Embedded

Done By:
Sarah Alkasasbeh

POWER UNIT

Lecture 1

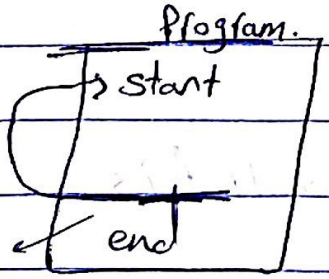
Real time :- there are deadlines for actions after which the actions are useless.

. To deal with any embedded system :-

1) Inputs
2) outputs
3) user interaction
4) link to other systems
5) Hardware
6) Software (code)

accessible | Can't be accessed.

. software driven → the embedded system is completly controlled by the software (Program) (always running)



before it ends

. The M.C is a computer on chip because it has all the main components of the computer

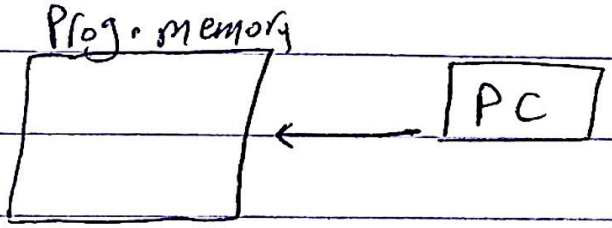
1) CPU :- control and program execution
2) memory :- storage
3) buses :- to transfer data between diff. modules
4) I/O :- to deal with the external world.

memory

Program memory
- holds the code (instruction)
- Permanent

data memory
- Voltaile (lost on power die)
- holds values of variables at execution time.

- data memory → takes much less power / much faster

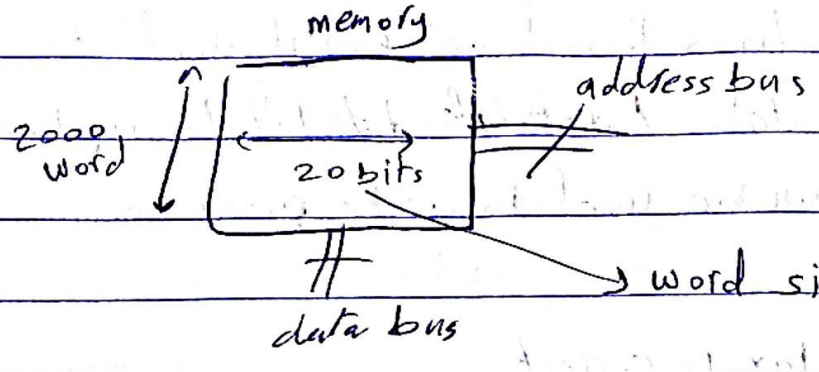


- PC → holds the address of next instruction to be executed (auto increment)

- reset vector → default value that is loaded in the PC on power up.

- fetch → the CPU brings the instruction that is address is in the PC.

- pipelining → while the CPU is executing on instructions, it fetches the next instruction.



- each memory has 2 buses.

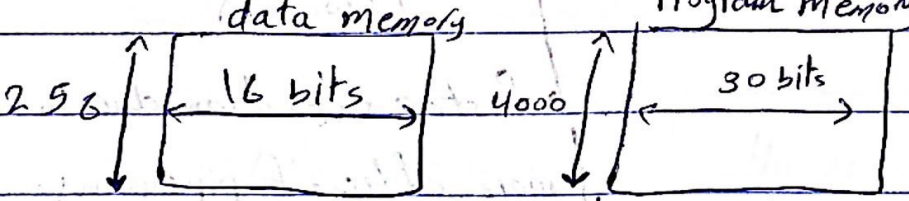
$$\text{address bus} = \lceil \log_2 2000 \rceil = 11 \text{ bits}, \quad 0 \rightarrow 2047 \text{ words}$$

(ceiling)

if less than 11 bits → missed words we can't read and write them.

data bus ≥ 20 bits

memory size = $2000 \times 20 = 40000$ bits $= \frac{40000}{8}$ bytes



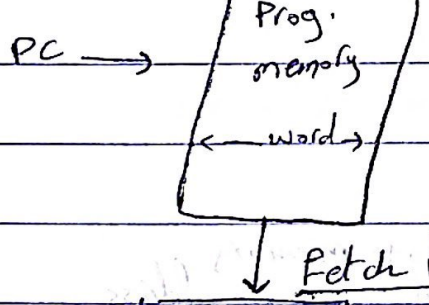
	data memory data bus	data memory address	Prog. memory data	Prog. memory address
VNA type	max{16, 30} = 30	12	max{16, 30} = 30	12 → large values
Harvard	16	8	30	12

- address of any variable → in data memory
- address of any instruction → in Program memory.

- Instruction set → instruction's format and how it should be written in order to be executed. / detailed description of all instructions when can be used to write programs.
- Compilation
  - ① check the syntax is correct
  - ② machine code generation.
- Syntax error is a result of not adhering to the IS.

instruction set types :-

1) CISC :- many types of instructions including complex instructions like mult. → many addressing modes
2) RISC :- only simple few instructions → longer code.
   - longer compilation (slower)
   - faster execution due to more efficient pipelining.
   - few addressing mode



all details required by the CPU to execute the instructions

- opcode → type of operation and format of instructions.
- operands → variables that the instruction works on
- addressing mode :- is how to name the operand in the instruction.

- Digital I/O → 0, 1
- Analog I/O → any value in a range. (signal)

- if a device doesn't support analog or Digital I/O → use convertor

- Reset → load the PC with the reset vector.
- interrupt → an event that needs immediate attention.

- M.C within the same family have the same core (=CPU) and the same IS, and the same instruction format.

- M.P is part of M.C

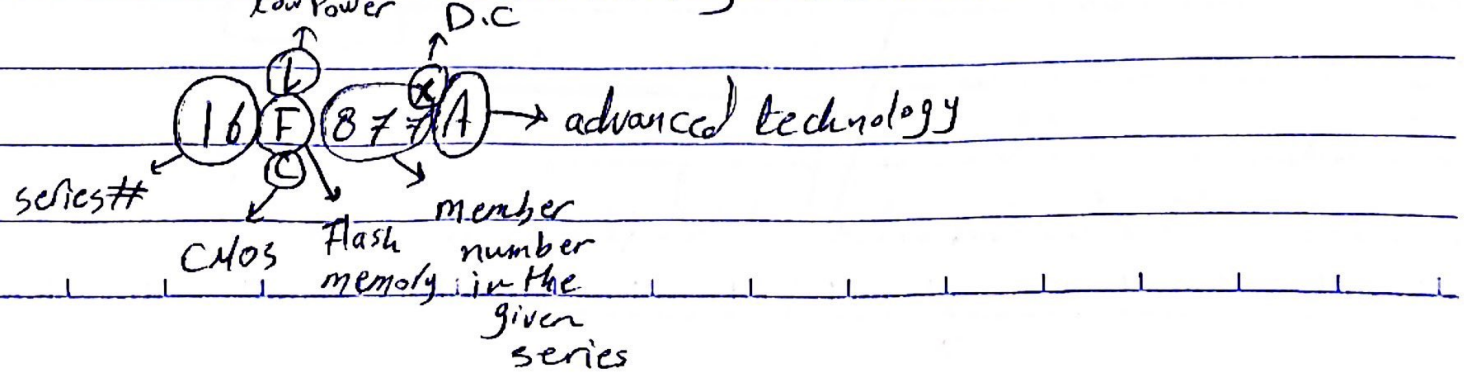
lecture 3

PIC M.C has its own design, RISC, Accumulator.

Accumulator :- it accumulates the results of the instructions

working register :- most of the instructions use this register as a variable.

16-bits computer ⟹ related to the size of the word in the data memory ⟹ all variables are of size 32 bits.



