

# 計算機結構學

## 16 bits MIPS CPU 設計

班級：資訊三乙

姓名：吳承翰

學號：D0908786

日期：2023/06/04

# CPU Design

## 關於指令的處理

- 從 Program Counter 抓取指令 (fetch)
- 讀取暫存器
- 根據指令類別，如 R, I, J 三種類別，決定以下
  - 使用 ALU 計算得到以下
    - ALU 計算結果
    - 用於 load/store 的記憶體地址
    - Jump 用於跳躍到的 Target address
- 將記憶體的資料讀出 (load/store)
- Write back 到暫存器
- Program Counter += 4

## 定義要實現的指令集

Name	Format	Bit Field					Ex
		opcode 4 bits	rs 3 bits	rt 3 bits	rd 3 bits	funct 3 bits	
add	R	0	2	3	1	0	add \$1, \$2, \$3
sub	R	0	2	3	1	1	sub \$1, \$2, \$3
mul	R	0	2	3	1	2	mul \$1, \$2, \$3
div	R	0	2	3	1	3	div \$1, \$2, \$3
slt	R	0	2	3	1	4	slt \$1, \$2, \$3
and	R	0	2	3	1	5	and \$1, \$2, \$3
or	R	0	2	3	1	6	or \$1, \$2, \$3
jr	R	0	7	0	0	7	jr \$7
lw	I	1	2	1	14		lw \$1, 7(\$2)
sw	I	2	2	1	14		sw \$1, 7(\$2)
beq	I	3	1	2	7		beq \$1, \$2, 7
bne	I	4	1	2	7		bne \$1, \$2, 7
addi	I	5	2	1	7		addi \$1, \$2, 7
subi	I	6	2	1	7		subi \$1, \$2, 7
stli	I	7	2	1	7		stli \$1, \$2, 7
j	J	14	7			j 7	

# 邏輯設計基本單元

---

我們可以簡單的將邏輯設計基本單元分成兩個部分，分別為 Combination Elements 以及 Sequential Element

## Combination Elements

包含 AND, OR, Adder, Multiplexer 等元件，將這一些元件與 Multiplexer 組合可以組成 ALU，根據給定到 Multiplexer 的訊號可以選擇 ALU 的功能。

## Sequential Elements

這部分為儲存元件，儲存元件在基本邏輯設計中，包含 latch，如 D-latch（電位觸發），flip-flop，如 D flip-flop（邊緣觸發），邊緣觸發又可以分成正緣觸發以及負緣觸發，通過組合這一些 Sequential Elements，我們可以構建出暫存器。

## 分析指令集需求

---

從定義要實現的指令集中可以看到，如果我們需要 LW, SW，我們會需要讀取 rs 和 rt 的值，而在寫入資料的部分，有時候我們要寫入到 rd，如 ADD，有時候則需要寫入到 rt，如 ADDI，一個是 R format，另外一個是 I format，這是與 RISC-V 不同之處，在 RISC-V 中沒有這個問題，但相對來說指令集有更多的種類接著我們需要 Program, zero-extension, sign-extension，用於 ADD, SUB，計算 Program counter += 4 的 ALU。

有了以上便可以實現出一個簡單的指令集子集。

關於 RISC-V 指令集的研究可以參考以下網址

[Day-06 RISC-V 簡介, Microkernel vs Monolithic kernel](#)

