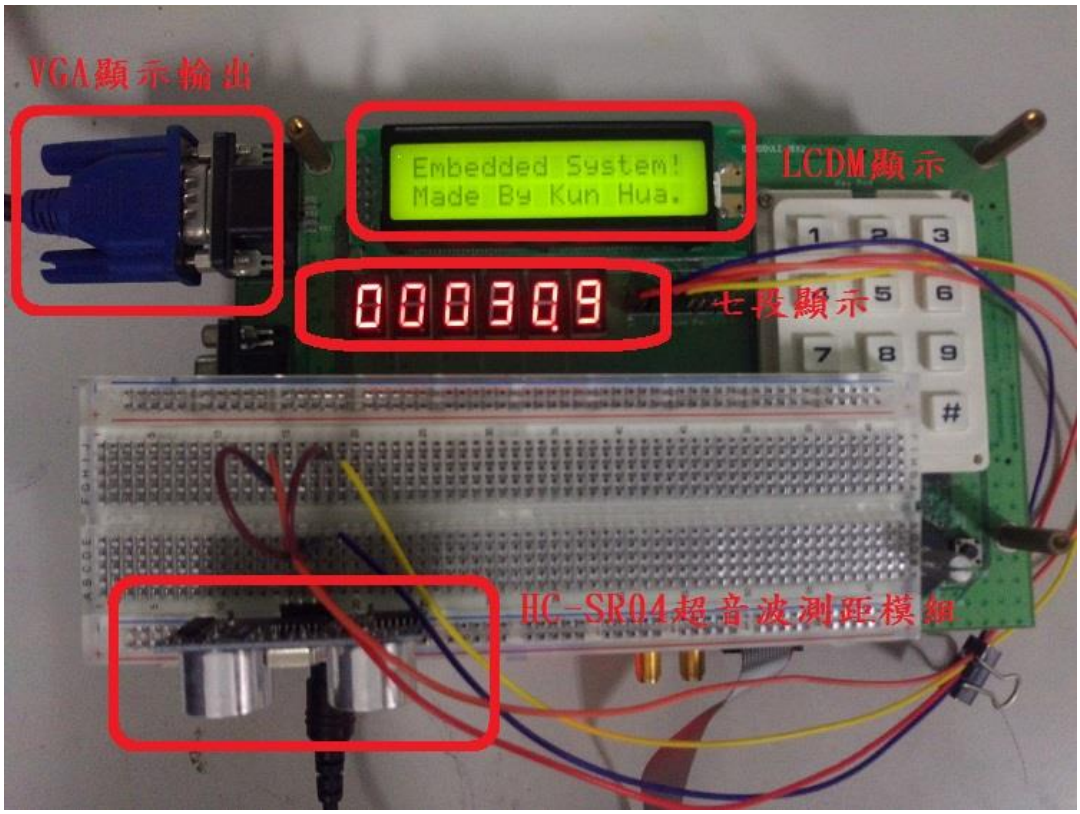

圖 6-2-1

6.3 成果展示

如圖 6-3-1，七段顯示器上所顯示的距離為 30.9 公分時，圖 6-3-2，在螢幕上繪出一個完整的圓型，並同時顯示文字介紹此系統。

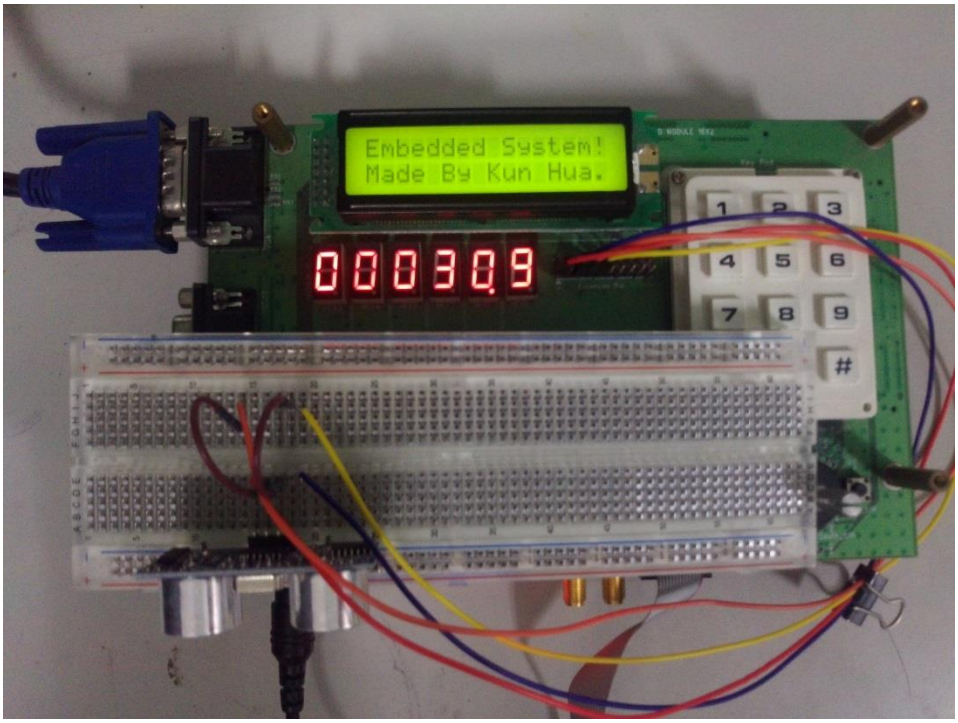

圖 6-3-1

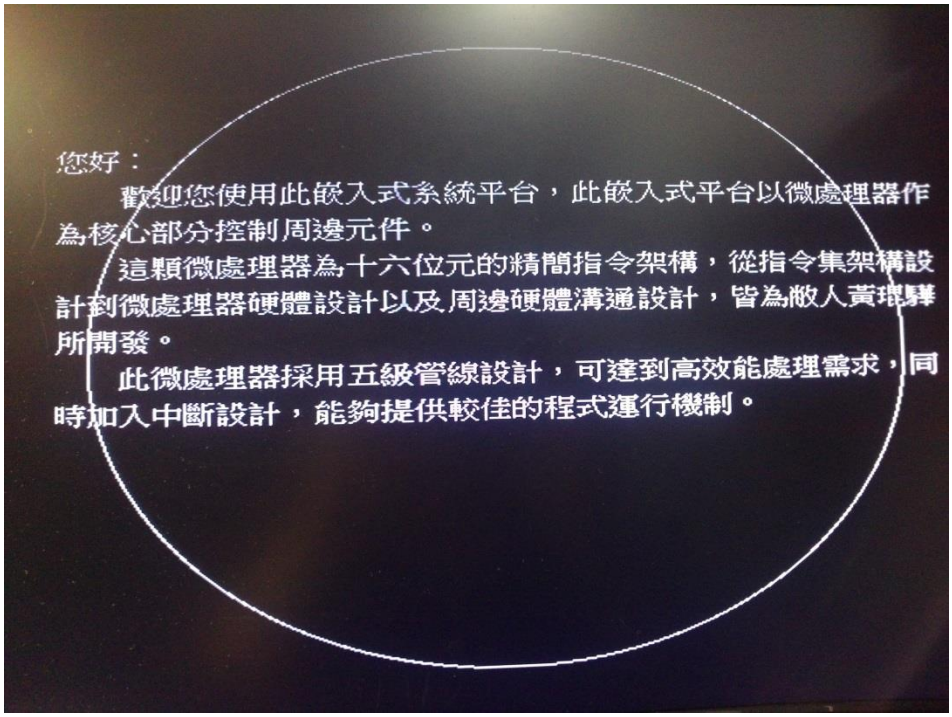

