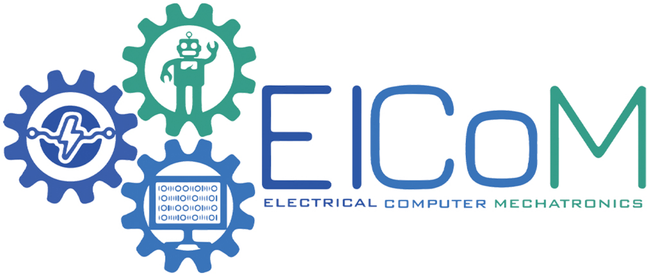



تقدم لجنة ElCoM الاكاديمية

دفتر لمادة:

# معالجات و متحكمات دقيقة

جزيل الشكر للطالبة:

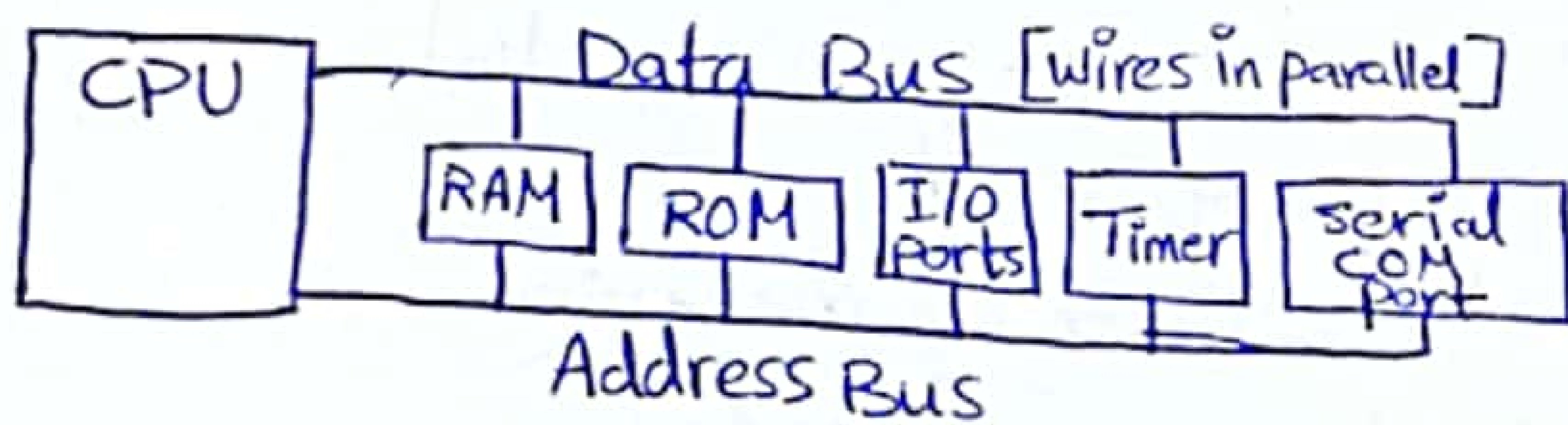
## سارة آبو سارة

# PIC microcontrollers and embedded systems

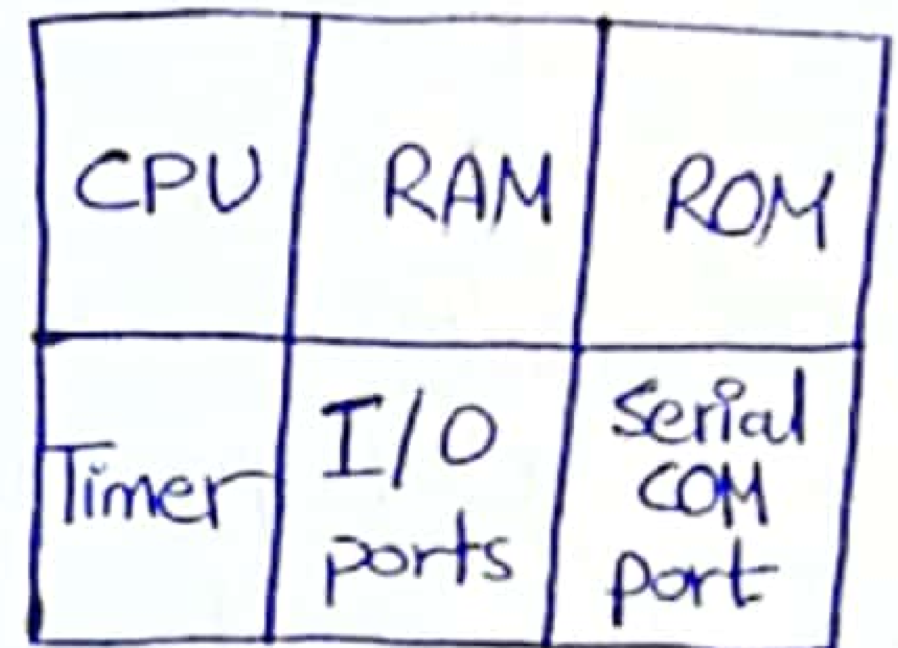
## Ch.1 The PIC microcontrollers :

### 1.1 micro controller and embedded processors →

- Embedded systems (الأنظمة المدمجة) : it means that the application and processor are combined into a single system. It's called a (dedicated system) (نظام مخصص "محدد") because it's dedicated to doing one type of job.
- Microcontroller (uc) VS microprocessor (up) :




a) General purpose Microprocessor system
- it has no (RAM, ROM, ....) [many chips]
- for example (Intel x86 , Pentium)




b) Microcontroller [embedded system]
- for home uses
- it's an one chip has a CPU with (RAM, ROM, ....) are embedded together.
- doesn't use in the high processing
ما بتحتاج للتطوير بعد كل فترة لأنو استخداماتها محددة وسكنية وروتينية :P

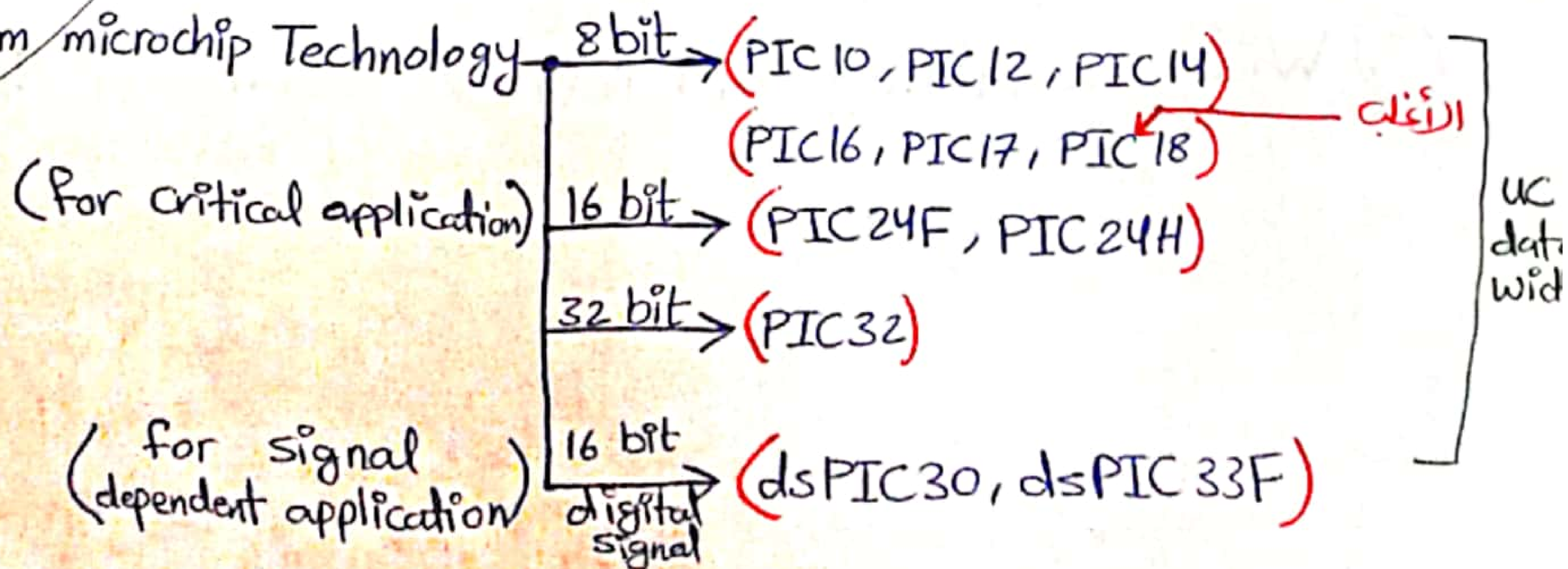
* [MC] based system is cheaper than general purpose [mp]

- Choosing a ucontroller ⇒

# The 5 major 8 bits [8 wires at databus]

ما بيتوافقوا مع بعض و البرامج الي بيشتغل على واحد منهم ما بيشتغل على الآخر

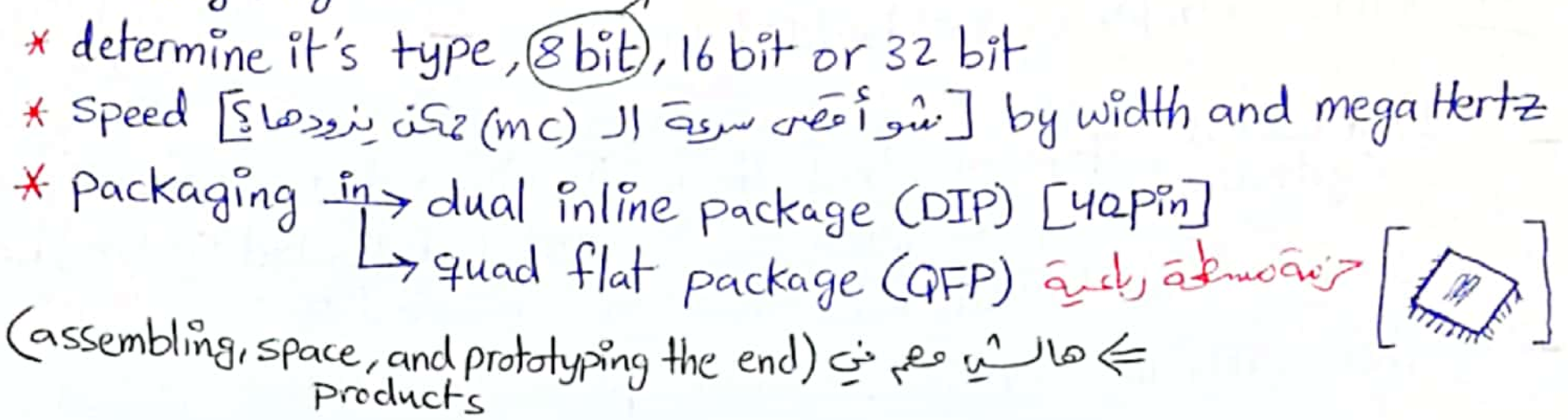
1. Freescale semiconductor's (formerly Motorola) [68HC08/68HC11]
2. Intel's 8051 → [887 for 16 bits]
3. Zilog's Z8
4. Atmel's AVR
5. PIC from microchip Technology




أمثلة على متحكمات دقيقة (8-bits)

# Criteria for choosing ucontroller ⇒
معايير

① Meeting the computing needs of the task at hand efficiency and cost effectively, by: منصيش على الأغلب بيغطي الحاجة

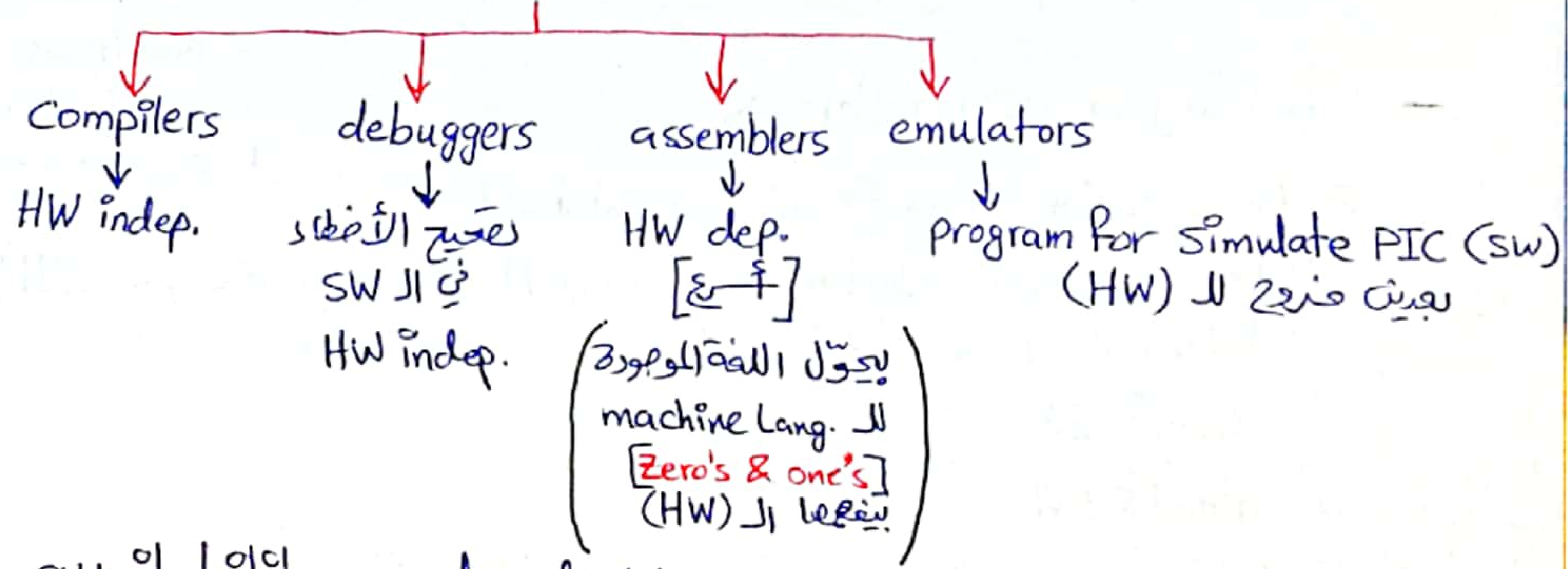


* determine it's type, (8 bit), 16 bit or 32 bit
* Speed [شو أقصى سرعة ال (mc) ممكن يزودها؟] by width and mega Hertz
* Packaging in → dual inline package (DIP) [40pin]
  → quad flat package (QFP) حزمة مسطحة رباعية

(assembling, space, and prototyping the end products) ⇐ هالشي مهم ني

* power consumption for battery powered products.
* The amount of RAM and ROM on the chip.
* The no. of I/O pins and the timer on the chip.
* Cost per unit, for production process needs.
* Ease of upgrade, to a higher performance or lower power consumption for design process needs.

② Availability of software & hardware development tools
[How easy mc is to develop products around it?]




③ Wide availability and reliable Sources of the ucontroller.

→ [توفر أكثر من مصدر للـ (mc) بيخلي المستخدم مو رهين لقطعة معينة فبقدر يحصل على السعر المناسب والأرخص بالنهاية لتوفر عدة أنواع ومصادر إلها...]

Scanned with CamScanner

## 1.2 Overview of the PIC18 family :

- An [8 bit] mc called PIC is introduces 1989 by Microchip Technology Corporation
- It includes "in one chip"
  - → small data RAM [for temporary data], بحيث بس و تشيل الكهربا بتروح
  - → Few bytes of ROM [for permanent data], It's a code program (SW)
  - → One timer
  - → I/O ports

PIC18 has (26) Chips

- PIC18 Features :

① RISC Architecture [Reduced Instruction Set Computer]
In : Cellular phones and tablet computers.

② On chip code ROM and Data RAM, Data EEPROM [Electrically Erasable Programmable Read Only Memory] "non volatile memory"

* EEPROM used to store relatively small amounts of data but allowing individual bytes to be erased and reprogrammed.

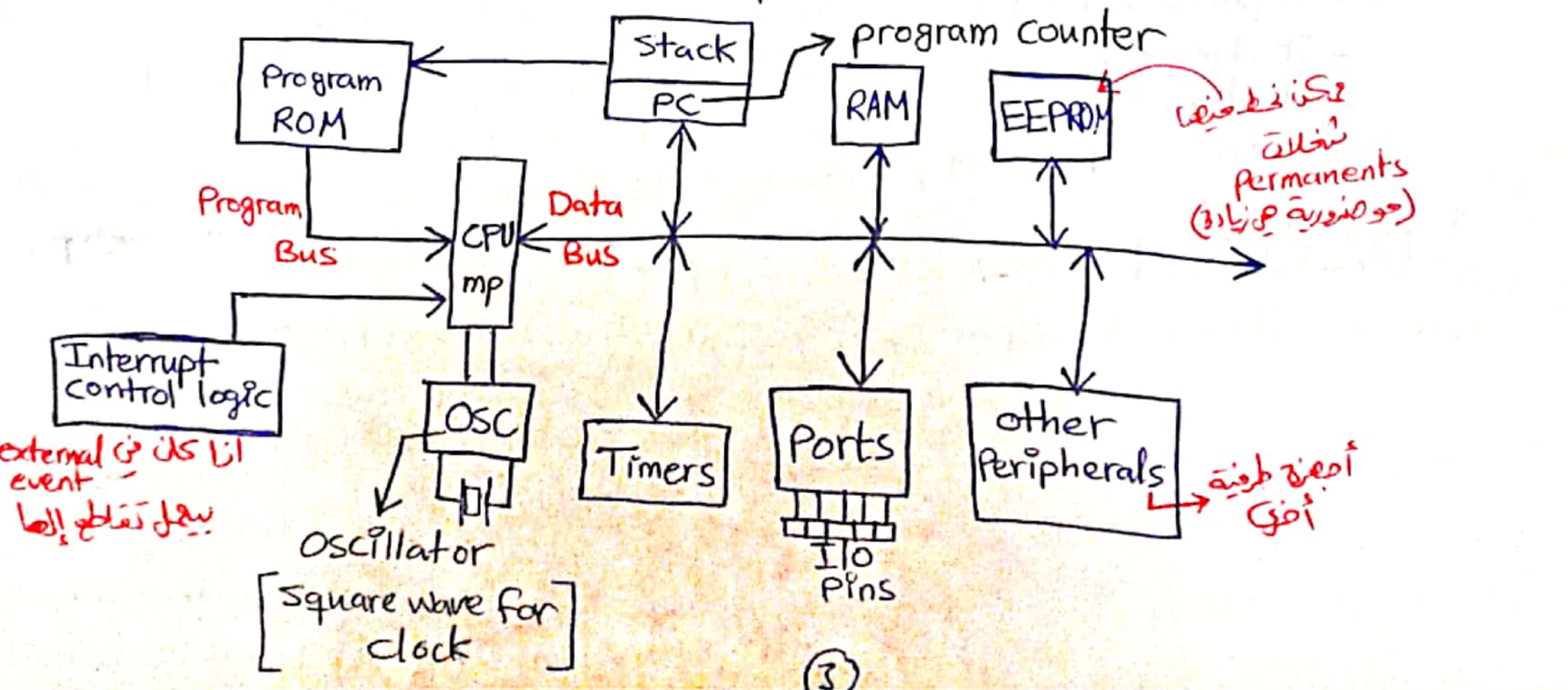
③ Timers

④ ADC

⑤ USART [Universal Synchronous and Asynchronous receiver-transmitter]
device that can be programmed to communicate sync. and async. (for serial data transfer)
→ intel
→ Zilog

ملاحظة لا يحتوي ال (mc) على DAC ؟
لأنه بيتم استبدالها بال [Pulse with modulation] اللي بيغير ال width تبعت السيغنال (بيغير ال Power density)
[because it's replaced by PWM that changes in the signal width, thus the power density]

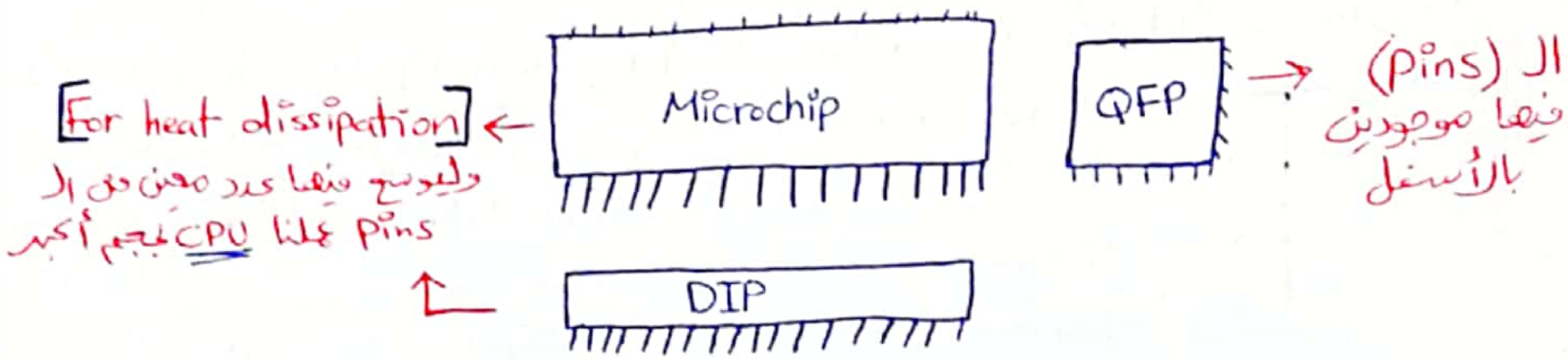
# Simplified view of a PIC the mc :






Scanned with CamScanner

⇒ Microcontroller has lots of forms depending on its features :-




# PIC mc program ROM ⇒

- PIC exists in terms of different speed and the amount of on-chip RAM/ROM
- In ucontroller, the ROM is used to store programs so it's called [program] or [code] ROM
- PIC18 has (can support up to) 2 megabytes of program (code) ROM space
- The program ROM size can vary from [4K–128K] depending on the family member.
- The PIC18 program ROM is available in different memory types such as :-

1) Flash [PIC18FXXX], used for product development (لأنها قادرة على مسح الذاكرة بثواني فقط وبالتالي) غالية

(It's eliminate wasting time needed to erase the chip thus speeding up the development time)

* To use the PIC18F to develop a mc based system :-
  - requires a ROM burner that supports flash memory.
  - it doesn't require a ROM eraser because flash is an EEPROM
  - it doesn't require separate erasable to program it again needs because the ROM programmer itself erase the entire contents of ROM

2) UV_EPROM [ultraviolet EPROM] →

To use it for development, it requires ① access to a PROM burner
② UV-EPROM eraser to erase the contents of ROM
[لكنها تقوم بعملية المسح خلال (20 min.) مقارنة مع الفلاش السريعة جدا]
by the window of it that allows the UV-light to erase the ROM



Scanned with CamScanner

3] One Time programmable [OTP] → [PIC18 C XXX] (indicates the OTP ROM)

- Cheaper than flash
- used for mass production

مشكلتها: You can't reprogram it if you want to modify your program

4] Masked

(العيب/)
- It's the cheapest of all types, if the unit no.'s are high enough because there's a minimum order for the masked version of the PIC uc.
- Program will be burned into the PIC chip during the fabrication process (عملية التصنيع)

---

# PIC mc data RAM and EEPROM ⇒

[to store program (code) ← ROM حكينا قبل بإنو ال]

RAM:
- The RAM space is used for (data storage).
- The PIC18 has a max. of 4096 bytes (4K) of data RAM space

  1K = 1024 bytes

- The data RAM space has two components : "file register"
  1. GPR (general purpose RAM) → [varied from chip to chip]
     - it's used for read/write scratch pad and data manipulation
     - it divided into banks of 256 bytes each
  2. SFR (Special Function Registers) → (موجود بكل mc)
     - they are fixed and every mc must have them.

- Some of PIC's have a small amount of EEPROM [ROM زي ال] that used for critical data storing.

---

## * Abbreviations * اختصارات [from PIC18 family table]

PWM ≡ Pulse With Modulation
CCP ≡ Capture/compare/PWM
SPI ≡ Serial peripheral Interface
$I^2C$ ≡ Inter-Integrated Circuit
USB ≡ Universal Serial Bus
LIN ≡ Local Interconnect Network
CAN ≡ Controller Area Network

PMM ≡ Power managed Mode

* other peripherals :
  + ADC
  + serial or ethernet interface
  + USART



Scanned with CamScanner

# PIC microcontrollers and embedded systems

## Ch.2 PIC Architecture and Assembly Language Programming

- CPUs use many registers to store information temporarily.
  That information could be → a byte of data to be processed.
  → an address pointing to the data to be fetched. → ليتم جلبها

  To program in Assembly language, we must understand the registers and architecture of a given CPU and the role they play in processing data ⇒ Reg. in mp

- PIC mc have many registers for arithmetic and logic operations
  such as → WREG reg. "the widely used"
  → PIC file reg. → General-purpose reg. (GPR)
  → Special function reg. (SFR) الموجودين بالـ RAM

  ⇒ Reg. in mc

[ recall) Registers: هي ذاكرة سريعة تستخدم لتخزين المعلومات
في المعالج بحيث يمكن القراءة والكتابة
منه لذلك سميت بالسجلات. → خاصية الإزاحة
→ خاصية الذاكرة

أنواعها ( Parallel in / Parallel out
Serial in / serial out ) ]

---

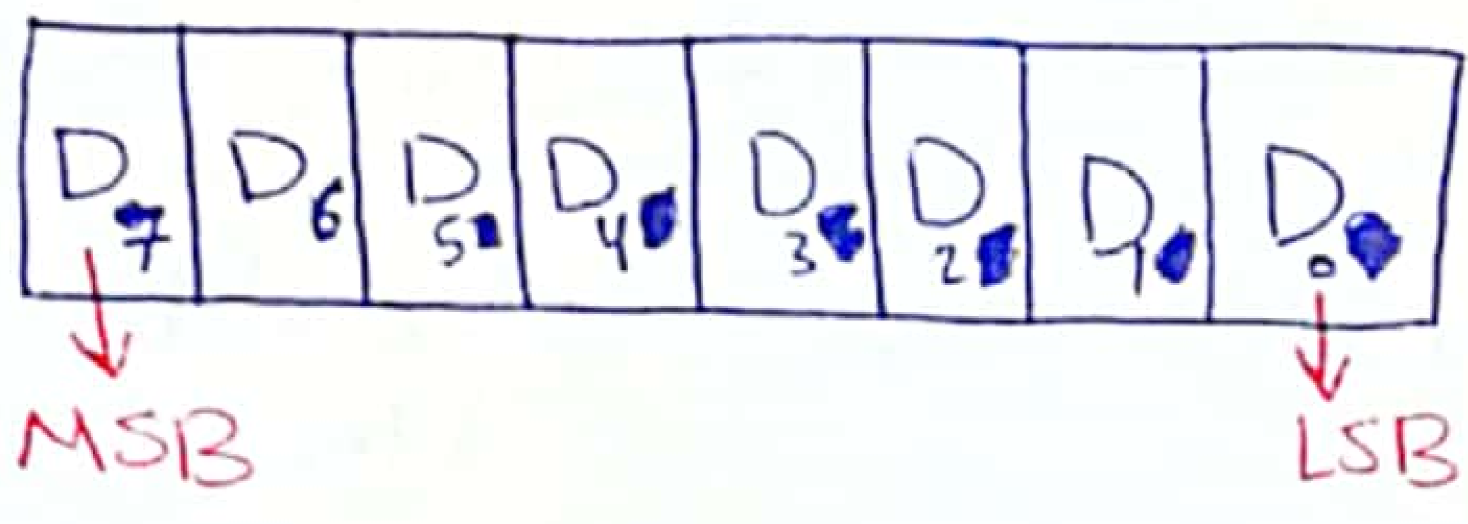
Note / Intel has 4 registers

Scanned with CamScanner

## 2.1 The WREG register in the PIC (Working Register)

[The vast majority (الغالبية العظمى) of PIC reg. are 8 bit reg.]

* In the PIC there is only one data type : 8 bit → (بمعنى أن) any data Larger than 8 bits must be broken (يتقطعوا) into 8 bits chunks before it's processed




* The WREG reg. is the same as the accumulator (مُجمّع) in other microprocessors

# WREG reg. instructions :

① **MOVE**

- MOVLW [كلمة محجوزة ازا أغلطت بكتابتها بيعطيك Syntax Error]

* it moves 8 bit data into WREG

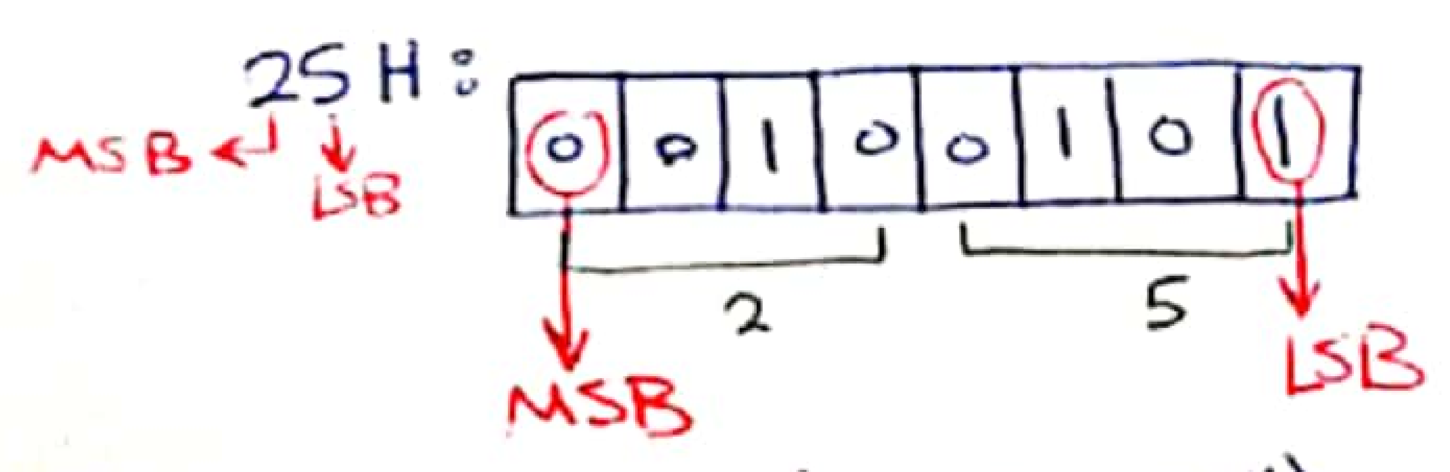
| MOVLW K / move literal value k into WREG |
|---|

K is an 8 bit value that can range from (0-255) in decimal.
(00-FF) in hexadecimal.

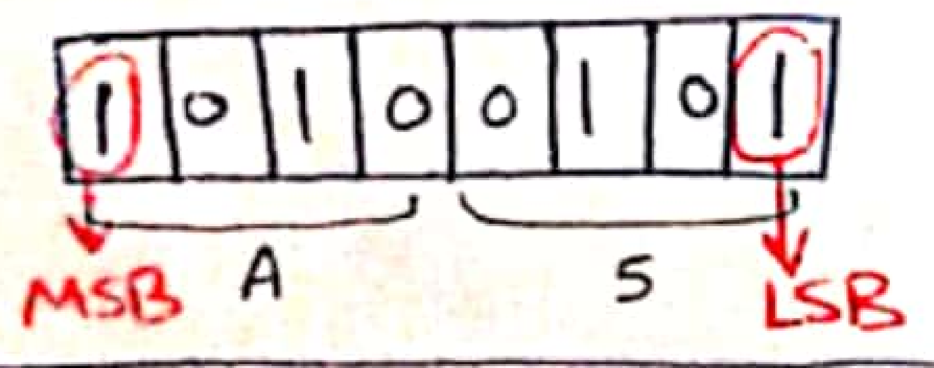
**Ex**

* MOVLW 25H (معناها) → move literial value 25H into WREG (WREG=25H)

25H :




* MOVLW A5H (WREG = A5H)




Q: Is the following code correct ?

① MOVLW 9H → 0 0 0 0 1 0 0 1 (MSB, LSB) 09H افترضها [صحيحة]

② MOVLW A23H → more than 8 bits [خاطئة] Syntax Error

② **ADD**

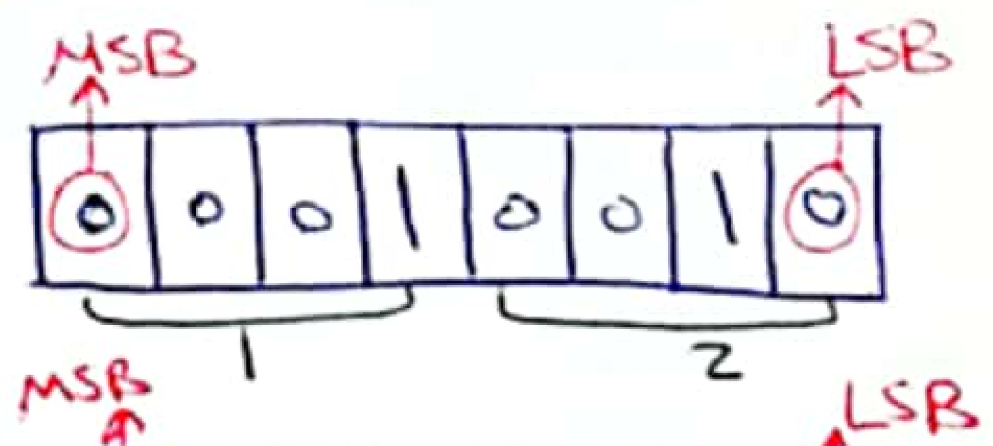
- ADDLW [كلمة محجوزة ازا أغلطت بكتابتها بيعطيك Syntax Error]

| ADDLW K / Add literal value k to WREG |
|---|

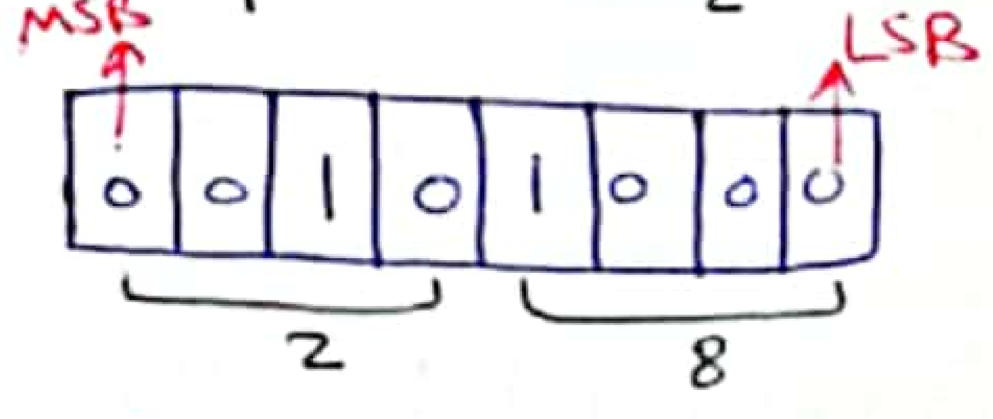
K: فوريا "immediate"

**Ex** (MSB LSB: 1 2)

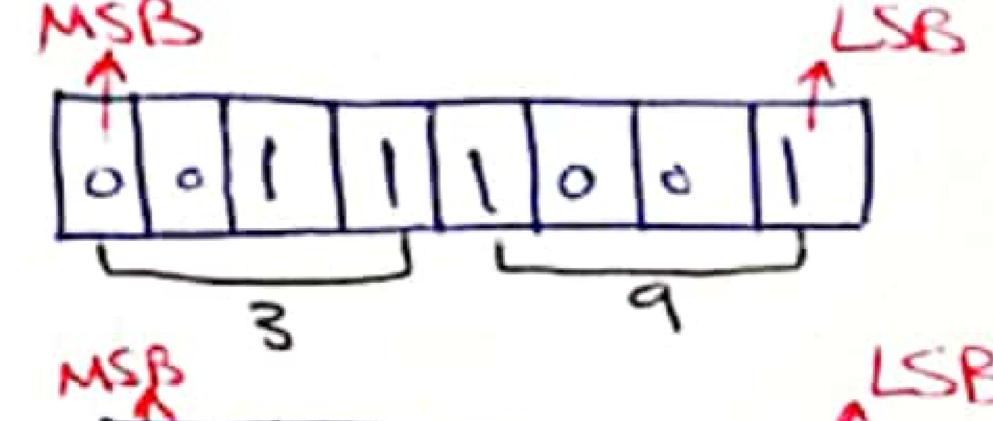
MOVLW 12H (نفس الفكرة بنرتبها ونحطها للـ WREG)



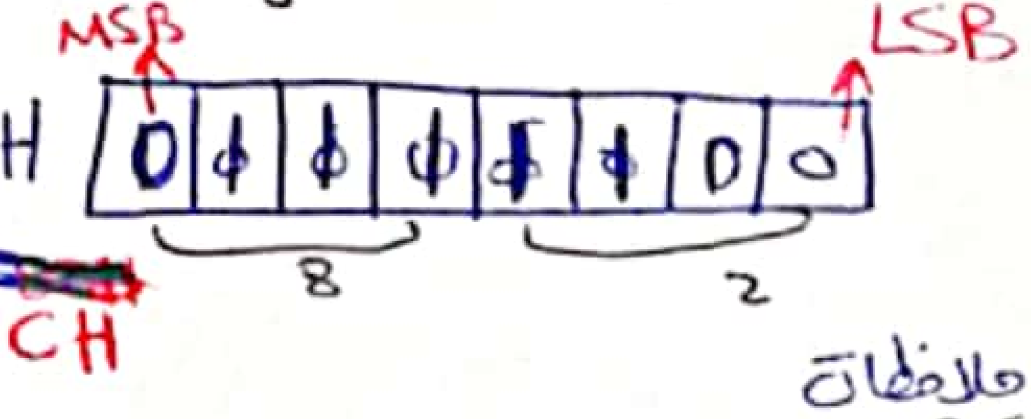

ADDLW 16H (بنضيفها للقيمة الموجودة في W السابقة) 16H + 12H = 28 H




ADDLW 11H
28H + 11H = 39 H




ADDLW 43H
39+43 = 7CH




ملاحظات

* بيصير عندك (overflow) ازا ضفت عدد داخل تجاوز الـ 8bits الناتج فيعطيك Syntax Error

* MOVLW 0H بترجع للصفر



Scanned with CamScanner

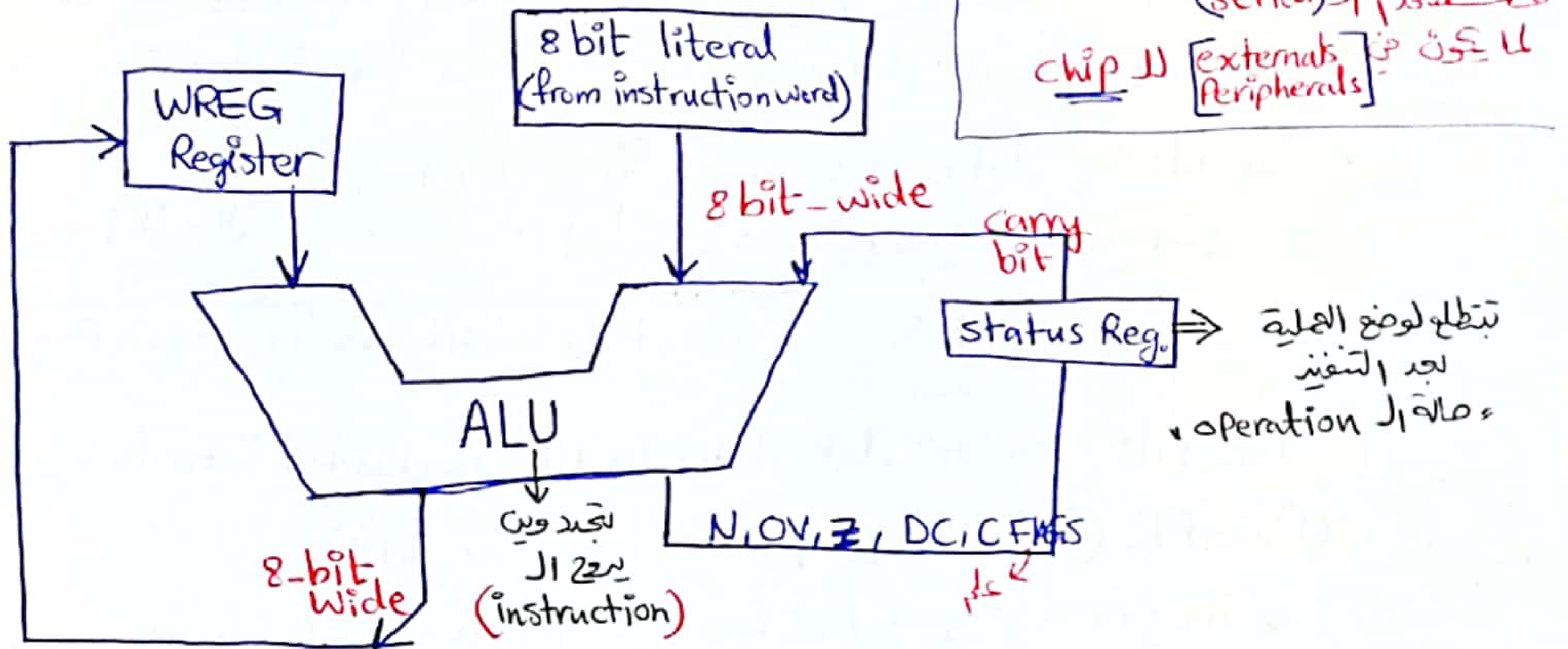
← الطريقة اللي بيتم فيها تنفيذ ال Commands → add / move ⇐ Immediate (Literal) addressing mode

بحيث تكون ال Value جزء من ال [Instruction]

* PIC WREG and ALU Using Literal :- [Parallel in the chip itself]

مش مستخدم ال (serial)

لا تكون في [externals Peripherals] لل chip




status Reg. ⇒ تتطلع لوضع العملية بعد التنفيذ حالة ال operation.

Note: CPU efficiency : [تتحدد سرعة الكمبيوتر و أدائه]

تضم (ALU, CU + buses ⋮)

---

## Review Questions

1. Write instructions to move value 34H into the WREG reg.?

   MOVLW 34H

2. Write instructions to add 16H and CDH. Place the result in the WREG reg.?

   MOVLW 16H
   ADDLW CDH

3. No value can be moved directly into the WREG reg. False [it moves directly]

4. What's the largest hex value that can be moved into an 8 bit register? and what's the decimal equivalent of that hex value? FF in hexadecimal ⇓ 255 in decimal

5. The vast majority of registers in the PIC are 8 bits.

Scanned with CamScanner

## 2.2 The PIC file register " بال RAM "

الرجوع لصفحة ⑤ من الشابتر الأول

* They're the second type of PIC registers
* They're called (data memory space) which is read/write memory. (static RAM)

[سميت هكذا للتمييز بينها وبين ال ROM (program memory space) (code)]

* The file register is read/write memory used by the CPU for data storage, scratch pad, and registers for internal use and functions.

* The file reg. data RAM varies from chip to chip
  لكن شكل الأحوال هي سعتها تتراوح [32 bytes → several thousands bytes] حسب نوع الشركة

  (so it has a size-byte width as in WREG Reg.)

مكتوبة بالشابتر الأول

# The file register data RAM in PIC is divided into two sections:

① GPR (general purpose registers or RAM)

* they're a group of RAM locations that are used for data storage and scratch pad.
* Each location is 8 bits wide, [ويخزن أي معلومة لغاية 8 bits]
* The no. of these locations in GPR varies from chip to chip.

(ميزاته)
- ① Larger than SFR
- ② Difficult to manage them by using assembly language.
- Easier to handle them by C compiler.

* The microchip website provides the data RAM size, which's the same as GPR size

② SFR (special function registers)

* They are 8 bit registers.
* They are dedicated to specific functions [ALU status, timers, serial communication, I/O ports, ADC, ...]
* The functions of SFR are fixed by the CPU designer at the time of design
  because ⇒ It's used for control of mc or peripheral.
* The no. of locations in the file reg. (المخصصة لل SFR) depends on:
  ① the pin no.'s ② peripheral functions supported by that chip
  ويختلف هالعدد من شركة لأخرى

**NOTE** In a PIC chip, no more than 100 bytes used for SFRs
والباقي بيكون لل GPR

File reg.(bytes) = SFR(bytes) + GPR(bytes)



Scanned with CamScanner

# File Register Size

| | File register (bytes) | = | SFR (bytes) | + | GPR (bytes) |
|---|---|---|---|---|---|
| PIC12F508 | 32 | | 7 | | 25 |
| PIC16F84 | 80 | | 12 | | 68 |
| PIC18F1220 | 512 | | 256 | | 256 |
| PIC18F452 | 1792 | | 256 | | 1536 |
| PIC18F2220 | 768 | | 256 | | 512 |
| PIC18F458 | 1792 | | 256 | | 1536 |
| PIC18F8722 | 4096 | | 158 | | 3938 |
| PIC18F4550 | 2048 | | 160 | | 1888 |

4096 → max. size of data RAM space (4K)

فكرة سؤال

ممكن يعطينا حجم الـ File register لنوع [PIC] معين مع حجم الـ SFR منه أو الـ GPR ويطلب كم الحجم حيكون للآخر

(Ex): For PIC 18F4550 has 160 SFR byte, while it's file reg. size is an 2048 bytes. Calc. it's GPR size?

$$\text{GPR size (bytes)} = 2048 - 160 = 1888 \text{ bytes } \#$$

---

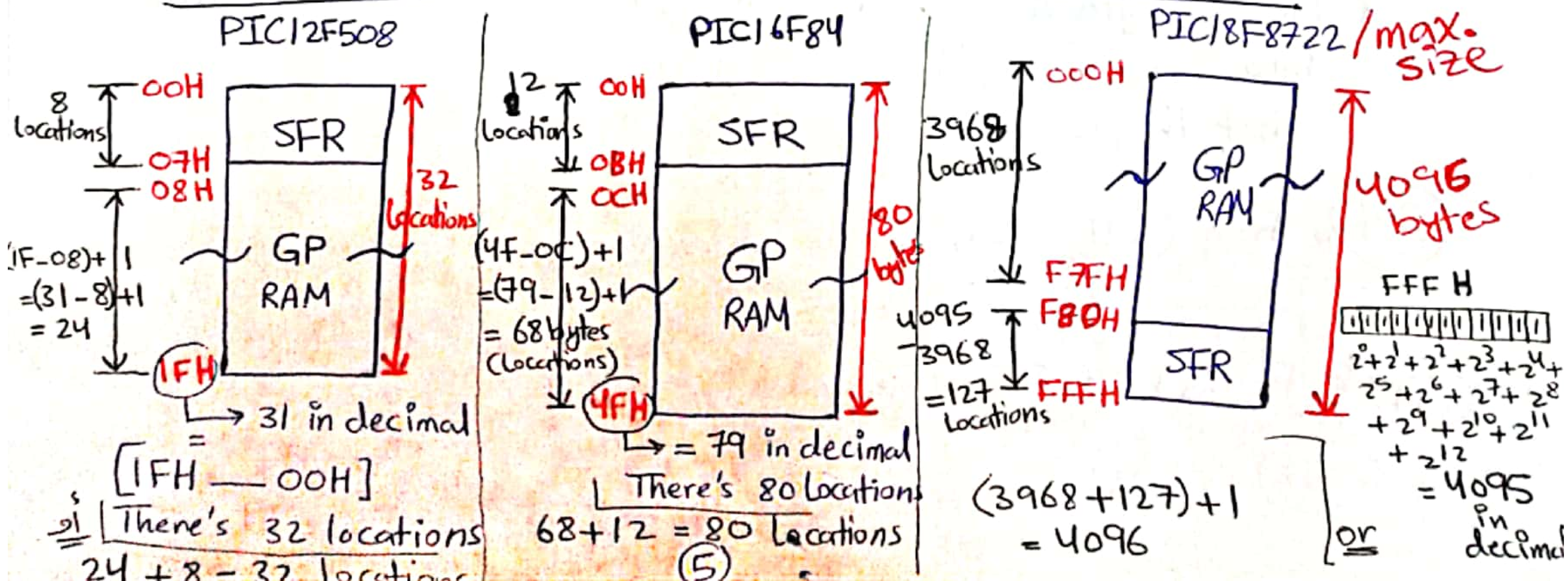
* What's the difference between GPRAM and EEPROM?

general purpose RAM ← [الاثنين موجودين ضمن الـ RAM]
Electrically Erasable programming ROM ←

GPRAM → used for internal data storage by the CPU [لازم تكون موجودة]

EEPROM → considered as an add-on-memory that one can also add externally to the chip. [مو دايماً تكون موجودة]

# File Register for PICs example:





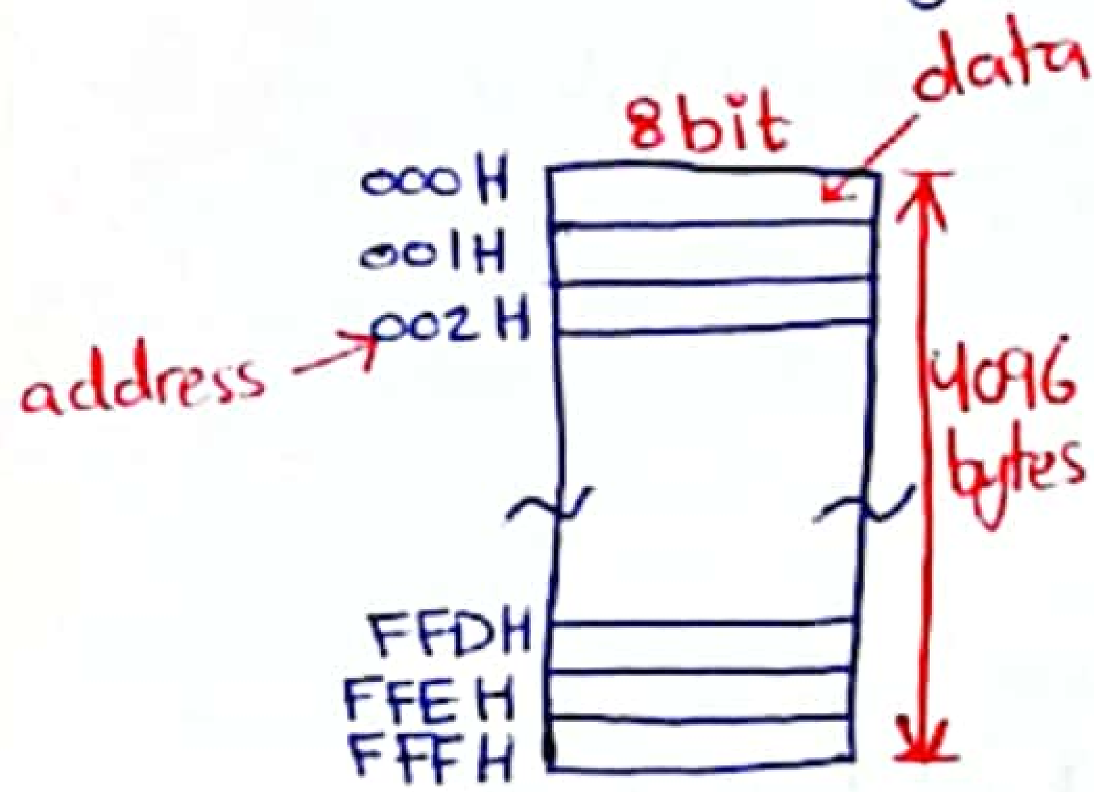

Scanned with CamScanner

# File register and access bank in the PIC18

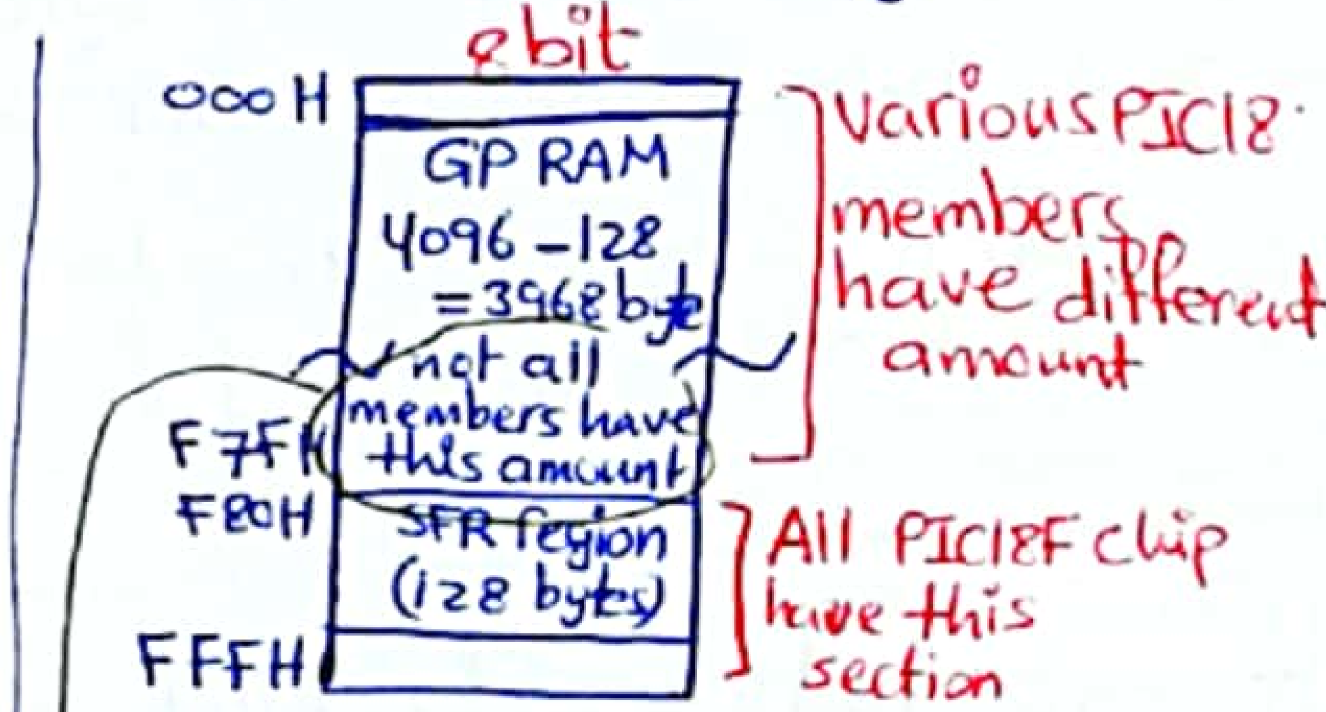
data RAM space

- The PIC18 Family can have a max. of 4096 bytes (4K), it has addresses (000 – FFF) H
- It's divided into 256 byte banks [ $2^8 = 256$ byte ; 8 bits ], so we can have up to a max. of 16 banks (16 * 256 = 4096 bytes)
- At least there's one bank for the file register in PIC18 family known as default "access bank"
- Bank switching is a method used to access all the banks that have more than the minimum access bank of the file reg. for PIC18 family. (more than 256 bytes)

## * File Register for PIC18 family

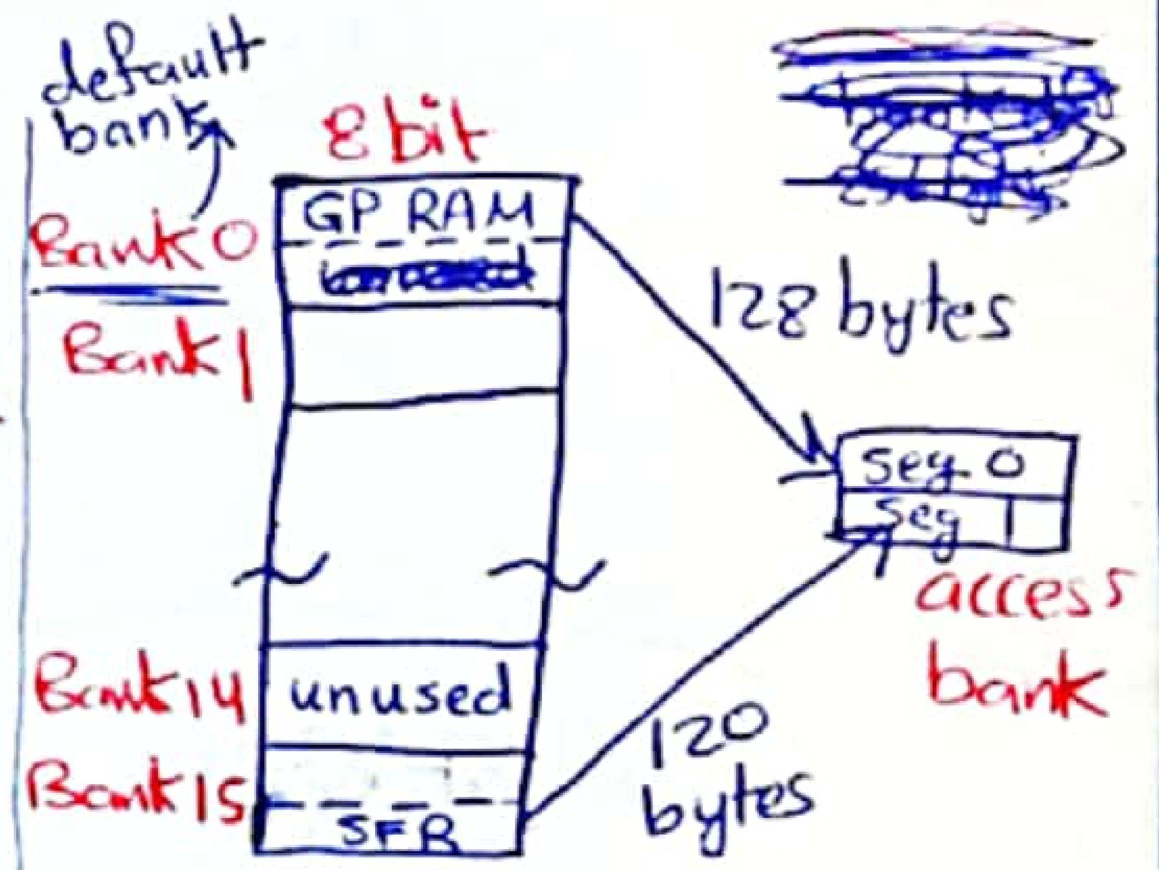



a) max. space of file reg. (data RAM) in PIC18F (4096 byte)

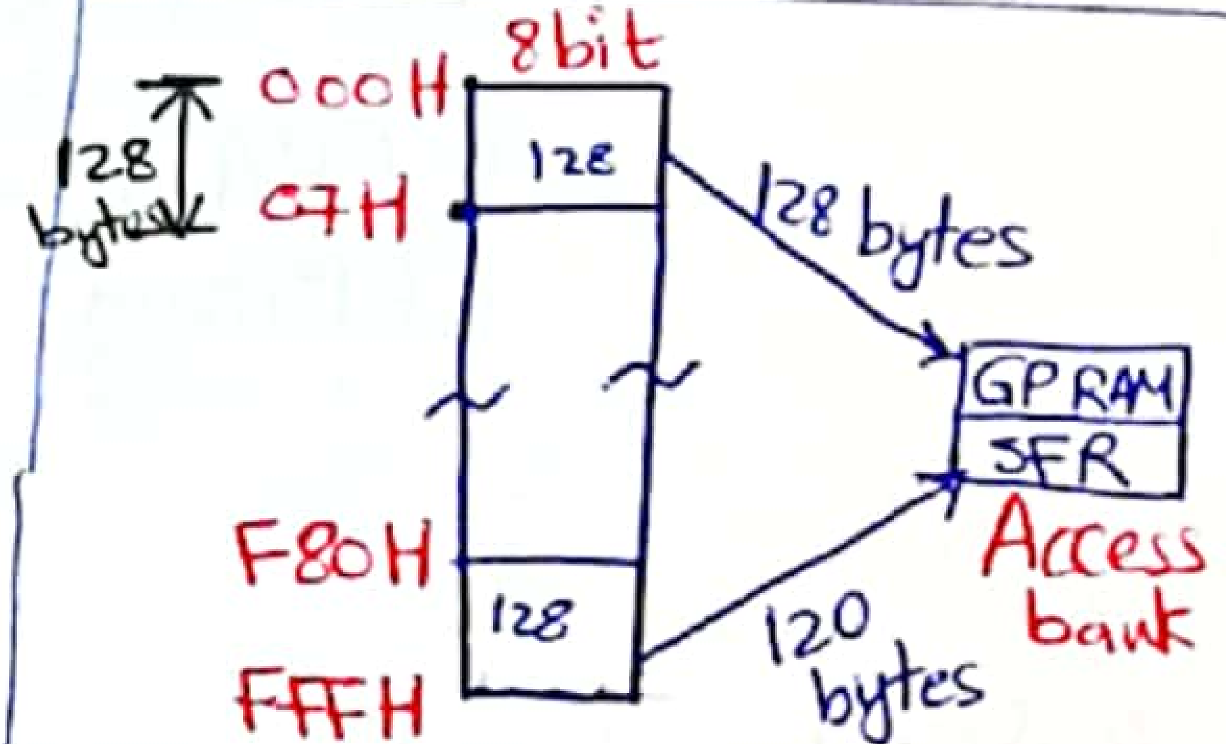



b) File reg. allocation between GP RAM and SFR

→ according to the chip itself cuz we don't need all of this amount sometimes.




c) data memory map




d) access bank

كل data إلها address (memory) إل

file Reg.