作者：我自己

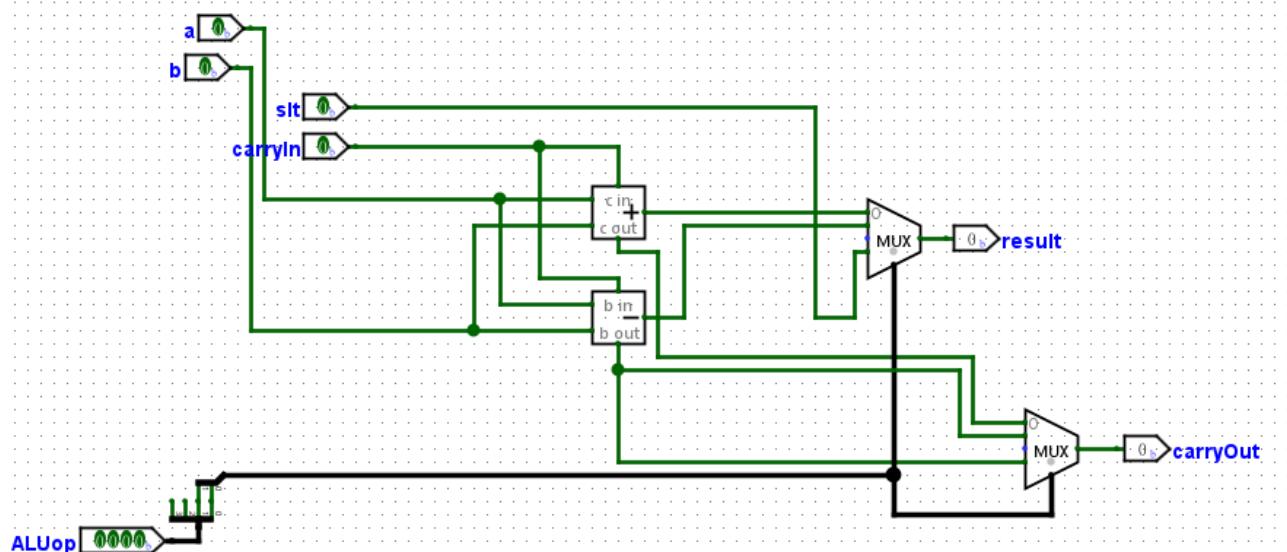
# 建立 Datapath

對於一個 CPU 的設計，我們需要以下基本元件

- Register file
- ALU
- Instruction Memory

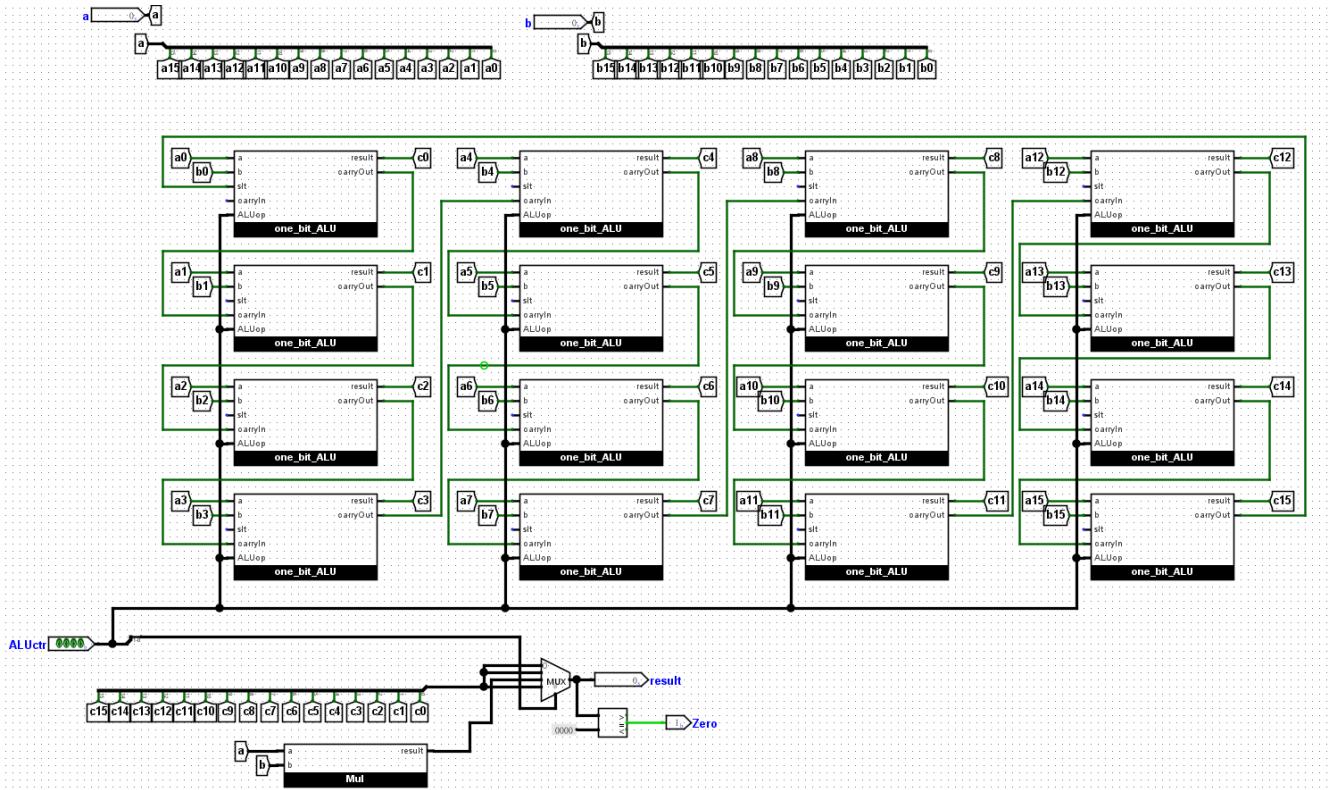
## ALU

ALU 設計十分簡單，我們只需要將 ALU 需要實現的功能，通過多工器進行串聯即可，以下為 1 bit ALU 設計。



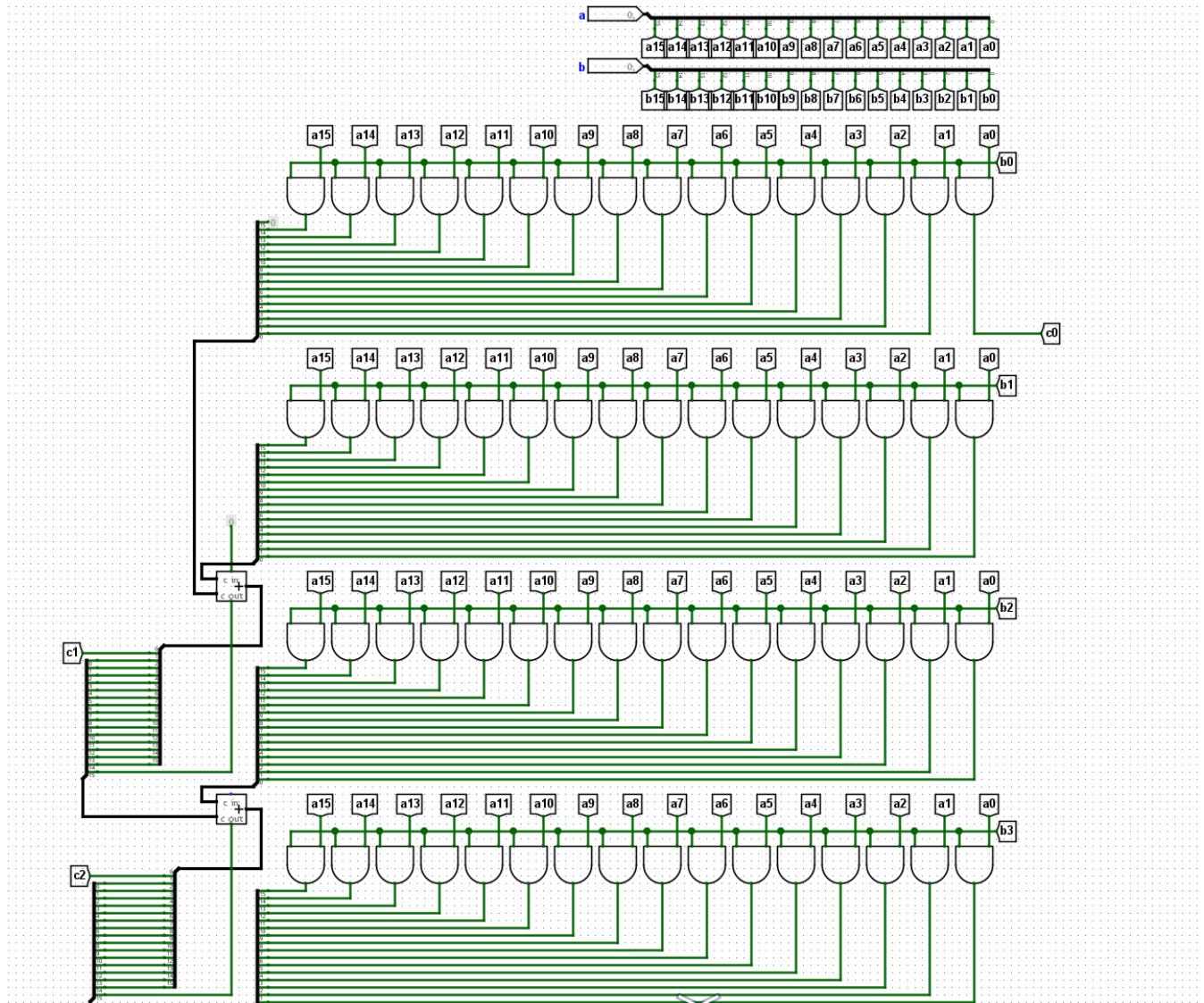
- *a*: 輸入資料 A
- *b*: 輸入資料 B
- *ALUop*: 控制 ALU 行為，00 表示加法，01 表示減法，10 表示乘法，11 表示 set on less than
- *carryIn*: 進位輸入
- *carryOut*: 進位輸出
- *result*: 計算結果

目前在 ALU 中實現了加法，減法，乘法運算，乘法的部分後面將介紹，通過 *ALUop* 控制 ALU 行為，並得到運算結果，接著將多個 1 bit ALU 進行串接，便可以得到 16 bits ALU



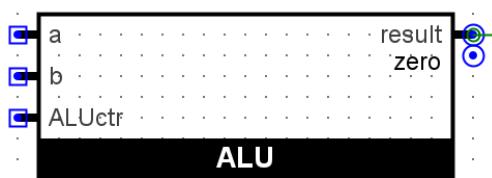
可以看到我們將多個 1 bit ALU 進行串接，將輸出的 16 bit 進行串接，在下方我們串接了 1 個多工器，多工器訊號為 ALUctr，0000 表示加法，0001 表示減法，0010 表示乘法，這邊將乘法器 Mul 的輸出拉到了多工器上，使用 ALUctr 選擇輸出，而 ALU 我們加上了 Zero Flag，判斷運算結果是否為 0，可用於 bne, beq。

為了實現乘法功能，單獨設計了 16 bit 的乘法運算單元，以下為電路，同樣參考數位邏輯設計第 6 版得到以下設計



串接多個 1 bit 乘法便可以實現，overflow 的部分忽略。

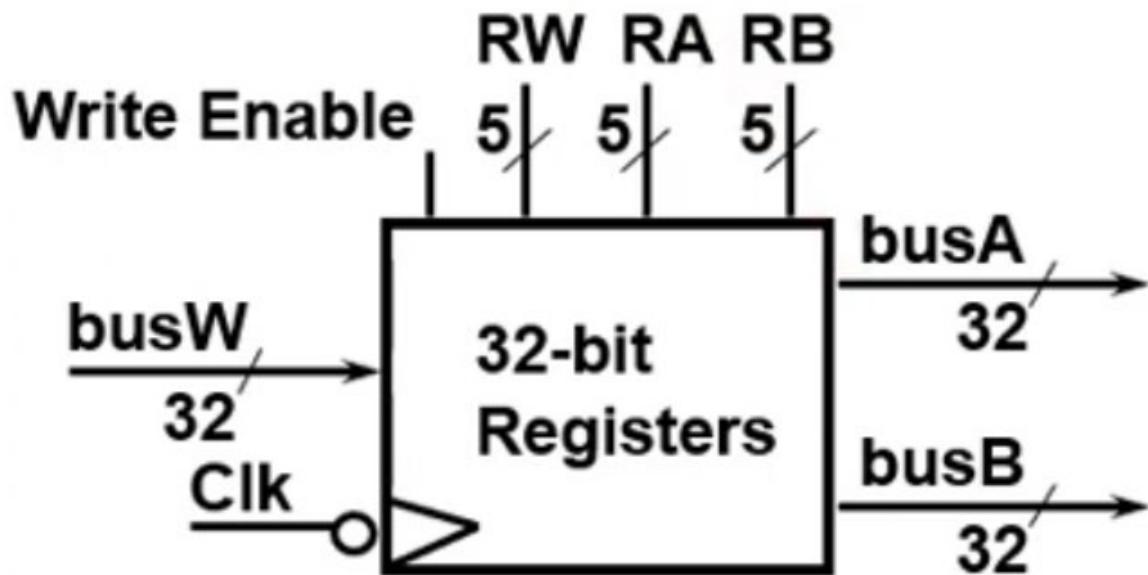
接著看到 16 bits ALU 的 Block Diagram



- $a$ : 輸入資料 A
- $b$ : 輸入資料 B
- $ALUctr$ : 控制 ALU 行為，0000 表示加法，0001 表示減法，0010 表示乘法
- $result$ : 計算結果
- $zero$ : 計算結果為 0 則 1，反之為 0

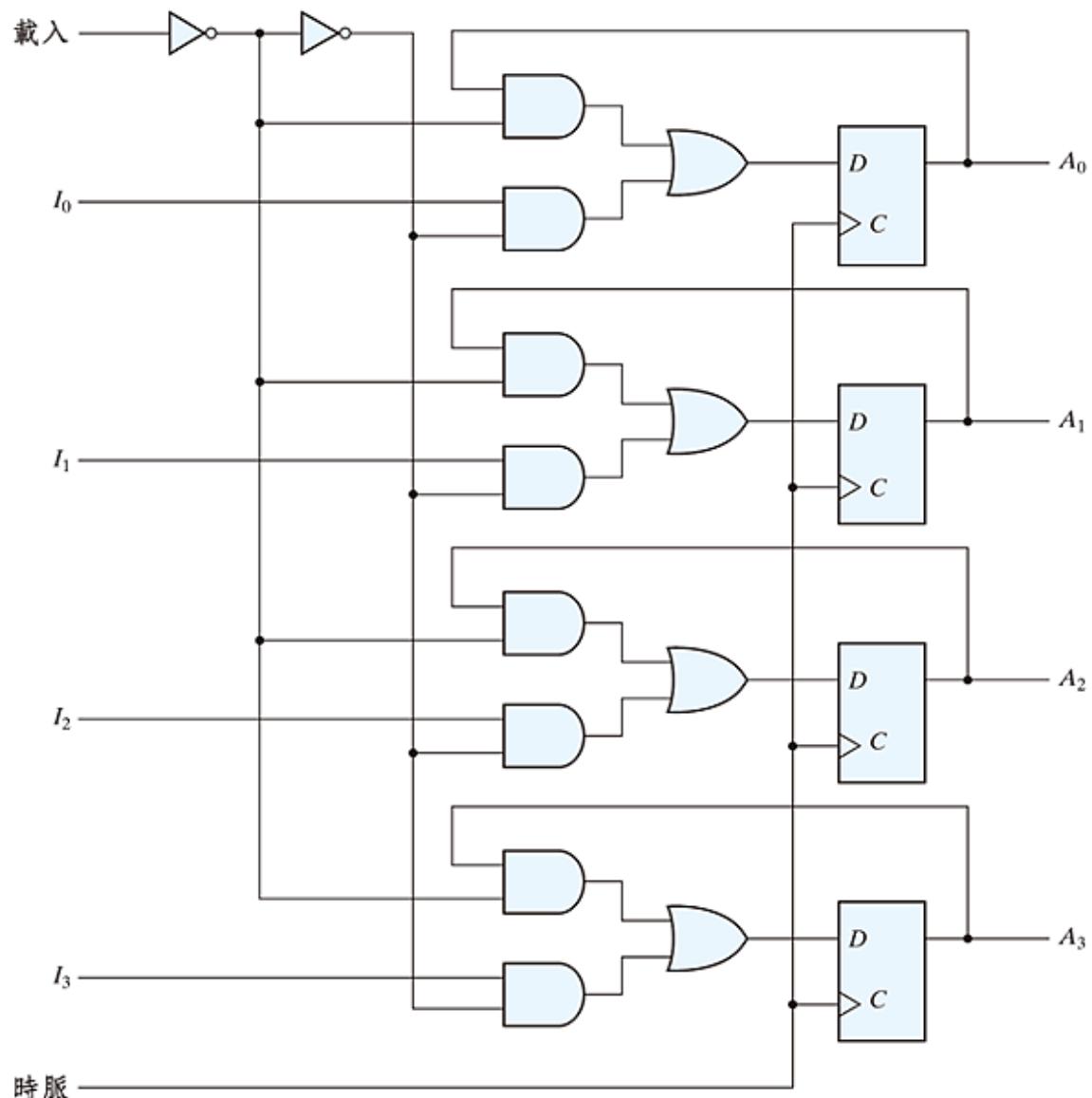
## Register file

在本次設計中，我們設計的為 16 位元的 MIPS 暫存器，接著分析我們設計的指令集，發現 rs, rt 欄位只有 3 bits，也就是說我們只有 8 個暫存器可以使用，其中將 0 號暫存器作為 zero 進行保留，7 號暫存器引用了 x86 微架構的想法，我將 7 號暫存器保留作為 EFLAGS 暫存器使用，可以紀錄兩個數字的大小關係，而後可以用於 bne, beq 以及 if-else 的處理，以下為 Register file 的概念圖。