圖 6-3-2

如圖 6-3-3，七段顯示器上所顯示的距離為 19.2 公分時，圖 6-3-4，在螢幕上變更半徑並繪出一個完整的圓型，且同時顯示文字介紹此系統。

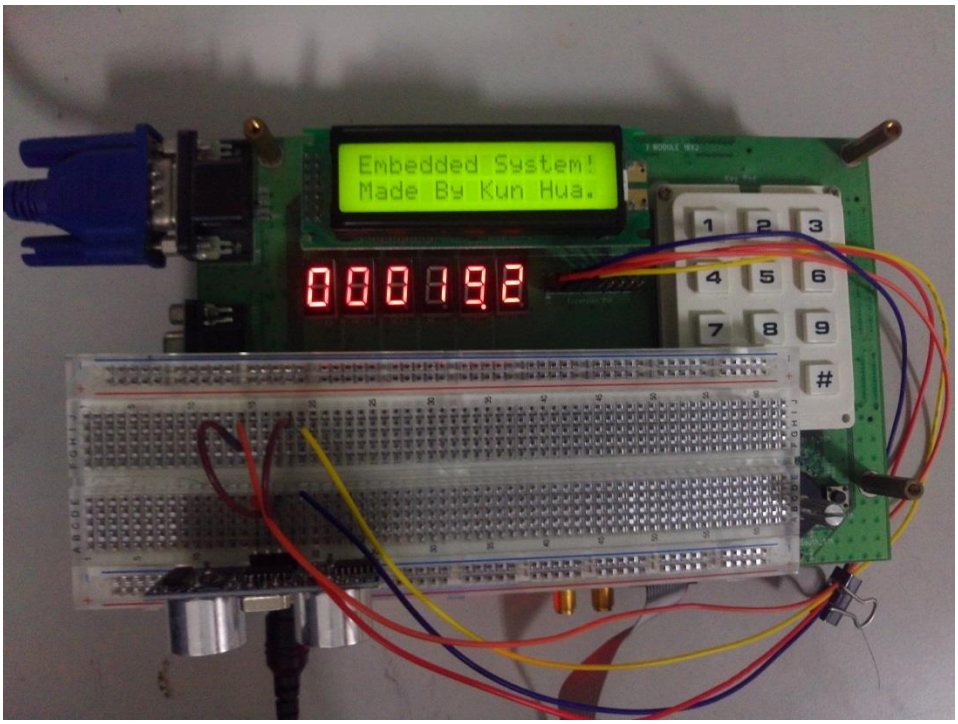

圖 6-3-3

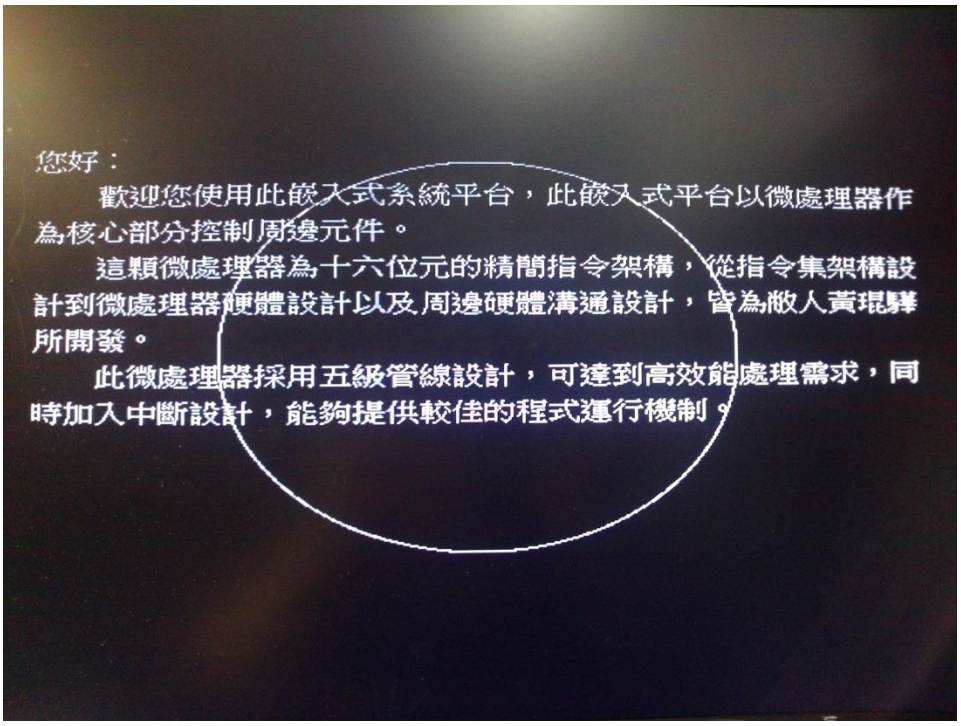

圖 6-3-4