* The 256 bytes access bank is divided into 2 equal sections (128 bytes) "discontinuos" that they're given to → GPR → SFR

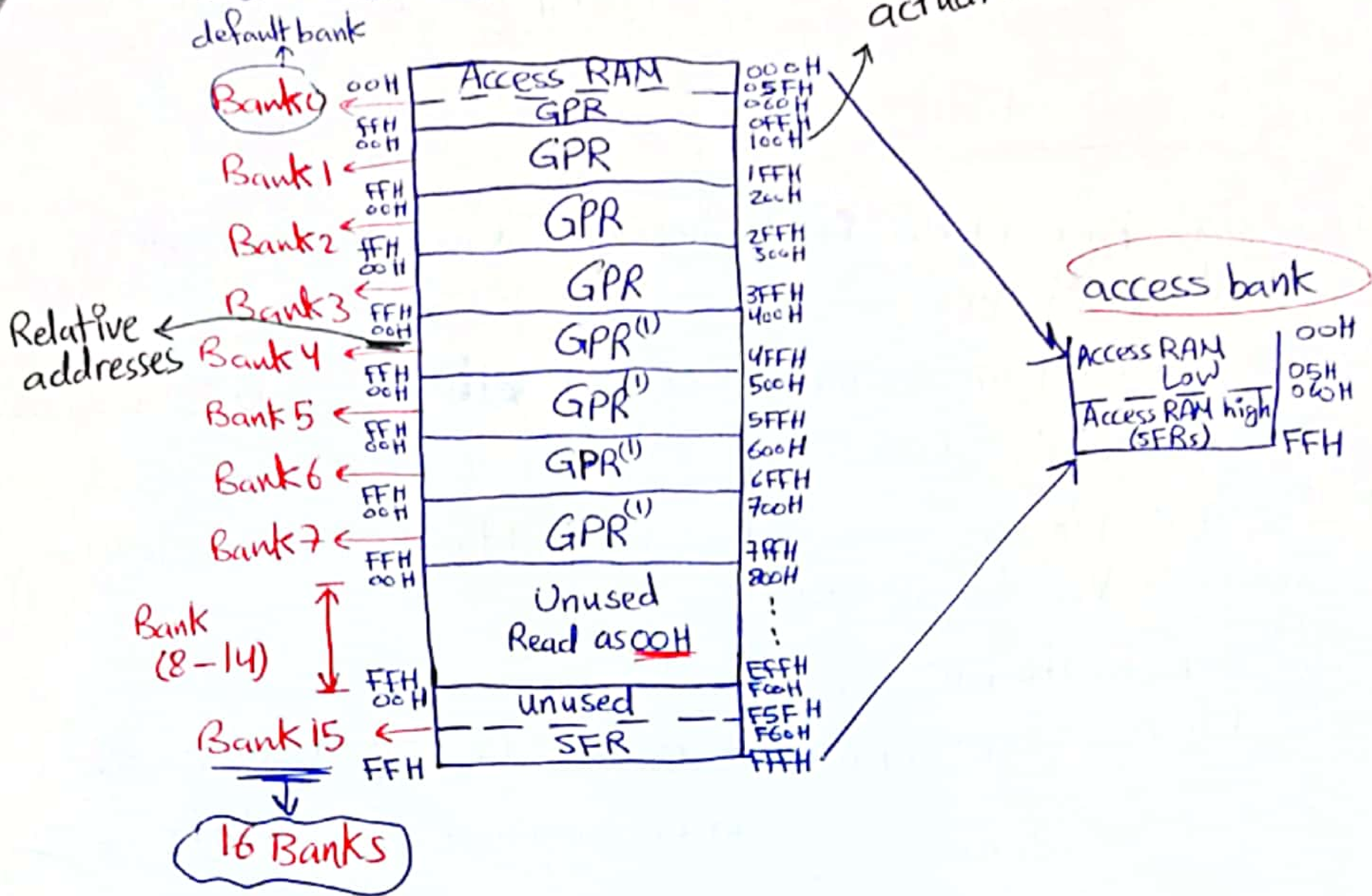
GPR • From (00H – 7FH) 128 bytes → are set for GPR, used for read/write storage and scratch pad. → that can be accessed directly by its address

SFR • From (F80H – FFFH) 128 bytes → used for SFR



Scanned with CamScanner

* Data memory map :-

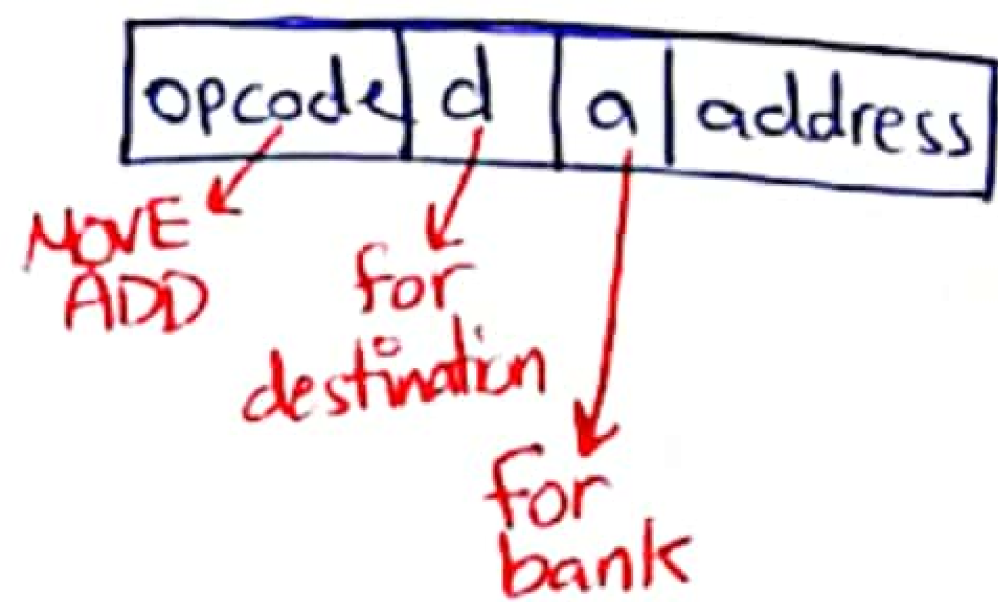



* When [a = 0] → The BSR (Bank select Reg.) is ignored and the access bank is used.

The first 96 bytes are GP RAM (from bank 0)

[SFR ⇐ (160 bytes) والباقي (from bank 15)]

* When [a = 1] → The BSR specifies the bank used by the instruction



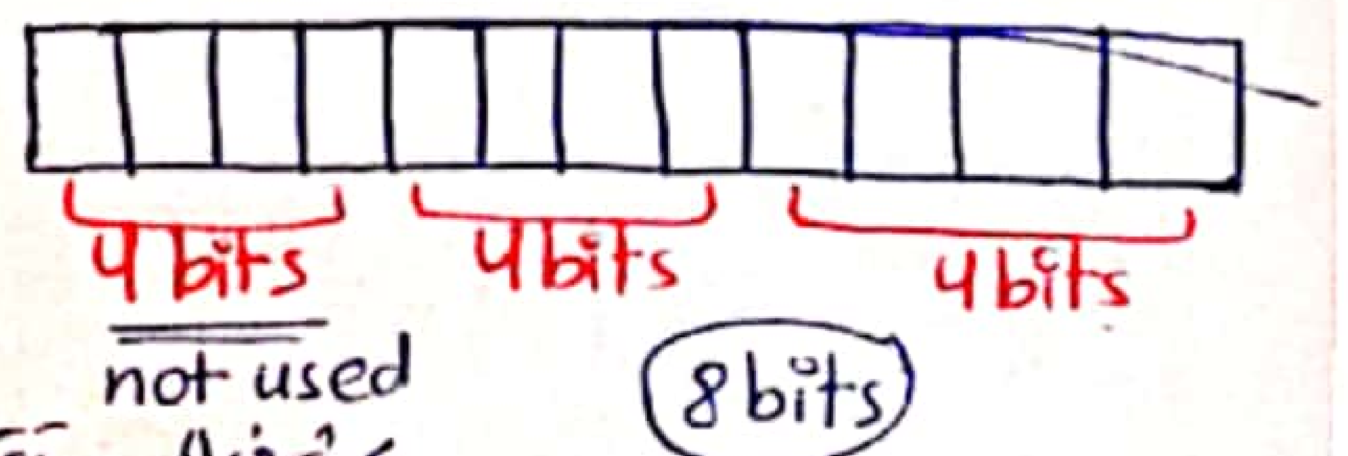

---

**Note** RAM space memory = 4096 bytes = 4K

$$4096 = 2^2 * 2^{10}$$ [12 bits]

4, 1024

[000H : FFFH] (Hexa) يعني لازم يكون عندي ٣ منازل من ال = 




بحيث في المستقبل اذا ضرب سال معين أقدر أنقله ع address تاني

⑦ بعد ما أغير من ال software على ال address الجديد اللي بدي أنقل السلاح اله.

## 2.3 Using Instructions with the default access bank :

* As we mentioned before, the addressing modes are →

① Immediate "Literal"

ال value تكون جزء من ال instruction

[ADDLW, MOVLW]

W : indicates for WREG

② Register

It allows to access other locations in the file reg. for ALU & for other operations.

[MOVWF, ADDWF, MOVF, MOVFF, COMF, INCF, DECF]

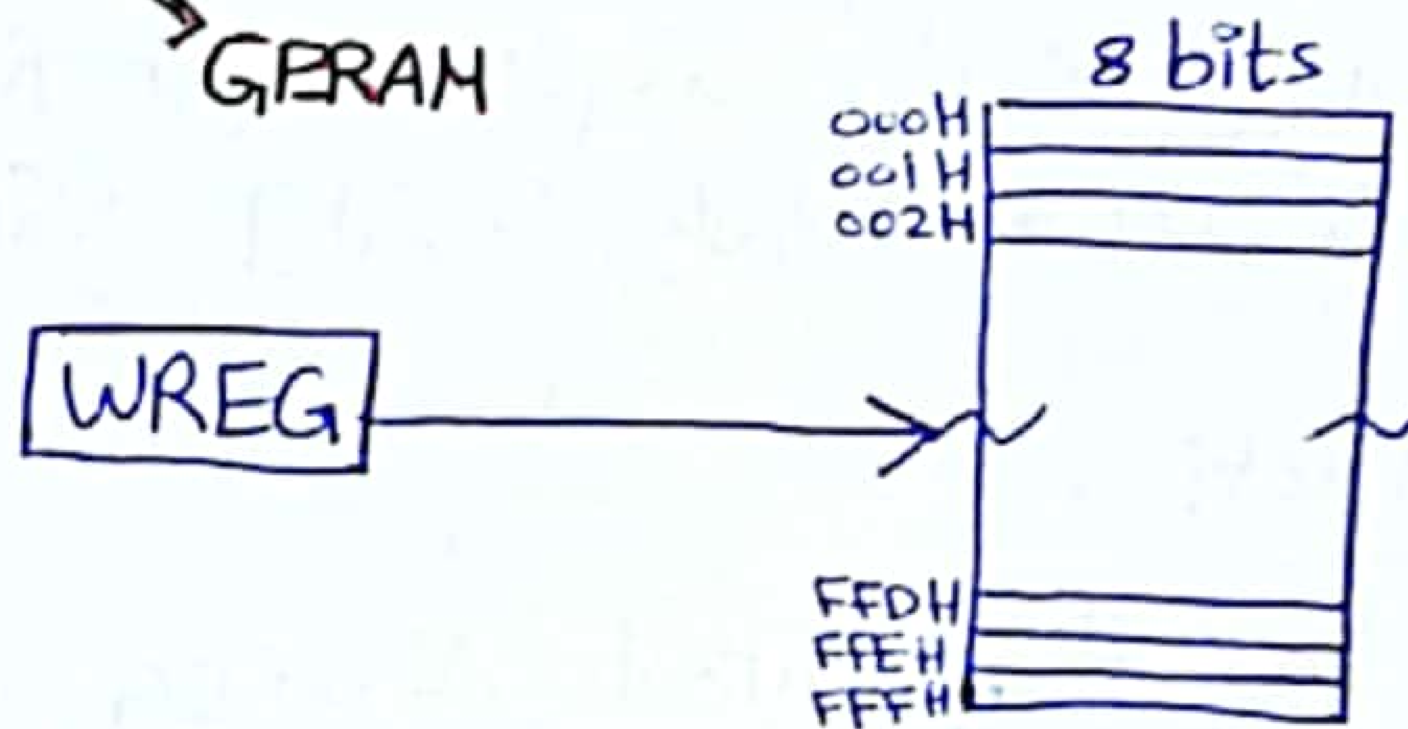
F : indicates for a file reg.

---

a) MOVWF instruction → [منزل ال WREG بالفايل ريجستر] (F ال)

* It tells the CPU to copy the source reg. (in WREG) to a destination in the file reg. (F), so after that they have the same value (WREG → file reg)

↙ SFR ↘ GPRAM




الفرق مع

MOVLW : data بنحط بال address

MOVWF : address بنحط فيه ال data اللي جاي من MOVLW (WREG)

**EX**

① The following program first loads the WREG with value 55H, then moves it around to various SFRs of ports B, C, and D : (are SFRs)

```
→ MOVLW   55H     : [WREG = 55H] حطينا 55H على WREG
  MOVWF   PORTB   : COPY WREG to PORTB [PORTB = 55H]
  MOVWF   PORTC   :  "    "    "   C  [ " C = 55H]
  MOVWF   PORTD   :  "    "    "   D  [ " D = 55H]
```

② The following program will put 99H into locations 0-4 of the GPR region in the file reg. :

```
→ MOVLW   99H  (data)     : WREG = 99H
  MOVWF   0H   (address)  : copy 99H (move) to location 0H
  MOVWF   1H              .
  MOVWF   2H              .
  MOVWF   3H              .
  MOVWF   4H              .
```

| Address | Data |
|---|---|
| 000 | 99 |
| 001 | 99 |
| 002 | 99 |
| 003 | 99 |
| 004 | 99 |



Scanned with CamScanner

Ex (2.1) State the contents of file reg. RAM locations after following program :

```
MOVLW  99H
MOVWF  12H
MOVLW  85H
MOVWF  13H
MOVLW  3FH
MOVWF  14H
MOVLW  63H
MOVWF  15H
MOVLW  12H
MOVWF  16H
```

| Address | Data |
|---|---|
| 012H | 99 |
| 013H | 85 |
| 014H | 3F |
| 015H | 63 |
| 016H | 12 |

بعد تنفيذ البرنامج

* الهدف من الـ MOVWF انو مو دايمًا منقدر ننقل الـ data مباشرة للـ (file reg.) مثل الـ (GP-RAM) اللي لازم عن بالأول على WREG ومن هناك ننقل الـ data للـ (GP-RAM) [ ليش معتمد على الـ WREG بعاد التركيبة؟ ]

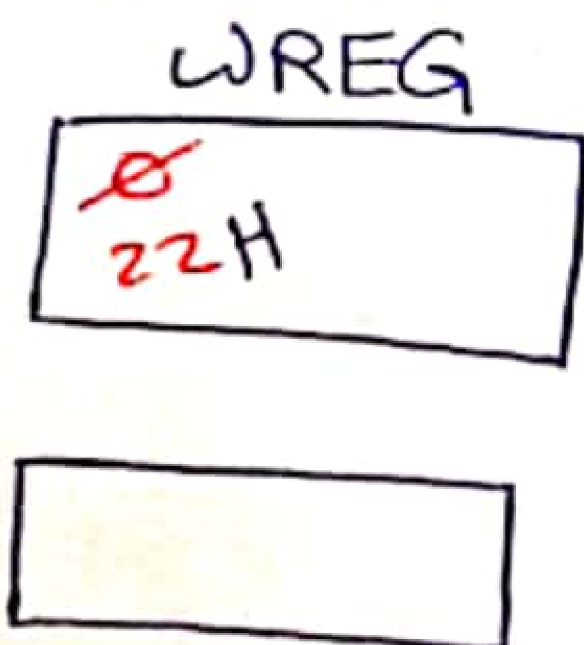
⇒ ADDWF Instruction :

* it adds together the contents of WREG and a file reg. location that can be one of the SFRs or GPR.
* the destination for the result can be the WREG or the file reg.

Ex (2.2) State the contents of file reg. RAM locations 12H and WREG after the following program :

```
MOVLW  0
MOVWF  12H
MOVLW  22H
ADDWF  12H , F     (WREG = 22, 012H = 22)
ADDWF  12H , F     (WREG = 22, 012H = 44)
ADDWF  12H , F     (WREG = 22, 012H = 66)
ADDWF  12H , F     (WREG = 22, 012H = 88)
```

F ← address معطى بالجهاز
ADDWF: اجمع اللي بـ WREG مع اللي بـ 012H
F: حطها بالـ file reg. (مطها بالعنوان)




| Address | Data |
|---|---|
| 012H | Ø 22H 44H 66H 88H AAH |

$22 + 88$ → 00100010 + 10001000 = 10101010 → AAH

MOVLW 22H
(WREG = 22, 012H = 0) ADDWF 12H, F ← حفظ النتيجة
(WREG = 22, 012H = 22) ADDWF 12H, W ← بال W
(WREG = 44, 012H = 22) ADDWF 12H, W

← لو كان البرنامج السابق :

| | Address | Data |
|---|---|---|
| ① | 012 H | 22 |
| ② | 012 H | 22 |
| ③ | 012 H | 22 |

(or)

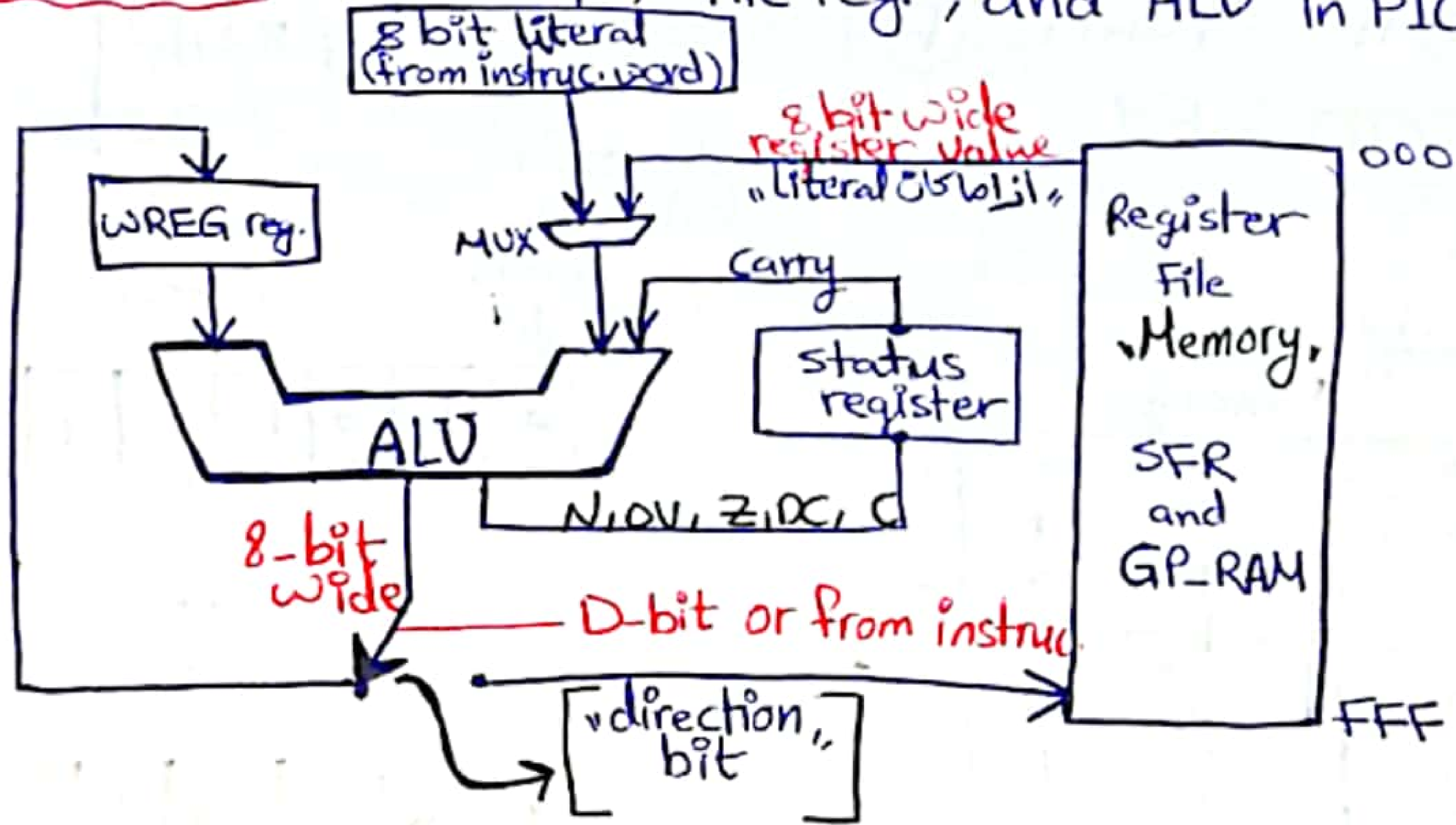
MOVLW 0
MOVWF 12H ← صفّر ال 12H
MOVLW 22H
(WREG = 22, 012H = 0) ADDWF 12H, F = 22
(WREG = 22, 012H = 22) ADDWF 12H, W = 44
(WREG = 44, 012H = 22) ADDWF 12H, W

| Address | Data |
|---|---|
| 012H | ∅ 22 |

Figure (2-5) : WREG, file reg., and ALU in PIC18

مهم للامتحان

حفظ




* اللي بيروح عشان ال (Status Reg.) بيكون إما

- Negative
- Overflow ← (+ve) = 1
- Zero
- Digital Carry
- Carry

they occur after the ALU operation

[موجود بالمانيوال تبع البيك نفسه بخصوص كل وحدة فيهم]

b) COMF instruction →

[COMF filereg. , Address D] → Destination Bit = [F , 1 / W , 0]

* It tells the CPU to complement the content of filereg. and places the results in WREG or in file reg.

Ex In the following program, put 55H into WREG and then send it to SFR location of PortB, then its content is complemented.

```
→ MOVLW  55H         [WREG = 55H]
  MOVWF  PORTB       [PORTB = 55H]
  COMF   PORTB , F   [PORTB = AAH]
```

Ex(2.4) Write a simple program to toggle the SFR of PORTB continuosly forever → Loop يعني بدنا

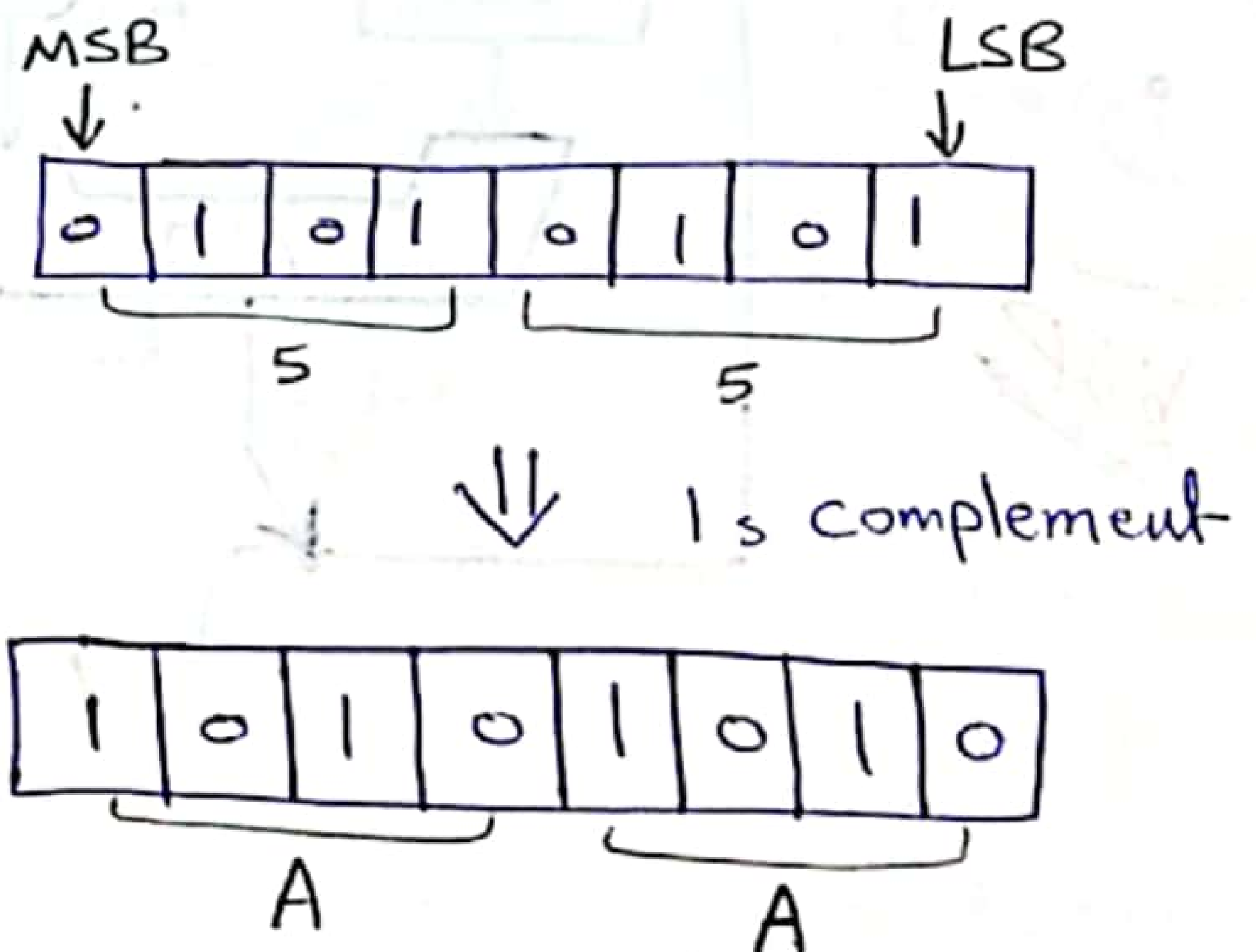
```
→     MOVLW  55H        WREG [55]
      MOVWF  PORTB
[B1   COMF   PORTB, (F)   → Destination Bit
      GOTO   B1 ]
```

بحط ال label على ال command الي بدي تكراره

| | Address | Data |
|---|---|---|
| PortB ← | F81 H | ~~55~~ AA |
| PortC ← | F82 H | وهكذا |
| PortD ← | F83 H | |

**Remember**

| Binary | | | | H |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |
| 1 | 0 | 0 | 0 | 8 |
| 1 | 0 | 0 | 1 | 9 |
| 1 | 0 | 1 | 0 | A |
| 1 | 0 | 1 | 1 | B |
| 1 | 1 | 0 | 0 | C |
| 1 | 1 | 0 | 1 | D |
| 1 | 1 | 1 | 0 | E |
| 1 | 1 | 1 | 1 | F |




* حتى نشكل Loop
منحدد عالأمر الي بدنا اياه يتكرر (معين Label) مثلاً L1
GOTO L1

Not reserved word

## c) DECF instruction →

[DECF File reg. , Address D]

* It tells the CPU to decrement the content of file reg. and places the results in WREG or in file reg. .

الطرح من المحتوى (-1)

**Ex** In the following program, put the value 3 into file reg. location 0X20, then the value is decreamented and placed in file reg. :

(it's an address not a data)

```
MOVLW   3H
MOVWF   20H
DECF    20H, F
DECF    20H, F
DECF    20H, F
DECF    20H, F
```

WREG: 3

| Address | Data |
| --- | --- |
| 020H | ~~3~~ ~~2~~ ~~1~~ ~~0~~ FF (-1) |

---

```
MOVLW   3H
MOVWF   20H
DECF    20H, W
DECF    20H, W
DECF    20H, W
DECF    20H, W
```

WREG: ~~3H~~ ~~2~~ ~~1~~ ~~0~~ FF

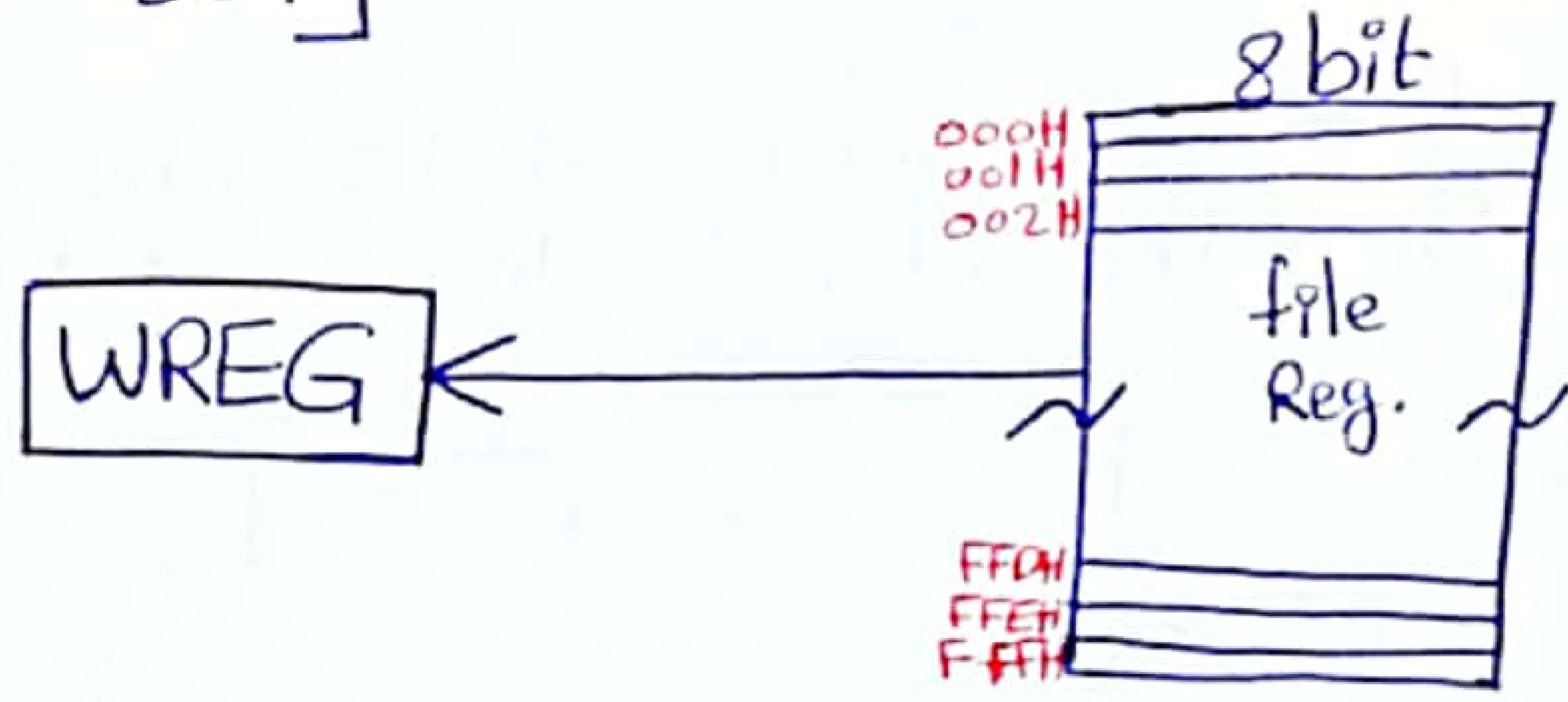
| Address | Data |
| --- | --- |
| 020H | 3H |

(ما بتغير شي على ال File Reg. صورت)

d) MOVF instruction →

[MOVF file Reg. , Address D] → D = [0 , Copy file reg. content to WREG ; 1 , the content is copied to itself of the file reg.]

* We use this instruction to bring data into WREG from I/O pins, and sometimes use it to copy file reg. to itself for the purpose of testing file reg. contents (status updating Process) [When D=1]




Ex(2.5)

Write a simple program to get data from the SFRs of PORTB and send it to the SFRs of PORTC continuosly :

```
→ AGAIN  MOVF   PORTB, W
         MOVWF  PORTC
         GOTO   AGAIN
```

- MOVF → من موقع معين لـ W أو من موقعه لنفسه
- MOVWF PORTC → حط الشي اللي جاي من (Port B) لـ (Port C)
- address لتغير الـ
- AGAIN → Not Reserved Word "Label"

WREG: XX

| Address | Data |
|---|---|
| F81 H | XX |
| F82 H | XX |
| F83H | |

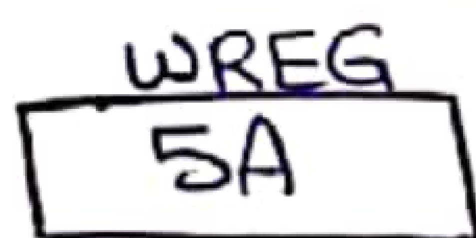
Ex(2.6)

Write a program to get data from the SFRs of PORTB. Add the value 5 to it and send it to the SFRs of PORTC :

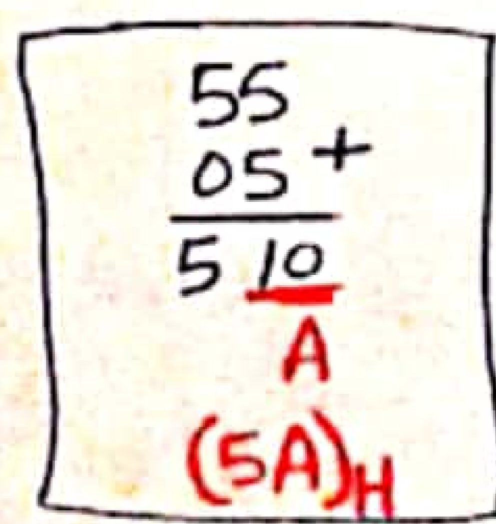
```
→ MOVF   PORTB , W   ; bring data from Port B into WREG
  ADDLW  05H         ; add 5 to WREG
  MOVWF  PORTC       ; Copy WREG to port C
```

MOVF نقل الى — bring

MOVWF — نسخ المحتوى من WREG — Copy



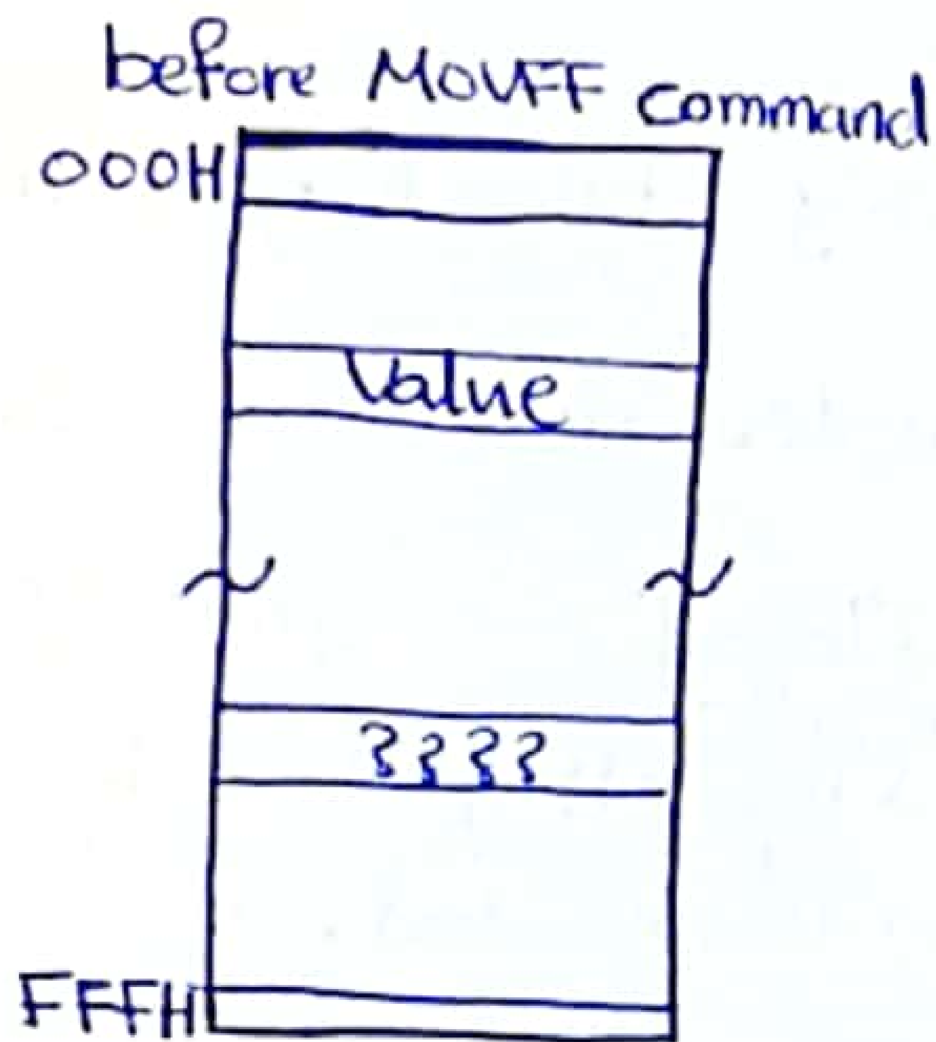

| Address | Data |
|---|---|
| F81 H | 55H |
| F82H | |
| F83H | |

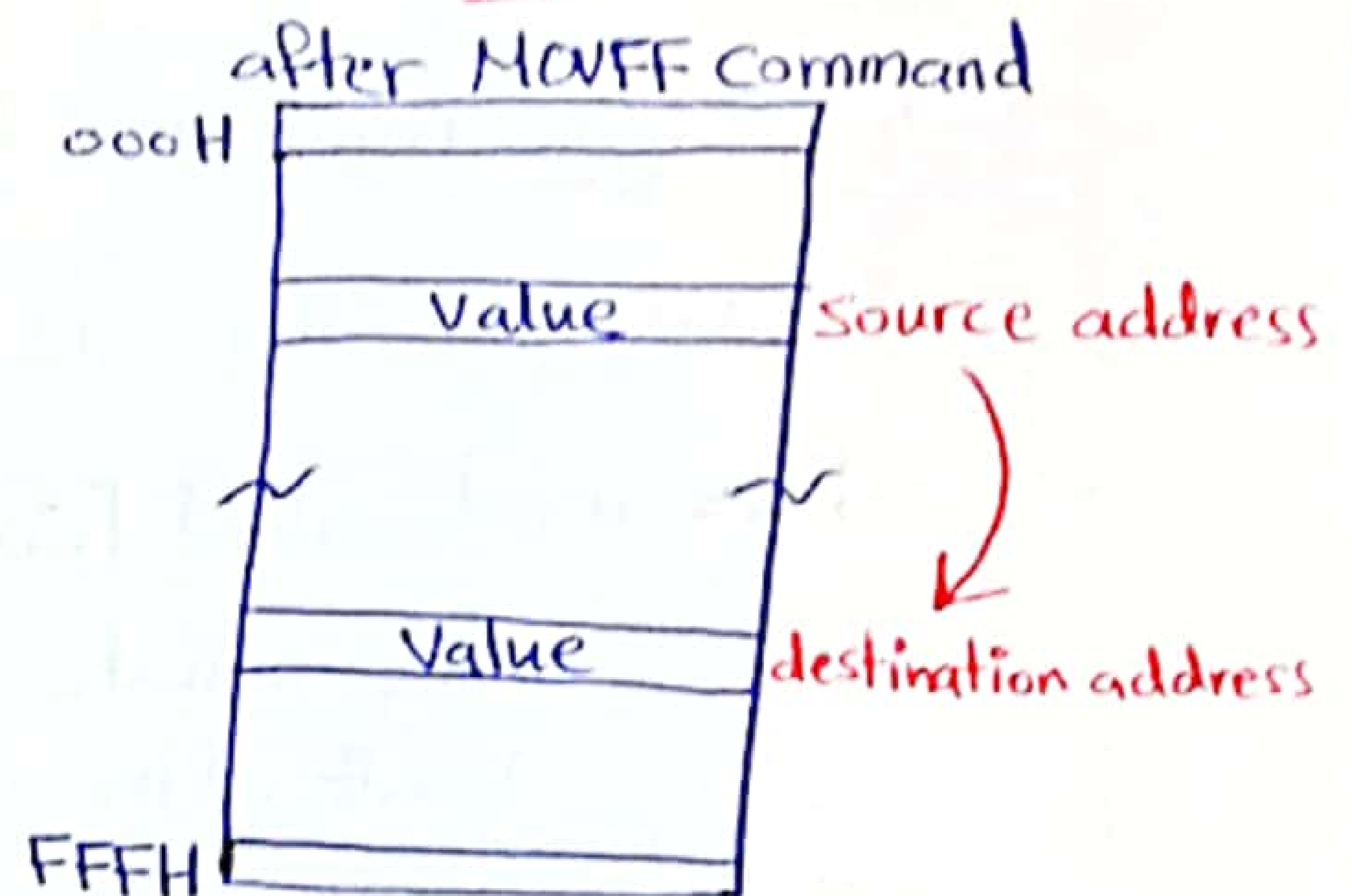


## e) MOVFF instruction →

* It copies data from one location in file reg. to another location in file reg. "Without going to WREG"

[MOVFF source file reg. / destination file Reg.]




**Ex(2.7)** Write a program to get data from the SFRs of Port B and send it to the SFRs of Port C continuosly:

```
AGAIN  MOVFF  PORTB , PORTC
       GOTO   AGAIN
```

copy data from Port B and send it to Port C

keep doing it forever

WREG: XX

| Address | Data |
|---|---|
| F81 H | ~~XX~~ XX |
| F82 H | XX |

### Review Questions

1] True or False:

* The access bank is 256 bytes divided evenly between GPRs and SFRs. T
* No value can be moved directly into GP-RAM. T

2] Write instructions to add the values 16H and CDH. Place the result in location 0 of the file reg.?

```
MOVLW 16H
MOVWF 0
MOVLW CD
ADDWF 0H, F
```

3] What's the largest hex value that can be moved into a location in the file reg.? and what's the decimal equ.? FF, 255

4.] [ADDWF PORTB, W] puts the result in WREG.

Scanned with CamScanner

## 2.4 PIC Status Register

* "Status Register": is a flag register in PIC that indicating arithmetic conditions such as the carry bit.

### PIC18 Status Register

→ The status "flag" reg. is an 8 bit reg., only 5 bits of them are used by the PIC18

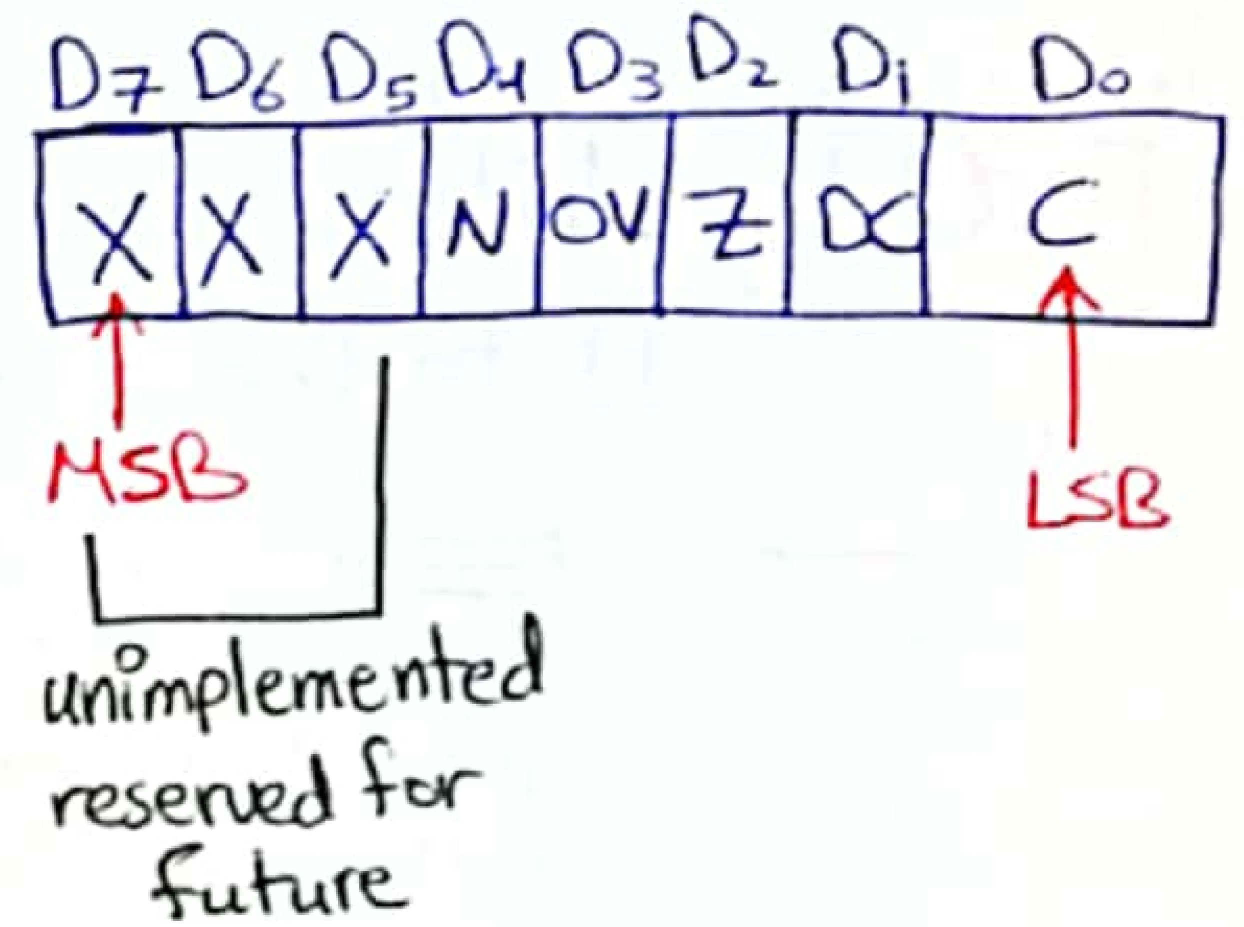
the 3 bits others are unimplemented and read as 0.

→ 5 bits used called [Conditional Flags] معنى هاد الحكي

they indicate some conditions that ⇐ بإنو result after an instruction is executed.

they're:

① $D_0$ (C) → carry flag
② $D_1$ (DC) → Digital Carry Flag
③ $D_4$ (N) → Negative flag
④ $D_3$ (OV) → Over Flow flag
⑤ $D_2$ (Z) → Zero flag




* بنقرأ عند كل واحد فيلم ← 0 , لما شرط الـ(flag) نفسه ما يصير.
← 1 , لما شرط الـ(flag) نفسه تحقق ويصير.

### Flags Condition

1 Carry flag → It's set whenever there's a carry out from the $D_7$ bit.

2 Digital Carry flag → It's set whenever there's a carry from $D_3$–$D_4$ during an ADD or SUB operation

3 Zero flag → It's set when the result is zero, then Z=1
" " isn't zero, then Z=0

4 Over flow flag → It's set whenever the result of asigned no. operation is too large



Scanned with CamScanner

5. Negative Flag → It's set whenever the $D_7$ bit is one, N=1 / يعني النتيجة سالبة

If 〃 〃 〃 〃 zero, N=0 / النتيجة موجبة

---

**Ex(2.8)** Show the status of the C, DC, Z, N and OV flags after the addition of 38H and 2FH in the following instructions :

```
MOVLW 38H
ADDLW 2FH
```

| [illegible] | [illegible] | [illegible] | A | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |
| 1 | 0 | 0 | 0 | 8 |
| 1 | 0 | 0 | 1 | 9 |
| 1 | 0 | 1 | 0 | A |
| 1 | 0 | 1 | 1 | B |
| 1 | 1 | 0 | 0 | C |
| 1 | 1 | 0 | 1 | D |
| 1 | 1 | 1 | 0 | E |
| 1 | 1 | 1 | 1 | F |

→
```
   38 H  :   0 0 1 1 1 0 0 0
 + 2F H  :   0 0 1 0 1 1 1 1  +
 -------   -----------------
 = 67 H      0 1 1 0 0 1 1 1
               6       7  H
```

① C = 0 , ما في [Carry bit] لبعد ($D_7$)

② DC = 1 ; في (Carry) لبعد ال [$D_4 - D_3$]

③ Z = 0 , because the WREG has a value other than zero after the addition

④ N = 0

⑤ OV = 0

---

**Ex(2.9)** Show the status of the C, DC, Z and OV flags after the following instructions :

```
MOVLW 9CH
ADDLW 64H
```

→
```
   9C H  :     1 0 0 1 1 1 0 0
 + 64 H  :     0 1 1 0 0 1 0 0  +
 -------     -----------------
             1 0 0 0 0 0 0 0 0
```

(1) ↓ في ال C

C = 1

DC = 1

Z = 1

OV = 0



Scanned with CamScanner

# Review Questions

1. The flag register in the PIC is called the <u>Status Register</u>
2. What's the size of the flag register in the PIC? <u>8 bits</u>
3. Which bits of the status Register are unused? <u>3 bits [$D_5 - D_7$]</u>
4. Find the C, Z and DC flag bits for the following codes:

* MOVLW 9FH → 
ADDLW 61 H

```
 ①1111111
  10011111
  01100001 +
 ----------
 100000000
```

[ Z = 1
C = 1
DC = 1 ]

* MOVLW 82H →
ADDLW 22H

```
       1
  10000010
  00100010 +
 ----------
  10100100
```

[ Z = 0
C = 0
DC = 0 ] → لأنو كاع
مش موجود $D_2$ [$D_4 - D_3$]

* MOVLW 67H →
ADDLW 99H

```
 ①1111 111
  01100111
  10011001 +
 ----------
 100000000
```

[ Z = 1
DC = 1
C = 1 ]

Scanned with CamScanner

## 2.5 PIC Data Format And Directives "Supported by PIC assembler"

* There's one data type (8bits – 1byte)
* It's the job of the programmar to break down data larger than 8bits.
* Data type → +ve
  ↳ -ve

* Data Format representation :

① Hex numbers →
- use h (or H) right after the no. (MOVLW 99H)
- put 0x (or 0X) infront of the no. (MOVLW 0x99)
- put nothing in front or back of the no. (MOVLW 99)
- put h in front of the no., but with single quotes around the no. (MOVLW h'99')

notes

# ADDLW E5H → invalid (0E5H)
ADDLW 6H → valid

# MOVLW C6 → invalid (0C6)

لاحظ إنو لما يبلش العدد بالهيكسا بالأحرف (A-F) لازم عيساره يكون في 0 بعكس لما يكون مبلش برقم بيفرضه منه نفسه إنو في صفر

② Binary numbers →
- B'10010101' [the lowercase b will also work]

③ Decimal numbers →
- D'12' [the lowercase d will also work]
- .12

④ ASCII Character → "American Standard Code for Information Interchange"
- A'12' [the lowercase a will work also]

[مشان ال machine language زي الي بالكيبورد]

Scanned with CamScanner

# Assembler Directives

## # What is the difference between instructions and directives?

instruction → tell the CPU what to do

directives (pseudo-instruction) → give directions to the assembler.

## # Example:

① EQU → Defines a constant value, or fixed address.
(equate)

ex
```
COUNT EQU 0X25
      MOVLW COUNT
```
→ WREG = 25H (the WREG will be loaded with the value 25H)

② SET → Defines a constant value or fixed address. (زي الـ EQU)
but it can reassigned later.

using EQU for fixed data assignment

```
Data1   EQU  39          → hex data is default  ┐
Data 2  EQU  0X39                                │ for hexa's
Data 3  EQU  39H                                 │
Data 4  EQU  H'39'                               │
Data 5  EQU  h'39'                               ┘

Data 6  EQU  b'00110101'  → Binary (35 in hex)
Data 7  EQU  B'00110101'

Data 8  EQU  D'28'        → Decimal (1C in hex)
Data 9  EQU  d'28'

Data 10 EQU  A'28'        → in ASCII character
Data 11 EQU  a'28'
Data 12 EQU  '28'
```

---

Using EQU for SFR address assignment:

```
COUNTER EQU 0X00      → COUNTER = 00 قيمة ال
PORTB EQU 0XFF6       → SFR PORTB address
MOVLW COUNTER         → WREG = 00H
MOVWF PORTB           → PORTB = WREG = 00H
INCF PORTB, F         → PORTB = 01
INCF PORTB, F         → PORTB = 02
INCF PORTB, F         → PORTB = 03
```

this program is for PIC18 family ← لأنو إذا بدك تستخدم PIC16 مثلاً عنوان PORTB مختلف وبالتالي محتاج عنوان لـ EQU لل PORTB and re-assemble the program and run it.

## Using EQU for RAM address assignment (GPRAM)

```
MYREG  EQU  0X12      → assign RAM loc to MYREG
MOVLW  0              → WREG=0H
MOVWF  MYREG          → MYREG =0H
MOVLW  22H            → WREG =22H
ADDWF  MYREG,F        → MYREG = 0+22 = 22H
ADDWF  MYREG,F        → MYREG = 22H + 22H = 44H
ADDWF  MYREG,F        → MYREG = 44H +22H = 66H
ADDWF  MYREG,F        → MYREG = 66H + 22H = 88H
```

(*) The following program will move value 9 into RAM locations 0-4, then add them together and place the sum in location 10H :

```
MYVAL  EQU  9         → MYVAL=9
R0     EQU  0         → assign RAM addresses to R0
R1     EQU  1         → to R1
R2     EQU  2         → to R2
R3     EQU  3         → to R3
R4     EQU  4         → to R4
SUM    EQU  10H       → 10H بـ جمعهم

MOVLW  MYVAL          → WREG = MYVAL = 9
MOVWF  R0             → R0 = WREG = 9
MOVWF  R1             → R1 = 9
MOVWF  R2             → R2 = 9
MOVWF  R3             → R3 = 9
MOVWF  R4             → R4 = 9
MOVLW  0              → WREG = 0
ADDWF  R0, W          → WREG = R0+WREG = 18
ADDWF  R1, W          → WREG = R1+WREG = 27
ADDWF  R2, W          → WREG = R2 + WREG = 36
ADDWF  R3, W          → WREG = R3 + WREG = 45
ADDWF  R4, W          → WREG = R4 + WREG = 54
MOVWF  SUM
```

أو

```
MOVLW  9H
MOVWF  0H
  =    1H
  =    2H
  =    3H
  =    4H
ADDWF  0H, W
ADDWF  1H, W
ADDWF  2H, W
ADDWF  3H, W
ADDWF  4H, W
MOVWF  10H
```

* directives : don't generate any machine code (opcode) وتستخدم فقط من قبل الـ(assembler)
* instructions : are translated to machine code (opcode) for the CPU to execute.

---

③ **ORG** → used to indicate the beginning of the address.
(origin) It can be used for both code and data.

the no. that comes after ORG must be in hex. ← مهم

④ END → indicates to the assembler the end of the source (asm) file

⑤ LIST → it's unique to the PIC assembler, it indicates to the assembler the specific PIC chip for which the program should assembled. [LIST P=184F58]

## * Rules for Labels in Assembly Language *

1. it must be a unique name
2. it must be consist of alphabetic letters in both (upper & lowercase), (digits [0-9]), (Special char. $, @, _, .)
3. The first character of the label must be an alphabetic letters.
4. Not a reserved word.

[ما بصير يكون رقم أول شي]

---

## Review Questions

1. Give 3 ways for hex data representation?
2. Show how to represent decimal 99 in [hex, decimal, binary]?
3. What's the advantage in using the EQU directive to define a constant value?
4. Show the hex no.'s value used by the following directives:-
   a) ASC_Data EQU A'4'
   b) MY_DATA EQU B'00011111'
5. Give the value in WREG for:

```
MYCOUNT  EQU 15
MOVLW    MYCOUNT
```

6. Give the value in filereg 0X20 for:

```
MYCOUNT  EQU 0X95
MYREG    EQU 0X20
MOVLW    MYCOUNT
MOVWF    MYREG
```

7. Give the value in filereg 0X63 for:

```
MYDATA   EQU D'12'
MYREG    EQU 0X63
FACTOR   EQU 0X10
MOVLW    MYDATA
ADDLW    FACTOR
MOVWF    MYREG
```

**Answers:**

① DATA1 EQU 9FH
DATA2 EQU 0X9FH
DATA3 EQU H'9F'

② DATA1 EQU 99F
DATA2 EQU D'99'
DATA3 EQU B'10011001'

③ If the value is to be changed later, it can be done once in one place instead of at every occur-rence.