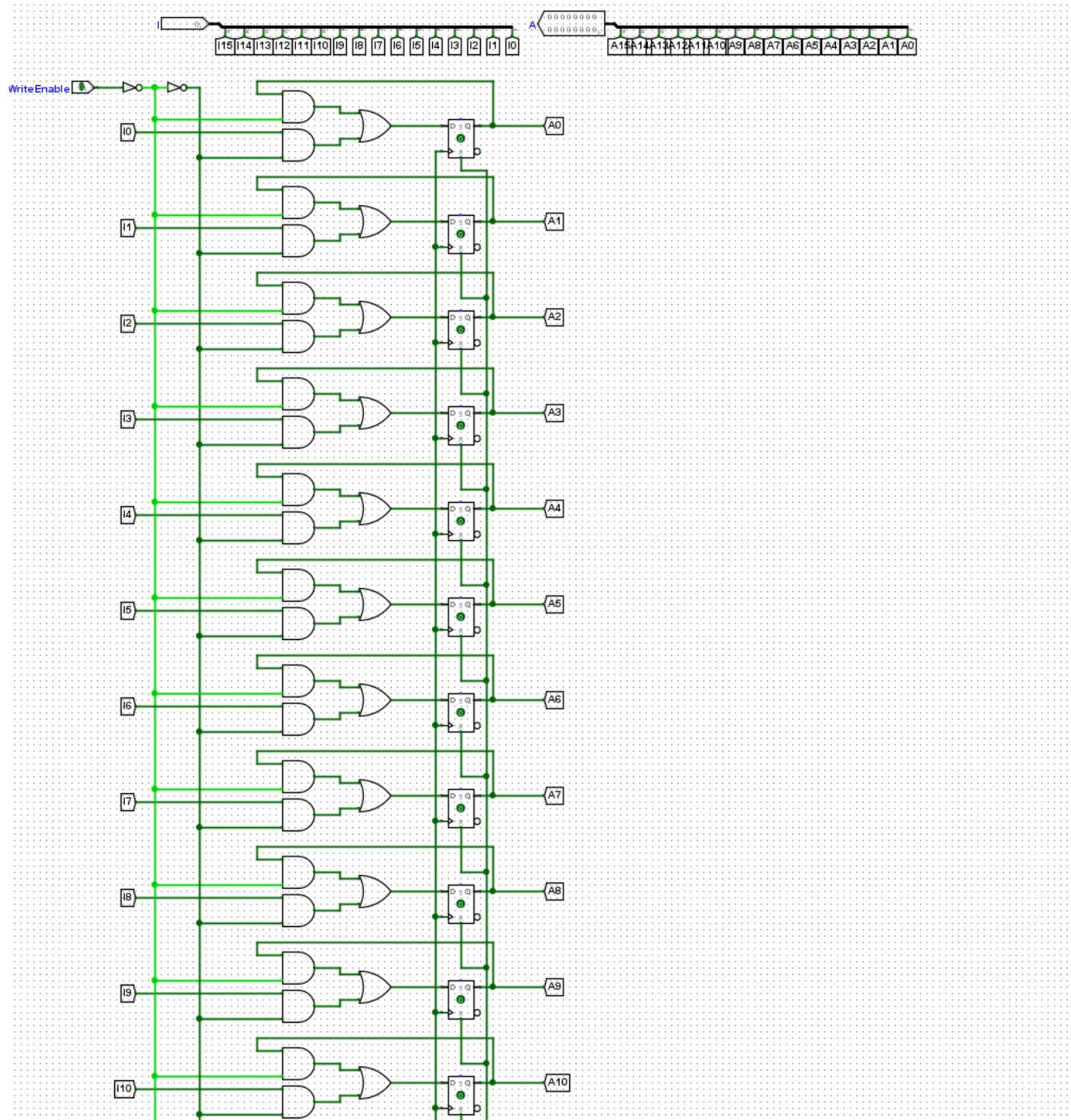
- RW: 指定要寫入到的暫存器，總共有 5 bits 長度
- RA: 讀取 A 暫存器的資料，並且由 busA 輸出
- RB: 讀取 B 暫存器的資料，並且由 busB 輸出
- busW: 要寫入到 RW 暫存器的資料

以下為 Register file 在電路圖中的實現，在 Register file 中，我們會輸入兩個暫存器編號，選擇我們要存取的暫存器，由於我們要選擇暫存器，因此我們必須先構造 16 bit 暫存器的部分，參考數位邏輯設計課本，如下圖所示

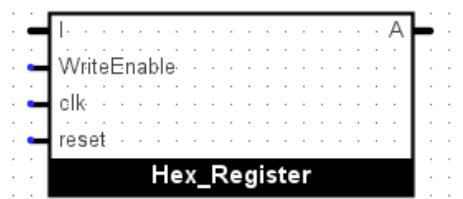


來源：數位邏輯設計 第 6 版

以下為 16 bit Register 的電路設計

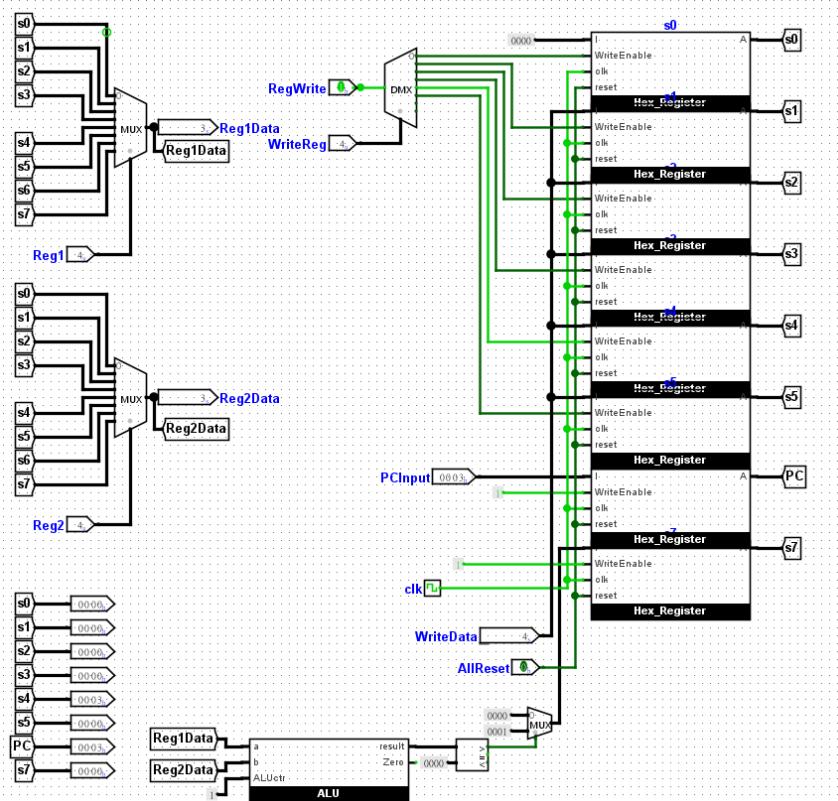


概念上為將多個 D flip-flop 串聯在一起，以下為 Block Diagram



- I: 輸入的資料
- WriteEnable: 0 表示不允許寫入暫存器，1 表示允許寫入暫存器
- clk: 為 clock
- reset: 重設暫存器的值，重設為 0，概念上為串接 D flip-flop 的 reset bit
- A: 從暫存器中讀取到的資料

接著將 16 bit Register 與多工器串接在一起，便可以完成 Register file 的設計



為了避免線路過於雜亂，使用觸點，如圖上的 s0 表示連接關係。

左半邊部分，輸入 Reg1 作為 Mux 的訊號線，用來選擇接下來要讀取的暫存器。右半邊的部分，RegWrite 會通過 DeMux 將訊號從 3 bit 分成多個通道，用於選擇接下來要寫入到的暫存器，而控制訊號如同左半邊的設計想法，給定 WriteReg 3 bit 控制訊號。

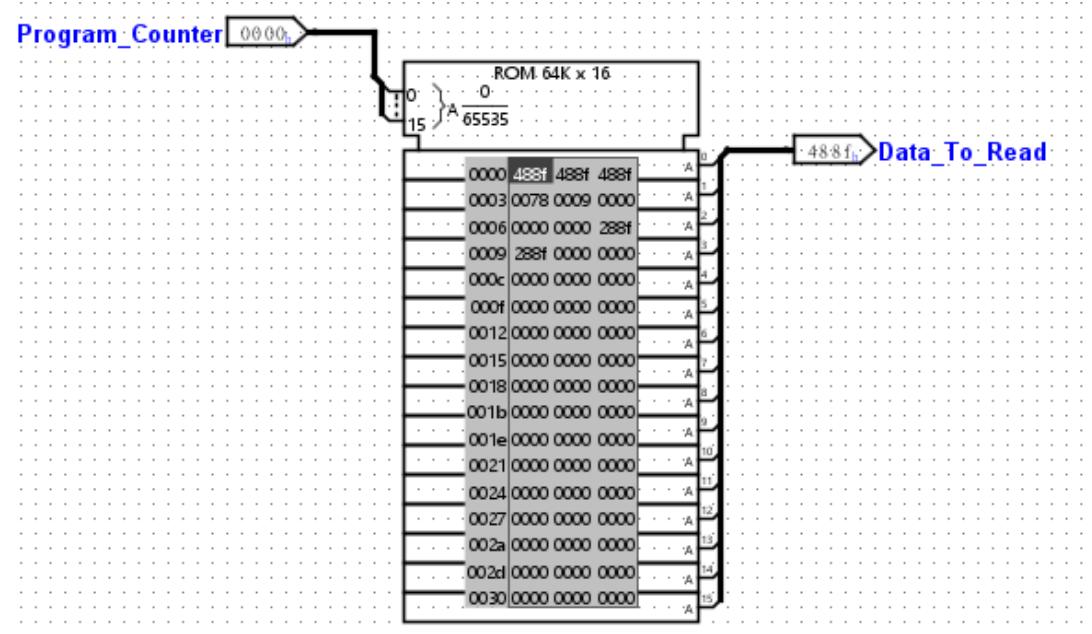
WriteData 為要寫入到暫存器的值，給一個暫存器都會接受到這個值，但是否能夠成功寫入取決於前面 RegWrite 給定的訊號，如此便可以達到指定暫存器寫入。

接著是 7 號暫存器，做為 EFLAGS 使用，會判斷兩個暫存器讀出來的值，接著通過 ALU 進行減法運算，通過比較器，和 0 進行比較，便可以知道大小關係，並將比較器做為多工器的控制訊號，將 7 號暫存器寫入成 0 或是 1。

為了 function 跳轉，保留 6 號暫存器作為 Program Counter 使用。

## Instruction memory

給定記憶體地址，讀取該地址中的資料，這邊可以直接使用 logisim-evolution 裡面自帶的 ROM 實作



到這邊，完成了 CPU 基本三大元件的設計了，接著要完成 Controller 的設計。

# Controller 設計

Controller 意義為給定 opcode，接著產生對應的訊號去驅動其他單元，或是選擇暫存器，舉例如果輸入 opcode 為 R-type，則會將控制權轉到 ALU Controller。

