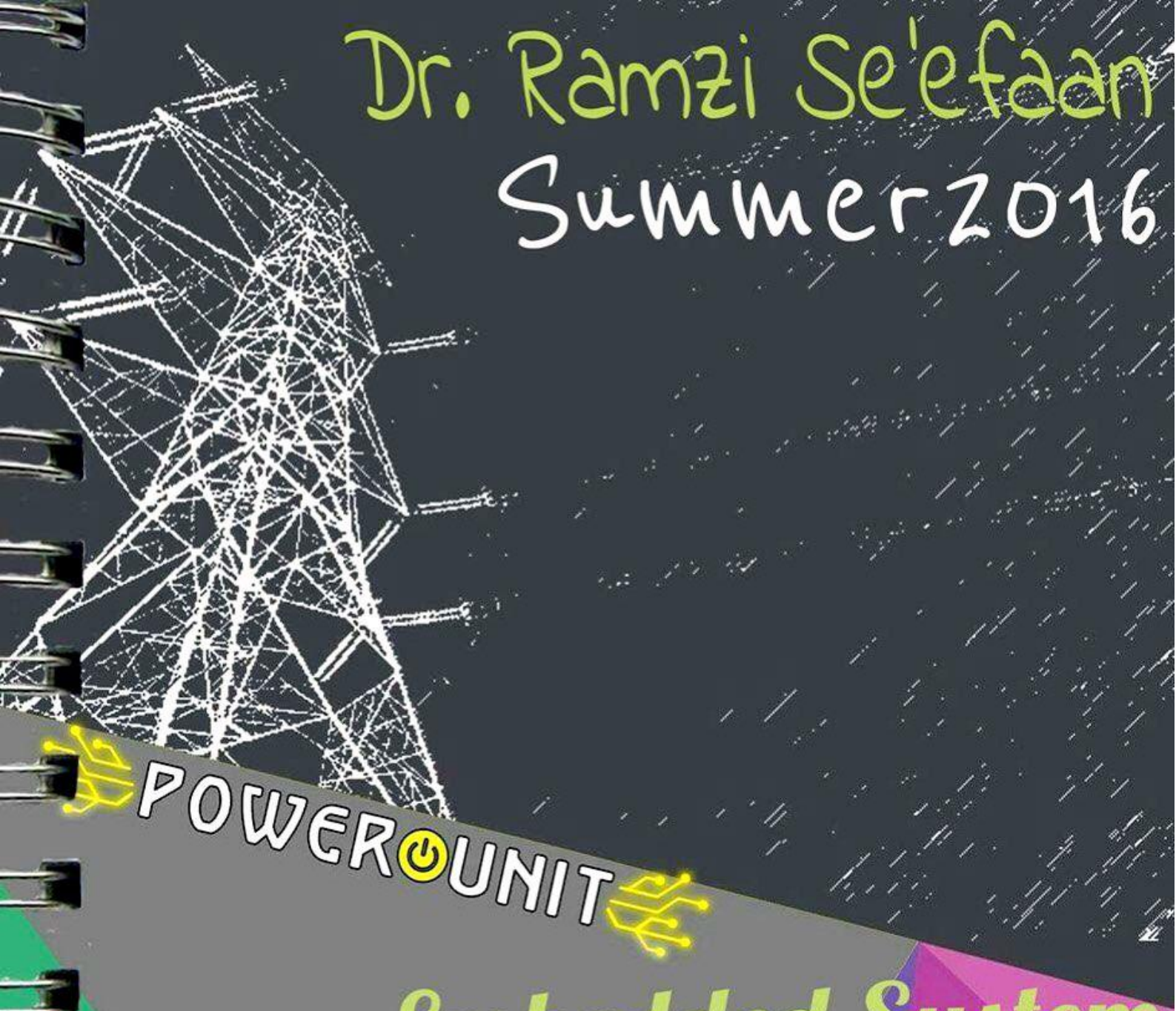


Dr. Ramzi Se'efaan

Summer2016

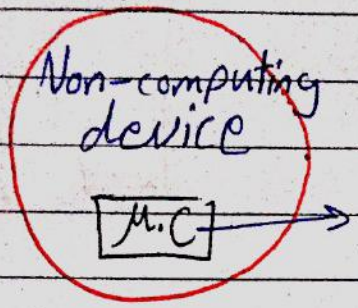
POWERUNIT

Embedded System

By: Mohammad Abuhashia

CHAPTER (1): Introduction.

* Embedded system:



* every M.C has its own DATA sheet.

* Real Time Constraints:

- There are deadlines for the actions or results.
- if the result came after the deadline, it is useless

* To understand any ES:

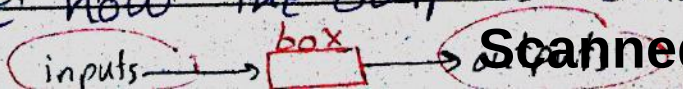
1. Inputs.
2. Outputs.
3. User Interaction.
4. Link to other systems

⟹ After That:

1. Hardware (M.C)
2. Software (code)
   - sometimes we have the code and some times we haven't

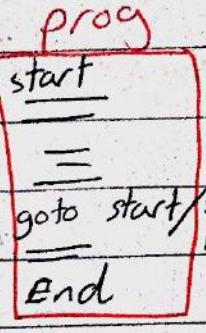
⟹ But if we could have the code we can rebuilt the whole system.

⟹ if we don't have the code: we can imagine that the system is a box and change the inputs to see how the outputs change.



* Software Driven: all ES components are controlled by program.

* Reliable: it should give true results almost 100% of the time.



* Example: (refrigerator)

T_{desire} if $T_{act} > T_{des}$ ⟹ switch ON compressor.

T_{Actual} else ⟹ switch off compressor

* Main Components of the computer:

① CPU: control & execution & processing.

② Memory: storage → DATA memory
→ Program memory

it holds the program (permanent)

it holds the values of the variables. (volatile): it is lost when power is off!

③ Input/output: To deal with the external world.

④ Buses: To connect the components above.

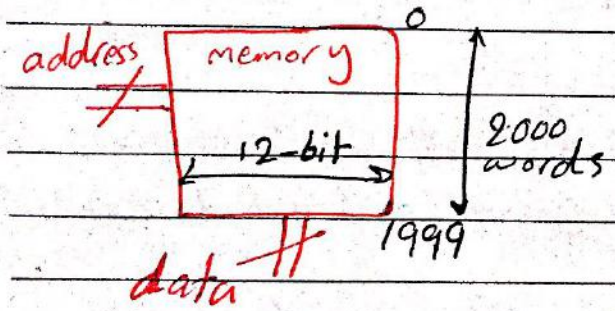
* DATA memory: faster and takes less power in writing.

* when we give power to CPU it will take the first instruction and start execution, while executing the instruction the CPU brings (fetch) the next instruction and that called ⇒ "pipelining"

* VNA (Von Neumann Architecture): only 2 buses and all modules take from them.
  - bottleneck (buses)
  - slower [because pipelining is hard]
  - is simpler.

* Harvard: 2 separate buses for each module.
  - faster.

* word size: the size of block that can be read or written in one shot.



$\Rightarrow$ size of memory $= (12 * 2000)$ bits $= \left(\frac{12 * 2000}{8}\right)$ bytes

$\Rightarrow$ minimum data bus $= 12$ bits

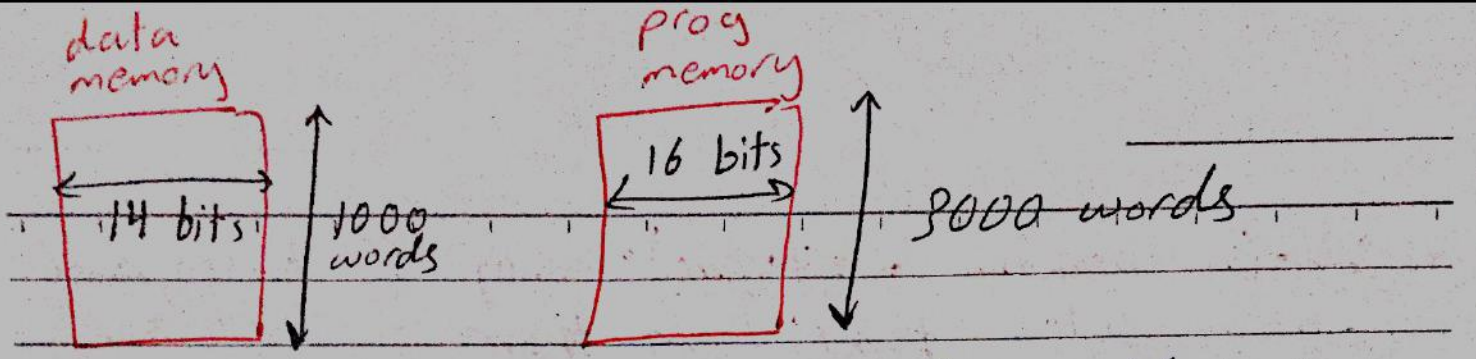
minimum address bus $= \lceil \log_2 2000 \rceil = 11$ bits

address 10 bits
  ↳ address 0 - 1023

address 11 bits
  ↳ address 0 - 2047

* Harvard arch: variable bus sizes for each module.

* PIC µ.c's use Harvard arch.



$\text{memory size} = \frac{1000 * 14}{8}$ bytes

$\text{memory size} = \frac{3000 * 16}{8}$ bytes

	DATA memory	prog memory
Harvard	$\text{data}_{\text{bus}} = 14$ bits $\text{address}_{\text{bus}} = \lceil \log_2 1000 \rceil = 10$ bits	$\text{data}_{\text{bus}} = 16$ bits $\text{address}_{\text{bus}} = \lceil \log_2 3000 \rceil = 12$ bits
VNA	data bus = max{14, 16} = 16 bits address bus = max{10, 12} = 12 bits	

* **Instruction Set:** description for all instructions that you can build your programs out of them.

→ **CISC:**
- too many instructions and addresing modes (many of them are complicated) SQRT, MVI, DIV
- more powerful programs.
- faster compilation.

→ **RISC:**
- less number of instructions (only the simple instructions) ADD, SUB, shift, XOR
- few addressing modes.
- longer programs.
- faster execution.

assembly → compiler → machine

compilation time is longer.

All instructions have same size & exec. time

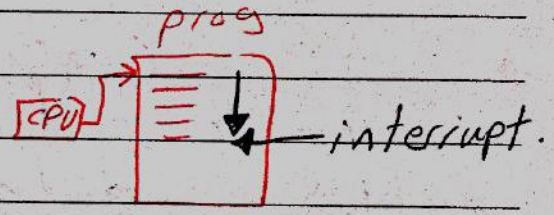
because pipelining is more efficient.

* PIC μ.C's ⇒ RISC.

* Addressing mode: how to identify the variables in the instruction.

- μ.C ⇒ we don't care alot for the speed since μ.c just take decisions.
- μ.processor ⇒ it must be more powerful because that, we care for the speed.

* Interrupt: something that interrupts the execution of the code.



* All members within the same family have the same core (CPU and instruction set), same instructions set, and same format.

⇒ the program that is written on μ.C can be moved to any μ.C within the same family without modifications.

⇒ when move from μ.C to another within the same family, memory size changes and number of I/O's change

16F84A
16F87XA
⇒ the same family. (They have the same instruction set)

* DIP: Dual In-line Packaging:

* # of pins and spacing between pins determine chip size.

↳ more pins ⇒ more peripherals.

* Working Register (W-Reg) [Accumulator]

↳ holds the result of the last instruction

* 8-bits μ.C ⇒ the word size of data memory is 8-bits and all operations are done on 8-bits values ⇒ data memory data bus = 8 bits.

* We will deal with 8-bits PIC μ.C

⇒ 8-bits μ.C are divided into 3-families:

1. Base line.
2. Mid range.
3. High performance.

* PIC are divided into series.

12F508 → 12 series

* Note: from the name we Can't determine the family.

16F87X A → don't care.

- 16 → 16 series
- F → means that uses flash memory.
- 87X → μ.C # in the series.
- A → Advanced Technology:

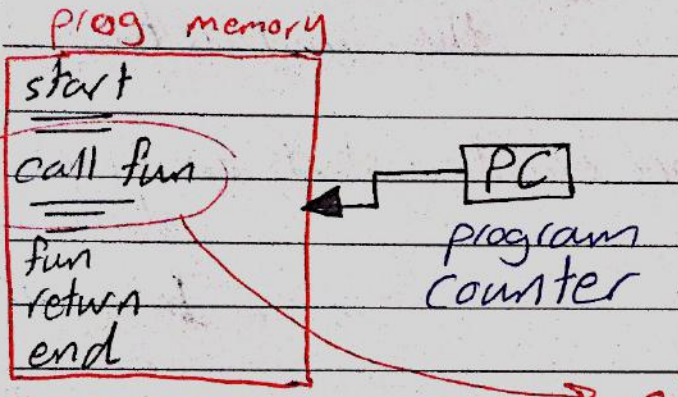
There is another μ.C but without the letter "A", both have the same architecture but the one with "A" has specific advanced technology like saving power.

16C84 C → CMOS
L → Low power.

* stack: it holds return address.

* on power up ⇒ PC = 0

⇒ program is executed sequentially.



PC: it holds the address of next inst. to be executed and it is auto increment.

a) stack = PC
b) PC = address (fun)

* 2-levels stack ⇒ means that you can do 2 nested calls.

* stack: first in - last out.

```
start
   call sub1 → PC → stack
               address(sub1) → PC
   Call sub2 → PC → stack
               address(sub2) → PC
sub1:
   return → stack → PC
sub2:
   return → stack → PC
```

This code is single level stack.

```
start
   call sub1 → stack ← PC
               address → PC
               (sub1)
sub1
   call sub2 → PC → stack
   return
sub2:
   return
```

This code needs 2 levels stack.

Five Apple

* instruction:

opcode | operands → variables that the instruction works on..

→ unique (can't be two inst. have the same op code-)

→ ① specifies type of instruction. ⇒ ② and the format of the instruction.

* Reset vector: The address from where execution starts on Reset (address 00)

* Interrupt vector: " " " " " " " Interrupt.

* ALU (Arithmatic logical Unit): is the part of the CPU that is responsible about executing the instructions.

* In case the instruction works on two operands:

① one of them comes from W-Reg.
② the second is from either the instruction itself (literal) or data memory.

data mem.

15	5
16	6

Add &15,&16

⇓

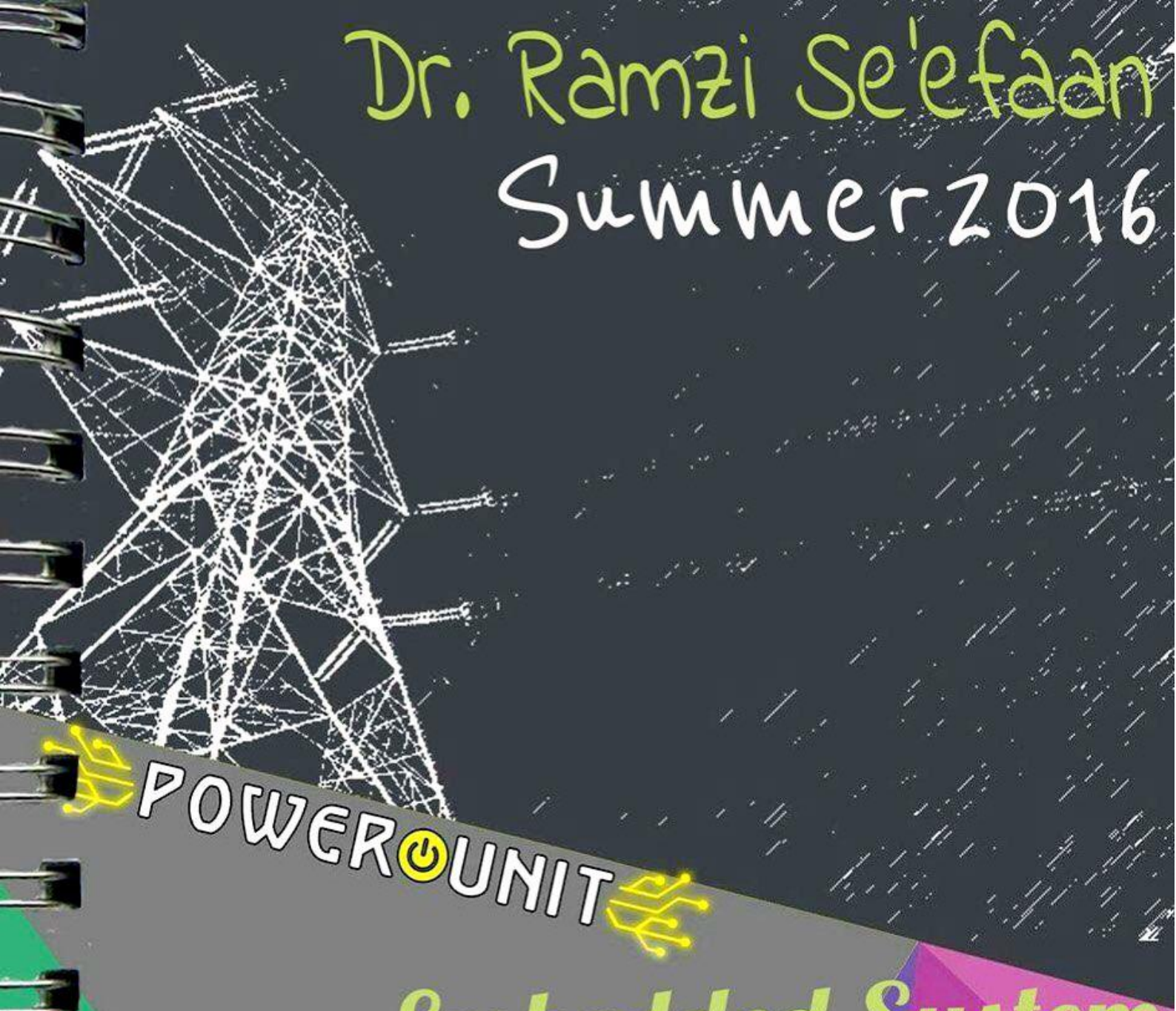
move &15 to W-Reg

Add &16

12-bits

opcode | operands

⇒ size of the instruction is the limitation that prevents using 2-data memory locations in one instruction.



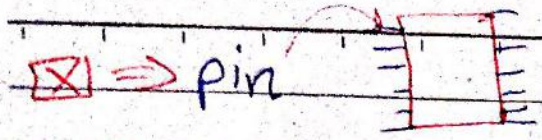
Dr. Ramzi Se'efaan

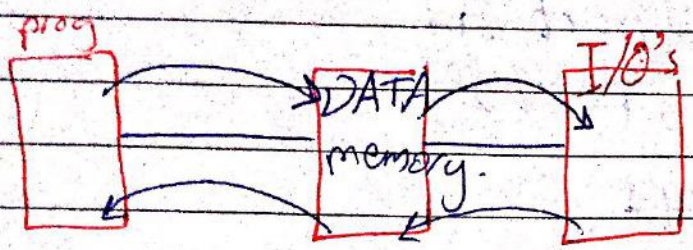
Summer2016

POWERUNIT

Embedded System

By: Mohammad Abuhashia





⇒ "memory mapped I/O's"

⇒ memory mapped I/O's: The program deals with I/O's through the data memory, not directly.

* status Reg: it contains information about last executed instruction (Zero, negative ---).

end of CH1

we studied 12F508 / 12F509 Now in CH2 we will study (16F84A) ⇒ mid range family.

⇒ CHAPTER (2): The PIC 16-series.

* data memory تحتوي على variable زي x, y program memory تحتوي على address زي inst. زي

EEPROM: it holds data ⇒ this data not frequently changed. (settings)
↳ (permanent memory)

1K instructions: is the longest program that can be stored On PIC 16F84A.

In PIC 16F84A: There are 3 peripherals: ⟹

1. 8 bits timer (for delays).
2. 5 pins parallel port (port A)
3. 8 " " " (Port B)

* 18 pins in PIC 16F84A

* PIC 16F84A has 18 pins such that:

1- 5 for port A 2- 8 for port B

3- 2 V_{DD} (power supply) V_{SS} (ground) 4- 2 oscillator (for clock)

5- 1 for $\overline{MCLR}$ (external reset)

↳ Active low: active when = 0

↳ it will reset when $\overline{MCLR} = 0$

points 3 & 4 & 5 must be connected without them the PIC won't work.

* we will deal in CH2 ⟹ with 14-bit instruction.

* The maximum number of I/O's devices could connect to PIC 16F84A → (13) devices.

↳ come from port A & B

* for 16F84A:

prog memory size = $\frac{1K * 14}{8}$ bytes.

minimum address bus required for PIC 16F84A:

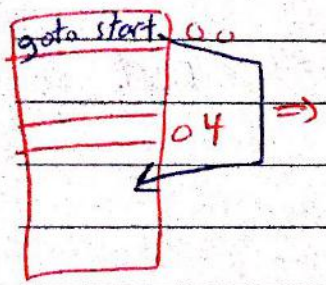
$\lceil \log_2 1K \rceil = 10$ bits However, 13 bits is used.

* file register = data memory.

=> program memory:

* address 00: reset vector

* address 04: interrupt vector. => 04 always used for interrupt vector so we can't write on it.



=> in order not to overwrite the interrupt code.

* reset & interrupt vector for 16F84A always fixed.

* call

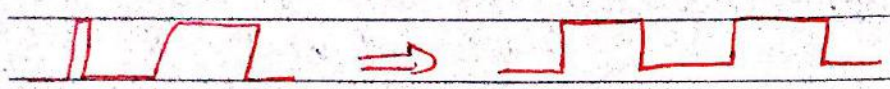
PC --interrupt / push--> stack

PC <--return / POP-- stack

* Configwation word: 14 bits stored in program memory.
  - can be written one time at program download.
  - can't be modified after program download.
  - it holds configwation information about the program.
    - ① OSC type: There are 4-types of OSC; crystal (XT), Low power (LP), High speed (HS), and RC OSC (RC) => They differ in freq range.
    - ② WDTE: watch dog timer enable
      WDT: it monitory the M.C and resets it if crash happens
      -> if WDT = 1 => the WDT is enabled.

* إذا أردنا تغيير نوع OSC يجب إعادة تنزيل البرنامج مرة جديدة.

③ PWRTE: if enabled (=0), it holds the µ.c in reset mode for short time on power up. in order for power to be stabilized on all components and for OSC to give reguler clock



④ CP: code protection (=0) ⇒ code is protected if someone want to read the code.

1K = 2^{10} = 1024

1K bytes = 1024 bytes

1K instructions = 2^{10} instructions = $\frac{2^{10} * 14 \text{ bits}}{8 \frac{\text{bits}}{\text{bytes}}}$

* DATA memory is divided in two ways:

① Vertically
- → special function registers (data memory locations, but have additional functions mainly for dealing with I/O's. in other words each of these SFRs have specific function.)
- → general purpose registers (data memory locations to hold values of variables)

② Horizontally: divided into 2 banks each of size 80 address

0 → 4FH
4 * 16 + 15
0 → 79
(80 locations)

⇒ total = 160 word

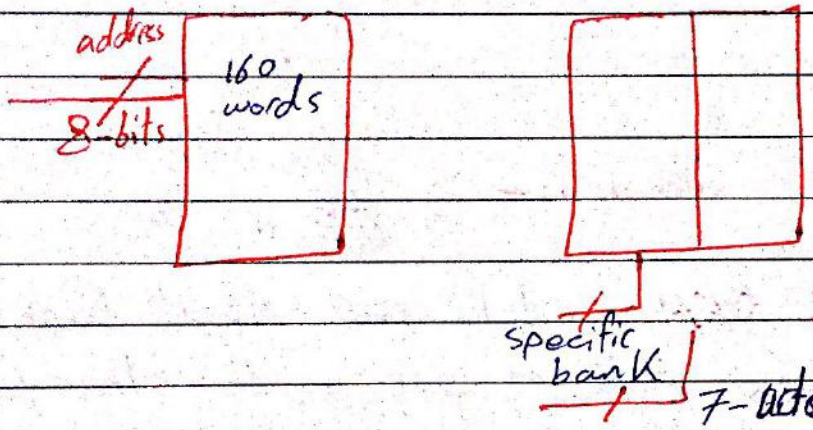
minimum address bus = $\lceil \log_2 160 \rceil$ = 8 bits

* If the memory is flat ⇒ 8 bits address is needed.