④ a. 34H  b. 1FH

⑤ WREG = 15H

⑥ value of location 0X20 = (0X95)

⑦ 0CH + 10H = 1CH will be in filereg location 63H
(0CH = $12_{10}$)

Scanned with CamScanner

## 2.6 Introduction to PIC assembly language

→ difficult for us to deal with machine code (0s and 1s)

→ eventually, assembly languages were developed, which provides:

- mnemonic: for the machine code instructions and abbreviations that are easy to remember
- faster programming and less prone error.

→ assembly languages referred to as a low-level-language (LLL): because it deals directly with the internal structure of the CPU.

→ Assembler: program used to translate the assembly code into machine code (object or opcode)

to program in assembly language, the programmar must know all the details about registers of the CPU.

Compilers: programs that used to translate the High level language programs to machine language.
C++, Java, Basic, Pascal, C

---

# Structure of assembly language →

* Assembly language: is a series of statements (or lines) → instructions, → directives

→ it consists of four fields:

[label] mnemonic [operands] [;comment] ← optional field ⇐ الأقواس تدل على

- [label]: it allows the program to refer to a line of code by name.
- mnemonic [operands] (أشياء بتتنفذ): they are perform the real work of the program and accomplish the tasks for which the program was written.
- [;comment]: it maybe at the end of a line or on a line by themselves (assembler ignores it, but they're indispensable to programmers)

```
MOVLW 55H
ADDLW 67H
```

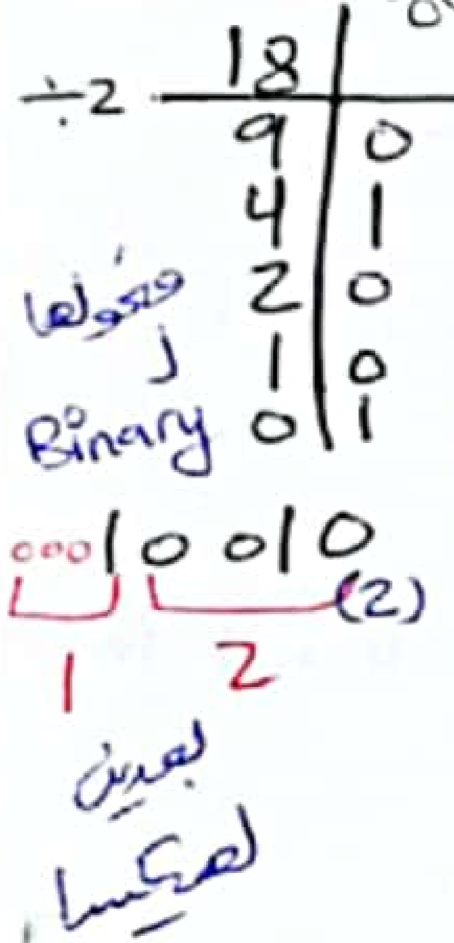
MOVLW, ADDLW → mnemonics that produce opcodes; 55H, 67H → operands

الهدف منها
جعل البرنامج أسهل للقراءة والفهم
كالوصفة أو أمر معين يبغى توضيحه



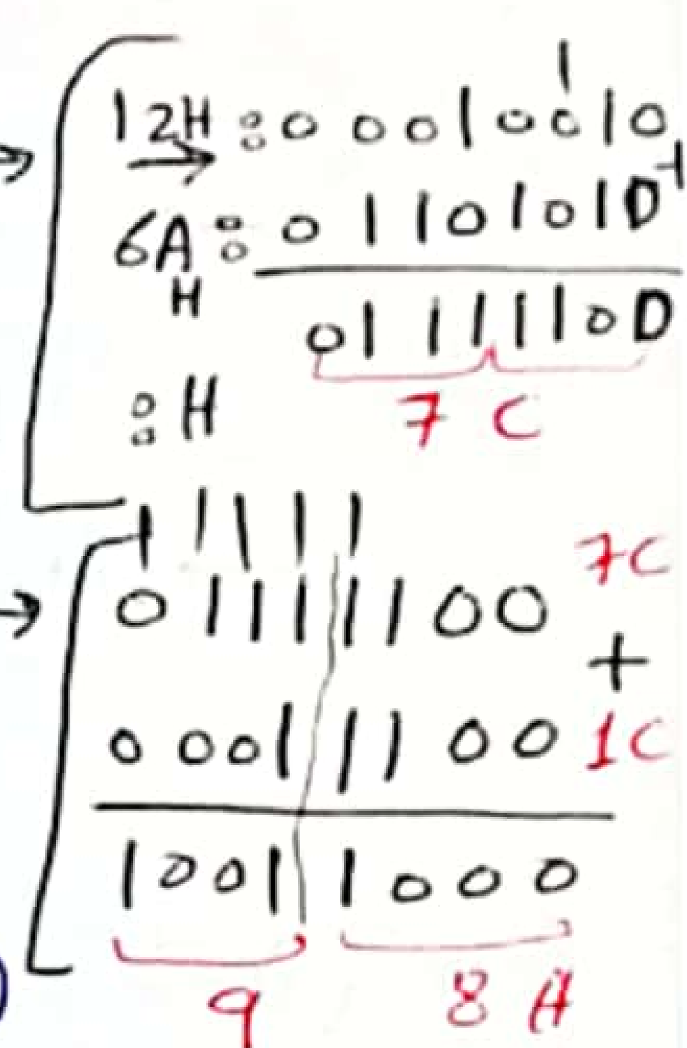
Scanned with CamScanner

## * Sample of Assembly language program :-



```
                          directive
SUM     EQU     10H       ; RAM loc 10H for SUM (SUM يعني 10H)
        ORG     0H        ; Start at address 0
        MOVLW   25H       ; WREG = 25H
        ADDLW   0x34      ; WREG = 25H + 34H = 59H
        ADDLW   11H       ; WREG = 59H + 11H = 6AH
        ADDLW   D'18'     ; WREG = 6AH + 12H = 7CH
        ADDLW   1CH       ; WREG = 7CH + 1CH = 98H
        ADDLW   B'00000110' ; WREG = 98H + 06H = 9EH
        MOVWF   SUM       ; Save the result in SUM Location
HERE    GOTO    HERE      ; Stay here forever     10H [9EH]
        END
```

HERE → Label; HERE → address; loop to itself (for executing for one time)



لو كانت MOVLW SUM

[WREG = SUM = 10H]

مو 9EH

* Reset (ازا بدنا نعيد أول لوب Loop)

---

## Review Questions

1. What is the purpose of pseudo_instructions? the real work is performed by instructions such as MOV and ADD, instruct the assembler to do something

2. The instruction (mnemonics) are translated by the assembler into machine code (assembly code), whereas pseudo_instructions are not.

3. Assembly language is a High level language ___F___ (True or false)

4. which of the following instructions produces opcode? list all that do
   a) MOVLW 25H   b) ADDLW 12   c) ORG 2000H   d) GOTO HERE
   (a, b, d) — machine code (instructions for operation)

5. Pseudo_instructions are also called assembly directives

6. Assembler directives aren't used by the CPU itself. They're simply a guide to the assembler ___T___ (True or False)

7. In Question 4, which one is an assembler directive? C

---

# 2.7 Assembling and linking a PIC program

## # Steps to create a program →

- **Editor program** — for the text (البرنامج السابق)
  - ↓ myfile.asm — source file containing the program code creating in step 1 is fed to the PIC assembler
- **assembler program** — converts the instructions into machine code
  - → myfile.err — error file; contains any syntax errors and their line no.'s
  - ↓ myfile.o — object file
- **Linker Program**
  - ← .lib additional library files
  - ← .o additional object files
  - ← .lkr linker script file
  - Outputs:
    - myfile.out — intermediate object file
    - myfile.cod — debug file
    - myfile.hex → download TO PIC's ROM
    - myfile.map — map file
    - myfile.lst — list file

> بقدر أكتشف ال (Logical Error) لما بعمل debugging للبرنامج وبرمج تنفذ قبالي سطر بسطر

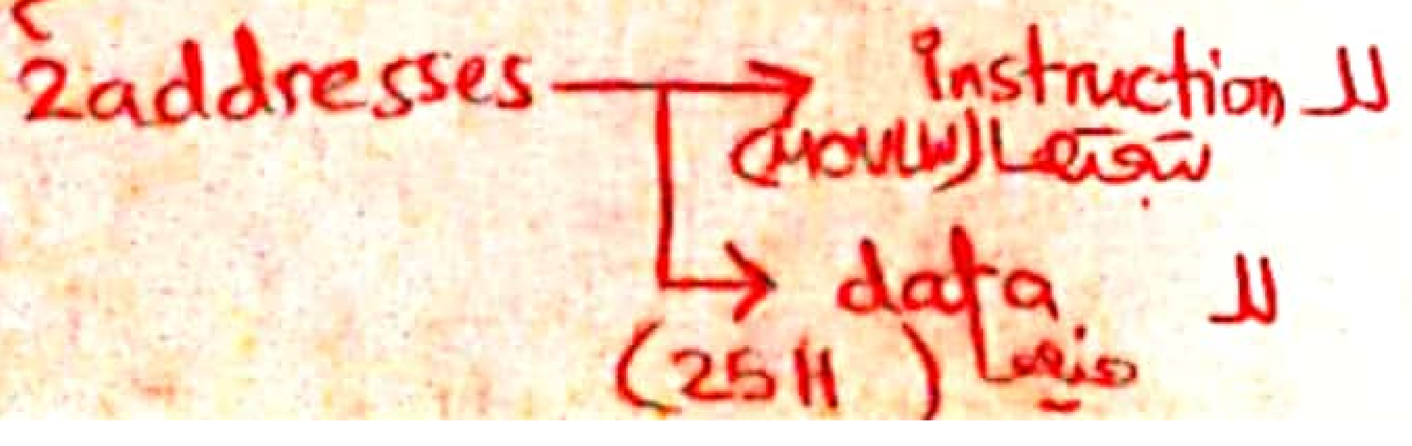
## List file

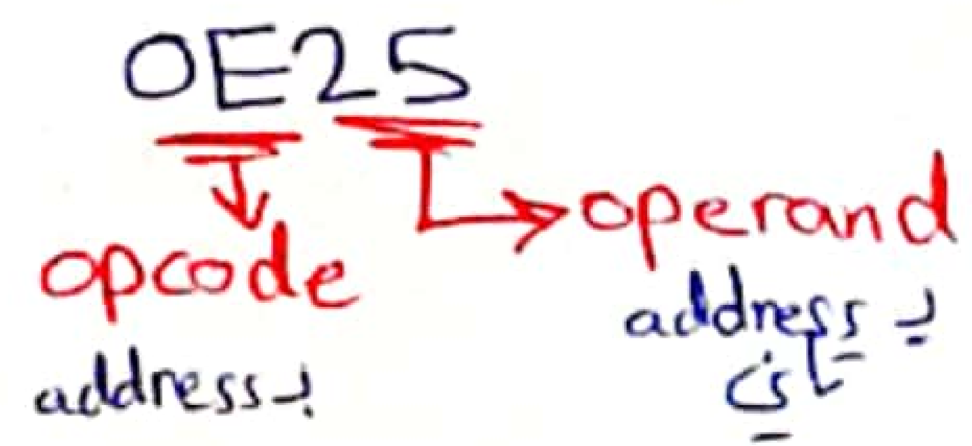
| LOC OBJECT CODE VALUE | | LINE | SOURCE TEXT |
|---|---|---|---|
| 00000010 | | 00001 | SUM EQU 10H , SUM in 10H |
| 000000 | | 00002 | ORG 0H |
| 000000 | 0E25 | 00003 | MOVLW 25H |
| 000002 | 0F34 | 00004 | ADDLW 0x34 |
| 000004 | 0F11 | 00005 | ADDLW 11H |
| 000006 | 0F12 | 00006 | ADDLW D'18' |
| 000008 | 0F1C | 00007 | ADDLW 1CH |
| 00000A | 0F06 | 00008 | ADDLW b'00000110' |
| 00000C | 6E10 | 00009 | MOVWF SUM |
| 00000E | EF07 F000 | 00010 | HERE GOTO HERE |
| | | 00011 | END → end of asm source file |

- For MOVLW → 0E25
- For ADDLW → 0F11 (the opcode & operand for ADDLW 11H)
- For MOVWF → 6E10
- 0F34: located in memory location 0002 (0F) (content) 0003 (34H)

> لما سوي للبرنامج هاد download على ال RAM بتحول لل (Binary) كلو ما بيظل بال (Hexa)

الأرقام زوجية هنا لأنه كل وحدة منهم إلها





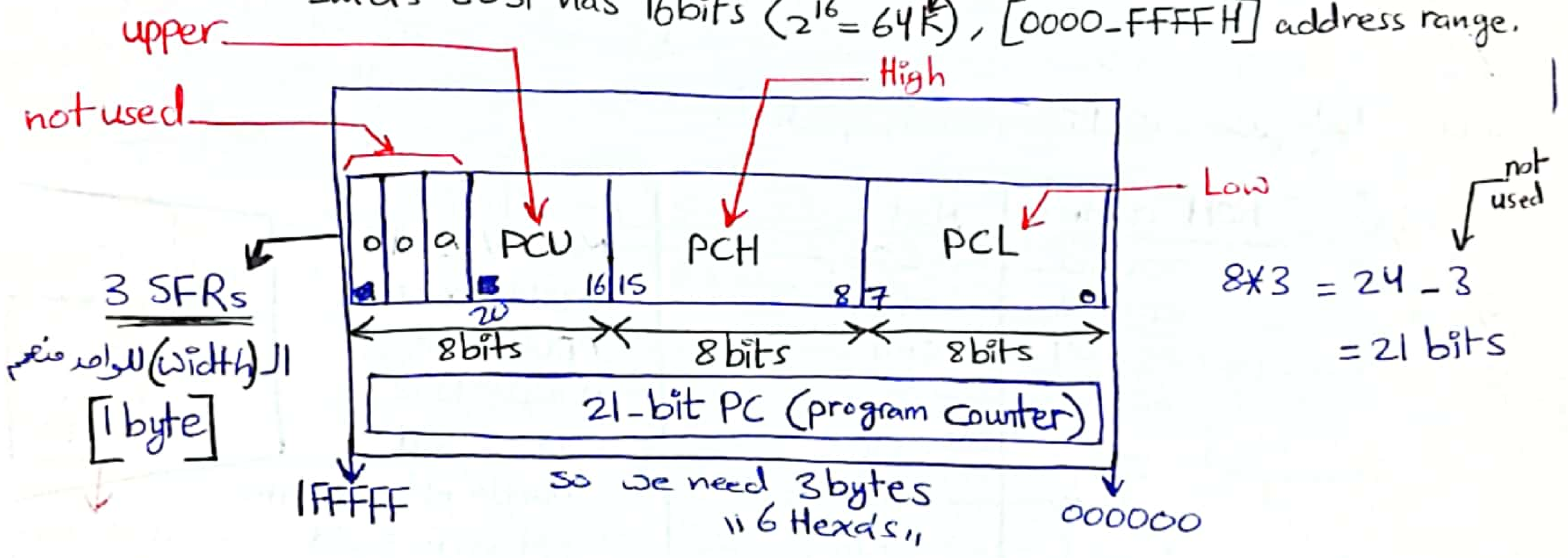

Scanned with CamScanner

## 2.8 The Program Counter and program ROM space in the PIC

* Program counter in the PIC → "instruction pointer"
(PC) is an important register in the PIC MC, which is used by the CPU to point to the address of the next instruction to be executed.

→ The wider the program counter, more the memory Locations a CPU can access.

- that means that a 14 bit PC can access a max. of 16 K ($2^{14} = 16K$) of code from addresses (0000-3FFF)H
- PIC18 has 21 bits ($2^{21} = 2M$) ← max.
- Intel's 8051 has 16bits ($2^{16} = 64K$) ← on a chip ROM, [0000_FFFFH] address range.




# ROM memory map in the PIC18 family →

**Ex (2.11)** Find the ROM memory address of each of the following PIC chips:-

a) PIC18F2220 with 4KB
b) PIC18F2410 with 16KB
c) PIC18F458 with 32KB

لو ما كانت معطيني عناوين بالحجم بشوف كم 1's و بفصلها أرباع

→ (a) $4K = 4*1024 = 4096$ bytes

000000 ← byte → PIC18F2220, 000FFF

000FFF
− 000000
000FFF
000111111111111
$3*4 = 12$ one's → $2^{12} = 4K$

(12 bits)

(b) 000000 ← byte → PIC18F2410, 003FFF

3FFF
00 0011111111111111
$2^{14} = 16,384$ bytes
= 16K bytes

(14 bits)

(c) 000000 PIC18458, 007FFF

6 Hexa's = 3 bytes
فيرة كل byte بعبر عن 2 Hexa's

000111111111111111
$2^{15} = 32KB$

(15 bits)



Scanned with CamScanner

# Powering up the PIC :-

- At what address does the CPU wake up when power applied?
  - The mc wakes up at memory address 0000
  - The PC has the value 0000
  - ORG directive put the address of the first opcode at the memory location 0000

PIC18 Program ROM Space

| Address | Contents |
|---|---|
| 000000H | Reset Vector |
| 000008H | High Priority interrupt vector |
| 000018H | Low priority interrupt vector |
| | On-chip program memory |
| | External/unimplemented program memory (Read as '0' in mc mode) |
| 1FFFFFH | |

---

# Placing Code in program ROM :- تقرأ كلها

الصفحة 25 في برنامج مخطوط على الجهة اليسار منه (even addresses)
الوحدة منها مرتبة معها [operand+opcode] كل وحدة منهم إلها address

من البرنامج 25 ص

| ROM address | Machine Language | Assembly language |
|---|---|---|
| 00000 | 0E25 | MOVLW 25H |
| 00002 | 0F34 | ADDLW 0X34H |
| 00004 | 0F11 | ADDLW 11H |
| 00006 | 0F12 | ADDLW D'12' |
| 00008 | 0F1C | ADDLW 1CH |
| 0000A | 0F06 | ADDLW B'00000110' |
| 0000C | 6E10 | MOVWF SUM |
| 0000E | EF07 F000 | HERE GOTO HERE |

EF07 = 2byte, F000 = 2byte → 4 Hexa's

all instructions are 2bytes except the GOTO, which has 4 byte → each one takes two Memory Locations

## Executing a program byte_by_byte

1. When the PIC is powered up, the PC starts to fetch the first opcode from location 000000 (0E) which is the code for moving an operand (25) to WREG.
   وضع لل MOVLW
   فاول ما يتنفذ بتروح 25H لل WREG من قبل CPU
   الـ (0E)opcode
   Then the PC is incremented to point 000002 which contains opcode 0F for instruction (ADDLW 34H)
2. Upon executing the opcode 0F, the value 34H is added to WREG. Then the PC is incremented to 000004
3. ROM location 0004 has the opcode for instruction (ADDLW 11H). This instruction is executed and now PC=0006
4. This process goes on until all the instructions up to "MOVWF SUM" are fetched and executed.

ROM Contents

| address | code |
|---|---|
| 000000 | 0E |
| 000001 | 25 |
| 000002 | 0F |
| 000003 | 34 |
| 000004 | 0F |
| 000005 | 11 |
| 000006 | 0F |
| 000007 | 12 |
| 000008 | 0F |
| 000009 | 1C |
| 00000A | 0F |
| 00000B | 06 |
| 00000C | 6E |
| 00000D | 10 |
| 00000E | 07 |
| 00000F | EF |
| 000010 | 00 |
| 000011 | F0 |



Scanned with CamScanner

∴ Now PC = 000E points to the next instruction, which is "GOTO HERE".

This is a 4 byte instruction.

It takes ROM addresses of 0E, 0F and 10, 11

[لعيد التشغيل]

PC = 000E → keeps the program in an infinite loop.

---

# ROM Width in the PIC18 ⇒

(ROM) الي بنكتب عليه الـ(code) عليه access أكثر من الـ(RAM)

→ A microprocessor's memory that holds code is byte-addressable and under the control of the program counter.

that means: each location holds only one byte.

→ If we have 16 address lines, it will give us $2^{16}$ ⇐ يعني locations, which is 64K of memory space with an address map [0000-FFFFH]

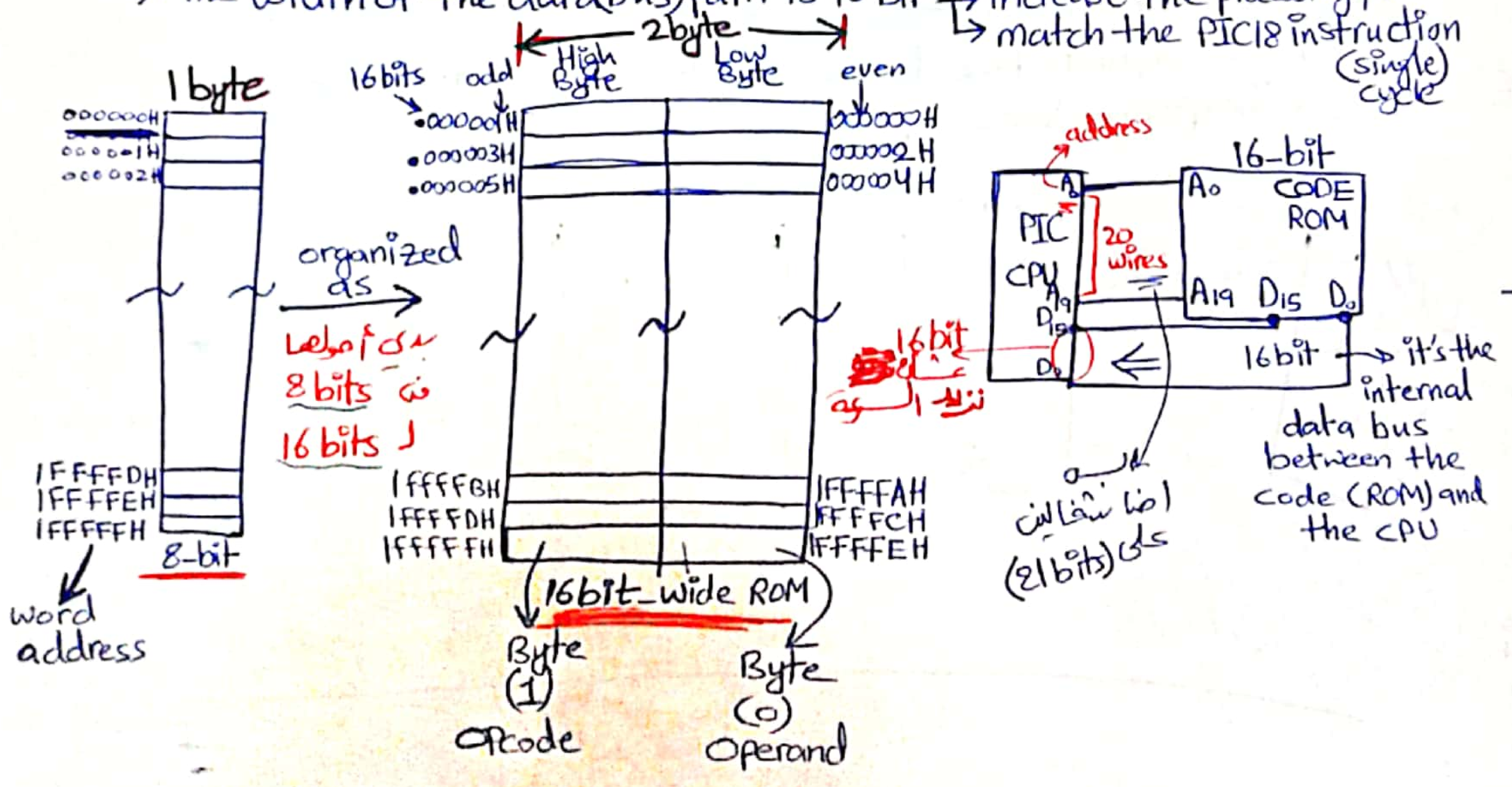
→ CPUs with 8 bits data will fetch one byte at a time.

to bring in more information (code or data) into the CPU we can increase the width of the data bus to 16 bits.

→ The databus is like traffic lanes on the high-way where each lane is 8 bits wide.

the more lanes, the more info. we can bring to CPU for processing.

→ The width of the data (bus) path is 16 bit → increase the processing power
↳ match the PIC18 instruction (single cycle)



# Little endian VS big endian War :-

- The low byte goes to the low memory location
- The high byte goes to the high memory address

* Intel mp and many mc's use little endian.

PIC18 program ROM contents
program (2-1) list file

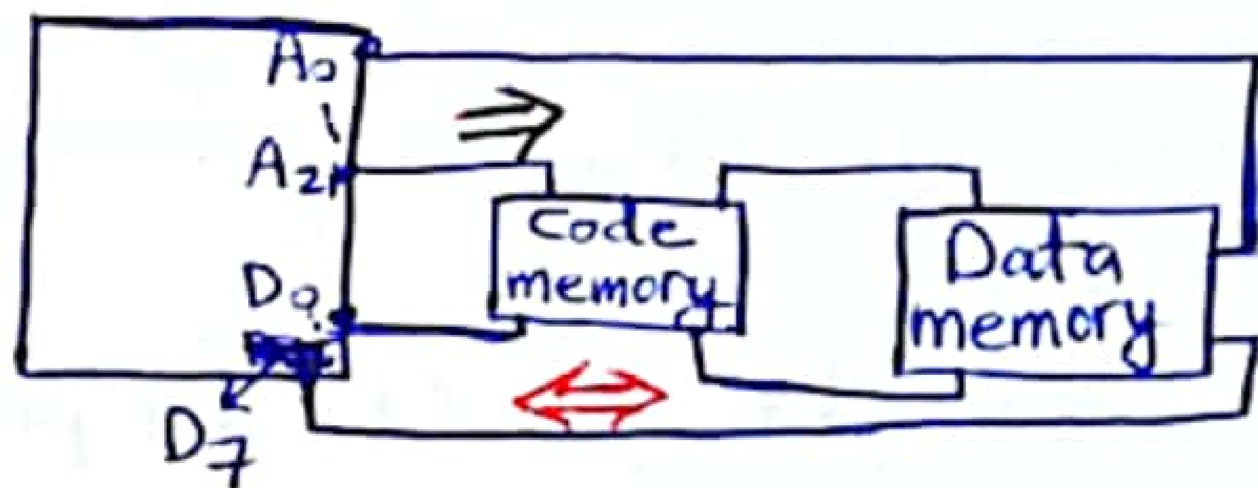
| word address | high byte | low byte |
|---|---|---|
| 000000H | 0EH | 25H |
| 000002H | 0FH | 34H |
| 000004H | 0FH | 11H |
| 000006H | 0FH | 12H |
| 000008H | 0FH | 1CH |
| 00000AH | 0FH | 06H |
| 00000CH | 6EH | 10H |
| 00000EH | EFH | 07H |
| 000010H | 0FH | 00H |

# Harvard Architecture in the PIC :-

(*) Von Neumann architecture →

It uses the same bus for accessing both the code and data memory.

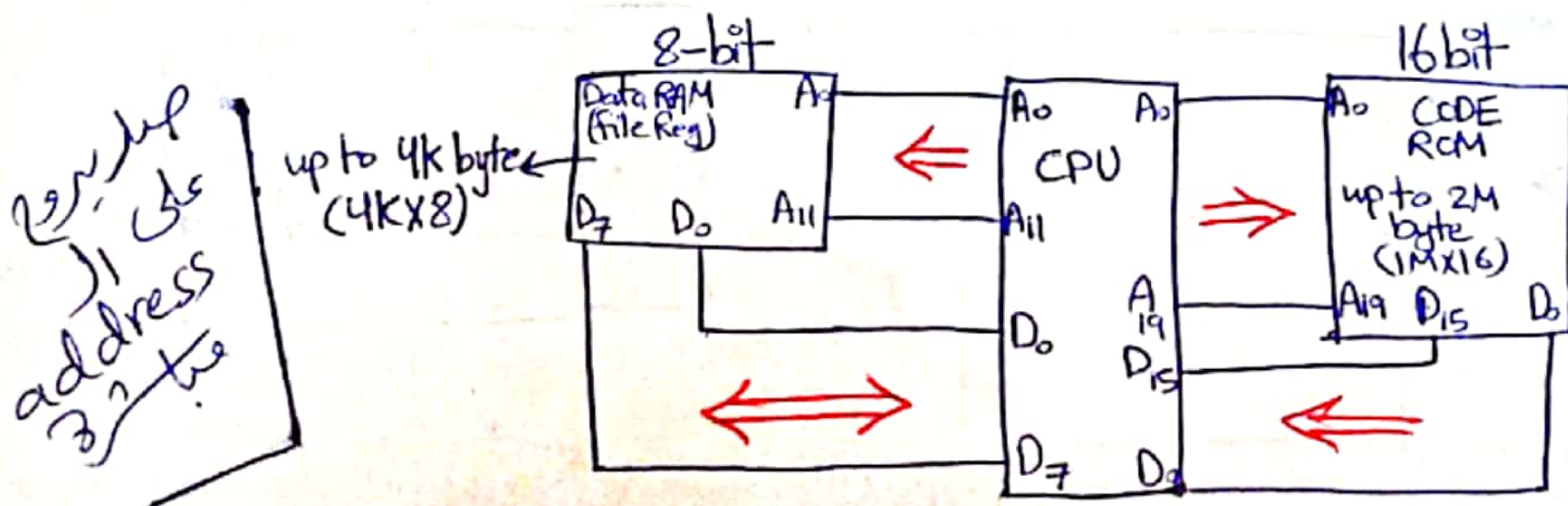
- slow down the processing speed.
- Get in each other's way.




(*) Harvard Architecture →

It uses separate buses for accessing the code and data memory.

- inexpensive for a chip.



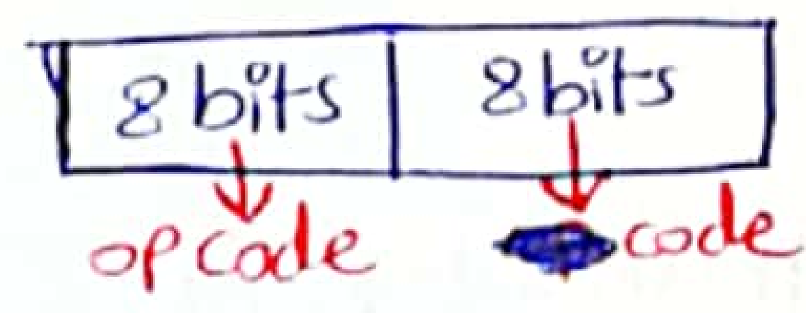
# Instruction Size of the PIC18 →

We talk before that PIC18 program memory is byte-addressable. and the instructions → 2byte (for almost all instructions) ← حاليا

→ 4 byte (for MOVFF, GOTO, ..)

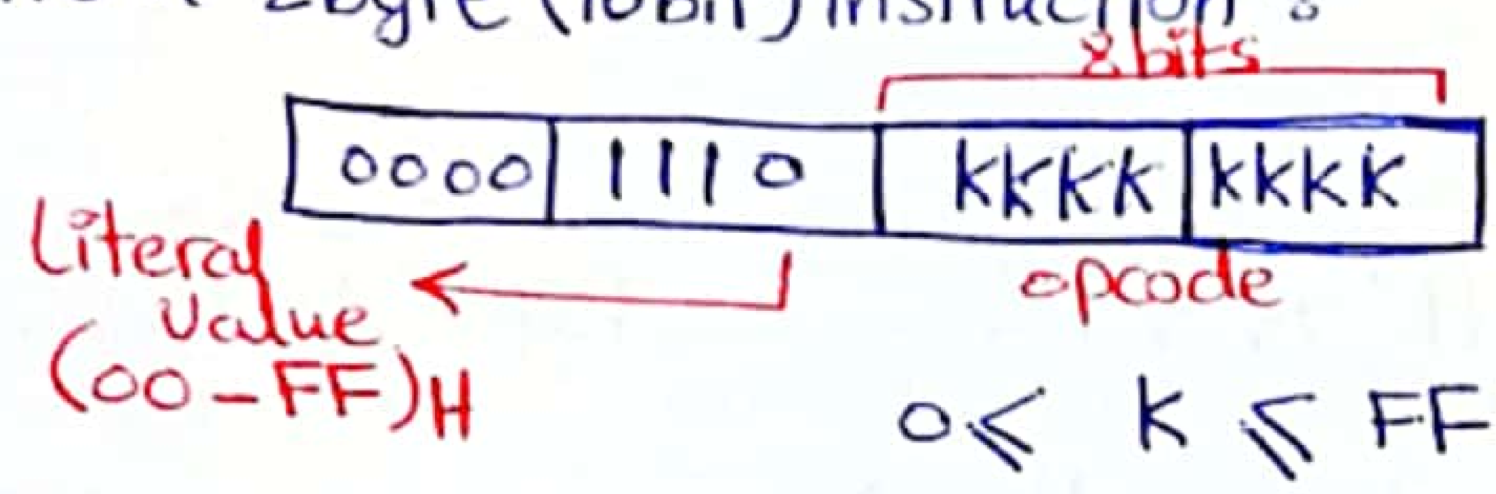
## 2byte (16bit) instructions

The first seven or eight bits (op-code) :




① MOVLW →

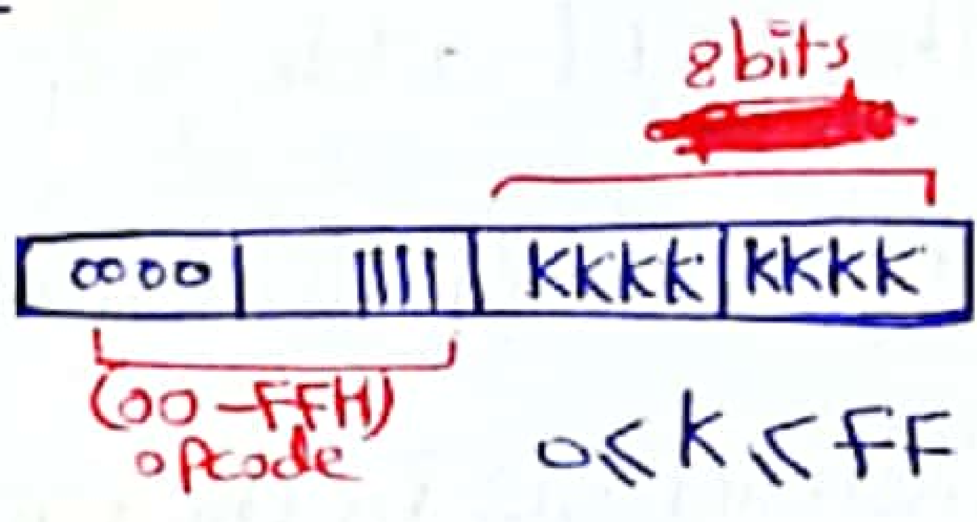
it's a 2byte (16bit) instruction :




$0 \leq K \leq FF$

② ADDLW →

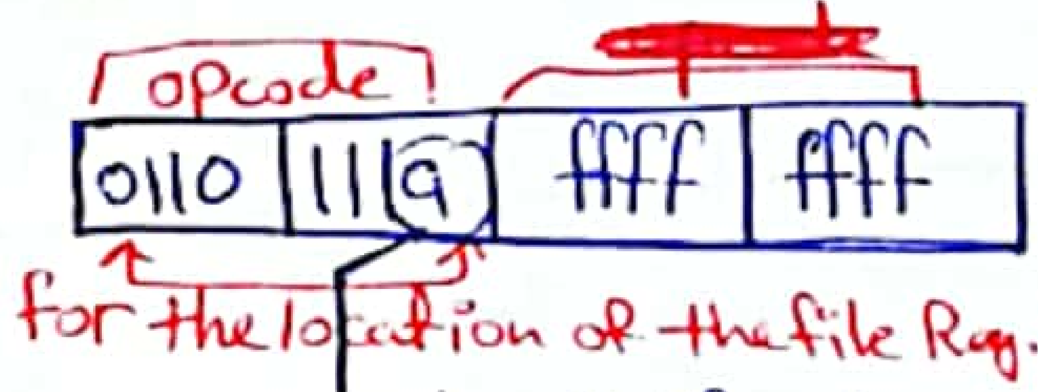
it's a 2byte (16bit) instruction :




$0 \leq K \leq FF$

③ MOVWF →

it's a 2byte (16bit) instruction :



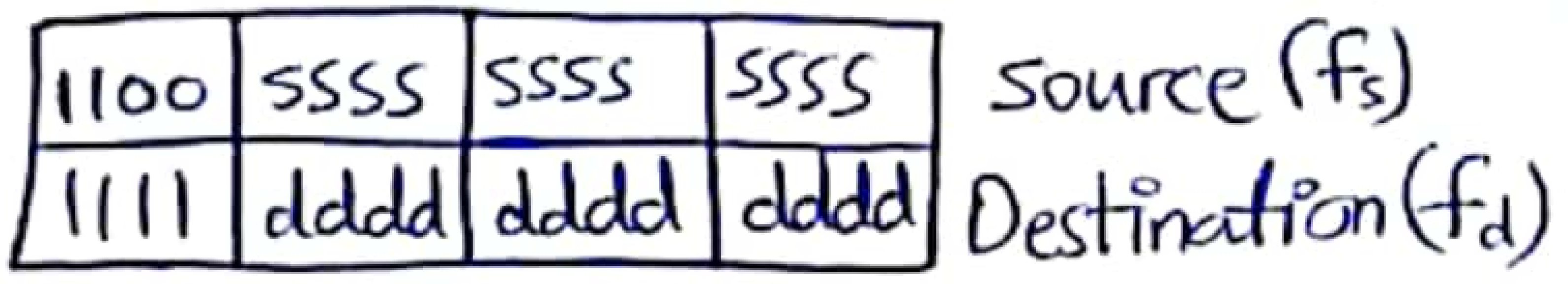

for the location of the file Reg. in the data RAM

→ to signify the access from the access bank or the other bank in the 4096 location

a = 0 , the file Reg. is in the access bank
a = 1 , we have to use bank-switching and the access bank is specified by BSR

## 4 bytes (32 bit) instructions

① MOVFF → move data within RAM, which is 4K)

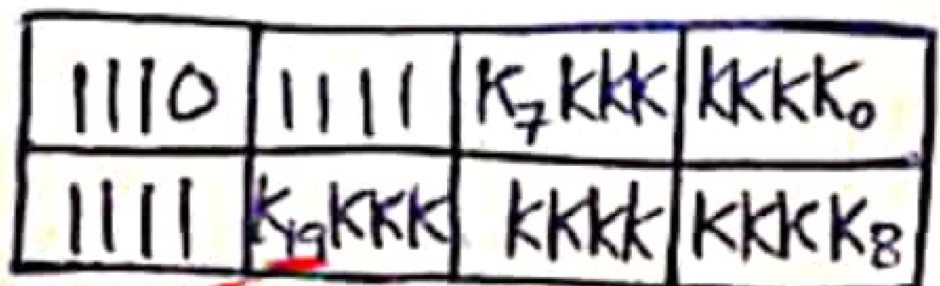



→ each of them, 12 bits are used for the file Reg. address of the PIC18.

$0 \leq f_s \leq FFF$

$0 \leq f_d \leq FFF$

② GOTO → (the code address bus width is 21, which is 2M)




$0 \leq K \leq FFFFF$

even address

20 address



Scanned with CamScanner

## 2.9 RISC Architecture in the PIC

* To increase the processing power of the CPU :-

1. increase the clock frequency of the chip.
2. Use Harvard architecture
3. Change the internal architecture of the CPU and use what is called RISC Architecture

| RISC | CISC |
| --- | --- |
| * Simple and small instruction set | * complex and large instruction set. |
| * Regular & fixed instruction format | * Irregular instruction format |
| * Simple address modes | * complex address mode |
| * Pipelined instruction execution → 95% executed in one cycle | * May also pipeline instruction execution. |
| * Provide large no. of CPU Reg. | * provide smaller no. of CPU Reg. |
| * Separated data & program memory | * Combined data and program memory |
| * most operations are reg. → reg. | * Most operations can be register to memory |
| * Take shorter time to design & debug. | * Take longer time to design & debug. |

Scanned with CamScanner

# PIC [Ch.3]

## # Branch, Call and time delay Loop ⇒

In the sequence of instructions to be executed, it's often necessary to transfer program control to a different location.
There are many instructions in PIC to acheive this.
→ this chapter covers the control transfer instructions available in PIC assembly language.

→ هندرس هالشابتر عن ال instructions اللي بتنقل ال data عبر مواقع مختلفة (for transfering program counter)

## 3.1 Branch instructions and looping:

In this section we first discuss how to perform a looping action in PIC then the branch (jump) instructions, both conditional and unconditional.

### Looping in PIC

Loop: Repeating a sequence of instructions or an operation a certain no. of times.

ex

```
MOVLW 0    ; WREG=0
ADDLW 3    ; WREG=3
ADDLW 3    ; WREG=6
ADDLW 3    ; WREG=9
ADDLW 3    ; WREG=12 (0CH)
ADDLW 3    ; WREG=15 (0FH)
```

⇐ why we need a loop?

لحتى أكرر هاي العملية (اضافة 3 لل WREG) 5 مرات إحنا بحاجة إلى [too much code space] لهيك منستخدم اللوب

### * Two ways to do a loop in PIC ?

1. Using DECFSZ instruction
2. Using BNZ/BZ instruction

→ لازم يكونوا موجودين باللوب لحتى ينقصوا من قيمة العداد ليوصل للصفر فنطلع برا اللوب لأنه بتكون خلصت تنفذت
ما وصل للصفر يرجع للّوب

① DECFSZ Instruction

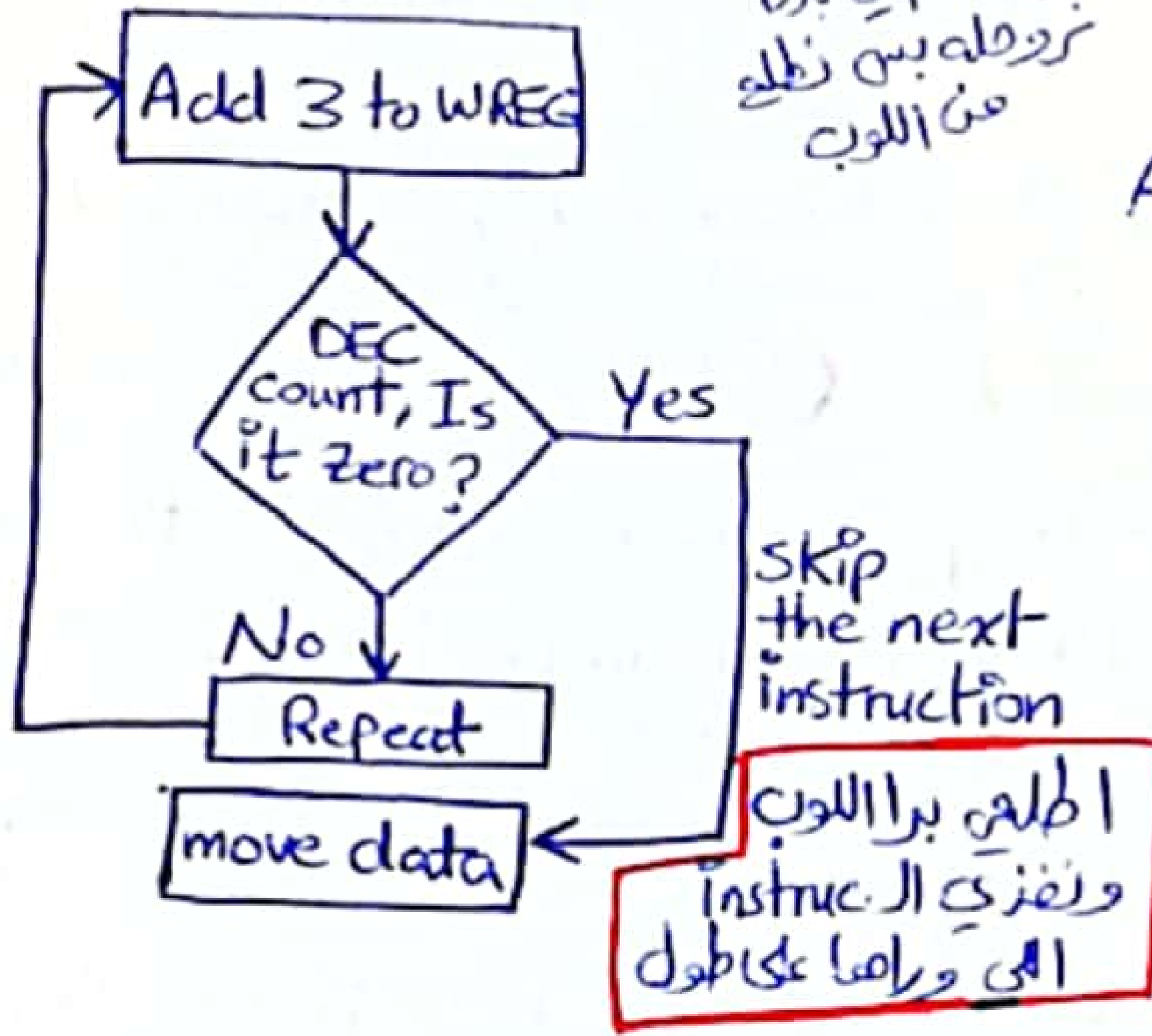
[decreament file reg. skip zero]

أطلع برا اللوب لما الـ fileReg بيصير zero
(لما عداد اللوب يخلص)

- It has the following format :-

DECFSZ fileReg. , d → decremeant fileReg. and skip next instruction if 0

-




المكان الي بدنا نروحله بس نطلع من اللوب

Instructions

```
AGAIN  ADDLW 3

       DECFSZ COUNT

       GOTO AGAIN      (Repeat)
       MOVWF PORTB     (Move data) →
```

اذا كانت Count = 0

Ex(1) Write aprogram to a) Clear WREG b) add 3 to WREG ten times and place the result in SFR of PORTB.

- Use the DECFSZ instruction to perform Looping ?

قيمة العداد → ten ; → (f)

a)

b)+

```
;this program adds 3 to WREG ten times
COUNT  EQU    0X25      → 25H [location] ← COUNT مطّينا للـ (بيمثل العدّاد)
       MOVLW  d'10'
       MOVWF  COUNT     → [10, decimal] data مطّينا فيه
       MOVLW  0         → Clear WREG
AGAIN  ADDLW  3         → عملنا لوب
       DECFSZ COUNT,f   → عملنا DECFSZ للـ COUNT
       GOTO   AGAIN
       MOVWF  PORTB
```

حطط العداد بموقع معين ← COUNT

ضفنا على العداد الرقم الي بيمثل كم مرة بدي أحاوي هاد اللوب ← MOVLW / MOVWF

ازا ما تحقق DECFSZ ← GOTO AGAIN بعيد اللوب [إضافة 3 الـ WREG]

ازا تحقق COUNT = 0 → MOVWF PORTB

والقيمة تتخزن بالـ f لأنه

[DECFSZ COUNT, w إزا حطينا
ما رح نطلع من اللوب بوقتها]

لأنه الـ count

الي بالـ fileReg

رح تضل قيمته 10 ← تقلل منه وبحط بالـ WREG

ويرجع يضيفله ٣

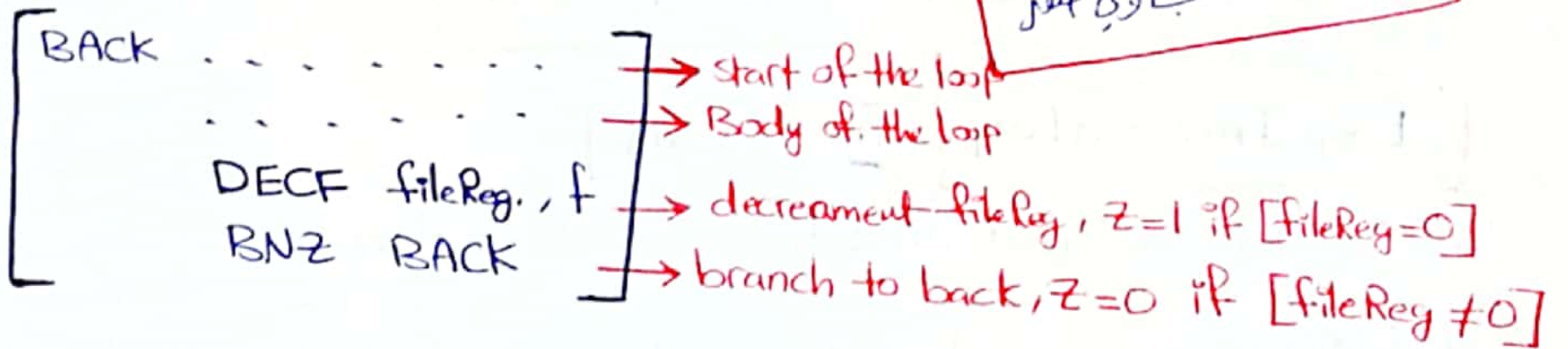
اذن ما منطلع من اللوب.

## ② BNZ/BZ Instructions

[branch if not zero] / [branch if zero]

> يدل ال GOTO باللوب لأنها بترجعك عبداية اللوب ازا كان ال DECF اللي قبلها ما بساوي صفر

* it uses the zero flag in the status register
* the BNZ instruction is used as follows ⇒




**Ex(2)** Write a program to a) Clear WREG, then b) add 3 to WREG ten times. Use the zero flag and BNZ.

```
COUNT   EQU    0X25
        MOVLW  d'10'
        MOVWF  COUNT
        MOVLW  0
AGAIN   ADDLW  3
        DECF   COUNT, f
        BNZ    AGAIN
        MOVWF  PORT B
```

لأنه مطلوب بالسؤال نعيد 10 مرات (COUNT = 10 "decimal")

DECF COUNT, f → fileReg = 0 / fileReg ≠ 0

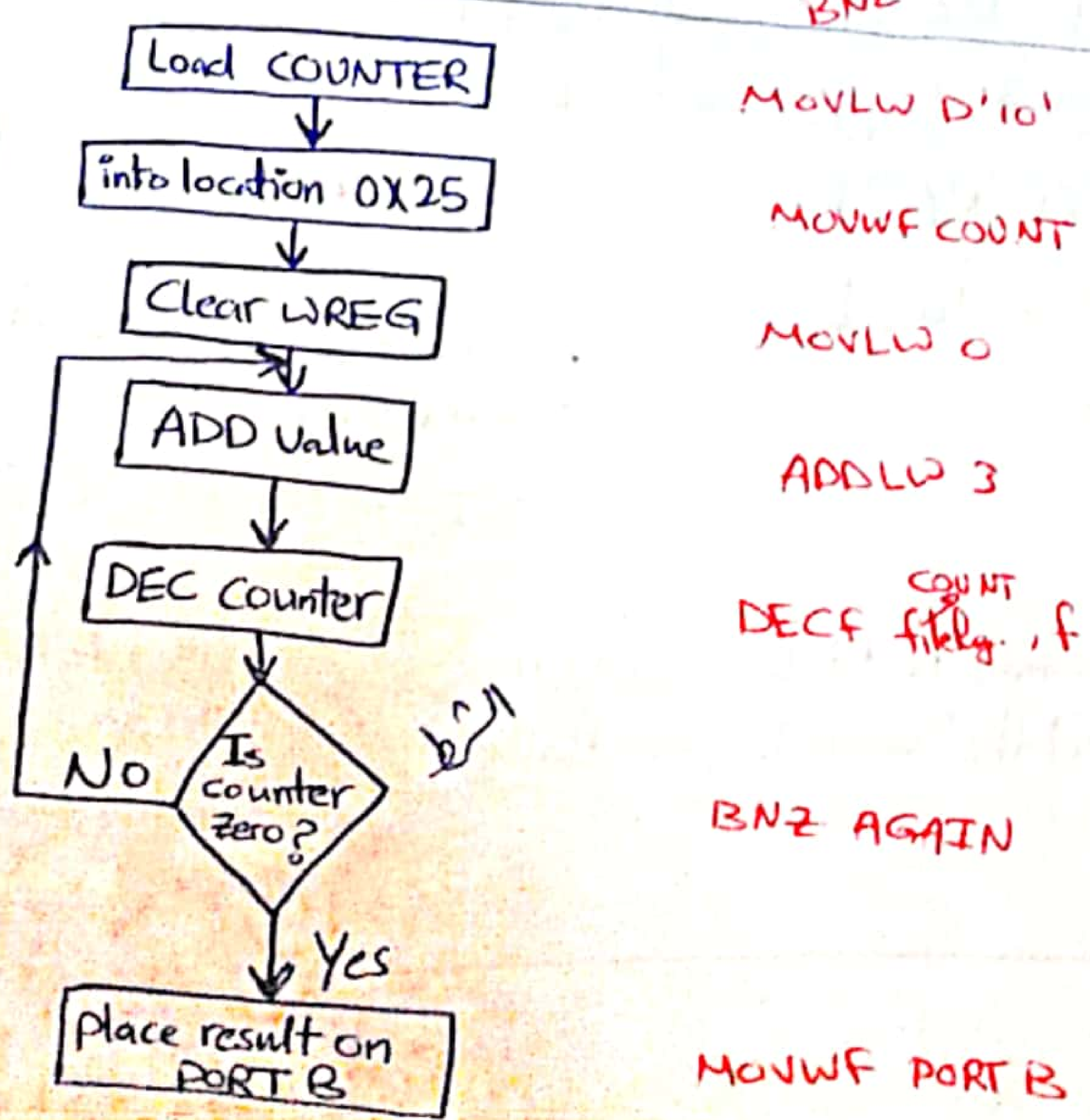
الشرط للوب — BNZ

بترجع للبداية

بتطلع من اللوب لأنه اللي برجعك إلها BNZ AGAIN

منعرف كم اللوب قيمته من خلال القيمة اللي فاتت على ال COUNT من ال WREG وازا كان في أكثر من قيمة بنضربهم ببعض عشان نطلع ال total

### Flowchart for EX(2)




MOVLW D'10'

MOVWF COUNT

MOVLW 0

ADDLW 3

DECF fileReg., f (COUNT)

BNZ AGAIN

MOVWF PORT B

Scanned with CamScanner

**Ex(3)** What's the max. no. of times that the loop in [ex(2)] can be repeated?

→ because location COUNT in fileReg. is an [8 bit reg.] it can hold a max. of 255_0FFH (COUNT = 255 'decimal') max.

∴ The max. size of loop is 255 times.

---

## Loop Inside a Loop [nested loop]

If we want to repeat an action more times than 255.

**Ex(4)** Write a program to: a) load the PORTB SFR Reg. with the value 55H, and b) complement PORTB 700 times?

↳ 70 * 10 (بدالي) — 70 < 255, 10 < 255

[700 > 255 صار بحتاج ل (two Reg.) لنعمل هاد البرنامج]

```
R1      EQU   0X25        → two locations for COUNT1 and COUNT2
R2      EQU   0X26
COUNT1  EQU   d'10'       → two data's for COUNT1 and COUNT2
COUNT2  EQU   d'70'
        MOVLW  0X55       → WREG = 55H
        MOVWF  PORTB      → PORTB = 55H
        MOVLW  COUNT1     → WREG = 10 (decimal)
        MOVWF  R1         → load 10 into location 25H (outer loop count)
loop1   MOVLW  COUNT2     → WREG = 70₁₀                     ] nested
        MOVWF  R2         → R2 = 70₁₀ (fileReg. بال)        ] Loop
Loop2   COMPF  PORTB,f    → complement PORTB, SFR
        DECF   R2, F      → dec. fileReg. (inner loop)
        BNZ    Loop2      → repeat it 70 times
        DECF   R1, F      → dec. fileReg (25H) (outer loop)
        BNZ    loop1
```

BNZ Loop2 ← لما تكون $R_2 = 0$ بتروح لـ DECF R1, F لتنفذها

BNZ loop1 → repeat it 10 times

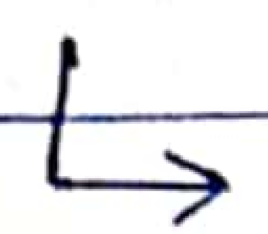
MOVLW COUNT1 → WREG = d'10'
MOVWF R1 → R1 = d'10'

[file Reg.] memory location

| Address | Data |
|---|---|
| (R1) 025 H | $10_{10}$ |
| (R2) 026 H | $70_{10}$ |

(R2) 026 H → used to keep the inner loop count

∴ this process will continue until location 25H becomes zero and the outer loop is finished.