/ /

- Stack is a kind of memory (volatile), automatically written and read, holds return address.

1 level stack → it can holds only single value. (return value)
x levels stack → you can do "x"-nested calls.

- Interrupt vector:- value loaded in the PC on ~~reset or power~~ Interrupt.

(chapter 2)

PIC → 16F84A

~~Pins~~ ↑ Pins ↑ peripherals.

1K Program memory	68 bytes data memory	64 bytes EEPROM
↓	↓	↓
1K instruction	max 68 variables word = 8 bits	Values of Variables (Permenant) ↓ settings and configuration of M.C

- pins → used for different I/Os

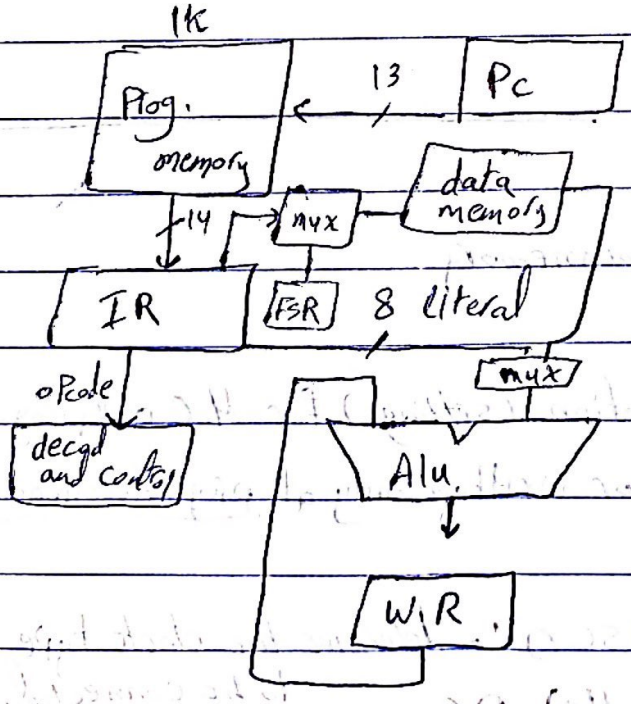
lecture 4

16F84A (Mid range family) (up to 20MHz) ← max to operate

↳ 18 pins
- ↳ 5 port A } for I/Os
- ↳ 8 portB }
- ↳ 8 bits timer
- ↳ (2) external clock → must be connected even if: zero or more I/Os are Connected.
- ↳ 2 Vdd and Vss
- ↳ 1 external reset

1k memory need 10-bits address or more

($\overline{MCLR}$: external reset = 0 ⇒ reset.
↳ active low ↳ must be connected to high to make the M.C works.



* addressing modes :-
1) literal / 2) direct access / 3) indired ll.

- if the operation have 2 inputs
  - ↳ 1) working Register
  - ↳ 2) either from the instruction it self (literal) ex:- add value 5, or from the (data memory)
    - ↳ a) direct address which means that the address is in the instruct...
    - ↳ b) indired address (in FSR)

- The result of any operation is written in either W-reg or data memory.

- any instruction needs 2 variables have 2 versions:
  1) (W) with (literal)
  2) (W) with (data memory)

---

status register → holds information about last executed instruction.

- stack → holds the return address (PC)
  ↳ write (call) / read (return)

- Data memory ≡ file register

- Configuration word:
  ↳ 1) stored in the program memory
  2) not code
  3) Configuration information (settings) for M.C and prog.
  4) Can be modified or written only at prog. downloaded time.

↳ it contains: 1) FOSC1 and FOSC0 :- determine the clock type to be connected.
↳ (LP, XT, HS), RC
crystal ↓ ↓ ↳ resistor capacitor
differ in ↓ ↓ ↓
frequency range.

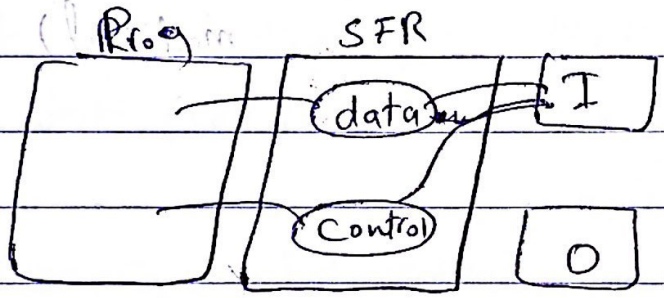
2) WDT (watch dog timer) :- is to monitor M.C and reset it on crash.

3) Power up timer enable (active low): if 0 → keep the M.C in the reset mode for a while after power up.
if 1 → (disabled) → works directly.

4) Cp (code protection) → if 0 → able (the code can't be accessed)
if 1 → disable (but the code can't be modified)

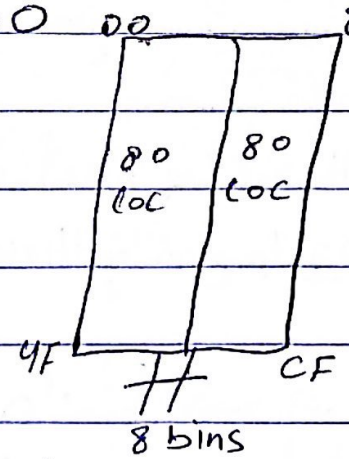
- Data memory is divided in 2 ways
  - 1) Vertical → to deal with I/Os:
    - a) upper :- special function Registers (SFR)
    - b) lower :- General purpose registers (GPR) (for your own variables)

.memory mapped I/O :- each I/O device has a part of the memory to deal with it through the SFR



Control :- settings and Config. of I/O
data :- data entered or data to output.

  - 2) Horizantal
    - 2 banks : b1 and b0



.if the memory of size 160 words is flat ⇒ 8 bits address

$$\lceil \log_2 160 \rceil = 8$$

.if I want to address a word in a specific bank ⇒ I need 7 bits address → $\lceil \log_2 80 \rceil = 7$, still, I need to choose the bank

. If the memory is divided in 2 banks, addressing any word is done in 2 steps :-

1) choose bank (1-bit), if not chosen ~~from status~~

2) choose address in the bank (7-bits) ~~from FSR~~

~~redirect~~ . Bank selection depends on the addressing mode

1) literal → no need to address the data memory

2) Direct → status (5) RP0

3) indirect :- FSR (7)

---

* write a pseudo code to read the data memory locations 4E, 6F, 8A, 93, 25, assume there is an instruction (RDx) which reads the data memory world at address x.

4E → 0100 1110 → bank 0

6F → 0110 1111 → bank 0

8A → 1000 1010 → bank 1

93 → 1001 0011 → bank 1

25 → 0010 0101 → bank 0

address in the instruction → direct

. Choose bank 0 → throug status (5)