如果為 lw，則將 RegDst 設為 1，因為 lw 和 R-type 指令寫入的暫存器不同，以下為 Controller 訊號對照表

Binary											
	RegDst	Branch	MemRead	MemtoReg	ALUOp_2	ALUOp_1	ALUOp_0	MemWrite	ALUSrc	RegWrite	
add	1	0	0	0	0	0	0	0	0	0	1
sub	1	0	0	0	0	0	0	0	0	0	1
mul	1	0	0	0	0	0	0	0	0	0	1
div	1	0	0	0	0	0	0	0	0	0	1
slt	1	0	0	0	0	0	0	0	0	0	1
and	1	0	0	0	0	0	0	0	0	0	1
or	1	0	0	0	0	0	0	0	0	0	1
jr	1	1	0	0	0	0	0	0	0	0	1
lw	0	0	1	1	0	0	0	0	1	0	0
sw	0	0	0	0	0	0	0	1	1	0	0
beq	0	1	0	0	0	0	1	0	0	0	0
bne	0	1	0	0	0	0	1	0	0	0	0
addi	0	0	0	0	0	0	0	0	1	1	1
subi	0	0	0	0	0	0	1	0	1	1	1
stli	0	0	0	0	0	0	1	0	1	1	1
j	0	1	0	0x	x	x		0	1	0	

輸入為 opcode，輸出為 Controller 訊號，我們可以使用卡諾圖進行邏輯電路畫簡，經過化簡後，得到以下布林邏輯表達式

$$RegDst = a'b'c'd'$$

$$Branch = a'b'cd + a'bc'd' + abcd'$$

$$MemRead = a'b'c'd$$

$$MemtoReg = a'b'c'd$$

$$ALUOp_2 = x$$

$$ALUOp_1 = x$$

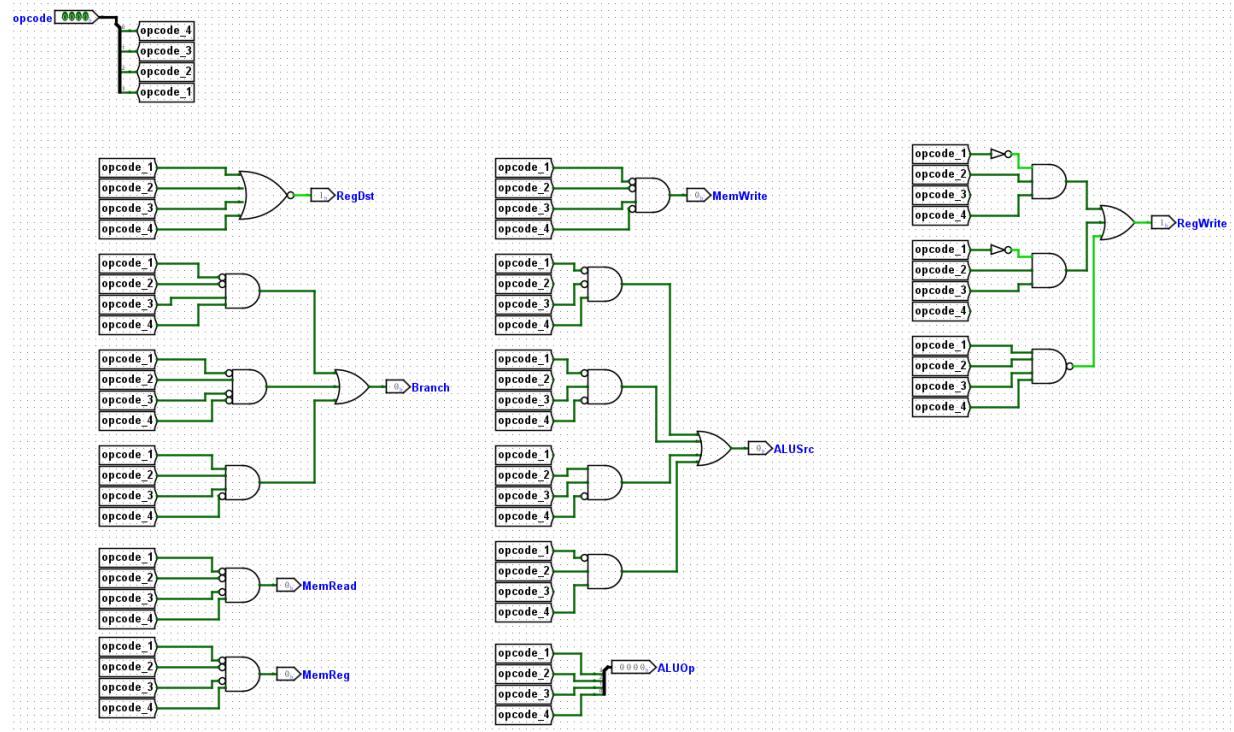
$$ALUOp_0 = a'cd + a'bd'$$

$$MemWrite = a'b'cd'$$

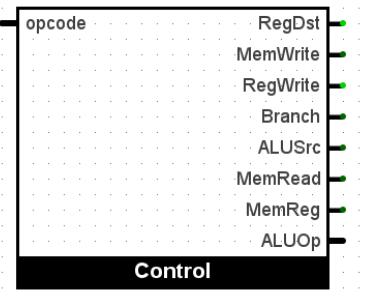
$$ALUSrc = a'c'd + a'cd' + bcd' + a'bd$$

$$RegWrite = a'bd + a'bc$$

接著將以上布林邏輯表達式轉換成電路，串聯後便完成 Controller 的設計



以下為 Control 的 Block Diagram



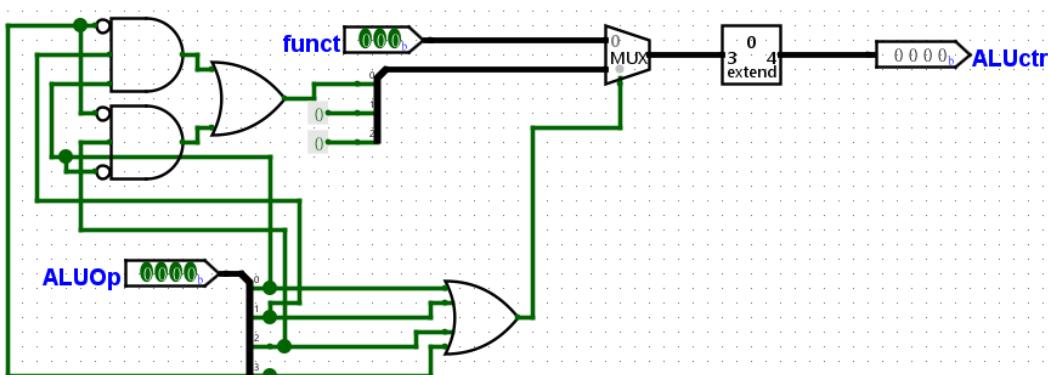
# ALU Controller 設計

ALU Controller 如同前面，輸入的為 ALUOp 加上 funct，總共 7 個 bit，如果 ALUOp 為 0，表示為 R-type 指令，後面 ALUctr 就直接取決於 funct 欄位了。

接著是其他 type，根據 ALUOp 我們決定其對應到的 ALU 行為，如 bne 對應到的 ALU 行為為減法，因此 ALUctr 應該為 001，根據這樣的關係得到下表

	opcode	ALUOp_2	ALUOp_1	ALUOp_0	funct_2	funct_1	funct_0	ALUctr_3	ALUctr_2	ALUctr_1	ALUctr_0
R-type	add	0	0	0	0	0	0	0	0	0	0
	sub	0	0	0	0	0	1	0	0	0	1
	mul	0	0	0	0	1	0	0	0	1	0
	div	0	0	0	0	1	1	0	0	1	1
	slt	0	0	0	1	0	0	0	1	0	0
	and	0	0	0	1	0	1	0	1	0	1
	or	0	0	0	1	1	0	0	1	1	0
	jr	0	0	0	1	1	1	0	1	1	1
	lw	1	0	0 x	x	x		0	0	0	0
	sw	1	0	0 x	x	x		0	0	0	0
	beq	0	1	0 x	x	x		0	0	0	1
	bne	0	1	1 x	x	x		0	0	0	1

接著將上表使用電路進行實作

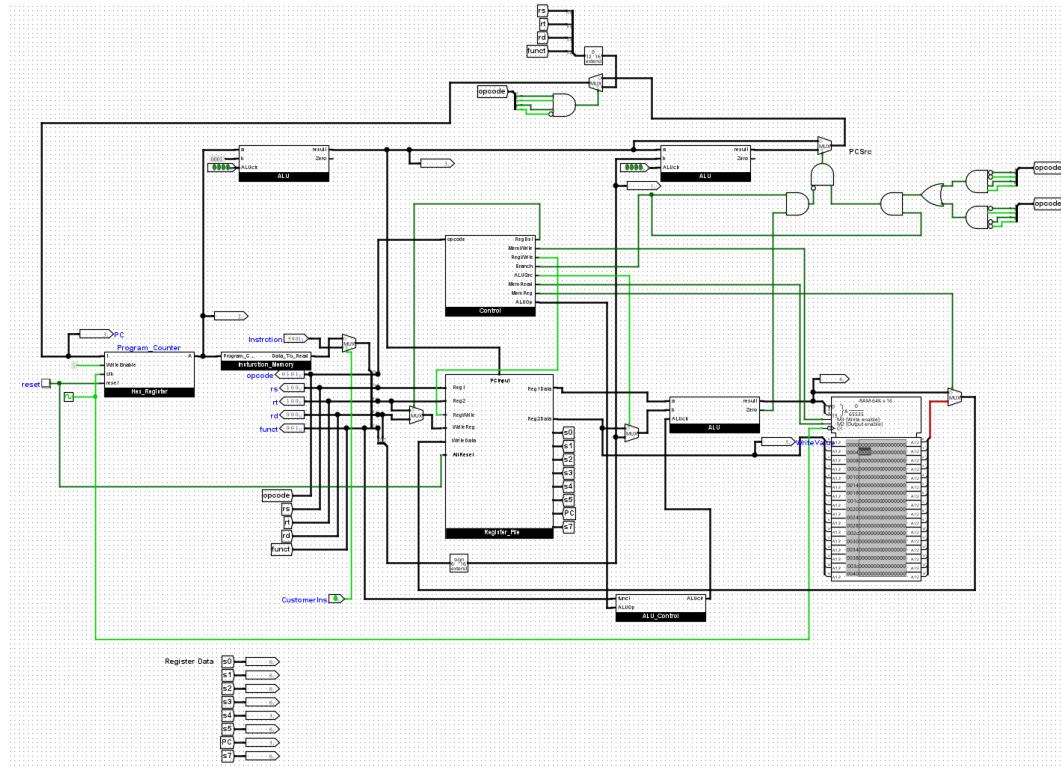


以下為 Block Diagram

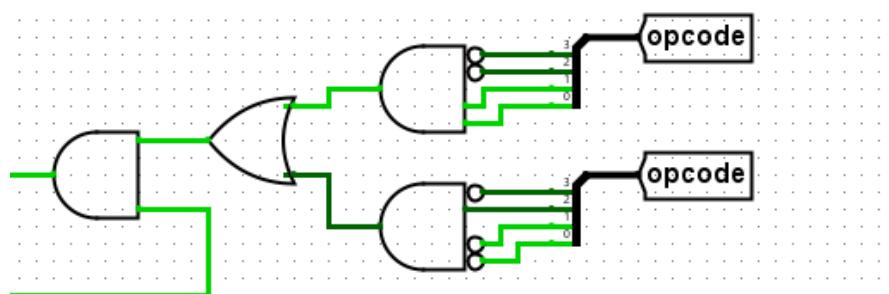


# CPU 設計

有了 Controller 以及所有基本元件，我們可以將所有元件進行串接，完成 CPU 設計



實際執行程式時，我們需要將程式碼放入到 Instruction Memory 中，而我們要對記憶體寫入資料時，我們可以觀察最右邊記憶體的狀況，在圖上可以看到 CustomerIns，如果這個 bit 為 1，表示我們可以自行輸入指令，並觀察 CPU 行為，而非讀取 Instruction Memory 的指令，這是方便測試所設計的功能。在測試過程中，發現 bne 以及 beq 不會正確地進行跳轉，因此在最右邊的部分，設計對應的邏輯控制 Program Counter 的值