Scanned with CamScanner

→ If we want to make a loop repeating 100,000 times :

$255 * 255 = 65025$

$255 * 255 * 255 = 16$ million [so we use 3 Reg. to obtain 100,000 times loop]

```
R1      EQU   0X1
R2      EQU   0X2          ; assign RAM loc for the R1-R3
R3      EQU   0X3
COUNT1  EQU   d'100'
COUNT2  EQU   d'100'       ; 100*100*10 = 100,000 times
COUNT3  EQU   d'10'          fixed values for ↑
        MOVLW  55 H
        MOVWF  PORTB
        MOVLW  COUNT3
        MOVWF  R3
Loop3   MOVLW  COUNT2
        MOVWF  R2
loop2   MOVLW  COUNT1
        MOVWF  R1
Loop1   COMPF  PORTB, F
        DECF   R1, F        ← address data
        BNZ    Loop1
        DECF   R2, f
        BNZ    loop2
        DECF   R3, f
        BNZ    loop 3
```

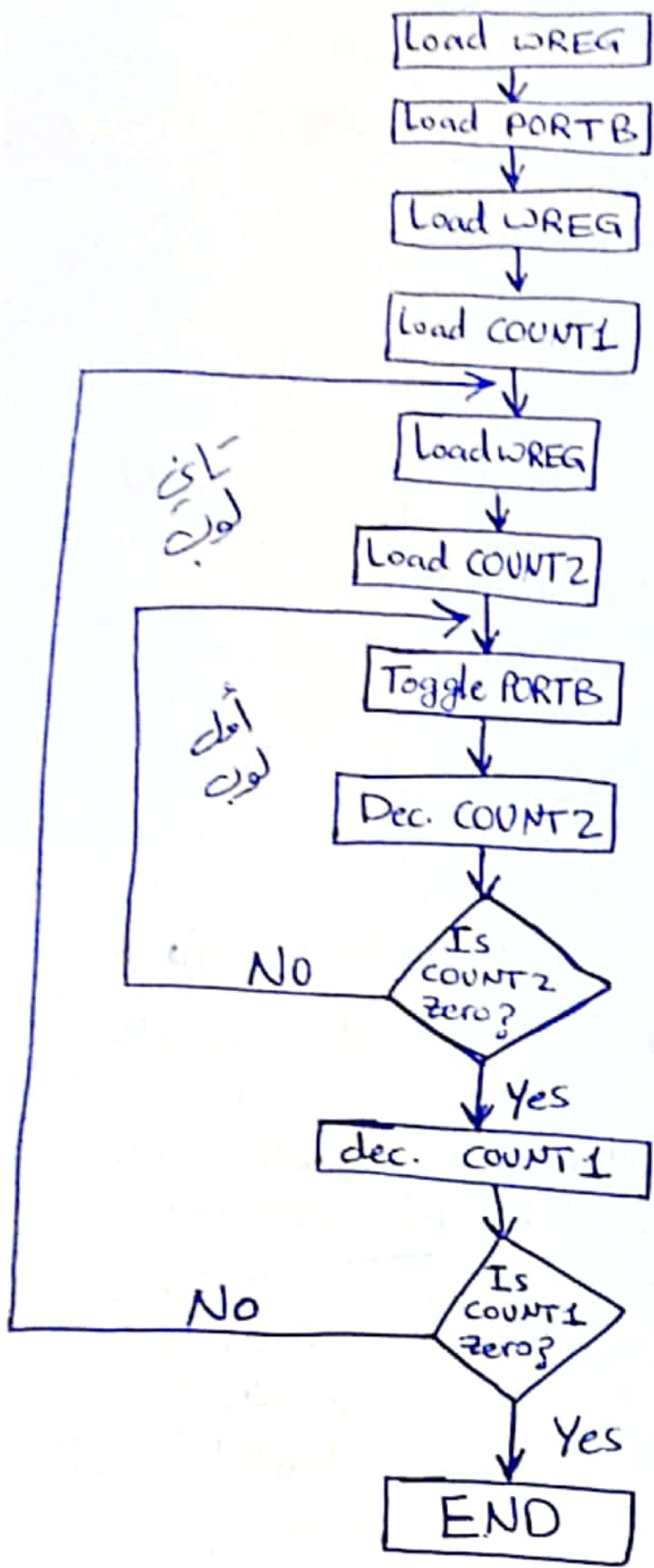
* بال (Loop inside a loop) :

1. منحدد عدد الـ counters والـ loops اللي منحتاجها
2. منحدد الـ address والـ data's للـ counters باستخدام EQU directive
3. منفذ الأمر اللي بدي أسويه اللوب (نقل داتا للـ address اللي بدي أنفذ عليها أمر اللوب)
4. بدخل الـ address تبعت الكاونتر الـ data تبعها من خلال WREG لآخر لوب
5. منفذ أول لوب بعد أول قيمة (تنفيذ نقطة 4)
6. منفذ ثاني لوب اللي تحتوي الشرط (COMP/...) (اللوب المطلوب تكراره)
7. نستخدم DECF للكاونتر الموجود فيه حاليًا أصغر لوب BNZ للوب اللي أنا فيها
8. اذا zero بطلع للوب اللي بعدها ← DECF للكاونتر اللي قبل

BNZ ← للوب اللي قبل

وهكذا

# Flowchart for ex(4)




## Instructions

```
        MOVLW 55H
        MOVWF PORTB
        MOVLW COUNT1
        MOVWF R1
loop1 _ MOVLW COUNT2
        MOVWF R2
loop2 _ COMPF PORTB, f
        DECF R2, f
        BNZ loop2
        DECF R1, f
        BNZ loop1
```

---

# Other Conditional Jumps →

- they jump according to certain condition
- all of them are **2byte** instructions
- They requires the target address
  - 1 byte address (Short branch address)
  - Relative address

**Recall:** MOVF will affect the Status Reg.

### PIC Conditional branch (jump)

| Instruc. | Action |
|---|---|
| BC | branch if C=1 |
| BNC | branch if C≠0 |
| BZ | branch if Z=1 |
| BNZ | branch if Z≠0 |
| BN | branch if N=1 |
| BNN | branch if N≠0 |
| BOV | branch if OV=1 |
| BNOV | branch if OV≠0 |

Scanned with CamScanner

## * BZ (Branch if Z=1) ⇒

- In this instruction, the zero flag (Z) is checked.
- If it's high, it jumps to the target address.

(ex)
```
OVER    MOVF    PORTB,W    ; read PORTB and put it in WREG
        JZ      OVER       ; jump if WREG is zero [بيضل ينط (PORTB=0) لـ OVER]
```

لا تظل مساوية للصفر منطلع من اللوب

**Ex(5)** Write a program to determine if file Reg. location 0X30 contains the value 0. If so, put 55H in it:

```
        MYloc   EQU   0X30
        MOVF    MYloc, F     -> copy MYloc to itself
        BNZ     NEXT         -> branch if MYloc isn't zero (Z=0) flag
        MOVLW   55H          -> put 55H in WREG
        MOVWF   MYloc        -> put 55H if MYloc has zero value.
NEXT    ...
```

هيك منكون عينا هاي الـ address لـ fileReg

اذا كان هو صفر بيروح لـ next اللي بعدها

---

## * BNC (Branch if no carry, C=0) ⇒

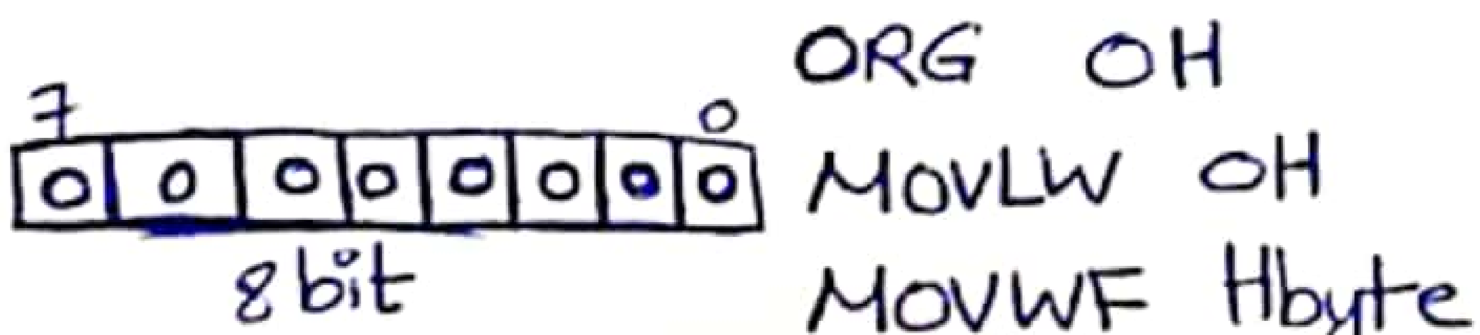
- In this instruction, the carry flag bit used to make the decision whether to jump.
- If C=1, it will not branch but will execute the next instruction below BNC

**Ex(6)** Find the sum of the values 79H, F5H, and E2H. Put the sum in fileReg locations 5 (low byte) and 6 (high byte)? [Using BNC]

(low byte → result without carry bits; high byte → carry bits)

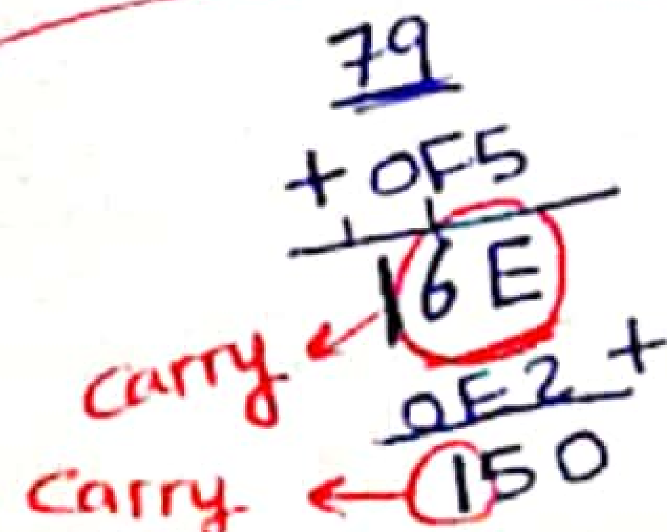
```
→ Lbyte   EQU   0X5
  Hbyte   EQU   0X6      ; assign RAM locations

          ORG    0H
          MOVLW  0H        ; clear (WREG = 0H)
          MOVWF  Hbyte     ; (Hbyte = WREG = 0H) in 06H (address)
          ADDLW  79H
          BNC    N1
          INCF   Hbyte, f      ← ما تحققت
N1        ADDLW  0F5H
          BNC    N2            INCF Hbyte, f ← تحققت
N2        ADDLW  0E2H
          BNC    OVER
          INCF   Hbyte, f      ← تحققت صارت Hbyte (2)
OVER      MOVWF  Lbyte
```




لازم قبل الجمع يكون الصفر عشان الجمع

| | address | data |
|---|---|---|
| (LByte) | 5H | 0 |
| (HByte) | 6H | 0 → 1 → 2 |






Scanned with CamScanner

* All Conditional branches are short jumps → that means the address of the target must be within 256 bytes of the contents of the program counter (PC).

8 bit

Ex(7) Using the following list of ex(6), verify the jump forward address Calculation :-

| Line PC | | Opcode | | Mnemonic Operand |
|---|---|---|---|---|
| LOC | OBJECT CODE VALUE | LINE | SOURCE TEXT | |
| | 00000005 | 00001 | Lbyte EQU 0X5 | ; assign RAM location 5 to Lbyte |
| | 00000006 | 00002 | Hbyte EQU 0X6 | ; " " Loc 6 to Hbyte of sum |
| | | 00003 | | |
| | | 00004 | | |
| 000000 | 0E00 | 00005 | ORG 0H<br>MOVLW 0H | ; Clear WREG (WREG = 0) |
| 000002 | 6E06 | 00006 | MOVWF Hbyte | ; (Hbyte = 0) |
| 000004 | 0F79 | 00007 | ADDLW 79H | ; WREG = 0H + 79H = 79H (C=0) |
| 000006 | E301 | 00008 | BNC N1 | ; while C=0, so go to N1 |
| 000008 | 2A06 | 00009 | INCF Hbyte, F | ; if C=1, inc. (Hbyte = 1) |
| 00000A | 0FF5 | 00010 | N1 ADDLW 0XF5 | ; (C=0) [WREG = 79H + F5 = 6E] C=1 |
| 00000C | E301 | 00011 | BNC N2 (ما تحقق) | → branch if C=0 |
| 00000E | 2A06 | 00012 | INCF Hbyte, F | → C=1 so now [Hbyte = 1] |
| 000010 | 0FE2 | 00013 | N2 ADDLW 0XE2 | → [WREG = 6E + E2 = 50] C=1 for now |
| 000012 | E301 | 00014 | BNC OVER (ما تحقق) | |
| 000014 | 2A06 | 00015 | INCF Hbyte, F | → [Hbyte = 2] now |
| 000016 | 6E05 | 00016 | OVER MOVWF LByte | |
| | | 00017 | END | |

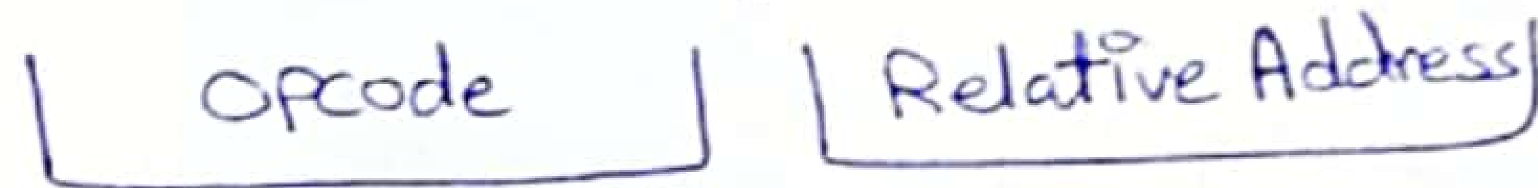
```
         1111   1
79 H →   0111 1001
F5 H + → 1111 0101  +
         ---------
       (1)0110 1110
           6    E

 1 11 1 11
   0110 1110
   1110 0010 +
   ---------
   0101 0000
    5    0
```

فيدة الـ [Carry bits]
بالـ (Hbyte) بـ نستخدم (INCF) if there's a carry
والناتج بالـ (Lbyte) (using BNC)

* Calculating the short branch address ⇒

All conditional branches [BNZ, BZ, BNC, ...] are short branches due to the fact that they're all 2byte instructions




1 byte — 8bit — | 1 byte — 8bit —

* ↳ If this byte is (+ve), the jump forward; (-ve), the jump backward

Q: which's better to use? BNZ along with DECF or DCFSNZ?

DCFSNZ, because it combines the decrement and jump into a single instructure.

* it can be a value from [-127 _ 128]

* to calculate the target address = [(2nd byte of instruction * 2) + PC] (PC of the next instruction)

Ex(8) Verify the calculation of backward jumps for the listing of ex(2), shown below :-

| LOC | OBJECT CODE | LINE | | SOURCE TEXT |
|---|---|---|---|---|
| VALUE | | | | |
| 00000025 | | 00001 | COUNT | EQU 0X25 |
| 000000 | | 00002 | | ORG 0H |
| 000000 | 0E0A | 00003 | | MOVLW d'10' |
| 000002 | 6E25 | 00004 | | MOVWF COUNT |
| 000004 | 0E00 | 00005 | | MOVLW 0H |
| 000006 | 0F03 | 00006 | AGAIN | ADDLW 3H |
| 000008 | 0625 | 00007 | | DECF COUNT, F |
| 00000A | E1FD | 00008 | | BNZ AGAIN |
| 00000C | 6E81 | 00009 | | MOVWF PORTB |
| | | 0010 | | END |

(relative address; opcode — 2byte instructions: opcode / relative address)

BNZ AGAIN → E1 / FD (E1 = opcode, FD = relative address (-3))

$-3 * 2 = -6$ → backward Jump

* when the relative address of (-6) is added to 00000CH, the address of the instruction below the byte, we have $-6 + 0CH = 06H$ (the carry is dropped)

Notice that ⇒ 000006 is the address of the label AGAIN.

* FDH is a negative no. and that means it'll branch backward (how we know that?)

# GOTO to itself [Using $ sign]

It can be used to keep microcontroller busy (jump to the same location)

* HERE GOTO HERE
* GOTO $
* OVER BRA OVER
* BRA $

---

## Review Questions

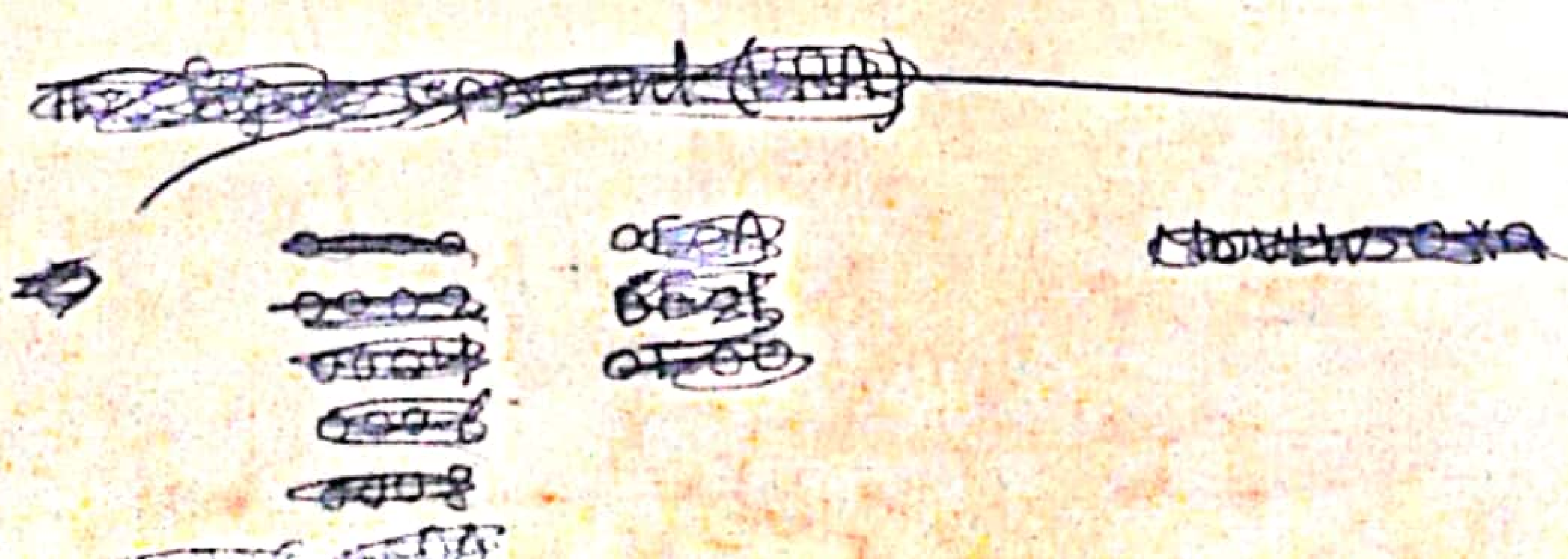
1. The mnemonic BNZ stands for branch if not zero
2. "BNZ BACK" makes its decision based on the last instruction affecting the Z flag. T
3. "BNZ HERE" is a 2 byte instruction
4. In the "JZ NEXT", which register's content is checked to see if it's zero? Z flag of status register.
5. GOTO is an 4 byte instruction
6. Find the no. of times the following loop is performed :-

```
        MOVLW   d'200'      WREG = 200            2 Loops
        MOVWF   REGA  -->   REGA = 200
BACK    MOVLW   d'100'      WREG = 100
        MOVWF   REGB  -->   REGB = 100
HERE    DECF    REGB, f   -> 90                    100
        BNZ     HERE      -> now, go to here 80 ..... 0
        DECF    REGA, f   -> 199
        BNZ     BACK      -> now go to
```

100 * 200 = 20000 times

decrement the fileReg contents 20,000 times [Using BNZ]

already DECF

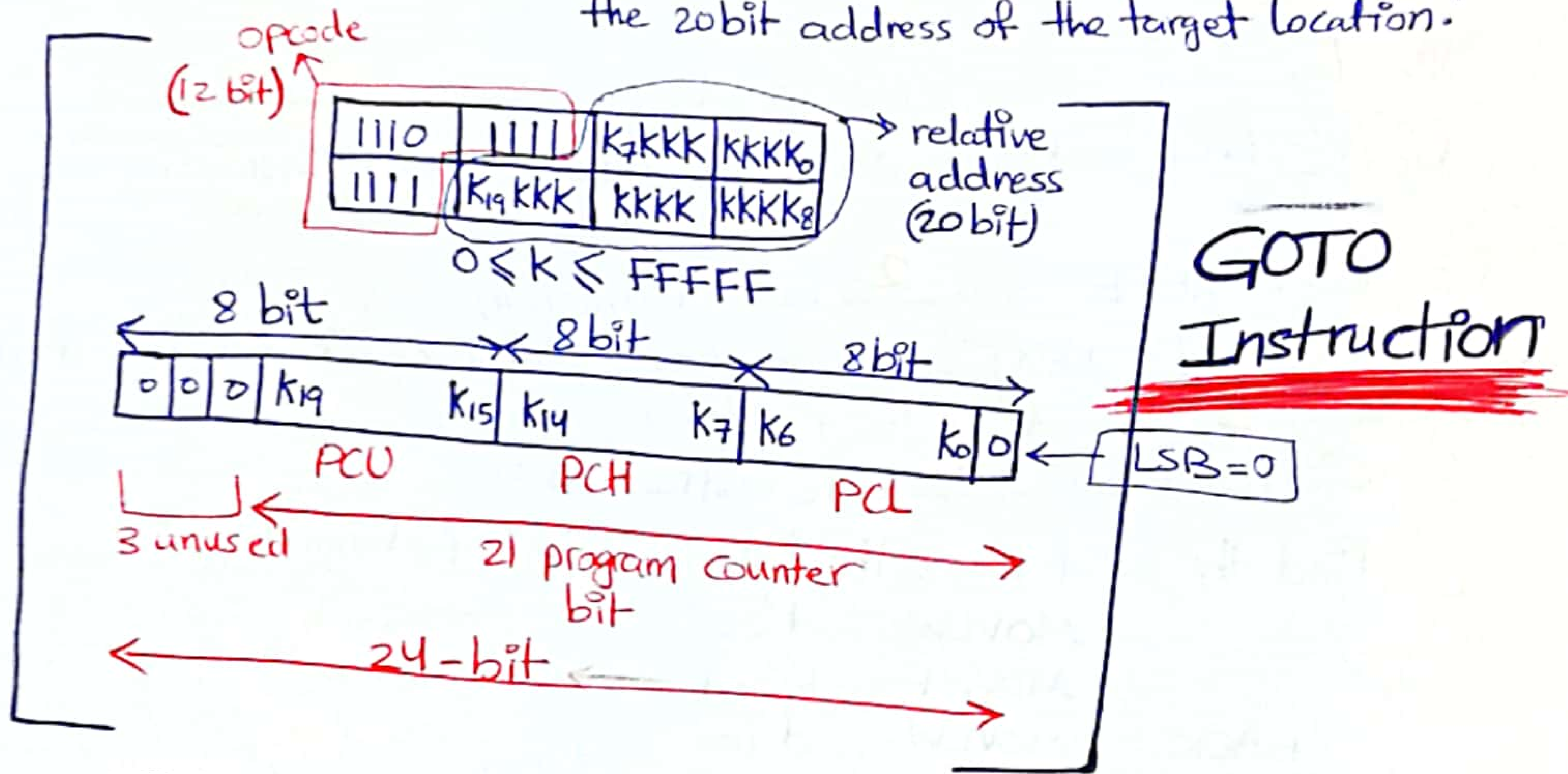


# Unconditional branch instruction ⇒

* It's a jump in which control is transferred unconditionally to the target location.
* [GOTO / BRA] unconditional branches in PIC18
  (target address) بحدد اللي بدي أستخدمه حسب

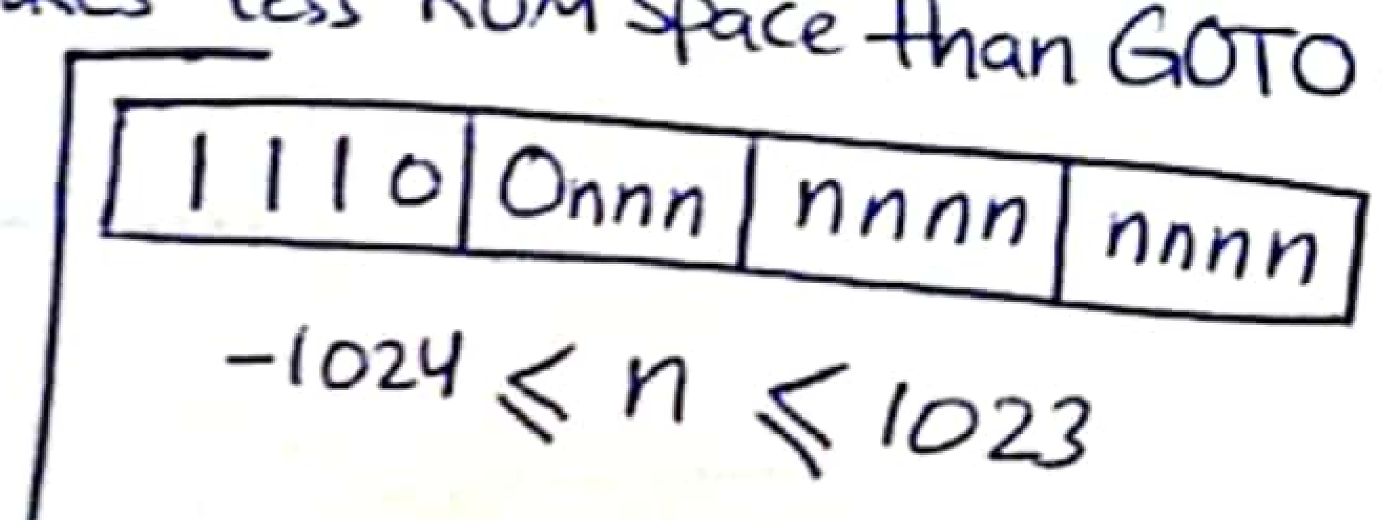
**GOTO "Long Jump"**

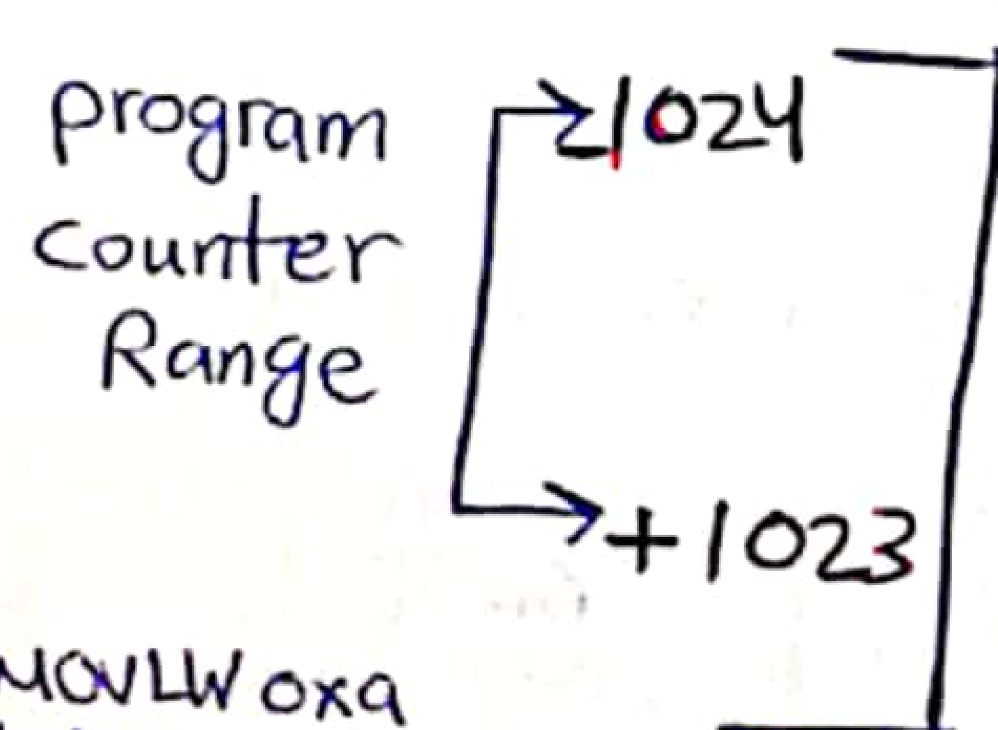
4 byte instruction (32 bit) → 12 bits for opcode and the other 20 bit represent the 20 bit address of the target location.




**BRA Instruction** "branch unconditionally instruction address range" "Short Jump"

* 2 byte instruction (16 bit) → 5 bits for opcode and the other Lower 11 bit is the relative address of the target location
* the relative add. range of [000-FFFH] divided into (forward & backward) jumps.
* It takes less ROM space than GOTO.



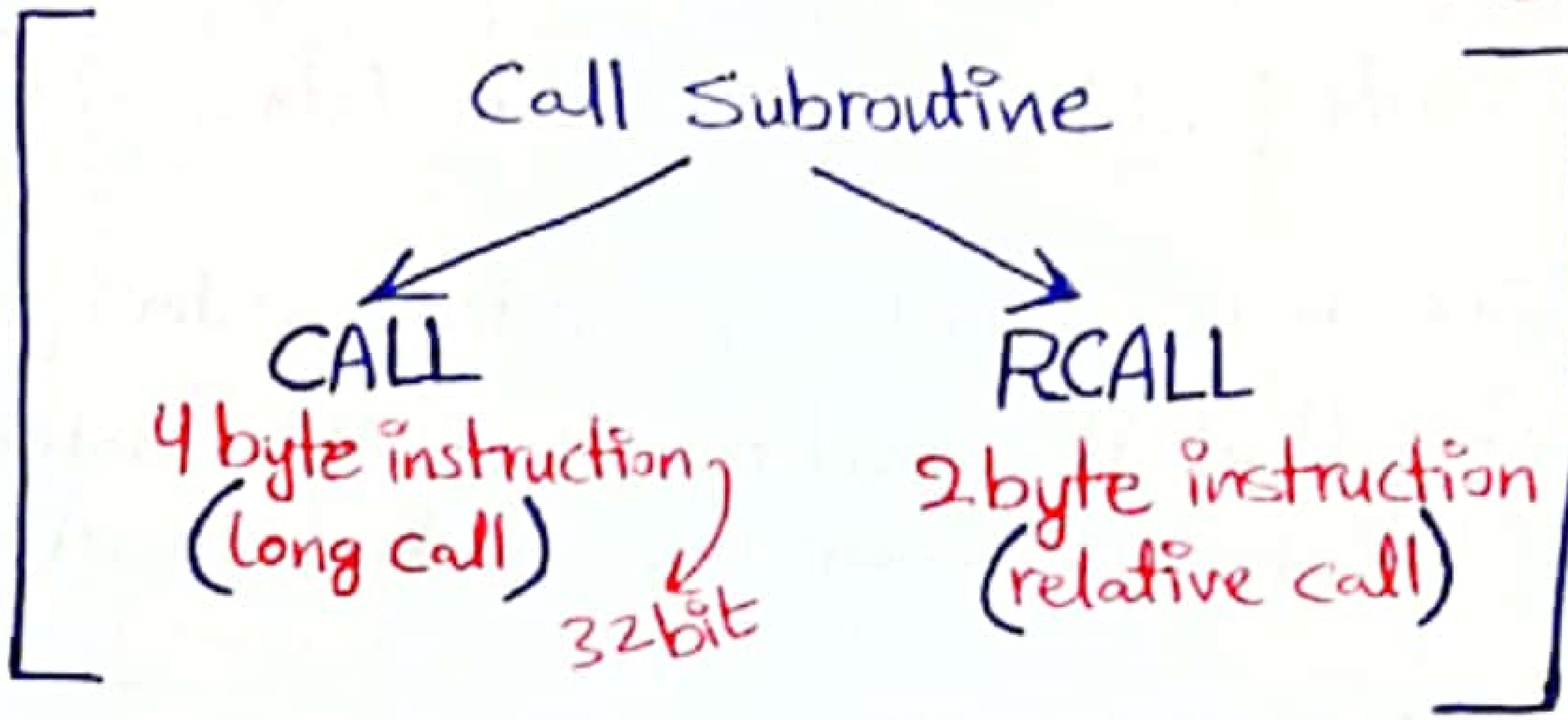




BRA →

| Line | address | opcode | Label | |
|---|---|---|---|---|
| 1 | 0000 | 0E0A | | |
| 2 | 0002 | 6E25 | | MOVLW 0x9 |
| 3 | 0004 | D001 | | MOVWF COUNT, ACCESS |
| 4 | 0006 | 0E00 | | BRA AGAIN → forward jump |
| 5 | 0008 | 0F03 | | MOVLW 0 |
| 6 | 000A | 0625 | AGAIN | ADDLW 0X3 |
| 7 | 000C | E1FD | | DECF COUNT, F, ACCESS |
| 8 | 000E | 6E81 | | BNZ AGAIN |
| 9 | 0010 | D7FB | | MOVWF PORT B, ACCESS |
| 10 | 0012 | FFFF | | BRA AGAIN → backward |
| | | | | NOP |

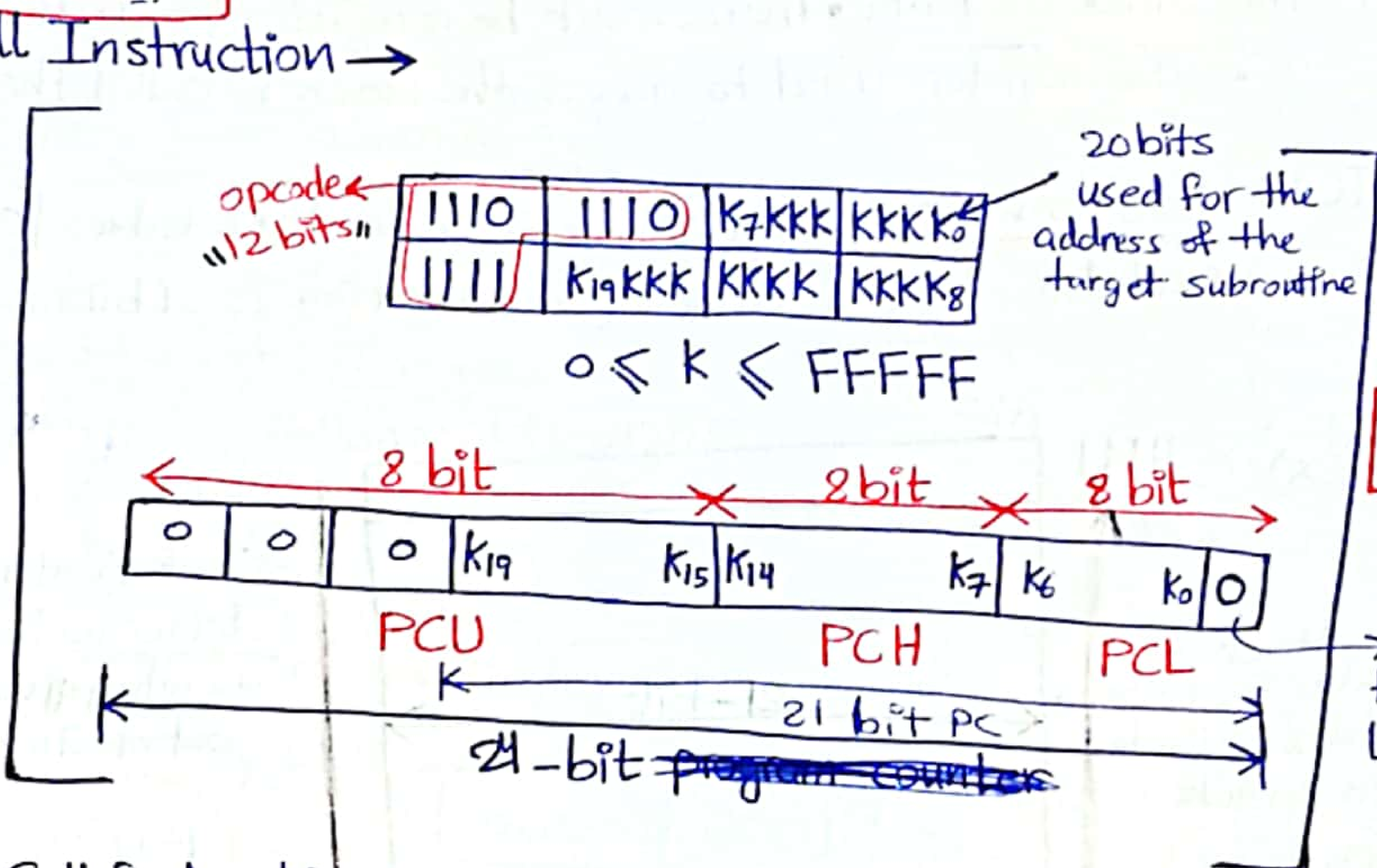
## 3.2 Call instructions and stack

- used to call a subroutine, which is often used to perform tasks that need to be performed frequently. [saving memory space & makes a program more structured]




نعمل Calling لـ (subroutine) معين في البرنامج بدي يتكرر تنفيذه أكثر من مرة .
فلازم أحط بنهاية الـ subroutine الي بدي أعملها Call لجوين RETURN
العمل والـ instruction الي بعد الـ CALL بيحطه بالـ STACK لحتى تعرف الـ PIC وين بدها ترجع بعد تنفيذ الـ subroutine الي عملناله call

*Call Instruction →




الـ PC لل GOTO و CALL

LSB = 0 to ensure it lands on an even address

∴ Call instruction can be used to call subroutines located anywhere within the 2M address space of [00000 – 1FFFFH] for the PIC18

* The mc saves on the stack the address of the instruction (below CALL) [لنتأكد بإنه الـ PIC عارفة لوين ترجع بعد تنفيذ أمر الـ CALL]

- When a subroutine is called, control is transferred to that subroutine, & the processor saves the PC (program counter) of the next on STACK & begins to fetch instructions from the new location.

وبس ينتهي تنفيذ الـ CALL ← the instruction **RETURN** transfers control back to the caller.

(note) Every subroutine needs **RETURN** as the last instruction.

Scanned with CamScanner

## ⇒ Stack and Stack Pointer in the PIC18 (SP)

* The stack is :

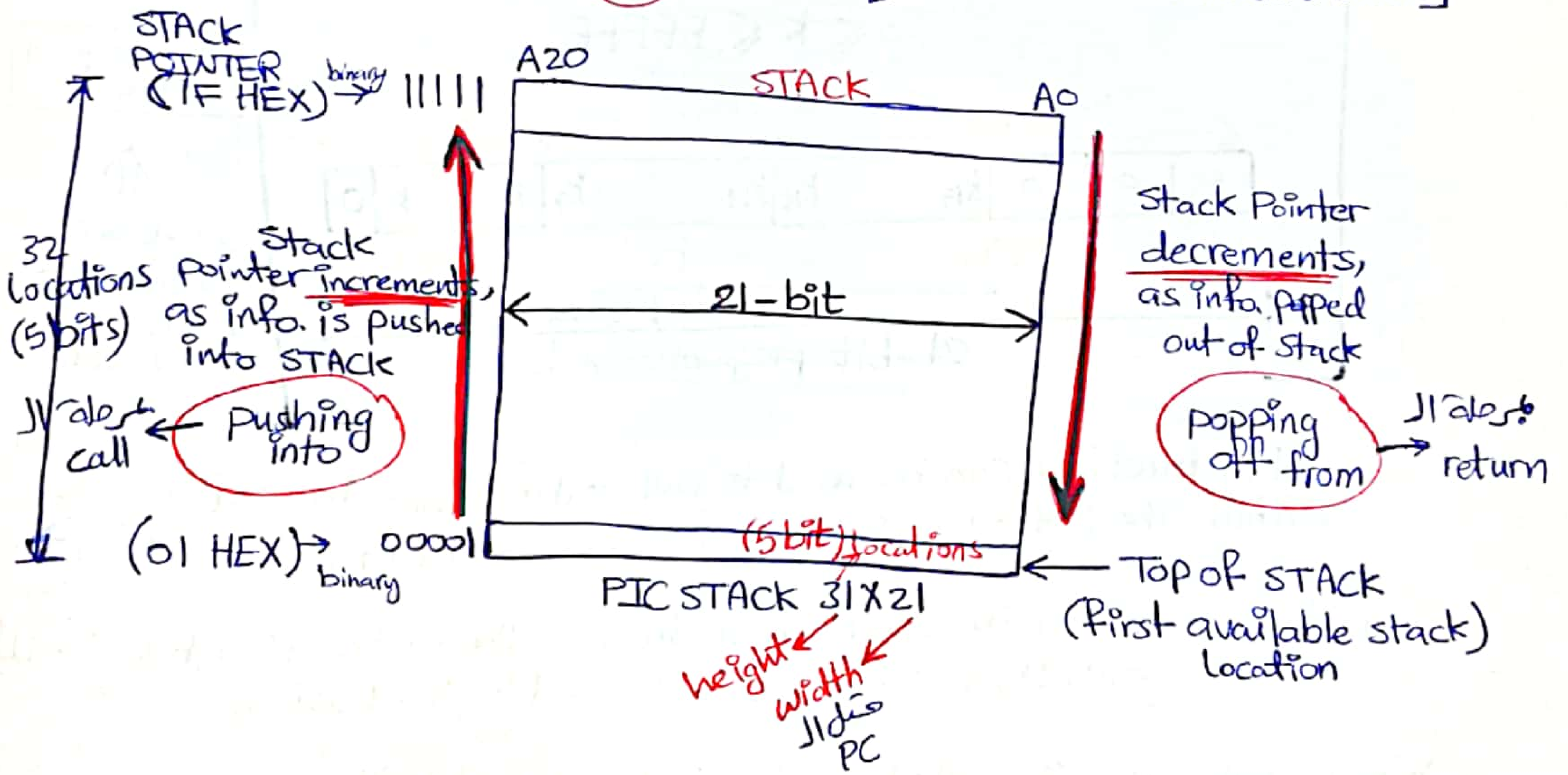
فخلي منها الـinstructions الموجودة بعد الـ call لتعرف الـPIC وين بدها ترجع بعد ما رتجي تنفيذ الـsubroutine

① Read/write memory (RAM) used by the CPU to store some very critical info. temporarily [address and could be data as well]

② 21 bit [cuz the PC is 21 bit also] can take values [00000 – 1FFFFFH]
(that means that it's used for the CALL instruction to make sure that the PIC knows where to come back to after execution of the subroutine)

- If the stack is RAM, there must be a register inside the CPU to point to it. The register used to access the stack is called the SP (stack pointer)

- PIC18 has a 5bit stack pointer, which can take values [00 – 1FH] gives us a total of 32 locations [each location is 21 bits wide]




→

# How Stacks are accessed in the PIC18 :-

The storing of CPU info. such as the PC on the STACK is called a **PUSH**, and loading the contents of the stack back into a CPU register is called a **POP**

باختصار : [a register is pushed onto the STACK to save it, and popped off the stack to retrieve it] ← استرجاع

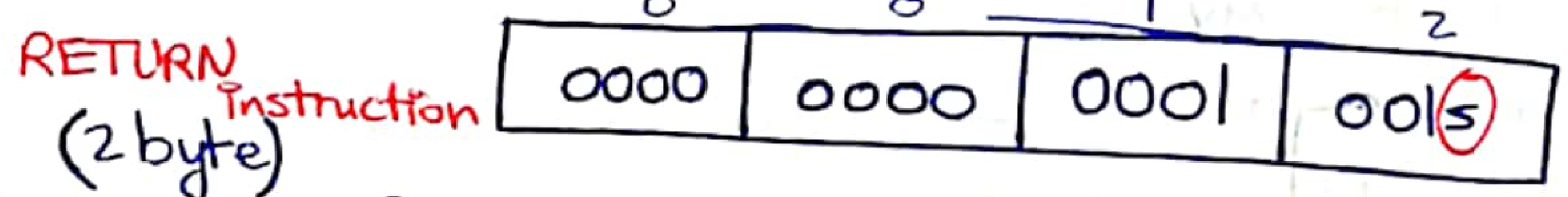
| **Pushing onto the stack** (SP increment) | **Popping from the stack** (SP decrement) [عكس ال pushing] |
|---|---|
| SP is pointing to the last used location of the stack (the top of the stack_TOS) As data is pushed onto the stack, the stack pointer is incremented. | when the RETURN instruction at the end of the subroutine is executed, the top location of the stack is copied back to the PC and the SP is decremented once. |

---

*** Call instruction and the role of the stack →**

- In the PIC, the CPU uses the stack to save the address of the instruction just below the call instruction.
- This is how the CPU knows where to resume when it returns from the called subroutine.

- The STACK is popped and the top of the STACK (TOS) is loaded into the PC [وظيفة ال Return]




If **S=1**, the contents of the shadow registers WS, STATUSS and BSRS are loaded into their corresponding registers, W, status, and BSR. ↙ return مع

**S=0**, no update of these registers occurs (default) ↙ return بلا

## Ex(3.9)

Toggle all the bits of the SFR register of PORTB by sending it to the values 55H and AAH continuosly. Put a time delay in between each issuing of data to PORTB?

```
→ MYREG   EQU    0X08          ; use location 08 as counter
          ORG    0
BACK      MOVLW  0X55          → Load 55H into WREG
          MOVWF  PORTB         → Send 55 to PORT B
          CALL   DELAY         → time delay
          MOVLW  0XAA          → send AAH to PORTB
          MOVWF  PORTB
          CALL   DELAY
          GOTO   BACK          → Keep doing this indefinitely
                                 this is the delay subroutine
          ORG    300H          → put time delay at address 300H
DELAY     MOVLW  0XFF          → WREG = FFH, the counter (255)
          MOVWF  MYREG
AGAIN     NOP                  → no operation wastes clock cycles
          NOP                    5*255*1µs = no. of delay time
          DECF   MYREG,F
          BNZ    AGAIN         → repeat until MYREG becomes 0
          RETURN               → return to caller
          END                  → end of asm file
```

delay ↓ not reserved word

بعد تنفيذها مباشرة is pushed onto the stack, and the PIC starts to execute instructions at address 000300H

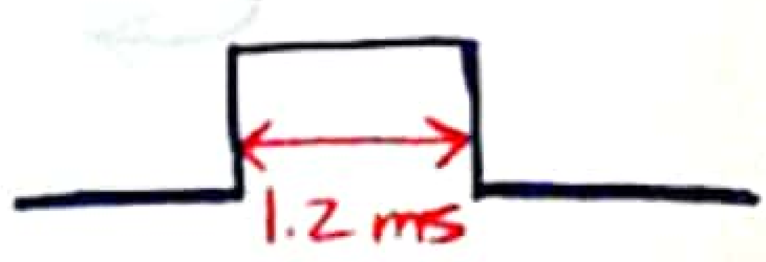
زي كأنك دخلت الcompl لـ 55 — $\overline{55} = AA$

يعني الديلاي بتبلش من 300H

لما يصير قيمته 0 control falls to the RETURN which pops the address from TOS into the PC and resumes executing the instructions after the CALL

إزا تحقق BNZ بيأفز بمقدار (2 instructions)

In the delay subroutine, the counter (MYREG = FFH) the loop is repeated 256 مرة




$$f = \frac{1}{2ms} = 500H$$

## Ex (3.10) Analyze the stack for the CALL instructions in the following program

اكتب للكتاب

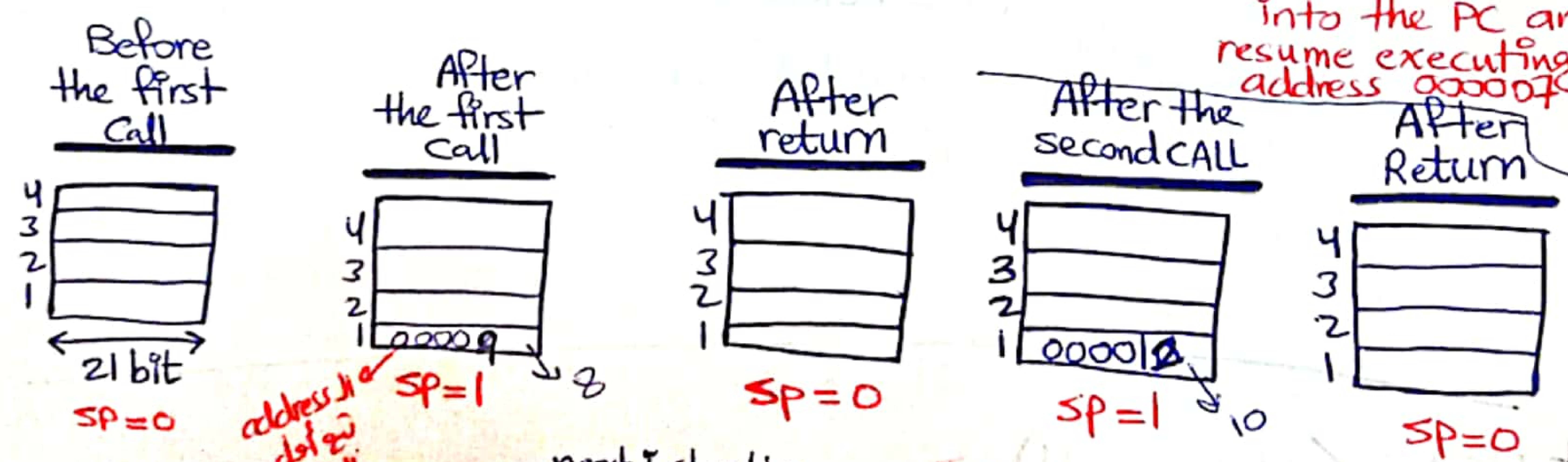
CALL 4 byte instruction, 6 Hexa's

| LOC | OBJECT CODE VALUE | LINE | SOURCE TEXT |
|---|---|---|---|
| | 00000008 | 00001 | #DEFINE PORTB 0XF81 |
| | | 00002 | MYREG EQU 0X08 ; use location 08 as counter |
| | | 00003 | |
| | | 00004 | |
| 000000 | | 00005 | ORG 0 |
| 000000 | 0E55 | 00006 | BACK MOVLW 0X55 |
| 000002 | 6E81 | 00007 | MOVWF PORTB |
| 000004 | EC80 F001 | 00008 | CALL DELAY |
| 000008 | 0EAA | 00009 | MOVLW AAH |
| 00000A | 6E81 | 00010 | MOVWF PORTB |
| 00000C | EC80 F001 | 00011 | CALL DELAY |
| 000010 | EF00 F000 | 00012 | GOTO BACK |
| | | 00013 | |
| | | 00014 | ; this is the delay subroutine |
| | | 00015 | |
| 000300 | | 00016 | ORG 300H |
| 000300 | 0EFF | 00017 | DELAY MOVLW FFH ← WREG = 255 |
| 000302 | 6E08 | 00018 | MOVWF MYREG ← Counter = 255 |
| 000304 | 0000 | 00019 | AGAIN NOP |
| 000306 | 0000 | 00020 | NOP |
| 000308 | 0608 | 00021 | DECF MYREG, F |
| 00030A | E1FC | 00022 | BNZ AGAIN |
| 00030C | 0012 | 00023 | RETURN |
| | | 00024 | END |

NOP, NOP → wastes clock cycle

DECF MYREG, F → للتقليل من العداد لعد كل عملية تكرار

RETURN → directs the CPU to pop the contents of the top location of the stack into the PC and resume executing at address 000007..



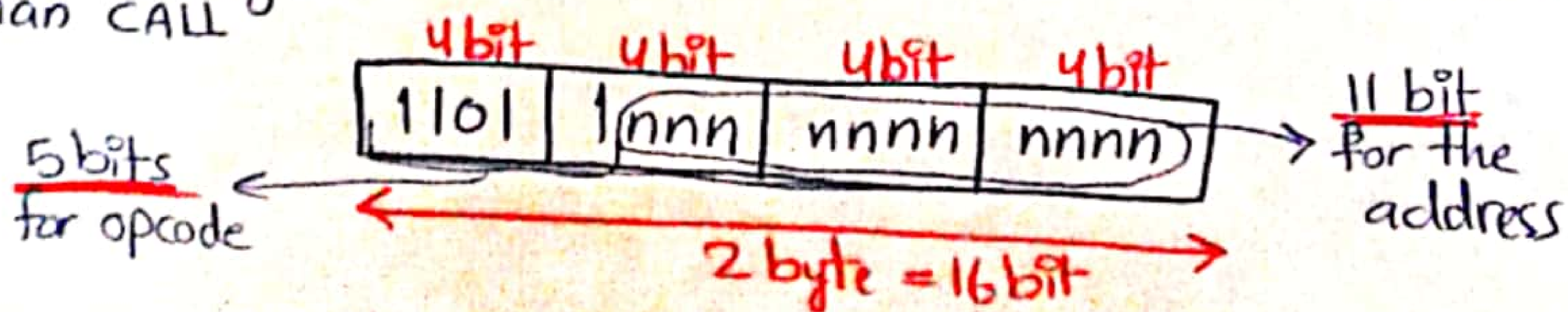

* لما تنفذ أول CALL , الـ address تبع [MOVLW AAH] next instruction saved on the stack (pushed)

## Figure (3.8) PIC Assembly Main Program That Calls Subroutines

```
; Main Program Calling subroutines
          ORG   0
Main      CALL  SUBR_1
          CALL  SUBR_2
          CALL  SUBR_3
HERE      BRA   HERE        ; stay here
; ———————— end of main
;
SUBR_1    ....
          ....
          RETURN
; ———————— end of subroutine 1
;
SUBR_2    ....
          ....
          RETURN
; ———————— end of subroutine 2
SUBR_3    ....
          ....
          RETURN
; ———————— end of subroutine 3
          END
```

# RCALL (Relative Call) →

* 2 byte instruction
* the target address must be within 4KB (11 bits of the 2 byte used of the address)
  - CALL ال عكس can be anywhere within 2M address space (4 byte) لأنه
* Save a no. of bytes more than CALL

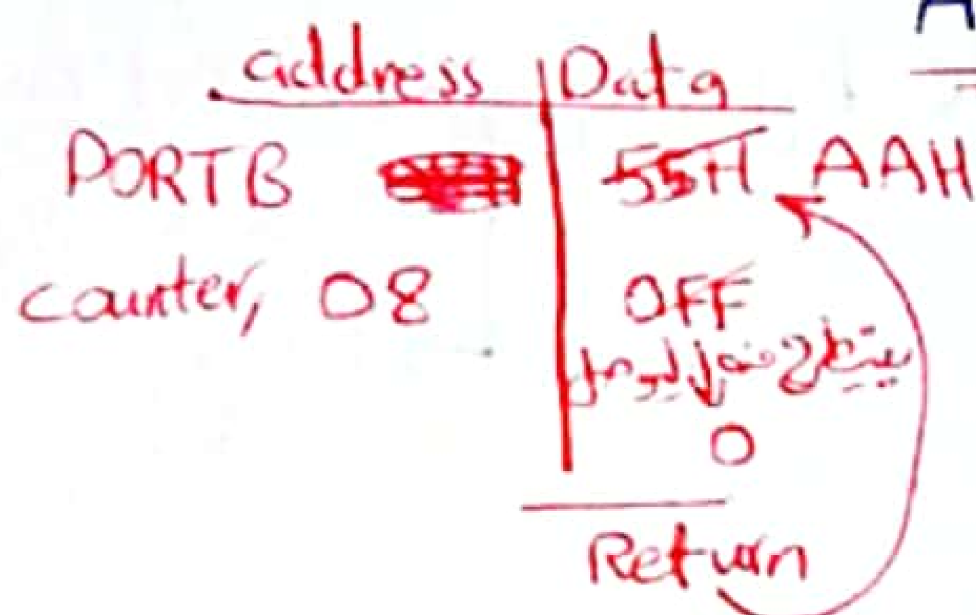


Ex(3.12) Rewrite the main part of ex(3.9) as efficiently as you can?

Using RCALL

```
→        MYREG    EQU     0X08
                  ORG     0
                  MOVLW   55H
         BACK     MOVWF   PORTB
                  RCALL   DELAY
                  COMPF   PORTB,F
                  BRA     BACK
;                 this is the delay subroutine
         DELAY    MOVLW   0FFH      , WREG = 255 , the counter
                  MOVWF   MYREG
         AGAIN    NOP               → no operation wastes clock cycles
                  NOP
                  DECF    MYREG,F
                  BNZ     AGAIN     → repeat until MYREG=0
                  RETURN            → return to caller (MYREG=0)
                  END
```

toggle the contents of PORTB 256 times using RCALL and time delay after put in PORTB 55H




## Review Questions

1. How wide is the size of the stack in the PIC18? 21 bit
2. In the PIC18, control can be transferred anywhere within the 2M of code space by using the CALL instruction. T
3. With each CALL instruction, the stack pointer register, is Inc
4. With each RETURN " , " " " " " (inc, dec) DEC
5. On power up, the PIC uses location 31 X 21 (locations) as the first location of the stack
6. (RCALL, CALL) takes more ROM space → 4byte لأنه
7. upon reset, the first available location of the stack is 1F لأنه SP decrement للأعلى

## 3.3 | PIC18 Time Delay And Instruction Pipeline

In this section, we discuss how to generate various time delays and calculate exact delays for the PIC18.
Discuss instruction pipelining and its impact on execution time.

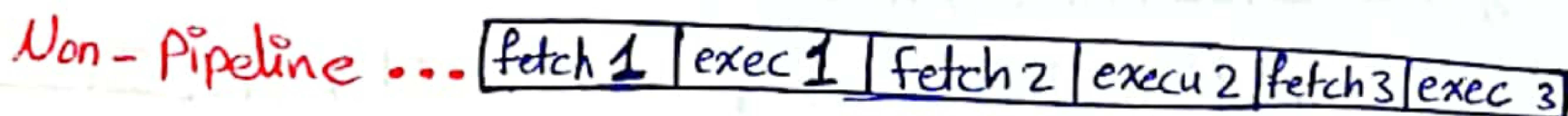
### # Delay Calculating for the PIC18 →

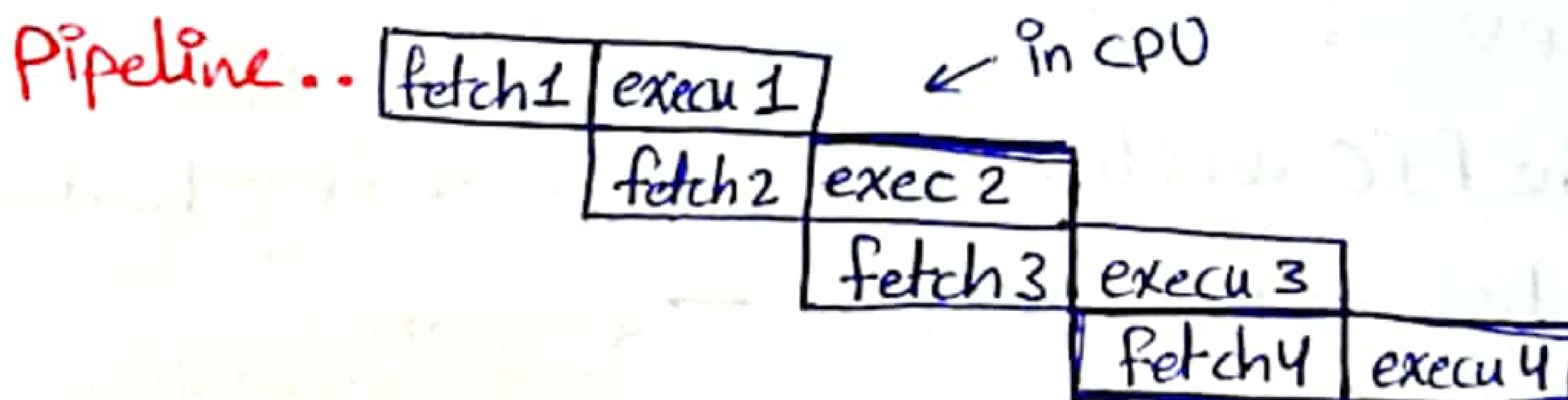
- Two factors can affect the accuracy of the delay :-
  - the duration of the clock period, which is function of the crystal frequency that connected to the OSC1 and OSC2 input pins.
  - the instruction cycle duration,
    most of the PIC18 instructions consumes 1cycle by using → Harved Arch. → RISC Arch. → pipeline concep between fetch & execute

مع تطور المتحكمات والمعالجات الدقيقة أصبحت تستهلك (1 cycle)
كانت قديماً أكثر من هيك تستهلك فكانت الطريقة الوحيدة
لزيادة كفاءة المتحكم وأداؤه من غير (losing code compatibility)

⇒ By reducing the no. of instruction cycles it takes to execute an instruction

Pipelining :- to allow the CPU to fetch and execute the instruction at the same time.

Non - Pipeline ...



Pipeline ..