RD 4E

RD 6F

choose bank 1

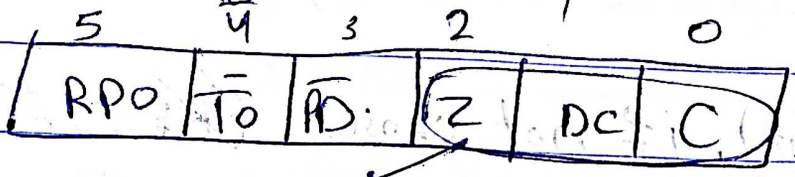
RD 0A

RD 13

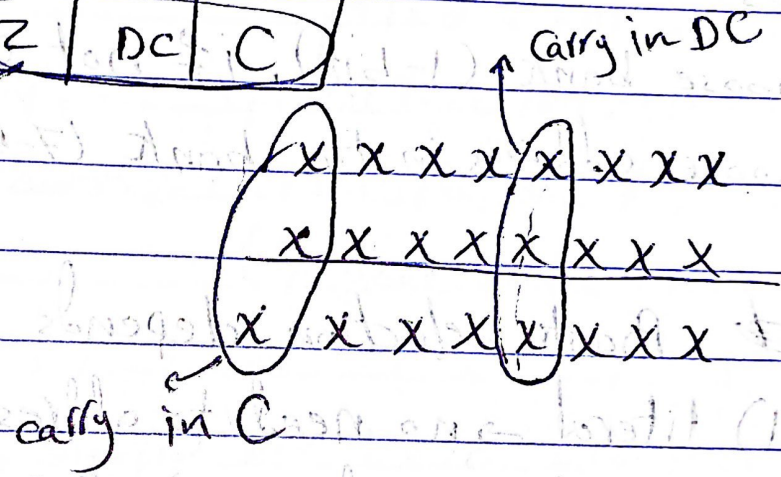
choose bank 0

RD 25

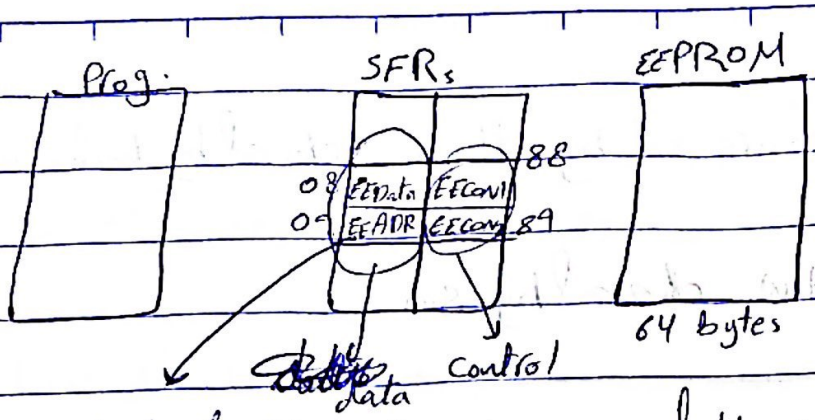
- Status register



if the result = 0 Z becomes 1



RPO → bank selection in case of direct address.



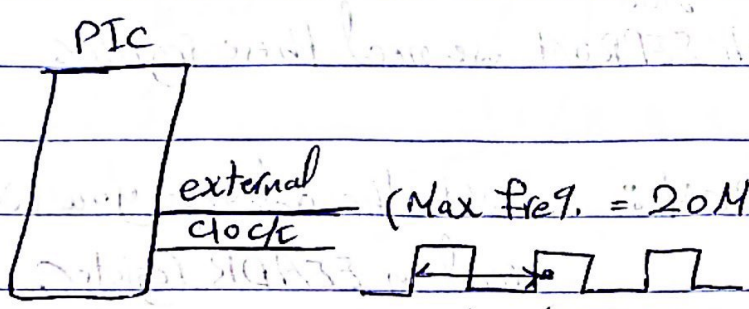
- to deal with EEPROM we need these registers

- How to read EEPROM :- . write the address you want to read in EEADR register.
switch to bank 1 → . set RD bit in EECON1 (=1)
once it is set, start reading, copying from EEPROM into EEDATA.
switch to bank 0 → . To deal with the copied value, read it from EEDATA

- RD → set by software, cleared by H.W.
1 0

- To write the EEPROM :- 1) put the value in EEDATA
2) Put it address in EEADDR. ← bank switching
3) set WREN (write enable) in EECON1 // writing to EEPROM is allowed.
4) write 55h to EECON2
5) write AAh to EECON2
↳ = are you sure because once you write you can't undo it.
6) set WR bit start copying the value from EEDATA to EEPROM
• once writing finish successfully, EEIF=1 in EECON1, if writing failed => WRERR=1 in EECON1.

- WR and RD bits are set by software, cleared by hardware
- EEIF set by H.W, cleared by SW.



↓ Tosc: cycle length of the external clock, Fosc = 1/Tosc

- To execute or fetch single instructions, Tosc most of the time is not enough.
- each instruction to be fully executed needs fetch then execute.
- Tinst: The time that is enough to fetch or execute a single instruction.
- Tinst = X * Tosc, where T is a factor.

example: assume we have 2 MCs ↑ MC1 and MC2

	Fosc	Tinst
MC1	50MHz	10Tosc
MC2	20MHz	2Tosc

$T_{osc1} = \frac{1}{50MHz} = 20ns$

$T_{osc2} = \frac{1}{20} = 50ns$

$\Rightarrow$ Tinst1 = 10 * Tosc = 10 X 20 = 200 ns

Tinst2 = 2 * Tosc = 2 X 50 = 100 ns

• each instruction takes 1 Tinst. for fetch and one Tinst for execute = 2Tinst, to be fully execute.

• In PIC M.C. Tinst. = 4Tosc

ex:- Fosc = 20MHz ⇒ Tosc = $\frac{1}{20}$ = 50ns ⇒ Tinst = 4 * 50 = 200 wich is the time to make fetch or execute.

ex:- given a M.C with freq 4MHz, How long does it need to fully execute a prog. composed of 10 instructions ?

Fosc = 4MHz

Tosc = $\frac{1}{4}$ = 0.25 MHz

Tinst = 4tosc = 1Ms , each instruction requires 2 Tinst.

for fetch exect = 2Ms

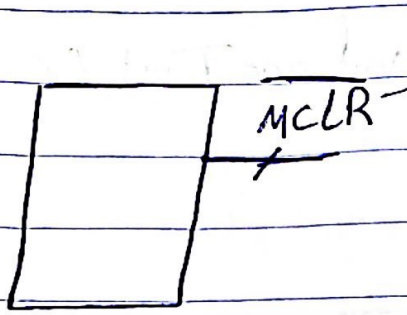
a prog. of 10 instructions needs 10 inst x $\frac{2\mu s}{inst}$ = 20Ms

"without pipelining"

• with pipelining, the time goes approx. to half ⇒ sometimes, the pipeline fails.

⇒ The pipline fails when the fetched instructions will not be executed next.

⇒ with pipelining, all instructions require Tinst to be fetched and executed, except the instructions which generates pipeline failure ⇒ They need 2 Tinst.



. reset → PC becomes = 0, and all SFRs are default value

. On power-up, to keep the PIC in reset mode for a while, there are 2 ways :-

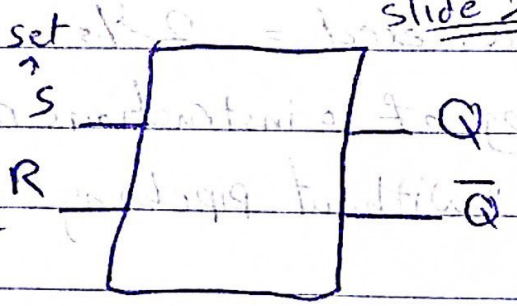
1) external

↳ after 3T the Vc is high and the PIC exits from reset.

. R → current limiting resistor to protect the PIC.

2) internal

slide 23



• when does resct happens?

when $\overline{Q} = 0$, when S=1

• when does S=1 ?