判斷是否為 bne 或是 beq，接著看 Controller 的 branch bit 進行 AND，判斷是否為跳轉類型的指令，如果是的話，將 ALU 計算出來的值作為 Program Counter 的值使用。

最上面的邏輯是用於處理 Jump，因為 Jump 指令格式特殊，因此另外單獨處理。

## 程式碼測試與執行

以下為測試程式碼

```
#include <stdio.h>

int F(int input1, int input2) {
    return input1 * input2;
}

int main(void) {
    int A[21];
    int B[21];
    int Y;
    int x;
    int sum;

    A[20] = x + B[20] - 30;
    Y = F(20, 30) - 100;

    for(int x = 0; x < 5; x++) {
        sum = sum + x;
    }
    if(x <= 5) {
        A[x] = x + B[20] - 30;
    }
    else {
        A[x] = 0;
    }
}
```

要執行 C 語言程式碼，我們要先翻譯成組合語言，再將組合語言翻譯成機器語言，這邊可以看到我們有函式存在，因此我們需要維護 stack，以下先看到一段 x86 為架構底下程式碼執行流程，從中了解 stack 分配以及 stack frame 的使用

以下為測試 C 語言程式碼

```

#include <stdlib.h>
#include <stdio.h>

void func(int a, int b)
{
    a + b;
}

int main(void)
{
    int a = 2;
    int b = 3;
    func(a,b);
}

```

使用 gdb 進行反組譯

```

0x000000000000113a <+0>:    endbr64
0x000000000000113e <+4>:    push    rbp
0x000000000000113f <+5>:    mov     rbp,rs
0x0000000000001142 <+8>:    sub    rs,0x10
0x0000000000001146 <+12>:   mov     DWORD PTR [rbp-0x8],0x2
0x000000000000114d <+19>:   mov     DWORD PTR [rbp-0x4],0x3
0x0000000000001154 <+26>:   mov     edx,DWORD PTR [rbp-0x4]
0x0000000000001157 <+29>:   mov     eax,DWORD PTR [rbp-0x8]
0x000000000000115a <+32>:   mov     esi,edx
0x000000000000115c <+34>:   mov     edi,eax
0x000000000000115e <+36>:   call    0x1129 <func>
0x0000000000001163 <+41>:   mov     eax,0x0
0x0000000000001168 <+46>:   leave
0x0000000000001169 <+47>:   ret

```

在函式的開始，有一段程式碼我們稱為 prologue，作用為儲存函式中需要使用到的 stack 空間以及暫存器，而在結束的地方，我們需要將 stack 和暫存器回覆到函式呼叫前的狀態，而這段程式碼我們稱為 epilogue，我們可以試著找到上方組合語言中 prologue 以及 epilogue 的部份。

```

0x0000000000000000113a <+0>:    endbr64
0x0000000000000000113e <+4>:    push    rbp
0x0000000000000000113f <+5>:    mov     rbp,rsp
0x00000000000000001142 <+8>:    sub    rsp,0x10
0x00000000000000001146 <+12>:   mov     DWORD PTR [rbp-0x8],0x2
0x0000000000000000114d <+19>:   mov     DWORD PTR [rbp-0x4],0x3
0x00000000000000001154 <+26>:   mov     edx,DWORD PTR [rbp-0x4]
0x00000000000000001157 <+29>:   mov     eax,DWORD PTR [rbp-0x8]
0x0000000000000000115a <+32>:   mov     esi,edx
0x0000000000000000115c <+34>:   mov     edi,eax
0x0000000000000000115e <+36>:   call    0x1129 <func>
0x00000000000000001163 <+41>:   mov     eax,0x0
0x00000000000000001168 <+46>:   leave
0x00000000000000001169 <+47>:   ret

```

prologue

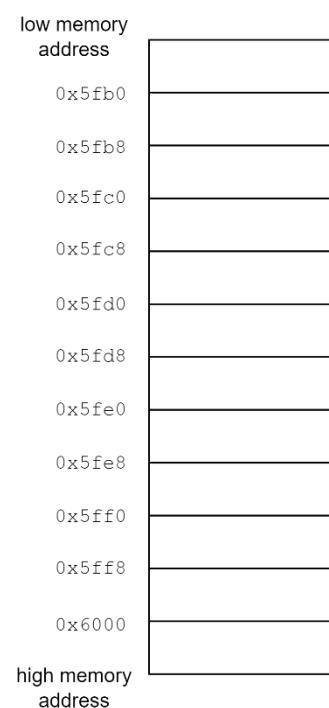
epliogue

接著我們通過實際案例觀察 prologue 和 epliogue 是如何分配 Stack Frame

```

0x0000000000000000113a <+0>:    endbr64
0x0000000000000000113e <+4>:    push    rbp
0x0000000000000000113f <+5>:    mov     rbp,rsp
0x00000000000000001142 <+8>:    sub    rsp,0x10
0x00000000000000001146 <+12>:   mov     DWORD PTR [rbp-0x8],0x2
0x0000000000000000114d <+19>:   mov     DWORD PTR [rbp-0x4],0x3
0x00000000000000001154 <+26>:   mov     edx,DWORD PTR [rbp-0x4]
0x00000000000000001157 <+29>:   mov     eax,DWORD PTR [rbp-0x8]
0x0000000000000000115a <+32>:   mov     esi,edx
0x0000000000000000115c <+34>:   mov     edi,eax
0x0000000000000000115e <+36>:   call    0x1129 <func>
0x00000000000000001163 <+41>:   mov     eax,0x0
0x00000000000000001168 <+46>:   leave
0x00000000000000001169 <+47>:   ret

```



在 prologue 部份，我們將 rbp 的值推入到 stack 中，接著 rsp 的值 -8。



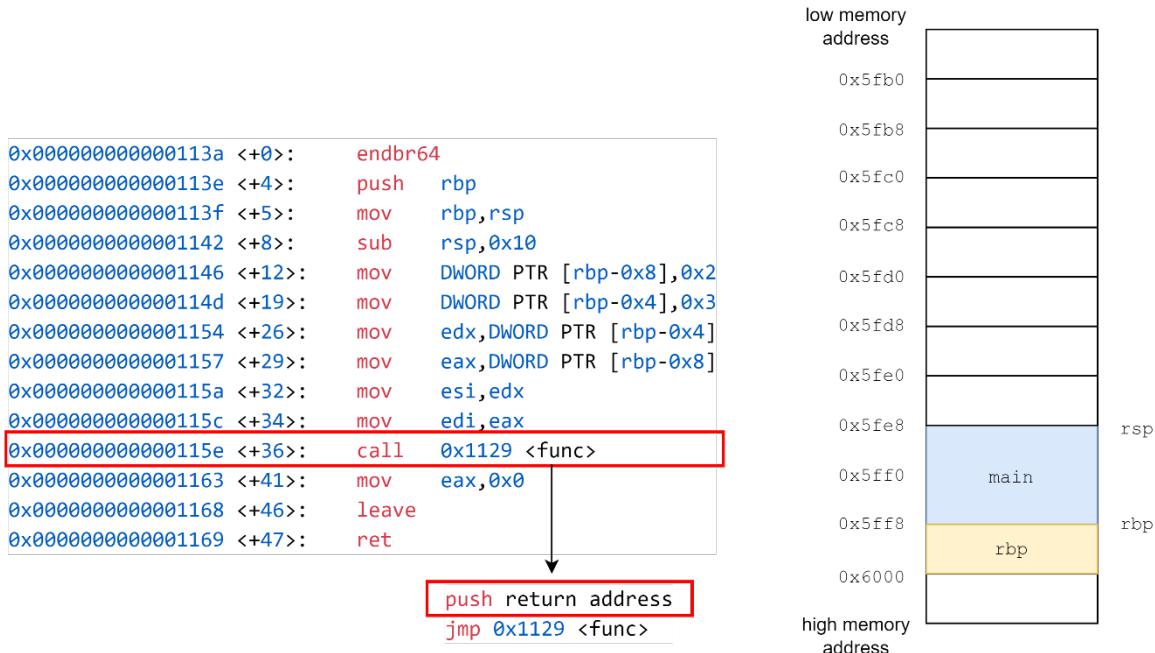
再將 rsp 的值設為 rbp 的值，因此 rbp 的值更新成新的 rsp 的值，舊的 rbp 的值位於 stack 中，如下圖所示



接著通過調整 rsp，開闢出一段給 main 函式使用的 stack 空間，可以在接下來函式主體中使用



經過中間 mov 操作，將函式需要的資料移入 stack，參數放置到暫存器後，接著我們看到 call 0x1129 <func>，call 會執行兩件事情，分別為 push return address，接著 jmp func。





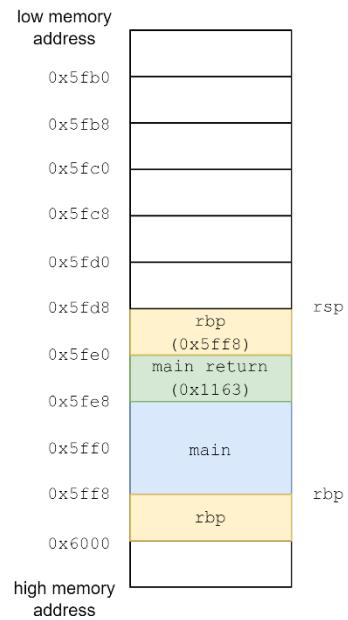
上面完成了 push return address 操作後，我們便 jmp 進入 func