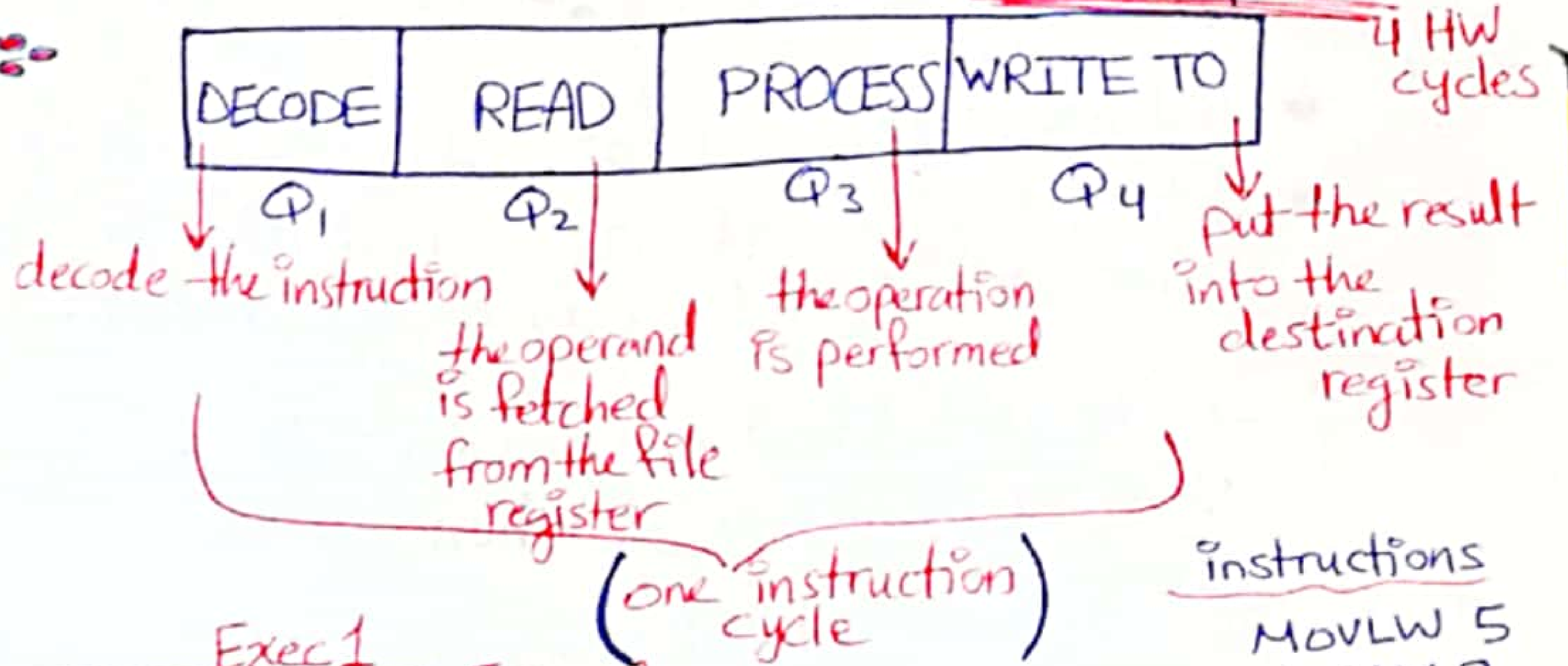

### # PIC multistage execution pipeline →

- Superpipeline used to speed up execution of instructions, that split into many small steps that are all executed on parallel.
- In this way, the execution of many instructions is overlapped. One limitation of superpipelining is that the speed of execution is limited to the slowest stage of the pipeline.

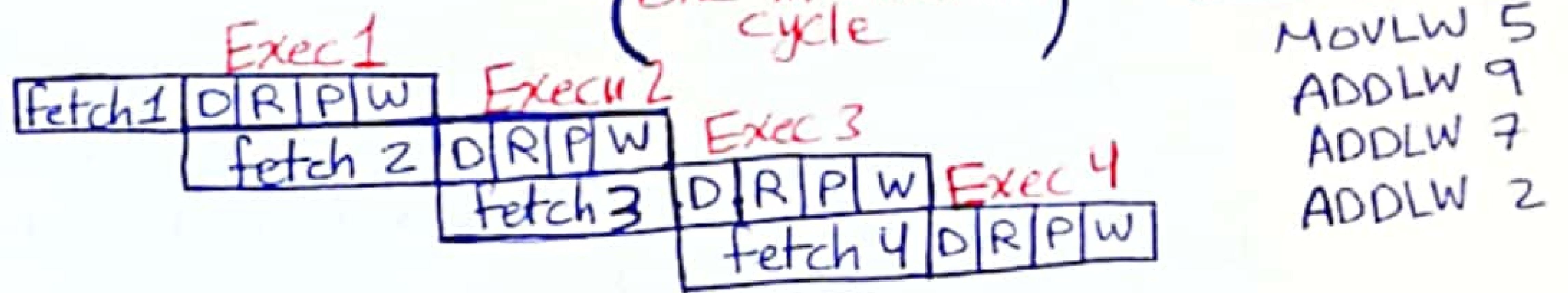
Scanned with CamScanner

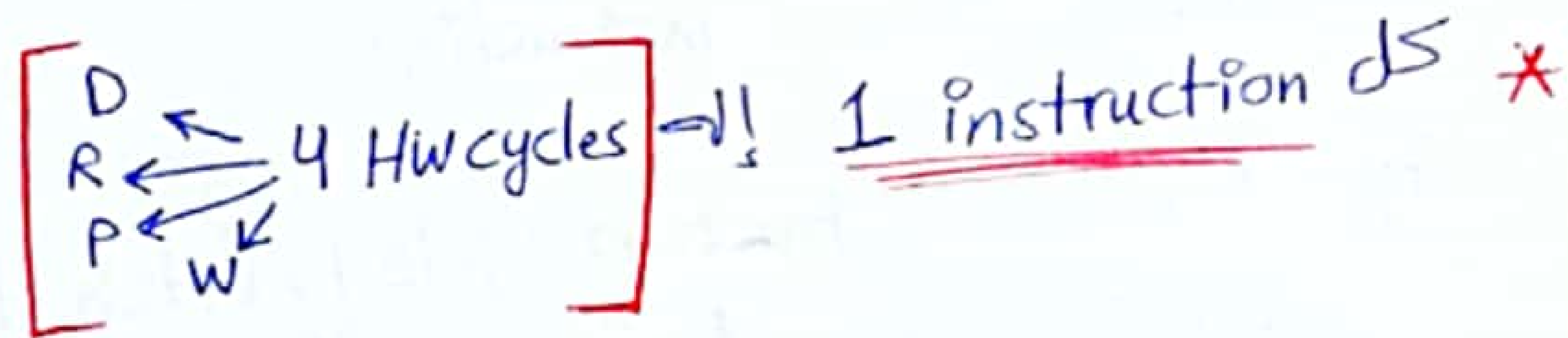
→ In the PIC18, the execution unit takes 4 clock periods of the oscillator:

Pipeline activity after the instruction has been fetched ⇒




Pipeline activity for both fetch and execute ⇒




# Instruction cycle time for the PIC:

- It's the amount of time for the CPU to execute an instruction.
- Most instructions take one or two cycles to execute (some instructions such as BTFSS can take up to 3 cycles) → Bit test file skip if set
- instruction cycle depends on the frequency of oscillator.
- Clock source → crystal oscillator and on-chip circuitry
- Each 4 oscillator period (→ DRPW) represents ≡ One instruction cycle (Clock Period)

**Ex(3.14)** The following shows the crystal frequency for (3) different PIC-based systems. Find the period of the instruction cycle in each case:

a) 4MHZ    b) 16MHZ (ميجاهيرتز)    c) 20MHZ

→ a- $\frac{4MHZ}{4} = 1\,MHZ$, $T = \frac{1}{1MHZ} = 1\,\mu s$ (مايكرو ثانية)

b- $\frac{16MHZ}{4} = 4\,MHZ$, $T = \frac{1}{4MHZ} = 0.25\,\mu s = 250\,ns$

c- $\frac{20MHZ}{4} = 5\,MHZ$, $T = \frac{1}{5MHZ} = 200\,ns$

Instruction cycle time period (fetch cycle period (time))

كل الفترة مقسمة ال crystal frequency على 4 (4 osc. period) لأنه ... وهنا ضد ال inverse للناتج لتطلع الوحدة

$$\left[f = \frac{1}{T_p}\right]^{-1} = [s]$$

"s" → وحدة زمن (ثانية)

# Branch Penalty :-

- The overlapping of fetch and execution of the instruction widely used.
- For pipeline concept, we need a buffer or queue in which an instruction is prefetched and ready to be executed.

- If the prefetched instruction isn't correct, the CPU must flush out the queue (the memory) ? **When**

  - When a branch instruction is executed
  - the CPU starts to fetch codes from the new memory location, and the code in the queue that was fetched previously is discarded.
  - the execution unit must wait until the fetch unit fetches the new instruction. يعني هاي الحالة ⇐ "Branch Penalty"

> ازا تحقق ال branch بيصير ال (flushing)

- Penalty → extra instruction cycle to fetch the instruction from the target location instead of executing the instruction right below the branch

---

* If Jump :

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'n' | Process Data | Write to PC |
| No Operation | No Operation | No Operation | No Operation |

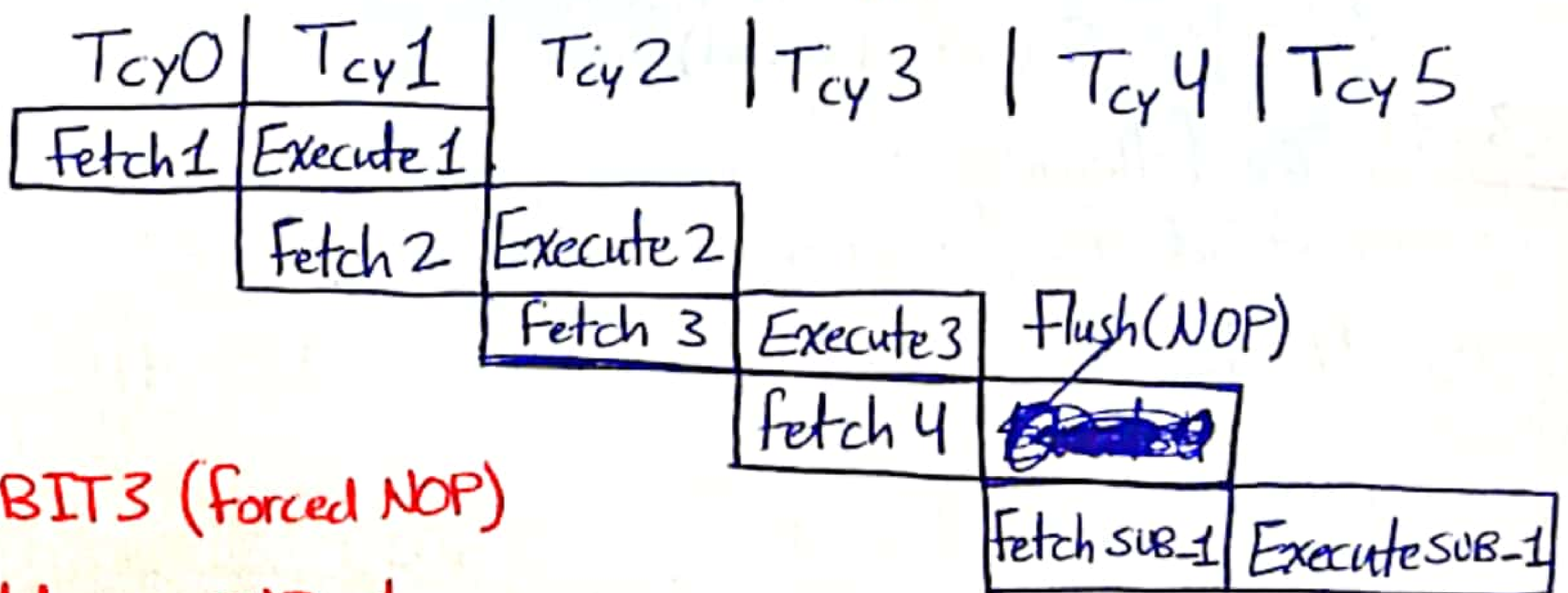
If No Jump :

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'n' | Process Data | No Operation |

> يعني ال execution للسابق وال fetching لل instruc اللي بعد الي بتنفذ منحط NOP = "flushing"

* **example** :

MOVLW 55H
MOVWF PORTB
BRA SUB-1
BSF PORTA, BIT3 (Forced NOP)
Instruction @ address SUB-1




∴ all instructions are single-cycle except for any program branches. These take two cycles since the fetch instruction is "flushed" from the pipeline, while the new instruction is being fetched and then executed.

Scanned with CamScanner

**Ex (3.15)** For a PIC18 system of 4 MHZ, find how long it takes to execute each of the following instructions :-

a) MOVLW  b) ADDLW  c) DECF
d) MOVWF  e) CALL  f) BNZ
g) NOP  h) GOTO

$\frac{4}{4} = 1MHz$, $T = 1\mu s$ → $= 1\mu s *$ no. of instruc. cycles

→

| Instruction | Instruction cycles | Time to execute |
|---|---|---|
| MOVLW 55h | 1 (R, W, D, P) | $1*1\mu s = 1\mu s$ |
| DECF MYREG | 1 | $1*1\mu s = 1\mu s$ |
| MOVWF | 1 | $1*1\mu s = 1\mu s$ |
| ADDLW | 1 | $1*1\mu s = 1\mu s$ |
| NOP | 1 | $1*1\mu s = 1\mu s$ |
| GOTO | 2 | $2*1\mu s = 2\mu s$ |
| CALL | 2 | $2*1\mu s = 2\mu s$ |
| BNZ | 2/1 or | $2*1\mu s = 2\mu s$ (1 $\mu s$ if it falls through) |

2 byte ← (MOVLW, DECF, MOVWF, ADDLW, NOP)

4 byte لأنه (GOTO, CALL)

NOP ما بتتعامل مع ال WREG ولا تقرأ منه

Delay ما بتتعامل مع fileReg ولا تقرأ منه fileReg ولا تكتب

**Ex (3.16)** Find the size of the delay of the code snippet ~~before~~ below if the crystal frequency is (4 MHZ) :- → crystal freq.

```
MYREG  EQU   0X08            ; use location 08 as counter
DELAY  MOVLW FFh      → 1    (FFh → 255)
       MOVWF MYREG    → 1
AGAIN  NOP            → 1
       NOP            → 1
       DECF  MYREG,F  → 1
       BNZ   AGAIN    → 2
       RETURN         → 1
```

Instruction cycle

RETURN → 1 , لأنه دايمًا عارفين وين مخزن ال address

Calling استدعاء

period of the instruction cycle $\Rightarrow \frac{4MHZ}{4} = 1MHZ$

$T = \frac{1}{1MHZ} = 1\mu s$

# القانون

time delay = [(no. of instructions * no. of loop value) + Overhead instruction] * the Period of the instruction cycle [s]

time delay $= [(5 * 255) + 3] * 1\mu s = 1278\,\mu s$

3 → MOVLW, MOVWF, RETURN (برا اللوب)

الفرق بين مفهوم ال timedelay وال time to execute

time to execute = $\frac{1}{f/4}$ (period of the instruc. cycle) * no. of instruc. cycle



Scanned with CamScanner

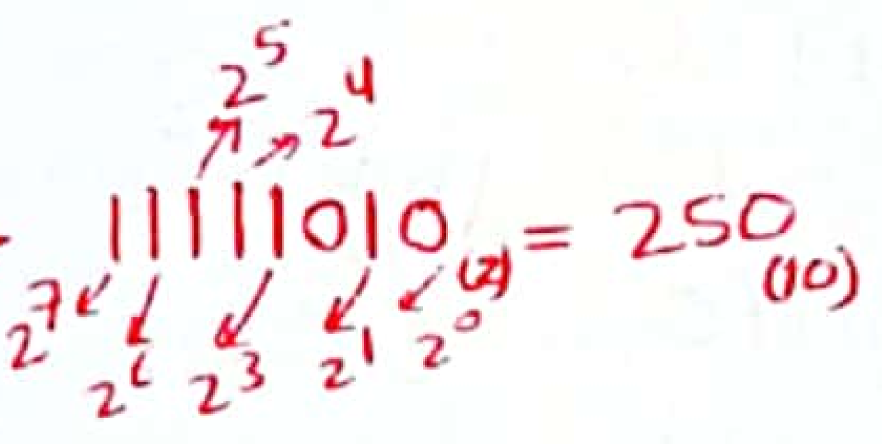
Ex (3.17) Find the size of the delay in the following program if the crystal frequency is [4 MHZ]?

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |
| 1 | 0 | 0 | 0 | 8 |
| 1 | 0 | 0 | 1 | 9 |
| 1 | 0 | 1 | 0 | A |

```
MYREG   EQU   0X08
        ORG   0
BACK    MOVLW  55h
        MOVWF  PORTB
        CALL   DELAY
        MOVLW  AAh
        MOVWF  PORTB
        CALL   DELAY
        GOTO   BACK
        ORG    300H
DELAY   MOVLW  FAh
        MOVWF  MYREG
AGAIN   NOP
        NOP
        NOP
        DECF   MYREG, F
        BNZ    AGAIN
        RETURN
```

طالب بالسؤال ال size تاع ال delay

هون بيلش أحسب بالقانون




time delay = $[(6 * 250) + 3] * 1\mu s = 1503\ \mu s$

---

* better way to use nested loop * because so [تقلل منها من عدد ال NOPs المستخدم في البرنامج]

→ NOP (no operation) simply wastes time, but takes 2 bytes of program ROM space and that is too heavy a price to pay for just one instruction cycle

Scanned with CamScanner

## Ex (3.20) Write a program to toggle all the bits of SFR PORTB every 1s. Assume that the crystal frequency is 10MHz and the system is using PIC18F458 :-

time delay ← 1s

Nested Loop

$\frac{10}{4} = 2.5MHz$

$T = \frac{1}{2.5MHz} = 400ns$

```
R2      EQU  0X2
R3      EQU  0X3
R4      EQU  0X4
        MOVLW   55h
        MOVWF   PORTB
BACK    CALL    DELAY_500MS   (melli)
        COMF    PORTB
        GOTO    BACK

DELAY_500MS
        MOVLW   D'20'
        MOVWF   R4
BACK    MOVLW   D'100'
        MOVWF   R3
AGAIN   MOVLW   D'250'
        MOVWF   R2
HERE    NOP
        NOP
        DECF    R2,F
        BNZ     HERE
        DECF    R3,F
        BNZ     AGAIN
        DECF    R4,F
        BNZ     BACK
        RETURN
```

دخلنا ع ال PORTB القيمة اللي بدنا نعملها toggle

حفظ الصيغة

5MS

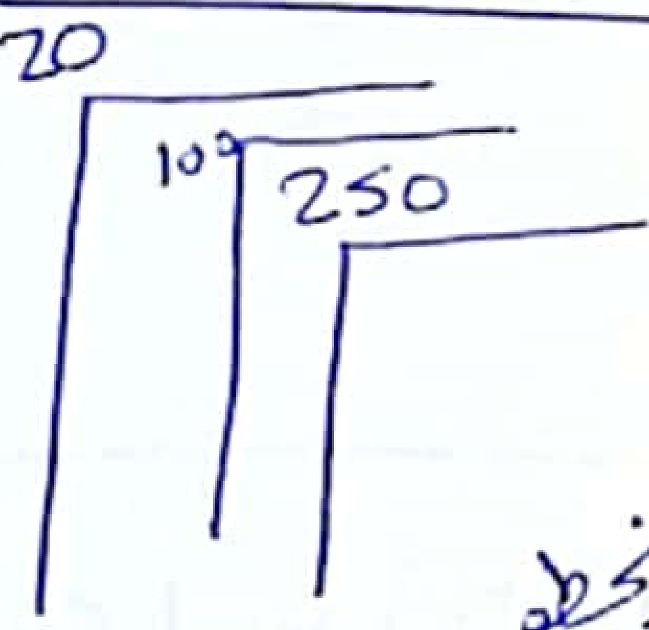
عدد ال instructions اللي بأعمق اللوب

① في عندي 5 instructions أصغر من صيغة ال HERE Loop

```
HERE NOP
     NOP
     DECF R2,F
     BNZ HERE
```

$\frac{1}{5MS} = 0.2MHz = 200,000 = 200 * 10 * 200$

② بستنتج عدد اللوبات اللي بحتاجها (3loops)




مع ال delay بنحطو قيمة ال MOVLW في MOVWF R مثلاً

$$\text{Time Delay} = 20 * 100 * 250 * 5 * 400ns = 1,000,000,000\ ns = 1s$$

- $20 * 100 * 250$ → no. of loop
- $400ns$ → period time of instruc. cycles

(للتحقق / for check)

---

$\frac{1}{5MS} = 0.2M = 0.2 * 10^6$

كيف بدي أقسمه ل لوب

$= 200,000$

$= 200 * 200 * 10$

# Review Questions

1- Assuming a crystal frequency of 4MHZ (→ 1 μs), find the time delay associated with the loop section of the following delay subroutine:

```
DELAY   MOVLW   D'100'
        MOVWF   MYREG
HERE    NOP
        NOP
        NOP
        NOP
        NOP
        DECF    MYREG,F
        BNZ     HERE
        RETURN
```

time delay = ([100 * 8] + 3) * 1 μs
= 803 μs

---

2- • BRA and CALL will always take 2 instruction cycles __T__ (4 byte ركز)

• The BNZ instruction will always take 2 instruction cycles __F__
only if it branches to the target address.

• The CALL and RCALL instructions take the same amount of time to execute even though one is a 4byte instruction and the other is a 2byte __F__

---

3- Find the time delay for the delay SUBROUTINE shown below?

a. (4MHZ)

```
        MOVLW   D'200'
        MOVWF   REGA
BACK    MOVLW   D'100'
        MOVWF   REGB
HERE    NOP
        DECF    REGB,F
        BNZ     HERE
        DECF    REGA,F
        BNZ     BACK
```

T = 4 * 100 * 200 * 1 μs = 80 ms

b. (16MHZ)

```
        MOVLW   D'200'
        MOVWF   REGA
BACK    MOVLW   D'100'
        MOVWF   REGB
HERE    NOP
        NOP
        DECF    REGB,F
        BNZ     HERE
        DECF    REGA,F
        BNZ     BACK
```

T = 5 * 100 * 200 * 0.25 μ
= 25,000 [s]

c. (4MHZ)

```
        MOVLW   D'200'
        MOVWF   REGA
BACK    MOVLW   D'250'
        MOVWF   REGB
HERE    NOP
        DECF    REGB
        BNZ     HERE
        DECF    REGA
        BNZ     BACK
```

T = 200 * 250 * 1 μ * 4
= 0.2 s

d. (10MHZ)

```
        MOVLW   D'200'
        MOVWF   REGA
BACK    MOVLW   D'100'
        MOVWF   REGB
        NOP
        NOP
        NOP
HERE    DECF    REGB,F
        BNZ     HERE
        DECF    REGA,F
        BNZ     BACK
```

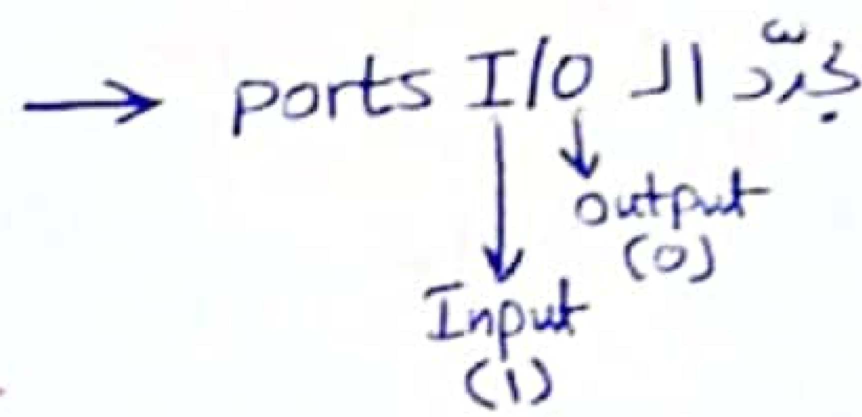
T = 200 * 100 * 3 * 0.4 μ
= 24 ms

Scanned with CamScanner

# (PIC) سؤال سكشن

**Q1** Write a program to test PORTB to see whether it has $ sign. If it does, send 'Y' to PORTC; otherwise, PORTC stays cleared?

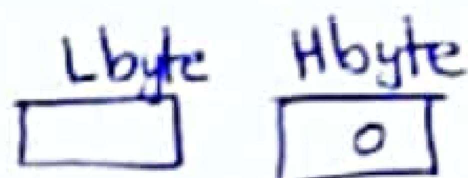
```
→   CLRF    TRISC          → ports I/O بحدد ال
    SETF    TRISB
CLRF PORTC ←  MOVLW  A'$'
    CPFSEQ  PORTB   ← fileReg loc.
    BRA OVER              ) ازا تحقق شرط السؤال
    MOVLW   A'Y'        ←
    MOVWF   PORTC
OVER ...
```




**Q2** Explain what does the following program do:

```
L-byte  EQU  0x6
H-byte  EQU  0x7

        MOVLW   0
        MOVWF   H-byte
        ADDWF   40H, W
        DAW          → Decimal adjust WREG
        BNC     N-1
        INCF    H-byte, f
N-1     ADDWF   41H, W
        DAW
        BNC     N-2
        INCF    H-byte, f
N-2     ADDWF   42H, W
        DAW
        BNC     N-3
        INCF    H-byte
N-3     ADDWF   43H, W
        DAW
        BNC     N-4
        INCF    H-byte, f
N-4     MOVWF   Lbyte
```



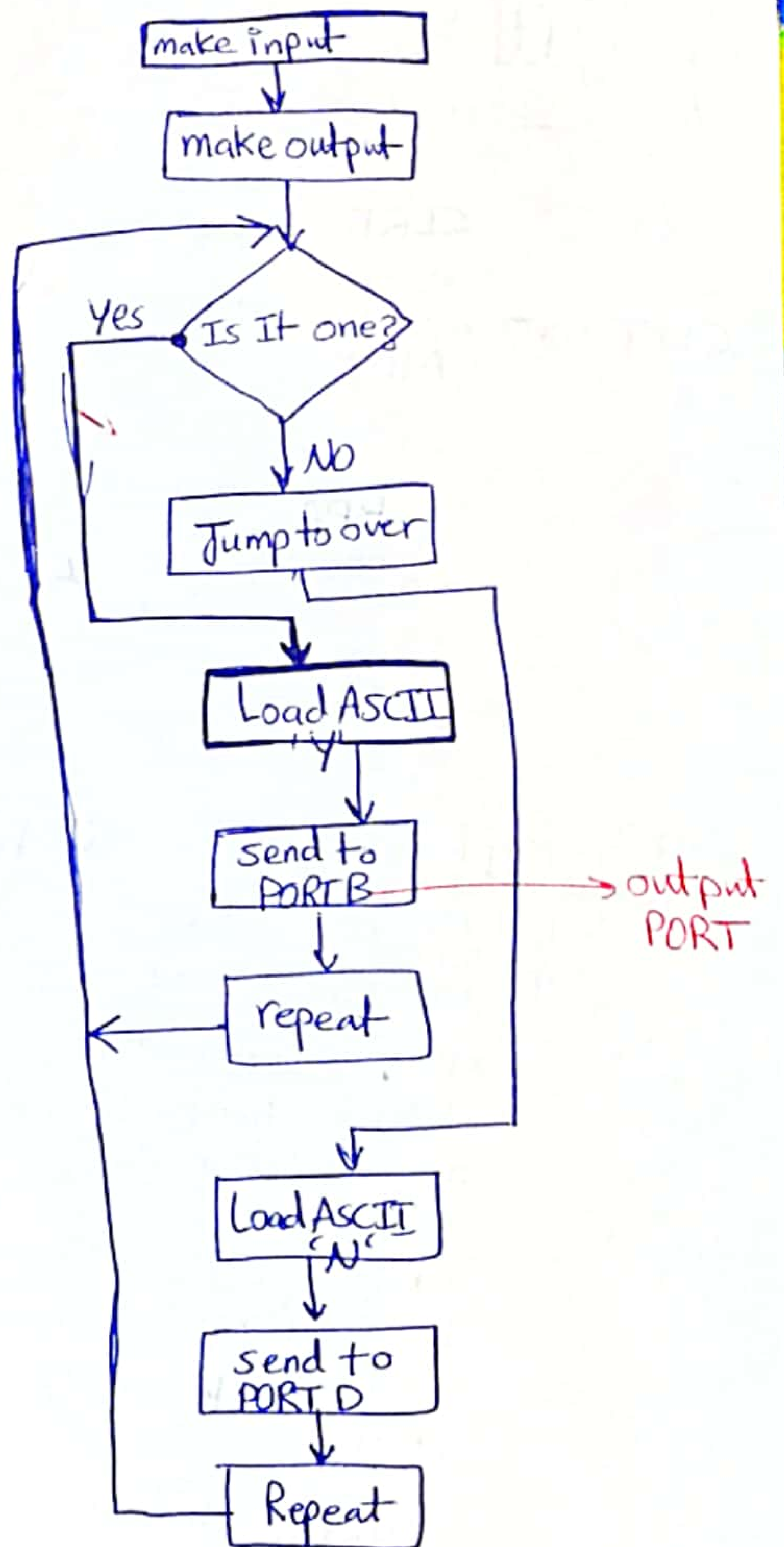
- * this program find the sum of the values 40, 41, 42, 43. put the result in 6 and 7 locations.
  (6 → Lbyte, 7 → H-byte)
- * and this program using DAW to adjust WREG values to obtain a valid BCD values

Scanned with CamScanner

## Q3 | Write a program based on the flowchart below :-

```
      BSF    TRISA , 1
      CLRF   TRISB
L1    BTFSS  PORTA , 1
      GOTO   OVER
      MOVLW  A'Y'
      MOVWF  PORTB
      GOTO   L1
OVER  MOVLW  A'N'
      MOVWF  PORTD
      GOTO   L1
```

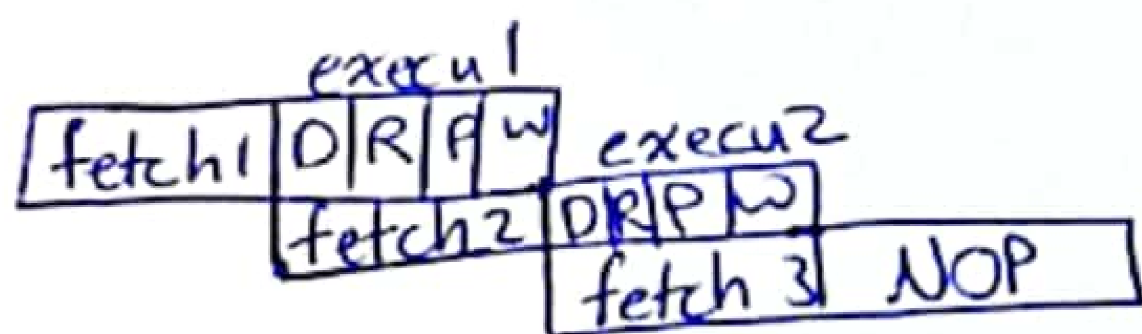
(لأن نتعامل مع pins ← PORTA , 1)




## Q4 | Explain data dependency by drawings & list 2 ways to get rid of it :-

① by RAW (read-after write)

② by NOP instruction




## Q5 | What does the following program do in two lines :-

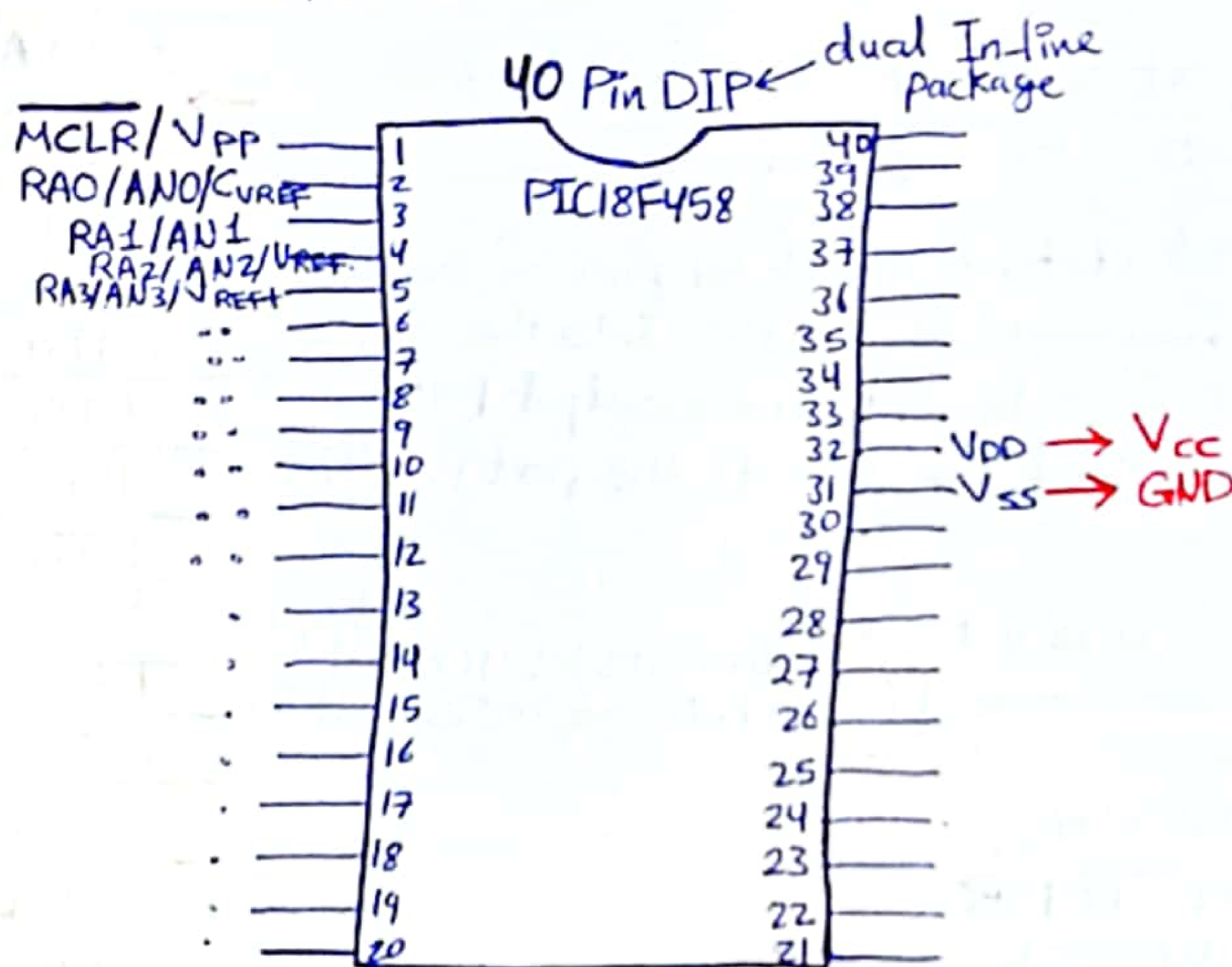
```
RCNT   EQU   0X20
MYREG  EQU   0X21
       BCF   TRISB,1      (RB1, output Pin)
       MOVLW 0X41
       MOVWF MYREG
       BCF   STATUS,C
       MOVLW 8H
       MOVWF RCNT
       BSF   PORTB,1
AGAIN  RRCF  MYREG,F
       BNC   OVER
       BSF   PORTB,1
       BRA   NEXT
OVER   BCF   PORTB,1
NEXT   DECF  RCNT,F
       BNZ   AGAIN
       BSF   PORTB,1
```

# Ch.4 : PIC I/O Port Programming

## 4.1) I/O port programming in PIC18

* PIC18 has many ports
  - → depending on the family member
  - → depending on the no. of pins on the chip
  - → Each port can be configured as input or output "Bidirectional Port"
  - → Each port has some other functions, [timers, ADC, interrupts and serial communication]

* Some ports have 8 bits, while others have not.




* The no. of ports in the PIC18 family varies depending on the no. of pins on the chip

| Pins (chip) | 18-Pin (PIC18F1220) | 28-Pin (PIC18F2220) | 40-Pin (PIC18F458) | 64-Pin (PIC18F6525) | 80-Pin (PIC18F8525) |
|---|---|---|---|---|---|
| PORTA | X | X | X | X | X |
| PORTB | X | X | X | X | X |
| PORTC | | X | X | X | X |
| PORTD | | | X | X | X |
| PORTE | | | X | X | X |
| PORTF | | | | X | X |
| PORTG | | | | X | X |
| PORTH | | | | X | X |
| PORTI | | | | X | X |
| PORTJ | | | | | X |
| PORTK | | | | | X |
| PORTL | | | | | |

X : Indicates that the port is available



Scanned with CamScanner

→ To use any of these ports as an input or output port, it must be programmed.

* Each port has three SFRs associated with it (for its operation):

1. TRIS register (Data Direction Register) → [بيحدد ال data إذا Input or output]
   - If the corresponding bit is 0 : output
   - 〃 〃 〃 〃 = 1 : input
2. PORT register (reads the levels on the pins of the device)
3. LAT register (Output latch) [بيفضل حتى أسويله on/off العكس]

a. → TRIS register role in outputting data :

ex to make a port an output, we write 0s to the TRISx register ⇐ يعني آخر

to output data to any of the pins of the PORTB, we must first put 0's into the TRISB register to make it an output port, and then send the data to the port B SFR itself.

| PORT | Address |
| --- | --- |
| PORTA | F80H |
| → PORTB | F81H |
| PORTC | F82H |
| PORTD | F83H |
| PORTE | F84H |
| LATA | F89H |
| LATB | F8AH |
| LATC | F8BH |
| LATD | F8CH |
| LATE | F8DH |
| TRISA | F92H |
| TRISB | F93H |
| TRISC | F94H |
| TRISD | F95H |
| TRISE | F96H |

# Upon reset, all ports are configured as input

# TRISx register has (0FFH)

* the following code will toggle all (8 bits) of [PORTB] forever with some time delay in between "on" and "off" states : * PORTB صح كلها *

```
      MOVLW  0H        → WREG = 0
      MOVWF  TRISB     → let port B an output port (لأنه دخلنا zero عليه)
L1    MOVLW  55h       → WREG = 55H
      MOVWF  PORTB     → put 55h on PORTB pin
      CALL   DELAY
      MOVLW  0XAA      → WREG = AAh
      MOVWF  PORTB     → PORT B = AAh
      CALL   DELAY
      GOTO   L1
```

(COMPF PORTB for the circled MOVLW 0XAA / MOVWF PORTB)

[ملاحظة] أول سطرين مهمين إنهم خلّوا ال PORTB كـ output مثل ال PIC18 pins وفي حال ما كانوا موجودين في AAH / 55H هتخزن جوا ال [SFR of PORTB] في ال CPU فقط.

→ TRIS register role in inputting data :

to make a port an input port, we must first put 1s into the TRISx register for that port, and then bring in (read) the data present at the pins.

* the following code will get the data present at the pins of PORTC and send it to PORTB indefinitely, after adding the value 5 to it →

```
      MOVLW  B'00000000'
      MOVWF  TRISB
      MOVLW  B'11111111'
      MOVWF  TRISC
L2    MOVF   PORTC, W
      ADDLW  5H
      MOVWF  PORTB
      GOTO   L2
```

+5 = PORTC → input
PORT B → output

# PORTA

* It's a 7 bit wide [A0_A6], but for PIC18F458 pin A6 is used for the OSC2 pin. A6 isn't available if we use a crystal frequency that provided to the PIC18F
* It can be used as an input or as an output depending on the Status of TRIS register
* On a Power-on reset, these pins are configured as inputs and read as '0' [default]
* The following code will continuosly send out to PORTA the alternating values of 55H and AAH :-

```
      ; toggle all bits of PORTA
      MOVLW  B'00000000'
      MOVWF  TRISA        → (output) PORTA ليك
L1    MOVLW  55H          → WREG = 55H
      MOVWF  PORTA        → PORTA = 55H
      CALL   DELAY
      MOVLW  AAH          → WREG = AAH
      MOVWF  PORTA        → PORTA = AAH
      CALL   DELAY
      GOTO   L1
```

سؤال عن مفهوم

[Port A as output]

$(55H)' = AAH$

by sending 55H and AAH to PORT A continuosly, we toggle all 8 bits of the PORTA register, only 6 pins (RA0–RA5) will show the toggling data.

Scanned with CamScanner

→ To make PORT A as input:

In the following code, PORTA is configured first as an input Port by writing all 1's to register TRISA, and then data is received from PORTA and saved in some RAM location of the file Reg.

```
MYREG   EQU 0X20         → save it here
MOVLW   B"1111 1111"     → WREG = 11111111(2)
MOVWF   TRISA            → make PORTA an input port
MOVF    PORTA,W          → move from fileReg of PORTA to WREG
MOVWF   MYREG            → Save it in fileReg of MYREG
                           (20H) يعني
```

MOVFF PORTA, MYREG
or
نقلنا من PORTA لـ W
ومن W لـ myreg (fileReg)

# PORT B, C, D and E →

PORTs (B & C, D) are 8 pins

PORTE is 3 pins

* Read followed by write I/O operation:-

due to the timing issue, we must be careful not to have two I/O operations one right after the other.

- Examine the following rewrite of an earlier code fragment in which data was read from PORTC and sent it to PORTB ⇒

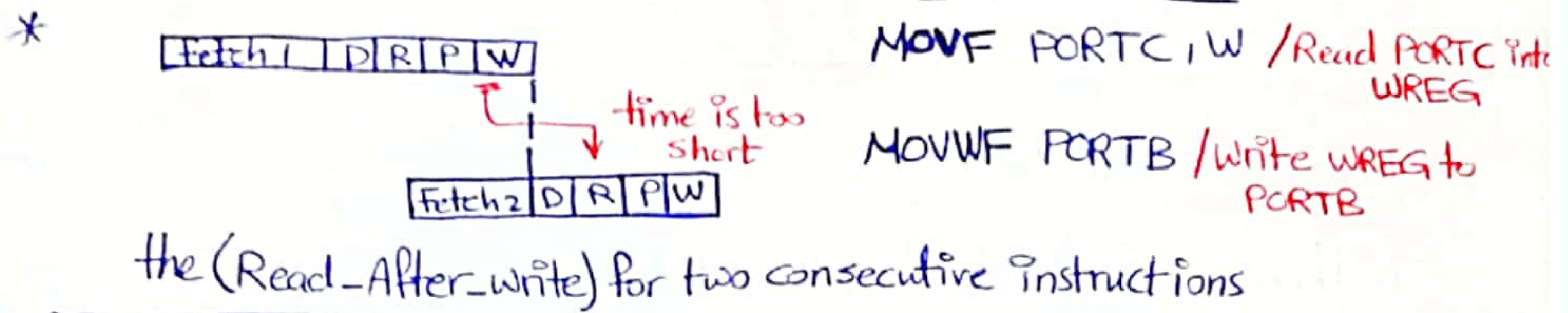
```
    CLRF    TRISB     → clear TRISB to make PORTB an output
    SETF    TRISC     → make PORTC an input port
L4  MOVF    PORTC,W   → get data from PORTC to WREG
    NOP               → to ensure that data is in WREG
    MOVWF   PORTB     → before it is sent to PORTB
    BRA     L4        → Keep doing it
```

* الـ (NOP) مفتاحها لنتأكد إنه العنصر صار بداخل الـ WREG before it's read for outputting to PORTB

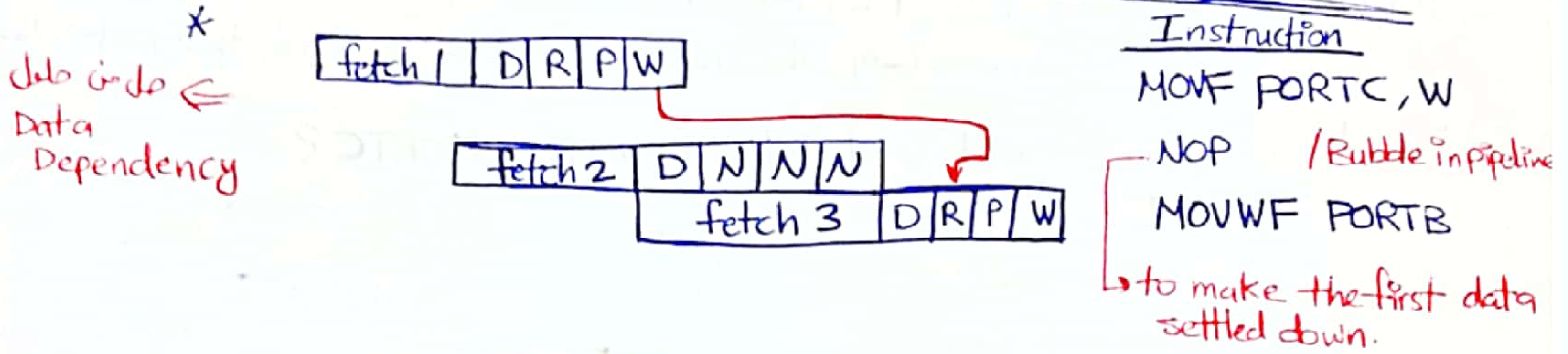
→ Data Dependency In CPU design
ويطلع منه الـ (data depends) referred to as RAW (Read-After-Write)

والـ (NOP) هاي ← will introduce a bubble into the pipeline to remove data dependency due to RAW

## Figure (4.7) Pipeline for read followed by write I/O




the (Read-After-write) for two consecutive instructions




← للبرنامج السابق كان في NOP فيه، لو قارناه مع الي بدون NOP ↓

```
      CLRF    TRISB          → 4 byte instruction
      SETF    TRISC                 [By using MOVFF]
L4    MOVFF   PORTC, PORTB
      BRA     L4
```

**Note**

Upon reset, all ports have FFH value on their TRIS register (this makes them input ports upon reset)

---

**Ex(4.1)** Write a test program for the PIC18 chip to toggle all the bits of PORTB, PORTC, and PORTD every 1/4 of a second. Assume a crystal freq of 4 MHz?

time delay = 250,000 μs = 0.25
1μ * 200 * 250 * 5 ← عدد ال HERE
4/4 ⇒ 1 μs

```
→ List P=PIC18F458
  #include P18F458.INC
  R1 EQU 0X07  ]→ for counters (Loops) → 250, 100
  R2 EQU 0X08  ]
  ORG 0
  CLRF  TRISB     → PORTB (output port)    } make ports output
  CLRF  TRISC     → PORTC (output port)
  CLRF  TRISD     → Port D (output port)
  MOVLW 0X55      → WREG = 55H             } move the value to PORTS
  MOVWF PORTB     → PORTB = 55H
  MOVWF PORTC     → PORTC = 55H
  MOVWF PORTD     → PORTD = 55H
L3 COMF PORTB, F  → toggle bits of PORTB
  COMF  PORTC, F  → 〃 〃 〃 C
  COMF  PORTD, F  → 〃 〃 〃 D
  CALL  QDELAY    → quarter of a second delay
  BRA   L3

QDELAY
    MOVLW D'200'
    MOVWF R1
D1  MOVLW D'250'
    MOVWF R2
D2  NOP
    NOP
    DECF  R2, F
    BNZ   D2
    DECF  R1, F
    BNZ   D1
    RETURN
    END
```

delay ← عشان نجهز ال counters
جملة ال HERE
نهاية البرنامج

$Delay = 250 * 200 * 5MC * 1\mu s = 250,000 \mu s$

وازا بدنا نضمّن ال overhead بحساباتنا بيصير عنا 250,800

---

## Review Questions

1. there are total of 5 pins in the PIC18F458
2. (True or False) * All of the PIC18F458 ports have 8 pins F
   * Upon power-up the I/O pins are configured as output ports F
3. Code a simple program to send 99H to PORTB and PORTC?

```
MOVLW 99H
MOVWF PORTB
MOVWF PORTC
```

* to make PORTB an output port, we must place 00H in register TRISB.
* to make PORTB an input port, we must place FFH in register TRISB.

4. Write a program to get 8 bit data from PORTC and send it to PORTs B & D?

```
    SETF TRISC
    CLRF TRISB
    CLRF TRISD
L1  MOVF PORTC, W
    NOP
    MOVWF PORTB
    MOVWF PORTD
    BRA L1
```

ازا في بالسؤال كلمة continuosly بحطها

5. Write a program to toggle all the bits of PORTB and PORTC continuosly?
   a- using the COMF instruction

```
    CLRF TRISB
    CLRF TRISC
L1  MOVLW 55H
    MOVWF PORTB
    MOVWF PORTC
    COMF PORTB
    COMF PORTC
    GOTO L1
```

## 4.2 I/O bit manipulation programming

In this section we will examine the PIC18 I/O instructions. We pay special attention to I/O bit manipulation because it's a powerful and widely used feature of the PIC family.

### [I/O Ports and bit-addressability]

* Sometimes we need to access only 1 or 2 bits of the port instead of the entire 8 bits.
* For all PIC ports, we can access either all 8 bits or any single bit without altering the rest.

Table: single-bit (bit-oriented) Instructions for PIC18

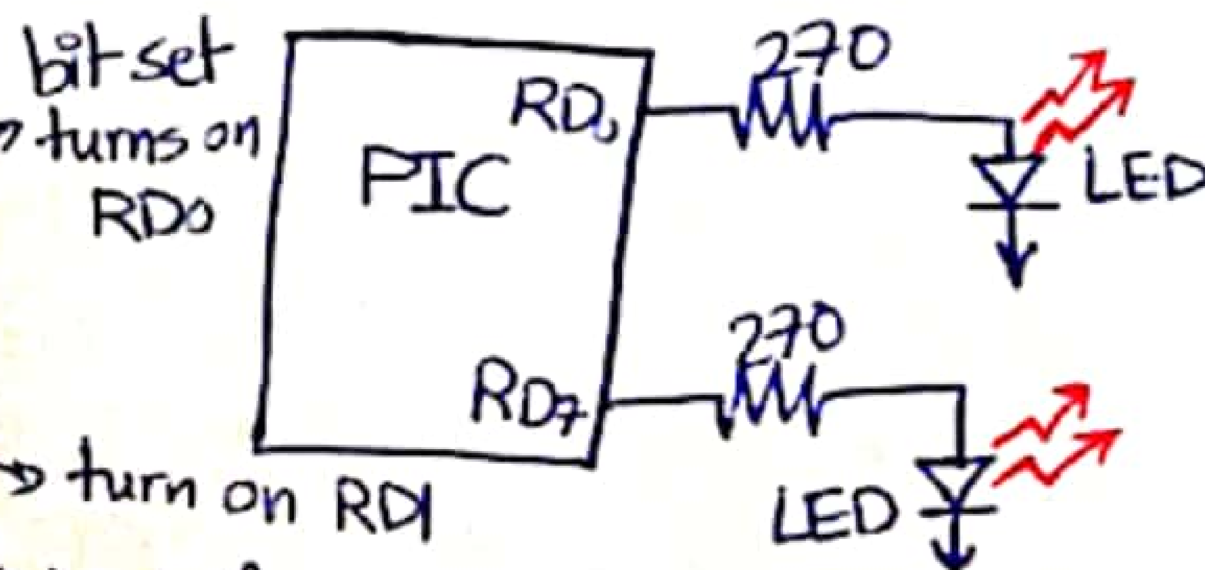
| | Instruction | Function |
|---|---|---|
| ① | BSF fileReg, bit | → Bit Set fileReg (set the bit: bit=1) |
| ② | BCF fileReg, bit | → Bit clear fileReg (clear the bit: bit=0) |
| ③ | BTG fileReg, bit | → Bit toggle fileReg (complement the bit) |
| ④ | BTFSC fileReg, bit | → Bit test fileReg, skip if clear (skip next instruction if bit=0) |
| ⑤ | BTFSS fileReg, bit | → Bit test fileReg, skip if set (skip next instruction if bit=1) |

1- "BSF fileReg, bit_num" (اللي بدنا نغيرها) to set HIGH a single bit of a given fileReg where this fileReg can be any location in the fileReg and bit_num is the desired bit no. from 0 to 7.

Although the bit-oriented instructions can be used for manipulation of bits $D_0$-$D_7$ of any fileReg, they are mostly used for I/O ports in embedded systems. For example "BSF PORTB, 5" sets HIGH bit 5 of PORTB

**Ex (4.2)** An LED is connected to each Pin of PORTD. Write a program to turn on each LED from ($D_0$-$D_7$). Call a delay module before turning on the next LED? (→ BSF)

```
CLRF TRISD       ; make PORTD an output port
BSF PORTD,0      ; bit set turns on RD0
CALL DELAY       ; delay before next one
BSF PORTD,1      ; turn on RD1
CALL DELAY       ; delay before next one
BSF PORTD,2
CALL DELAY
BSF PORTD,3
CALL DELAY
BSF PORTD,4
CALL DELAY
```



② "BCF fileReg, bit_number" to clear a single bit of a given fileReg.

* Remember that for I/O ports, we must activate the appropriate bit in the TRISx register if we want the pin to reflect the changes. for example, the following code toggles pin RB2 Continuosly :-

```
        BCF   TRISB , 2   → bit=0 , make RB2 an output pin
AGAIN   BSF   PORTB , 2   → bit set (RB2 = HIGH)
        CALL  DELAY
        BCF   PORTB , 2   → bit clear (RB2 = LOW)
        CALL  DELAY
        BRA   AGAIN       → continuosly
```

(Pin RB2 خليتها an output pin)

في فرق بين output pin
BCF TRISB, 2 ← output pin
و output PORT ← CLRF TRISB

Ex (4.3) Write the following programs :-
a) Create a square wave of 50 % duty cycle on bit 0 of PORTC
b) Create " " " " 66 % " " " " 3 of "

a – the 50 % duty cycle means that the (on) and (off) states (or the high & low portions of the pulse) have the same length.
therefore, we toggle RC0 with a time delay between each state.

```
        BCF   TRISC , 0   → clear TRIS bit for RC0 = out
HERE    BSF   PORTC , 0   → set to HIGH RC0 (RC0=1)
        CALL  DELAY       → call the delay subroutine, 50% on   (for "on" state)
        BCF   PORTC , 0   → RC0 = 0
        CALL  DELAY       → for 50% off                         (for "off" state)
        BRA   HERE        → keep doing it
```

on → BSF
off → BCF

or

```
        BCF   TRISC , 0   → make RC0 = Out
HERE    BTG   PORTC , 0   → complement bit 0 of PORTC
        CALL  DELAY       → call the delay subroutine
        BRA   HERE        → keep doing it
```

b) A 66 % duty cycle means that the "on" state is twice the "off" state

```
b–      BCF   TRISC , 3   → clear TRISC3 bit for output
BACK    BSF   PORTC , 3   → RC3 = 1
        CALL  DELAY       → call the delay subroutine   (for "on")
        CALL  DELAY       → twice for 66 %
        BCF   PORTC , 3   → RC3 = 0
        CALL  DELAY       → call delay once for 33 %     (for "off")
        BRA   BACK        → keep doing it
```

| 66 % → on |
|---|
| 33 % → off (يعني) |

∴ قيمة ال duty cycle اللي بيعطيني اياها by default بتمثل ال (on state) وما تبقى بتمثل مقدار ال off state



Scanned with CamScanner

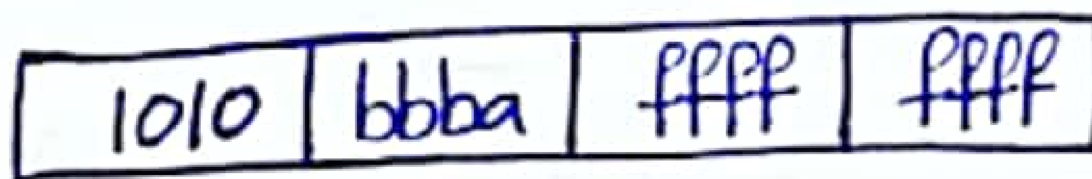
"BTG fileReg, bit_number" to toggle a single bit of a given fileReg

```
       BCF   TRISB,2   → make RB2 an output pin
BACK   BTG   PORTB,2   → toggle pin RB2 only
       CALL  DELAY
       BRA   BACK
```

* RB2 is the 3rd bit of PORTB (the first bit RB0, the second RB1, ...)

④ BTFSS (bit test fileReg, skip if set)

to monitor the status of a single bit for HIGH.
this instruction tests the bit and skips the next instruction if it's HIGH.



| 1010 | bbba | ffff | ffff |
|---|---|---|---|

$0 \leq f \leq FF$

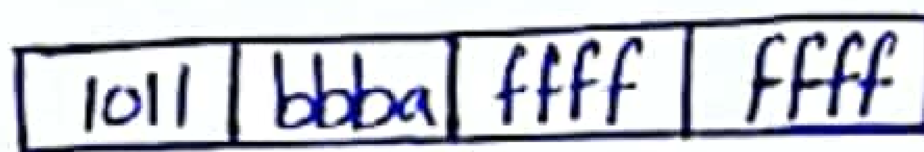
$0 \leq b \leq 7$

"BTFSS fileReg, bit_num"

بتقل الجملة اللي بعدها ← monitoring if it's high

⑤ BTFSC (bit test fileReg, Skip if clear)

to monitor the status of a single bit for LOW.
This instruction tests the bit and skips the instruction right below it if the bit is LOW.