↳ when any of the following happens :-

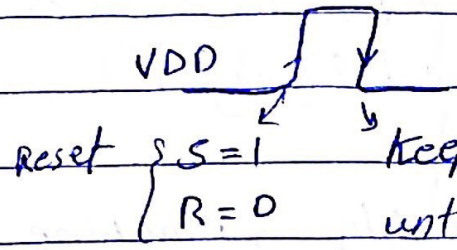
a- $\overline{MCLR} = 0$

b- WDT detects a problem while PIC not in sleep

c- Vdd rise detect

S	R	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$ → no change
0	1	∅	1
1	0	1	∅
1	1	unstable	

- VDD rise detect slide 23

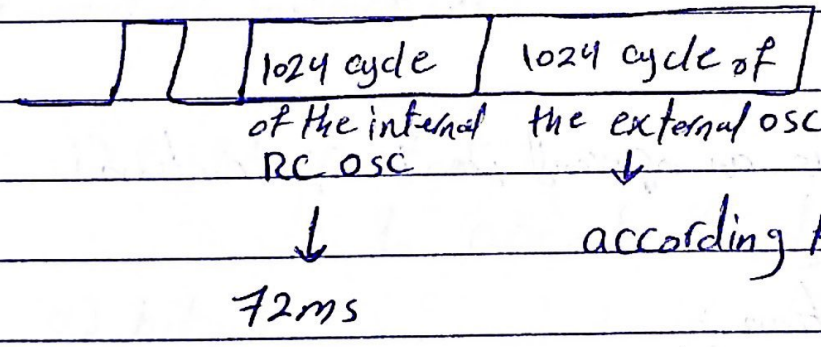


when does R = 1 after power up?

- assume Enable PWRT, OST = 1 (enable)

10-bit counter → becomes 1 after 1024 cycles of the osc connected to it. (because 10-bits)

- OST counter starts working after PWRT counter



- if one of them is disabled (PWRT = 0, OST = 0), no need to wait for counters.

- PWRT is enabled from config. word (bit in config. word)
- OST by default enabled for osc types XT, HS, LP.

- each line in the Assembly code = a line in the machine code. (more efficient than high level language)

  - high-level language → Assembly → machine

- instructions are executed by the CPU (ALU)

- if destination bit "d" = 0 → the result stored in the working register.
- if destination bit "d" = 1 → the result stored in the data memory.

- The destination bit is an operand in the instruction.

- Categories of instruction :-
  1) Byte oriented file register (file register ≡ Data memory)
     ↳ works on a full Byte in the data memory
  2) bit oriented file register
     ↳ works on a single bit.
  3) literal operations
     ↳ no data memory access (the value is in the instruction)
  4) Control instructions (change path of execution)
     ↳ like (jump, return ...)

Parameters of each category :-

1) Byte oriented → address of the file register (a)
   - f (7-bits) (not 8 because of status register)
   - destination bit "d" (1-bit)

2) bit oriented → a = f (7-bits), b = bit to work on (3-bits)
   - stored in the same location.

3) Literal → value to work on "k" (8-bits)
   - always stored in working register

4) Control → "k" (11-bits) call / GoTo

. Format of instructions :-

1) Byte oriented
   - opcode, f, d

2) bit "
   - opcode, f, b

3) literal
   - opcode, k

4) Control
   - opcode, k

- Arithmetic instructions

because we add 2 values ← OP ← ADDLW (literal → working) K

ADDWF (data memory) F, d

OP ← SUBLW K

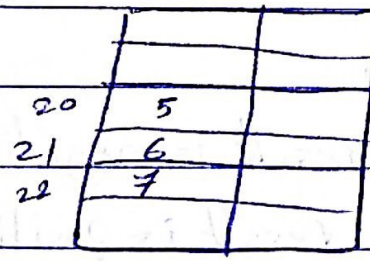
SUBWF F, d

OP ← INCF f, d

DECF f, d

COMF f, d

Example:-



store in working

address

1) ADDWF 20, 0 → ADD [w] + [20]

2) ADDWF 21, 1

3) ADDLW 3

4) ADDWF 22, 1

5) ADDLW 20

working register ← iW = 2

1) W = 7

2) address 21 becomes = D_h (7+6)

3) W = 10 = A_h

4) address 22 becomes 11_h (7+10)

5) W = 2A

- SUBLW k
- SUBWF f,d

} we always subtract the value of the working register

↳ [k] - [w] → [w]

↳ [F] - [w] d=0 → [w]

d=1 → [F]

- assume A = 5, B = 7

A 00000101 $\xrightarrow{2\text{'s comp.}}$ 11111011

B 00000111 $\xrightarrow{2\text{'s comp.}}$ 11111001

A - B (5 - 7) = A + 2's comp. (B)

```
c=0 ← 00000101
    +
      11111001
    ----------
      11111110
```

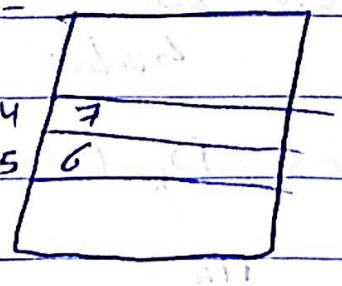
B - A = B + 2's comp of A.

```
c=1 ← 00000111
      11111011
    ----------
      00000010
```

if c "carry" = 0 → The result is negative

" " " = 1 → The result is positive

Example :-



1) SUBLW 9

2) SUBWF 24, 0

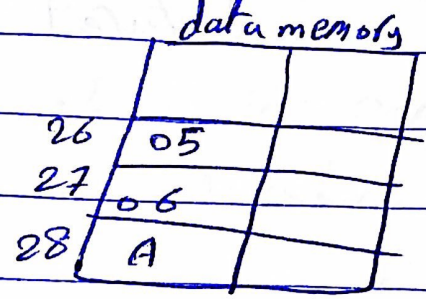
3) SUBWF 25, 1

1) 9 - 5 = 4 → W = 4

2) 7 - 4 = 3 → W = 3

3) address 25 becomes 3

↳ 6 - 3



W = 0F

1) INCF 26,0

2) DECF 27,1

3) COMF 28,0

1) W = 06 (5+1 → [W])

2) address 27 becomes (05)

3) W = F5 → A: 0000 0101

$\bar{A}$ = 1111 0101 = F5

- each instruction cycle = 4Tosc
  - 1Tinst = 4Tosc

- Logic instructions (used to do bits masking)
  - set, clear, or complement Part of the word and keep the others.

- $A \cdot 0 = 0$ } clear
- $A \cdot 1 = A$ }

- $A \oplus 0 = A$ } complement
- $A \oplus 1 = \bar{A}$ }

- $A + 0 = A$ } set
- $A + 1 = 1$ }

ANDLW k
ANDWF f,d

IORLW k
IORWF f,d

XORLW k
XORWF f,d

- Example: write instruction to set the least sig. 4 bits of the w-Reg.

answ:- set and no address
- IORLW

IORLW 0F (0000 1111)

Example:- write instruction to clear the most sig. 4 bits in w-reg.

ANDLW F0

• Example :- Comp. even position bits in w-reg.

XORLW 55

• Example :- write code to clear the least sig. 4 bits of the address 22.

~~ANDWF 22,1~~

↳ 1) write in the working register the value F0

2) ANDWF 22,1

. Data movement instructions.

• MovLW → move literal to working register

• MovWF → move from working register to data memory

• MovF → from data memory to working register or to itself

. To write (intialize) any data memory location with value

↳ A) movLW B) MovWF

• To copy a value from data memory location to another

↳ A) MovF B) MovWF

. SWAPF → swapping between the most sig. 4 bits with the least sig. 4 bits → the result stored in working register

- MOVF to itself like (MOVF 21) → to check the z-flag

MOVF 20, (1) ← d

z == 1 → [20] = 1

z ≠ 1 → [20] = 0

- Call instructions

- CALL k : Pc → k (instruction in address k)
- GOTO k : Pc → k (instruction in address k)

- At first, CALL instruction stores the PC in a stack in order to return and complete the execution

(Pc —Push→ stack)

- RETURN (Pc ←Pop— stack)

- Conditional branch :- GOTO if condition is true

(loops, if, if-else, switch ...)

INCFSZ f, d
DECFSZ f, d
BTFSS f, b
BTFSC f, b

} → Conditional skip next instruction

Example :- INCFSZ 22, 1 → increment, then if = zero skip.