如圖 6-3-5，七段顯示器上所顯示的距離為 6.2 公分時，圖 6-3-6，在螢幕上變更半徑並繪出一個完整的圓型，且同時顯示文字介紹此系統。

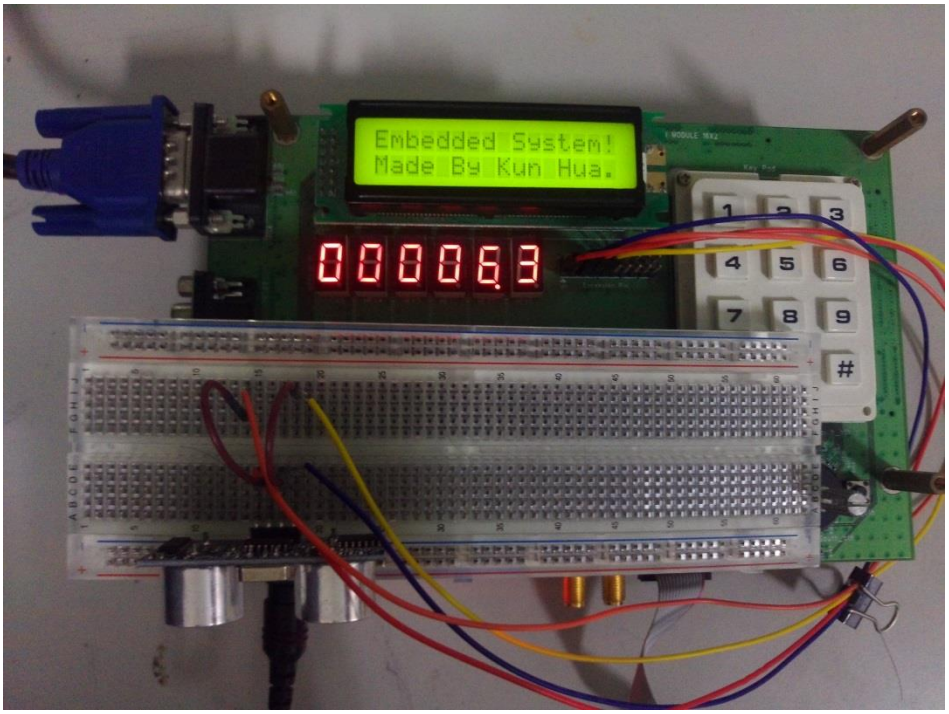

圖 6-3-5

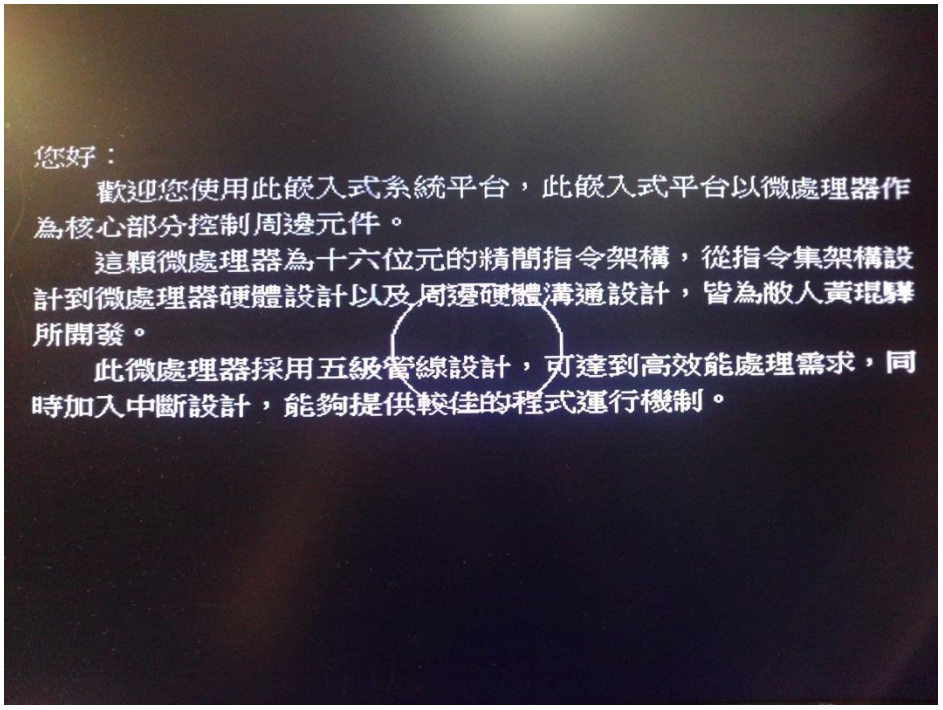

圖 6-3-6

結語

在此嵌入式研究開發上，投入最大心力的便是微處理器的設計與開發，最一開始的設計是從八位元的簡易微處理器開始設計的，經過層層的開發、改良、擴充，進而開發出十六位元的微處理器，同時為了深入發展微處理器的應用，透過研究各種周邊元件的行為模式，以合理的方式進行周邊元件控制設計，並開發出此嵌入式平台。

此嵌入式平台的最大特點在於可靈活改變各種元件的操作行為，利用硬體電路對元件的各種操作行為進行控制，微處理器並不需要負擔太大的運算量，便能夠輕易地對各種元件進行控制。

未來的研究方向將會是利用此嵌入式系統平台開發機器人，利用此平台靈活的控制方式，可輕鬆的外接各種模組架構，並透過微處理器進行控制。