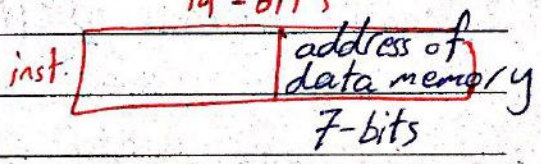
⇒ when the memory is divided into 2 banks

↳ we need 7 bits to address a data memory location inside the bank.

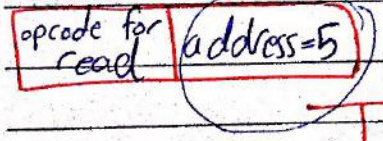
7-bits = $\lceil \log_2 80 \rceil$, However, we still need to select the bank.



* data memory address come from the instruction.



read address 5



↳ 8 bits if one bank
↳ 7 bits if two bank } we saved one bit due to dividing into 2 banks.

⇒ cost: we have to select the bank first.

* To access any data memory location:

1) select the bank, if not selected.
2) select the address inside the bank.

* bank switching is needed ⇒ only if you want to access a word on the other bank.

* Write a code to read data memory location 95H.

address: 8 bits
[1 bit | 7 bits] → address inside the bank.
bank

95H ⇒ [1 | 001 0101]
1 → bank 1
15H inside bank 1

⇒ code:

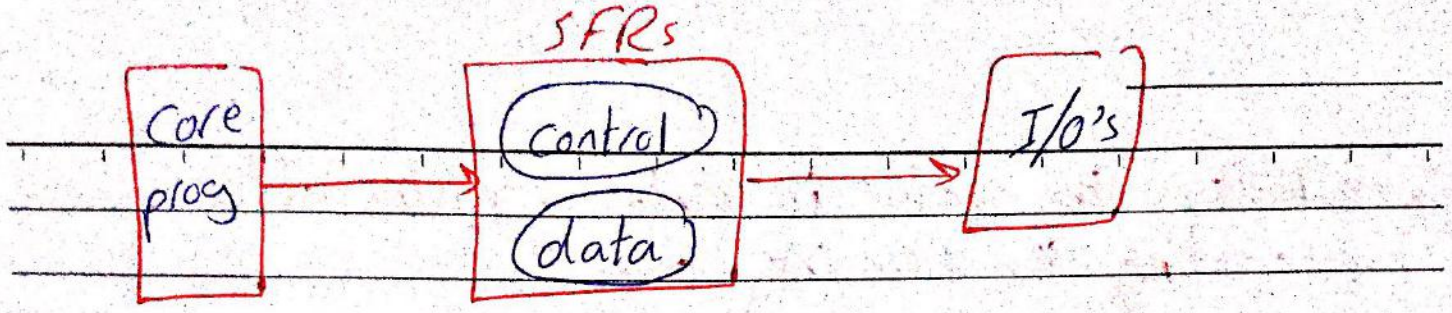
select bank 1
read address 15H
read address 25H → 1010 0101 ⇒ address A5 H

* when you deal with SFRs ⇒ take care about banks switching because some of them are on the other bank

* Some SFRs are in the two banks ⇒ if you modified one, the other will be also modefied.

instruction
[address] → 7-bits
bit → bank.

address 7-bits



⇒ control: the setting of the I/O.
data: input ⇒ the entered data.
output ⇒ the data you want to output.

* Bank selection: 1- direct addressing 2- indirect addressing

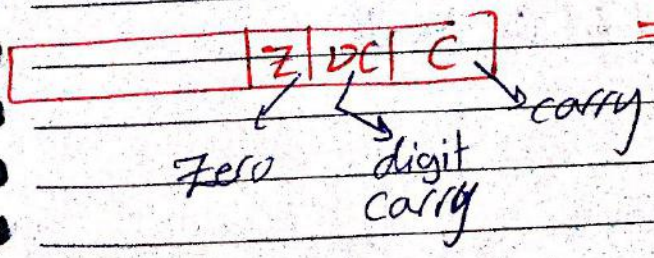
① Direct addressing: value is in the data memory and its address is in the instruction itself.

② Indirect addressing: value in data memory and its address is in SFR called File Select Register (FSR)

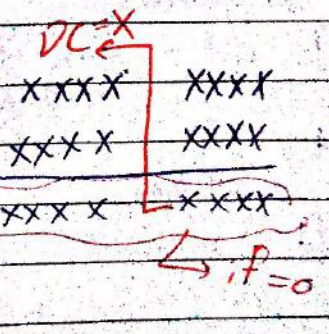
Direct → single bit if two banks (bit 5 of status reg.) (RP0)
→ 2 bits if 4 banks (bits 5 and 6 from status reg.) (RP1 & RP0)

Indirect → 2 banks ⇒ 8th bit of FSR
→ 4 banks ⇒ bit 7 of status (IRP) & bit 7 of (FSR)

* in slide (15): status reg.

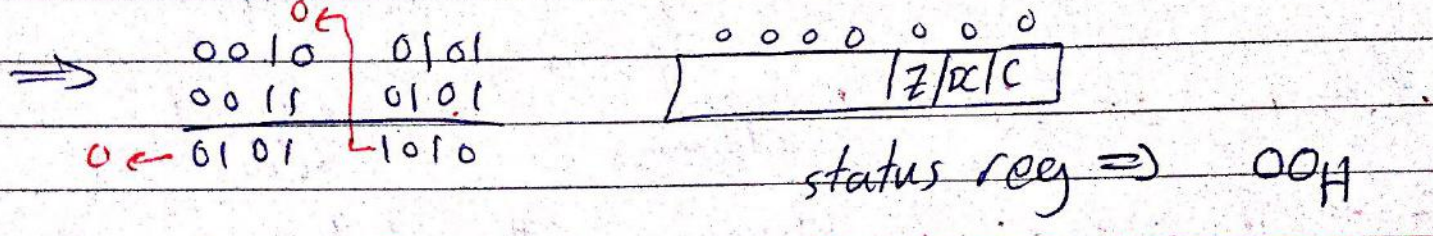


⇒



Ex. what is the value in status register after executing the following code?

code: clear status register → status 00
add 35H & 25H



Ex. Write pseudo code to read locations 15H, 33H, 95H, 82H, 31H.

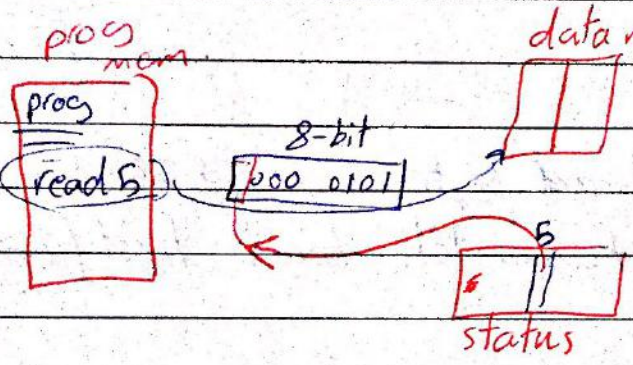
15H ⇒ 0001 0101 ⇒ b0
33H ⇒ 0011 0011 ⇒ b1
95H ⇒ 1001 0101 ⇒ b1
82H ⇒ 1000 0010 ⇒ b1
31H ⇒ 0011 0001 ⇒ b0

⇒ pseudo code:

clear RP0 in status
read 15H
33 { set RP0 in status
read 33H → 011 0011 (7-bits) & 1 bit for bank.
read 15H
read 02H
clear RP0 in status
read 31H

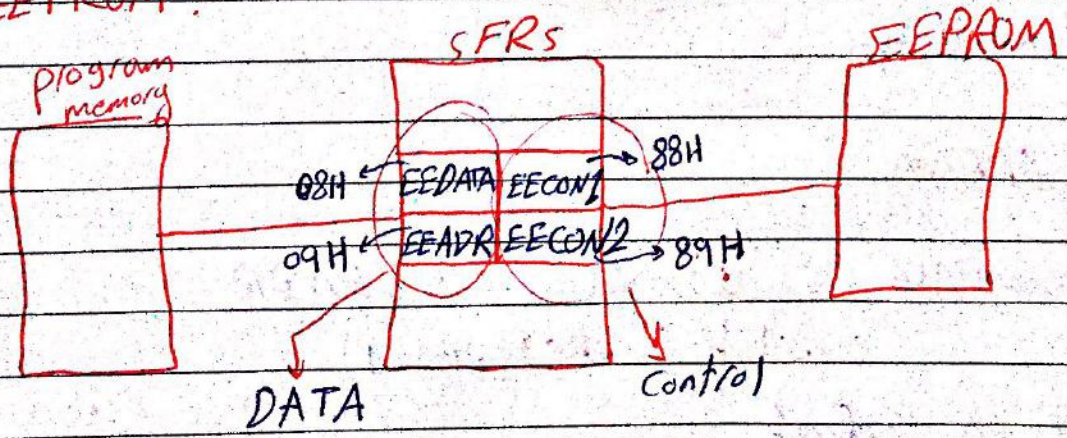
* we could divid into 4 banks but the problem would be in switching process (a lot of switching).

* direct & indirect addressing: value is in Data memory.
  - address is in FSR (FSR(7))
  - address is in the instruction. (RP0 status(5))



* addressing modes:
  1. Direct address.
  2. Indirect address.
  3. Literal (value is in the instruction) ⇒ Literal: we don't refer to the data memory.

* EEPROM:



⇒ most of the time: the data registers are in bank 0
" control " " " " " 1
⇒ Bank switching

slide (17):

* Read in EEPROM:

1) store the address in EEADR
2) switch banks (RP0 = 1)
3) Set the (RD) in EECON1
   ↳ start reading.
4) * after reading is over, the read byte is in EEDATA.
   To read the byte, you have to switch bank first (RP0=0)

⟹ To read the next byte:
- you have to repeat all steps above.
- RD bit is set by SW, and cleard by HW when read is over.

* WREN in EECON1 ⟹ means enable writing (Not start writing)
* WR ⟹ start writing.

* Writing to EEPROM:

1) DATA ⟶ EEDATA 2) address ⟶ EEADR (switch banks) (RP0=1)
3) set WREN in EECON1
4) write 55H in EECON2 } → To make sure that writing is intentional.
5) " AAH " "
6) Set WR bit in EECON1 (start writing)
7) when write is over EEIF=1

set by SW
cleared by HW

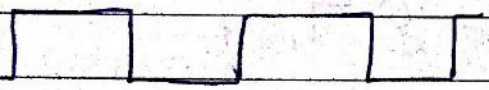
Five Apple

switch bounces

* To read the next byte: repeat all steps except step (3) since WREN already set.

(clear RD in EECON1) ⇒ X since RD cleared by HW not SW.

* The Clock:



T_{cycle}

$T_{cycle} \equiv T_{osc}$

$F = \frac{1}{T_{cycle}}$

$F_{osc} = \frac{1}{T_{osc}}$

⇒ The clock generated by OSC.

* T_{osc} is not enough to execute single instruction.

⇒ $T_{instruction}$: instruction cycle
↳ it is the time that is enough to fetch or execute single instruction.

* Each instruction:
is fetched then executed.

⇒ Time to fetch and execute one inst = $T_{inst}(fetch) + T_{inst}(exec.)$

factor

$T_{inst} = X * T_{osc}$ (in general)

→ (in PIC): $T_{inst} = 4\,T_{osc}$

each instruction requires $2\,T_{ins} = 8\,T_{osc}$ to be fetched and executed.

a prog composed of 8 instructions

$\Rightarrow$ 8 * The time per inst

$= 8 * 2\ T_{ins}$

$= \boxed{16\ T_{inst}} \Rightarrow$ without pipelining.

$\Rightarrow$ with pipelining: the time goes to half

$= 8\ T_{inst}$.

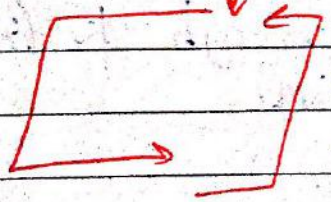
So, each instruction requiers 1 T_{inst} to be fetched and executed.

* Pipelining fails sometimes:

when program execution becomes not seq.

$PC \neq PC+1$ call, goto, return

seq. $\Rightarrow PC = PC+1$



failure of pipelining requires one additional T_{inst} to fetch another inst.

$T_{inst} = 4\ T_{osc}$

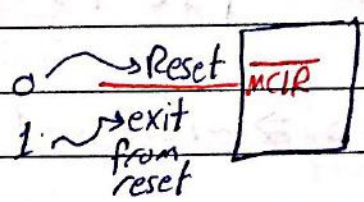
example: $F_{osc} = 1\ MHz \Rightarrow T_{osc} = \frac{1}{1MHz} = 1\ \mu s$

$\Rightarrow T_{ins} = 4\ T_{osc} = \boxed{4\ \mu s}$.

Power up & Reset:

* In power up: we want to Keep the PIC in Reset mode
  → inorder for power to be stabilized on all components.
  → ① Internal: manipulate some settings by SW.
  → ② External: connect some HW.

Reset: PC ← 00
↳ all registers & pins go into their default value.



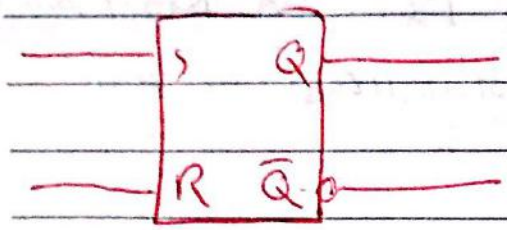
Reset ≡ MCLR → slide 22

* On power up: it takes time related to $\tau = RC$ until the capacitor charges and high enters on MCLR.

slide 22 → 2 problems in figure (a): → high value of current
↳ slow process of discharging.
(figure (b)):
* for high value of current:
→ we use 

* for slow process of discharging:
→ we use diode: 

at the begining .... V_{th} for $V_{DD} > V_{th}$ for V_C so the diode in reverse mode, Then after we finish V_{th} $V_C > V_{th}$ for V_{DD} so the diode would be in forward and that will speed up process of discharging.



S	R	Q	$\bar{Q}$	
0	0	Q	$\bar{Q}$	
0	1	0	1	
1	0	1	0	Reset
1	1	Unstable.		

* as long as $\bar{Q}=0$:

⇒ the PIC is in Reset.

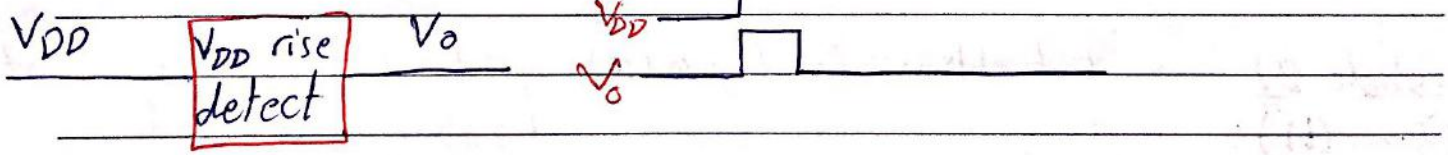
* $\bar{Q}=0$ ⇒ when $S=1$ & $R=0$

$S=1$ ⇒ when any of the following $=1$: (see figure slide 23)

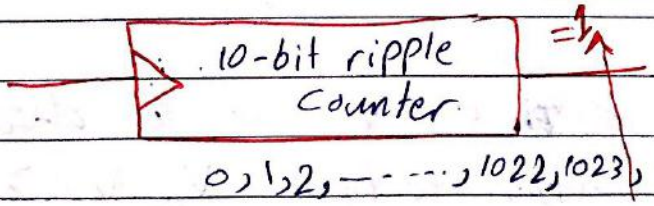
① $\overline{MCLR}=0$ ② WDT resets if and only if not in sleep mode

⇒ if PIC is in sleep mode ⇒ No Reset By WDT.

③ On power up.



* when we give power to V_{DD} a pulse will created and $S=1$, $R=0$ ↓ reset.
when the pulse end $S=0$, $R=0$ ⇒ No change.
⇒ No change until $R=1$ to exit the reset mode.



→ R=1 when all 3-inputs in AND gate are (1).

⇒ if (Enable PWRT) & (Enable OST) are zeroes.
No need for the counter since the output from them will be (1) anyway.

* After power up, the PIC stays in Reset mode

time = 1024 internal RC osc cycles + 1024 external osc cycles

72 ms + $\frac{1024}{F_{osc}}$

⇒ time = 72 ms + $\frac{1024}{F_{osc}}$ ⟹ if PWRTE = 1 & Enable OST = 1

by default enabled for osc types LP, XT, & HS in config. word.

in configuration word.

In config. word: $\overline{PWRTE}$ = 0 ⇒ delay on power up. (Active low)

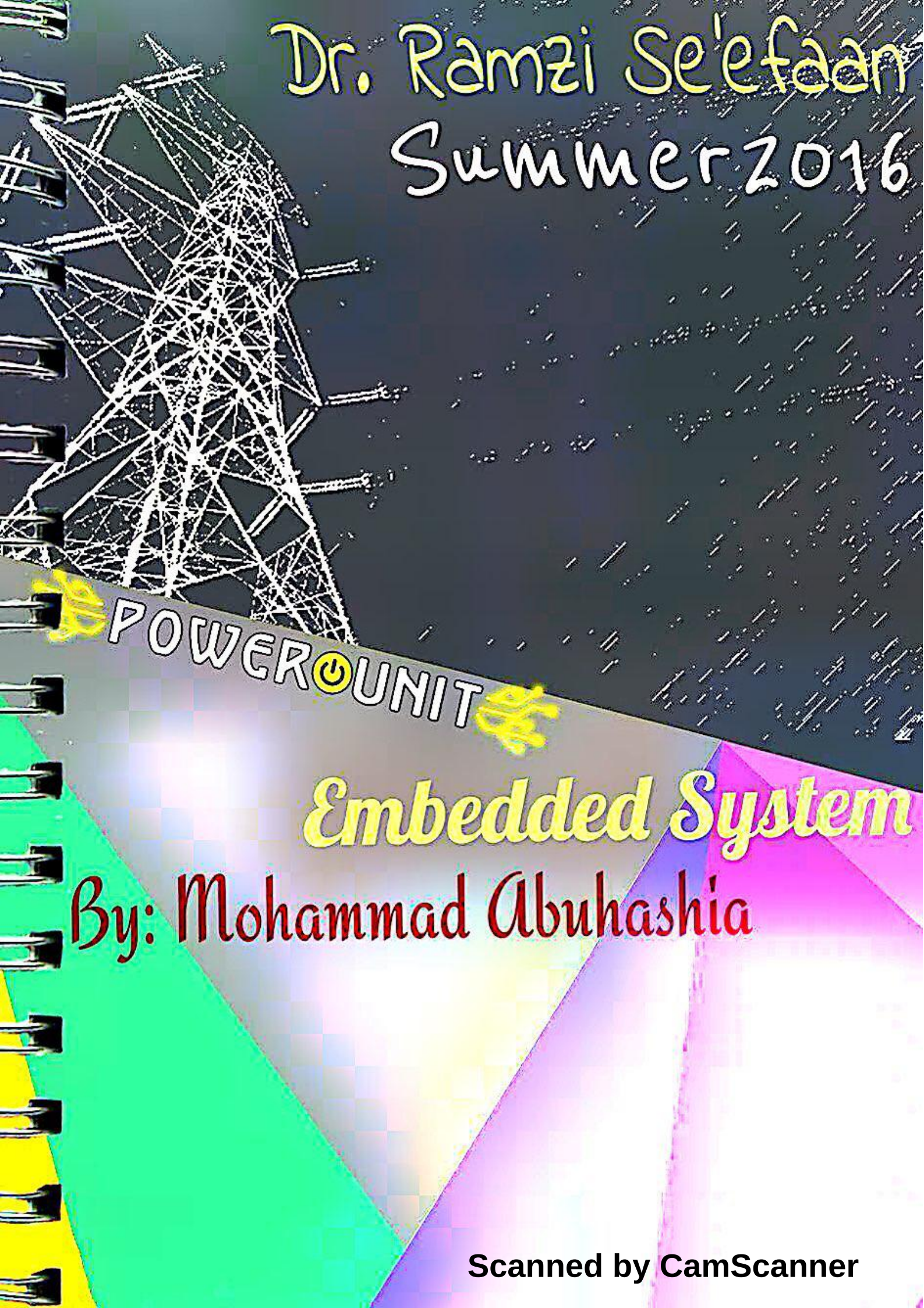
↳ in the circuit in slide (22) PWRTE = 0 it will output 1 (Active high)

⇒

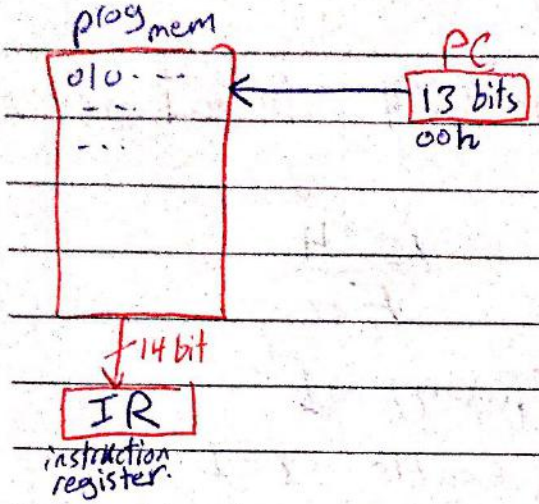
counter هل ينتظر
أم لا !!
↳ PWRTE = 0
لا ينتظر ⇐ (1)
PWRTE = 1
ينتظر

End of CH2

week 2



CHAPTER (4): Starting to Program.



High level lang.

HLL ——compiler——→ machine code.

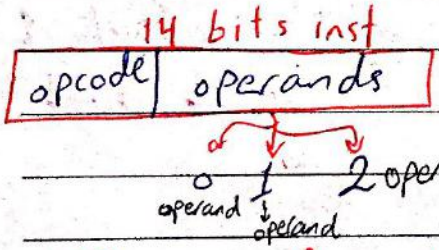
⟹ Not efficient execution time & memory usage.

slide (6):

* Destination bit (d)

if d=0 ⟹ result is written in W-Reg;

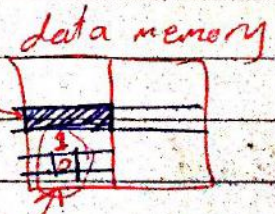
if d=1 ⟹ result is written in Data Memory;



* Types of instructions:

① Byte Oriented file register operations.

data memory address (f) ⇒ (7-bits)



② Bit oriented file register operations

↳ f, bit # (b) ⇒ 3 bits

③ Literal instructions, The value is in the instruction itself ⟹ No need for data memory

↳ Literal operand (K) ⇒ 8 bits

④ Control, The instruction that may change the path of execution (call, goto, return ...)

- operations: (slide 7)

* Arithmetic: Add/sub/inc/dec.
* Logic: AND/OR/XOR

* Types of Operands:

① f (7 bits): address for data memory.
② b (3 bits): bit location in file reg.
③ d (1 bit): destination bit.
④ K (8 bits): literal.
⑤ K (11 bits): address for program memory (address of instruction)
↳ goto K
call K
8-bit 0 → 255 (not enough)
so 11-bit 0 → 2047 ✓

* Assembly is case insensitive ⇒ "a" = "A"

* CLRW: clear working reg. ⇒ W-reg [0]
↳ ∅ operands.