接著我們看到 func 的組合語言程式碼



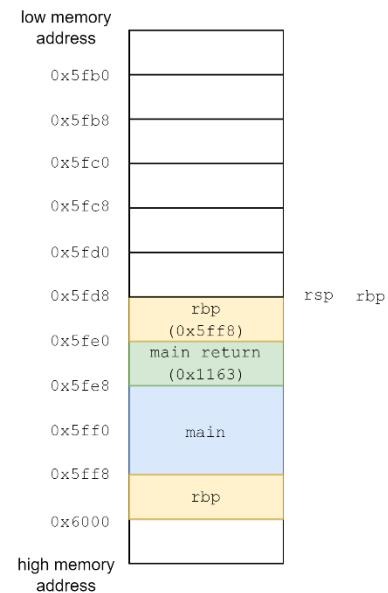
將 rbp 儲存到 stack 中，這個 rbp 原先指向到 main 的 base

```
0x00000000000000001149 <+0>:    endbr64
0x0000000000000000114d <+4>:    push   rbp
0x0000000000000000114e <+5>:    mov    rbp,rsp
0x00000000000000001151 <+8>:    sub    rsp,0x20
0x00000000000000001155 <+12>:   mov    DWORD PTR [rbp-0x14],edi
0x00000000000000001158 <+15>:   mov    DWORD PTR [rbp-0x18],esi
0x0000000000000000115b <+18>:   mov    eax,DWORD PTR [rbp-0x14]
0x0000000000000000115e <+21>:   mov    DWORD PTR [rbp-0x8],eax
0x00000000000000001161 <+24>:   mov    eax,DWORD PTR [rbp-0x18]
0x00000000000000001164 <+27>:   mov    DWORD PTR [rbp-0x4],eax
0x00000000000000001167 <+30>:   mov    edx,DWORD PTR [rbp-0x8]
0x0000000000000000116a <+33>:   mov    eax,DWORD PTR [rbp-0x4]
0x0000000000000000116d <+36>:   add    eax,edx
0x0000000000000000116f <+38>:   mov    esi,eax
0x00000000000000001171 <+40>:   lea    rdi,[rip+0xe8c]      # 0x2004
0x00000000000000001178 <+47>:   mov    eax,0x0
0x0000000000000000117d <+52>:   call   0x1050 <printf@plt>
0x00000000000000001182 <+57>:   nop
0x00000000000000001183 <+58>:   leave
0x00000000000000001184 <+59>:   ret
```

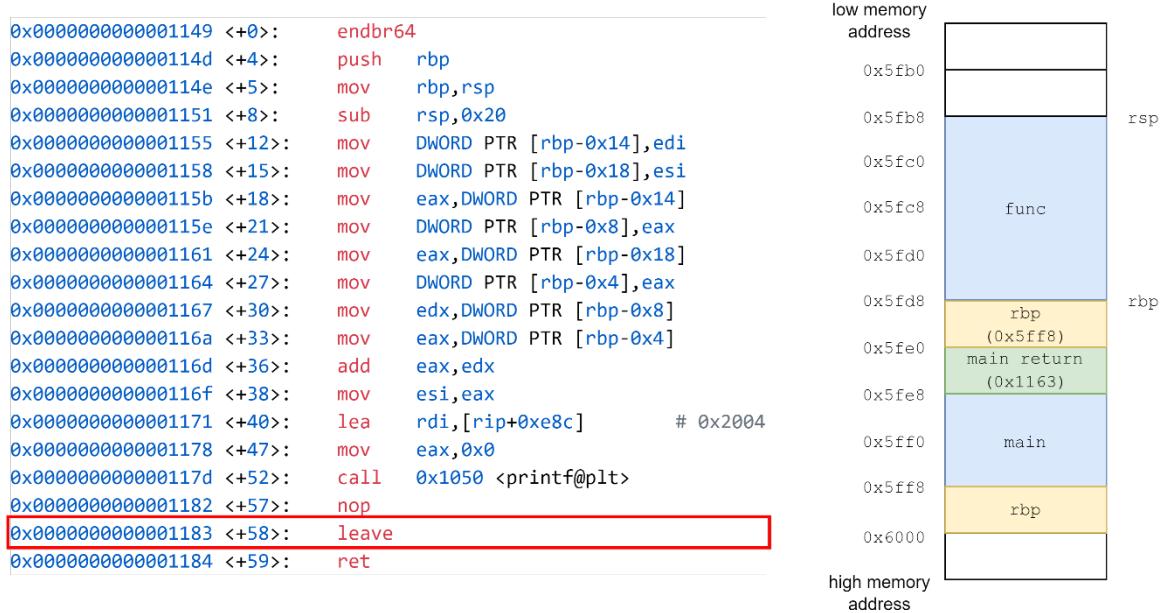


接著將 rsp 的值複製到 rbp 中

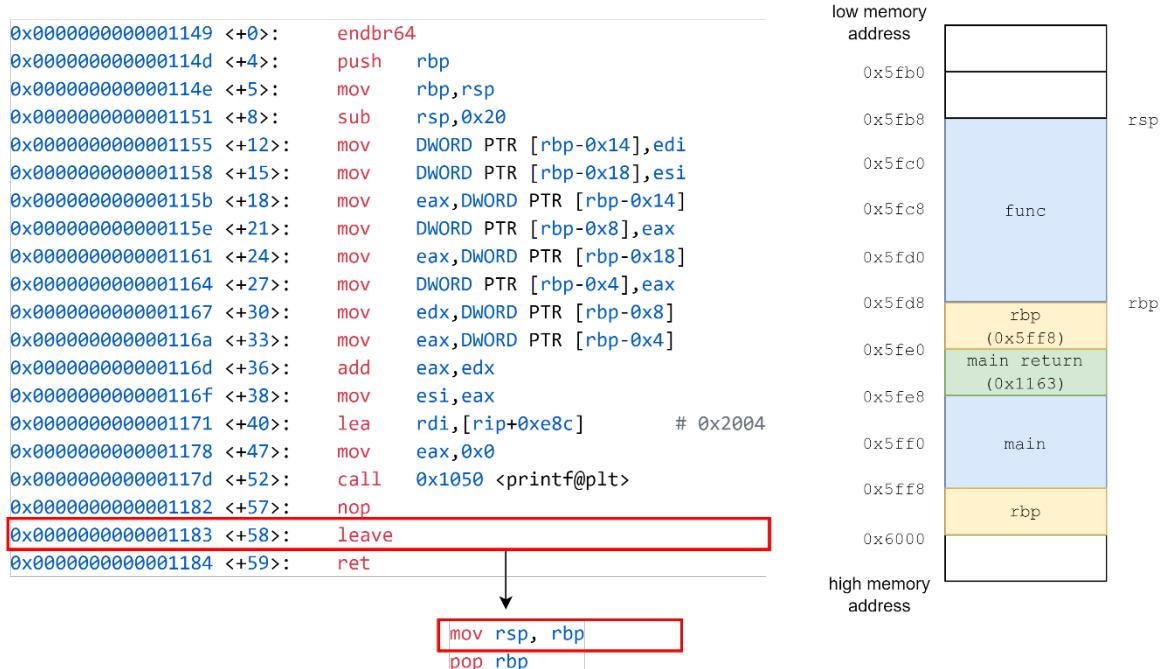
```
0x00000000000000001149 <+0>:    endbr64
0x0000000000000000114d <+4>:    push   rbp
0x0000000000000000114e <+5>:    mov    rbp,rsp
0x00000000000000001151 <+8>:    sub    rsp,0x20
0x00000000000000001155 <+12>:   mov    DWORD PTR [rbp-0x14],edi
0x00000000000000001158 <+15>:   mov    DWORD PTR [rbp-0x18],esi
0x0000000000000000115b <+18>:   mov    eax,DWORD PTR [rbp-0x14]
0x0000000000000000115e <+21>:   mov    DWORD PTR [rbp-0x8],eax
0x00000000000000001161 <+24>:   mov    eax,DWORD PTR [rbp-0x18]
0x00000000000000001164 <+27>:   mov    DWORD PTR [rbp-0x4],eax
0x00000000000000001167 <+30>:   mov    edx,DWORD PTR [rbp-0x8]
0x0000000000000000116a <+33>:   mov    eax,DWORD PTR [rbp-0x4]
0x0000000000000000116d <+36>:   add    eax,edx
0x0000000000000000116f <+38>:   mov    esi,eax
0x00000000000000001171 <+40>:   lea    rdi,[rip+0xe8c]      # 0x2004
0x00000000000000001178 <+47>:   mov    eax,0x0
0x0000000000000000117d <+52>:   call   0x1050 <printf@plt>
0x00000000000000001182 <+57>:   nop
0x00000000000000001183 <+58>:   leave
0x00000000000000001184 <+59>:   ret
```