DECFSZ 23, 1 → decrement " " " "

BTFSC 24, 3 → if bit 3 in address 24 = 0 skip

- skip takes 2 cycles, if there is no skip → only 1 cycle
  - cycle = 1 Tinst.
- skip only one instruction.

Example :-

```
for(int x=50; x>0; x--){
        code
}
```

```
MOVLW 50
MOVWF 15  ~~> x in address 15
DECFSZ 15, 1
GOTO label
```

block

lecture 10

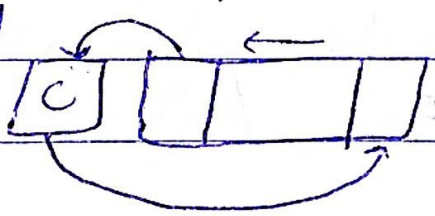
- Miscellaneous Instructions

- CLRF f → the value in address "f" becomes 0
- CLRW → " " " working register " "
- NOP → don't do anything
- BCF f, b → clear bit in certain address.
- BSF f, b → set bit in certain address

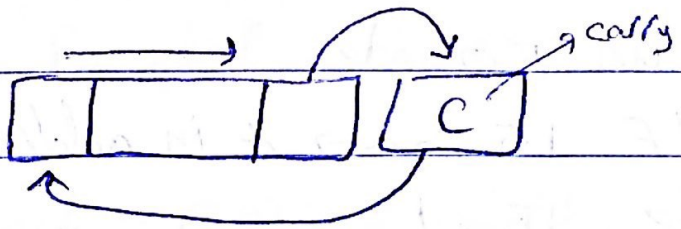
selecting banks

↳ to select bank1 → BSF STATUS 5
to select bank0 → BCF STATUS 5.

- CLRWDT → clear watchDog timer
- sleep → pic in sleep mode.
- RLF f, d

↳ 

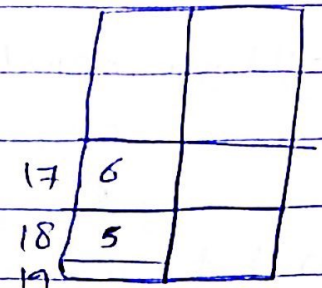
- RRF f, d

↳ 

- RLF → if C=0 → *2
↳ if C=1 → *2+1

- RRF → if C=0 → int div by 2.

example :- RLF 17, 1
RRF 18, 0
what are the final values of address 17 and 18 after executing the previous code.



[17] = 0000 0110
[18] = 0000 0101

[17] = 13
[18] = 5
W = 10

Example :- write code to multiply the value in address 19 by 8.

```
BCF STATUS C
RLF 19,1
BCF STATUS C
RLF 19,1
BCF STATUS C
RLF 19,1
```

Note :- andlw B'11110110' → clear bits 0, 3.
xorlw B'01010101' → complement the even position bits.

Remember:- 1) byte oriented file reg.

op, f, d
6 7 1 → # of bits

2) bit " " "

op f, b
4 7 3 → # of bits

3) Literal " " "

op k
6 8 → # of bits

4) control " " "

op k,
3 11 → # of bits.

- opcode must be unique for each instruction.
- if the opcode is of fixed length
  ↳ $\lceil \log_2 35 \rceil$ = 6bits
  ↳ # of instructions.

- Any assembler line have four elements :-
  1) label → at very begning of the line
  2) instruction memory → 35 instructions.
  3) operands → according to the instructions
  4) comments.

* label → once defined, I can use it as operand and its value = the address of first instruction comes after it.
  * label is case sensitive
  ↳ ex:- call label → 11-bits
  ≡ label (instruction) → 14-bits

- Comment starts with (;)
  - ; comment
  - can be stand alone in a line.
  - comments are skipped by the compiler (doesn't convert it to machine code).

- Values bases:- (operands)
  1) Decimal
     - D'value' / d'value'
  2) binary
     - B'value' / b'value'
  3) octal
     - O'value' / O'value'
  4) ASCII
     - ' ' / 'A' / 'a'
  5) Hexa.
     - H'value' / h'value' / 0x'value' / no symbol

default base

- if the hex. value starts with letter it most be preceded by either "H" or '0x' / o.w the compiler will deal with at as a label.

lecture 11

- Assembler directives :- gives information to the assembler at compilation time. then they are discarded.
  - 1) #include :- allows the assembler to open a library file and use anything in it. (#include 'PIC16F84A.INC')
    - you can use all registers and flags by their name.
      ex :- BSF STATUS, C
  - 2) EQU :- to define constant and can be used as you want. ex:- STATUS EQU 03 (03 instead of STATUS) / RP0 EQU 5
    - from datasheet.
  - 3) ORG :- Tells the assembler where to store the next instruction after compilation.

```
ORG 0005
inst 1
inst 2
inst 3
```

→ Assembler → instruction 1 stored in 5
instruction 2 stored in 6
" 3 stored in 7.

- at the begning of the code use (ORG 0000) to start the code from the first instruction.
- at the begning of the interrupt code use (ORG 0004), but its not needed except if you enabled the interrupts / by default the interrupts are disabled.

4) Cblock → define a block of variables with sequential values

↳ ex:- Cblock 20
Var 1
Var 2
Var 3

→ Var 1 = 20
// 2 = 21
// 3 = 22

≡ Var, EQU 20
Var, EQU 21
Var, EQ 22

5) end : tells the assembler not to compile any line after this point

Sample program / slide 24:-

GoTo start → ORG 0000
1) MOVF 31,0 (move from address 31 to the working register)
2) ADDWF 45,0
3) ADDWF 47,0
4) MOVWF 22
end

. if the interrupt is enable →
ORG 0004
GoTo interrupt vector

. To know the bank → Convert the addresses to binary.

slide 25 → we have 8 instructions, each instruction is 14-bits

. Done GOTO Done → 14 bits
↳ opcode (3-bits) ↳ k (11-bits)
↳ address of next instruction after done = A

from data-sheet.

Sample Program 2 slide 25 → 10 instruction = 140 bits

- swapping between 2 variables → needs 3 variables

```
        ORG 0000
        goto START
        bcf STATUS, RP0  → bank selection
        movf 0x33, 0
        movf 0x22
        movf 0x11, 0
        movwf 0x33
        movf 0x22, 0
        movwf 0x11
Done    GoTo Done  → to keep the PIC always
        end          running.
```

```
        ORG 05  → no address
        inst 1  → address 5
        ORG 09  → no address
L3      inst 6  → address 9
        inst 7  → address 10
L5      ORG 50  → no address
L6      inst 8  → address 50
L7      inst 9  → address 51   (Goto L5)
```

Note :- ORG deals with Program memory.

Note :- GoTo takes 2 cycles to be executed.

50

L3 → 09

Chapter 5 | lecture 12

if address in 27 greater than 0

ex: if [27] > 9

block 1

else

block 2

```
        MOVLW   9
        SUBWF   27,0
        BTFSC   STATUS,C
        GOTO    block1
        GOTO    block2
```

skip if [27] < 9

block1 ≡

→ GOTO NEXT → if block 1 is executed skip block 2

block2 ≡

NEXT

---

ex: for (int i=40, i>0, i--)

block1

```
Counter EQU 21
MOVLW   D'90'
MOVWF   Counter
block
  ≡
  ≡
DECFSZ  Counter,1   → if 0 → infinit loop
GOTO    block1
```

ex:- for (int Counter = 7 ; Counter < 100 ; Counter ++)

block 2

```
Counter EQU 22
MOVLW 7          } 7 iterations
MOVWF Counter
block2 =
       =
INCF  Counter, 1
MOVLW D'100'
SUBWF Counter, 0
BTFSS STATUS, Z
  GOTO block1
```

```
MOVLW 0
MOVWF Counter        } 256 iterations → max.
loop =
     =
INCFSZ Counter, 1    Note:- 0 → dec → FF(255)
GOTO loop
```

```
for(int i = 7 ; i > 0 ; i--)
    for (int j = 15 ; j > 0 ; j--)
MOVLW D'15'          decfsz Count, 1
MOVWF Count1         GOTO loop
MOVLW D'7'           decfsz Count2, 1
MOVWF Count2         GOTO loop
loop = block1
```

Example1:- [11] + [22]

if C=0 result in 33

else result in 44

```
    MOVF 11,0
    ADDWF 22,0
    BTFSS STATUS,C
    GOTO Clear
    MOVWF 44
    GOTO DONE
Clear MOVWF 33
Done GOTO Done
```

Note:- (if-else statments need 2-3 GoTo).