* CLRF (f): ex: clrf 15 → 15 [00]
7 bits

* addwf f,d : ex: addwf 16,1
7 bits ↳ 1 bit
if (addwf 16,∅) → result is stored in W-Reg.

for example let:
[W=2]
16 [3 5]

* bcf f,b: ex: bcf 22,2
7 bits ↳ 3 bits
↳ it clears bit number 2 in address 22

22 [2 1 0]

* addlw K: 8 bits
ex. addlw 7
⇒ add value 7 to the value in the W-Reg. and store it back in W-Reg.

Five Apple

7-bits 1 bit

* op f,d — 6 bits opcode, 8-bits

op f, b — 4, 7, 3

op K — 6, 8

op K — 3, 11

if it was fixed length:

35 instructions ⟹ opcode = $\lceil \log_2 35 \rceil$ = 6 bits.

opcode for inst 1 = 110... } X impossible.
& opcode for inst 2 = 1101... }

* 3 bit أول PC ينظر
إذا كانت valid يأخذها
أما إذا كانت invalid ينظر الـ bit الرابع.

ADD & SUB & INC & DEC:

* Note: all Literal operations have No destination bit, The result always stored in W-Reg.

ADDWF f,d
ADDLW K
INCF f,d
DECF f,d
SUBWF f,d
SUBLW K

value in W-Reg

[f] + [W] → d=0 [W], d=1 [f]

[f] - [W] → d=0 [W], d=1 [f]
K - [W]

we always subtract the value from W-Reg

* Literal always store in W-reg.

Ex. W=2

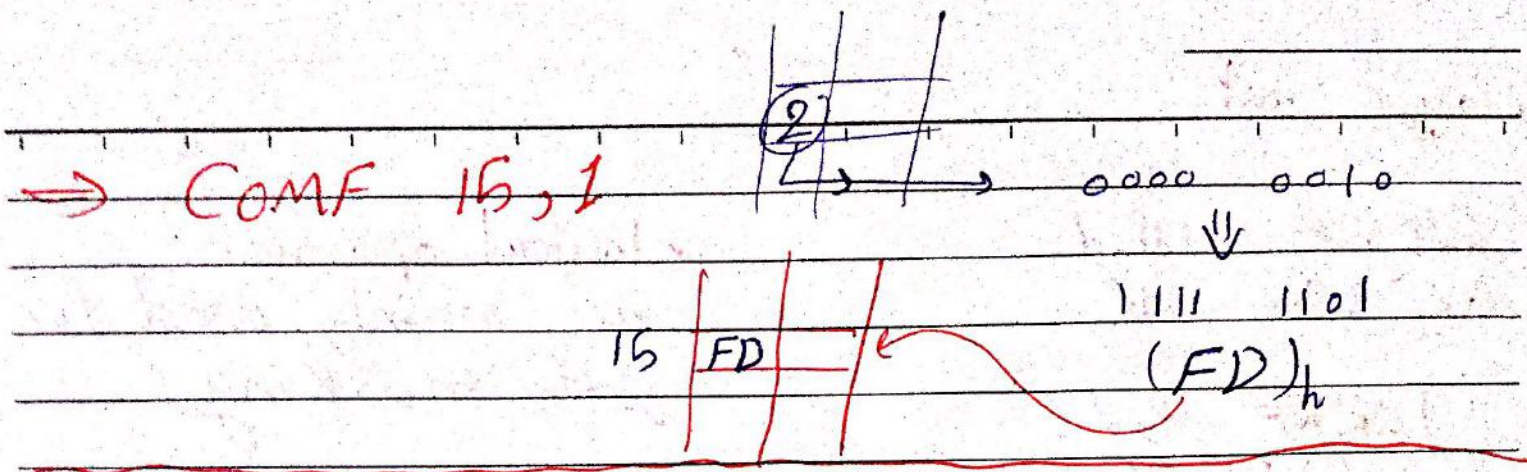
ADDLW 2 ⟹ W = 4
ADDWF 15,0 ⟹ W = 7
DECF 15,0
DECF 15,0
SUBLW 9
SUBLW 8
SUBWF 15,1

[W] = 2 → 4 → 7 → 2 → 2 → 1

[15] - [W] = [15]

15 3 2

Five Apple



A - B = A + 2's comp of B

2's comp of B = comp of B + 1

Ex: A = 2, 0000 0010

2's comp of A = 1111 1110

B = 4, 0000 0100

2's comp of B = 1111 1100

A - B = A + 2's comp of B

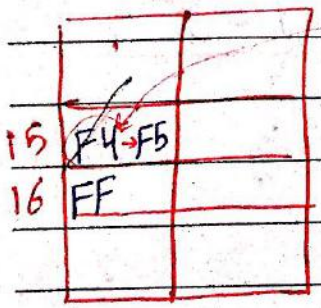
C = 0 → result is negative

0000 0010
1111 1100
1111 1110

B - A

C = 1 → positive

0000 0100
1111 1110
0000 0010



Z = 0

INCF 15, 1

INCF 16, 0 ⇒ W = 00 , Z = 1

* The status register flags are effected by the result regardless where it is stored (regardless to the value of d)

*logic inst.:

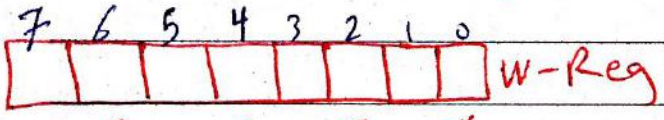
ANDWF f,d
ANDLW K

IORWF f,d
IORLW K

XORWF f,d
XORLW K

* Logical operations: are mainly used for data masking;

Example: Complement the even bits in working Reg and keep the odd bits as they are?



0 1 0 1 0 1 0 1

⇒ 55h

⇒ XORLW 55

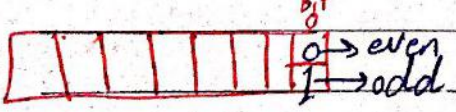
* Review:

$X \cdot 0 = 0$
$X \cdot 1 = X$
$X + 0 = X$
$X + 1 = 1$
$X \oplus 0 = X$
$X \oplus 1 = \overline{X}$

Example: Write instruction to set the most sig. 4 bits?

IORLW F0

Example: Write inst. to check if the W-Reg value is even?



ANDLW 01
check Z flag

since we just care for the first bit.

if Z=1 (even)
if Z=0 (odd)

i.e:

W = FF = 1111 0111 (odd)
0000 0001

AND | XXXX 0001 ⇒ Z = 0

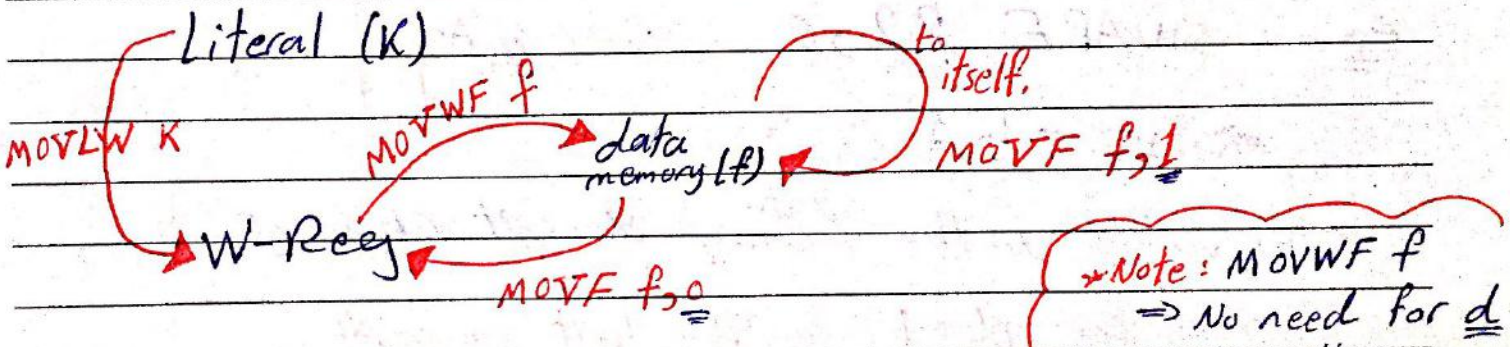
W = F6 = 1111 0110 (even)
0000 0001

AND | XXXX 0000 ⇒ Z = 1

ANDLW F0

XXXX XXXX ⇒ for this value to be > 15
⇒ one of the most sig. 4 bits should equal 1.

* Data movement inst.:



* Move instructions are Copy → Not cut.

* Note: MOVWF f
⇒ No need for d since the movement Known (W-Reg → data mem)

MOVF 15,1
check Z flag if Z=1.
⇒ [15] = 0

Example: write code to initialize data memory address 20 with the value 5 ?

MOVLW 5
MOVWF 20

Example: write code to copy address 15 to 16 ?

```
MOVF  15,0
MOVWF 16
```

** why There is No instruction such that:

MOVLF ⟹ since it take K,f:

⟹ MOVLF K,f

8 bits + 7 bits → 15 bits

Can't be for instruction.

SWAPF f,d

ex. SWAPF 22,0 given 22 A5

⟹ W = 5A

* the value in the data memory doesn't change.

↳ if the d=1 in the last example

⟹ the value in data memory will change to 5A

* Conditional branch instructions:

BTFSS, BTFSC, INCFSZ, DECFSZ

↳ goto if condition. we use them in loops if else.

* In PIC the conditional branch is skip next inst. if condition.

Examples:

INCFSZ 15,1

inst 1 → if [15] initially ≠ FF

inst 2 → in this code always will be executed.

DECFSZ 16,1

inst 1 → if [16] initially ≠ 1

inst 2 → always.

* if the condition is true ⟹ 2 T_{inst}
, otherwise ⟹ 1 T_{inst}.

BTFSC 15,2

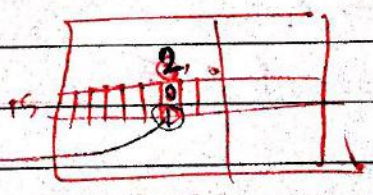
inst 1 → will be executed if bit number 2 in address 15 equal to 1.

inst 2 → always.

BTFSS 15,2

inst 1

inst 2



Call K stack ← PC
PC ← K

goto K PC ← K (No stack here)

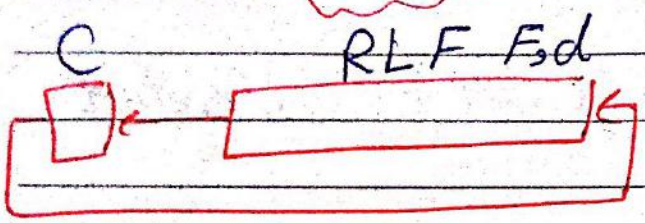
There is No return from goto.

Return

PC ← stack

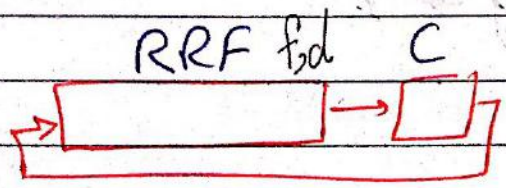
* Miscellaneous inst.:

Rotate:



*2 ; if C=0

*2+1 ; if C=1



/2

Example: write code to *8 address 15 ?

```
bcf   status, 0
RLF   15,1
bcf   status, 0
RLF   15,1
bcf   status, 0
RLF   15,1
....
```

Example: write a code to clear the least 4 sig. bits in address 15 ?

```
MOVLW  F0
ANDWF  15,0
```

Week 3

MOVF f,d

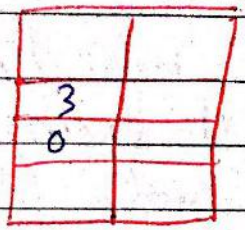
→ =0 ⇒ F → W

→ =1 ⇒ f → f itself

MOVF 15,1

MOVF 16,1

⇒ affects Z flag.



⇒ Z=1

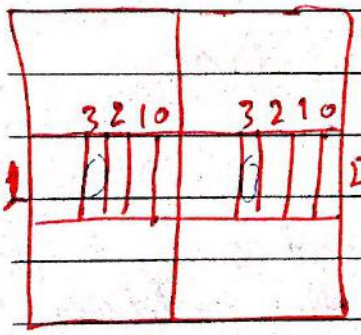
* To select bank 1:

BSF STATUS, 5

* To select bank 0:

BCF STATUS, 5

* slide (18):



means hexadecimal.

BCF 0x31, 3

bsf status, RP0 } we have to select the bank at first.

bcf 31, 3

⇒ This inst. is wrong: bcf D1, 3 (x)

movf 0x21, 0x33 (wrong instruction) → d = 0 or 1

No address → address

MOVF 21, 0 }

MOVWF 33 } correct.

movWF 0x22, 1

↳ wrong inst. since movWF doesn't take a destination bit.

for slide (21):

bsf 31, 2 ⇒ opcode = 1011

31 → 011 0001

2 → 010

1011 010 011 0001

4-bits opcode | 3-bits for b | 7-bits for f.

MOVF 21, 1

⇒ opcode | 1 | 010 | 0001

* Note: opcode will be given in the exam for each instruction.

BTFSC 22, 2 ⇒ 0110 010 010 0010

opcode

* A program in assembly language may have 4 things:

1- label: starts at the very begining of the line.

↳ Label ⇒ is a case sensitive.

* once you write a label, I can use it as operand in the program. Its value = the address of the first instruction after it

* characteristics of label:
- optional.
- can be in a stand alone line.

2- mnemonic:

ex. bcf, MOVF, ...

3- Operands: based on the instruction.

4- Comments: starts with (;)

* To specify numerals:

1- Decimal: D'22'
2- Hexadecimal: H'A2', 0x23, 23 → By default always hexa.
3- Octal: O'22'
4- AsCII: 'G', A'G'
5- Binary: B'10110111'

* If the hex value starts with letter (A-F)
⟹ Then you should precode it with either 0x or H''

MOVF A1,0 X ⟹ MOVF 0xA1,0 ✓

---

assembly ⟶ Assembler ⟶ machine code ⟶ download execution.

** Assembler: has nothing to do with execution.
⟹ the directives does not exist in the program memory.

* Directives: give information to the assembler at compilation time, Then they are discarded.

* #include: it opens a file and you can use anything inside it.

for example: to use cout in C++:

```
#include<iostream.h>,
cout << " ";
```

* **org**: it defines the address of the next instruction

```
org 5
MOVF 3,0
```
⇒ it tells the assembler that after converting the inst. "MOVF 3,0" into Binary store it in address (5) in the prog. memory.

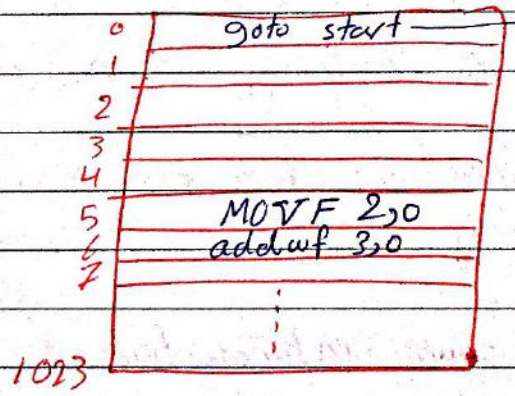
always in the code we have:

```
org 0
start of code
org 4
interrupt code
```
→ if interupt is needed.

ex.

```
    org 0
    goto start

start org 5
    MOVF 2,0
    addwf 3,0
    end
```

prog mem.



→ it will store as machine code not "goto start" as itself.

* **equ**: it defines a constant, this constant can be used anywhere in the program.

```
STATUS EQU 3
RP0    EQU 5
```
directive (takes 0 bit)

⇒ bsf STATUS,RP0 = bsf 3,5

constants

* Nothing wrong in this inst:

bsf STATUS,STATUS = bsf 3,3

takes 7bits
takes 3bits

since STATUS constant here.

#include PIC16F84A.INC

↳ if this file is included:
all registers and flags can be used by their names.

* cblock (& endc): define a block of variables.

```
cblock  0x20
    var1
    var2
    var3
endc
```

→ var1 = 20
var2 = 21
var3 = 22

same as ↳

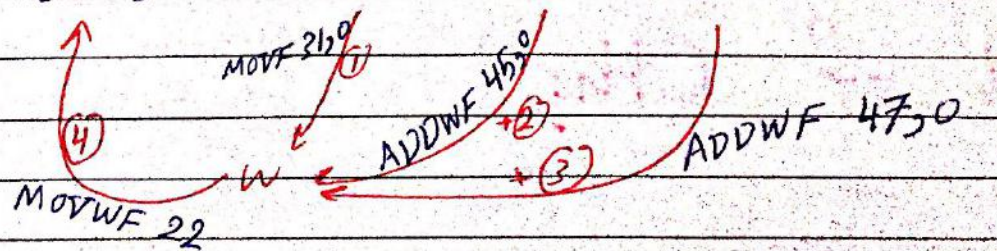
```
var1 EQU 0x20
var2 EQU 0x21
var3 EQU 0x22
end
```

→ end: tells the assembler about the end of the code.

* for writing any program: we must write in it "org 0" at first "end" when we finish.

program (slide 25):

[22] ← [31] + [45] + [47]



see the full program slide 26

→ slide 26:

DONE goto DONE ⇒ it will never reach "end" so it is infinte Loop.

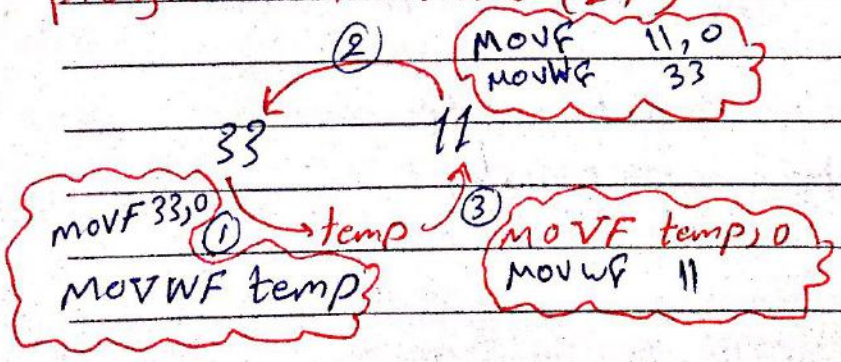
* what is the addres of START?
for the interupt ⇒ address 04 so when "goto" executed it will go to START ⇒ bcf has an address of 05 ⇒ START is a Lable so it will take the same address of bcf (05)

⇒ This prog is 8 instructions each of size 14 bits

⇒ Total size: 14 * 8 bits = 112 bits

* Without (DONE goto DONE) in the prog.:
it will take 7 instruction cycles.

program in slide (27)



End of CH4

CHAPTER(5): Building Assembler Programs.

* Conditional Branch: goto if condition.

- Conditional branch in PIC: is just skip if condition.

* If you want to write conditional if or loop,

Then you need one of the instructions: btfsc, btfss, INCFSZ, DECFSZ (without them the code is incorrect).

```
btfss  0x22,2
clrw  ------> executed if bit#2 in 0x22 = 0
clrf  0x22
```

```
if (bit 2 in 22 != 1)
    clrw
clrf 22
```

```
if (bit 2 in 22 != 1)
   clrw
else
   clrf 22
```

```
    btfss 22,2
    goto label3 ---> code else
    clrf 22     ---> code if
    goto next
Label clrw
next
    end
```

to skip else code

```
    btfss 22,2
    goto code else
    goto code if
code else
    goto next
code if
next
    end
```

=> for general way

for i=25 , i≥0

incf 31,1

```
MOVLW 25
MOVWF 22
loop  incf   31,1
      decfsz 22,1
      goto   loop
      end
```

* if it was decfsz 22,0 it will be infinite loop we will never get out of it since the value (25) in address 22 won't change.

Example: slide (8):

[11] + [22]

if C=0

[33]

else

[44]

```
      MOVF 11,0
      W
      ADDWF 22,0
```

```
      BTFSC STATUS,0
      goto else
      MOVWF 33
      goto next
else  MOVWF 44
next  end
```

Or using BTFSS:

```
BTFSS STATUS,0
goto  else
MOVWF 44
goto  next
MOVWF 33
```

using (2) goto:

```
      BTFSS STATUS,0
      goto  elsecode
      goto  ifcode
elsecode MOVWF 33
      goto next
ifcode MOVWF 44
next
```

* The code in slide (9):
has (9) instructions.

```
mov wf 0x44
goto DONE
```

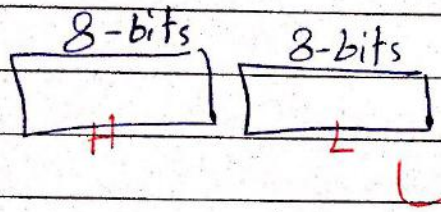
→ takes 7 bits
→ take the address
$(D)_h$ or $(13)_{10}$

8-bits $\Rightarrow 2^8 = 256$

```
clrf 22
loop Decfsz 22,1
goto loop
```

$\Rightarrow [22] = 0 \xrightarrow{①} 255 \xrightarrow{②} 254 \rightarrow \cdots \xrightarrow{256} 1$

→ 256 iterations.



* نطرح من (L) ← إذا كان صفر بنطرح من (H)
ولكن نتأكد أولاً من H صفر أو غير صفر
إذا كان يساوي صفر يمكن أن نطرح منه
إذا كان صفر ما بنطرح.

* Subroutines:
  - is just code that starts with label and ends with either "return" or "retLW K"

```
Label  code
       ...
       return
       retLW K
```

→ return
→ MOVLW K

* The subroutine is invoked by: call Label of subroutine.

* call & return retLW takes 2 Tinst

Example (slide 16):

$$[W] = [30] * [31]$$

* subroutine doesn't use parameters so we put them in a location in the data memory and call these locations.

* Generating time delays:

delay: → SW: non-useful code
→ HW: timers timer 0

```
Loop    nop      → 1 Tinst.
        nop
        goto loop
```
} max 256 iterations.

* for longer delay:

```
Loop1
        Loop2
                goto loop2
        goto loop1
```
} ⇒ max 256 * 256 iterations.

write code for → { if [22] < 10
multiply it by 4
else
add to it 5

⇒

```
        MOVLW   10
        SUBWF   22,0
        BTFSS   STATUS,C
        goto    MUL
        goto    ADD
MUL     BCF     STATUS,C
        RLF     22,1
        BCF     STATUS,C
        RLF     22,1
        goto    next
ADD     MOVLW   5
        ADDWF   22,1
next    goto    next
        end
```

MUL by 4

[22] > 10 ⇒ C = 1

[22] < 10 ⇒ C = 0

ADD 5

* To know the total delay; we usually trace the code

non useful code → the total delay = $\frac{4}{F_{osc}}$ * # T_{inst}

$\frac{4}{F_{osc}}$ → T_{inst}

T_{inst} → trace the code

single loop:

1- initialization.
2- all iterations except the last one.
3- last iteration.

for the code slide (20):

of T_{insts} = (1 + 1) initialization

+ 199 * (1 + 1 + 1 + 2)

1 → nop, 1 → nop, 1 → decfsz, 2 → goto

+ (1 + 1 + 2)

nop nop → DECFSZ

= 2 + 199 * 5 + 4

= 1001 T_{insts}

$\Rightarrow$ delay = $\frac{4}{800 \times 10^3}$ * 1001

= 5.005 ms

↳ for this code to be a subroutine:

call 2 T_{inst}
return 2 T_{inst}

(# of T_{inst} = 1001 + 4)

$\Rightarrow$ delay = 4 * $\frac{4}{800KHz}$ + 5.005 ms

or delay = $\frac{4}{800K}$ * 1005

* Modify the given code to give 4 ms delay?
↳ slide (20)

$T_{ins} = \frac{4}{800K} = 5\,\mu s$ $\Rightarrow$ delay = T_{ins} * # of T_{inst}

$\Rightarrow 4 * 10^{-3} = 5 * 10^{-6} * N \Rightarrow N = 800$

$\Rightarrow$ $800 = (1+1) + (1+1+1+2) * (X-1) + (1+1+2)$

$800 = 2 + 5X - 5 + 4$

$\Rightarrow X = \frac{799}{5} = 159.8$

No use for fraction $\Rightarrow$ $X = 159$

```
MOVLW   D'159'
MOVWF   COUNTER
del     nop
        nop
        nop
        nop
        nop
        nop
        decfsz  counter,F
        goto    del
```

delay = 796 $\Rightarrow$ N = 800

so we need 4 nop

Ex. Write a subroutine that gives 10ms delay given that $F_{osc} = 100KHz$?