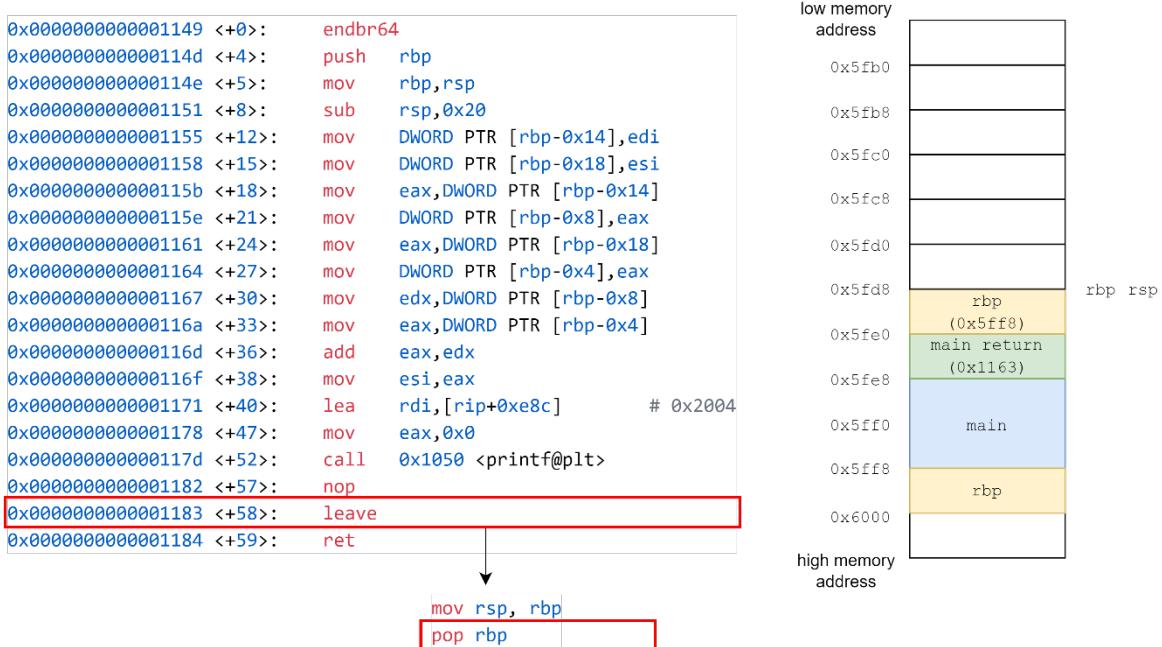


接著通過調整 rsp 的值，在 stack 中開闢一段空間給 func 使用



接著我們要執行 leave，leave 會 mov rsp, rbp，接著 pop rbp。



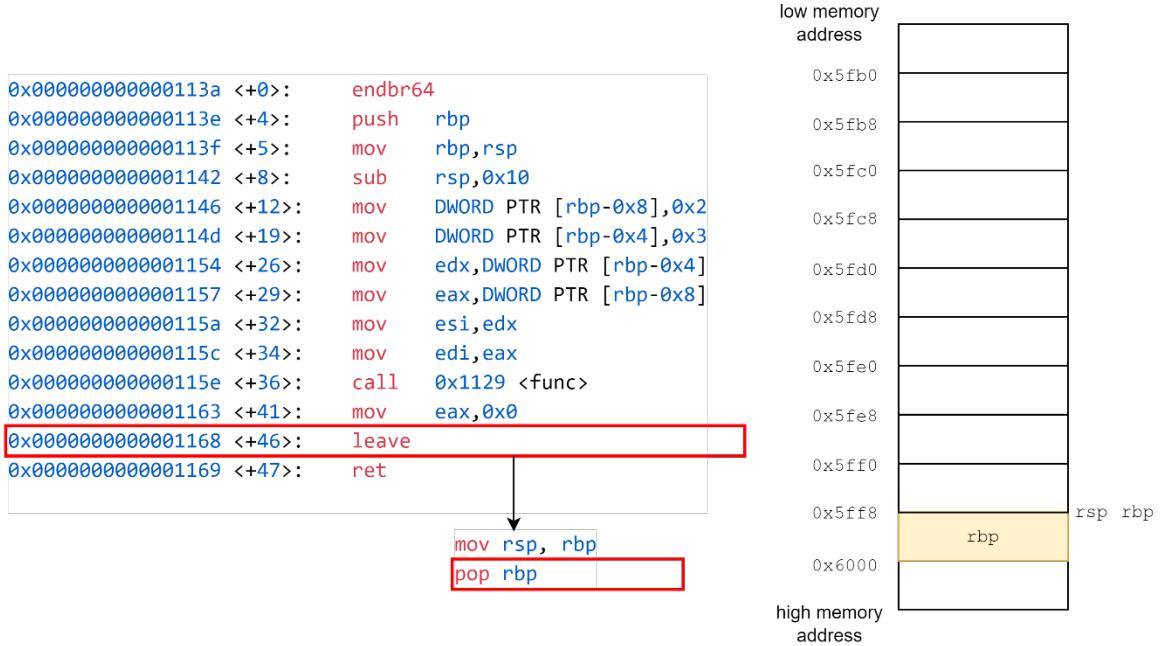


接著是 `ret`，作用為 `pop rip`，這邊對應的意義為將 `return address` 放置到儲存 `instruction pointer` 的暫存器，對應到圖上的意義，就是改變紅色框框的位置，執行完畢後，我們便跳回到 `main` 函式中，紅色框框的位置在 `ret` 時設定完畢。



接著我們跳回了 main 函式，依序執行 leave, ret。

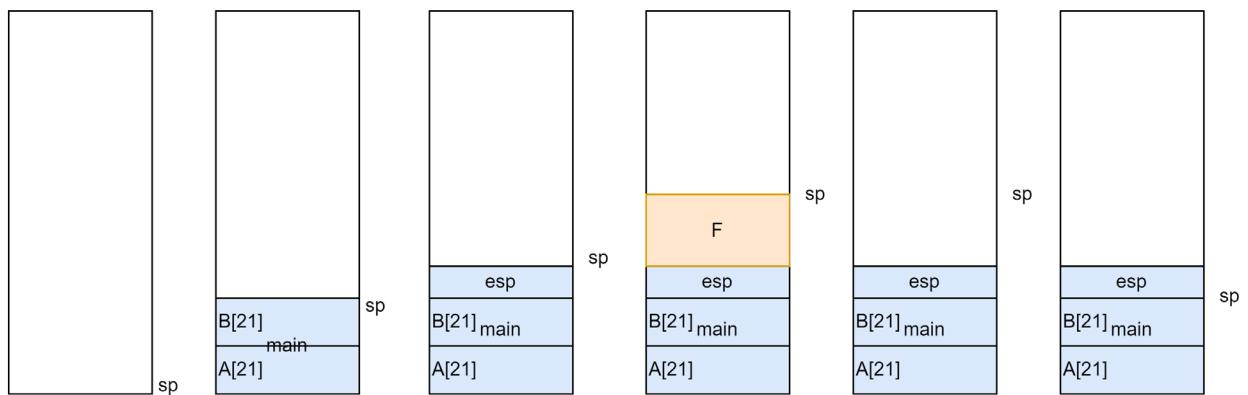




到這裡整個函式呼叫流程結束。

從 x86 觀察 MIPS，我們知道 A 陣列和 B 陣列需要在 stack frame 中佔用空間，接著呼叫 function 的部分，需要保持 return address，才能夠在呼叫完畢 F 之後回到 main 繼續執行。

整個 stack 分布執行流程如下



接著我們將 C 語言翻譯成組合語言與機器語言，這邊需要注意到，由於我們 I type 的指令只能夠接收 -32 到 31 的數值，因此我們如果遇到 +100 等情況，我們需要進行類似以下處理

```
subi $3, $3, 25
subi $3, $3, 25
subi $3, $3, 25
subi $3, $3, 25
```

完成  $\$3 += 100$  的操作，以下為組合語言的翻譯結果

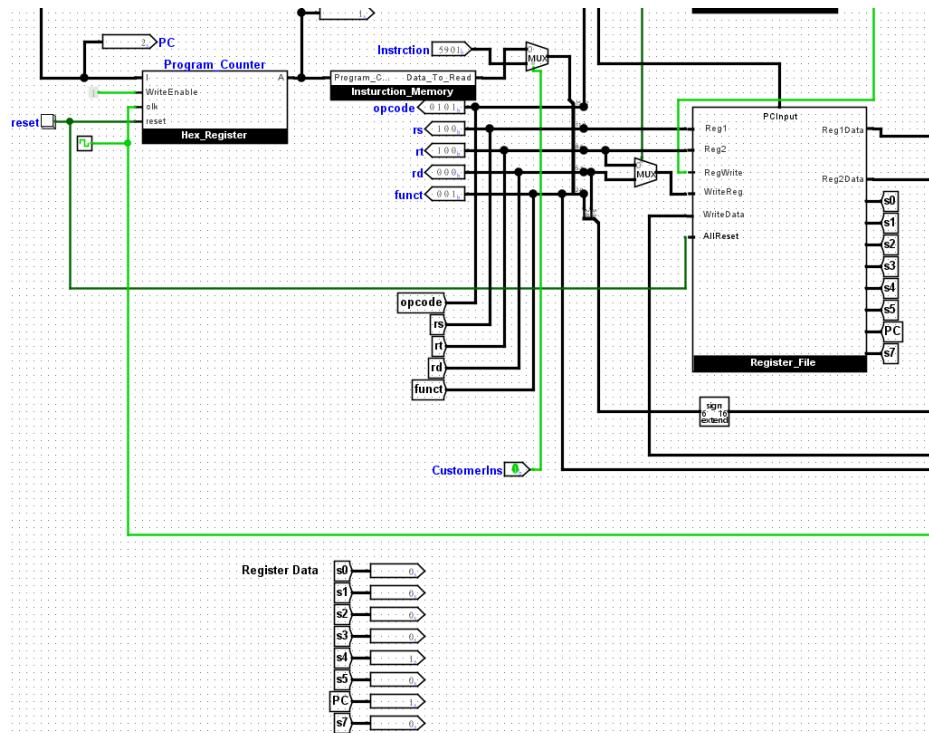
addi \$1, \$0, 21	0101 000 001 010101	5055		
addi \$1, \$1, 21	0101 001 001 010101	5255	sp = 42	stack for B
subi \$2, \$1, 1	0110 001 010 000100	6284	Mem[B[20]]	
lw \$3, 0(\$2)	0001 010 011 000000	14C0	\$3 = B[20]	B[20]
subi \$3, \$3, 30	0110 011 011 011110	66DE	\$3 -= 30	
addi \$1, \$1, 21	0101 001 001 010101	5255		
addi \$1, \$1, 21	0101 001 001 010101	5255	sp = 84	stack for A
\$1 = sp				
subi \$2, \$1, 4	0110 001 010 000100	6284	Mem[A[20]]	
sw \$3, 0(\$2)	0010 010 011 000000	24C0	A[20] = -30	
addi \$2, \$0, 20	0101 000 010 010100	5094		
addi \$3, \$0, 30	0101 000 011 011110	50DE		
sw \$PC, 0(\$1)	0010 001 110 000000	2380	store PC	
j 100	1110 000 001 100100	E064	F in address 100	
mul \$3, \$2, \$3	0000 001 010 010 010	292	Y = F(20,30)	
lw \$PC, 0(\$1)	0001 001 110 000000	1380	load PC	
subi \$3, \$3, 25	0110 011 011 011001	66D9		

subi \$3, \$3, 25	0110 011 011 011001	66D9	
subi \$3, \$3, 25	0110 011 011 011001	66D9	
subi \$3, \$3, 25	0110 011 011 011001	66D9	Y = F(20,30) - 100
addi \$4, \$0, 0	0101 000 100 000000	5100	x = 0
addi \$5, \$0, 5	0101 000 101 000101	5145	
addi \$3, \$0, 0	0101 000 0 000000	5180	sum
add \$3, \$3, \$4	0000 110 100 110 000	0D30	sum = sum + x
addi \$4,\$4, 1	0101 100 100 000001	5901	x += 1
bne \$4, \$5, -3	0011 101 100 111101	3B3D	
bne \$5, \$4, 12	0100 100 101 001100	494C	x == 5
bne \$7, \$0, 11	0100 000 111 001011	41CB	x < 5
subi \$3, \$1, 21	0110 001 011 010101	62D5	
subi \$3, \$3, 21	0110 011 011 010101	66D5	
subi \$3, \$3, 1	0110 011 011 000001	66C1	Mem[B[20]]
lw \$2, 0(\$3)	0001 011 010 000000	1680	B[20]
subi \$3, \$3, 20	0110 011 011 010100	66D4	
subi \$3, \$3, 21	0110 011 011 010101	66D5	Mem[A[x]]
add \$4, \$3, \$4	0000 011 100 100 000	720	x + B[20]
subi \$4, \$4, 30	0110 100 100 011110	691E	x + B[20] - 30
sw \$4, 0(\$3)	0010 011 100 000000	2700	A[x] = x + B[20] - 30
sw \$0, 0(\$3)	0010 011 000 000000	2600	A[x] = 0