Note:- any loop iterates 256 time max.

b because we have 8-bits variables

Example2:-

```
CountH EQU
CountL EQU
```


```
loop MOVF CountL,1
     BTFSS STATUS,Z
     GOTO DecL
     MOVF CountH,1
     BTFSC STATUS,Z
     GOTO DONE
```

```
     DECF CountH,1
DecL DECF CountL,1
     GOTO Loop
Done GOTO Done
```

- Subroutine:- code, starts with label, and ends with Return, return lw → executed when invoked by (call label).

Call instruction

PC Push → stack PC ← address of 1st instruction in sub.

- ~~Return~~ .retlw K - Return

  ↳ stack Pop → PC

1) movlw ... Return

- To pass parameters to the subroutine store them in the data memory and make the function works on them.

- Delay :-

1) by software delay: add NOP instructions as much as you want. each Nop takes 1 Tinst.

2) by Hardware delay : timer(0)

Example:- Assume Fosc = 400 Hz, delay = 0.5

loop Turn on LED.

delay = 4/Fosc * #of Tinst = (Tinst * #Tinst)

0.5 = 4/400 * Tinst

# of NOP = 50

Turn off LED

Goto loop

- instead of using number of (NOP) → use loops.

Delay = 4/Fosc * #of Tinst

↳ known by tracing the code.

↳ 1) intialization
2) all iterations except the last one
3) last iteration.

Example 1 (slides) → Delay = $5X10^{-6} X [\overbrace{(1+1)}^{mov\ mov} + 199 \times \overbrace{(1+1+1+2)}^{nop\ nop\ decf\ goto} + \overbrace{(1+1+2)}^{nop\ nop\ decf}]$

= 5.005 ms.

call ↑ return ↑

↳ if it is a sub. (5.005 +2+2)*5

= 2.025 ms

- modify the code to give exactly 4 ms

↳ start from the counter (assume counter = X)

delay = Tinst X # of Tinst

$4X10^{-3} = 4/800 X$ #of Tinst

#of Tinst = 800 = (1+1) + (X-1) x (1+1+1+2) + (1+1+2)

X = 159.8

so Counter = 159 (the delay is not exactly 4 now, because # of Tinst = 796) → so add 4 NOP before the loop.

Ex:- write sub. to give exactly 10ms including the call inst. assuming Fosc = 1MHz

$$T_{inst} = \frac{4}{1MHz} = 4\mu s$$

delay = Tinst x # of Tinst

$10 \times 10^{-3} = 4 \times 10^{-6} \times$ # of Tinst

# of Tinst = 2500

```
Sub   Movlw X
      Movwf Counter

Label Nop
      Nop
      Nop
      Nop
      decfsz Counter,1
      GoTo Label
      return
```

2500 = 2+2+7 x(x-1) +8

x = 356 → large number → add Nop

- Literal → value in the instruction
- Direct → " " " data memory, address in the instruction
- indirect → " " " data memory, address in the FSR.

- Nested loop :-
  1) initialization
  2) first external iteration
     { all internal except last.
       last internal iteration.
  3) all external except first and last
     {
  4) last external iteration
     {

- example slide 23 :

delay = 10ms / Fosc = 4MHz

delay = 4/Fosc * # of Tinst

$10 \times 10^{-3} = 4/4 \times 10^{-6} \times$ # of Tinst

# of Tinst = 10,000

call nop mov mov mov mov

2 + (1+1+1+1+1)

dec goto dec dec goto

+ 249 * (1+2) + (2+1+2) + 11 * [255 * 3 + 5] + 255 * 3 + (2 + 2 + 2)

example:- write code to clear the data memory of addresses 0x10 → 0x4F (64 location)

```
        CLRF  10
        CLRF  11
64      CLRF  12
times    .
         .
         .
        CLRF  4F
```

so use loop

```
        movlw D'64'
        movwf counter
loop    clrf  counter
        .
        .
        decfsz counter
        GoTo  loop
```

error → here we need indirect addressing because we are dec the address.

Q:- Why do we need indirect address?

answ:- when we need to manipulate the address of data memory (↳ (++, --, +=))

1) write the address in FSR
2) replace operand by either 00 or INDF

```
        movlw D'64'
        movwf counter
        movlw 10
        movwf FSR
loop    clrf  INDF
        INCF  FSR,1
        decfsz counter
        Goto  loop
```

example:- write code to complement value of address 17 using indirect addressing.

```
MOVLW 17
MOVWF FSR
COMF INDF,1
```

COMF 17,1 ⇐

- we can't write value in INDF
  - it is not real register. it is just a Pointer

- how to choose the bank in indirect addressing?

example:- write code to copy the value in address 0x83 to the W-Reg using both direct and indirect addressing

indirect

```
BSF STATUS, RP0
MOVF 02,0
MOVLW 0x82
MOVWF FSR
MOVF INDF,0
```

~~direct~~ example slide 27 direct

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

- Array is an example of a look-up table

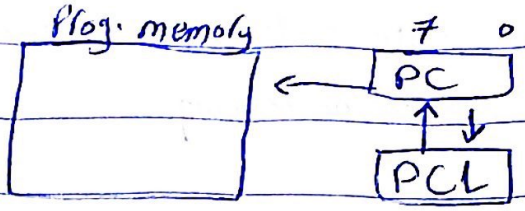
```
int a[6] = {0,1,4,9,16,25};
```

movlw 2 = cout <<a[2]
Call lookup

- we have instruction (retlw = # of elements) →

↳ PC low

```
ADDWF PCL, 1
0) retlw 0
2) retlw 1
3) retlw 4
4) retlw 9
5) retlw 16
6) retlw 25
```



modify the array to give squares of number from (3-8)

```
Lookup  movwf  Temp
        movlw  3
        SUBWF  Temp, 0
        ADDWF  PCL, 1
        retlw  9
        retlw  D'16'
        "      D'25'
        "      D'36'
        "      D'49'
        "      D'64'
```

ex- movlw 5
call lookup

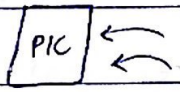
Chapter 6 — Lecture 15

interrupt:- is a code (starts at 0004, ends with RETFIE) is executed when called by hardware event

Note :- (0004 → interrupt vector)

- interrupts have higher priority than the main code.

- Types of interrupt :-

1) external 

2) internal 

3) maskable :- Can be disabled. (In PIC all interrupts are maskable, because they are disabled by default).

4) none maskable :- Can't be disabled.

- for an interrupt to happen (PC → 0004) (slide 6)

1) Global Interrupt enable (GIE) = 1
2) Local enable for any of the 4 interrupt types } → by S.W (set and cleared)
3) Flag = 1 for an enabled interrupt. → by hardware event, cleared by software.

- Interrupt flag can be set regardless of enable bits.
- when R = 1 → reset → Flag = 0.