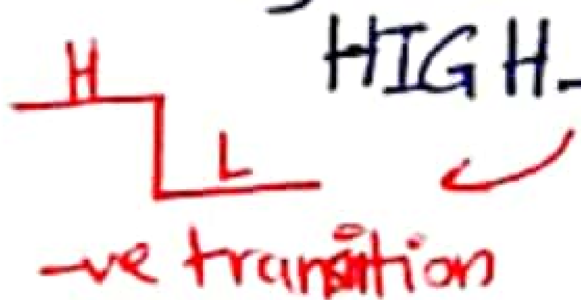
| 1011 | bbba | ffff | ffff |
|---|---|---|---|

$0 \leq f \leq FF$

$0 \leq b \leq 7$

Ex(4.4) Write a program to perform the following:-

a) Keep monitoring the RB2 bit until it becomes HIGH(1)
b) when RB2 (input) becomes HIGH, write 45H value to PORTC, and also send a HIGH-to-Low pulse to RD3 ? (output)



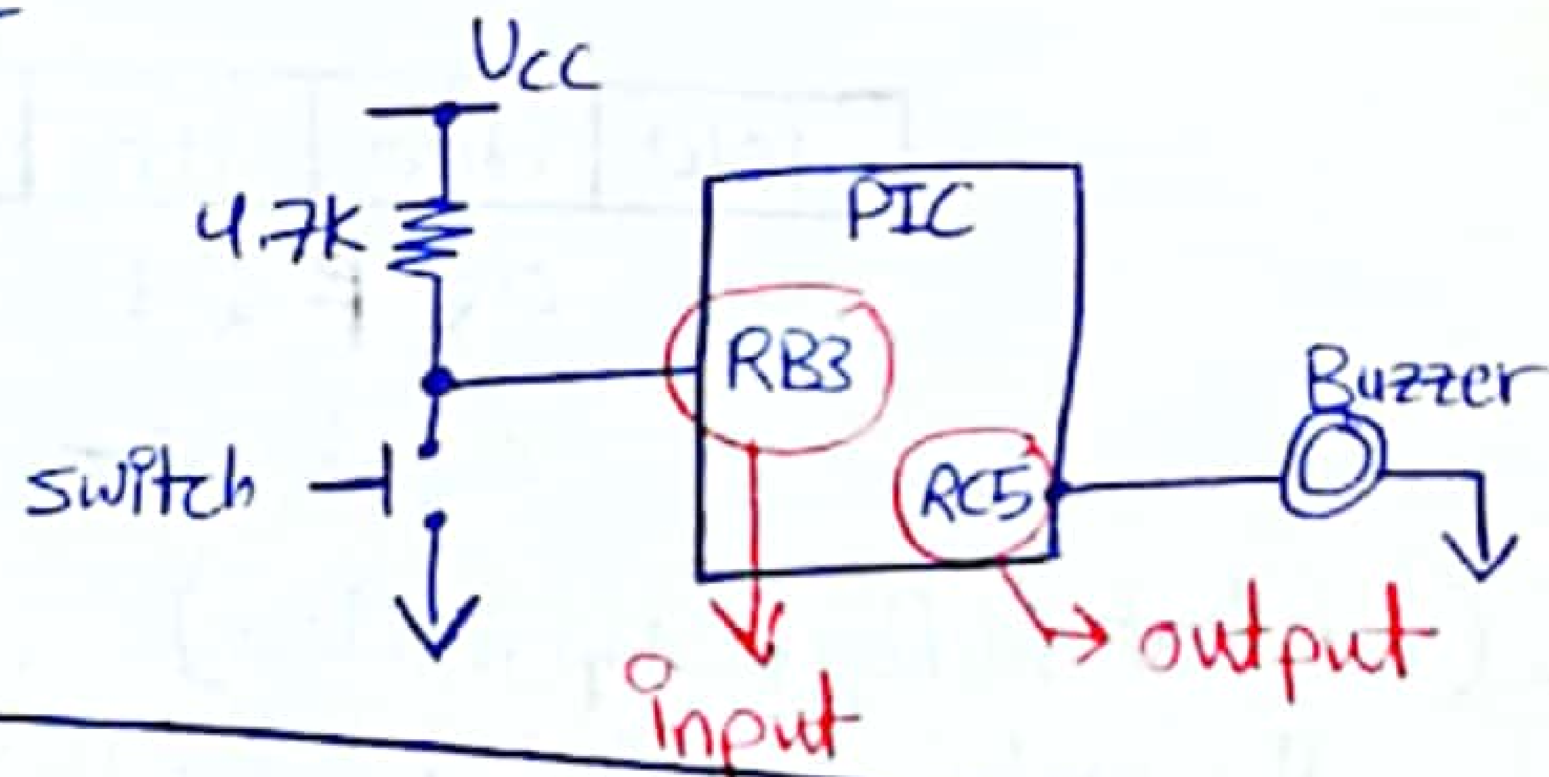

output PORT
CLRF TRISx

```
       BSF    TRISB,2    → RB2 input
       CLRF   TRISC      → PORTC output
       BCF    TRISD,3    → RD3 output
       MOVLW  45H        → WREG = 45H
AGAIN  BTFSS  PORTB,2    → bit test RB2 for HIGH
       BRA    AGAIN      → keep checking if Low
       MOVWF  PORTC      → issue WREG to PORTC
       BSF    PORTD,3    → bit set fileReg RD3 (H-to-L)
       BCF    PORTD,3    → bit clear fileReg RD3 (L)
```

Instruction الـ ... يضل بالـ loop طالما RB2 (Low) لما يصير HIGH بيطلع برا الـ loop وبيكتب الـ 45H في PORTC
on → off
It also sends a HIGH-to-low pulse to RD3

Ex(4.5) Assume that bit RB3 is an input and represents the condition of a door alarm. If it goes LOW it means that the door is open. Monitor the bit continuosly. Whenever it goes LOW, send a HIGH-to-Low pulse to PORT RC5 to turn on a buzzer

```
        BSF    TRISB,3    → RB3 input
        BCF    TRISC,5    → RC5 output
HERE    BTFSC  PORTB,3    → keep monitoring RB3 for LOW
        BRA    HERE       → stay in the loop
        BSF    PORTC,5    → RC5 High
        BCF    PORTC,5    → RC5 Low for H-to-L
        BRA    HERE
```




## Reading a single bit

We can use the bit test instructions to read the status of a single bit and send it to another bit or save it.

Ex(4.8) A switch is connected to pin RB0 and an LED to pin RB7. Write a program to get the status of SW and send it to the LED?

```
        BSF    TRISB,0    → RB0 input
        BCF    TRISB,7    → RB7 output
AGAIN   BTFSS  PORTB,0    → bit test RB0 for HIGH
        GOTO   OVER       → it must be Low
        BSF    PORTB,7    ← skip if set
        GOTO   AGAIN
OVER    BCF    PORTB,7
        GOTO   AGAIN
```

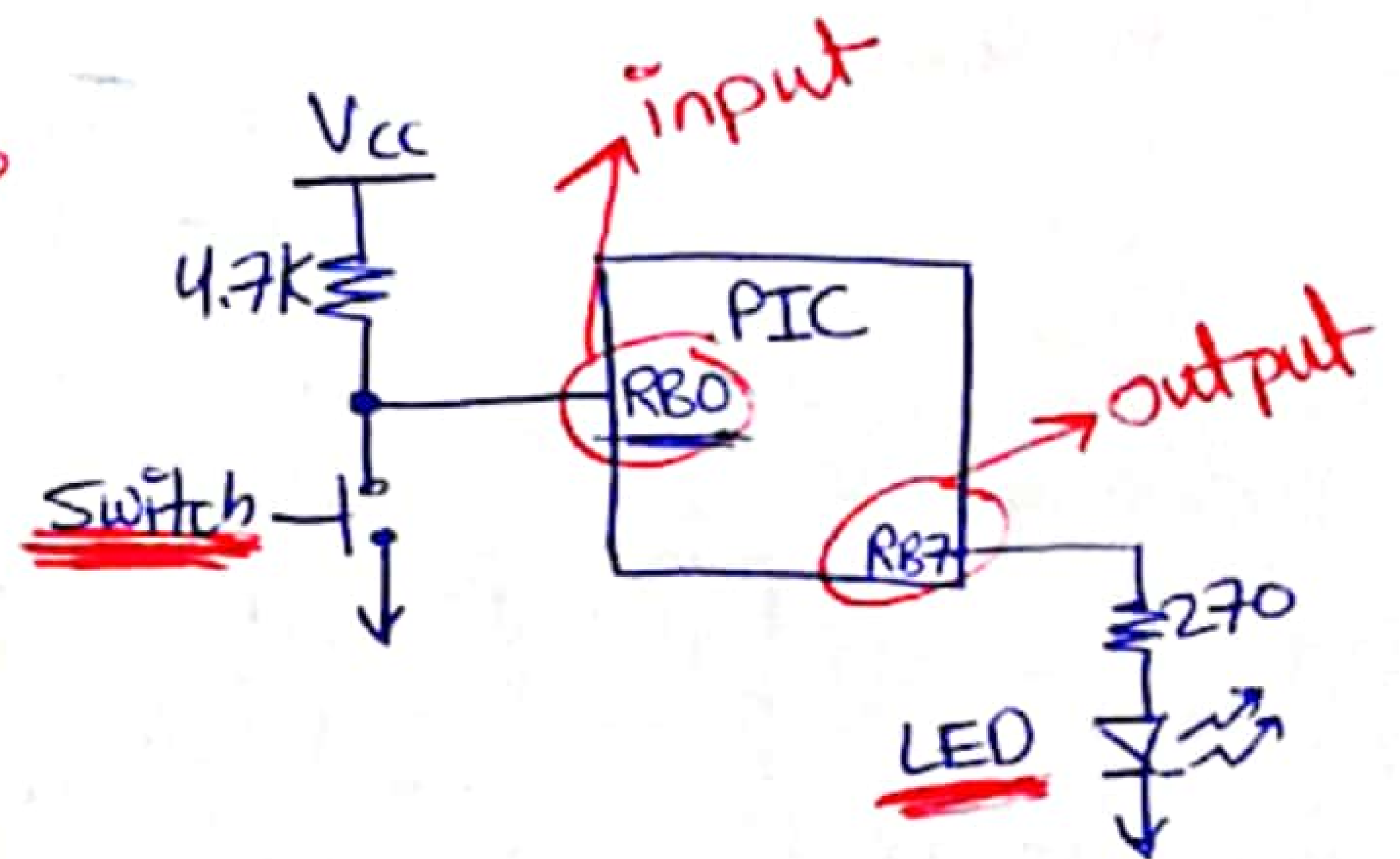
بيقرأ اللي جوا

عن طريق RB0 BTFSS

من اللوب كله

ازا ما تحقق الشرط وكان ال switch zero

منقدر نستخدم BRA مكانه




Scanned with CamScanner

## Reading input pins VS. LATx port

* In reading a port, some instructions read the status of the port pins while others read the status of an internal port latch called LATx.
* When reading ports there are two possibilities:-
  1. Read the status of the input pin
  2. 〃 〃 internal latch of the LAT register

## Reading LATx for port

Consider "COMF PORTB" instruction. The sequence of actions taken when such an instruction is executed is as follows:-

1. the instruction reads the internal latch of the LATB and brings that data into CPU
2. Complemented the data
3. the result is rewritten back to the LATB latch.
4. the data on the pins are charged only if the TRISB bits are cleared to 0's

* this is called (read-modify-write). To use it, the port must be configured as output:-

| Some of the instructions (Read-modify-write) |
|---|

| | | |
|---|---|---|
| ADDWF | fileReg, d | → add WREG from f |
| BSF | fileReg, bit | → bit set fileReg |
| BCF | fileReg, bit | → bit clear fileReg |
| COMF | fileReg, d | → Complement f |
| INCF | fileReg, d | → increment f |
| SUBWF | fileReg, d | → subtract WREG from f (F-W) |
| XORWF | fileReg, d | → exclusive-OR WREG with f |

XOR الـ bit كل مع نظيرتها ↲

| x | y | f |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Scanned with CamScanner

Fig (4.8) LATx register role in reading a port or latch

RD LAT

1 Data Bus x D Q x

WR PORT CLK $\overline{Q}$ 0

Data Latch

$V_{DD}$

P off

0 x

x D Q 0

WR TRIS CLK $\overline{Q}$ 1

TRIS LATCH

N off

$V_{SS}$

TRIS = x

RD TRIS

TTL or SCHMITT TRIGGER

Q D

En

RD PORT

## Review Questions

AGAIN BTFSS PORTB, 2
GOTO OVER

1. Write instructions to get the status of RB2 and put it on RB0?

```
      BSF TRISB , 2
      BCF TRISB , 0
AGAIN BTFSS PORTB, 2
      BRA OVER
      BSF PORTB, 0
      BRA AGAIN
OVER  BCF PORTB, 0
      BRA AGAIN
```

2. Write instructions to toggle both bits of RD7 and RD0 continuosly?

Pins

BTG

```
   BCF TRISD, 0
   BCF TRISD, 7
H2 BTG PORTD, 0
   BTG PORTD, 7
   GOTO H2
```

Scanned with CamScanner

# Ch.5 Arithmetic, Logic instructions and programs

This Chapter describes all PIC arithmetic and Logic instructions.
Program examples are given to illustrate the application of these instructions

## 5.1 Arithmetic Instructions →

Unsigned no.'s defined as (data in which all the bits are used to represent data, and no bits are set aside for the +ve or -ve sign.
this means that the operand can be between (00–FF)H for 8bit data.

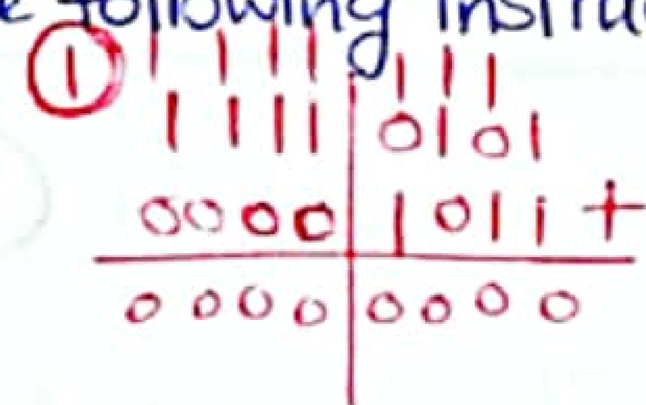
# Addition of unsigned no.'s :

① "ADDLW K" , WREG = K+WREG

Ex(5.1) Show how the flag register is affected by the following instructions :-

```
MOVLW 0F5H
ADDLW 0BH
```

0F5 + 0B →



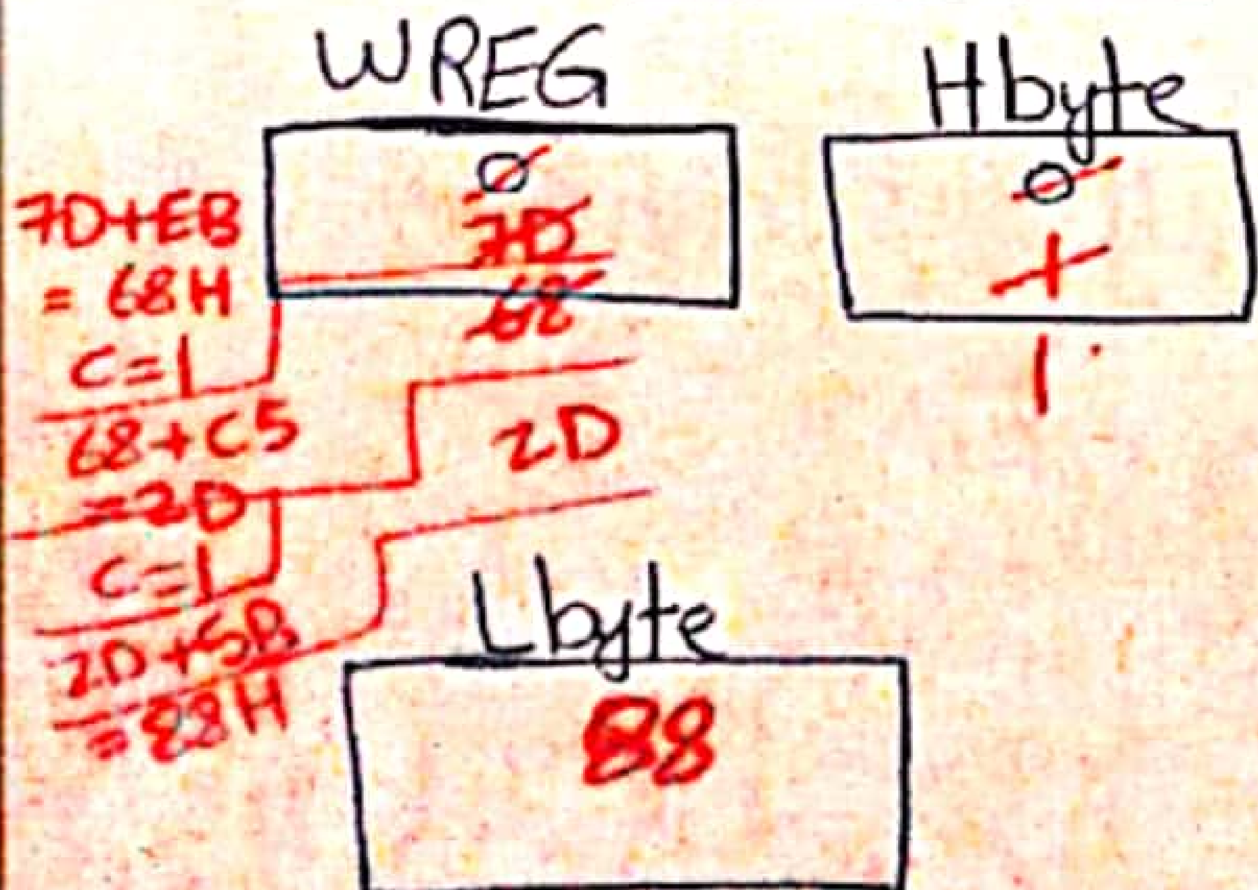

WREG

C=1 → there's a carry out from $D_7$
DC=1 → there's a carry from $D_3$ to $D_4$
Z=1 → the result is zero in WREG

② "ADDWF fileReg, d"

Ex(5.2) Assume that fileReg RAM Locations 40–43H have the following hex values. Write a program to find the sum of the values. At the end of the program, location 6 of the fileReg should contain the low byte and location 7 the high byte of the sum.

40 → 7D
41 → EB
42 → C5
43 → 5B




```
Hbyte EQU 0x07
Lbyte EQU 0x06

      MOVLW 0H
      MOVWF Hbyte
      ADDWF 0x40, W
      BNC N1
      INCF Hbyte, f    → تحقق ما
N1    ADDWF 0x41, W
      BNC N2
      INCF Hbyte, f    → تحقق [C=1]
N2    ADDWF 0x42, W
      BNC N3
      INCF Hbyte, f    → تحقق [C=1]
N3    ADDWF 0x43, W
      BNC N4
      INCF Hbyte, f    ↑ ما تحقق
N4    MOVWF Lbyte
```

[Lbyte = 88H]

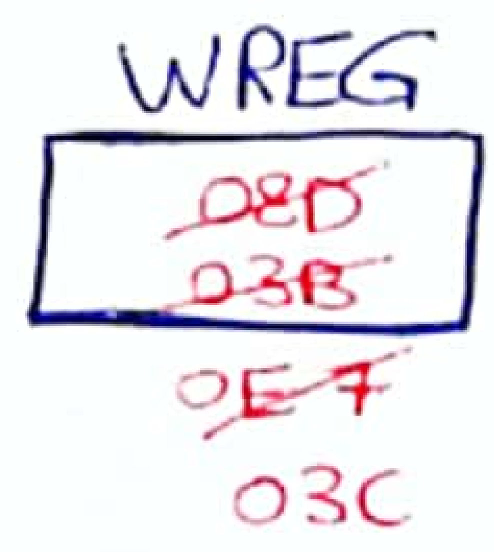
Scanned with CamScanner

③ ADDWFC (ADDW and fileReg with carry), adding two 16 bit numbers →

Ex(5-3)

Write a program to add two 16 bit no.'s. The numbers are 3CE7H and 3B8DH. Assume that fileReg location 6 = (8D) and location 7 = (3B). Place the sum in fileReg locations 6 and 7; 6 should have the lower byte.

```
→ MOVLW  08DH
  MOVWF  0x06
  MOVLW  03BH
  MOVWF  0x07
  MOVLW  0E7H
  ADDWF  06H, f     ← for the lower byte     → 0E7 + 08D =
  MOVLW  03CH                                → 03C + 03B =
  ADDWFC 07H, f     ← for the higher byte
```



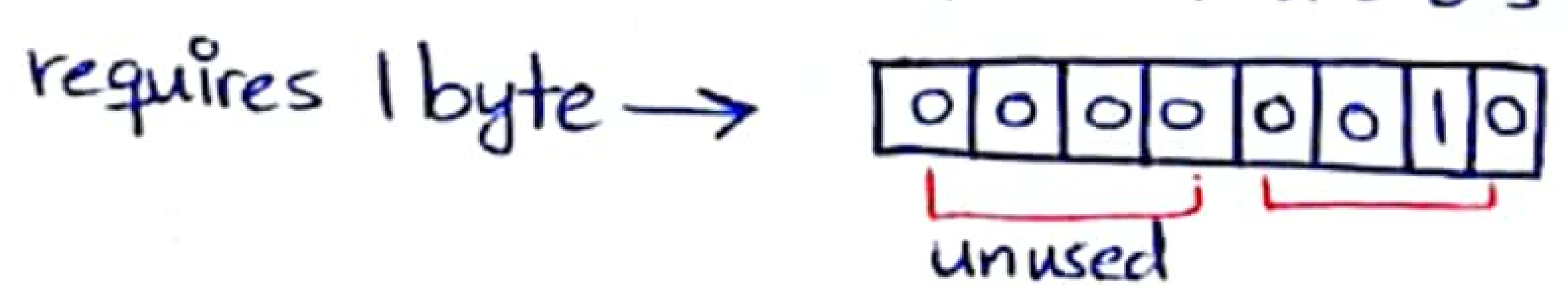
| Address | Data |
|---|---|
| 06H | 08D |
| 07H | 03B |

# BCD no. system (Binary Coded Decimal) →

* We use it in everyday life
* In computer literature, one encounters two terms for BCD numbers :

① Unpacked BCD : the lower 4 bits are just used and the rest are 0's

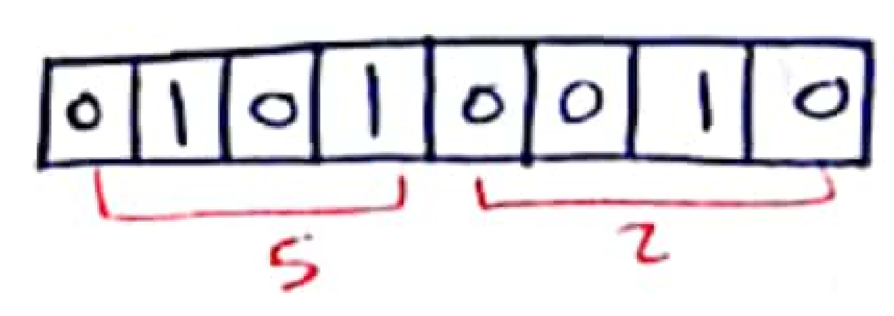
requires 1 byte →



| Digit | BCD |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |

② Packed BCD → a single byte has two BCD no.'s in it — in the lower 4 bit, " " higher 4 bit

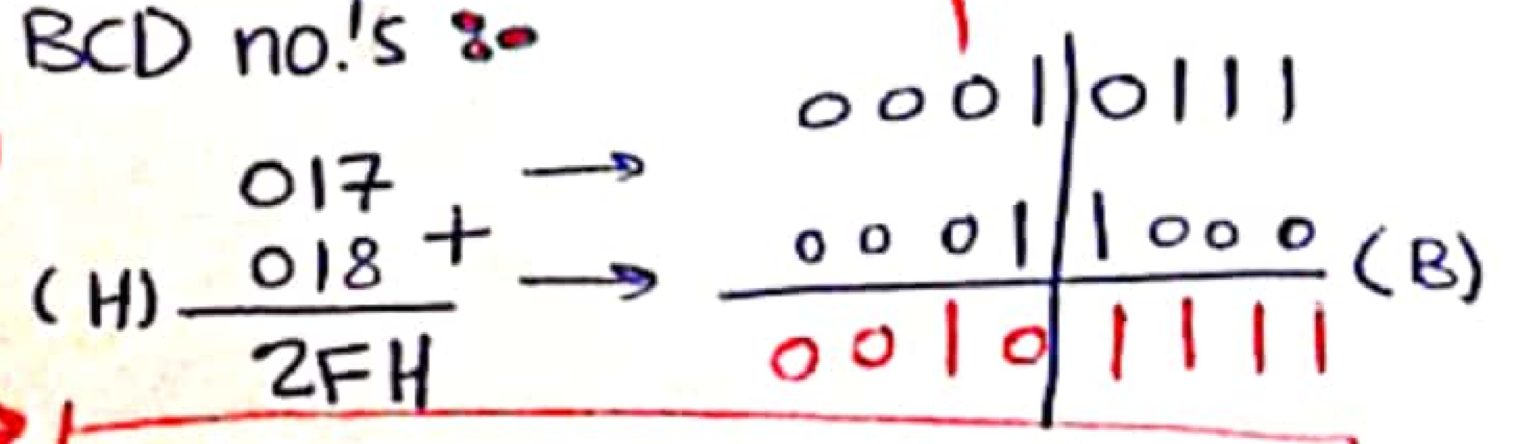
requires 1 byte of memory is needed to store it → 



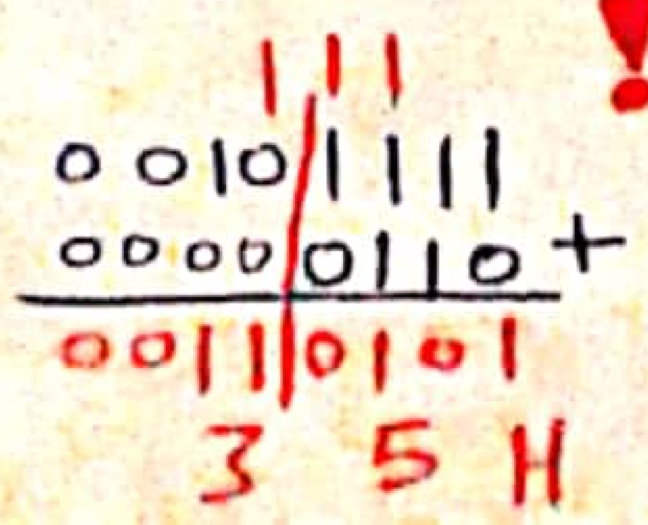
[so, it's efficient in storing data]

* the problem of using adding BCD no.'s :

```
MOVLW  017H
ADDLW  028H
```



which isn't BCD ⇐ (2FH)



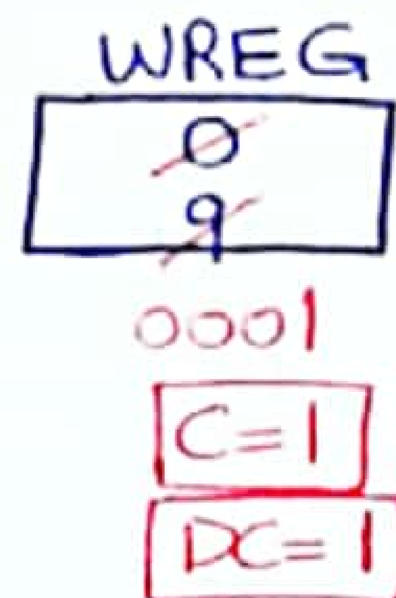
6 نضيف عشان نرجع (BCD no) لازم

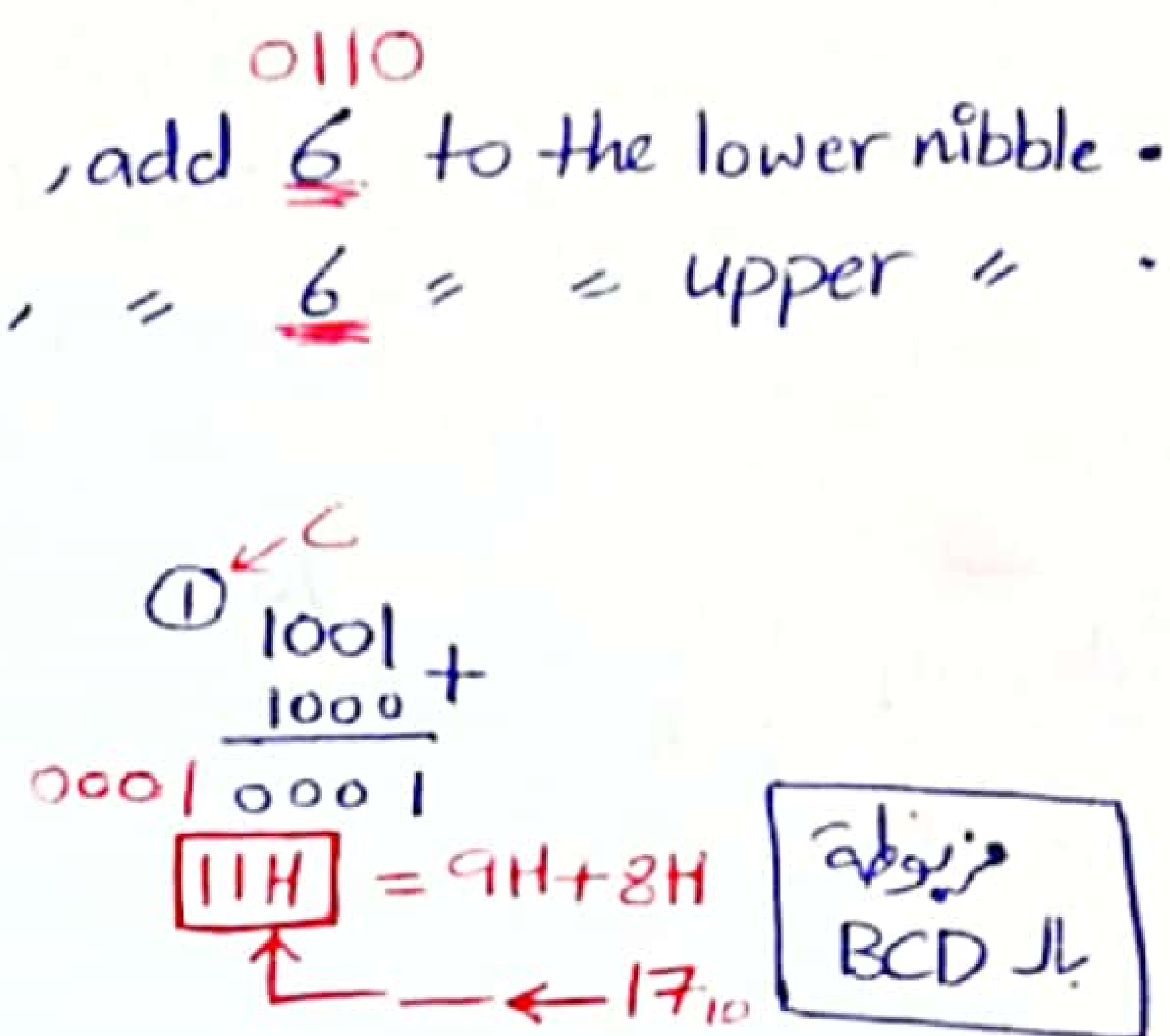
# DAW (Decimal Adjust WREG)

* it works only with an operand in the WREG
* add 6 to the lower or higher nibble if needed
* after execution →
  - If the lower (4bit) > 9 , or if DC=1 , add 6 (0110) to the lower nibble.
  - 〃 〃 upper (4bit) > 9 , 〃 〃 C=1 , 〃 6 〃 〃 upper 〃 .

**Ex ①**

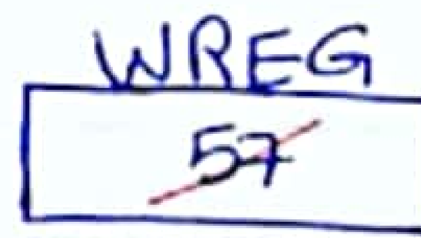
```
MOVLW  00H
ADDLW  09H
ADDLW  08H
DAW
```

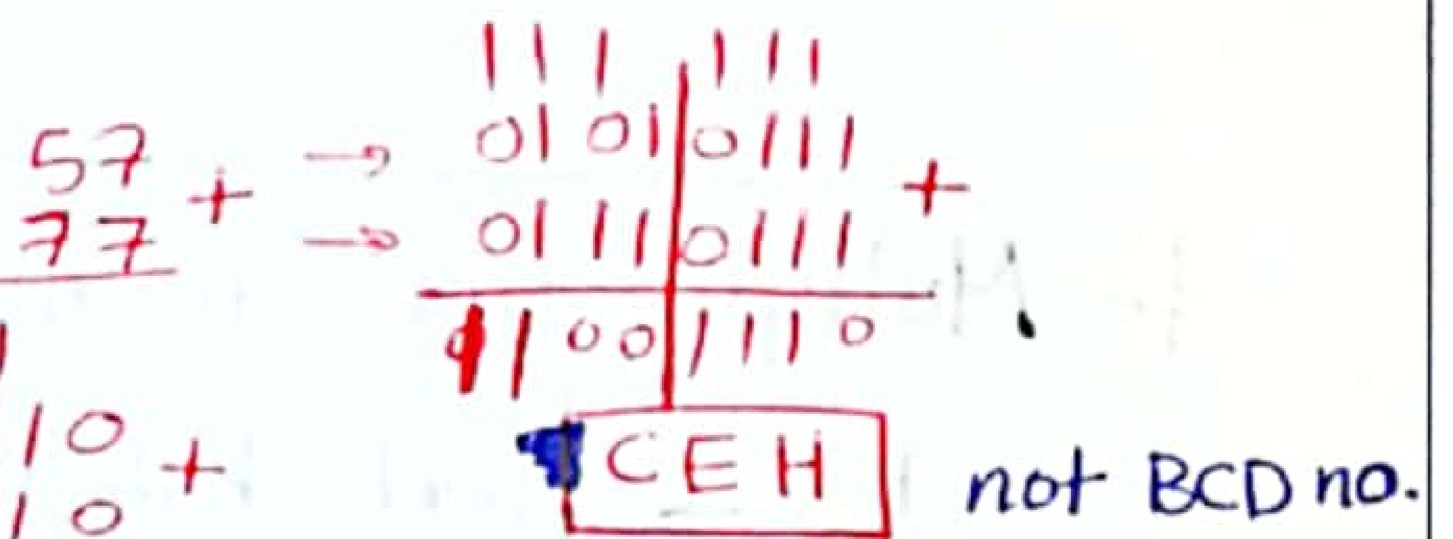


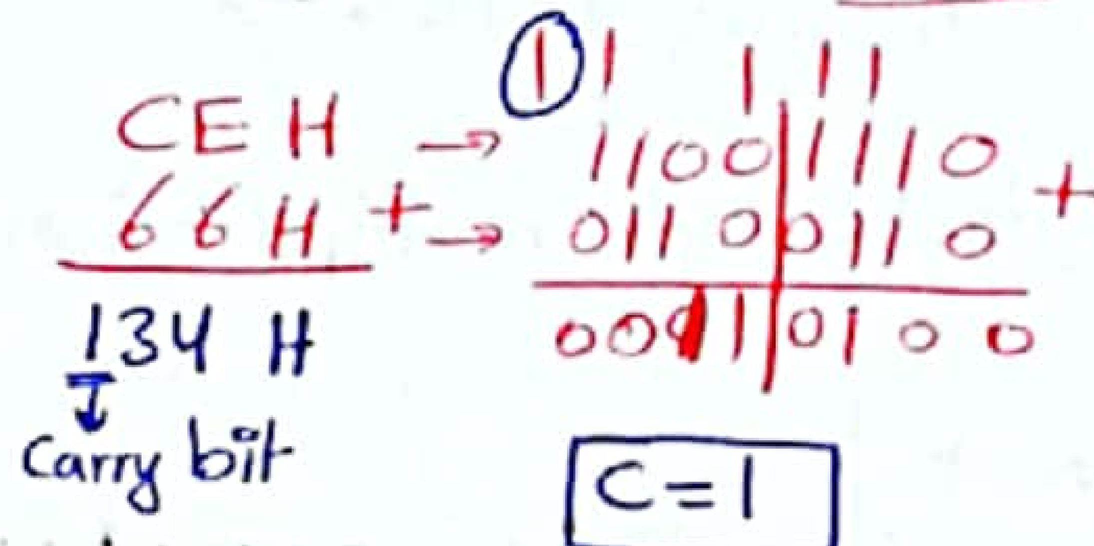




**Ex ②**

```
MOVLW  57H
ADDLW  77H
DAW
```



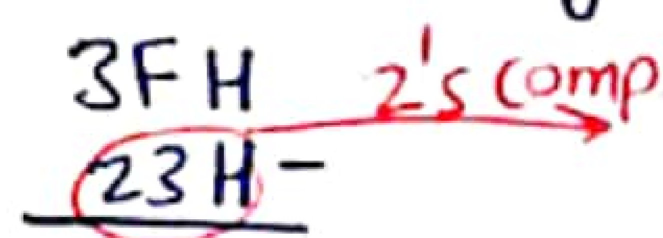

# Subtraction of unsigned no.'s →

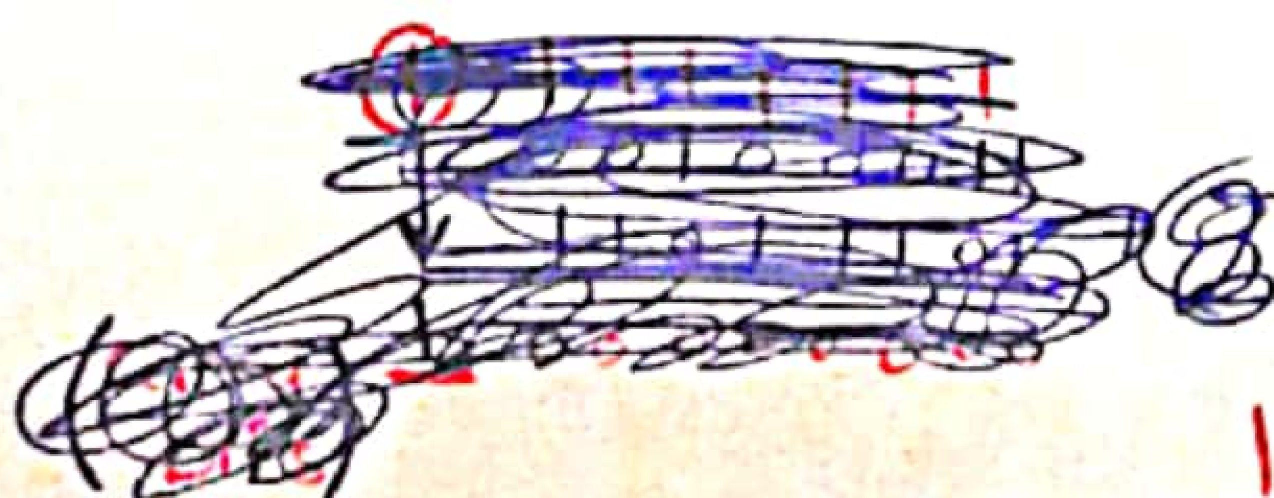
* In the PIC18 we have 4 instructions for subtraction : [SUBLW , SUBWF (f-W), SUBWFB (subtract with borrow), SUBFWB]
* ① SUBLW k (WREG = k - WREG)
  - take the 2's complement of WREG operand
  - add it to the k operand

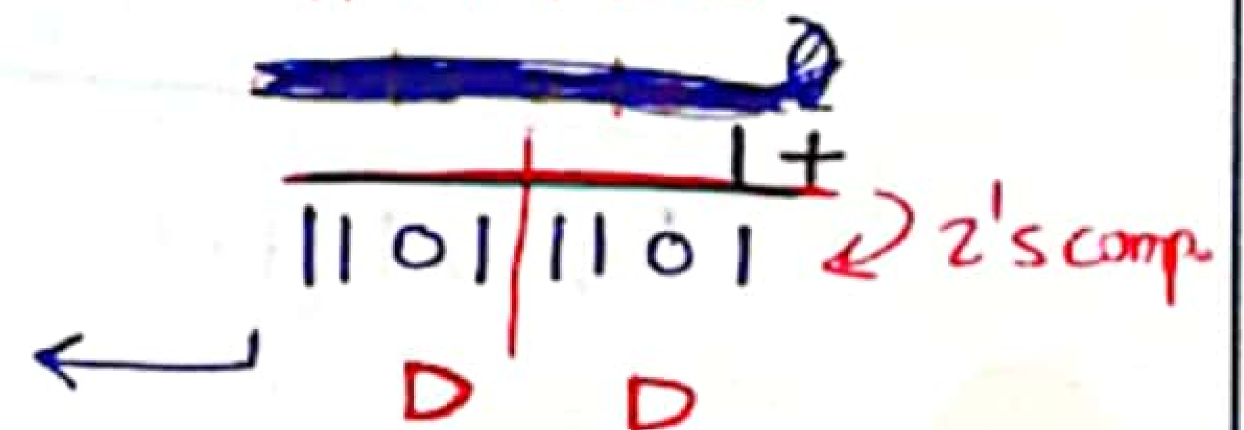
  ] they're performed by the internal hardware of the CPU

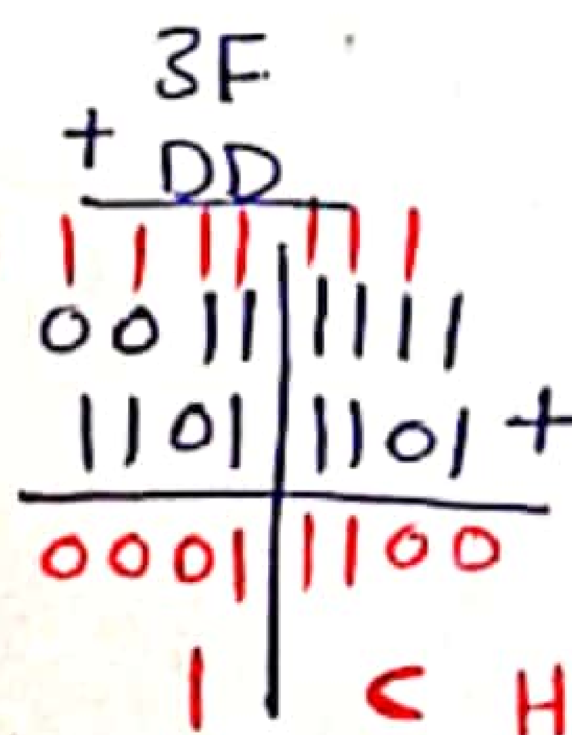
**Ex (5.5)** show the steps involved in the following :-

```
MOVLW  23H
SUBLW  3FH
```








the flags : N=0, C=1, DC=1, Z=0 → if C=0 or N=1 , the result is (-ve)

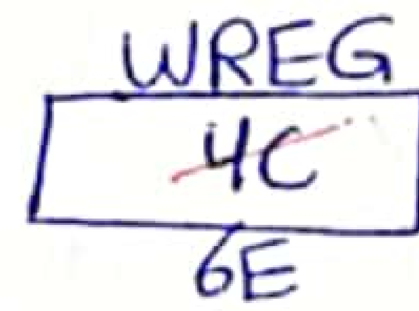


Scanned with CamScanner

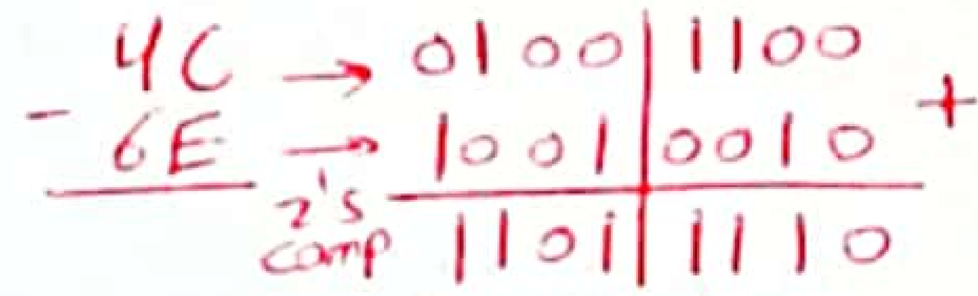
Ex(5.6) Write a program to subtract 4C−6E ? [using SUBWF]

```
→  MYREG   EQU 0X20
   MOVLW   4CH
   MOVWF   MYREG
   MOVLW   6EH
   SUBWF   MYREG, W
   BNN     NEXT
   NEGF    WREG
NEXT MOVWF MYREG
```

F−W




| address | Data |
|---|---|
| 0X20 | 4C |

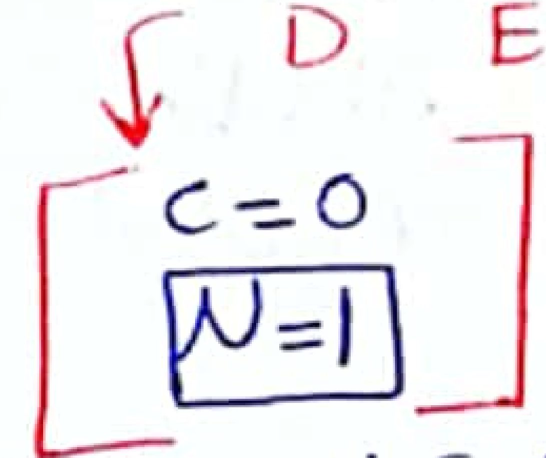


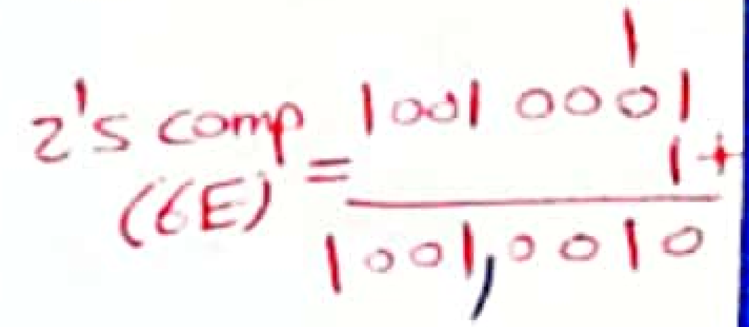

ازا كان الناتج −ve → NEGF WREG → تحقق

2's comp صافر الـ WREG للـ مناظر مساوي للـ (22H)

NEXT MOVWF MYREG ← حط الناتج صورت ازا ماكان −ve

لم تتحقق




the result is (−ve) cuz C=0

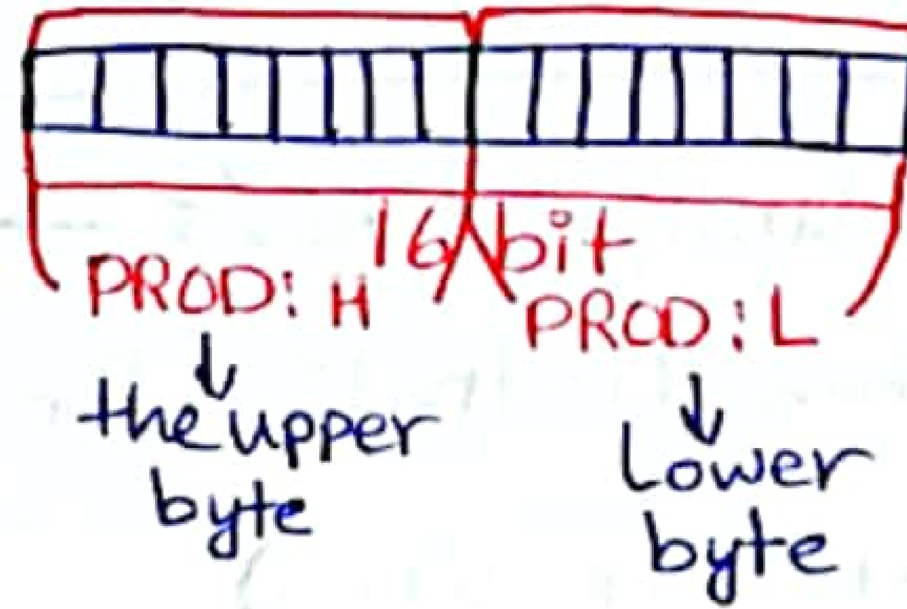
---

# Multiplication of unsigned no. →

* PIC supports byte-by-byte multiplication, the bytes are assumed to be unsigned data.

[MULLW k] → WREG = k * W

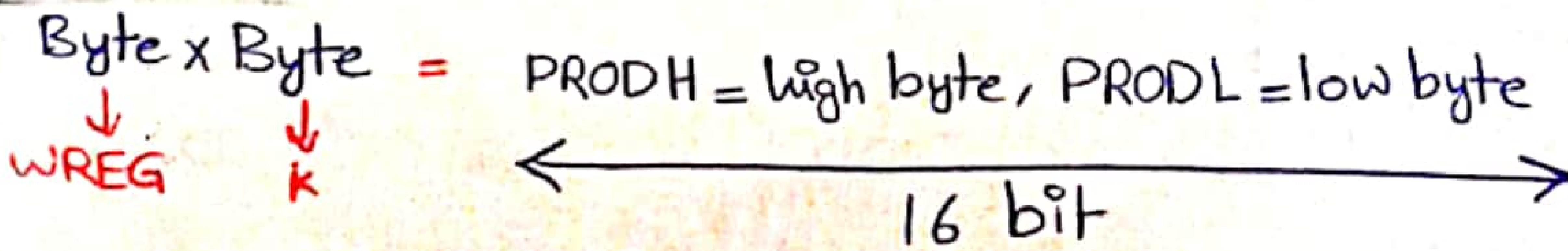
- one of the operand must be in WREG, and the second must be a literal value k




Ex

```
MOVLW 25H  → (WREG=25H)
MULLW 65H  → 25*65 =
```

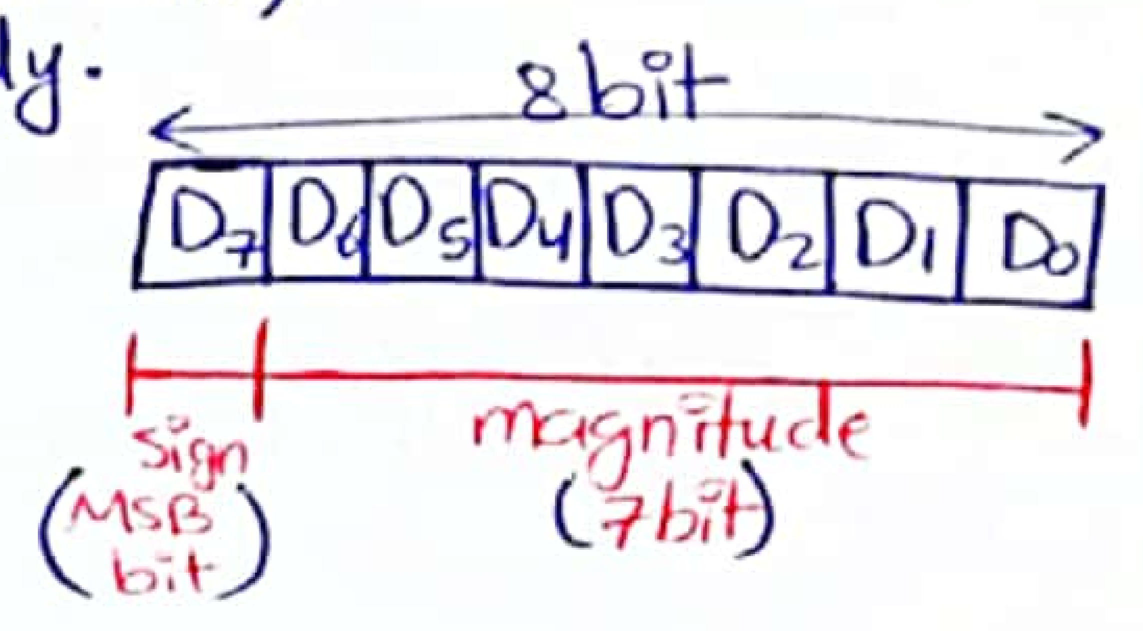
---




Scanned with CamScanner

## 5.2 Signed number concepts and arithmetic operations

* Many applications in everyday life requires signed data (to be +ve/-ve)
* The MSB is set aside for the sign → 0 (+ve no.)
  → 1 (-ve no.)
  the rest of 7 bit used for the magnitude only.
* You have 128 (-ve no.'s) and 127 (+ve no.'s)




* To convert to (-ve no.) representation:
  takes 2's comp. for this no. by

  1's comp
  1 +
  ---
  2's comp.

# Overflow problem in signed no. operations →

* An overflow occurs when the result of an operation is too large for the register.
* OV flag indicate whether the result is valid or not
  If OV=1, the result is erroneous
* When is the ov flag set? "In 8 bit signed no."
  1. there's a carry (D6-D7) but no carry out of D7 (C=0)
  2. there's a carry from D7 out (C=1) but no carry from (D6-D7)

∴ **OV = 1** If there's (a carry out from D7) or (a carry from D6 to D7) But not both

**OV=0** If there's (a carry from D6 to D7) and (from D7 out)

Ex 1. Examine the following code and analyze the result, including N & OV flags

→ MOVLW +d'96' → 1001 0110
ADDLW +d'70' → 0111 0000 +

0000 0110
$D_7\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0$
↓
+ve no. 0 6 H

OV = 0
N = 0
sum = 06H = $4_{10}$

2. MOVLW _D'128' →
-D'2' → 1110 +

-128
1100 1000
0011 0111 1's comp
1 +
0011 1000
3 8 H



Scanned with CamScanner

## 5.3 Logic and compare instructions

* the widely used logic instructions :

```
ANDLW   k          → WREG and K
ANDFW   fileReg,d
IORLW   k
IORFW   fileReg,d
XORLW   k
XORFW   fileReg,d
```

* It effects on only Z and N flags.

Ex ① MOVLW 35H
ANDLW 0F → WREG = 35H → W = 35 AND 0F

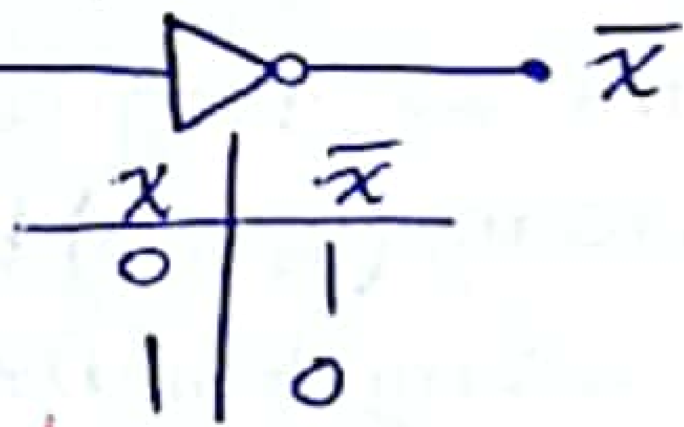
| | |
|---|---|
| 35H : | 0011 0101 |
| 0FH : | 0000 1111 (and) |
| | 0000 0101 |
| | 0 5 |

∴ WREG = 05H

نفس المبدأ على ال OR , XOR

| | | |
|---|---|---|
| 0 | متشابه | 00 , 0 / 11 , 0 |
| 1 | مختلف | 01 , 1 / 10 |

# [Complement Instructions in the fileReg] →




| $x$ | $\bar{x}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

COMF fileReg , d → taking 1's comp. of the fileReg contents

* It effect only Z & N flags.