將以上指令載入到 Instruction Memory 後，便可以完成程式執行。

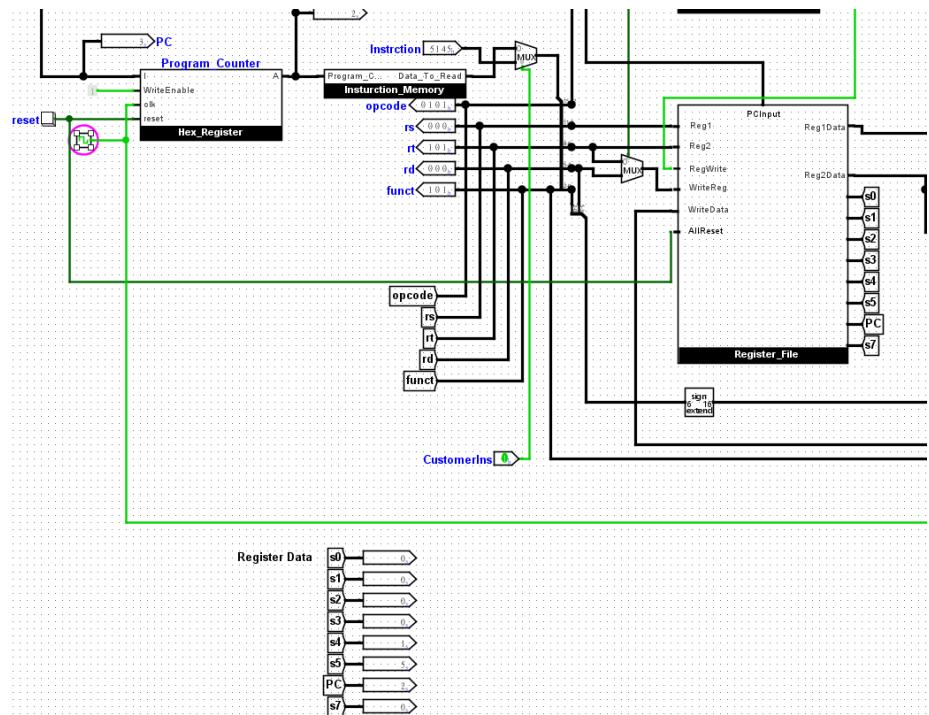
# 執行結果

addi \$4,\$4, 1



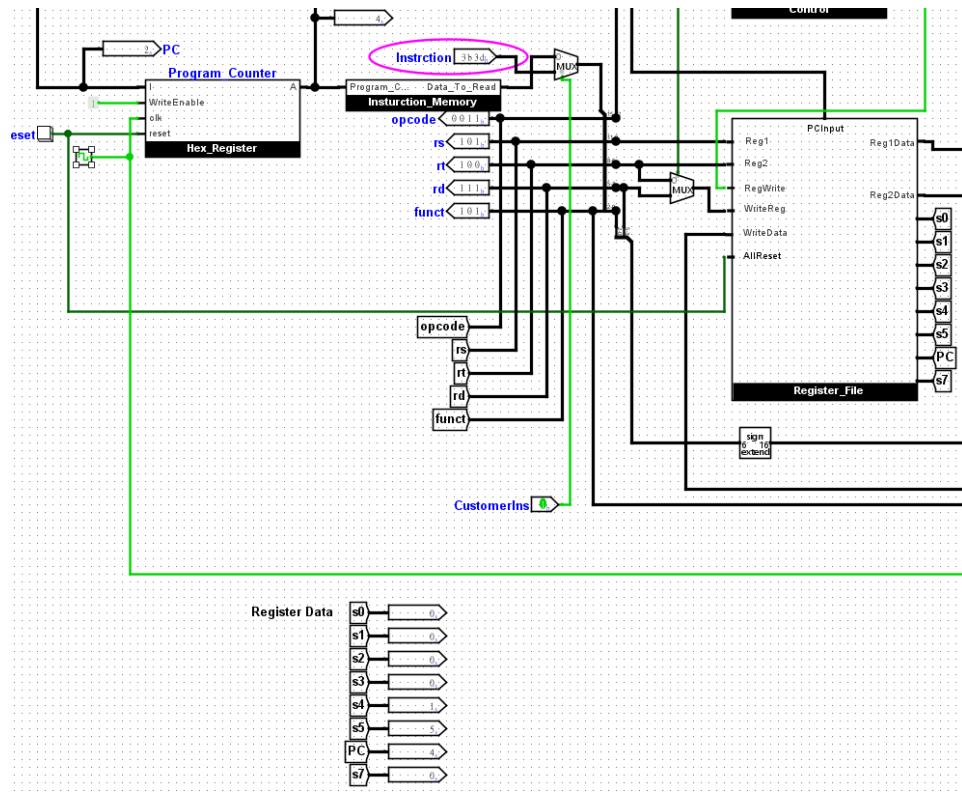
可以從下面 Register Data 看到執行結果正確

addi \$5, \$0, 5



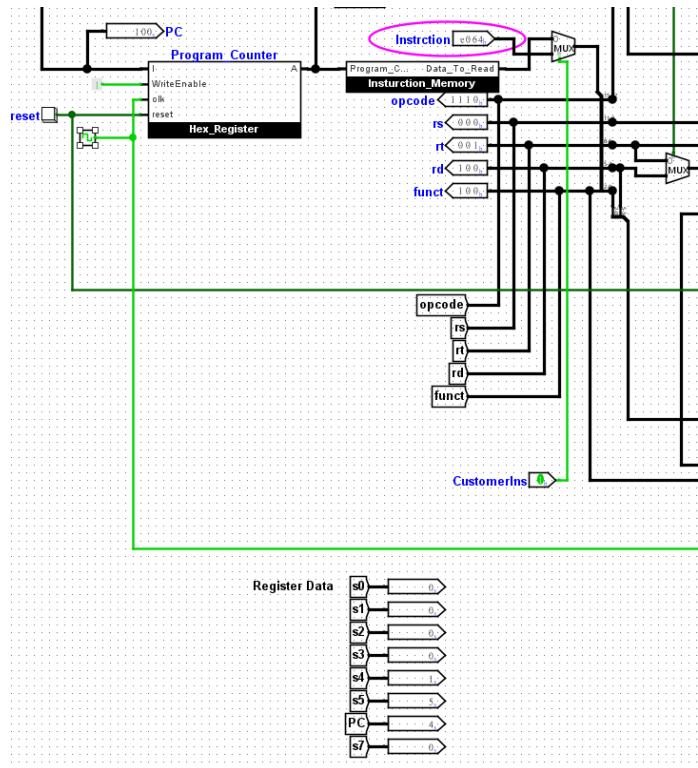
可以從下面 Register Data 看到執行結果正確

bne \$4, \$5, -3



可以看到 PC 的值確實進行了更動

j 100



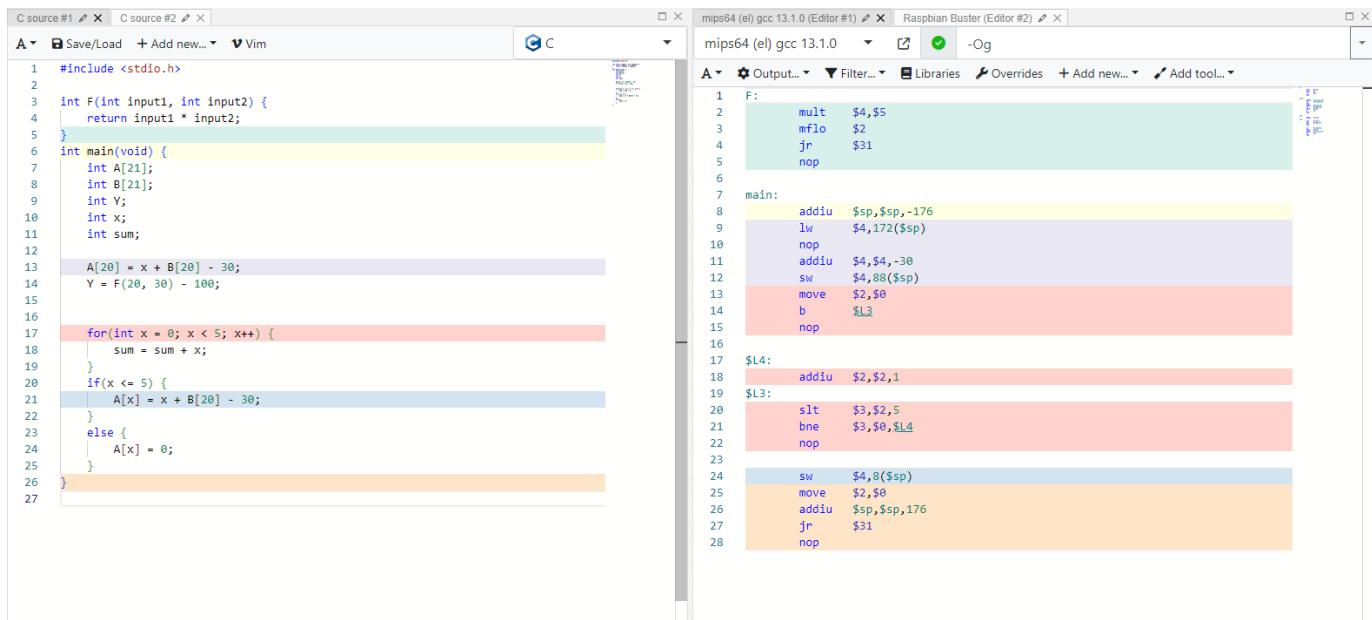
可以看到接下來 PC 的值確實為 100

# 心得

這次 CPU 設計，算是讓我搞懂邏輯設計到整個計算機運作的整個邏輯了，了解到機器語言是如何被 CPU 執行，而在翻譯 C 語言到組合語言的部分，還需要編譯器，組譯器的介入，這邊只是先手動進行翻譯，整個作業如果需要完成理論上需要完成編譯器撰寫，這部分可以在暑假研究一番，有了 CPU，也許可以著手撰寫對應的作業系統，讓這個 CPU 具有更強大的能力，如與其他周圍硬體進行互動等等。

16 bit CPU，讓我了解到硬體資源是如此的珍貴，光是保留出 3 個暫存器，用於 zero, EFLAGS, stack pointer，組合語言翻譯起來就要十分得小心了，而在這一次的實作中，也讓我了解到了為什麼在 32 位元程式的呼叫慣例中，會將函式的參數放到 stack 上，而 64 位元會直接使用暫存器進行處理了，根本原因就是暫存器數量不足以使用。

在翻譯 C 語言程式碼時，我也嘗試使用 MIPS 編譯器進行翻譯，以下為結果



The screenshot shows a terminal window with two tabs. The left tab contains the C source code, and the right tab contains the generated MIPS assembly code.

**C source #1:**

```
1 #include <stdio.h>
2
3 int F(int input1, int input2) {
4     return input1 * input2;
5 }
6 int main(void) {
7     int A[21];
8     int B[21];
9     int Y;
10    int x;
11    int sum;
12
13    A[20] = x + B[20] - 30;
14    Y = F(20, 30) - 100;
15
16
17    for(int x = 0; x < 5; x++) {
18        sum = sum + x;
19    }
20    if(x <= 5) {
21        A[x] = x + B[20] - 30;
22    }
23    else {
24        A[x] = 0;
25    }
26 }
```

**mips64 (el) gcc 13.1.0 (Editor #1):**

```
1 F:
2     mult    $4,$5
3     mflo   $2
4     jr     $31
5     nop
6
7 main:
8     addiu  $sp,$sp,-176
9     lw      $4,172($sp)
10    nop
11    addiu  $4,$4,-30
12    sw      $4,88($sp)
13    move   $2,$0
14    b      $L2
15    nop
16
17 $L4:
18    addiu  $2,$2,1
19 $L3:
20    slt    $3,$2,5
21    bne   $3,$0,$L4
22    nop
23
24    sw      $4,0($sp)
25    move   $2,$0
26    addiu $sp,$sp,176
27    jr     $31
28    nop
```

看到這樣的結果，深感現代編譯器所做的優化真是十分的驚人。