- There are 4 types of interrupt

1) Timer 0 overflow interrupt → when timer 0 delay is over
↳ T0IF = 1
(when T0IE = 1, IOIE = 1, GIE = 1) ↳ flag by hardware.
↳ by h.w ↳ by s.w
↳ then interrupt occurs.

2) External interrupt → when positive edge or negative edge on RB0 pin of PortB.
↳ INTE = 1
(when INTE = 1, GIE = 1)
↳ interrupt

3) Port B change → any change on the upper 4 bits of Port B.
↳ RBIF = 1
when (GIE = 1, RBIE = 1) → interrupt

4) EEPROM write complete
↳ when copy to EEPROM is over → EEIF = 1
(when GIE = 1, EEIE = 1)
↳ interrupt

- The PIC will wake up from sleep, if 1 of the flags = 1 while its corresponding enable bit = 1 regardless of the GIE value
- To deal with the interrupts:-
  - 1) GIE bit (1-bit) } → set, clear by s.w
  - 2) local enables (4-bits) }
  - 3) interrupt flags (4-bits) } set by h.w cleared by software
  - 4) for ↑ ↓ in extend interrupt (1-bit) }

  1 - 2 - 3 → in INTCON except EEIF in EECON1

  4 - in option register

Slide 11 — lecture 16

- when interrupt occurs, after saving the PC on stack, the GIE is cleared To prevent any other interrupt from interrupting the CPU.

- Why the interrupt should end with RETFIE ? because this instruction set the GIE by default (GIE = 1)

- the GIE is set only once by us at the begning of the interrupt code.

- What shall I do to deal with interrupts ?
  1) start interrupt code at address 0004
  2) ends the interrupt code with (RETFIE)
  3) enable GIE
  4) enable local enable (one of them only)
  5) if you deal with (PORT B change interrupt) clear the flag at the begning. (RBIF Flag)
  6) you should clear the flag before RETFIE (the flag which you used)

Note :- we can clear the flags, but we can't set them (we don't need to set them in the code)

Example slide 13 :-

```
        ORG 0000 → reset vector address
        GOTO START
        ORG 0004 → interrupt vector address.
        GOTO ISR
START   BSF INTCON.GIE → set the GIE (=1)
        BSF INTCON, INTE
        BSF STATUS, RP0 → switching to bank1
        BSF OPTION_REG,6 → rising edge
        BCF STATUS, RP0 → switching to bank0
        CLRW → clear the working register
loop    ADDWF 0X0A,0
        GOTO loop. → until interrupt occurs
ISR     MOVWF 0X10 → when interrupt
        CLRW
        BCF INTCON, INTF
        retfie → return from the ISR
        end
```

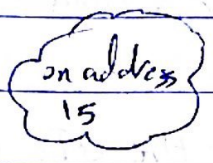
~~another~~ we need context saving iff the interrupt change a value needed by the main code.

1) ~~Any Reg (file register) except W-Reg and status~~ →

- How to save context :-

1) any file register except w-reg and STATUS

```
ISR MOVF (15),0      ← example.
    MOVWF Temp
```

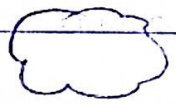



```
    MOVF Temp,0
    MOVF 15
    retfie
```

We can't deal with the STATUS REG. like this because it may affect the Z-flag if STATUS = 0 ↳ so, use swap.

2) W-REG

```
ISR MOVWF Temp
```



```
    MOVF Temp,0
    RETFIE
```

Note:- according to the H.W, no priority between interrupts

3) STATUS REG.

```
ISR SWAPF STATUS,0
    MOVWF Temp
```

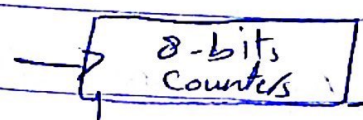
```
    SWAPF Temp,0
    MOVWF STATUS
    retfie.
```

Example :- write a code that when an external interrupt occurs, multiply value in address 15 by 4. and when Timer0 overflow interrupt then subtract 9 from address 15. (on the falling edge)

```
#include PIC16F84A.INC
        ORG 0000
        GOTO START
        ORG 0004
        GOTO ISR
START   BSF INTCON, GIE
        BSF INTCON, T0IE
        BSF INTCON, INTE
        switch to bank
        BCF OPTION_REG, 6
        switch to bank 1
loop    GOTO loop
ISR     BTFSC INTCON, INTF
        GOTO external
        BTFSC INTCON, T0IF
        GOTO timer0
        retfie
external BCF STATUS, C
        RLF 0x15, 1
        BCF STATUS, C
        RLF 0x15, 1
        retfie
timer0  MOVLW 4
        SUBWF 15, 1
        BCF INTCON T0IF
        retfie
        end
```

→ to Prevent the CPU from executing the interrupt code without an interrupt occurs.

Mid



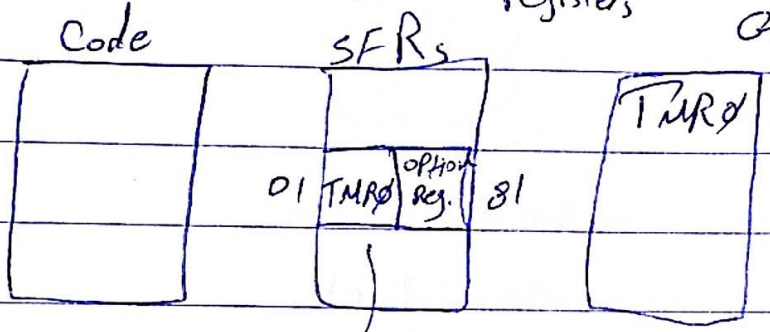
on edge

Counts 0,1,2,3 ... 255 → 0

on timer0

T0IF = 1
T0IE = 1
GIE = 1

special function registers



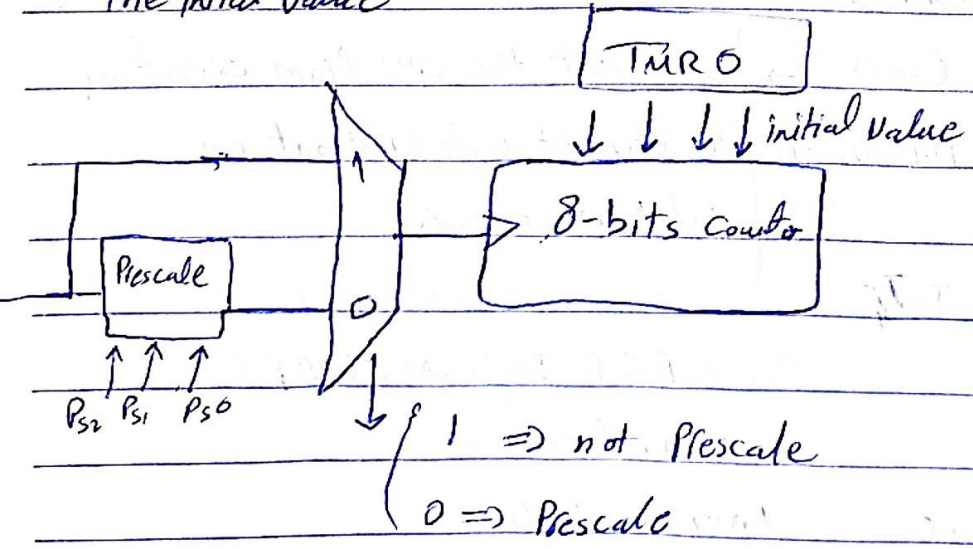
to control the Timer0

TMR0 → To control the initial value

option Reg.