# NEGF [negative fileReg] → this takes the 2's comp. of a fileReg. , it effect all flags.

Ex. Find the 2's comp. of 85H. 85H is (−123) :

```
MYREG   EQU   0X10
        MOVLW 85
        MOVWF MYREG
        NEGF  MYREG
```

| | |
|---|---|
| 85H = | 1000 0101 |
| 1's = | 0111 1010 |
| | 1 + |
| | 0111 1011 |

2's comp. = (7 B)H

# Compare Instruction →

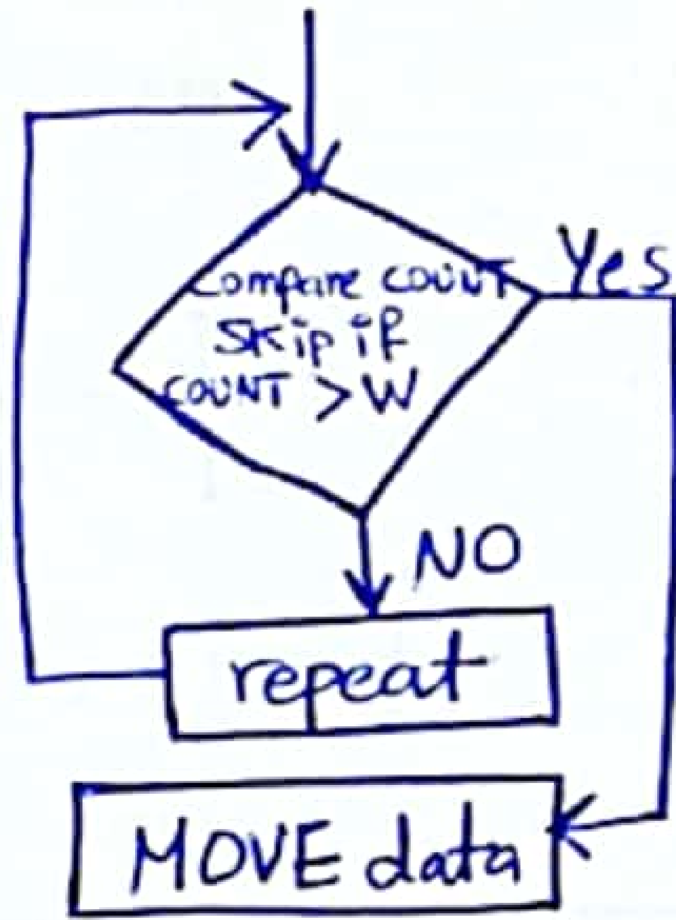
these instructions take 1/2 cycle(s)

CPFSGT fileReg → compare fileReg with WREG, skip if greater than (fileReg > W)
(GT = greater than)

CPFSEQ fileReg → 〃 〃 〃 〃 , 〃 〃 equal (f = W)
(EQ = equal)

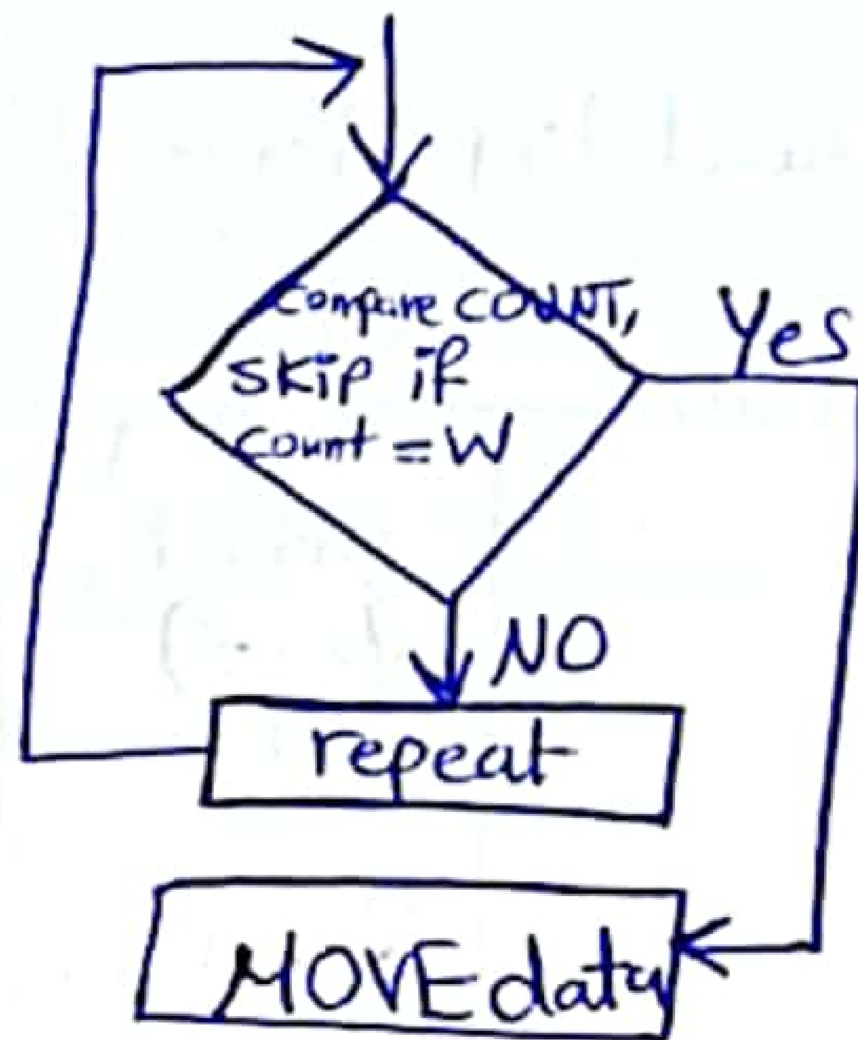
CPFSLT fileReg → 〃 〃 〃 〃 , 〃 〃 less than (f < W)
(LT = less than)

Fig(1) Flowchart for CPFSGT :-



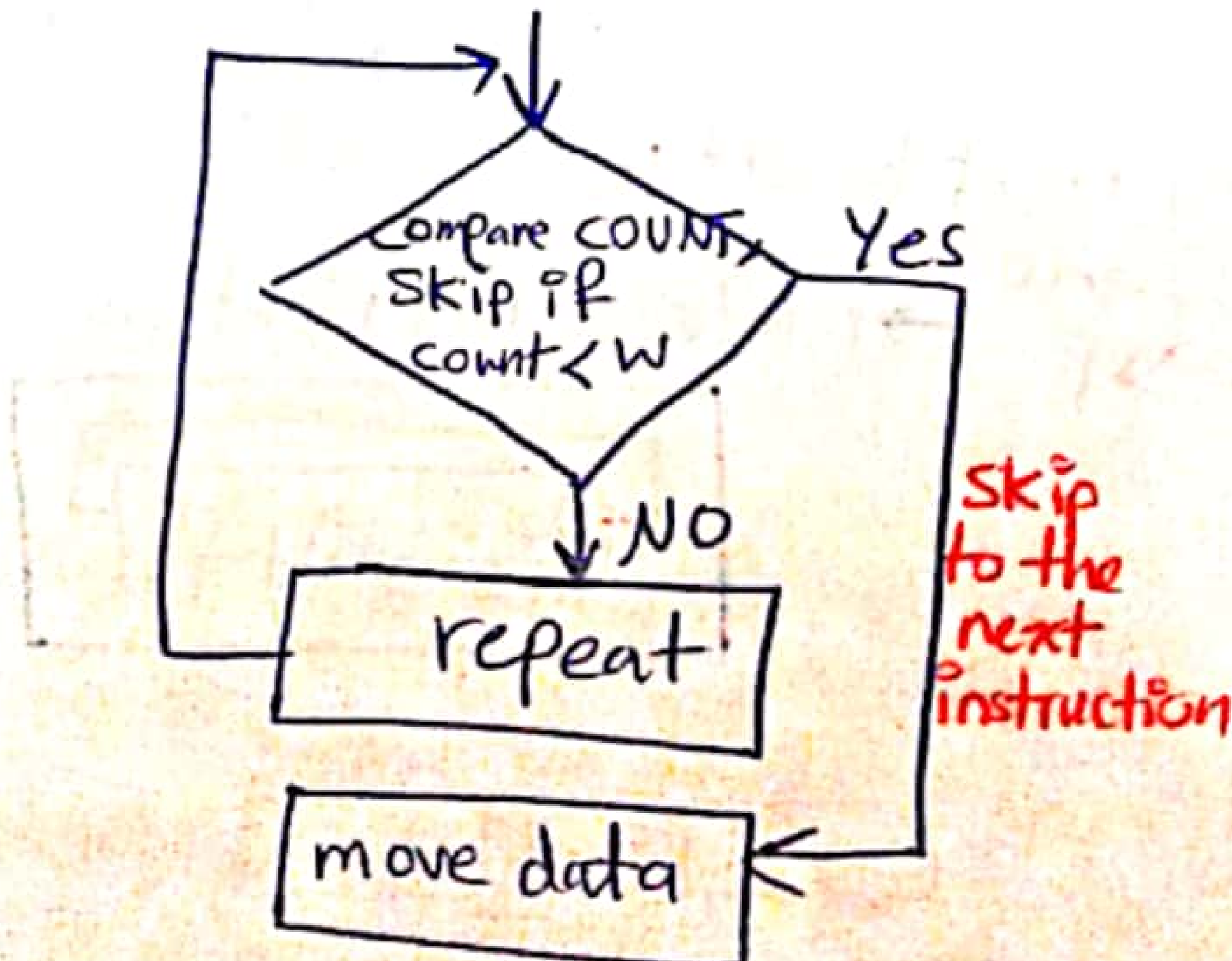

```
AGAIN  CPFSGT  COUNT
       GOTO    AGAIN
       MOVWF   PORTB
```

Fig(2) Flowchart for CPFSEQ :-




```
AGAIN  CPFSEQ  COUNT
       GOTO    AGAIN
       MOVWF   PORTB
```

Fig(3) Flowchart for CPFSLT :-




```
AGAIN  CPFSLT  COUNT
       GOTO    AGAIN
       MOVWF   PORTB
```

Scanned with CamScanner

**Ex (5.27)** Write code to determine if data on PORTB contains the value 99H. If so, write letter 'Y' to PORTC; otherwise, make PORTC = 'N' ?

('Y' ← ascii code)

```
→   CLRF   TRISC   ← output PORTC
    SET    TRISB   → input PORTB
    MOVLW  A'N'
    MOVWF  PORTC
    MOVLW  99H
    CPFSEQ PORTB
    BRA    OVER
    MOVLW  A'Y'
    MOVWF  PORTC
    OVER . . . . . .
```

CPFSEQ ← بتقارن بين قيمة الداتا بال WREG والـ fileReg يعني PORTB لل = 99

BRA OVER / MOVLW A'Y' ← ازا كانوا متساوين (تحقق شرط السؤال)

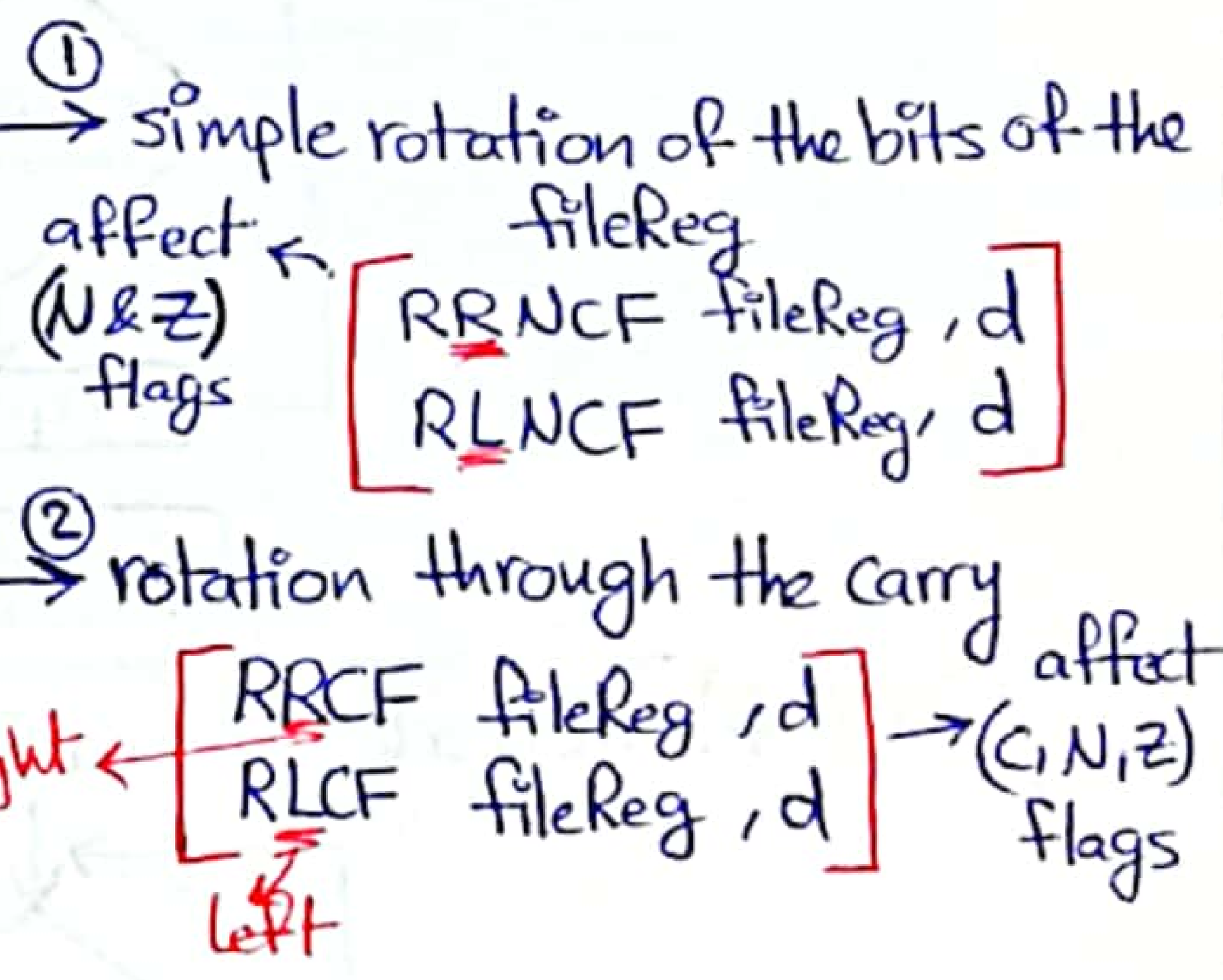
```
    CLRF   TRISC
    SETF   TRISB
    MOVLW  99 H
    CPFSEQ PORTB
    BRA    OVER
    MOVLW  A'Y'
    MOVWF  PORTC
OVER MOVLW A'N'
    MOVWF  PORTC
    END
```

(CPFSEQ → skip if equal)

---

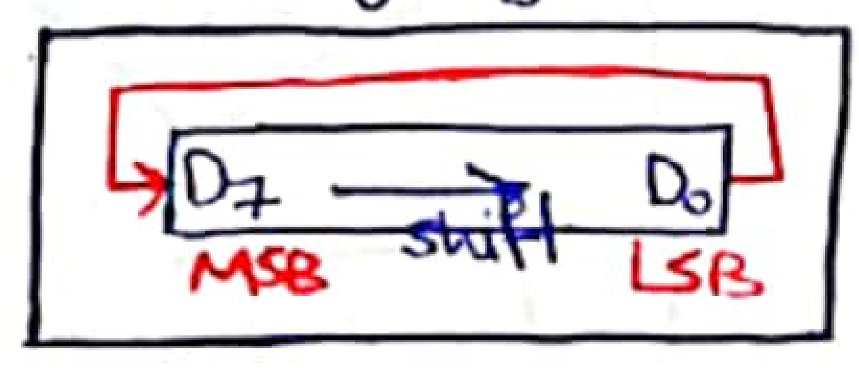
## 5.4 | Rotate instruction and data Serialization

* In many applications there's a need to perform a bitwise rotation of an operand.
* In the PIC18 the rotation instructions




***1* Rotating the bits of fileReg right to left :-**

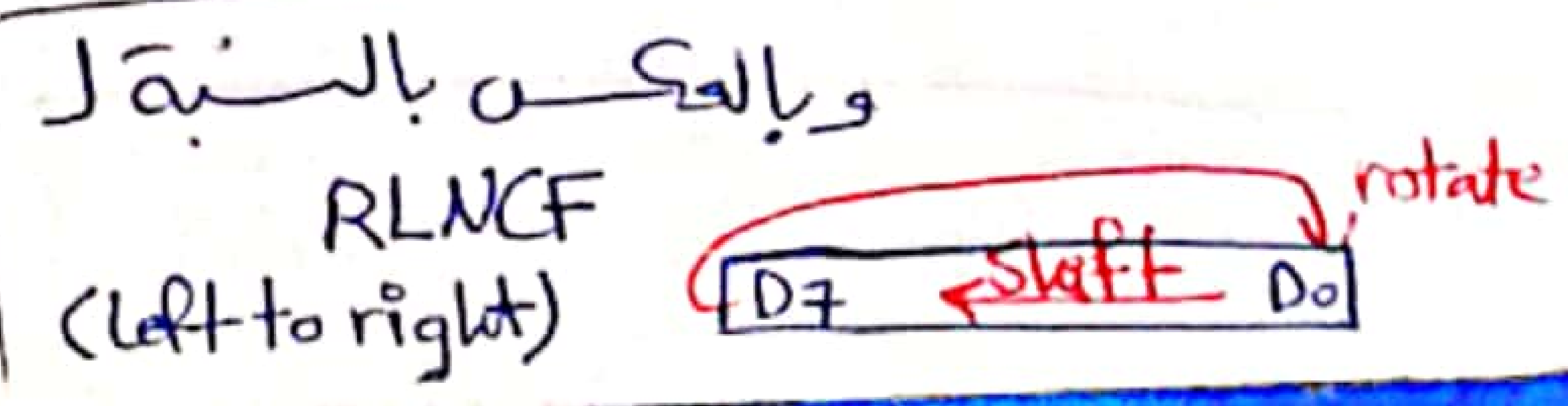
[RRNCF fileReg , d] (d → W / f) → rotate fileReg right (no carry)




**Ex**

```
          MYREG  EQU  0X20
                 MOVLW 36H
                 MOVWF MYREG
00011011 ← RRNCF MYREG , f
10001101 ← RRNCF MYREG , f
11000110 ← RRNCF MYREG , f
01100011 ← RRNCF MYREG , f
```

وبالعكس بالنسبة ل RLNCF (Left to right)

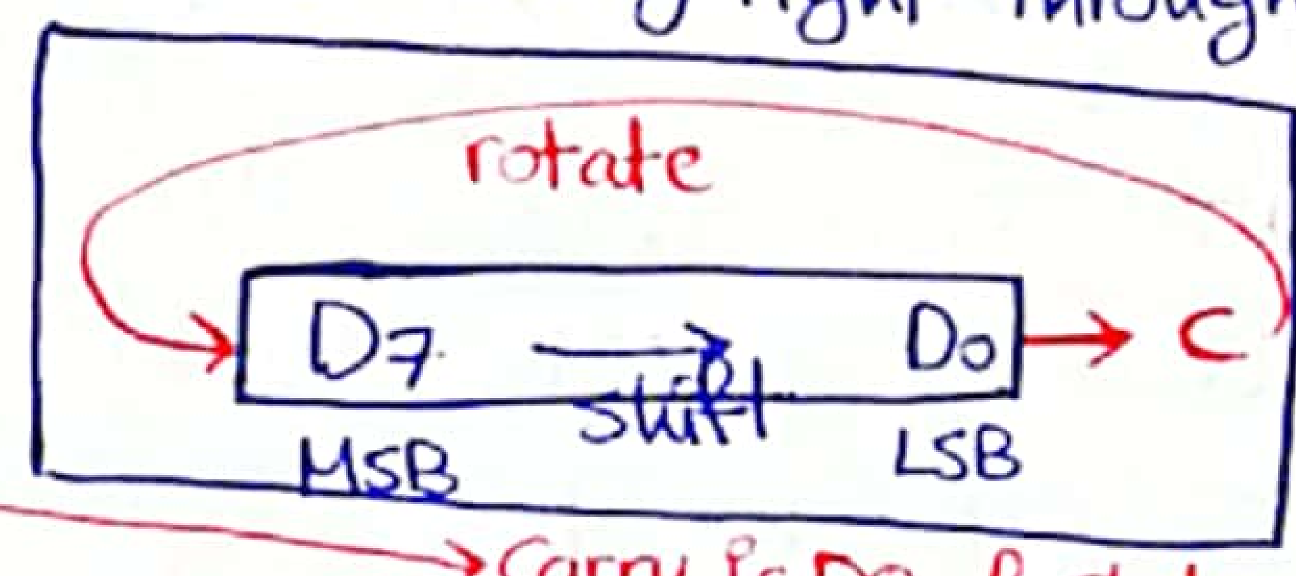





Scanned with CamScanner

*2* rotating through the carry → [the carry bit, making the register 9bit]

[RRCF fileReg , d] , rotate fileReg right through a carry



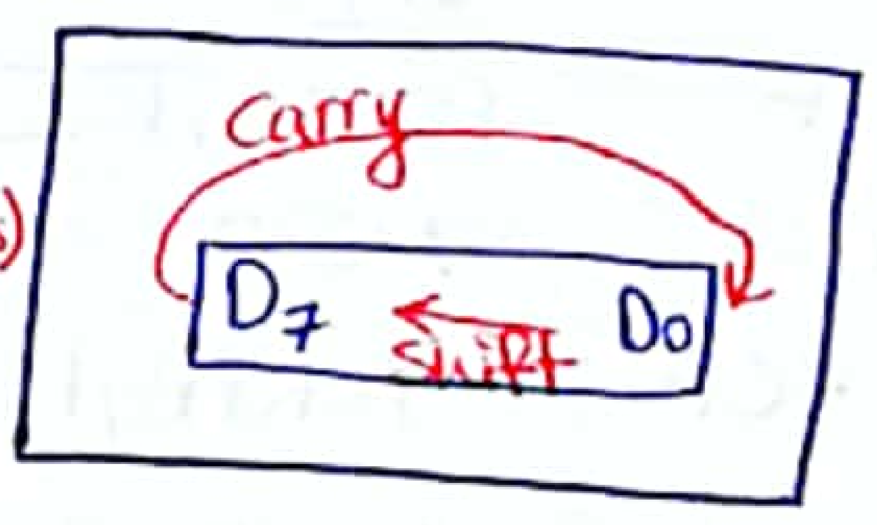

from R to L

Ex
```
MYREG   EQU 0X20
BCF     Status , C      → C=0
MOVLW   26H             → WREG = 0010 0110      Carry is D0 of Status
MOVWF   MYREG           → MYREG = 00100110
RRCF    MYREG , f       → = 0001 0011 , C=0
RRCF    MYREG , f       → = 0000 1001 , C=1
RRCF    MYREG , f       → = 10000100  , C=1
```

إذا كان D0
① في Carry
⓪ في Carry
تنتقل قيمة الـ Carry إلى D7

* [RLCF fileReg , d] , rotate fileReg left to right through a carry

Ex
```
MYREG   EQU 0X20
BSF     STATUS , C      ; make C=1 (D0 status)
MOVLW   15H
MOVWF   MYREG           → = 00010101
RLCF    MYREG , F       → = 00101011 , C=0
RLCF    MYREG , f       → = 01010110 , C=0
RLCF    MYREG , f       → = 10101100 , C=0
RLCF    MYREG , f       → = 01011000 , C=1
```




9bit register

ما يطلع من D7 إلى D0 يمر بالـ Carry

# Serializing Data → "make the data serial"

It's one of the most widely used applications of the rotate instruction.

* Sending a byte of data, one bit at a time through a single pin of mc :-

① using the serial port

② Using a programming technique to transfer data one bit at a time and control the sequence of data and spaces between them.

Ex(5.28) Write a program to transfer value 41 serially (one bit at a time) via pin RB1. Put one high at the start and end of the data. Send the LSB first?

→

```
        COUNT   EQU  0X20    → fileReg location for counter
        MYREG   EQU  0X21    → fileReg location for rotate
        BCF     TRISB, 1     → RB1 output pin
        MOVLW   41H          → WREG = 41    ] value to be serialized
        MOVWF   MYREG                       ]
        BCF     Status, C    → C=0
        MOVLW   08H          ]
        MOVWF   COUNT        ] load the counter
        BSF     PORTB,1      → RB1 = high
AGAIN   RRCF    MYREG, f     → rotate right via carry
        BNC     OVER
        BSF     PORTB, 1     → set the carry bit to PB1
        BRA     NEXT
OVER    BCF     PORTB, 1
NEXT    DECF    COUNT, f
        BNZ     AGAIN
        BNZ     PORTB, 1     → RB1 = high
```

---

Ex(5.29) Write a program to bring in a byte of data serially (one bit at a time) via pin RC7 and save it in fileReg location 0X21. The byte comes in with the LSB first?

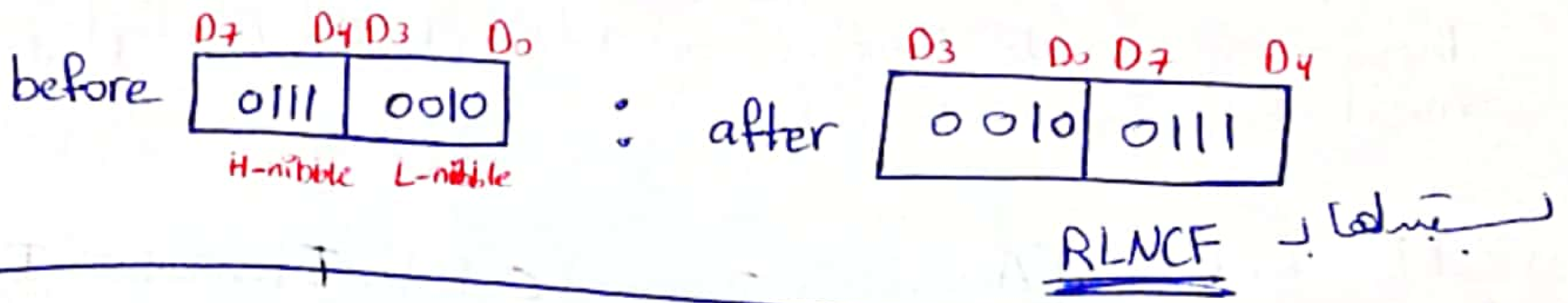
```
        COUNT   EQU 0X20     → fileReg location for counter
        MYREG   EQU 0X21     → "  "  → incoming byte
        BSF     TRISC, 7     → input RC7 pin
        MOVLW   8H           ]
        MOVWF   COUNT        ] → Load the counter
AGAIN   BTFSC   PORTC, 7     → if RC7 = 0, skip
        BSF     STATUS, C    ← status carry
                             → C=1
        BTFSS   PORTC, 7     → skip if RC7 = 1, if not ①
        BCF     STATUS, C    → C=0, otherwise
        RRCF    MYREG, f     → rotate right carry into MYREG
        DECF    COUNT, f     → decrement the counter
        BNZ     AGAIN        → repeat until COUNT = 0
                               now loc 21H has the byte
```



Scanned with CamScanner

# SWAPF → [SWAPF fileReg , d]

* It works on the fileReg.
* It swaps the lower nibble (4 bit) and the higher nibble (4 bit)




## Review Questions

① What's the value of MYREG in the fileReg after the following codes executed;

```
* MYREG EQU 0X40
    MOVLW  0X25
    MOVWF  MYREG      = 00100101
    RRNCF  fileReg,f  = 10010010
    RRNCF  "    "     = 01001001
    RRNCF  "    "     = 10100100
    RRNCF  "    "     = 0101,0010  ← answer  52H

* MYREG EQU 0X40
    MOVLW  0X25
    MOVWF  MYREG      = 0010 0101
    RLNCF  MYREG,f    = 01001010
    RLNCF  "    "     = 10010100
    RLNCF  "    "     = 00101001
    RLNCF  "    "     = 0101,0010  52H ← answer

* MYREG EQU 0X20
    CLRF   MYREG      → MYREG = 0000 0000   C=1
    BSF    STATUS,C   ; C=1
    RRCF   MYREG,F    → = 1000 0000   C=1
    BSF    STATUS,C   ; C=1
    RRCF   MYREG,f    → = 11000000
```

C0H : Answer

## 5.5 BCD and ASCII Conversion →

* Many newer mc's have a real time clock (RTC), where the time & da[te] are kept even when the power is off.
  These mc's provide the time and date in BCD. To display them, they must convert BCD values to ASCII.

### ASCII numbers [American Standard Code for Information Interchange]

On ASCII keyboards, when the key "0" is activated, "011 0000" is provided to the computer
3(0)H

* (3(1)H) → 0011 0001 provided for key "1" and so on.

| key | ASCII (hex) | Binary | | BCD (unpacked) | |
|---|---|---|---|---|---|
| 0 | 30 | 0011 | 0000 | 0000 | 0000 |
| 1 | 31 | 0011 | 0001 | 0000 | 0001 |
| 2 | 32 | 0011 | 0010 | 0000 | 0010 |
| 3 | 33 | 0011 | 0011 | 0000 | 0011 |
| 4 | 34 | 0011 | 0100 | 0000 | 0100 |
| 5 | 35 | 0011 | 0101 | 0000 | 0101 |
| 6 | 36 | 0011 | 0110 | 0000 | 0110 |
| 7 | 37 | 0011 | 0111 | 0000 | 0111 |
| 8 | 38 | 0011 | 1000 | 0000 | 1000 |
| 9 | 39 | 0011 | 1001 | 0000 | 1001 |

8 bit

# Packed BCD to ASCII Conversion →

* The RTC provides the date and the time in Packed BCD, regardless of whether the power is on or off.
* This data to be displayed on a device such as in LCD, it must be in ASCII format

| Packed BCD | Unpacked BCD | ASCII |
|---|---|---|
| 29 H | 02H & 09 H | 32H & 39 H |
| 0010 1001 | 0000 0010 & 0000 1001 | 0011 0010 & 0011 1001 |

* ASCII to Packed BCD conversion → نحولها لـ unpacked (to get rid of the high bit)

Ex ~~3~~(4)

| key | ASCII | Unpacked BCD | Packed BCD |
|---|---|---|---|
| 4 | 34 | 0000 0100 (04) | 01000111 |
| 7 | 37 | 00000111 (07) | 47H |

47H : answer

نأخذ الـ unpacked



Scanned with CamScanner

Ex (5.32) Assume that register WREG has packed BCD. Write a program to convert packed BCD to two ASCII no.'s and place them in fileReg locations 6 and 7?

→

```
BCD_VAL   EQU   0X29
L-ASC     EQU   0X06   ] set aside fileReg locations
H-ASC     EQU   0X07

MOVLW   BCD_VAL   → W = 29H , packed BCD
ANDLW   0FH       → (W = 09) mask the upper nibble
IORLW   30H       → make it an ASCII , W = 39H ('9')
MOVWF   L-ASC     → save it (L-ASC = 39H) ASCII char.
MOVLW   BCD_VAL   → W = 29H get BCD data once more
ANDLW   0F0H      → mask the lower nibble (W = 20H)
SWAPF   WREG, W   → swap nibbles (W = 02H)
IORLW   30H       → make it an ASCII , W = 32H ('2')
MOVWF   H_ASC     → save it (H_ASC = 32H ASCII char)
```

Scanned with CamScanner

# Ch6) Bank Switching, Table Processing, Macros and Modules

\# Addressing modes: the ways that the CPU can access the data, that could be in a register, or in memory, or provided as an immediate value.

- immediate value → Immediate (Literal)
- a register → Indirect Registor (FSRx)
- memory → Indexed ROM

The PIC18 provides a total of 4 distinct addressing modes →

1- Immediate
2- Direct
3- Register indirect
4- Indexed ROM

---

## 6.1 Immediate and Direct addressing modes →

* In immediate addressing mode, the operand is a literal (immediate data) constant, comes after the opcode.

this mode used to load info. into WREG and selected register, not to any file Register.

used also to perform arithmatic and logic operations on WREG only.

```
MOVLW   25H       → load WREG by 25H value
MOVLW   D'62'
SUBLW   B'0100'   → subtract WREG from '0100'
ANDLW   D'10'     → AND WREG with "10 decimal"
```

منقدر كمان نستخدم الـ immediate data مع الـ directives (EQU), مثلاً:

```
COUNT   EQU   0X20
 ...     ...
MOVLW   COUNT    / WREG = COUNT = 20H
```

(K) literal value

* In direct addressing mode → " register direct mode "

[ Recall
the 256 byte access bank file register is split into 2 sections → (00-7FH) to GPR's
→ (F80-FFFH) to SFR's
unused location فيها شوية SFR لل

the access bank is the default bank when the PIC18 is powered up. ]

→ the operand data is in a RAM location whose address is known and given as a part of the instruction.

Ex
MOVLW 56H → WREG = 56 H / an immediate addressing mode
direct mode → MOVWF 40H → copy WREG into fileReg. RAM location 40H
direct mode → MOVFF 40H, 50H → copy data from 40H location to 50H

# What is the difference between INCF fileReg, W
INCF fileReg, F ?

the difference is in the destination bit that gives the option to saving the result in the fileReg itself, or in WREG.
(the default destination is in f)

# What's the difference between DECFSZ and DECF ?
the differences between them are in the → operation
→ branch

---

* SFR registers and their addresses ⇒

- The SFR's can be accessed by their names (which is much easier) or by their addresses.

للتوضيح :
[ PORT B has address F81H ]

MOVWF F81 H
MOVWF PORT B
← نفس الشي بس الأفضل يتعرف باسمه PORTB

note | WREG is one of the SFR, has the address FE8H
Program counter (PC) is part of the SFR

Scanned with CamScanner

## 6.2 Register indirect addressing mode [FSR / INDF]

# - A register is used as a pointer to the data RAM Location

In PIC18, three registers are used for this purpose: FSR0, FSR1, FSR2

(File Select Register (FSR), a 12 bit reg. allowing access to the entire 4096 bytes of data RAM space in the PIC18.)

- FSR split by PIC18 into 2 sections → FSRxH
  → FSRxL

- We use LFSR to load the RAM address.

In other words, when FSRx are used as pointers, they must be loaded first with the RAM addresses as shown below:-

```
LFSR 0 , 30H   → load FSR0 with 30H
LFSR 1 , 40H   →  "   FSR1  "  40H        it needs 2 cycles
LFSR 2 , 6FH   →  "   FSR2  "  6FH
         n        → address in RAM memory locations
```

# The INDF is another register that associated with the reg. indirect addressing mode.

- Each of FSR0, FSR1, FSR2 registers has an INDF register associated with it (INDF0, INDF1, INDF2)

When we move data into INDFx we're moving data into a RAM location Pointed to by the FSR.

* In the same way, when we read data from the INDF register, we're reading data from a RAM location pointed to by the FSR.

```
LFSR 0 , 0X30   → FSR0 = 30H RAM Location pointer
MOVWF INDF0     → ;Copy contents of WREG into RAM
                  ;location whose address is held by
                  ;12 bit FSR0 reg.
```

WREG = FSR0 = 30H → [ ]

Scanned with CamScanner

* Advantages of register Indirect addressing mode →
- It makes accessing data dynamic (as direct addressing / static)
- Looping is possible to increment the address

**Ex(6.2)** Write a program to copy the value 55H into RAM memory locations 40H to 45H using :-

a) Direct

```
MOVLW  55H
MOVWF  40H
MOVWF  41H
MOVWF  42H
MOVWF  43H
MOVWF  44H
MOVWF  45H
```

حط ال 55 بال FSR0 الجديدة اللي موقعها صار 41

b) Indirect w/o a loop

```
MOVLW  55H
LFSR   0,40H        ; load the pointer, FSR0 = 40
MOVWF  INDF0        ; copy W to FSR0
INCF   FSR0L,f      ; FSR0 = 41
MOVWF  INDF0
INCF   FSR0L,f
MOVWF  INDF0
INCF   FSR0L,f
MOVWF  INDF0
INCF   FSR0L,f
MOVWF  INDF0
```

c) a loop

```
COUNT EQU 0X10      ; عنوان العداد
MOVLW  5H           ; COUNT = 5
MOVWF  COUNT
LFSR   0,0X40
MOVLW  55H
B1  MOVWF  INDF0
INCF   FSR0L,f
DECF   COUNT,f
BNZ    B1           ; Loop until COUNT = 0
```

⇓

the most efficient way

* بتعرّف من ال INDF → FSRx
* ك address لوين مروح ال data

← منلاحظ بأنه يجب استعمال INCF FSR0L,f → to increment the pointer

ومنلاحظ بأنه ما منقدر نساوي looping بهالحالة عند ال [direct addressing mode]

## * Auto_increment option for FSR →

- the normal increment for the pointer by using "INCF FSR0L,f" can cause problem since it increments 8 bit, when an address such as 5FFH is incremented.
  the (INCF FSR0L, f) will not propagate the carry into the FSR1H reg.
- Auto Inc/Dec solve the problem (without affecting the status flags) by using CLRF instruction → just for the entire 12 bits of the FSRn

| Instruction | | Function |
|---|---|---|
| CLRF INDFn | → | After clearing fileReg pointed by FSRn, the FSRn stays the same. |
| CLRF POSTINCn (post increment) | → | After clearing fileReg pointed by FSRn, the FSRn is incremented (Like x++) |
| CLRF PREINCn | → | The FSRn is incremented, then fileReg pointed to by FSRn is cleared (like ++x) |
| CLRF POSTDECn (post decrement) | → | After clearing fileReg pointed by FSRn, the FSRn is decremented (Like x--) |
| CLRF PLUSWn | → | Clears fileReg pointed by FSRn + WREG, and FSRnW are unchanged. |

**Ex 4** Write a program to clear 16 RAM locations starting at RAM address 60H.
Using :- a) INCF FSRnL b) Auto_increment

a)
```
COUNTREG  EQU  10H        → fileReg location for counter
CNTVAL    EQU  D'16'      → counter value
          MOVLW  CNTVAL   → w=16
          MOVWF  COUNTREG → Count=16
          LFSR   1, 60H   → FSR1 = 60H
B2        CLRF   INDF1    → clear RAM loc. FSR1 points to
          INCF   FSR1L, f ← increment FSR1L, point to next loc.
          DECF   COUNTREG, f → decrement counter
          BNZ    B2
```

b)
```
COUNTREG  EQU  10H
CNTVAL    EQU  D'16'
          MOVLW  CNTVAL
          MOVWF  COUNTREG
          LFSR   1, 60H     ← FSR0 = 60H
B2        CLRF   POSTINC1   → Clear RAM inc. FSR1 pointer
          DECF   COUNTREG, f → dec. counter
          BNZ    B2
```

Ex(5) Write a program to copy a block of 5 bytes of data from RAM locations starting at 30H to RAM locations starting a 60H

```
→ COUNTREG  EQU  0X10     → fileReg loc for counter
  CNTVAL    EQU  D'5'     → counter value
      MOVLW   CNTVAL      → 10 = WREG
      MOVWF   COUNTREG    → Count = 10
      LFSR    0 , 30H     → FSR0 = 30H , RAM address
      LFSR    1 , 60H     → FSR1 = 60H , " "
B3    MOVF    POSTINC0 , W → copy RAM to WREG and increment FSR0
      MOVWF   POSTINC1    → Copy WREG to RAM " " FSR1
      DECF    COUNTREG , f → decrement counter
      BNZ     B3          → loop until counter = zero
```

* before we run the above program, 30 = ('H')
31 = ('E')
32 = ('L')
33 = ('L')
34 = ('O')

* After the program is run, the addresses [60 – 64H] have the same data as [30 – 34H] →

30 = ('H')   31 = ('E')   32 = ('L')   33 = ('L')   34 = ('O')
60 = ('H')   61 = ('E')   62 = ('L')   63 = ('L')   64 = ('O')

---

Ex(6) Assume that RAM locations [40 – 43H] have the following hex data. Write a program to add them together and place the result in locations 06 and 07

40 = 7D
41 = EB
42 = C5
43 = 5B

| address | data |
|---|---|
| 040H | 7D |
| 041H | EB |
| 042H | C5 |
| 043H | 5B |

```
COUNTREG  EQU  20H     → counter location
L-Byte    EQU  0X06  ] locations
H-Byte    EQU  0X07  ]
CNTVAL    EQU  4       → counter value
  MOVLW   CNTVAL       → WREG = 4
  MOVWF   COUNTREG     → count = 4
  LFSR    0, 0X40      → FSR0 = 40H
  CLRF    WREG         → Clear WREG
  CLRF    H-Byte       → Clear H-Byte
```

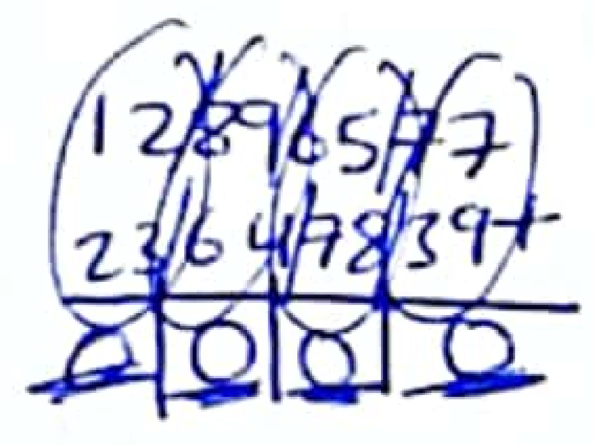
```
        B5      ADDWF   POSTINC0,W   -> add RAM to WREG
                                        and increment FSR0
        BNC     OVER         -> if C=0
                                go to next
        INCF    H_Byte,f     -> C=1, add 1 to H_Byte
OVER    DECF    COUNTREG,f   -> dec. counter
        BNZ     B5           -> loop until counter is zero
        MOVWF   L_Byte
```

---

**Ex(7)** How to use all three FSRn?

Write a program to add the following multi_byte BCD numbers and save the result at location 60H?

| address | data |
|---|---|
| 030H | 77 |
| 031H | 65 |
| 032H | 89 |
| 033H | 12 |
| 050H | 39 |
| 051H | 78 |
| 052H | 64 |
| 053H | 23 |



```
COUNTREG  EQU   20H
CNTVAL    EQU   D'4'
          MOVLW   CNTVAL
          MOVWF   COUNTREG
          LFSR    0,0x30   ]
          LFSR    1,0x50   ] loading pointer
          LFSR    2,0x60   ]
          BCF     STATUS,C       -> clear carry flag for the LSB
B3        MOVF    POSTINC0,W     -> copy RAM to WREG & INC. FSR0
          ADDWFC  POSTINC1,W     -> add RAM to W and increment FSR1
          DAW                    -> decimal adjust WREG
          MOVWF   POSTINC2       -> copy W to RAM and Inc. FSR2
          DECF    COUNTREG,f     -> dec. counter
          BNZ     B3
```

* before the addition we have:

33 = (12) 32 = (89) 31 = (65) 30 = (77)

53 = (23) 52 = (64) 51 = (78) 50 = (39)

* After the addition we have:

63 = (36) 62 = (54) 61 = (44) 60 = (16)

## 6.3 Lookup table and table processing

# We have seen that the PIC18 has a max. of 2M of code (ROM space ← Program) and 4KB of data RAM space.
While we never use any of the data RAM space for storing code, we can use the code space to store fixed data.

→ Beside instructions, ROM has enough space to stored fixed data
So in this section we discuss how to access fixed data residing in the program ROM space of the PIC18, by firstly examining how to store them in ROM using the DB (define byte) directive.

* DB directive → is widely used to allocate ROM program memory in byte-sized chunks. [used to define an 8bit fixed data]

- When DB is used to define fixed data, the no.'s can be in decimal, hex, binary, or ASCII formats.

- Use single quotes (') for a single character or (") for a string.

```
ORG     0X500
DATA1   DB    0X39
DATA2   DB    'Z'
DATA3   DB    "Hello All"
```

- ROM address must be even

# Reading table elements in the PIC18

- program Counter is 21 bit, which is used to point to any location in ROM space.

- How to fetch data from the code space?
  - Known as a table processing, register indirect ROM addressing mode.
  - there are table read and table write instructions.

- To read the fixed data byte :-
  - We need an address pointer: TBLPTR
    points to data to be fetched
    21 bits as the PC !!
    divided to 3 reg.'s 8 TBLPTRL / TBLPTRH / TBLPTRU (all parts of SFR's)
    8bit 8bit 8bit ↓ آخر 2bit (zeros) فيها مو مستخدمين

because we haven't an instruction that can load the 21 bit address into TBLPTR

- A reg. to store the read byte

TBLLAT : Keeps the data byte once it's fetched into the CPU.

Ex(8) Assume that we have burned the following fixed data into Program ROM of a PIC Chip. Give the contents of each ROM location Starting at 500H.

```
; MYDATA IN ROM
        ORG 500H              → it must be an even address
DATA1   DB  D'28'             → 1C in hexa
DATA2   DB  B'00110101'       → 35 in hexa
DATA3   DB  0X39

        ORG 510H              → even address
DATA4   DB  'Y'               → single ASCII character
DATA5   DB  '2','0','0','5'   → ASCII numbers

        ORG 518H              → even address
DATA6   DB "Hello ALL"        → ASCII string
        END
```

→ DATA1: 500 = (1C) DATA2: 501 = (35) DATA3: 502 = (39)

DATA4: 510 = (59) Y
DATA5: 511 = (32) 2, 512 = (30) 0, 513 = (30) 0, 514 = (35) 5

DATA6:
518 = (48) H, 519 = (65) e, 51A = (6C) l, 51B = (6C) l, 51C = (6F) o
51D = (20) space, 51E = (41) A, 51F = (4C) L, 520 = (49) I

∴ this example show how to place fixed data into ROM. The only problem is that the 2M of program Code (space) is under the direct control of the PC register. so, we need to have a SFR to point to the data to be fetched from the code space. For this reason we can call it register indirect ROM addressing mode. This is an addressing mode widely used to access data elements located in the program ROM space of the PIC18. This is often called (table processing)



Scanned with CamScanner

* Auto_increment option for TBLPTR →

- because it's a 21 bit_register, it can cover from [000000_1FFFFFH] which is the 2M ROM space of the PIC18.
- Using the "INCF TBLPTRL,f" instruction to increment the pointer can cause a problem when an address such as 5FFH is incremented.
- The carry will not propagate into TBLPTRH.

PIC18, Table Read Instructions

| | | |
|---|---|---|
| TBLRD* | Table Read | After read, TBLPTR stays the same |
| TBLRD*+ | Table Read with post-inc. | Reads and increments TBLPTR |
| TBLRD*- | = = = = -dec | = = decrements TBLPTR |
| TBLRD+* | = = = Pre-inc | Increments TBLPTR and then reads. |

**note** the byte of data is read into the TABLATch reg. from code space pointed to by TBLPTR

Ex(10). Assume that ROM space starting at 250H contains "USA", write a program to send all the characters to PORTB one byte at a time

→ a) using a counter →

```
RCOUNT  EQU  0X20   → counter loc
CNTVAL  EQU  0X3    → counter Value
        ORG  0H
        MOVLW 50H        → WREG = 50H, low byte address
        MOVWF TBLPTRL    → look up table low byte address
        MOVLW 0X02       → WREG = 2, high byte address
        MOVWF TBLPTRH    → look up table high byte address
        MOVLW CNTVAL     → WREG = 3, counter value
        MOVWF RCOUNT     → load counter
        CLRF  TRISB      → PortB, output port
B6      TBLRD*           → read table byte pointed to by TBLPTR
        MOVFF TABLAT, PORTB → send it to PORTB
        INCF  TBLPTRL, f → increment to point to next char.
        DECF  RCOUNT, f  → dec. the counter
        BNZ   B6         → repeat if counter ≠ 0
HERE    GOTO  HERE
```

(مهم) عالفاينل



Scanned with CamScanner

```
; data is burned into code (program) space starting at 250H
        ORG     0X250
MYDATA  DB      "USA"
        END
```

b) using null char. for end of string :-

```
        ORG     0H
        MOVLW   50H       → w = 50H , low byte address
        MOVWF   TBLPTRL   → look up table low-byte address
        MOVLW   2H        → w = 2 , high byte address
        MOVWF   TBLPTRH   → look up table high byte address
        CLRF    TRISB     → PORTB , output
B7      TBLRD*            → bring in next byte
        MOVF    TABLAT, W → copy to w (Z=1, if null)
        BZ      EXIT      → Is it null char.? exit if yes
        MOVWF   PORTB     → send it to PORTB
        INCF    TBLPTRL, F → increment pointing to next
        BRA     B7        → continue
EXIT    GOTO    EXIT
        ORG     0X250
MYDATA  DB      "USA", 0  → notice null
        END
```

---

c) * Using Auto_Increment ⇒

```
        ORG     0H
        MOVLW   50H
        MOVWF   TBLPTRL
        MOVLW   0X02
        MOVWF   TBLPTRH
        CLRF    TRISB
B6      TBLRD*+           → bring in next byte and inc. TBLPTR
        MOVF    TABLAT, W
        BZ      EXIT
        MOVWF   PORTB
        BRA     B6
EXIT    GOTO    EXIT
        ORG     250H
MYDATA  DB      "USA", 0  → null
        END
```



Scanned with CamScanner

# Lookup table and RETLW Instruction

- The lookup table is a widely used concept in mc programming. It allows access to elements of a frequently used table with minimum operations.
- As an example, assume that for a certain application we need $x^2$ values in the range of 0 to 9. We can use a look up table instead of calculating the values, which takes some time.
  - * In the PIC, to get the table element we first call the look-up table, then we add a fixed value to the PCL (low byte portion of the PC) to index into the look-up table.
  - * Upon return from the table, The RETLW (→ Return literal to W) instruction will provide the desired look-up table element in the WREG register.

**Ex(14)** Write a program to get the x value from PORTB and send $x^2$ to PORTC. Assume that RB0-RB3 has the x value of 0-9. Use a lookup table instead of a multiply instruction.
What's the value of PORTC if we have 9 at PORTB?

```
      ORG 0
      SETF TRISB   → PORTB, input
      CLRF TRISC   → PORTC, output
(B1)  MOVF PORTB,W → read x from PORTB into WREG
      ANDLW 0FH    → mask upper bits
      CALL XSGR_TABLE → get x² from the look up table
      MOVWF PORTC  → copy it to PORTC
      BRA (B1)     → continue

; lookup table for square of numbers 0-9 XSQR_TABLE
      MULLW 0X2    → align it for even address
      MOVFF PRODL,WREG → put it into WREG for indexing
      ADDWF PCL    → PCL = PCL+WREG
      RETLW D'0'   → square of 0
      RETLW D'1'   →   "     "  1
        "   D'4'   →   "     "  2
        "   D'9'   →   "     "  3
        "   D'16'  →   "     "  4 (10 hex)
        "   D'25'  →   "     "  5 (19 hex)
        "   D'36'  →   "     "  6 (24 hex)
        "   D'49'  →   "     "  7 (31 hex)
        "   D'64'  →   "     "  8 (40 hex)
        "   D'81'  →   "     "  9 (51 hex)
      END
```



Scanned with CamScanner

## Accessing a look-up table in RAM

* The lookup table elements can also be in RAM instead of ROM.
* Sometimes we need to bring in the elements of the lookup table from RAM because the elements are dynamic and can change.
* The PIC18 allows us to do that using the FSR as pointer.
* For example, the instruction "MOVFF PLUSW2, PORTD" will bring elements of the lookup table pointed to by the address location formed by the addition of FSR2 + WREG.

  In this case, WREG is used as an index into the lookup table.

**Ex(15)** Repeat ex(13) assuming that the lookup table elements are in data RAM locations starting at address 20H as shown below:-

```
20 = ('0')
21 = ('1')
22 = ('2')
23 = ('3')
24 = ('4')
25 = ('5')
26 = ('6')
27 = ('7')
```

```
      ORG    OH
      SETF   TRISC
      CLRF   TRISD
      LFSR   2,20H        -> FSR2 = 20H
B1    MOVF   PORTC,W      -> read x from PORTC into WREG
      ANDLW  B'00000111'  -> mask upper 5 bits
      MOVFF  PLUSW2,PORTD
      BRA    B1           -> get data pointed to by FSR2 + WREG
      END
```

**Ex(16)** Write program to get the x value from PORTB and send $x^2+2x+3$ to PORTC. Assume PB0 – PB3 has the value x of 0–9. Use a lookup table instead of a multiply instruction.

```
      ORG    OH
      SETF   TRISB
      CLRF   TRISC
B1    MOVF   PORTB,W       -> read x from PORTB into WREG
      ANDLW  OFH           -> mask upper bits
      CALL   XSQR_TABLE    -> get x² from the lookup table
      MOVWF  PORTC         -> copy it to PORTC
      BRA    B1

XSQR-TABLE
      MULLW  2H            -> align it for even address
      MOVFF  PRODL,WREG    -> put it into WREG for indexing
      ADDWF  PCL           -> PCL = PCL + WREG
      RETLW  D'3'          -> (0)² + 2(0) + 3 = 3
        "    D'6'          -> (1)² + 2(1) + 3 = 6
        "    D'11'         -> (2)² + 2(2) + 3 = 11
        ⋮
```



Scanned with CamScanner

## 6.4 Bit addressability of Data RAM

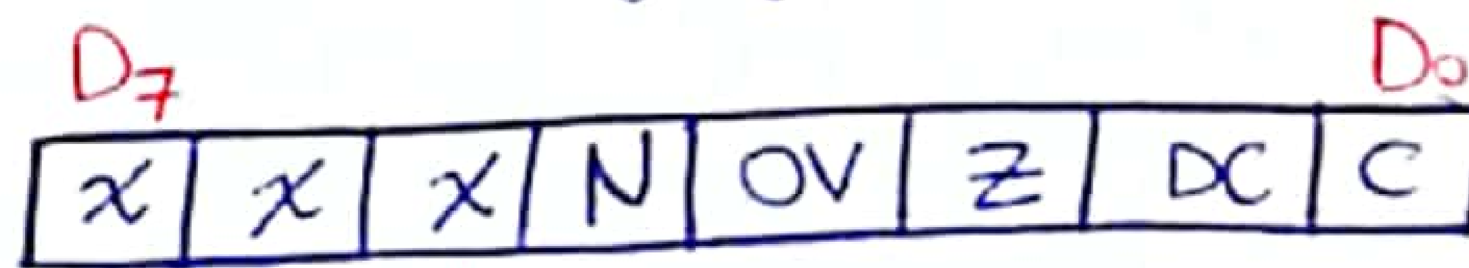
One of the most important features of the PIC is its ability to access the file register's RAM location in bits as well as bytes.

This means that all I/O ports, SFRs, and GPR-RAM areas for the PIC18 are bit addressable because they're part of the fileReg Data RAM.

WREG is also bit-addressable because it's part of the SFRs.

* Status register bit_addressability →

You can access any bit of the status reg. by their name.




ex:

```
BCF    STATUS,C
BTFSS  STATUS,Z
```

unused [for future]

BCF → bit clear file
BTFSS → bit test file skip if set

## 6.5 Bank Switching in the PIC18 →

* PIC18 has max. of 4KB of RAM, that not all of them in use.
* every PIC18 has the access bank of the fileReg (the first 128B of RAM+SFR)

where the fileReg is divided into 16 banks of [256 byte] each.

* The minimum bank that every PIC18 has is called the access bank, which is made of → 128 byte of Lower addresses [000-07FH] for GPR
  → 128 " " higher " [F80-FFFH] dedicated to SFR

* In this section we show how to use bank switching to take advantage of the entire data RAM space of the PIC18

→ Most PIC18 that access the data space in RAM has the ability to access any bank through setting an optional operand, called A

[MOVWF myReg, A] → if A=0, the access bank is the default bank
→ if A=1, " instruc. will use (BSR) to select the bank instead of using the access bank.



Scanned with CamScanner

* The BSR register and bank switching ⇒

- With A=1, we use the BSR to choose the desired bank.
- It's an 8 bit register and is part of the SFRs.

  the 4 LSB of it are in use in the PIC18, the upper 4bit (set to zero) ignored by the PIC18

  ↓ gives us 16 banks, each one has (256 B)

  so we cover the entire 4096 bytes of RAM fileReg using bank switching

- Bank 0 [00-FFH]
- Bank 1 [100-1FFH]
- Bank 2 [200-2FFH]

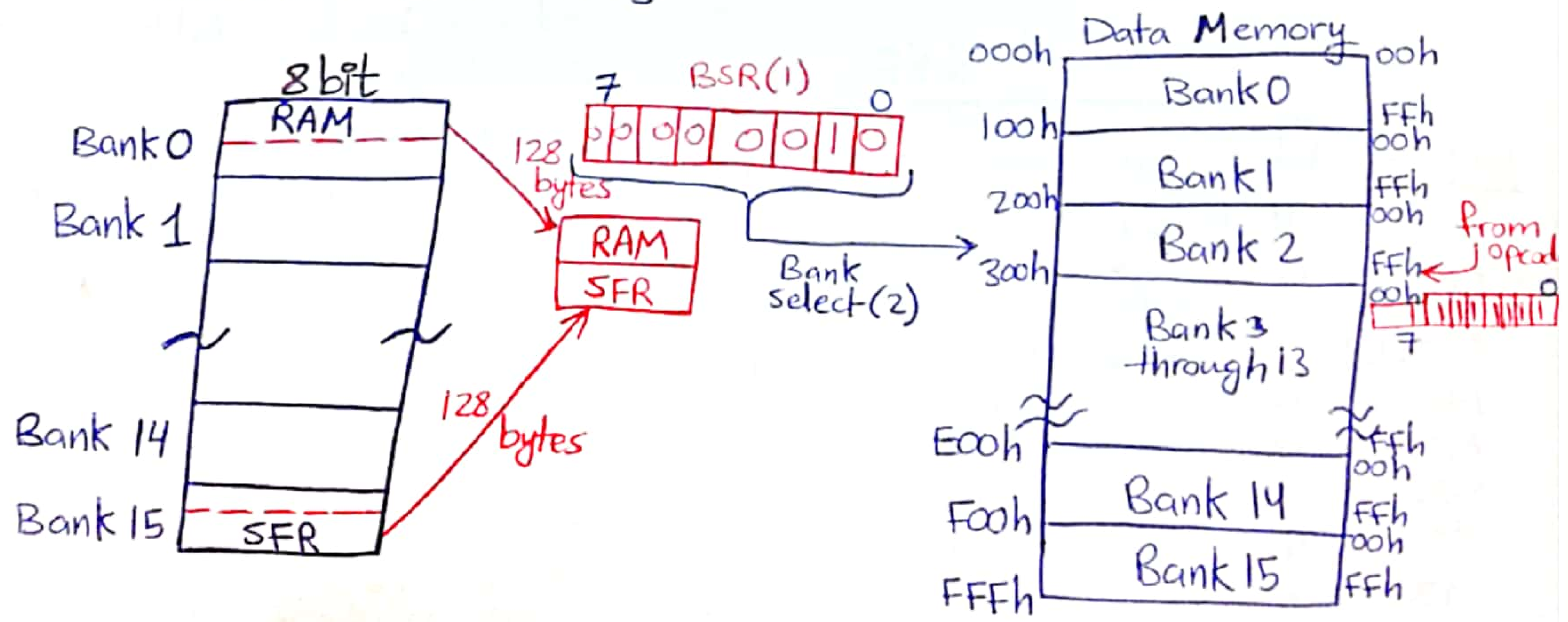
  ⋮

- Bank F [F00-FFFH] (includes SFR)

- Upon power-on reset, BSR=0 that indicates that only the lowest addresses of data RAM, from 000 to 0FFH, can be used for [GPR + SFR]'s

→ if we make BSR=1, then PIC18 selects bank 1 using [100-1FFH] addresses in addition to the SFRs, which use only the last half of the bank with addresses of [F80-FFFH]

→ To select Bank 2, we load BSR with value (0010 binary) (02 value), which allows access to the bank addresses [200-2FFH] in addition to the SFR addresses of [F80-FFFH].

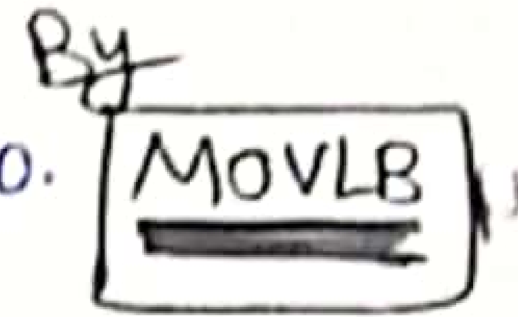



we use only direct addressing mode in accessing the SFR registers.

## * Bank switching and "INCF F, D, A" instruction →

- To use banks other than the access bank, two things must be done :-

① Load the BSR with the desired bank no. By

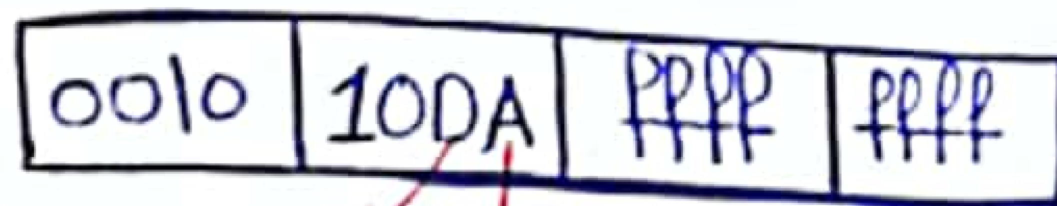



② make A=1 in the instruction itself

- The following code, we first load the bank no. into the BSR using the MOVLB instruction, and then manipulate the contents of RAM location 0X240 (location 40 of bank 2)

```
MYREG    EQU   0X40

MOVLB  0X2          ; load 2 into BSR (use Bank 2)
MOVLW  0            ; WREG = 0
MOVWF  MYREG, 1     ; loc 0X240 = (0) , W = 0 , notice A=1
INCF   MYREG, f, 1  ; loc 0X240 = (1) , W = 0 ,   "    "
INCF   MYREG, f, 1  ; loc 0X240 = (2) , W = 0 ,   "    "
INCF   MYREG, f, 1  ; loc 0X240 = (3) , W = 0
```




destination bit for operation (D) ; bank accessed for operation (A)

D=F , destination is fileReg
D=W , " " WREG

A=0 , use default access bank
=1 , " bank pointed to by BSR

$0 \leqslant f \leqslant FF$

---

**Ex(25)** Write a program to copy the value 55H into RAM memory locations 340H to 345H using:- (RAM locations, bank 3)

a) Direct addressing mode

```
MOVLB  3H       → Bank 3
MOVLW  55H      → W = 55H
MOVWF  40H, 1   → copy WREG contents to RAM loc 340H
MOVWF  41H, 1
MOVWF  42H, 1      وهكذا
MOVWF  43H, 1
MOVWF  44H, 1
MOVWF  45H, 1
```

340 = (55)
341 = (55)
342 = (55)
343 = (55)
344 = (55)

b) a Loop

```
COUNT  EQU    0x10       → location 10h
       MOVLB  3H         → Bank 3
       MOVLW  5H         → W = 5H
       MOVWF  COUNT      → count = 5H
       LFSR   0, 340H    → load pointer, FSR0 = 40H
       MOVLW  0X55       ← Value to be copied
B1     MOVWF  INDF0, 0   → copy W to RAM loc. FSR0 points to
       INCF   FSR0L      ← increment FSR0L pointer
       DECF   COUNT, f, 0  → decrement the counter
       BNZ    B1         → loop until counter = 0
```

## 6.6 Checksum and ASCII subroutines →

In this section we look at some widely used subroutines such as the Checksum byte, BCD, and ASCII conversion. We'll also examine the use of a stack in the PIC18.

