# Lab4: Subroutine & Macro

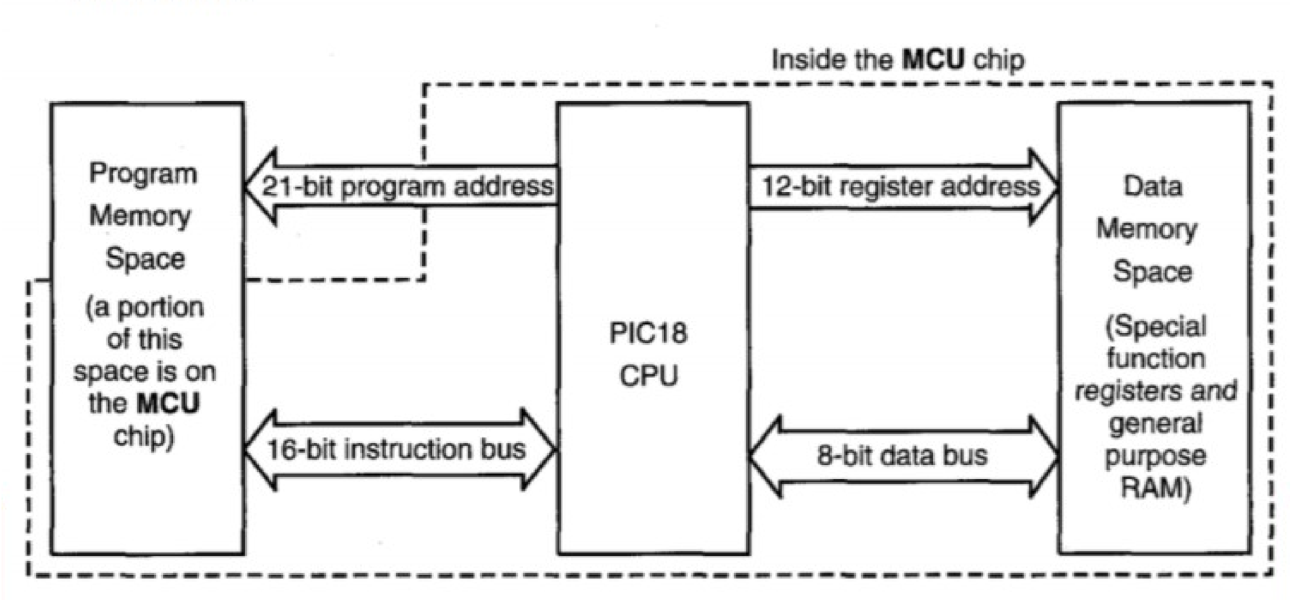
HackMD (https://hackmd.io?utm_source=view-page&utm_medium=logo-nav)

# Lab4: Subroutine & Macro

## 目錄

## 複習PIC18 memory organization

**Program Memory & Data Memory**



Inside the MCU chip
Program Memory Space (a portion of this space is on the MCU chip)
21-bit program address
16-bit instruction bus
PIC18 CPU
12-bit register address
8-bit data bus
Data Memory Space (Special function registers and general purpose RAM)