$T_{inst} = \frac{4}{100KHz} = 40\mu s$ $\Rightarrow$ # of inst cycles $= \frac{10 \times 10^{-3}}{40 \times 10^{-6}} = 250$

```
Sub     MOVLW   [X]        D'49'
        MOVWF   COUNTER
Loop    nop
        nop
        decfsz  counter,1
        goto    Loop
        return
```

call initializing

$\Rightarrow 250 = 2 + 2 + 5*(X-1) + 6$

$\Rightarrow X = 49$

* for nested loop:

1) initialization.
2) first external iteration
3) all external except last one
4) Last external iteration.

→ all internal except last internal iteration.

```
        MOVLW   20
        MOVWF   counterI
        MOVLW   30
        MOVWF   counterE
loop    DECFSZ  counterI,1
        goto    loop
        DECFSZ  counterE,1
        goto    loop
```

أول مرة سيقوم بتنفيذها 20 مرة بعدها لما يرجع (loop) بسبب external بينفذها 256 مرة.

slide (21):

$$T_{inst} = \frac{4}{4MHz} = 1\mu s \Rightarrow 10 * 10^{-3} = 1 * 10^{-6} * N$$

$$\Rightarrow \boxed{N = 10\,000}$$

call initia.

$$\#\ of\ T_{inst} = 2 + 5 + 249 * 3 + (2+1+2) + 11 * [255 * 3 + 5] + 255 * 3 + 6$$

$$= 10\,000$$

counterL = 250
counterH = 13

(1) first external — (11) all external — (1) last external

* Working with Data:

- indirect address: the value to work on is the data memory and its address is in FSR register.

* We usually use indirect addressing:

To deal with lists.

* Write a code to clear data memory location:
D'10' to D'70'

```
CLRF  D'10'
CIRF  D'11'
  :
CLRF  D'70'
```

⇒ indirect addressing: ① we store the address in FSR.

② The intructions should work on INDF or 0x00

```
counter EQU 21
MOVLW D'61'
MOVWF counter
MOVLW D'10'
MOVWF FSR
loop  CLRF  INDF
      INCF  FSR
      DECFSZ counter
      goto loop
```

→ once CPU see that f= 0x00 or INDF
⇒ The CPU gets the address from the FSR

it is inc. for an address.

Ex. Write code to Complement the least significant 4 bits of address 23, using indirect addressing?
(the value in)

```
MOVLW 23
MOVWF FSR
MOVLW 0x0F
XORWF INDF,1   => XORWF 23,1
```

23 ⊕ 0F

** XORWF FSR,1
↳ we can't use it since here will comp. the 23 not the value in address 23

Ex. Write code to copy the value in address 0x81 (using direct & indirect addressing) into W-Reg

81 → 1000 0001
↳ bank 1

=> direct addressing:

```
MOVF 0x81,0   (x) => since the address with MOVF = 7 bits.
```

=>
```
BSF STATUS,RP0
MOVF 0x01,0
```

=> indirect addressing:

```
BSF FSR,7
MOVLW 0x01
MOVWF FSR
MOVF INDF,0
MOVLW 0x81
MOVWF FSR
MOVF INDF,0
```

overwrites inst. الأول

INDF	INDF

=> INDF is not a physical register. (we can't store a value in it).

Example: slide (25)

it has 12 instructions.

if we don't have indirect:

the code will be:

see the full code using: indirect addressing: (slide 25)

```
MOVF  0x10,0
ADDWF 0x11,0
ADDWF 0x12,0
  .
  .
  .
ADDWF 0x1F,0
MOVWF 0x20
```

15 add instructions.

if we asked about the delay (without inst DONE goto DONE)

$\#T_{inst} = 2 + 5 + 14*5 + 5$ (2: goto, 5: initia.)

$delay = \frac{4}{F_{osc}} * \#T_{inst}$

* Look-up Tables:

```
int a[3] = {1, 2, 3};
```

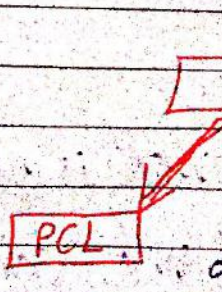
⇒ How can we define an array in PIC16F84A?

~~The look-up table is a method to define an array inside the code.~~

Look-up table: it is a subroutine.

```
⇒ lookup  ADDWF PCL,1
          retlw 0
          retlw 1
          retlw 4
          retlw 9
          retlw 16
          retlw 25
```

* PCL: it is a SFR in the two banks.



PC: not part of the mem. ⇒ Can't be read or written directly

* Note: Any instruction modifies PC takes 2 Tinst.

```
call   lookup
MOVLW 5
call   lookup

lookup  ADDWF PCL,1
        retLW 0
        retLW 1
        retLW 4
        retLW 9
        retLW 16
        retLW 25
```

→ array of 6 elements = {0, 1, 4, 9, 16, 25}

to call an element:

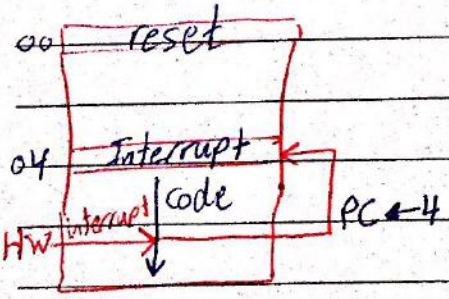
```
MOVLW  element Number
call   lookup
```

End of CH5

CHAPTER(6): Interrupts & Timers.

* Interrupts:
  - The interrupt is code written in the program memory starting at address 04, invoked by HW

Timer ∅
- we need to use interrupt.



* Interrupt code: it is a subroutine invoked by HW and has higher priority than regular code stored at address 04 in program memory, ends with RETFIE

* Interrupts Vs Polling:

⇒ Polling: Keep checking, sync.

⇒ interrupt: asynchronous.

* Interrupts → internal.
  → external.

→ maskable: can be disabled.
⇒ Non-maskable: Can't be disabled.

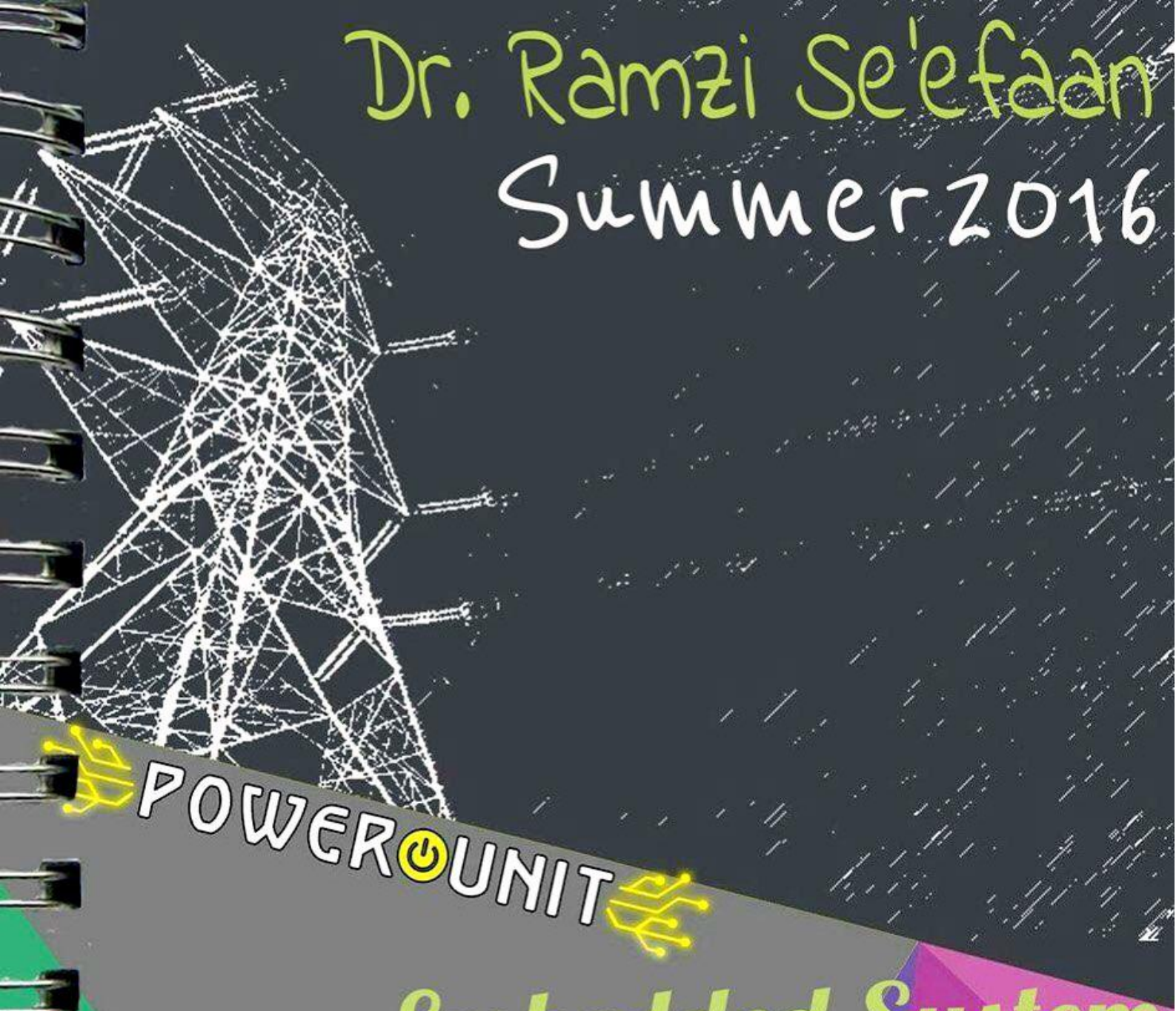
* In PIC: all interrupts can be disableed (are maskable).

* for an interrupt to happen:

1- GIE = 1 } ⇒ By SW
2- local enable = 1 }
3- flag bit = 1 (s=1) } ⇒ By HW
   → cleared on reset.

* There are 4 types of interrupt: each one has a local enable bit.
  → each of the 4 interrupts has flag bit

Week 4



Dr. Ramzi Se'efaan

Summer2016

POWERUNIT

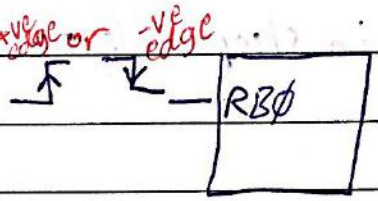
Embedded System

By: Mohammad Abuhashia

* interrupt flag: set by HW, cleared by SW.

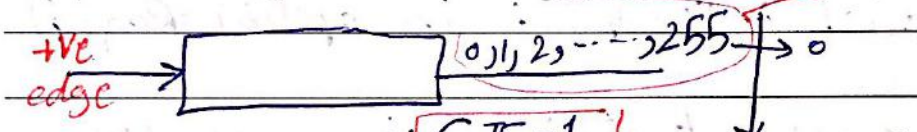
* Types of interrupt:

① External interrupt.



INTF = 1
GIE = 1
INTE = 1

② Timer0 overflow interrupt.

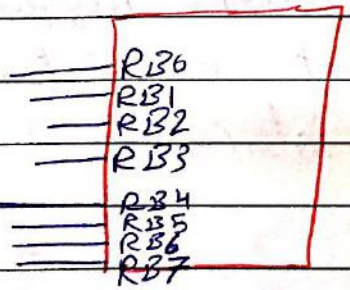


since it is 8 bits (Timer0)

GIE = 1
T0IE = 1

T0IF = 1

③ Port B change interrupt.



when any of the 4 pins RB4 → RB7 changes its value

⇒ RBIF = 1
GIE = 1
RBIE = 1

④ EEPROM write complete interrupt.

↳ after finish writing:

① WR bit ⇒ WR = 0
↳ 0

② EEIF = 1 → interrupt under conditions { GIE = 1, EEIE = 1

* If there was more than one type of interrupt enabled Then to Know the type check the flag bit.

* If local enable = 1 and its flag = 1
⟹ if the CPU is in sleep mode ⇒ it wakes up even if GIE=0 (see slide 8)

* INTCON Register:

4 flags
4 enables
GIE
} ⟹ in INTCON except EEIF is in EECON1

1 bit ↑ ↓ in external interrupt is in option-Reg(6)

≈> To deal with interrupt: we set GIE and its enable bit.
* External interrupt: requires also setting the edge bit (option(6))

in slide(11): all the steps in interrupt happens Automatically.
⟹ {clear GIE}: to prevent other interrupts from interrupting the current interrupt.

** Why the code of interrupt must end with "retfie"?

since retfie has implicitly → GIE = 1
                              → Return
But the "Return" doesn't include GIE=1 so we didn't use return.

✓ In PIC16F84A: all flags are cleared on reset except RBIF since RBIF could be related for another interrupt::

* In slide(12): Between step 4 & 5: you should clear the flag bit by SW before retfie.

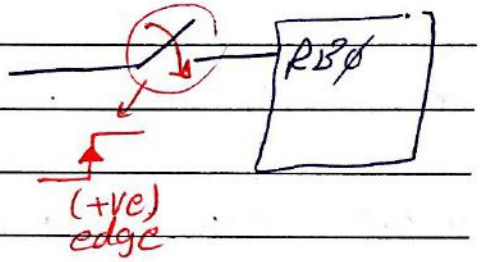
* Example (slide 13):

- 0x0A
- external interrupt (↓) ⇒ we must have in the code:
- 0x10 ← result
- [W-Reg] ← 0

GIE = 1
INTE = 1
option_Reg(6) = 1

```
BSF  INTCON, GIE
BSF  INTCON, INTE
BSF  OPTION_REG, 6
```

* (+ve) edge happen for example when we have switch:



* Context Saving:

if there is a register that you don't want to be modified by the ISR ⇒ "Context saving".

⇒ store it in the data memory at the begining of the ISR and read it before retfie.

```
ISR   movWF  Temp
      :
      MOVF   Temp, 0
      retfie
```

```
      movF   15, 0
      movWF  Temp
      :
      movF   Temp, 0
      movWF  15
```

context saving

we can't use this method for the STATUS reg. since movF affects carry flag.

(1)

* Exercise: write code that multiplies location 0x15 by 4 when external interrupt happens. and decrement by 4 location 0x15 when PORTB change interrup

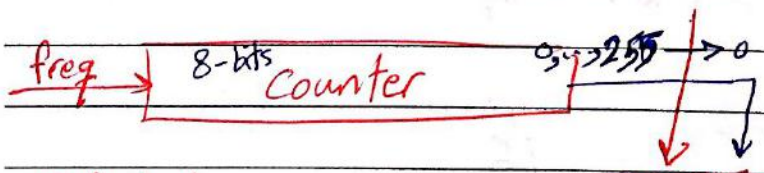
(2) When any of PORTB change or Timer0 overflow interrupts happens ⟹ clear W
However, if PORTB change interrupt happens ⟹ Increment W-Reg

---

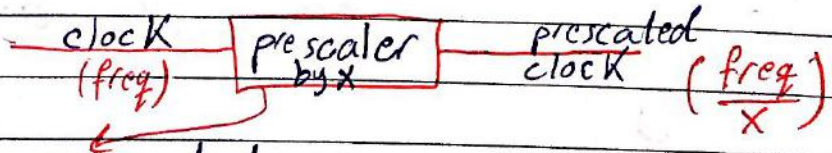
* Counters & Timers:

* For a counter to work as timer:
1) we should know the freq of the clock.
e.g. if freq = 1MHz ⟹ $T = \frac{1}{1MHz} = 1\mu s$
⟹ TOIF = 1 after 256μs (Timer0 (8bits))

2) we should be abled to initialize it.
e.g. to make the timer0 overflows after 100μs if freq=1MHz ⟹ we initialize it by 256-100 = 156



slide (19):
* The clock input to timer0:
if PSA = 0 ⟹ clock is prescaled.
if PSA = 1 ⟹ clock is Not prescaled.



* To increase the delay.

* The prescale value = $2^{PS2\ PS1\ PS0\ +1}$

Ex: if PSA = 0 and PS2 PS1 PS0 = 101 ⇒ prescale = $2^{101+1} = 2^6 = 64$

if PSA = 1 and PS2 PS1 PS0 = 100 ⇒ PSA=1 not prescaled.

* The clock source can be: either internal with $freq = \frac{Fosc}{4}$ or external from RA4 Pin

⇒ it is → internal: when TOCS = 0
→ external: when TOCS = 1

TOSE = 1 ⇒ if the clock is external ⇒ it counts on negative edge.

* To deal with timer0, we manipulate:
01 ← TMR0 register & ⇒ b0
81 ← OPTION_REG ⇒ b1

* delay of Timer0 in case we use internal clock $(\frac{Fosc}{4})$
$= \left(\frac{4}{Fosc} * \text{prescale}\right) * (256 - \text{Initial value})$
cycle of the clock entered to timer0

* Sync: it is used when the clock is external to synchronize.
... takes up to 2 cycles.
⇒ it is always ignored in calculations.

Example: write code to give 10ms delay using timer0 given that $F_{osc} = 1MHz$.

$delay = \frac{4}{F_{osc}} * prescale * (256 - N)$ → one equ. with two variables (we solve it by assuming a value for prescale then finding N)

$10 * 10^{-3} = \frac{4}{1M} * pres * \underbrace{(256 - N)}_{X}$

assume prescale = 2

$pres * X = 2500 \Rightarrow X = \frac{2500}{2} = 1250$ (impossible)

assume prescale = 16 $\Rightarrow X = \frac{2500}{16} = 156.25 \ (\approx 156)$

$\Rightarrow N = 256 - 156 \Rightarrow \boxed{N = 100}$

```
MOVLW   D'100'
MOVWF   TMR0
BSF     STATUS,RP0
MOVLW   B'00000011'   → from 16 pre scale (see slide 20)
MOVWF   OPTION_REG    → PSA
```

* Maximum delay of Timer0 $= \frac{4}{F_{osc}} * 256 * 256$

(256 → max prescale; 256 → max of (256 − N); 0 ↑)

Example: given $F_{osc} = 1MHz$, write code to give 1 second delay?

Timer0 can be asynchronous.

~~1 second = counter~~

Example → slide (22)

$F_{osc} = 800KHz \Rightarrow 5*10^{-3} = \frac{4}{800*10^3} * pres * \underbrace{(256-N)}_{X}$

$pres * X = 1000$

$pres = 8 \Rightarrow X = 125 \Rightarrow N = 256 - 126 \Rightarrow \boxed{N = 131}$ TMR0

option = 0000 0010
↓ internal clock

if we want to deal with interrupt in this example:

```
        org 0x0000
        goto start
        org 0x0004
        goto ISR
start   BSF INTCON, GIE
        BSF INTCON, T0IE
        MOVLW D'131'
        MOVWF TMR0
        BSF STATUS, RP0
        MOVLW B'0000 0010'
        MOVWF OPTION_REG
        BCF STATUS, RP0
loop    goto loop
ISR     MOVLW D'131'   } → without them Timer0 start counting from zero
        MOVWF TMR0     }
        BCF INTCON, T0IF
        retfie
```

(in this code interrupt will happen every 5ms)

1 second = counter * 5ms

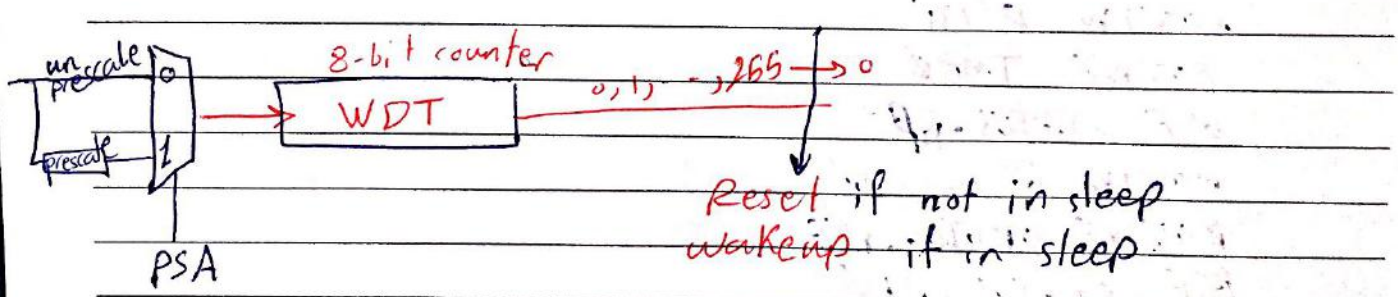
$counter = \frac{1}{5m} = \boxed{200}$ ⟹ The New code

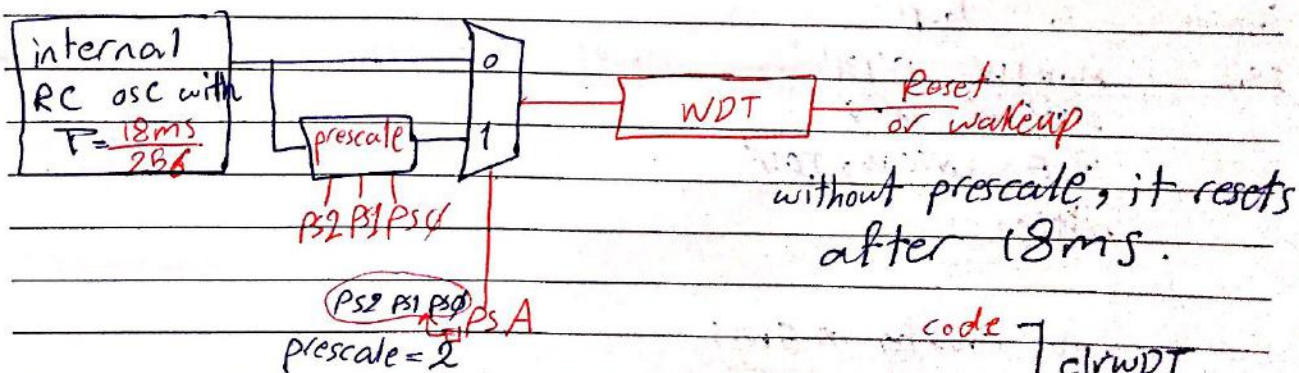
```
        org  0x0000
        goto start
        org  0x0004
        goto ISR

start   BSF   INTCON,GIE
        BSF   INTCON,T0IE
        MOVLW D'200'
        MOVWF counter
        MOVLW D'131'
        MOVWF TMR0
        BSF   STATUS,RP0
        MOVLW B'0000 0010'
        MOVWF OPTION_Reg
        BCF   STATUS,RP0
loop    goto  loop
```

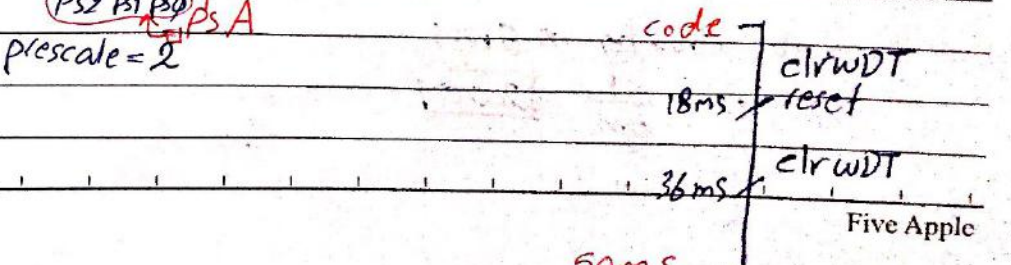
```
ISR     DECFSZ counter,1
        goto  label
        goto  done
label   MOVLW D'131'
        MOVWF TMR0
        BCF   INTCON,T0IF
        retfie
done    BSF   PORTB,2
        MOVLW D'200'
        MOVWF counter
        MOVLW D'131'
        MOVWF TMR0
        BCF   INTCON,T0IF
        retfie
```

* Watchdog Timer:

WDT timer 8 bits







Five Apple

⇒ maximum WDT after which if resets Then:

M.C is 18 ms * max prescale

18 ms * 128 = 2.3 seconds

INDTE in configration word = 1 ⇒ it is enabled.