### Checksum byte in ROM

* to ensure the integrity of ROM contents, every system must perform a checksum calculation, the checksum will detect any corruption of the contents of ROM (caused by current surge) either when the system is turned on, or during operation.

* The checksum byte is an extra byte that is tagged to the end of a series of bytes of data. To calculate it:
  1. Add the bytes together and drop the carries.
  2. Take the 2's complement of the total sum. " it becomes the last byte of the series "

* To perform a checksum operation, add all the bytes, including the checksum byte. the result must be 0, else error.

### Checksum program ⇒

The check sum program generation and testing is given in subroutine form. we have divided the following program into 3 subroutines (subprograms) that perform the following operations :-

① retrieve the data from code ROM ② Calc. the checksum byte ③ Test the checksum byte for any data error

```
RAM_ADDR  EQU  40H    ; RAM space to place the bytes
COUNTREG   "   0X20   ; FileReg loc. for counter
CNTVAL     "   4      ; Counter value = 4 for adding 4 bytes
CNTVAL1    "   5      ;  "   "  = 5  "   "  5  "
                      ; including checksum byte.
;____________________main program
ORG OH
CALL COPY_DATA
CALL CAL-CHECKSUM
CALL TEST_CHCKSUM
BRA $
;____________________Copying data from code ROM address 500H to data RAM loc.
COPY_DATA
  MOVLW LOW(MYBYTE)    → WREG=0 , low byte address
  MOVWF TBLPTRL    → ROM data  "  "  "
  MOVLW high(MYBYTE)   → WREG = 5,  high  "  "
  MOVWF TBLPTRH    → ROM data  "  "  "
  MOVLW upper(MYBYTE)  →   WREG=0 , upper byte address
```



Scanned with CamScanner

```
        MOVWF   TBLPRTRU  -> ROM data upper byte address
        LFSR    0, RAM_ADDR  -> FSR0= RAM_ADDR, place to save
C1      TBLRD*+   -> bring in next byte and inc TBLPTR
        MOVF    TABLAT, W  -> copy to WREG (Z=1, if null)
        BZ      EXIT  -> Is it null char.? exit if yes
        MOVWF   POSTINC0  -> copy W to RAM and inc. pointer
        BRA     C1
EXIT    RETURN

;------------------------------calculating checksum byte
CAL-CHKSUM
        MOVLW   CNTVAL -> WREG=4
        MOVWF   COUNTREG -> Count=4
        LFSR    0, RAM-ADDR -> FSR0=40H
        CLRF    WREG
C2      ADDWF   POSTINC0, W -> add RAM to WREG and inc. FSR0
        DECF    COUNTREG, f  -> dec. counter
        BNZ     C2  -> loop until counter zero
        XORLW   0FFH  -> 1's comp.
        ADDLW   1  -> 2's comp.
        MOVWF   POSTINC0
        RETURN
;------------------------------testing Checksum byte
TEST_CHKSUM
        MOVLW   CNTVAL1 -> WREG=5
        MOVWF   COUNTREG -> count=5
        CLRF    TRISB  ->  PORTB= output
        LFSR    0, RAM-ADDR -> FSR0 = 40H
        CLRF    WREG
C3      ADDWF   POSTINC0, W -> add RAM and inc. FSR0
        DECF    COUNTREG, F  -> dec. counter
        BNZ     C3   -> loop until counter=0
        XORLW   0H   -> EX-OR to see if W=0
        BZ      G_1  -> is result zero? then go to G_1 (good)
        MOVLW   'B'
        MOVWF   PORTB  -> if not, data is bad
        RETURN
G_1     MOVLW   'G'
        MOVWF   PORTB  -> data isn't corrupted.
        RETURN
;------------------------------my data in program ROM
        ORG  0X500
MYBYTE  DB   0X25, 0X62, 0X3F, 0X52, 0X00
        END
```



Scanned with CamScanner

Ex(29) Assume that we have 4 bytes of hexadecimal data: 25H, 62H, 3FH, and 52H?

a) Find the checksum byte?

```
  25 H
+ 62 H
+ 3F H
+ 52 H
------
 118 H
```

مهم جدا سؤال

(Drop the carry bit 1) we have **18H**

It's 2's comp. is E8H

∴ the checksum byte is E8H

b) perform the checksum operation to ensure data integrity?

```
  25 H
+ 62 H
+ 3F H
+ 52 H
+ E8 H
------
 200 H
```

(Dropping the carries, we see 00, so data isn't corrupted)

c) If the second byte, 62H, changed to 22H, show how the checksum method detects the error?

```
  25 H
+ 22 H
+ 3F H
+ 52 H
+ E8 H
------
 1C0 H
```

(Dropping the carry, we got C0H, not 00, so the data is corrupted here)

Scanned with CamScanner

## 6.7 Macros and models

* Macros allow the programmer to write the task (code to perform a specific Job) once only, and to invoke it whenever it's needed.

to reduce → time
→ errors

* MACRO definition →

```
name   MACRO   dummy1, dummy2, ..., dummyN
[ .....
  ..... ]
       ENDM
```

- name → must be unique
- dummy1, dummy2, ..., dummyN → names, registers, parameters that are mentioned in the body of the MACRO
- [ ..... ] the body of the MACRO

```
MOVLF   MACRO  K, MYREG
        MOVLW  K
        MOVWF  MYREG
        ENDM
```

(K, MYREG → dummy)

**Ex** Write a delay MACRO and a MOVLF macro?

```
;---------------------------- time delay MACRO
DELAY_1   MACRO  V1
          TREG
          LOCAL  BACK
          MOVLW  V1
          MOVWF  TREG
BACK      NOP
          NOP
          NOP
          NOP
          DECF   TREG, F
          BNZ    BACK
          ENDM
;---------------------------- sending data to fileReg MACRO
MOVLF     MACRO  K, MYREG
          MOVLW  K
          MOVWF  MYREG
          ENDM
;---------------------------- program starts
          ORG    0
          CLRF   TRISB
OVER      MOVLF  0X55, PORTB
DELAY_1   0X200, 0X10
MOVLF     0XAA, PORTB     → send value AAH to PORTB location
DELAY_1   0X200, 0X10
BRA       OVER
END
;---------------------------- End of file
```

LOCAL → عبارة عن directive بيجي قبل ال (body) تاع ال MACRO وبعد ال MACRO Directive

BACK → نستخدم ال local directive to allows the assembler to define the labels separately each time it encounters them.

# Ch.9 PIC18 timer programming in Assembly and C

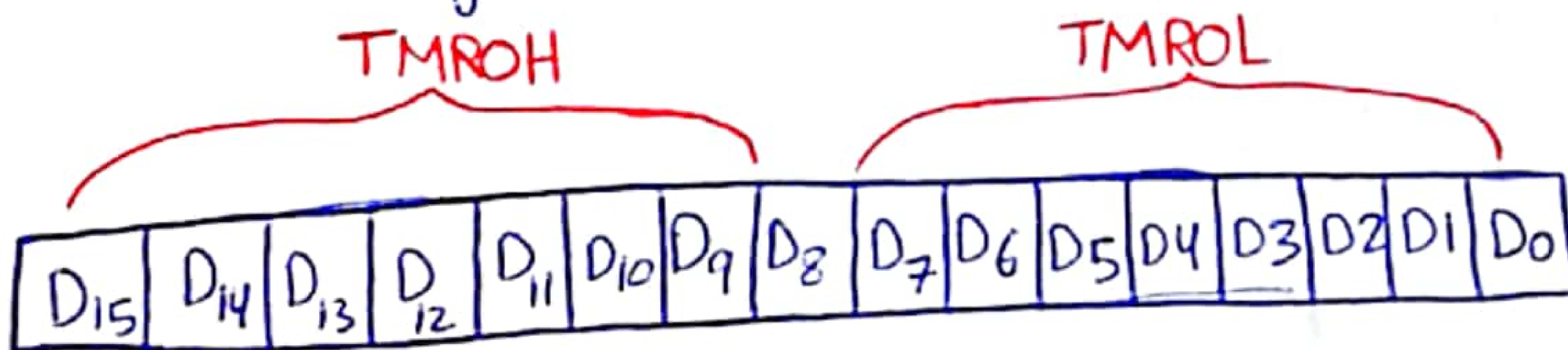
* PIC18 has 2-5 timers depending on the family number, referred to as timers 0, 1, 2, 3, and 4.
* These timers can be used as → timers to generate a time delay
  → counters to count events happening outside the microcontroller.

## 9.1 Programming timers 0 and 1 →

- Every timer needs a clock pulse to tick.
- The clock source can be → internal [1/4th of the frequency of the crystal oscillator (crystal frequency) on the OSC1 and OSC2 pins (FOSC/4) is fed into timer]
  → external [we feed pulses through one of the PIC18 pins that called a counter]
- Timers are 16 bit wide, can be accessed as two separate register of low byte (TMRxL) and high byte (TMRxH).
  Each timer has timer control (TCON) register, for setting modes of operation

# Timer0 registers and programming →

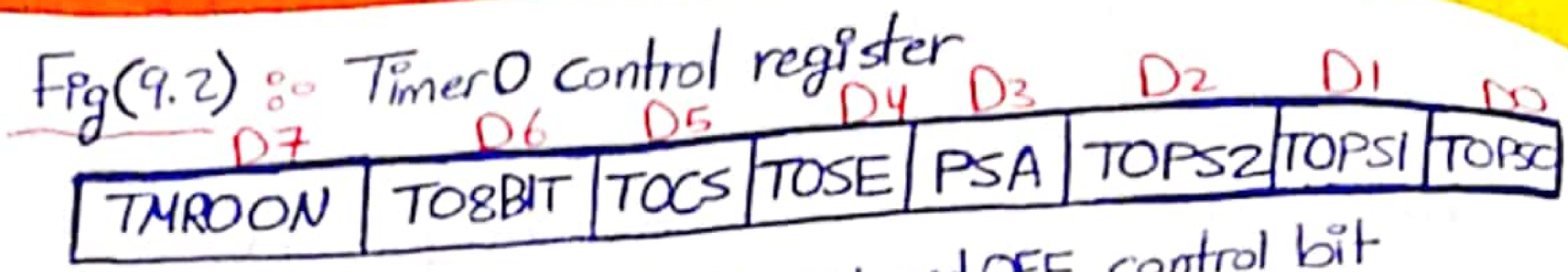
* Timer0 can be used as an 8 bit or a 16bit timer.
* The 16 bit register of it is accessed as Low byte and High byte as shown




* It can be accessed like any other SFR, and read like any other registers.

# Timer0 control register (TOCON), it's an 8 bit register used for control of timer0

Scanned with CamScanner




| | | |
|---|---|---|
| TMR0ON | D7 | Timer0 ON and OFF control bit<br>1 = Enable (start) Timer0<br>0 = stop Timer0 |
| T08BIT | D6 | Timer0 8bit/16 bit selector bit<br>8bit ← 1 = Timer0 configured as an 8 bit timer/counter<br>16bit ← 0 = 〃 〃 〃 16 bit 〃 〃 |
| T0CS | D5 | Timer0 clocksource select bit<br>counter ← 1 = external clock from RA4/T0CK1 pin<br>timer ← 0 = internal 〃 (FOSC/4 from XTAL OSC) |
| T0SE | D4 | Timer0 source edge select bit<br>-ve edge ← 1 = Increment on H-to-L transition on T0CK1 pin<br>+ve edge ← 0 = 〃 〃 L-to-H 〃 〃 T0CK1 〃 |
| PSA | D3 | Timer0 prescaler assignment bit<br>1 = Timer0 Clock input bypasses prescaler<br>0 = Timer0 〃 〃 comes from prescaler output |
| T0PS2 : T0SPS0 | D2 \| D0 | Timer0 prescaler selector |

$2^n$

| D2 D1 D0 | Ratio | |
|---|---|---|
| 0 0 0 | = 1:2 | → prescale value (FOSC/4/2) |
| 0 0 1 | = 1:4 | → 〃 = (FOSC/4/4) |
| 0 1 0 | = 1:8 | → 〃 = (FOSC/4/8) |
| 0 1 1 | = 1:16 | → 〃 = (FOSC/4/16) |
| 1 0 0 | = 1:32 | → 〃 = (FOSC/4/32) |
| 1 0 1 | = 1:64 | → 〃 = (FOSC/4/64) |
| 1 1 0 | = 1:128 | → 〃 = (FOSC/4/128) |
| 1 1 1 | = 1:256 | → 〃 = (FOSC/4/256) |

---

Prescale value = fosc/4/[ ]

D2/D1/D0 من $2^n$ حسب قيمة

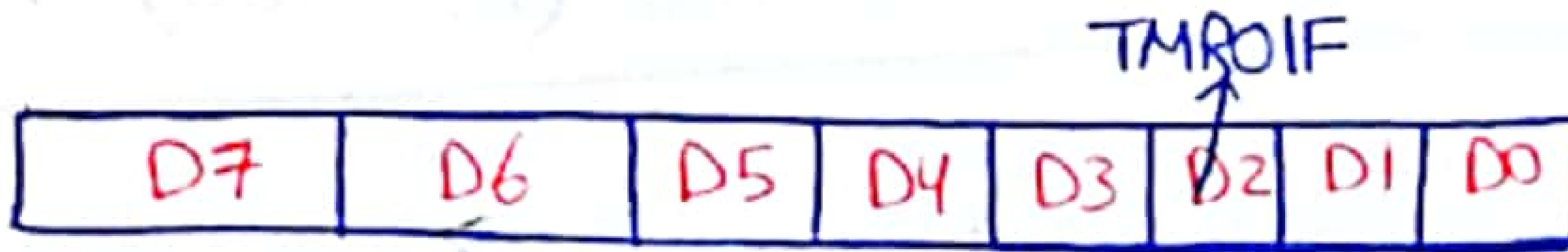
Scanned with CamScanner

Ex If TOCON = 00001000

Timer0 is off
16 bit register
Fosc/4 clock source
no prescaler
Rising edge (L to H transition) —— According to D4

# TMR0IF flag bit → [Timer0 interrupt flag]

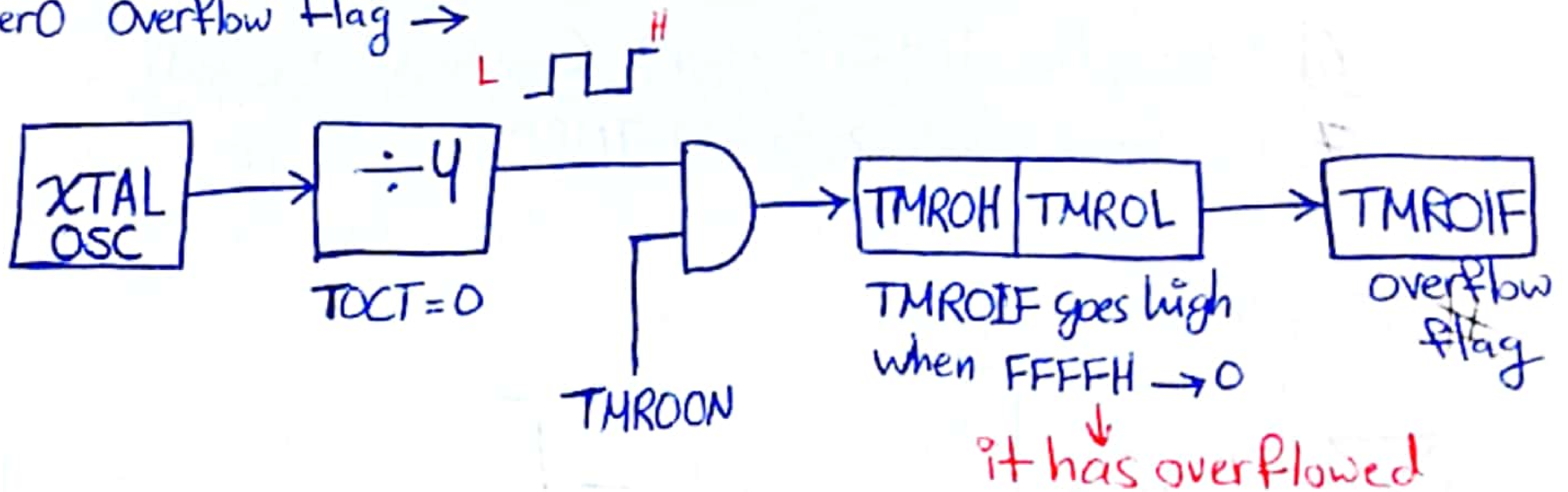
It's part of the INTCON (interrupt control) register




D2 → TMR0IF Timer0 interrupt overflow flag bit
0 = Timer0 didn't overflow
1 = Timer0 has overflowed (FFFF to 0000, or FF to 00 in 8 bit mode)

- the importance of TMR0IF →
  In 16 bit mode, when [TMR0H : TMR0L] overflows from FFFF to 0000 this flag is raised.
  In 8 bit, it's raised when the timer goes from FF to 00.
  We monitor this flag bit before we reload the [TMR0H : TMR0L] registers.

- Timer0 Overflow Flag →



* 16 bit timer programming →

The following are the characteristics and operations of 16 bit mode:-

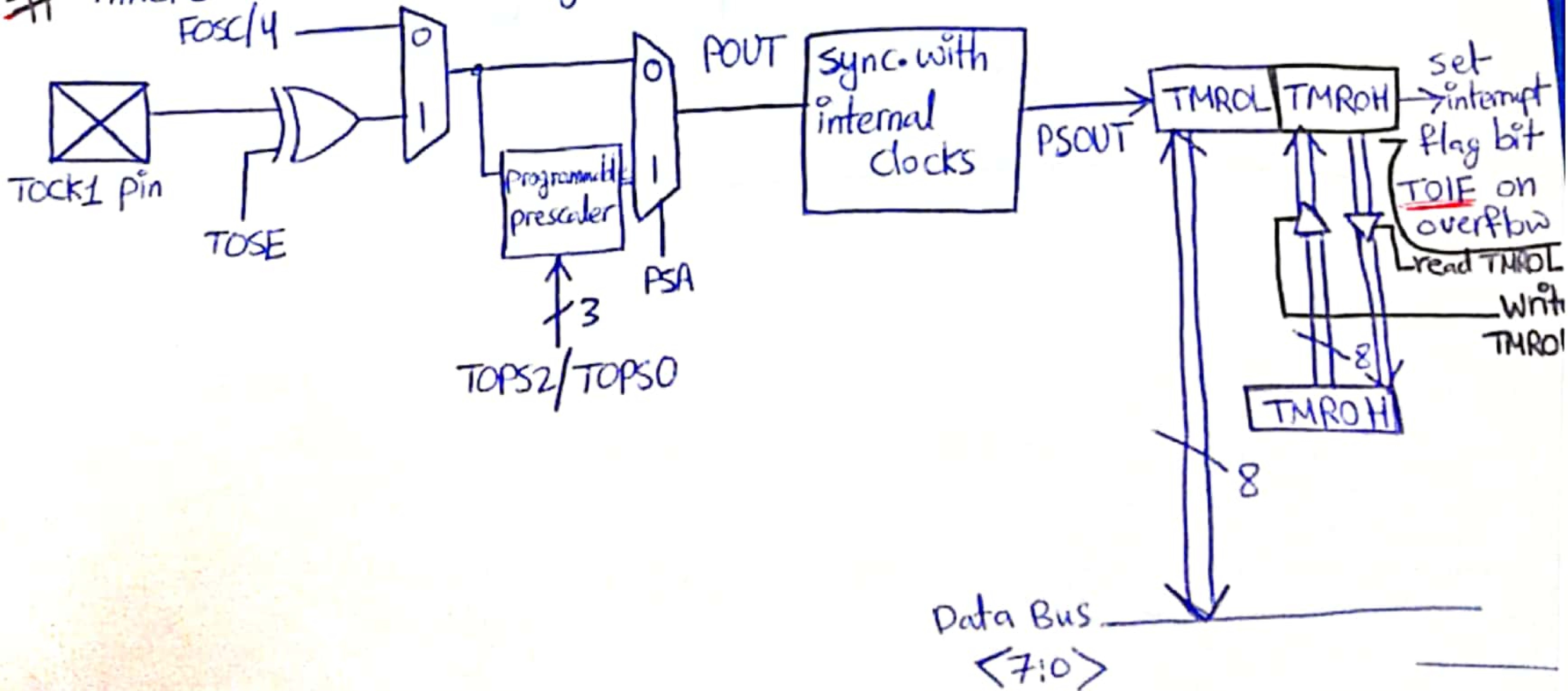
1) 16 bit timer, 0000 to FFFFH
2) After loading TMROL & TMROH, the timer must be started.
3) the timer starts to count up, till it reaches FFFFH. Then it rolls over to 0000 and activate TMROIF bit → sets to HIGH
4) To repeat the process, the registers TMROH and TMROL must be reloaded with the original value and deactivite (reset to 0) TMROIF bit.

---

* Steps to program Timer0 in 16 bit mode → to generate a time delay using the timer0 mode 16 :-

1) Load the value into the TOCON register, indicating which mode (8 or 16 bit) is to be used and the selected prescaler option.
2) Load register TMROH followed by reg. TMROL with initial count
3) Start the timer with instruction "BSF TOCON, TMROON"
4) Keep monitoring the timer flag (TMROIF) to see if it's raised. Get out of the loop when TMROIF becomes HIGH
5) Stop the timer with instruction "BCF TOCON, TMROON"
6) Clear the TMROIF flag 3 (for the next round)
7) Go back to step 2 to load TMROH & TMROL again.

# Timer0 16-bit block diagram →






Scanned with CamScanner

Ex(9-3) In the following program, we are creating a square wave of 50% duty cycle (with equal portions high and low) on the PORTB.5bit. Timer0 is used to generate the time delay?

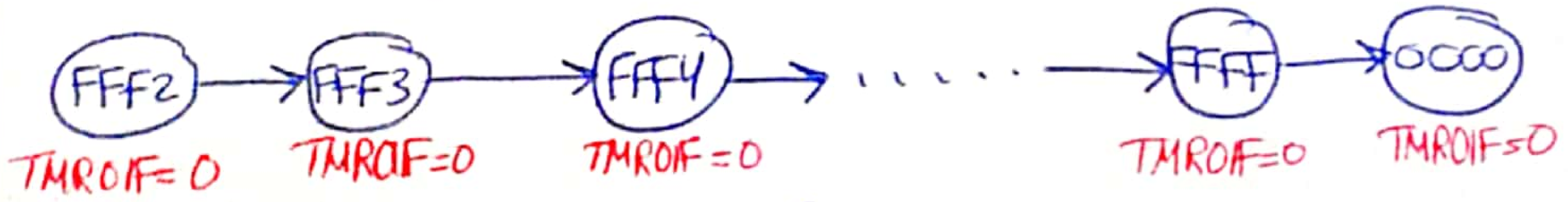
call delay وبعدها لا يجب ان في التايمر

```
      BCF     TRISB, 5       → PORTB, pin5, output PB5
      MOVLW   0X08           → Timer0, 16bit, int CLK, no prescale
      MOVWF   T0CON          → load T0CON reg.
HERE  MOVLW   0FFH           → TMR0H = FFH, the high byte
      MOVWF   TMR0H          → load Timer0 high byte
      MOVLW   0F2H           → TMR0L = F2H, the low byte
      MOVWF   TMR0L          → load Timer0 low byte
      BCF     INTCON, TMR0IF → clear timer interrupt flag bit
      BTG     PORTB, 5       → toggle PB5
      BSF     T0CON, TMR0ON  → start Timer0
AGAIN BTFSS   INTCON, TMR0IF → monitor Timer0 flag until
      BRA     AGAIN          → it rolls over
      BCF     T0CON, TMR0ON  → stop Timer0
      BRA     HERE           → load TH, TL again
```

stop timer → BCF
start timer → BSF

اذا ما TMR0IF = 1 ما بيطلع منها لحد ما يوصل ل FFFFH التايمر

1. T0CON is loaded
2. FFF2H is loaded into TMR0H-TMR0L
3. the timer0 interrupt flag is cleared by the "BCF INTCON, TMR0IF" instruction.
4. PORTB.5 is toggled for the high and low portions of the pulse
5. Timer0 is started by the "BSF T0CON, TMR0ON" instruction
6. " counts up with the passing of each clock, which is provided by the crystal oscillator. As the timer counts up, it goes through the states of FFF3, FFF4, FFF5, FFF6, FFF7, FFF8, FFF9, FFFA, and so on until it reaches FFFFH.
   One more clock rolls it to 0, raising the timer0 flag (TMR0IF = 1). At that point, the "BTFSS INTCON, TMR0IF" instruction bypasses the "BRA AGAIN" instruction.
7. Timer0 is stopped by "BCF T0CON, TMR0ON", and the process is repeated.






Scanned with CamScanner

Ex(9.5) Calculate the frequency of the square wave generated on pin PORTB.5 ?

```
        BCF     TRISB,5
        MOVLW   08H         ] Load T0CON
        MOVWF   T0CON
        BCF     INTCON,TMR0IF  -> clear interrupt flag bit
HERE    MOVLW   0FFH         |
        MOVWF   TMR0H        |
        MOVLW   -D'48'       |
        MOVWF   TMR0L        |
        CALL    DELAY        |
        BTG     PORTB,5      |
        BRA     HERE         |
;------------------------------ delay using Timer0
DELAY   BSF     T0CON,TMR0ON    | -> start Timer0
AGAIN   BTFSS   INTCON,TMR0IF   | -> monitoring TMR0IF
        BRA     AGAIN           |
        BCF     T0CON,TMR0ON    | -> Stop Timer0
        BCF     INTCON,TMR0IF   | -> reset TMR0IF
        RETURN
                                |
                               ---
                               13
```

$T = 2 * (48 + 13) * 0.4\ \mu s$

$= 48.8\ \mu s$

$F = 20.491\ KHZ$

---

# We can develop a formula for delay calculations using 16 bit mode of the timer for a crystal frequency of XTAL = 10MHZ

Fig. ⇒ Timer Delay Calculations for XTAL = 10MHZ with no prescaler:

$T = 4/10MHZ = 0.4\ \mu second$

a) In Hex

$(FFFF - YYXX + 1) * 0.4\ \mu s$

YYXX → TMR0H (hex), TMR0L initial values

b) In decimal

Convert YYXX values of the TMR0H, TMR0L register to decimal to get a NNNNN decimal no., then $(65536 - NNNNN) * 0.4\ \mu s$



Scanned with CamScanner

# Finding values to be loaded into the timer →

- We use in the following example a crystal frequency of 10MHZ for the PIC18 system so

* Assuming XTAL = 10 MHZ and no prescaler, we use the following steps for finding the TMR0H & TMR0L register's value :-

1. divide the desired time delay by 0.4 μs
2. Perform [65,536 − n], where n is the decimal value we got in step ①
3. Convert the result of step ② to hex, where yyxx is the initial hex value to be loaded into the timer's registers.
4. Set TMR0L = xx and TMR0H = yy

**Ex(9.8)** Assuming that XTAL = 10MHZ, write a program to generate a square wave with a period of 10 ms on pin PORTB,3 ?

→ for a square wave with T = 10ms, time delay = 5ms
the counter counts up every 0.4 μs. So we need 5ms / 0.4μs = 12,500 clocks
(65,536 − 12,500 = 53036 = CF2CH), So we have TMR0H = CF H
TMR0L = 2CH

```
      BCF    TRISB,3         → PB3 as an output
      MOVLW  08H             → Timer0, 16bit, int CLK, no prescale
      MOVWF  T0CON           → Load T0CON reg.
HERE  MOVLW  0CFH            → TMR0H = 0CFH, the high byte
      MOVWF  TMR0H           → Load Timer0 high byte
      MOVLW  2CH             → TMR0L = 2CH, the low byte
      MOVWF  TMR0L           → Load Timer0 low byte
      BCF    INTCON,TMR0IF   → Clear timer interrupt flag bit
      CALL   DELAY
      BTG    PORTB,3         → toggle PB3
      BRA    HERE            → Load TH, TL again
;_________________ delay using Timer0
DELAY BSF    T0CON,TMR0ON    → start Timer0
AGAIN BTFSS  INTCON,TMR0IF   → monitor Timer0 flag until
      BRA    AGAIN           → it rolls over
      BCF    T0CON,TMR0ON    → stop Timer0
      RETURN                 → BCF INTCON,TMR0IF
      END
```

مشغل التايمر ← BSF T0CON, TMR0ON
مطفيه ← BCF T0CON, TMR0ON

ازاي وصل ال TMR0IF (1) ال



Scanned with CamScanner

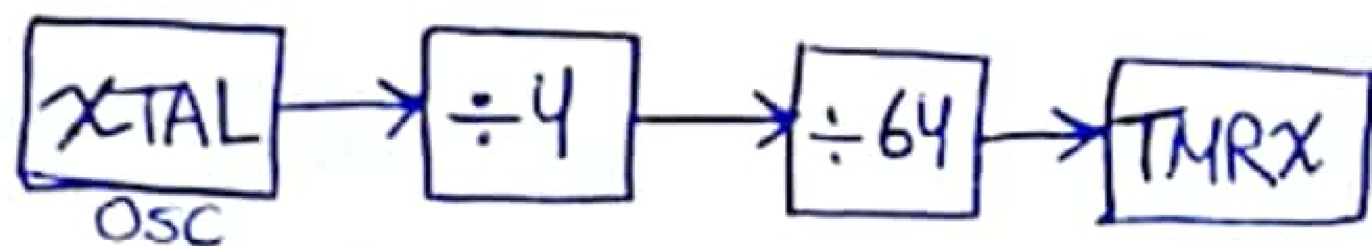
# Prescaler and generating a large time delay →

* the size of the time delay depends on 2 factors → the crystal freq.
  → the timer's 16 bit register

* The largest timer happens when TMROL = TMROH = 0

→ We use the prescaler option in the TOCON register to increase the delay by reducing the period, it allows us to divide the instruction clock by a factor of 2 (lowest no.) to 256 ← highest no.

If TOCON = 0000 0101, then $T = 4 * 64/f$




Ex (9.13) Examine the following program and find the time delay in seconds. Exclude the overhead due to the instructions in the loop. Assume XTAL = 10 MHZ

```
     BCF    TRISB,2          → PB2 as an output
     MOVLW  05H              → Timer 0, 16 bit, int CLK, prescaler 64
     MOVWF  TOCON            → load TOCON reg.
HERE MOVLW  01H              → TMROH = 1H, the high byte
     MOVWF  TMROH            → Load TMRO high byte
     MOVLW  8H               → TMROL = 8H, the low byte
     MOVWF  TMROL            → load Timer0 low byte
     BCF    INTCON,TMROIF    → clear timer interrupt flag bit
     CALL   DELAY
     BTG    PORTB,2          → toggle PB2
     BRA    HERE             → load TH, TL again
; ---------- delay using timer0
DELAY BSF   TOCON,TMROON     → start timer0
AGAIN BTFSS INTCON,TMROIF    → monitor Timer0 flag until
      BRA   AGAIN            → it rolls over
      BCF   TOCON,TMROON     → stop Timer0
      RETURN
```

TMROH : TMROL = 0108H = 264 in decimal

and 65,536 − 264 = 65,272

Now, 65,272 * 64 * 0.4 μs = 1.671 seconds (prescaler)

or 65,272 * 25.6 μs = 1.671 seconds → desired time delay

* مثال 8 ← أعطانا الـ [desired time delay]

والـ [XTAL Frequency]

ومطلوب منا نحسب القيمة اللي بدنا ندخلها على الـ TMR0H ؟

$$65{,}536 - \frac{\text{desired time delay}}{\left(\frac{1}{f/4}\right)} = \boxed{\phantom{0}}$$

منحوله للـ hexadecimal

---

* مثال 13 ← أعطانا القيمة المدخلة للـ (TMR0H / TMR0L) ومطلوب منا نحسب الـ time delay اذا عرفنا الـ XTAL freq ؟ والـ T0CON Value

القيمة المدخلة للتايمر بالـ (decimal)

$$65{,}536 - \boxed{1} = \boxed{2}$$

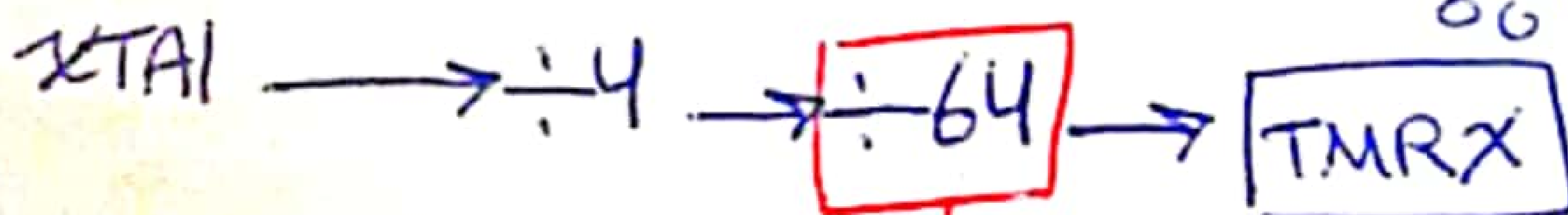
$$\text{time delay} = \boxed{2} * \frac{1}{f/4} * \text{prescaler}$$

prescaler: منجيبه من قيمة الـ T0CON المعطى

---

واذا ما كان معطينا القيمة المدخلة على التايمر

ممكن نوجدها من ؟؟

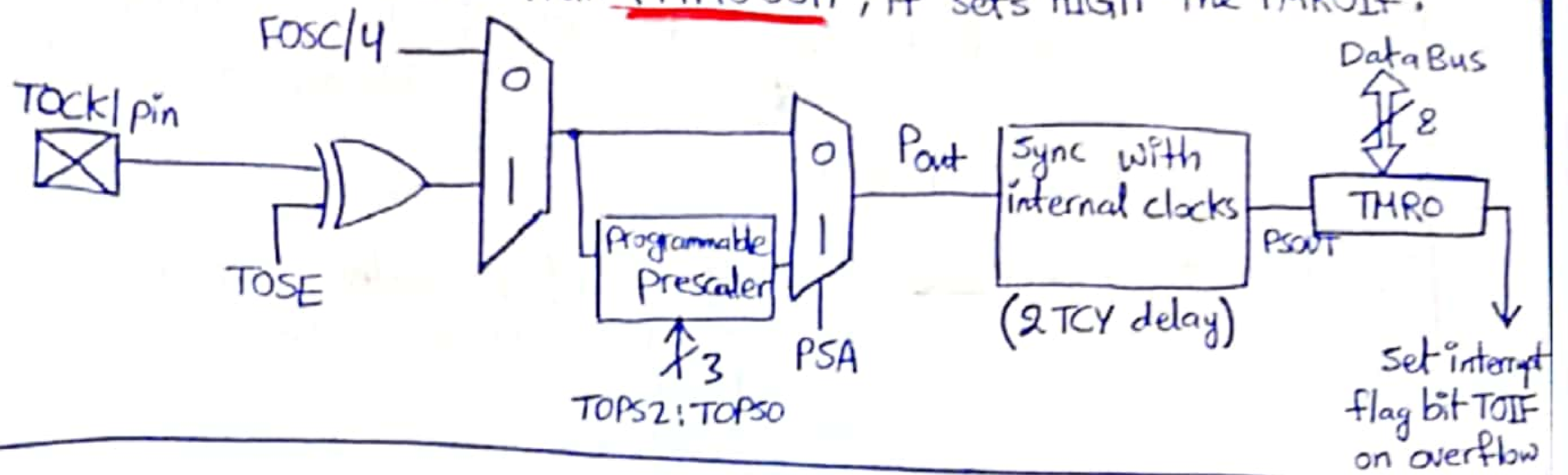



حسب الـ T0CON Value جاي منحسبها من Timer0 prescaler

# 8 bit mode programming of Timer0 →

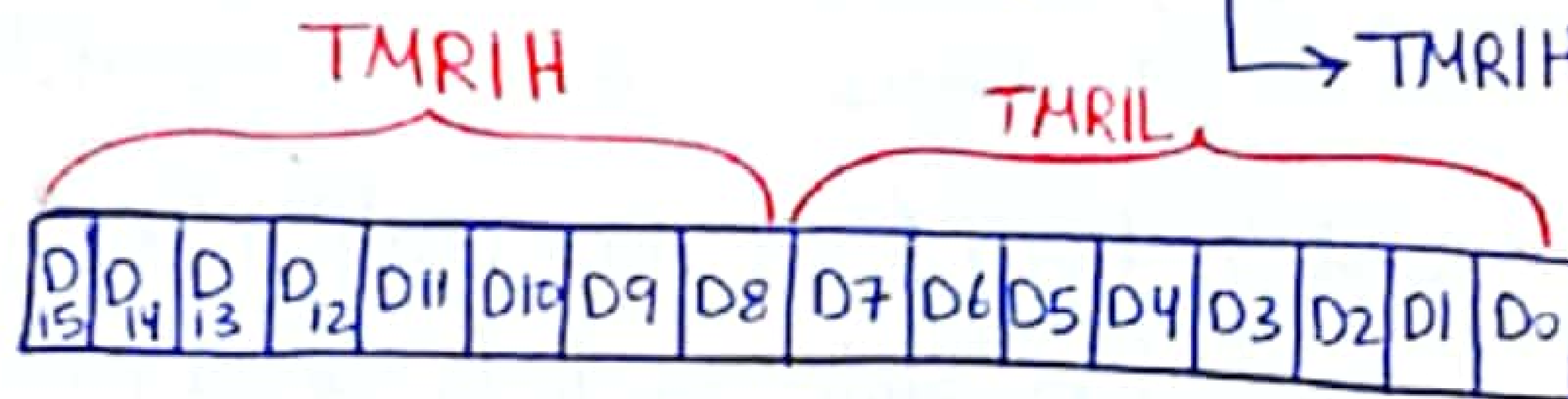
مو مقصود السلايدات
قراءة فقط لفهم الرسمة

It allows only values of [00 – FFH] to be loaded into the timer's register TMR0L. After the timer is started, it starts to count up by incrementing the TMR0L register. It counts up until it reaches its limit of FFH. When it rolls over from FFH to 00H, it sets HIGH the TMR0IF.

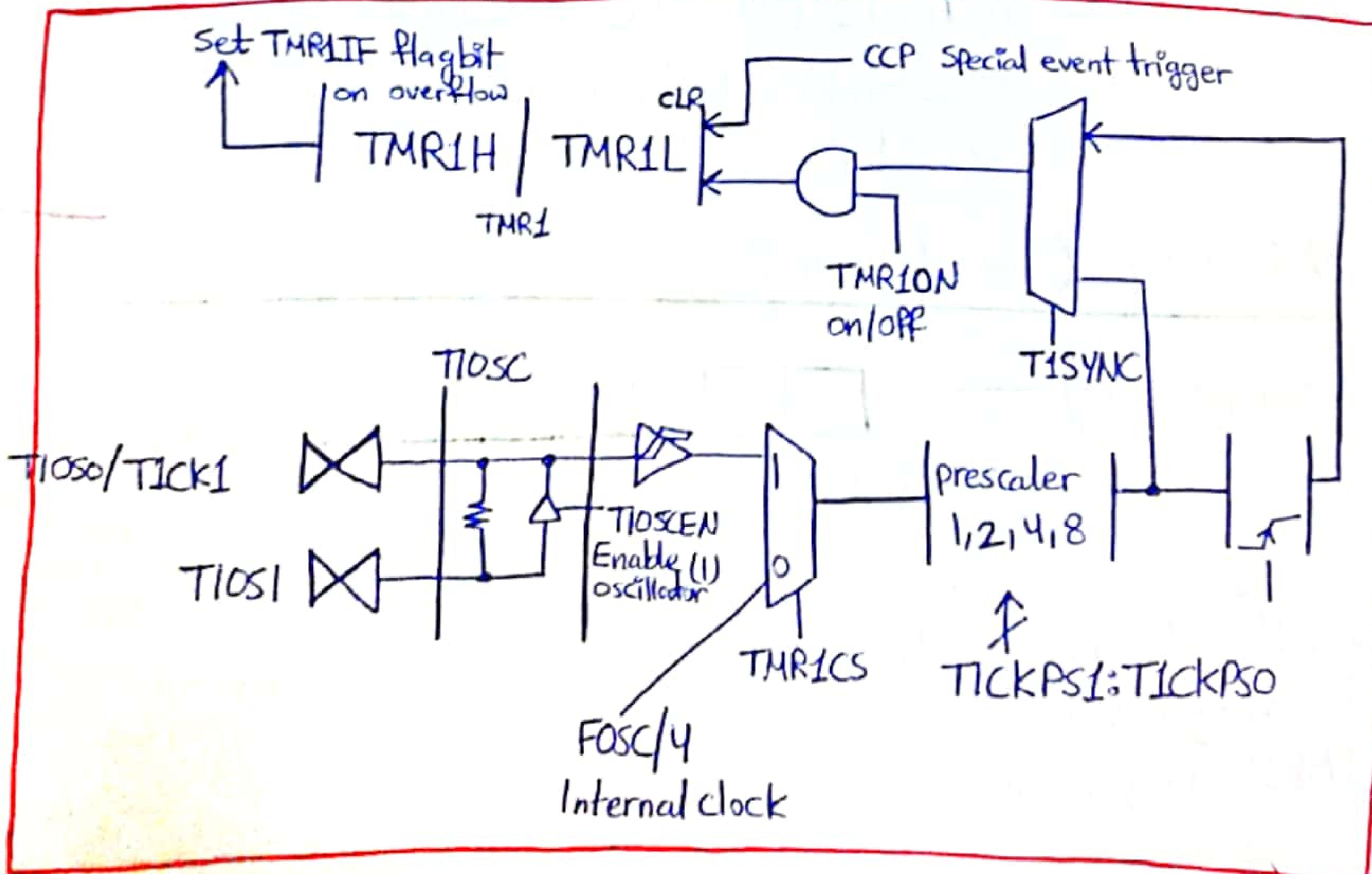



# Timer1 programming →

- It's a 16 bit timer, split into two bytes → TMR1L (Timer1 low byte)
  → TMR1H (Timer1 high byte)




- It doesn't support 8 bit mode unlike timer0
- It has T1CON (Timer1 control) register in addition to the TMR1IF (Timer1 interrupt flag)
- The timer1 interrupt flag (TMR1IF) bit goes high when TMR1H:TMR1L overflows from FFFF to 0000
- It has the prescaler option, only supports factors of 1:1, 1:2, 1:4 and 1:8

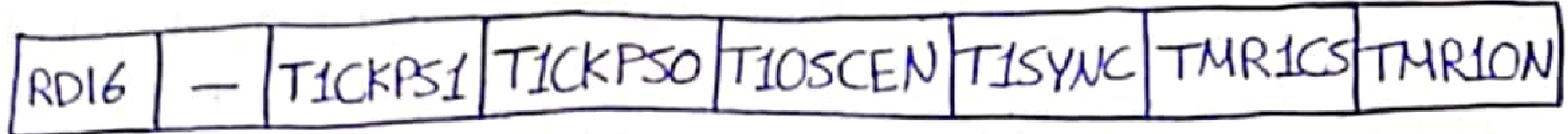



Timer1 Block Diagram



Scanned with CamScanner

* Timer1 control (T1CON) register →



| RD16 | — | T1CKPS1 | T1CKPS0 | T1OSCEN | T1SYNC | TMR1CS | TMR1ON |
|---|---|---|---|---|---|---|---|

RD16 → $D_7$ 16 bit read/write enable bit
1 = Timer1 16 bit is accessible in one 16 bit operation
0 = Timer1 16 bit is accessible in two 8 bit operations.

D6 not used

T1CKPS2:T1CKPS0 → D5 D4 Timer1 prescalar selector

| D5 | D4 | | |
|---|---|---|---|
| 0 | 0 | = 1:1 | Prescale value |
| 0 | 1 | = 1:2 | " " |
| 1 | 0 | = 1:4 | " " |
| 1 | 1 | = 1:8 | " " |

T1OSCEN → D3, Timer1 Oscillator enable bit
1 = Timer1 oscillator is enabled
0 = Timer1 " " shutoff

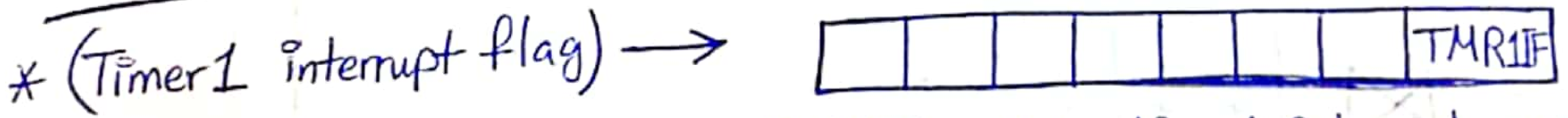
T1SYNC → D2, Timer1 synchronization (used only when TMR1CS=1 for counter mode to synchronize external clock input)
If TMR1CS=0, this bit isn't used.

TMR1CS → D1, Timer1 clock source select bit
1 = External clock from pin RC0/T1CK1
0 = Internal " (FOSC/4 from XTAL)

TMR1ON → D0, Timer1 ON and OFF control bit
1 = Enable (start) timer1
0 = Stop Timer1

↳ T1CON Register (Timer1 control)

* (Timer1 interrupt flag) →



| | | | | | | | TMR1IF |
|---|---|---|---|---|---|---|---|

TMR1IF, D1 timer1 interrupt overflow flag bit
0 = Timer1 didn't overflow
1 = " has overflowed (FFFF to 0000)

when TMR1H:TMR1L overflows from FFFF → 0000, this flag is raised.
We monitor this flag bit before we reload the TMR1H:TMR1L registers.

Scanned with CamScanner

## 9.2) Counter Programming →

- Used to counts event outside the PIC
- When the timer is used as a counter, if TOCS = 1, (→ timer0 clock source) it's a pulse outside the PIC18 that increments the TH, TL registers.

[TOCS bit in TOCON register] → $D_5 = 1$

TOCS in TOCON reg. determines the clock source

* If TOCS = 0, the timer gets pulses from the crystal oscillator connected to the OSC1 and OSC2 pins (FOSC/4)
* If TOCS = 1, the timer is used as a counter and gets its pulses from outside the PIC18

  So the counter counts up as pulses are fed from pin RA4 (PORTA.4) TOCK1 (Counter)
* If TMR1CS = 1, the timer1 counts up as clock pulses are fed into pin RC0 (PORTC.0)

**Ex(9.22)** Find the value for TOCON if we want to program Timer0 as an 8bit mode counter, no prescaler. Use an external clock for the clock source and increment on the +ve edge. → Low to high transition (+ve transition) / D4

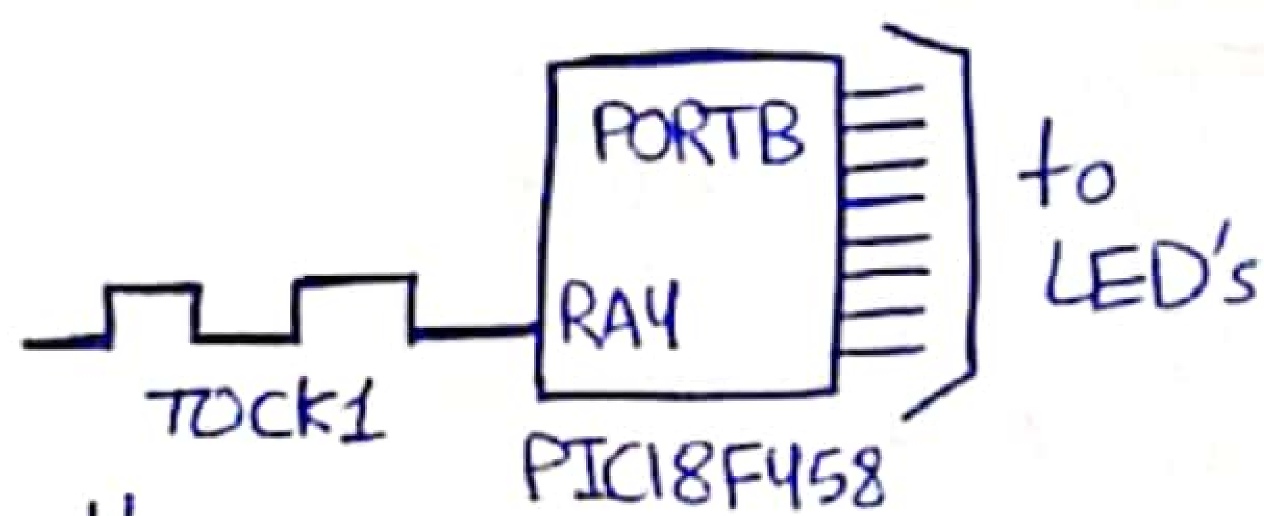
TOCON = 0110 1000   8 bit, external clock source, no prescaler

**Ex(9.23)** Assuming that clock pulses are fed into pin TOCK1, write a program for Counter0 in 8bit mode to count the pulses and display the state of the TMR0L count on PORTB?

```
        BSF    TRISA, RA4       → PORTA.4 as an input for clock
        CLRF   TRISB            → PORTB as an output
        MOVLW  68H              → Timer0, 8bit, ext CLK, no prescale (+ve edge)
        MOVWF  TOCON            → Load TOCON reg.
HERE    MOVLW  0H               → TMR0L = 0
        MOVWF  TMR0L            → Load Timer0
        BCF    INTCON, TMR0IF   → clear timer interrupt flag bit
        BSF    TOCON, TMR0ON    → start Timer0
AGAIN   MOVFF  TMR0L, PORTB     → display the count on PORTB
        BTFSS  INTCON, TMR0IF   → monitor Timer0 flag until
        BRA    AGAIN            → it rolls over
        BCF    TOCON, TMR0ON    → stop Timer0
        GOTO   HERE
```

(بالكاش Loading لحل عن طريق اعتبار صفر MOVFF TMR0L PORTB)

PORTB is connected to 8 LEDs and input TOCK1 to pulse.



# Using external crystal for Timer1 clock →

* Timer1 comes with two options :
It uses either (1) the clock fed into the T1CK1 pin or (2) the clock from a crystal connected to the T1OSI and T1OSO pins
  - (1) → T1OSCEN = 0
  - (2) → T1OSCEN = 1
    * 32 kHZ crystal is connected
    * Used for saving power during SLEEP mode → doesn't disable Timer1 while the main crystal is shut down.

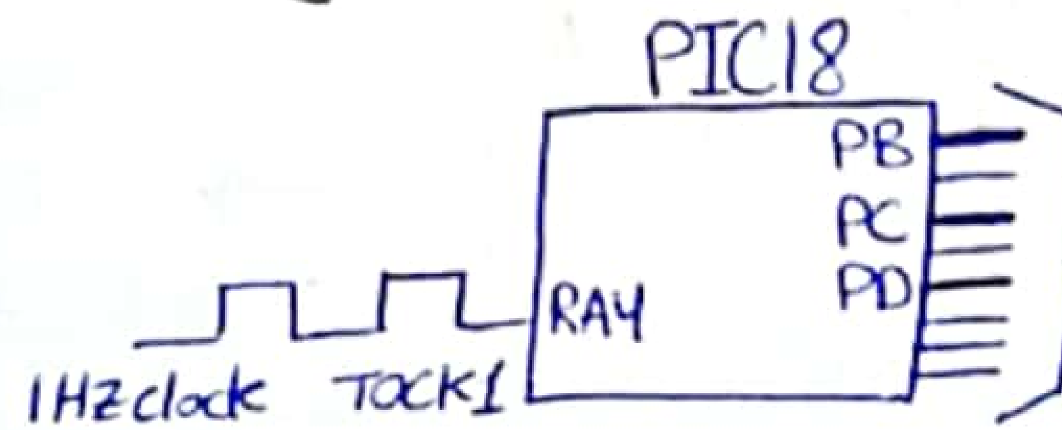
Ex(9.24) Assume that a 1HZ frequency pulse is connected to input for Timer0 (pin TOCK1). Write a program to display counter0 on PORTB, PORTC, and PORTD in decimal. Set the initial value of TMR0L to -60 ?

```
→  NUME     EQU   0X00 → RAM loc. for NUME
   QU       EQU   0X20 → " " quotient
   RMND_L   EQU   0X30 → the LSD loc.
   RMND_M   EQU   0X31 → the middle significiant digit loc.
   RMND_H   EQU   0X32 → the MSD loc.
   MYDEN    EQU   D'10' → Value for divide by 10
        BSF    TRISA, RA4     → RA4 as an input
        MOVLW  68H            → Timer0, 8bit, ext CLK, no prescale
        MOVWF  T0CON          → load T0CON reg.
HERE    MOVLW  0H             → TMR0L = 0
        MOVWF  TMR0L          → load timer0
        BCF    INTCON, TMR0IF → clear TMR0IF
        BSF    T0CON, TMR0ON  → start timer0
AGAIN   MOVF   TMR0L, W       → save the count in WREG
        CALL   BINLASC_CON
        BTFSS  INTCON, TMR0IF → monitor timer0 flag until
        BRA    AGAIN          → it rolls over
        BCF    T0CON, TMR0ON  → stop timer0
        GOTO   HERE
; ______________________ converting 8bit binary to decimal
BINLDEC_CON
        MOVFF  PORTB, W
        MOVWF  NUME      → load numerator
        MOVLW  MYDEN     → W = 10, the denomintor
        CLRF   QU        → clear quotient
D_1     INCF   QU        → inc. " for every subtract
        SUBWF  NUME      → subtract WREG from NUME value
        BC     D_1       → if positive go back
        ADDWF  NUME      → once too many, first digit
        DECF   QU        → " " " for quotient
        MOVFF  NUME, RMND_L → save the first digit
```



Scanned with CamScanner

```
      MOVFF  QU, NUME      → repeat the process one more time
      CLRF   QU            → clear quotient
D_2   INCF   QU
      SUBWF  NUME          → subtract w from NUME value
      BC     D_2
      ADDWF  NUME          → once too many
      DECF   QU
      MOVFF  NUME, RMND_M  → 2nd digit
      MOVFF  QU, RMND_H    → 3rd digit
      RETURN
```

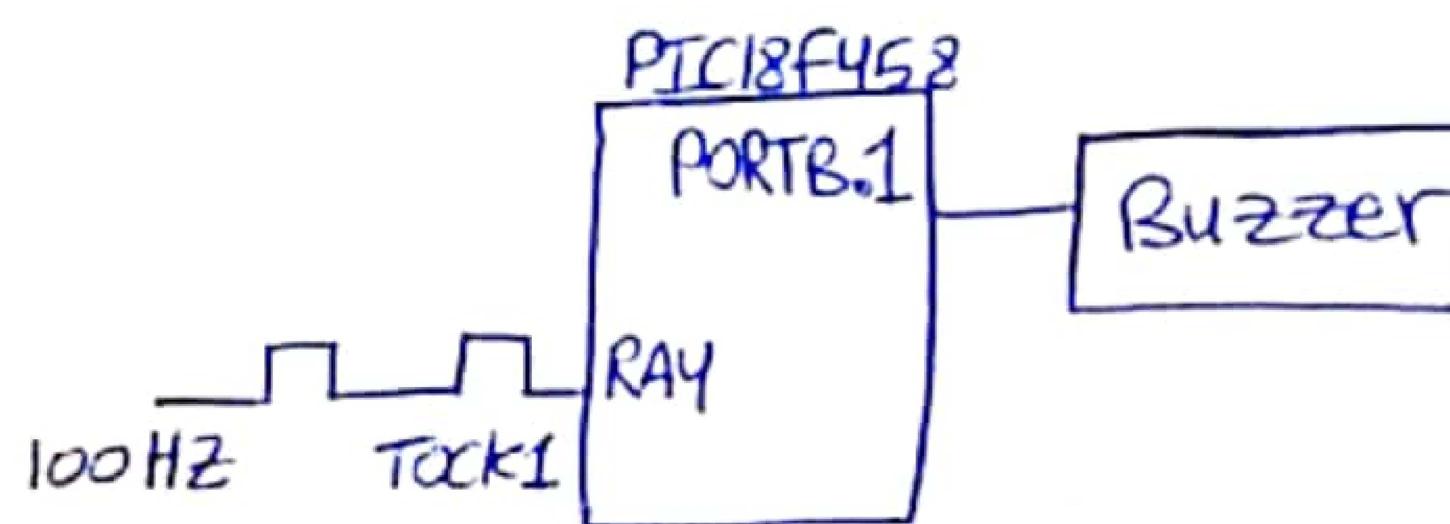



Ex (9.26) Assuming that clock pulses are fed into pin TOCK1 and a buzzer is connected to pin PORTB.1, write a program for counter0 in 8bit mode to sound the Buzzer every 100 pulses?

We set the initial counter value to −100 (9C in hex), then the counter counts up until it reaches FF. Upon overflow, we can count the buzzer by toggeling PORTB.1 pin

```
      BCF    TRISB,1         → RB1 as an output
      BSF    TRISA,4         → RA4 input for clock-in
      MOVLW  68H             → Timer0, 8bit, ext clk, no prescale
      MOVWF  T0CON           → load T0CON reg.
      MOVLW  -D'100'         → TMR0L = 0
      MOVWF  TMR0L           → load Timer0
      BCF    INTCON, TMR0IF  → clear timer interrupt flag bit
      BSF    T0CON, TMR0ON   → start Timer0
AGAIN BTFSS  INTCON, TMR0IF  → monitor timer0 flag until
      BRA    AGAIN           → it rolls over
      BCF    T0CON, TMR0ON   → stop Timer0
OVER  BTG    PORTB,1         → sound the buzzer
      CALL   DELAY           → quarter second delay
      GOTO   OVER            → forever
```

Bit 1 of PORTB is connected to a buzzer and input TOCK1 to a pulse.



Ex (9.27) Assume that a 1HZ freq. pulse is connected to input for Timer1 (pin PORTC.0). Write a program to display the counter values of TMR1H and TMR1L on ports B and D. Set the initial values to 0. Use Timer1, 16 bit mode, no prescaler, and +ve edge clock?

```
      BSF    TRISC, RC0      → RC0 as an input
      CLRF   TRISB           → PORTB, output
      CLRF   TRISD           → PORTD, "
      MOVLW  02H             → Timer1, 16bit, ext CLK, no prescaler
      MOVWF  T1CON           → load T1CON reg.
HERE  MOVLW  0H              → TMR1H = 0, the high byte
      MOVWF  TMR1H           → load Timer1 high byte
      MOVLW  0H              → Timer1 L = 0, the low byte
      MOVWF  TMR1L           → load timer1 low byte
      BCF    PIR1, TMR1IF    → clear timer interrupt flag bit
      BSF    T1CON, TMR1ON   → Start timer1
AGAIN MOVFF  TMR1H, PORTD    → display high byte count
      MOVFF  TMR1L, PORTB    → "  low  "  "
      BTFSS  PIR1, TMR1IF    → monitor Timer1 flag until
      BRA    AGAIN           → it rolls over
      BCF    PIR1, TMR1ON    → stop timer1
      GOTO   HERE
```

Scanned with CamScanner