**Figure 1.3** ■ The PIC18 memory spaces

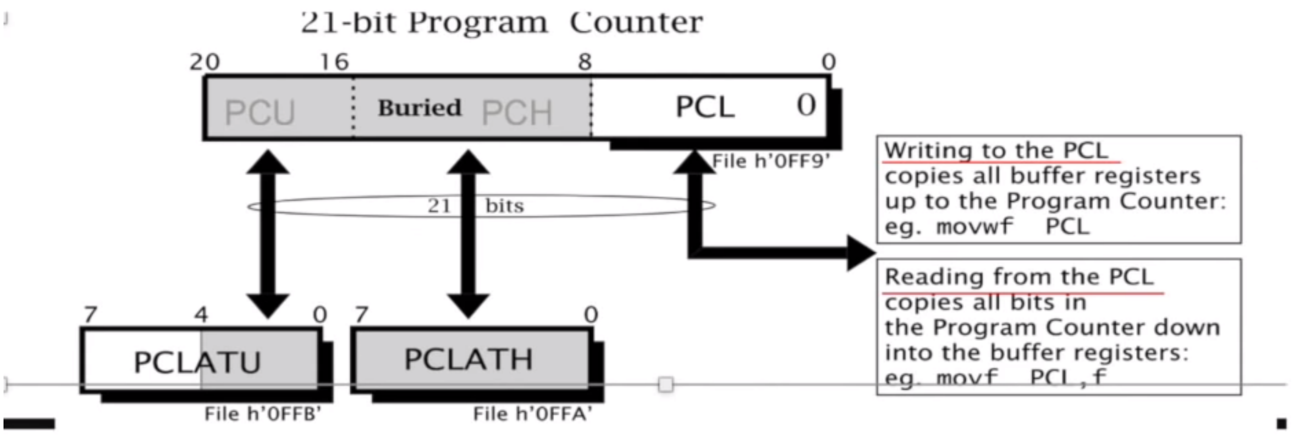
放資料的抽屜分成兩類

- 一種是用來存放Data的 (例：File register、SFR等等)
- 一種是用來放Program的 (例：你撰寫的程式碼)
- **Program Memory**
  - 每個抽屜號碼 **(Program Address)為21-bit** (最多可以有2^21個抽屜)
  - 每個抽屜只能放**8-bit的資料**，但**PIC18的每個指令會用掉16bit的大小**
  - 因此**執行一個指令會用掉2個抽屜**

- Data Memory
  - 抽屜號碼(Data Address)為12-bit
    也就是說共有2^12(4096)個抽屜
  - 裡面可以放8-bit大小的資料

## 介紹相關SFR（PCL、STKPTR、TOS）

### 21-bit program counter


21-bit Program Counter
20
16
8
0
PCU
Buried PCH
PCL 0
File h'0FF9'
21 bits
7 4 0
PCLATU
File h'0FFB'
7 0
PCLATH
File h'0FFA'
Writing to the PCL copies all buffer registers up to the Program Counter: eg. movwf PCL
Reading from the PCL copies all bits in the Program Counter down into the buffer registers: eg. movf PCL , f

- 是用來**存放**下一個要執行之指令的**位置**
- 由三個reg組合而成: PCU、PCH、PCL
  - 我們只能**直接**更改PCL的值
  - PCU和PCH需透過latch來做更改
    - PCU透過PCLATU來更改
    - PCH透過PCLATH來更改
- 可以利用改PCL的值來去指定的地方或拿來寫loop,不一定要用goto
- label那行不算進去

### 為甚麼裡面的值都是二的倍數?

- 因為一個指令**大小**就是16bit
  而PIC18的資料基本單位是8bit
  所以放一個指令需要兩個register~
  (不懂的話可看上面的複習資料)

### Stack(STKPTR、TOS)

- STKPTR: 存stack目前有幾個東西
- TOS: rcall時會儲存下一道指令的位置，Return時會跳過去執行。

# Subroutine

類似於在程式中呼叫一個function

## 使用方法

1. 寫好你要的label
2. 使用rcall label，開始執行subroutine
3. 結束時使用Return，會返回到當時rcall的下一行指令，繼續執行

## 範例

```
initial:
    MOVLW 0x03
    MOVWF 0x00
    CLRF WREG
    rcall func1
    goto finish
func1:
    ADDWF 0x00, W
    DECFSZ 0x00
    GOTO func1
    MOVWF 0x01
    RETURN

finish:
    end
```

# Macro

定義新的指令,可以**給定參數**

```
macro_name macro p1,p2,p3,p4
    ;your macro code
    endm
```