* Sleep Mode:

↳ To save power to minimum.

WDT: Keeps counting in sleep mode because it has its own oscillator.

sleep
{ code ⇒ will not be executed except after wakeup..

* To enter sleep mode: put instruction:
sleep

* The M.C wakes up in the following cases:

1- local interrupt is enabled and its corresponding flag became=1
even if GIE = 0
↳ all interrupts may wake up the M.C
except Timer0 interrupt
⇒ because the OSC is off.

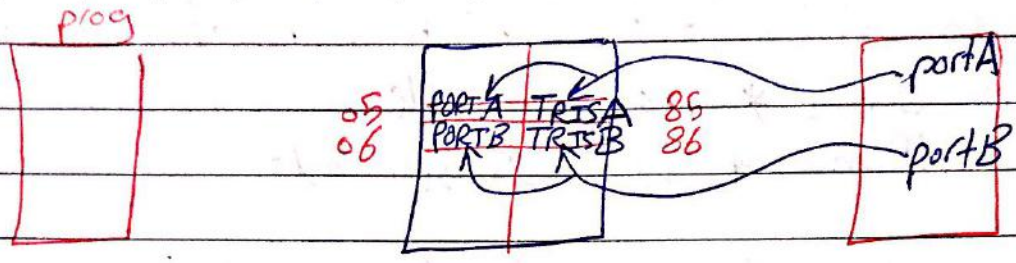
2- WDT overflows

3- $\overline{MCLR}$ = 0

WDTE=1
{ sleep inst. ⇒ max. sleep time = max WDT. delay
= 2.3 seconds.

End of CH6

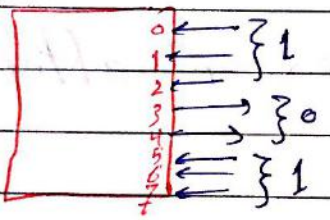
CHAPTER(3): Parallel Ports.



memory mapped I/O

* TRIS Reg: specify each of the pins it is input or output. pin → 0 ⇒ output.
  → 1 ⇒ input.

*Example slide (7):



111 00 111

```
BSF   STATUS,RP0
MOVLW B'11100111'
MOVWF TRISB
```

it affects only the output pins

```
BCF   STATUS,RP0
CLRF  PORTB
MOVF  PORTB,0
```

* The inputs take their values from the external devices.

*Example(1) slide (8):

```
bsf   STATUS,RP0
MOVLW 0x00
MOVWF TRISB
BCF   STATUS,RP0
MOVLW 0xAA
MOVWF PORTB
```

clrf TRISB

* Example (2) slide(8):

→ b1
```
MOVLW 0x1F
MOVWF TRISA
```
→ b0
```
MOVF  PORTA,0
MOVWF 0x0D
```

Ex. slide (9):

for external interrupt & PORTB change interrupt to happen the pins should be input.

RB0
RB1

GIE = 1
INTE = 1
TRISB(0) = 1
OPTION_REG(6) = 1
TRISB(1) = 0

⇒

```
loop  bsf  PORTB,1
      call delay 0.5sec
      call delay 0.5sec
      bcf  PORTB,1
      call delay 0.5sec
      goto loop
```

$F_{osc} = 4\,MHz$

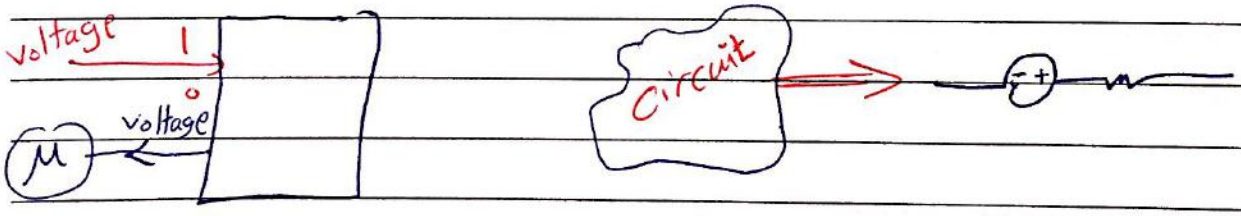
Flash = [ ][0] → not flashing
1 → flashing

⇒ ISR XOR 1

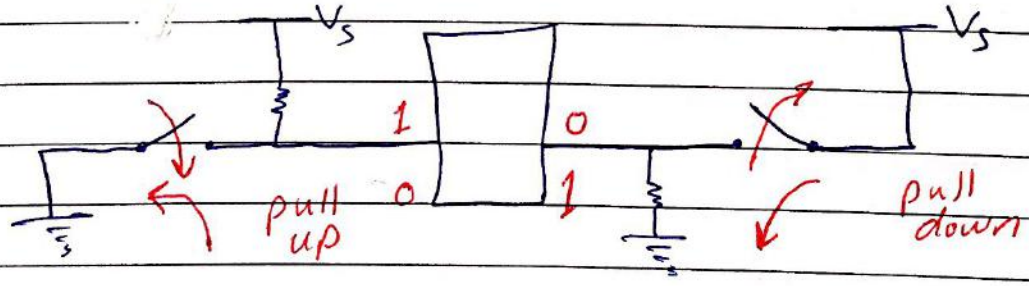
see full program slide (10) & (11):

slide (12 → 17) غير مطلوب

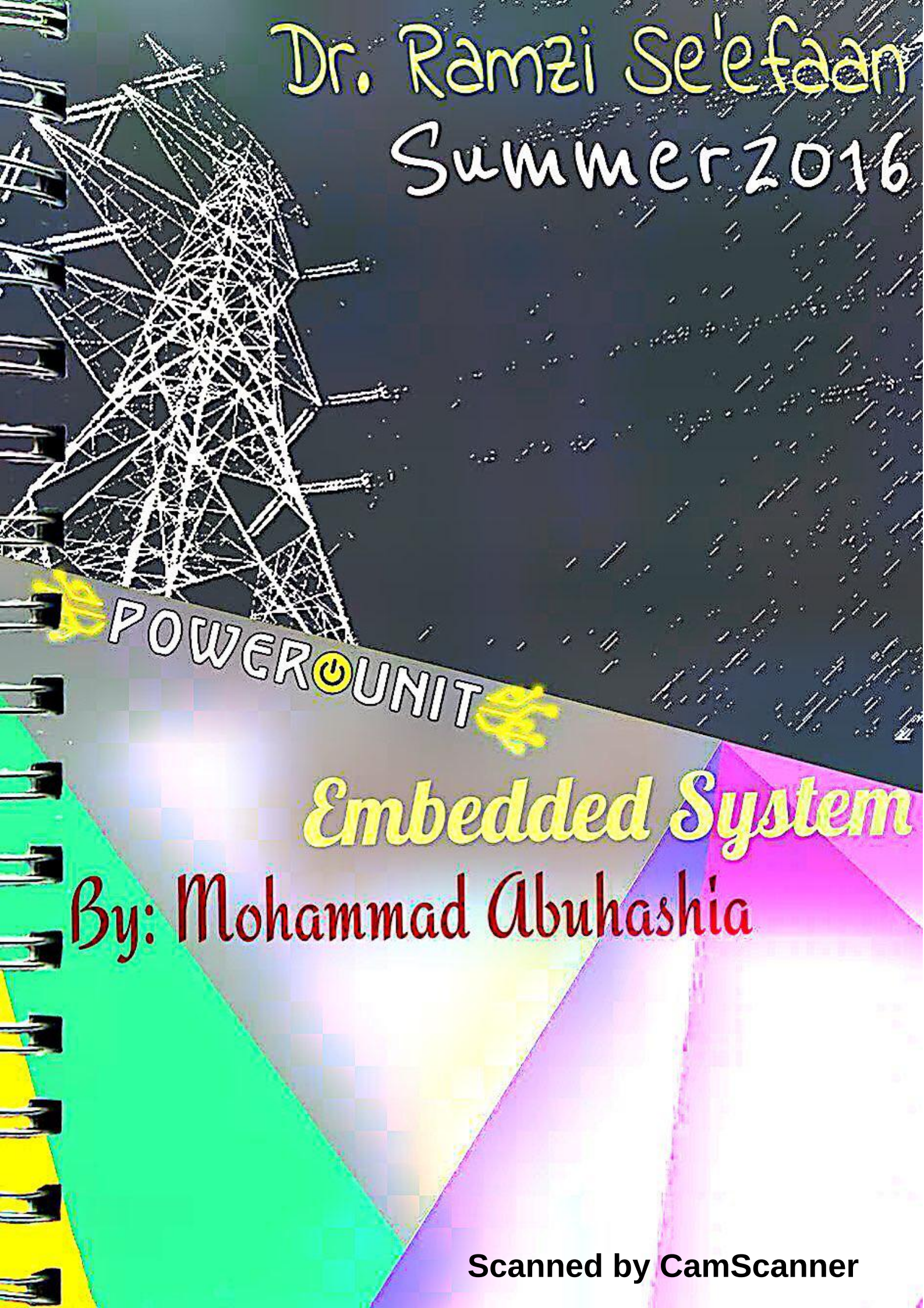
* Electrical characteristics:



* switches:
slide (19):

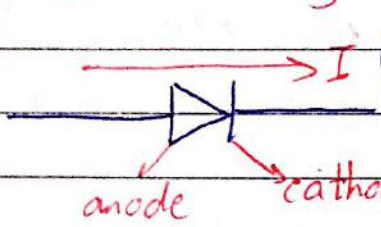


Week 5



* The pins in the parrallel are: half duplex (can send and recieve but not in the same time).

* Light Emitting Diodes (LEDs):



* Diode can be connected in either:

1- Current source: to turn on the LED ⇒ we output 1

2- Current sink: to turn on the LED ⇒ we output 0

(see slide 21)

* Current source: $V_{OH} = V_D + R * I_D \Rightarrow R = \frac{V_{OH} - V_D}{I_D}$

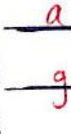
* current sink: $V_S = V_D + V_{OL} + R * I_D \Rightarrow R = \frac{V_S - V_D - V_{OL}}{I_D}$

(see slide 25)

* current source: $V_{OH} = V_D + R_{OH} * I_D + R * I_D \Rightarrow R = \frac{V_{OH} - V_D - R_{OH} * I_D}{I_D}$

and do the same for (current sink)

* 7-segment Display:

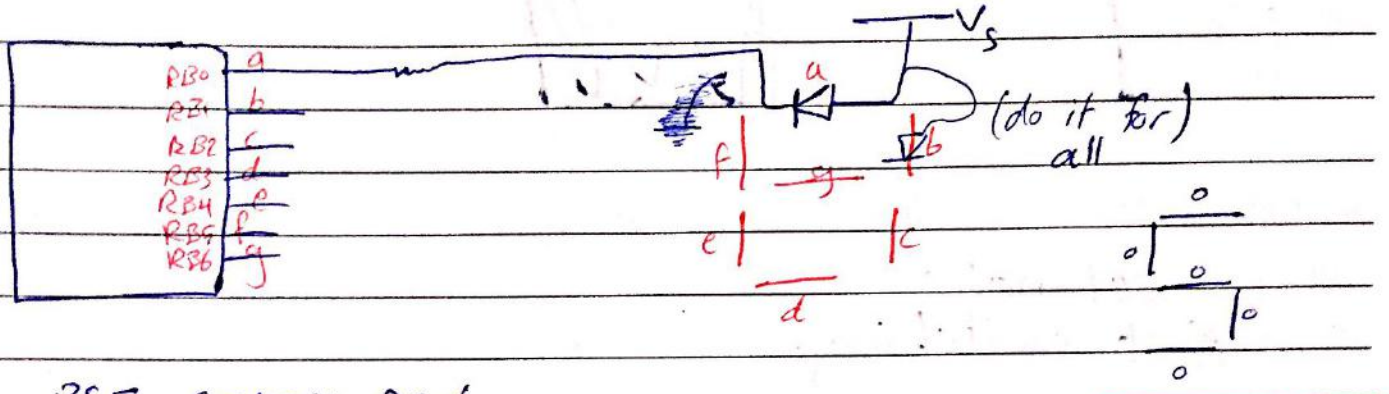


7-segment display ⇒ 7 LEDs

common cathode: all cathode are connected to ground, and turn on any seg we connect it to high (1)

common anode: all connected to V_{cc}, to turn on any seg we connect it to Low (0).

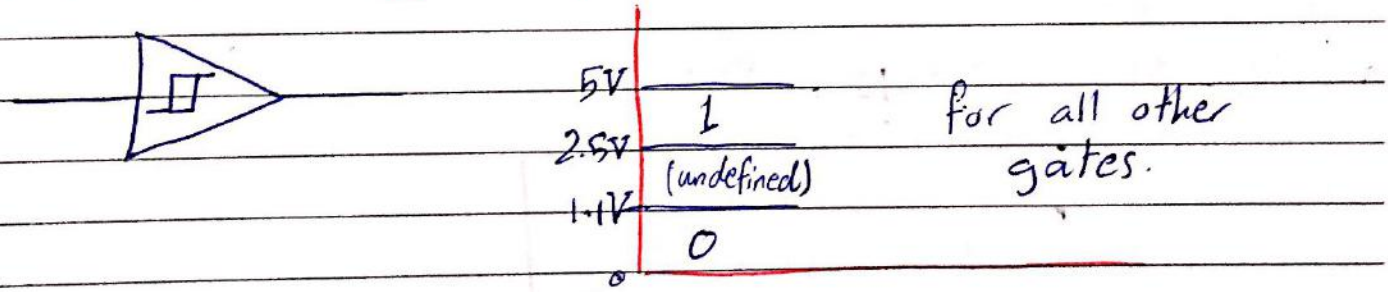
Ex. Connect the 7 SD to the PIC and write code to display number 5 on the 7 SD, assume the 7 SD is in common anode.




```
BSF   STATUS, RP0
CLRF  TRISB      ⟶ To make portB output
BCF   STATUS, RP0
MOVLW B'0001 0010'
MOVWF PORTB
```

⇒ X001 0010

* Schmitt Trigger Input:



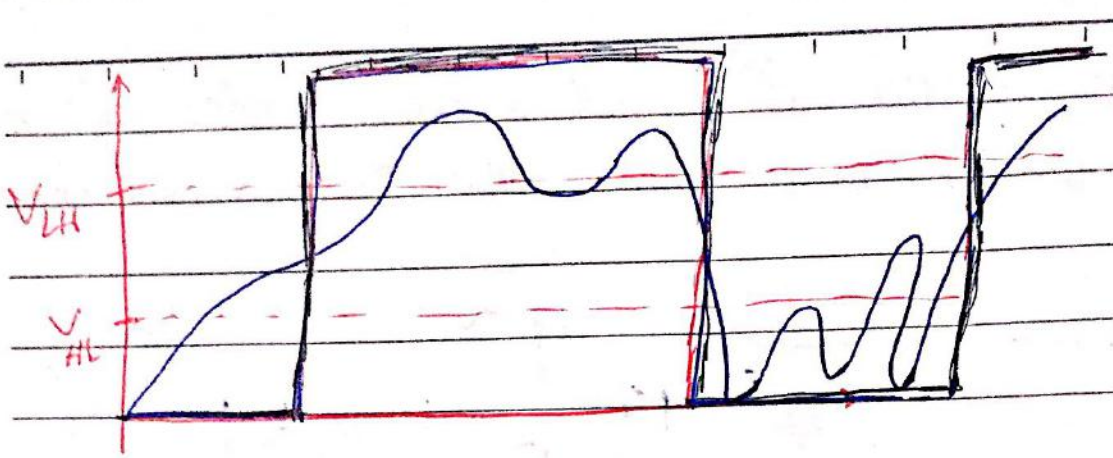
for all other gates.

* In schmitt trigger:

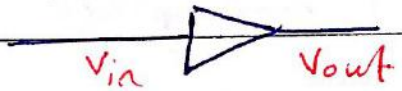
there are 2 threshold voltages $V_{HL} < V_{LH}$

V_{LH} → if the input was zero, to move into high ⇒ the voltage should exceed this threshold.

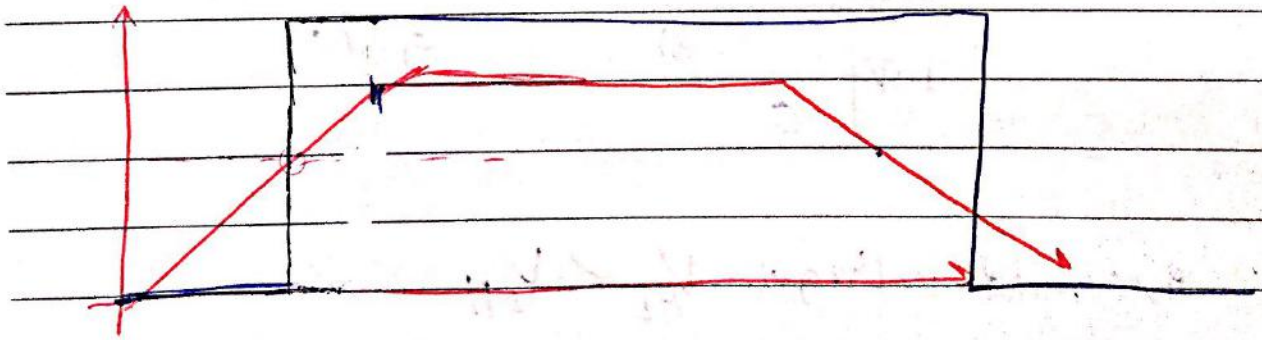
V_{HL} → if the input is 1, to move into low ⇒ the voltage should decrease below this value.

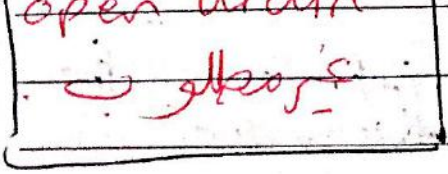


* advantages of schmitt trigger:
1- cancel noise.
2- fast switching.

* if we have: 

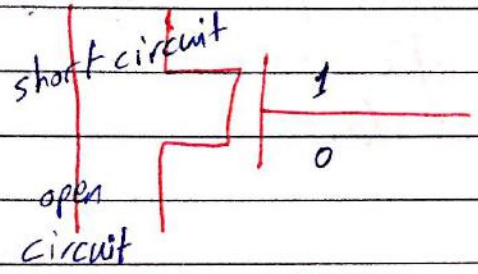
for schmitt: 



* open drain


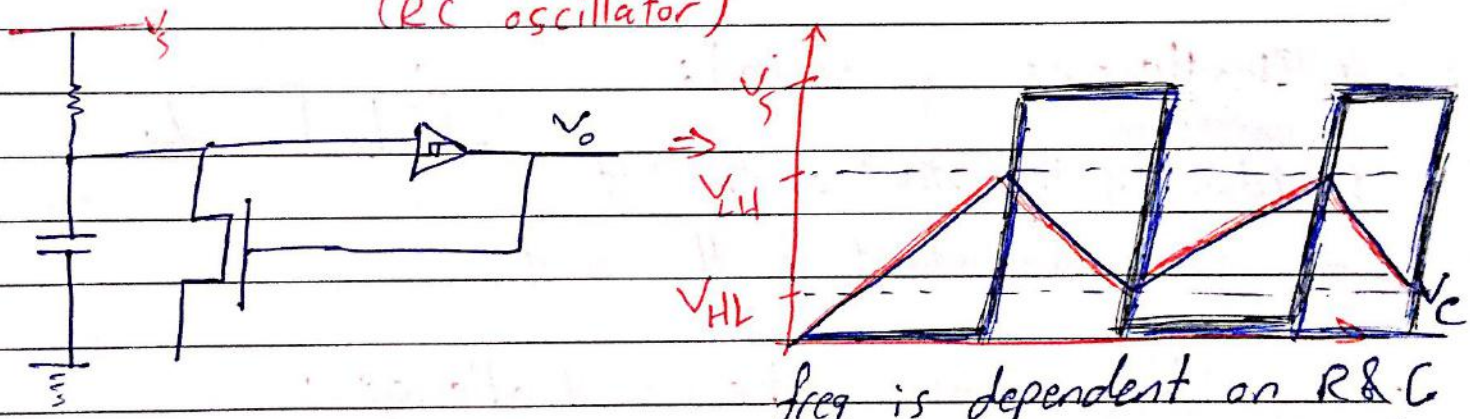
* The Oscillator: → RC oscillator
→ cristal oscillato

OSC2	OSC1
RC	
HS }	⇒ cristal, each one has freq range.
LP }	
XT }	



* The capacitors: used to stabelized and cancel noise

(RC oscillator)



freq is dependent on R & C values However, it is not very accurate.

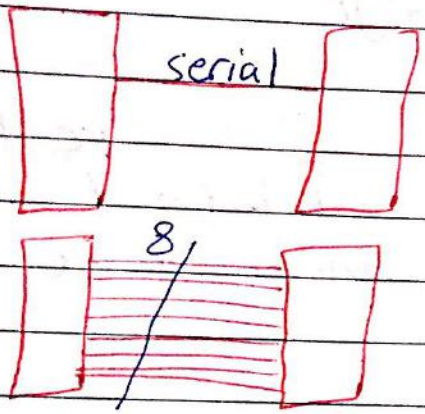
slide (35):
- the type of osc is RC (since osc2 not connected)

End of CH3

CHAPTER(10): Serial Communication.

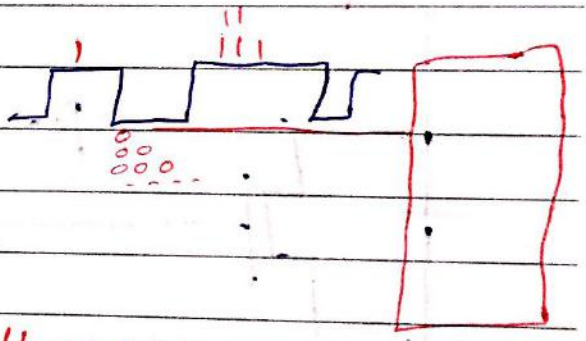
- we will study: PIC 16F87XA

16F84A ⟺ 16F87XA
we will use the one from the same family.



parallel is faster (less 8 times)
⟹ due to cross line interference and also it's not suitable for long distance.

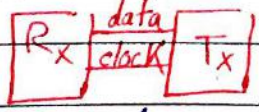
* Challenges in serial:



1- identify the start of the bit.
2- " the start of the word.

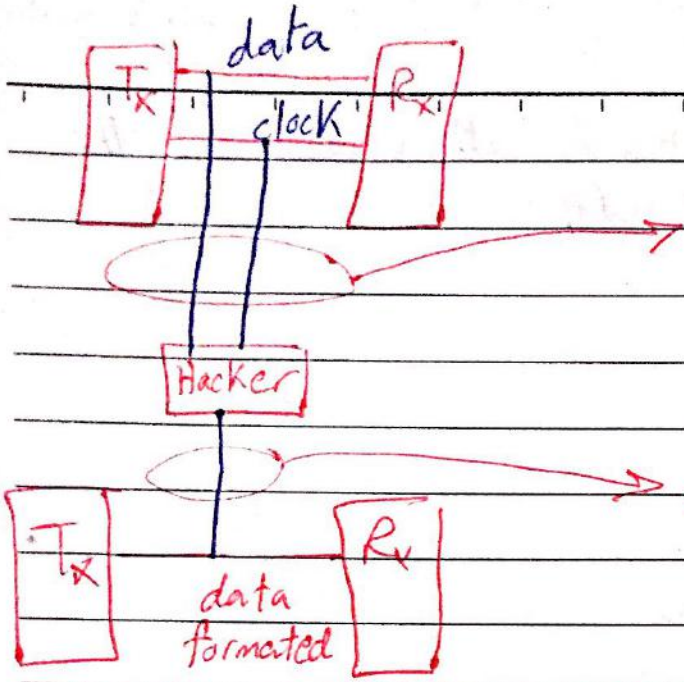
⟹ * To overcome these challenges:

1- **Synch**: there is a dedicated line sent for clock between sender & receiver in addition to the data line.



2- **Asynch**: the sender and receiver are operated by separable clocks, but same freq.
⟶ (There is a protocol)

* protocol: start bit(s), stop bit(s), data rate, encoding, parity or not, ...



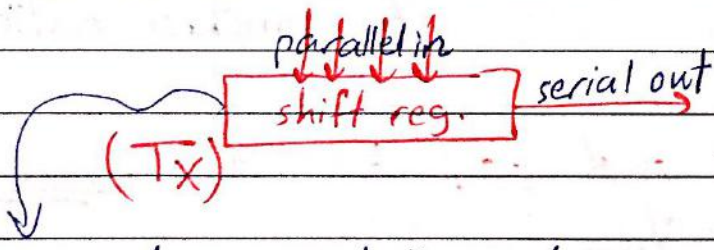
* The hacker can retreive the data without any extra information.

* The hacker requires extra information to understand the data (need to Know the protocol.

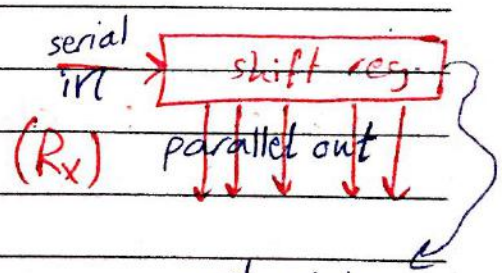
* Serial port is a shift register.

slide (7):
in the figure:

if we can load the 4 D-ffs in parallel, they need 4 cycles to be sent.



we need a way tells end of Tx

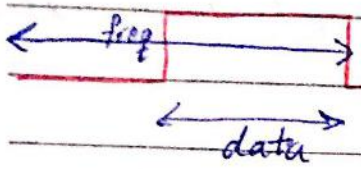


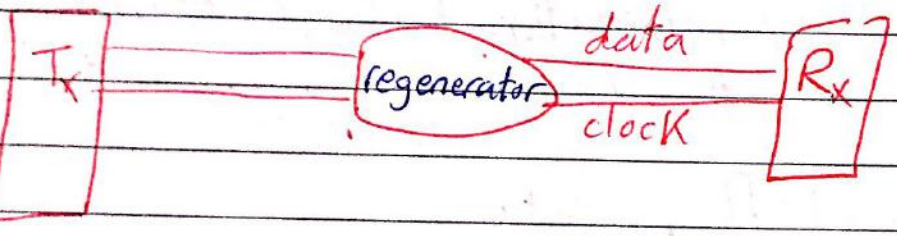
mechanisim tells Byte reception.

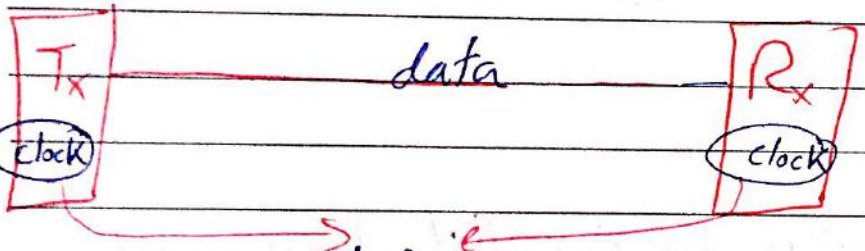
* In synchronous: if the clock is on, one of the nodes is called master and the other is slave.

* Synch is faster than asynch, Because in asynch, additional bits are sent with each bytes.

in slide (9): disadvantage: The bandwidth needed of the clock is twice the data bandwidth.





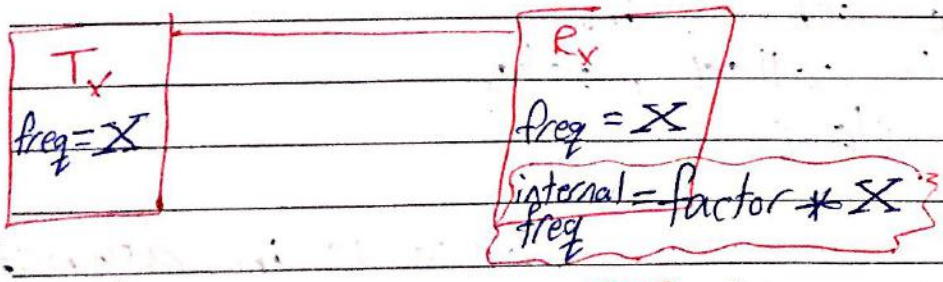


always there is "clock skew" ⟹ To deal with this: we use start and stop bits for synchronization.

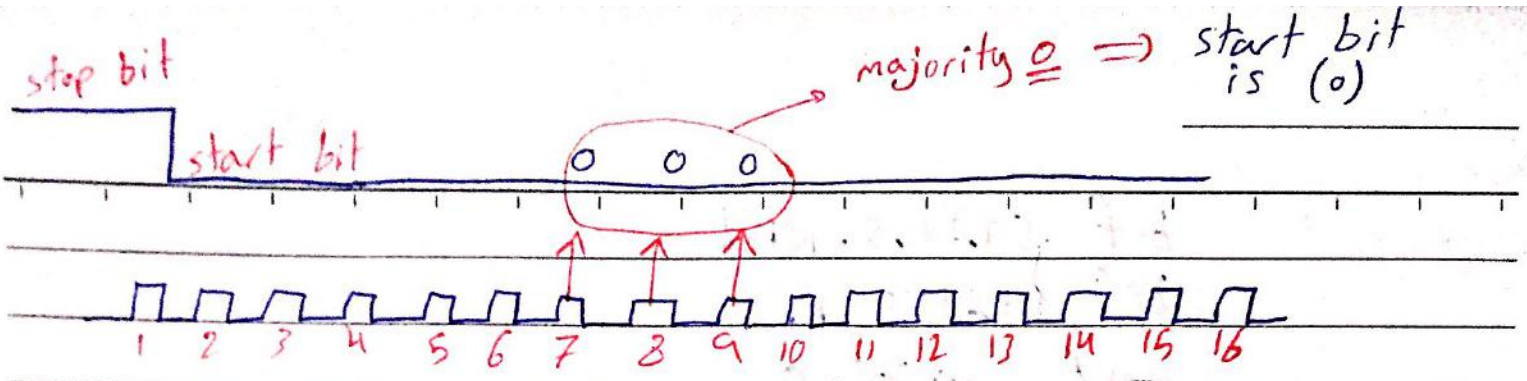
* always: start and stop bit are inverse to each other.

* دائماً عند القراءة من data نأخذ 3 samples وبدأ القراءة من منتصف الـ data وليس من الـ edge.

⟹ we take the majority for the 3 samples.



in PIC the receiver has internal freq = 16X



maj (1 0 1) = 1 (noise)
↳ it will back to the stop bit.

* physical limitations (Not included):

slide(17): overview of the PIC 16 series:

more peripherals ⇒ more pins
⇒ more SFRs

In data memory: Now we have 4 data memory banks.

⇒ we need: 2 bits to select the bank.

→ direct: RP1 & RP0 (STATUS(6) & STATUS(5))
ex 1 0 ⇒ bank 2

→ indirect:
IRP & FSR(7)
(STATUS(7)) & (FSR(7))

* Ex. write code to copy data memory location 0x175 into W-Reg using ① Direct addressing & ② Indirect addressing.

PIC 16F87XA ⇒ This arch. needs 9 bits for the data.

175 ⇒ (1) 0111 0101
↳ b2

* direct:
```
bsf STATUS, RP1
bcf STATUS, RP0
movF 0x75, 0
```

* indirect:

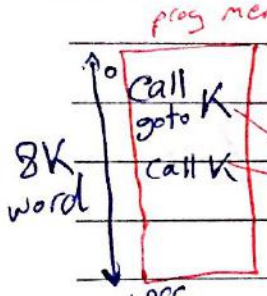
```
bsf STATUS, IRP
BCF FSR, 7
movlw 0x75
movwf FSR
movf INDF, 0
```

if it was 0x185 => 1 1000 0101

better not to write it

```
BSF STATUS, IRP
bsf FSR, 7
movlw 85
movwf FSR
movf INDF, 0
```

overwrite



minimum address = $\lceil \log_2 8K \rceil$ = 13 bits

call 1855

call 1 1 000 0101 0101 => call 055

Not the needed one.

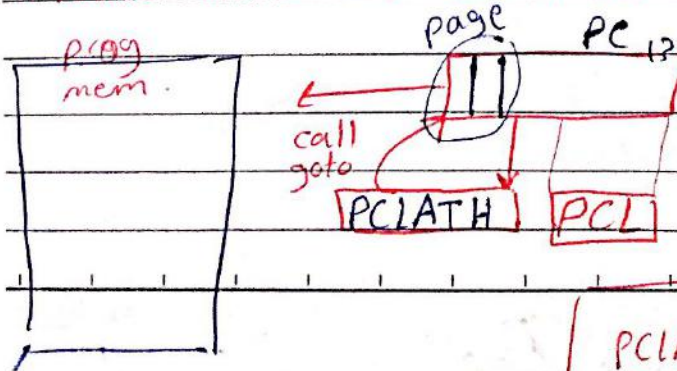
=> To solve this:

The memory is divided into 4 pages => each page is of size 2K word To move between pages:

1- select page

2- goto the address in the page.

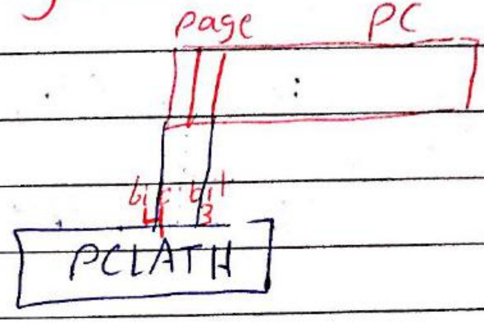
* Now how we select the page:



* bits 3 & 4 in PCLATH represent the page.

PCLATH: it is the last 5 bits in PC

Ex. suppose, the PC=5 and you want to call subroutine at address 0x1955 ⇒ page 3




```
bsf   PCLATH,4      ⇐ (11)
bsf   PCLATH,3
call  0x155
```

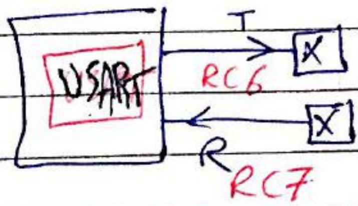
1955 ⇒ 1 1001 0101 0101

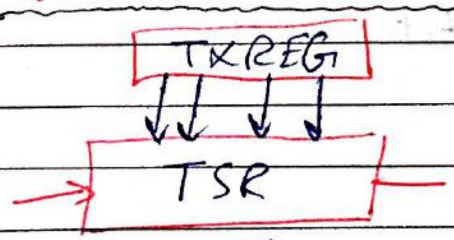
slide (19): Asynchronous Transmitter:

PEIE: peripheral interrupt enable.

* suppose you wrote a code on PIC 16F84A that deals with EEPROM write complete interrupt and you want to run the code on PIC 16F87XA
I have to enable PEIE for the interrupt to work.

* we Know that PIC is half duplex
but it work like full-duplex (how?)





see slide (22)

* TSR is not accessible by SW.
* TXREG is accessible by SW.
* if TSR is empty:
⇒ TXREG value moves into TSR.

* تصبح قيمة TXIF واحد :
إذا انتقل بايت من TSR ← TXREG

TXIF = → 1 (if TXREG is empty)
→ 0 (if TXREG is full)

(TXIF): is set/cleared by HW in PIR1.

(BCF PIR1, TXIF) X (set/cleared by HW) not SW

* write code to clear TXIF?:

MOVWF TXREG (putting any value in the TXREG will make TXIF=0.)

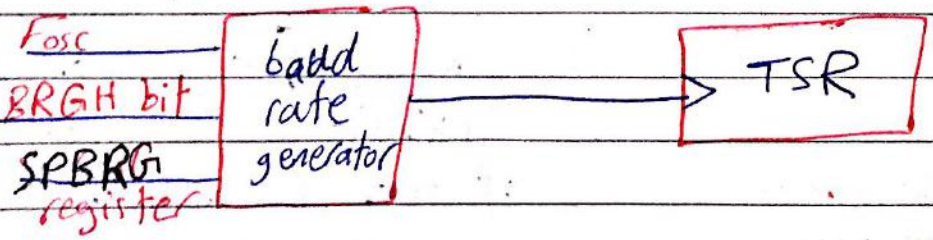
* There is two enables for Tx:

1- TXEN = 1
2- SPEN = 1 for Tx & Rx

* Two flags for Tx:

1- TXIF=1 when TXREG moves to TSR and may cause interrupt if GIE=1, PEIE=1, TXIF=1
2- TRMT when transmission is over.

* you can send 8bits words if Tx9 = 0. or bit #8
9 bits words if Tx9 = 1, The 9th bit should be written by the programmer in TX9D bit.



* we just care fo TX9D incase: Tx9 = 1

slide(27): Asynchronous Receiver:

* Two enables:
  1-SPEN.
  2-CREN.

* Two flags for RX:

1 - **RCIF=1** when stop bit of first byte is received and the byte moves to RCREG.
interrupt if: GIE=1, PEIE=1, RCIF=1

* RCREG: is 2 levels queue (opposite to stack)
  => first in - first out.
* if the stop bit of the third byte was received before reading the previous 2 bytes
  OERR=1 ⟹ 1) reception stops
  2) The third byte will be lost.

* RCIF: is read only
  - → =1 if one byte or 2 RCREG has
  - → =0 if RCREG is empty.

2 - **OERR**

* To clear RCIF: BCF PIR1, RCIF (X)

⟹
```
MOVF RCREG, 0
MOVF RCREG, 0
```

* FERR: → =1 if stop bit received wrong.
  stop bit should =1 if the received stop bit=0
  ⇒ FERR=1

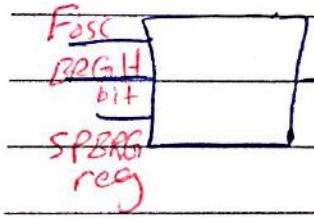
* if OERR = 1 ⇒ reception is stopped.
OERR is read only;

* To clear OERR ⇒ we clear CREN ⇒ OERR = 0
then re-enable CREN.

* Reception 8 bits per word if RX9 = 0
or 9 bits per word if RX9 = 1 and
the 9th bit is in RX9D

* PIE1 has interrupt enables.
* PIR1 has interrupt flags.

* slide(32): The Baud Rate Generator:



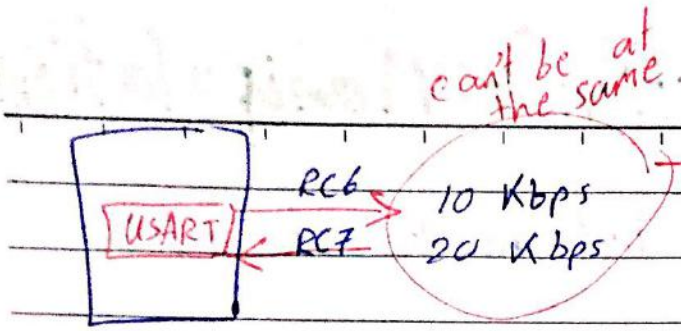
asynch.

$$\text{baud rate} = \frac{Fosc}{\frac{16}{64} * (1 + SPBRG)}$$

if BRGH = 1 ; BRGH = 0

* serial Tx: TXREG (data)
TXSTA (control)
PIR1, PIE1, INTCON } if interrupt
TRISC(6) } output
SPEN from RCSTA
SPBRG

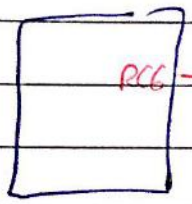
* serial Rx:
RCREG (data)
RCSTA (control)
PIR1, PIE1, INTCON } if interrupt
TRISC(7) } input
SYNC, BRGH } from TXSTD
SPBRG



→ Can't be / because you have only one { SPBRG reg / BRGH bit }

Ex slide (33):

40
41
42



no parity (8 bits)
⇒ TX9 = 0

baud rate = 9.6 Kbps
F_{osc} = 20 MHz

$$\Rightarrow \text{baud rate} = \frac{F_{osc}}{\frac{16}{64} * (1 + SPBRG)} = \frac{20 * 10^6}{16 * (1+SP)} = 9.6 * 10^3$$

using BRGH=1
⇒ SPBRG = 129.2 it is suitable since it is < 256
* we could use BRGH=0 ⇒ SPBRG = 31

TXSTA = X 0 0 1 0 0 1 0 0 X ... ⇔ TXREG = (40)

PEIE in INTCON
GIE in INTCON
TXIE in PIE1
↳ if we want to use interrupt

SPBRG = 129
TRISC (6) = 0
RCSTA (7) = 1

see the code slide (84)

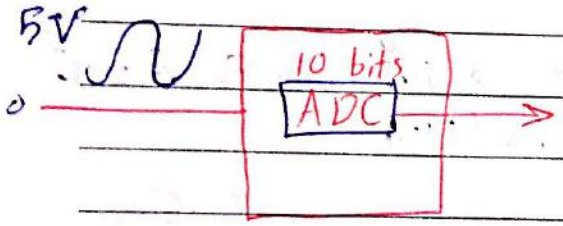
End of CH10

CHAPTER(II): Data Acquisition & Manipulation.

* Analog: the quantity takes unlimited number of values in specific range.

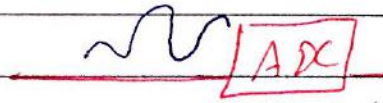
* Digital: the quantity takes limited number of values in specific range.

* The difference between:
- continuous in time: could change at any moment.
- discrete in time: could change at specific moment.

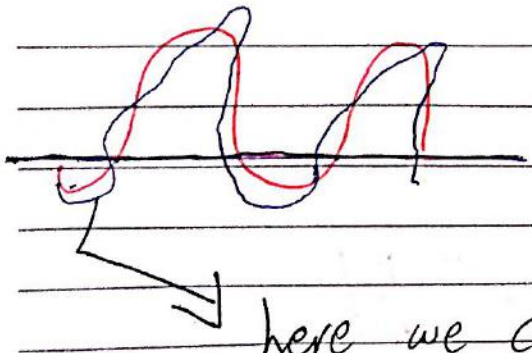


each analog value will be converted into 10 bits digital value
⇒ "regardless of the input range"

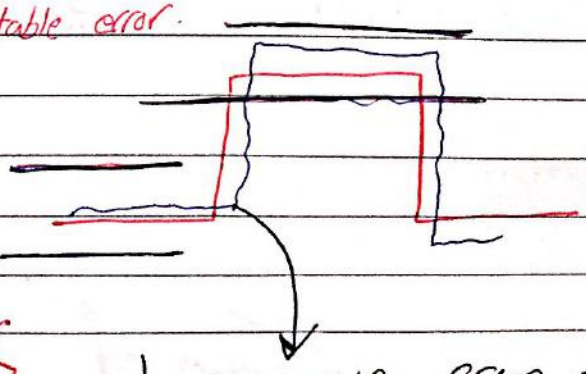
** Analog more "accurate" than digital:
because it take unlimited values.



acceptable error.



here we can't make the blue curve like the red since it is out of the range.



here we can make the blue curve like the red since it is included by the range.

ADC has 2 main steps:

① **sampling**: read the analog input.

- sampling rate: every how long we read a sample.

* more sampling rate ⟹ more accurate.

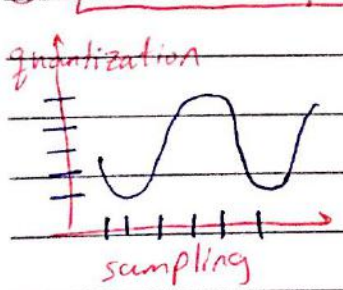
⟹ To go into analog ⟹ we need infinite sampling rate.

* disadvantage: more sampling ⟹ * more size → storage. → bandwidth.
* Cost.

* Minimum sampling rate that we could use: = $2 * f_{max}$

$f \leq 50KHz$ | ADC 100KHz sampling rate

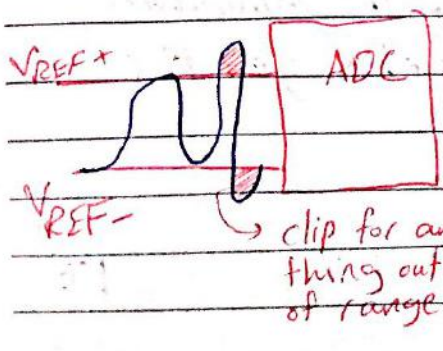
② **Quantization**: round to the closest acceptable digital value.



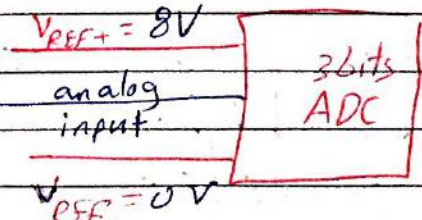
* more quantization levels ⟹
more accurate
more storage
more bandwidth.
more cost
slower.

slide (6): Conversion Characteristics:

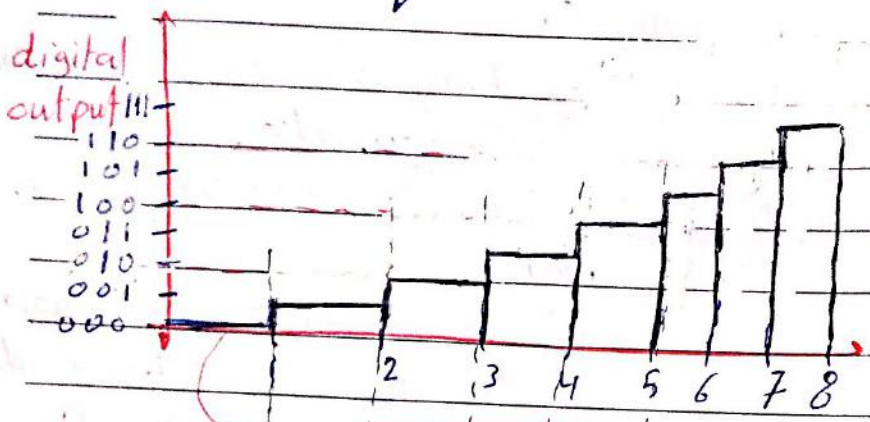
* voltage reference: The range of accessable input values.