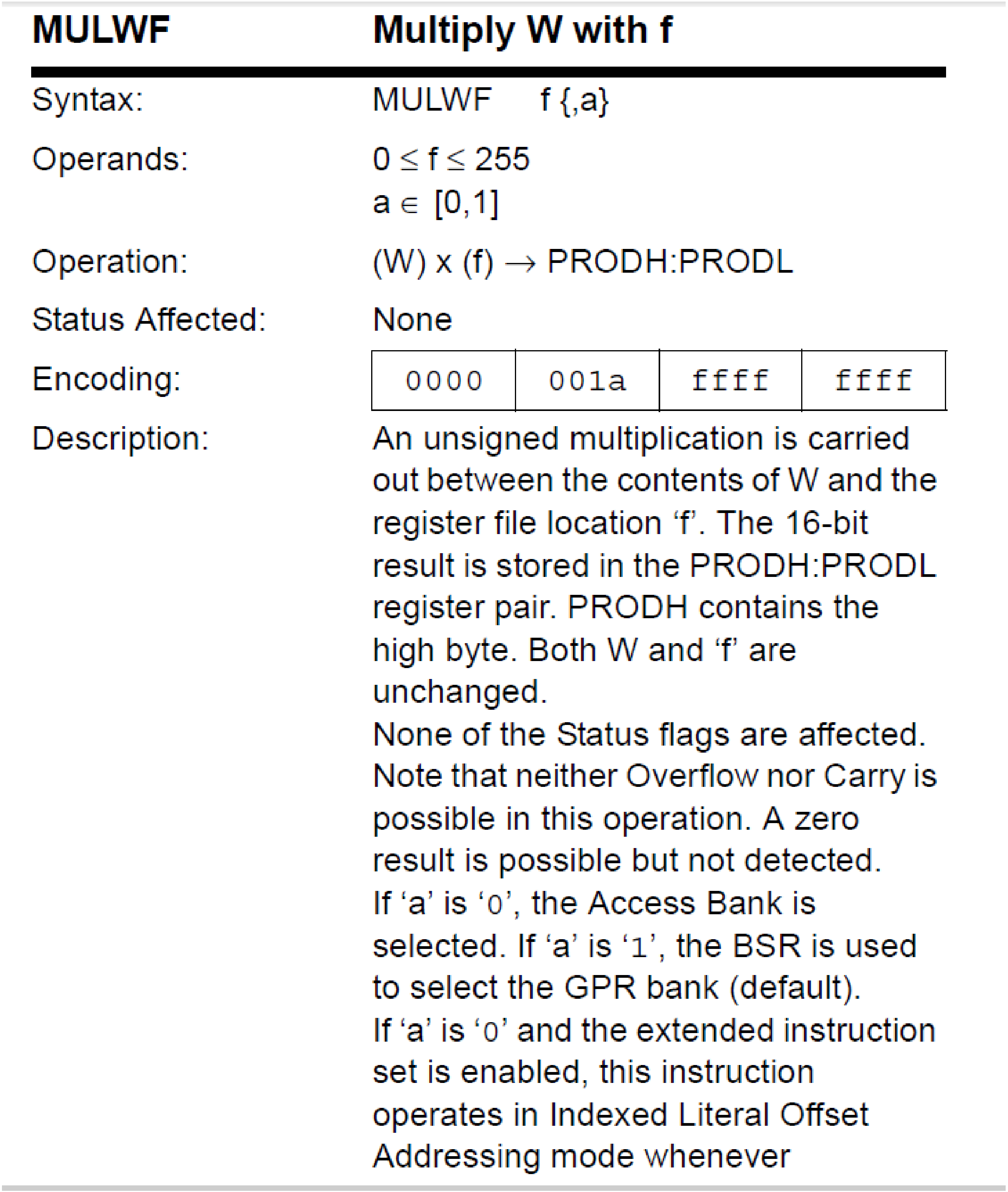
## Macro vs Subroutine

| **macro** | **subroutine** |
|---|---|
| 可直接傳參數 | 需在呼叫前,把參數的值放入register file |
| 碰到macro,compiler會直接複製那段code上去 | 重複使用那段code |
| 適合較短的code | 適合較長但須常用的code,<br>才不會佔空間 |