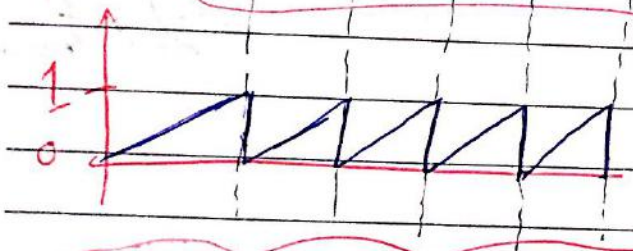


* if one V_{REF} given ⟹ assume the other one is the ground.



⇒ To do quantization: we typically draw graph called "quantization characteristics"





step width = $\frac{8V - 0V}{2^3}$

= 1 V

* error = real value - converted value
= in - out

* step width = 1 LSB

$$= \frac{V_{REF+} - V_{REF-}}{2^{(n)}}$$ → # of bits

* max quantization error = 1

* range error = $\frac{1}{2}$ V.

* LSB: least significant bit voltage, the minimum change in input that will difinitely change the digital output level.

⇒ LSB: measure of resolution.

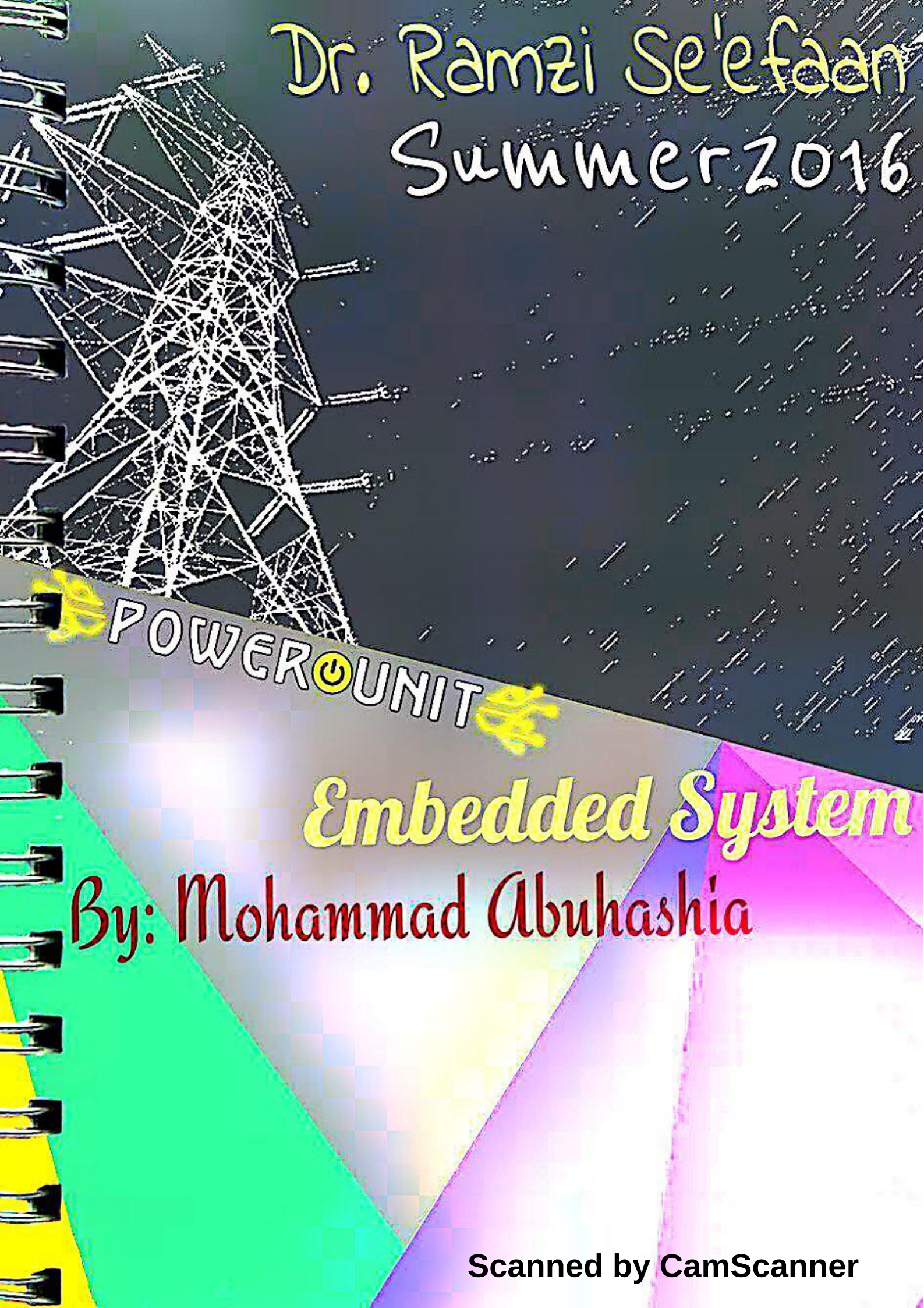
↓ LSB ⇒ higher resolution

⇒ less error

max quantization error = $\frac{1}{2}$ LSB

* max acceptable error = $\frac{1}{2}$ LSB = $\frac{V_{REF^+} - V_{REF^-}}{2^{n+1}}$

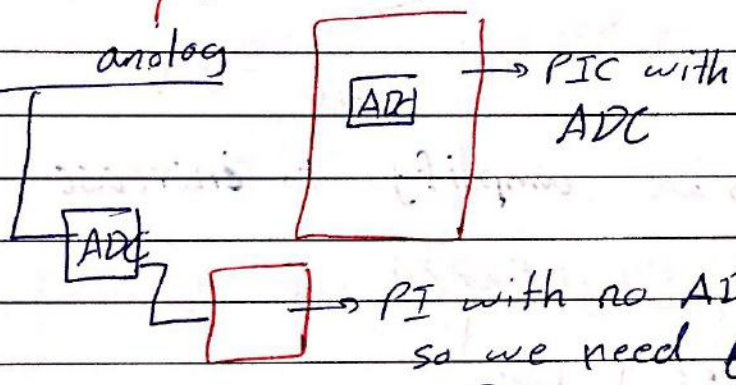
Week 6



* max acceptable error = $\frac{1}{2}$ LSB = $\frac{V_{REF^+} - V_{REF^-}}{2^{n+1}}$

Week 6

16F 84A → No ADCs in it.
16F 877A →



max quantization error = $\frac{1}{2}$ LSB = $\frac{V_{REF+} - V_{REF-}}{2^{n+1}} = \frac{V_r}{2^{n+1}}$ where ... n is # of bits.

slide (9):

if n = 3 bits & V_r = 5 V

⇒ max error = $\frac{5}{2^4} = \frac{5}{16}$ = 312.5m V

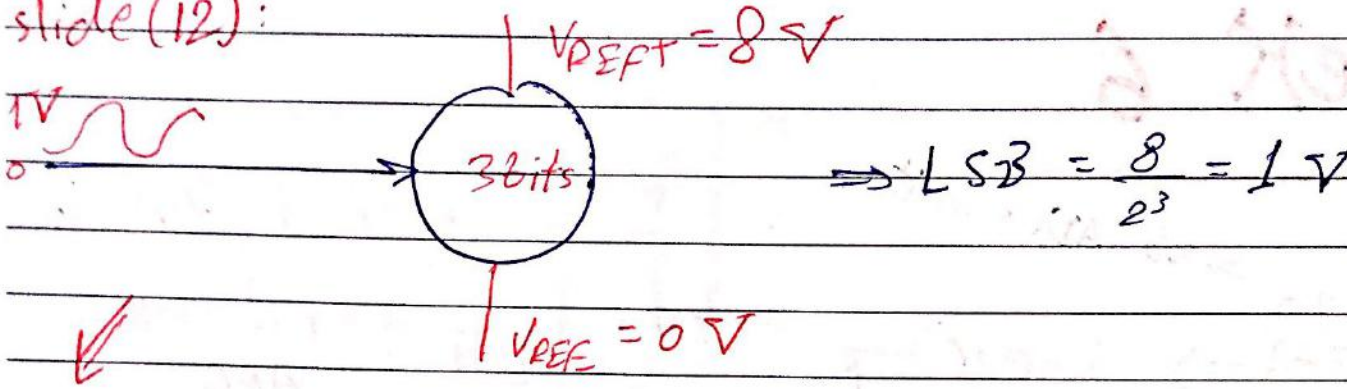
⇒ as percent of range = $\frac{312.5m}{5} \times 100\% = 6.25\%$

** larger n ⇒ (advantage) less error
⇒ (disadvantage) speed is slower
→ less conversion rate
→ smaller frequencies can be converted

* There is a trade off between accuracy & speed

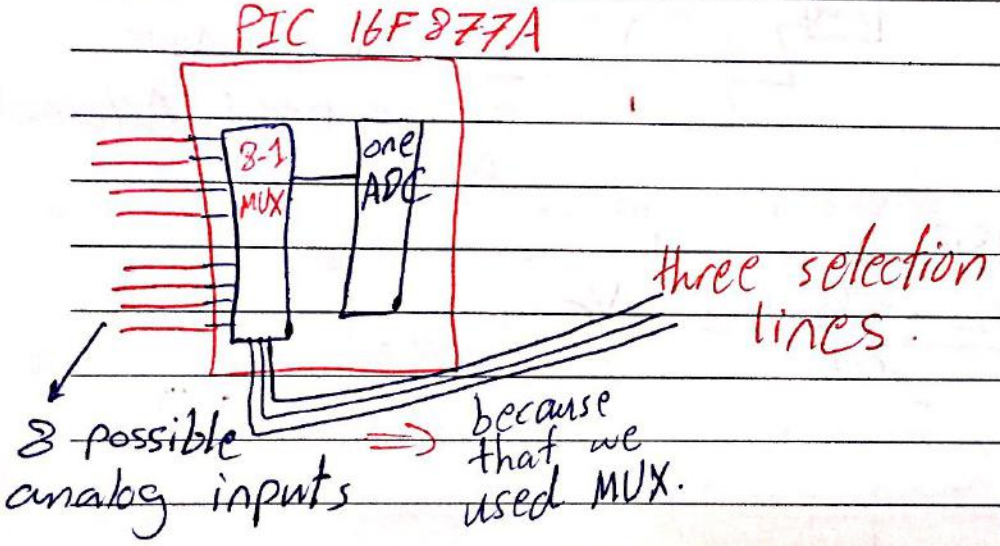
* The type of ADC that found in embedded systems: successive approximation. ADC

slide (12):

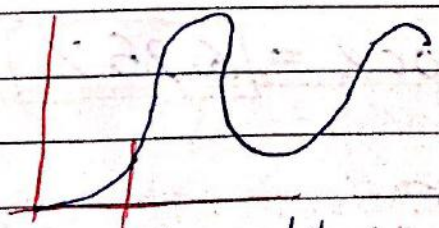


$\Rightarrow LSB = \frac{8}{2^3} = 1V$

$\Rightarrow$ we amplify to increase the accuracy.



* Sample and Hold: before take the sample the input is fixed, the sample is read, then release the input.



sampling while input is changing is not accurate

slide (14):

* When the switch is closed: the capacitor starts charging.

$\Rightarrow V_C$ tracks V_S.

$V_C = V_0 = V_S(1 - e^{-t/\tau})$, $\tau = RC$

* When open the switch: V_o is fixed (almost).

** sample & hold: close the switch when $V_o \simeq V_{in}$, open the switch ⇒ read sample, then repeat.

* Suppose we wait until $V_o = 0.9\,V_{in}$ ⇒ Then how long we should wait before open the switch?

$$V_o = V_{in}(1 - e^{-t/\tau}) = 0.9\,V_{in} \Rightarrow \boxed{t = 2.3RC}$$

from max acceptable error: $\boxed{V_o = V_{in} - \frac{V_{in}}{2^{n+1}}}$

$$\rightarrow V_{in}\left(1 - \frac{1}{2^{n+1}}\right) = V_{in}(1 - e^{-t/\tau}) \Rightarrow \boxed{t = -\ln\frac{1}{2^{n+1}} * \tau}$$

* if n = 8 bits:

$$\Rightarrow t = -\ln\frac{1}{2^9} * \tau = \boxed{6.2\tau}$$

* if n = 10 bits:

$$\Rightarrow t = -\ln\frac{1}{2^{11}} * \tau = \boxed{7.62\tau}$$

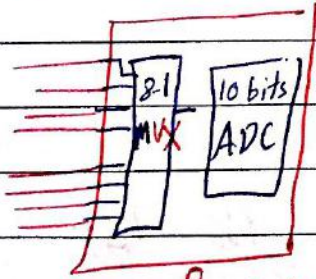
→ larger n → larger sampling time.

↳ see slide (15)

slide (16):

* Total conversion time for one sample = Acquisition time (sampling time) + Quantization time.

* The PIC that we deal with: (see slide 19)



↓ to connect analog input on RSØ

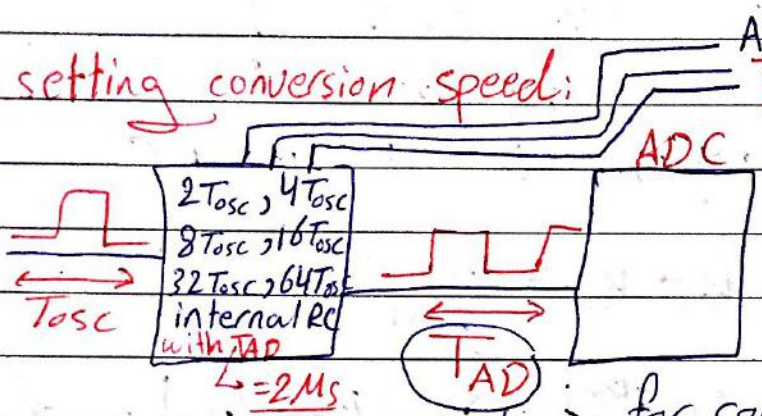
⇒ CHS2 - CHSØ = 101

* Voltage references: can be internal (V_{DD} & V_{SS}) and can be external (RA_3 & RA2). ⇒

* PCFG3-PCFG0 : specify each of the 8 pins whether it is analog or digital and whether the voltage references are internal or external.

* slide (21):

* setting conversion speed:



for conversion to be correct
$\Rightarrow T_{AD} > 1.6\mu s$

if $T_{AD} < 1.6\mu s$
$\Rightarrow$ conversion is wrong.

* The best value for T_{AD}: is 1.6 μs
$\Rightarrow$ select T_{AD} to be minimum but $\geq 1.6\mu s$.

e.g: if $F_{osc} = 1MHz$, what value is the best value for T_{AD}
$\Rightarrow T_{osc} = 1\mu s \Rightarrow T_{AD} = 2T_{osc} = 2\mu s$.

e.g: if $F_{osc} = 10MHz \Rightarrow T_{osc} = 0.1\mu s \Rightarrow T_{AD} = 16\,T_{osc}$
$= 1.6\mu s$.

e.g: if $F_{osc} = 100KHz \Rightarrow T_{osc} = 10\mu s$
$\Rightarrow T_{AD} = 2\mu s$ by selecting internal RC

* for 10-bit conversion $\Rightarrow$ we need 12 T_{AD}

X-bits $\Rightarrow$ needs X+2 T_{AD}
quantization time = (X+2) * T_{AD}

by SW GO/$\overline{DONE}$ = 1 ⇒ start conversion.

GO/$\overline{DONE}$ = 0 by HW when conversion is over

* suppose F_{osc} = 2MHz ⇒ T_{osc} = 0.5 μs ⇒ T_{AD} = 4 T_{osc} = 2 μs.

slide (26): ↙

take $F_{osc}/4$ ⇒ (1) (00) ADCON0 (7:6)

ADCON1(6)

*Most of the time we use right justified.

⇒ if you are interested in 8 bits result

⇒ select left justified.

from table slide (27):

PCFG = 0010 ⇒ 5 analog inputs and internal voltage reference.

Ex. *write all required settings to read only one single analog input and the voltage reference to be internal.

see table (slide 27): PCFG = 1110

CHS2 - CHS0 = 000 → see slide 19 (AN0)

TRISA(0) = 1

since the single analog on AN0.

sampling | conversion | quantization

$-\ln\frac{1}{2^{n+1}} * T$ | GO/$\overline{DONE}$ = 1 | 12 T_{AD}

∴ by SW

switch opens.

GO/$\overline{DONE}$ = 0 → By HW

ADIF = 1

GIE = 1

PEIE = 1

ADIE = 1

⇒

⇒ we can know when quantization ends

⇒ ① GO/$\overline{DONE}$ = 0

② ADIF = 1

③ from delay.

⇒ To know end of sampling we have to write code for delay.

** For conversion to be correct the sampling time should be calculated accurately.

slide (29) * The sampling time = OP Amp setting time (2 µs) + Temp coef ((T-25) * 0.05 µs) + $-\ln\frac{1}{2^{n+1}}(R_S + R_{SS} + R_{IC}) * C_{Hold}$

* for each degree > 25 needs 0.05 µs *

⇒ if T ≤ 25 * Temp coef = 0 *

* R_{SS} depends on V_{DD}.

Example slide (31): R_{SS} = 7K Ω, R_{IC} = 1K Ω, R_S = Ø, Temp = 35°C

T_{AD} = 1.6 µs, t_{ac} = 2 µs, C_{Hold} = 120 pf.

T_{aqu} = 2 µs + 7.6 * (7K + 1K + 0) * 120 P + (35-25) * 0.05 µ

= 9.8 µs write code to give this delay.

⇒ - write settings
- wait 9.8 µs
- GO/$\overline{DONE}$ = 1.

⇒ when GO/$\overline{DONE}$ = 0 or ADIF = 1

⇒ read digital result.

quantization time $= 12 * T_{AD} = 12 * 1.6\,\mu s$

$= 19.2\,\mu s$

$\Rightarrow$ here we don't have to write code for delay since we can know about G/D & ADIF.

* one sample only took $= (9.8 + 19.2)\,\mu = 29\,\mu s$

sample	quant	wait 2 T_{AD}	samp	qu	wait 2 T_{AD}
9.8 μs	19.2 μs	3.2 μs (1.6 * 2)			

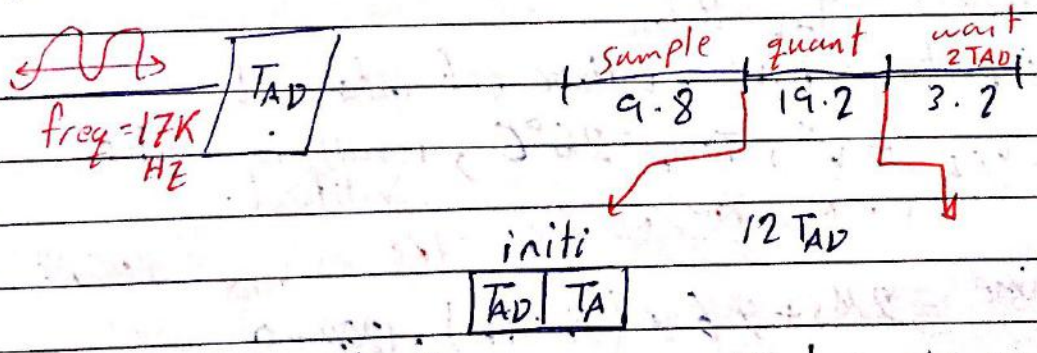
* one sample assume repetitive sampling takes $= (29 + 3.2)\,\mu s$

$= 32.2\,\mu s$

$\Rightarrow$ max sampling rate $= \frac{1}{32.2\mu} = 31\,KHz$

* max input freq $= \frac{1}{2}$ max sampling rate $= 16.5\,KHz$

if we ~~want to use input signal~~:



if the accuracy is less important

$\Rightarrow$ The first 6 bits to be converted correctly and the next 4 bits are not important.

$\Rightarrow$

⇒ The first 8 cycles ⇒ $T_{AD} = 1.6\mu s$
and the next 4 cycles ⇒ choose minimum T_{AD}

for ex. suppose $F_{osc} = 10MHz$

⇒ $T_{osc} = 0.1\mu s$ ⇒ $T_{AD} = 1.6\mu s = 16\,T_{osc}$

⇒ quantization time:
first 8 cycles with $T_{AD} = 1.6\mu s$
next 4 " " $T_{AD} = 2\,T_{osc} = 0.2\mu s$

⇒ quantization time $= 8 * 1.6 + 4 * 0.2 = 13.6\mu s$

⇒ code: write settings
↳ set $T_{AD} = 16\,T_{osc}$ (included by settings)
wait 0.8 μs:
GO/DONE = 1
wait 12.8 μs
set $T_{AD} = 2\,T_{osc} = 0.2\mu s$
wait $= 0.8\mu s$.

Example: slide (33): $T_{AD} = 8\,T_{osc}$
one analog input on RA0, voltage ref internal
$F_{osc} = 20MHz$, $V_{DD} = 5V$, Temp $= 25°C$, result is right justified.
↳ $R_{SS} = 7K\Omega$ / $R_S = 0$ / $R_{IC} = 1K\Omega$ / $C_{Hold} = 120PF$

⇒ sampling time $= 2\mu s + 7.6 * (7K + 1K) * 120p + 0$
$= \boxed{10\mu s}$ write code delay.

since one analog input
⇒ CHS2 − CHS0 = 000

ADCON0 = 01 00 0001
ADCON1 = 10 00 1110

C/R
# of analog inputs
# of external voltage ref.

setting wait GO/$\overline{D}$ = 1

End of CH11

CH(11):

1) Why Analog is more accurate than digital ?

Because it takes unlimited values.

2) What is the sampling rate ?

every how long we read a sample.

3) What is the advantage & disadvantages of more sampling rate ?

advantage: more accurate.

disadvantages: more cost/more size → storage
→ bandwidth

4) What is the advantage & disadvantages of more quantization levels ?

advantage: more accurate.

disadvantages: more storage/more bandwidth /more cost/slower.

5) What is LSB ?

measure of resalution (higher resolution → less error)

6) max error = $\frac{Vr}{2^{n+1}}$ ⇒ larger n:

advantage: less error.

disadvantages: slower/less conversion rate/ smaller freq can be converted.

7) What is the type ADC used in ES ?

Successive Approximation ADC.

8) In PIC 16F877A there is 8 possible analog inputs but ADC take just one how we solve this ?

By using 8-to-1 multiplexer.

9) What happens in the element sample & hold ?

input is fixed → sample is read → release the input.

10) Mention a situation sampling not accurate in it ?

while input is changing.

11) Which determine that voltage ref. internal or external. ?

internal: V_{DD} & V_{SS}.
external: RA_3 & RA_2.

12) What is the use of PCFG3-PCFG0 ?

(1) Determine which of the 8 pins Analog or Digital.
(2) Determine type of V_{ref} internal or external.

13) What is the best value of T_{AD} ?

$T_{AD} = 1.6\mu s$ (or the minimum value such that $T_{AD} \geq 1.6\mu s$).

14) How to set/clear the bit GO/$\overline{DONE}$?

set by SW: BSF ADCON0, GO/$\overline{DONE}$
clear by HW: when conversion is over.

15) When to use right justified or left justified ?

- Most of the time we use right justified.
- if we interested in 8 bits result we use left justified.

16) What can tell us that sampling & quantization end ?

end of sampling : from delay.

end of quantization : ① GO/$\overline{DONE}$ = 0
② ADIF = 1
③ from delay.

17) What is the needed time to wait to start new conversion ?

the converter waits 2 * T_{AD}.

18) When the first 6 bits to be converted correctly and the next 4 bits are not important ?

if the accuracy is less important.

CHAPTER (8): Human Physical Interface.

Switches → Keypads

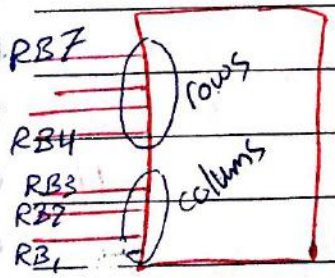
* The most simple way to connect a Keypad, is to connect each PB with single pin ⇒ However, it consumes the pins of the µ.C.

slide(7): 4 rows } each row is connected to one pin.
3 columns } " column " " " " " ".

* To read which button is pressed
⇒ There is two stages:

stage 1: read row pins.
① make row pins input.
② " column " output.
③ output ϕ or the column pins.

stage 2: read column pins.
① make column pins input.
② " row " output.
③ output Zero or row pins. → clrf portB



⇒ ① MOVLW B'1111 0000'
② MOVWF TRISB
③ clrf PORTB

To read it: MOVF PORTB, 0

if we push button (5): W = 1011 0000
5
W = 0000 1010

slide(10) : Example:

* # ⇒ all LEDs are ON.

*RBIF is not cleared on reset so we use in the code.

BCF INTCON, RBIF

To clear RBIF: ① MOVF PORTB, 0
② BCF INTCON, RBIF

if we push [5]: row index = (1011) 0000 → see the code slide 13 → = 1

W = 1

column index = 0000 (1010) → = 1

The first four inst. slide (14):

Value = 3 * row Index + column Index → max value = 3*3+2 = 11 (track it)

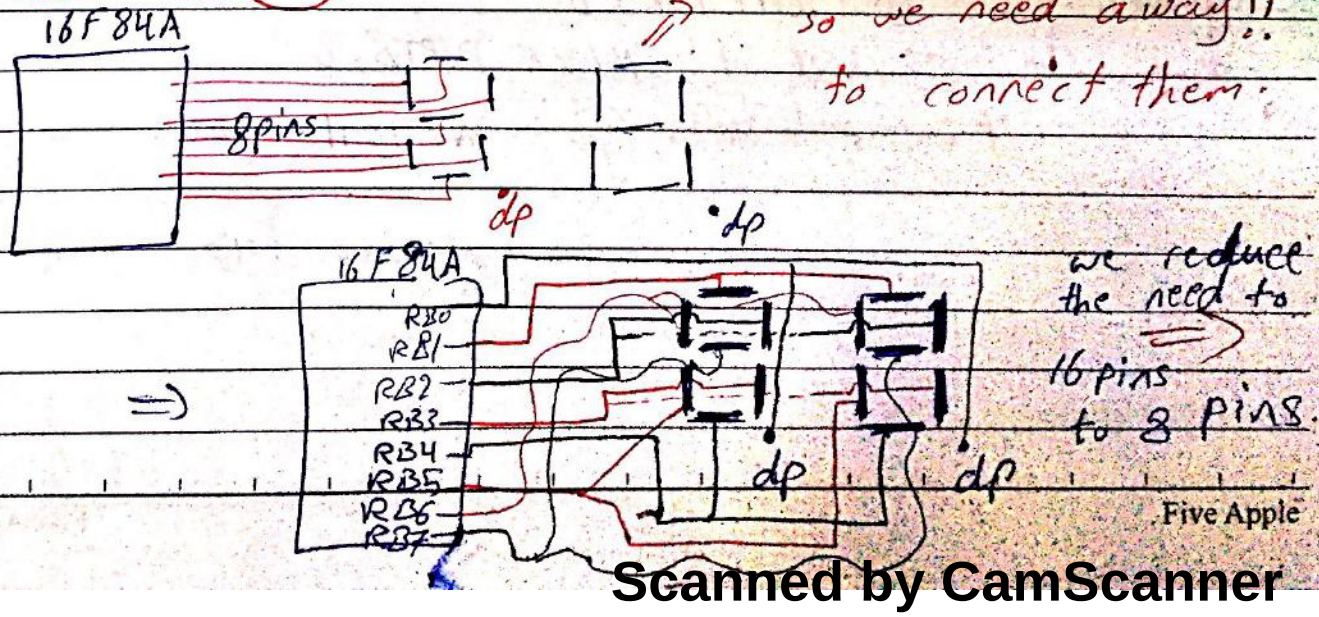
= 3 * 1 + 1 = 4

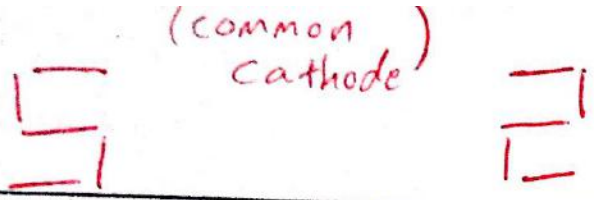
if we push (*): (1110) 0000 = row index ⇒ (3)

0000 (0110) = column index. ⇒ (0)

⇒ W=0 → 3 → 9

* LED Displays:



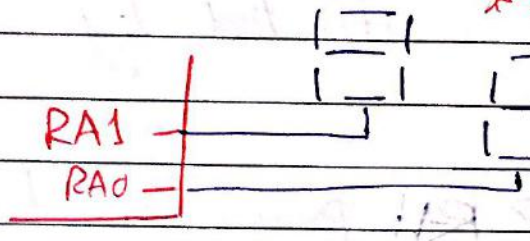



```
MOVLW B'1101 1010'
MOVWF PORTB
```

} This should output (5)

⇒ but the two 7-segment will display No.5

** To solve this connect two enables. ⇒ to the two 7segme



⇒ Now to display No.5 on the right 7-seg

loop

```
bsf PORTA,0
bcf PORTA,1
MOVLW B'11011010'
MOVWF PORTB
```

delay 10ms

if we want to show No.2 on the left 7-seg:

```
bcf PORTA,0
bsf PORTA,1
MOVLW B'10110110'
MOVWF PORTB
```

delay 10ms
goto loop

Now to display 25 on the two 7segments add the following instructions:

⇒ In Conclusion, we need 10 pins for this process.

*see slide (18) ⇒ for connecting 4-(7segments)

Sensors:

** Light dependent Resistors:

LDR — *the resistance value is affected by light intensity.

⇒ more light: ⇒ e-H pairs are formed
⇒ Conductivity increases.
⇒ resistance decreases.

** Optical Object sensing:

optical sensor: composed of 3 pairs:

1) IR LED.

2) Photo transistor.

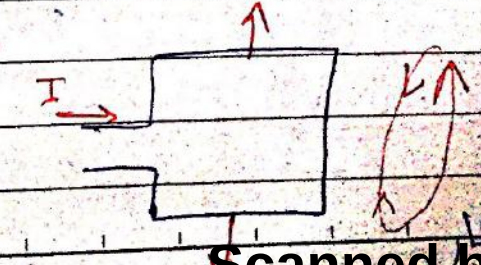
3) Plastic housing: that passes only unseen light like IR.

* optical sensor works in two modes: ① Cut. ② Reflex.

* sensors are connected to: Analog or Digital.

Motors:

** DC Motors:



* سرعة الدوران تعتمد على قيمة التيار
* الاتجاه (الدوران) يعتمد على اتجاه التيار

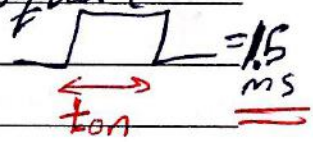
** **Servo motors**: * rotates in angles 0-180°
* works on PWM concept
↳ "Pulse Width modulation"

⇒ according to input pulse width:
the motor rotates in specific angle.

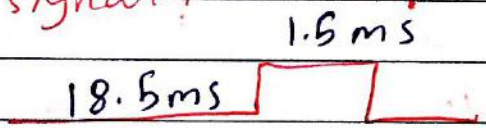
⇒ To rotate in 40°: (see slide 36)

$$1.25 + 40 * \frac{1.5 - 1.25}{90}$$

* To rotate the motor 90° ⇒ we sent a square wave signal with freq = 50Hz and pulse width = 1.5 ms



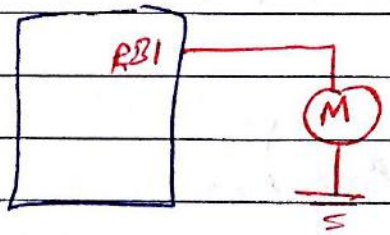
for a signal:



⇒
```
loop  BCF   PORTB,1
      delay 18.5ms
      BSF   PORTB,1
      delay 1.5ms
      goto  loop
```

Week 7

* Interfacing to Actuators:



BSF PORTB,1 ⇒ the motor is expected to rotate.

→ to turn off: BCF PORTB,1

⇒ However, motors require high voltage & current.
⇒ make the PIC as switch to the motor.

⇒ The PIC turn on/off the motor, and the motor is supplied by external voltage supply.

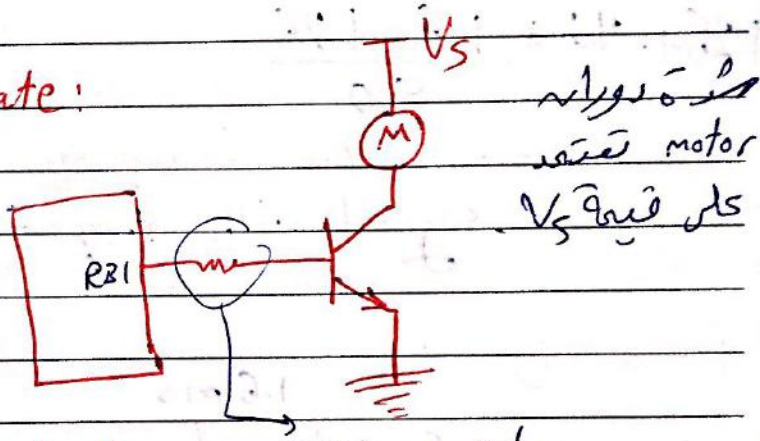
slide (38):

if the current goes into base ⇒ BJT is ON

otherwise ⇒ BJT is open circuit.

a code to make the motor rotate:

BSF PORTB, 1



(slide 39)

* if the load is inductive load:

⇒ there will be a stored power in the inductor so we used the diode to discharge the stored power very fast

(slide 42)

if you close two switches in the same side

⇒ short circuit.

if we close the two upper switches:

⇒ No current pass in the motor.

⇒ (so we just close two swiches diogonally)

(slide 43):

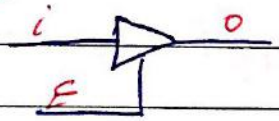
L293D: *provides 2 full H bridges ⇒ 2 motors rotate in 2 directions.

or

*provides 4 half bridges ⇒ 4 motors in one direction.

⇒ 16 pins:
- 4 inputs (1A, 2A, 3A, 4A)
- 4 outputs (1Y, 2Y, 3Y, 4Y)
- 4 pins ground and heat dissipation.
- 2 enable pins.
- 2 voltage supplys.

*Review:



E	i	o
0	x	open circuit
1	0	0
1	1	1

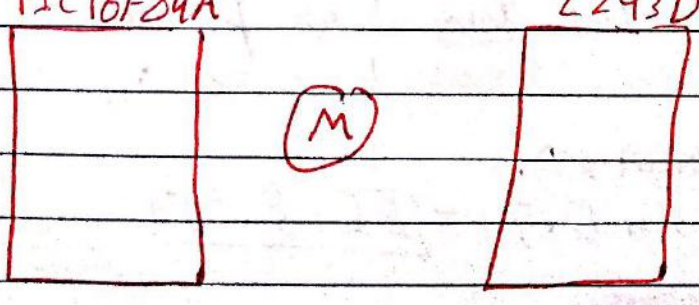
→ V_{LS}: $V_{Logic\ High}$, when the input is high
⇒ it is represented as V_{LS}

→ V_{OS}

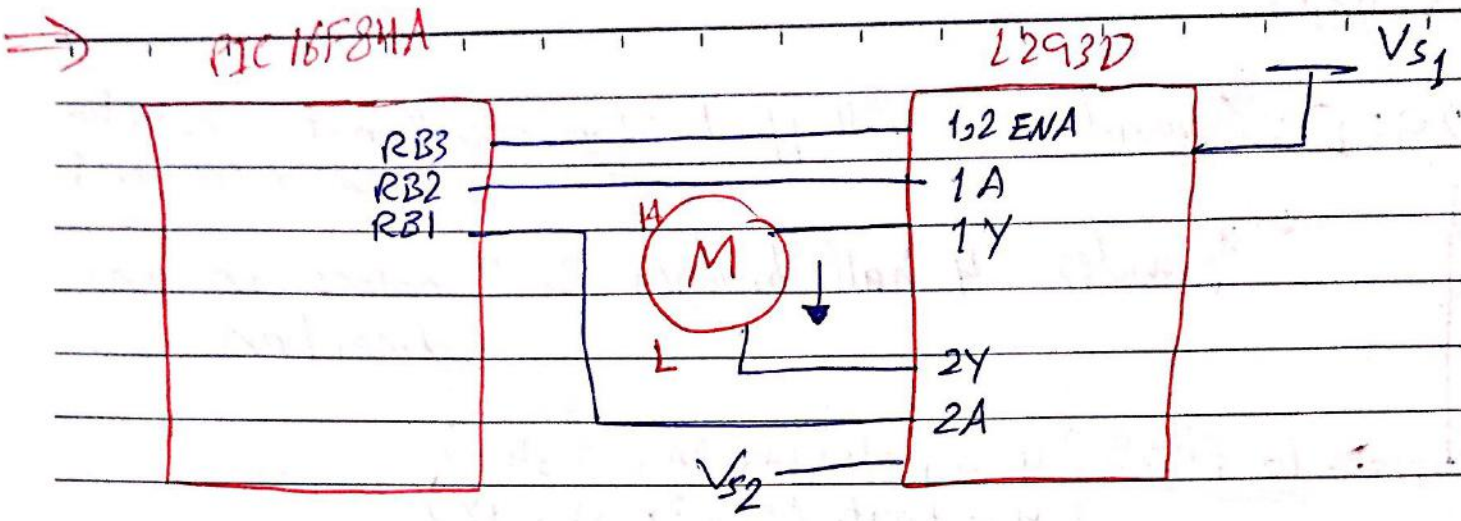
* V_{LS} should be less than V_{OS} ($V_{LS} < V_{OS}$)

⇒ if ($V_{OS} < V_{LS}$) ⇒ motors are off always.

~~V_{OS}: work as drive.~~
if $V_{OS} = 0$ ⇒ the PIC is off.



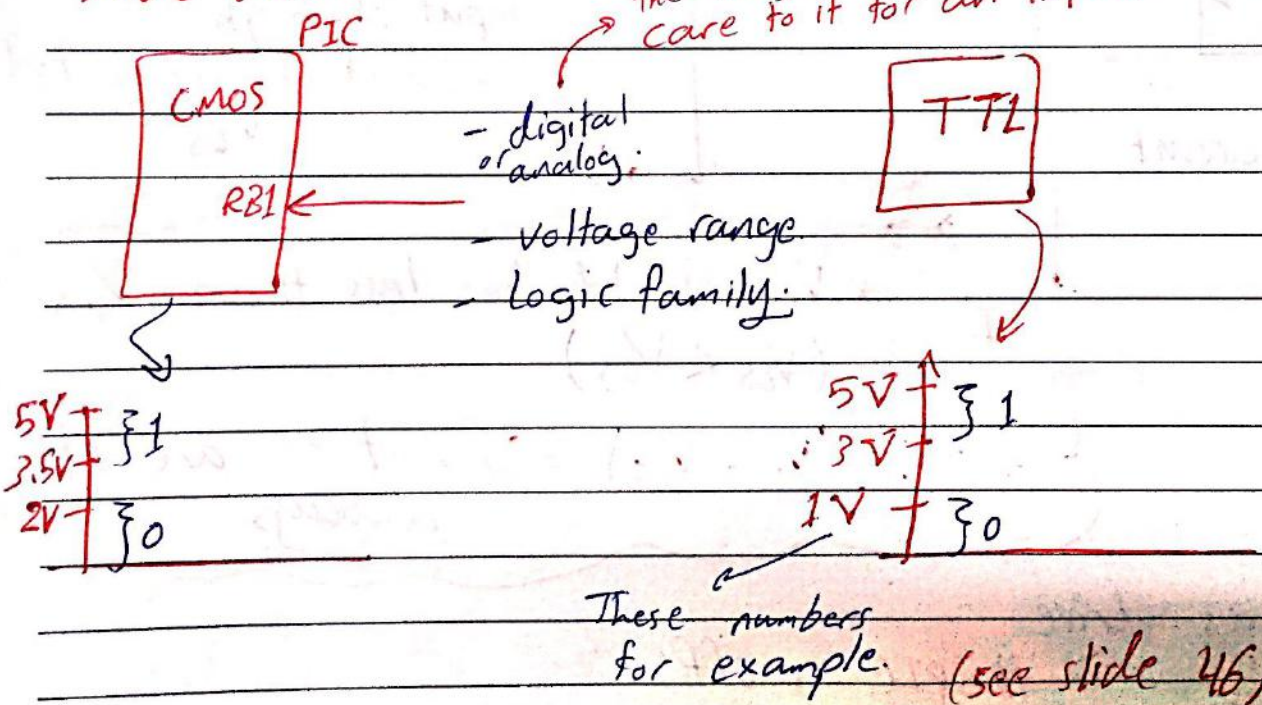
⇒ Connect this circuit !?



if we asked to write the code!

```
BSF  PORTB,3
BCF  PORTB,1
BSF  PORTB,2
```

slide (46):



slide (48): The red diode protect from high peack.
The blue diode " " low "

suppose: $V_D = 0.3V$, $I_{d\,max} = 20mA$, $R_{prot} = 1K\Omega$

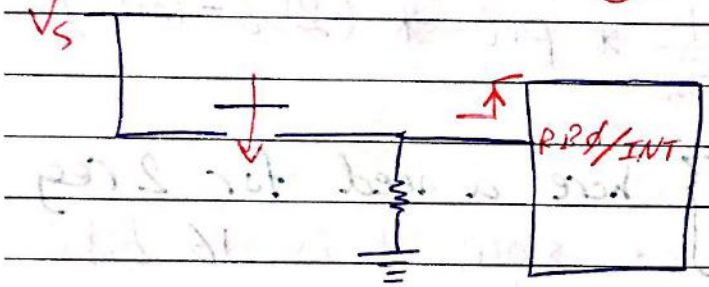
max input voltage $= 1K * 20m + 5.3 = \boxed{25.3\ V}$ ⇒

* any input voltage > 25.3 will destroy the diode and that will destroy the PIC.

⇒ minimum voltage = -0.3 - 20m * 1K = -20.3V

* filtering: we use the RC to cancel the noise. also the schmitt trigger used for cancel the noise and to be more faster for the slow logic edges.

* switch Debouncing:



external interrupt on rising edge at RB0:

```
BSF   INTCON, GIE
BSF   INTCON, INTE
BSF   option-Reg, 6
loop  goto  loop
ISR   INCF  counter,1
      BCF   INTCON, INTF
      retfie
```

This code if we execute it: could cause unpredictable behavior.

switch bouncing: ⟷ few µs to 10 µs ⇒ we write delay 10µs ⇒ ≥ bouncing time.

* Debouncing have 2 solution:

1- SW
2- HW

End of CH8

CHAPTER (9): Taking Timing Further.

* we will study in this chapter about three subjects: Timer1, Timer2, CCP.

* PIC 16F87XA contains:
  - Timer0
  - Timer1
  - Timer2

* Timer0:
  - TMR0
  - OPTION_REG

$\Rightarrow delay = \frac{4}{F_{osc}} * pre * (256 - initi.)$

* Timer1: contain 3 registers.
  - TMR1H } here a need for 2 reg since it is 16 bits timer.
  - TMR1L
  - TMR1CON

counter: 0, 1, ..., $2^{16}-1$ → 0

TMR1IF = 1 → do something

To interrupt to happen:
GIE = 1, PEIE = 1, TMR1IE = 1

* The clock can be synchronized (T1SYNC = 0) or unsynchronized (T1SYNC = 1).
* The clock can be prescaled or unprescaled.
* The clock can be: ① Internal (TMR1CS = 0)
  ② External (TMR1CS = 1)
  - RC0 if T1OSCEN = 0 → it is used for counting
  - RC1 if T1OSCEN = 1 → it is used for external osc.

* RC1: ⇒ TMR1 can keep counting in sleep mode.

⇒ **can wake the μ.C up from sleep mode.**

* T0 can't wake up μ.C.
* T1 can wake up μ.C.

* Delay for T1:

$$delay = \frac{4}{Fosc} * prescale * (2^{16} - initialize)$$

$$\Rightarrow max\ delay = \frac{4}{Fosc} * 8 * 2^{16} = 8 * max\ delay_{TMR0}$$

* When dealing with TMR1:
No need for bank selection since all of TMR1H, TMR1L and TMR1CON in bank(0).

* **Timer 2**: (8 bits)

* only Internal clock ($\frac{Fosc}{4}$).
* prescaled by 1, 4, 16.
* postscale. ⇒ postscale: when TMR2 = PR2
⇒ # of times equal the postscale
⇒ TMR2IF=1
* for interrupt to happen: GIE=1, PEIE=1, TMR2IE=1

* Comparator: it compares TMR2 with PR2, when they are equal ⇒ 1- reset TMR2

* Delay of T2:

$$delay = \frac{4}{F_{osc}} * prescale * postscale * (PR2+1)$$

⇓ assuming No initial value.

** if an initial value is used:

for ex.

postscale = 10

PR2 = 100

TMR2 = 50

$$\Rightarrow delay = \frac{4}{F_{osc}} * pre * (100+1-50) * 1 + \frac{4}{F_{osc}} * pre * (100+1) * 9$$

(9 → postscale)

* max delay (T2) $= \frac{4}{F_{osc}} * 16 * 16 * 256$

$=$ max delay (TØ)

* When we deal with TMR2:
There is a need for bank selection.

---

slide (13): CCP:

- capture.
- Compare.
- PWM.
- .....

⇒ for all three:
* We need timer.
* Know that non of them has its own timer.

* RC2: must be an input.

① Capture: when an event happen, read (capture) the time.

⇓

use timer1

* Events:
1. every negative edge on RC2.
2. every positive edge on RC2.
3. every 4th positive edge on RC2
4. every 16th positive edge on RC2.

② **Compare**: when time = value ⇒ do an event.

⇓

use timer1

* 4 modes: [1] set RC2, CCP1IF=1
[2] clear RC2, CCP1IF=1
[3] CCP1IF=1
[4] trigger a special event, CCP1IF=1

③ **PWM**: generate a square wave signal with specific t_{on} & T.

⇓

use timer2

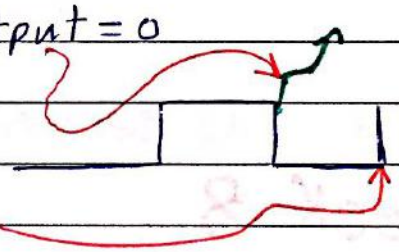
* Every time timer2 is incremented by 1 ⇒ it is compared with two values:

① 10 bits value CCP1H ⇒ 2 bits from CCP1CON

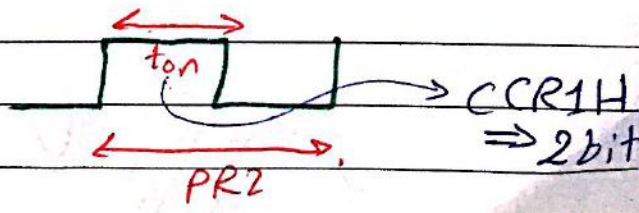
when they are equal: ⇒ output = 0

② when TMR2 = PR2

⇒ 1) set output.
2) Resets TMR2.
3) Latch the CCPR1L into CCPR1H



* PR2: determine the cycle



** To modify t_{on}:
I write CCPR1L, but it takes effect only after the current cycle.

** if PR2 < CCP1H ⇒ There is No PWM output.

* Always: postscale in PWM is disabled

⇓

the first of the slide (21).

assume: $t_{on} = 1\,ms$, if: pulse width = 64

64 ⇒ 10000000

CCP1CON<5:4>

CCP1L

Slide (22 → 26) Not included.

End of CHq

Week 8

The end

GOOD LUCK

Five Apple