## Multiply

- **unsigned**

把值都當成正的,不會有負的



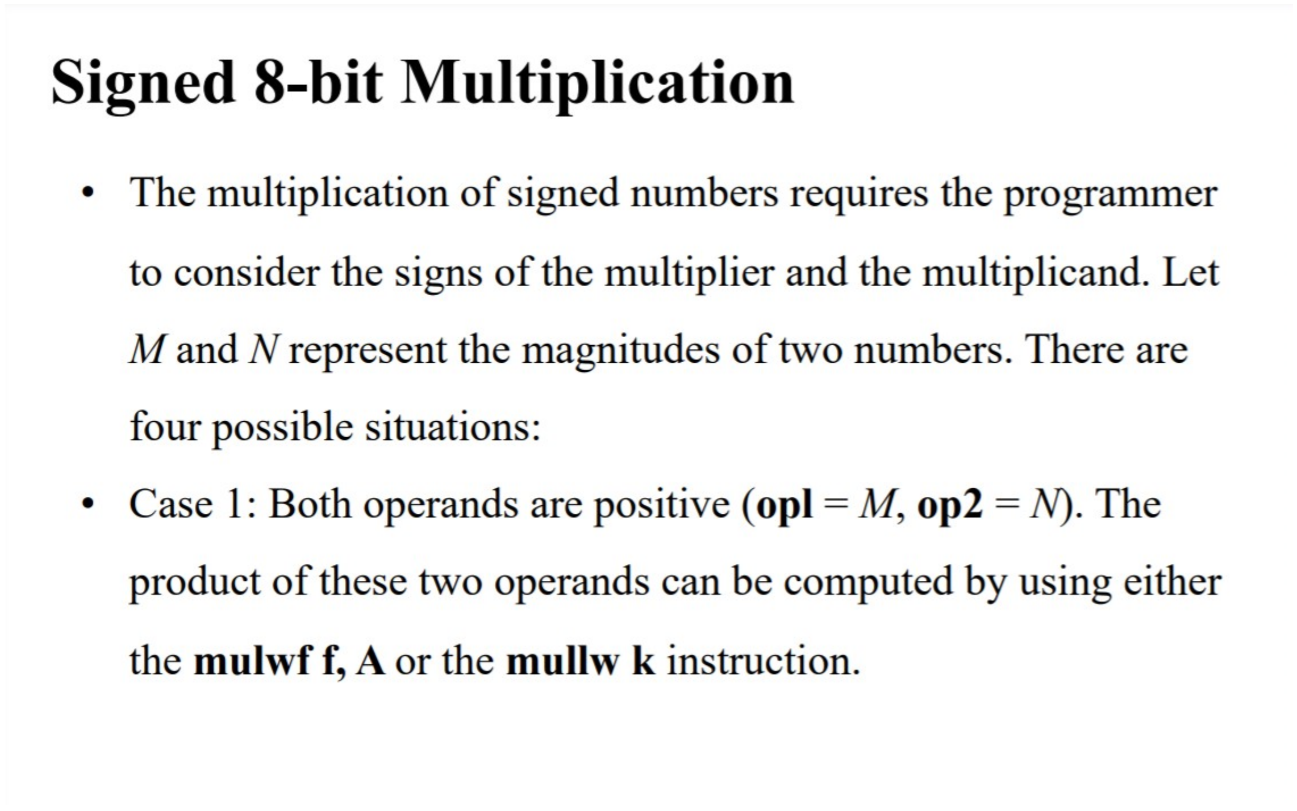
**MULWF** **Multiply W with f**

| | |
|---|---|
| Syntax: | MULWF f {,a} |
| Operands: | 0 ≤ f ≤ 255<br>a ∈ [0,1] |
| Operation: | (W) x (f) → PRODH:PRODL |
| Status Affected: | None |
| Encoding: | `0000` \| `001a` \| `ffff` \| `ffff` |
| Description: | An unsigned multiplication is carried out between the contents of W and the register file location ‘f’. The 16-bit result is stored in the PRODH:PRODL register pair. PRODH contains the high byte. Both W and ‘f’ are unchanged.<br>None of the Status flags are affected.<br>Note that neither Overflow nor Carry is possible in this operation. A zero result is possible but not detected.<br>If ‘a’ is ‘0’, the Access Bank is selected. If ‘a’ is ‘1’, the BSR is used to select the GPR bank (default).<br>If ‘a’ is ‘0’ and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever |

- **signed**

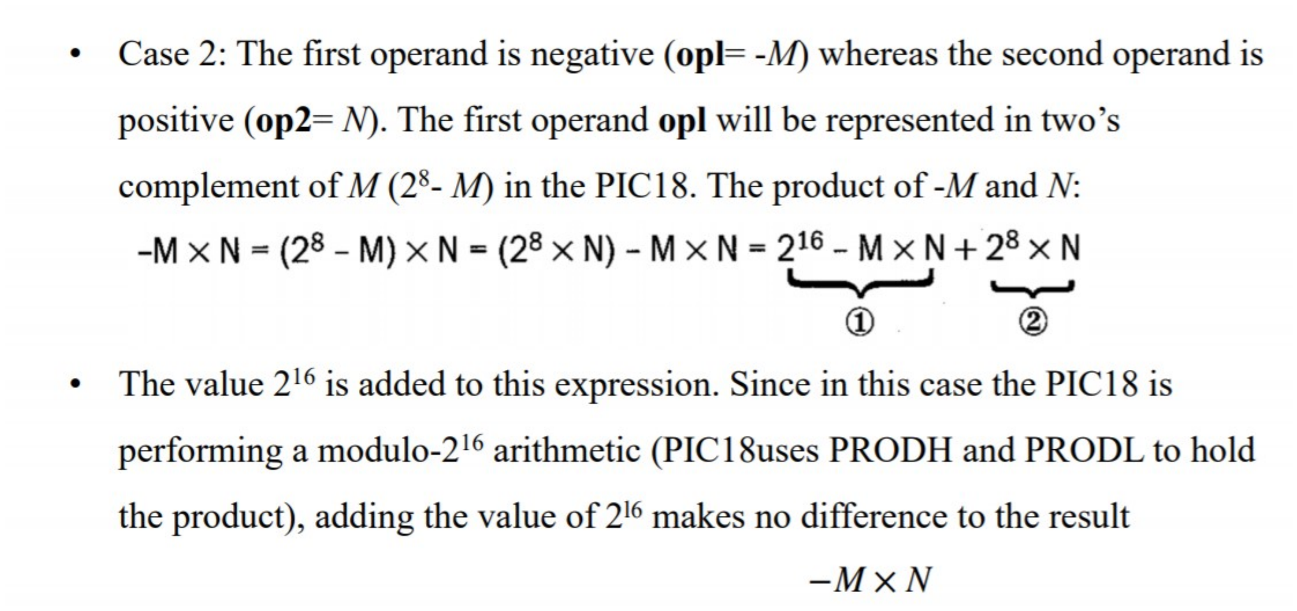
重點: 檢查MSB，並且做出負號的調整

**公式推導**



## Signed 8-bit Multiplication

- The multiplication of signed numbers requires the programmer to consider the signs of the multiplier and the multiplicand. Let $M$ and $N$ represent the magnitudes of two numbers. There are four possible situations:
- Case 1: Both operands are positive (**opl** = $M$, **op2** = $N$). The product of these two operands can be computed by using either the **mulwf f, A** or the **mullw k** instruction.



- Case 2: The first operand is negative (**opl**= -$M$) whereas the second operand is positive (**op2**= $N$). The first operand **opl** will be represented in two's complement of $M$ ($2^8$- $M$) in the PIC18. The product of -$M$ and $N$:

$$-M \times N = (2^8 - M) \times N = (2^8 \times N) - M \times N = \underbrace{2^{16} - M \times N}_{①} + \underbrace{2^8 \times N}_{②}$$

- The value $2^{16}$ is added to this expression. Since in this case the PIC18 is performing a modulo-$2^{16}$ arithmetic (PIC18uses PRODH and PRODL to hold the product), adding the value of $2^{16}$ makes no difference to the result

$$-M \times N$$

- Case 3: The first operand is positive (**opl**= *M*), but the second operand is negative (**op2**=-*N*). Similar to Case 2, the product of *M* and -*N*:

$$M \times (-N) = M \times (2^8 - N) = (2^8 \times M) - M \times N = \underbrace{2^{16} - M \times N}_{①} + \underbrace{2^8 \times M}_{②}$$

- The first term is the correct product, which is represented as the two's complement of $-M \times N$. The second term of this product is an extra term and can be eliminated by subtracting **opl** from the upper byte of the product of *M* and -*N*.

- Case 4: Both operands are negative (**opl**= -*M*, **op2** = -*N*). The product of -*M* and -*N*:

$$\begin{aligned}(-M) \times (-N) &= (2^8 - M) \times (2^8 - N) = 2^{16} - 2^8 \times M - 2^8 \times N + M \times N \\ &= M \times N + 2^{16} - 2^8 \times M + 2^{16} - 2^8 \times N \\ &= \underbrace{M \times N}_{①} + \underbrace{2^8 \times (2^8 - M)}_{②} + \underbrace{2^8 \times (2^8 - N)}_{③}\end{aligned}$$

- The first term is the product of -*M* and -*N*. The second term and the third term are extra terms and must be eliminated. The second term can be eliminated by subtracting **opl** from the upper byte of the product, whereas the third term can be eliminated by subtracting **op2** from the upper byte of the product.

## 實際步驟

- The signed 8-bitmultiplication can be implemented by the algorithm

**Step 1**
Multiply two operands (i.e., compute **op1** × **op2**) disregarding their signs.

**Step 2**
If **op1** is negative, then subtract **op2** from the upper byte of the product.

**Step 3**
If **op2** is negative, then subtract **op1** from the upper byte of the product.