$PS2\,PS1\,PS0 + 1$

Prescale = 2



1 ⇒ not Prescale

0 ⇒ Prescale

ex: Prescale on 16
↳ frequency div
then count after
16 cycle
↓
Prescale → to increase
the delay

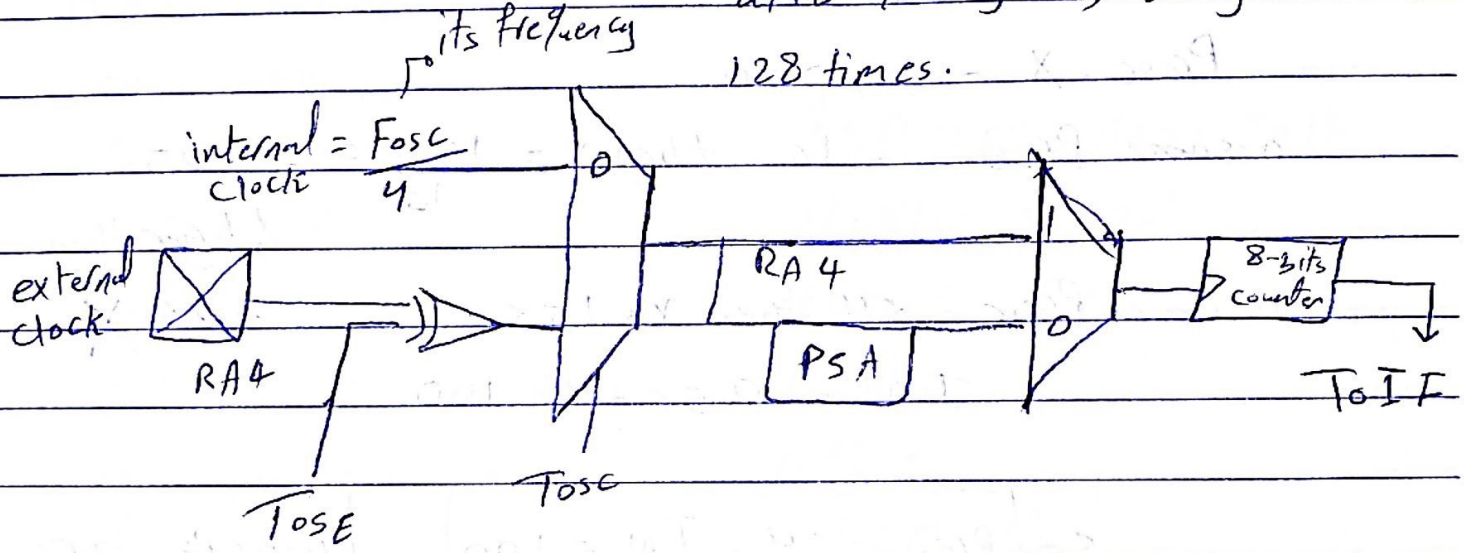
Example:- assume PSA = 1 and $P_{S2}P_{S1}P_{S0} = 110$

↳ no prescale

if PSA = 0 and $P_{S2}P_{S1}P_{S0} = 110$

↳ Prescale = $2^{6+1} = 128$

↳ this means instead of counting after each cycle, count after 128 cycle, delay increases 128 times.



↳ if 1 → invert the clock.

(ex. instead of counting on +ve edge, count on -ve edge)

(PSA, P_{S2}, P_{S1}, P_{S0}, TOSE, TOSC) → these bits are in the option register

Delay of Timer0 = $\frac{4}{Fosc}$ × Prescale × (256 − IN)

$\frac{4}{Fosc}$: Cycle length

(256 − IN) : # of cycles

Example:- write code to increment the value in address 18 every 10 ms, use timer0 and interrupt

Assume Fosc = 4 MHz

$$\text{Delay} = \frac{4}{Fosc} \times \text{Prescale} \times (256 - IN)$$

$$10ms = \frac{4}{4\times10^{-6}} \times Presc \times (X)$$

$$Presc \cdot X = 10{,}000$$

assume Presc = 16 then $X = \frac{10000}{16} = 625$ ↳ larger than 256 ✗

assume Presc = 64 then $X \approx 156$

$IN = 256 - X = 100$

→ So | Presc = 64, IN = 100 |

Note:- Pres = 256 at most

```
#include PIC16F84A.INC
        ORG 0000
        GOTO START
        ORG 0004
        GOTO ISR
START   BSF INTCON,GIE
        BSF INTCON,T0IE
        MOVLW D'100' → IN
        MOVWF TMR0
        MOVLW B'XXOX 0101'   internal, Prescaled, 64
        MOVWF OPTION_REG
loop    GOTO loop → after 10ms exit the loop

ISR     BCF INTCON,T0IF
        INCF 18,1
        MOVLW D'100'   } reset the settings to 100ms delay
        MOVWF TMR0
        retfie
        end
```

bank switching → b1, b0

, in the previous example, assume delay = 20ms

$$20 \times 10^{-3} = \frac{4}{4 \times 10^6} \times Pres \times X$$

Pre = 128 → X = 156 → IN = 100

Counter = 50

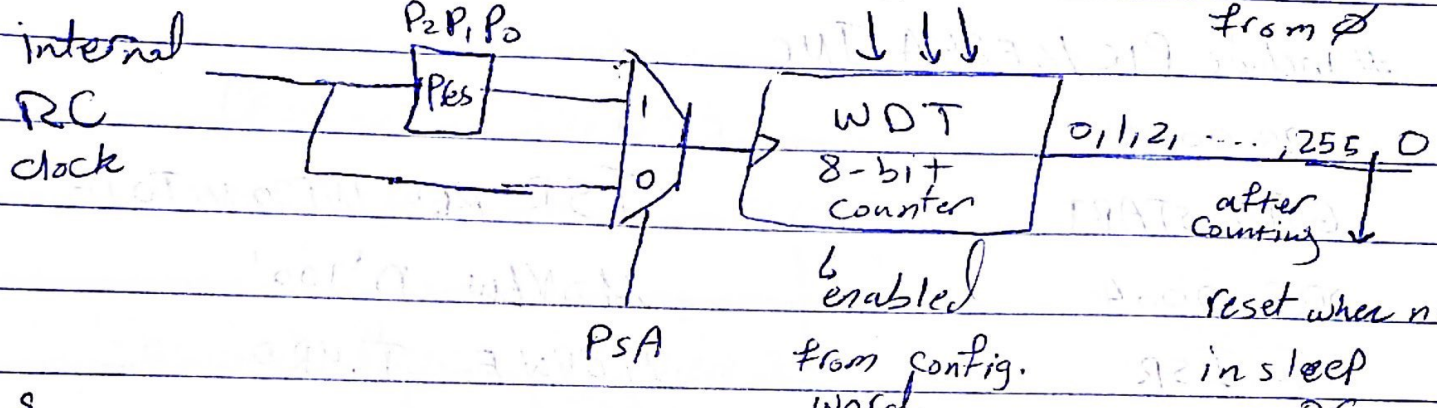
```
#include PIC16F84A.INC
    ORG 0000
    GOTO START
    ORG 0004
    GOTO ISR
START BSF INTCON, GIE
    BSF INTCON, T0IE
    MOVLW D'50'
    MOWF 20
    MOVLW D'100'
    MOWWF TMR0
    → b1
    MOVLW B'0XX0X011'
    MOVWF OPTION_REG
    → b0
Loop GOTO loop
```

```
ISR BCF INTCON, T0IF
    MOVLW D'100'
    MOVWF TMR0
    decfsz 20,1
    retfie
    INCF 18,1
    MOVLW D'50'
    MOVWF 20
    retfie
```

Note:- max delay → IN=0 / Pres = 256

- Presc of WDT = $2^{PS_2 PS_1 PS_0}$
  ↳ watch dog timer

- WDT is 8-bit counter CLRWDT → restart counting from ∅

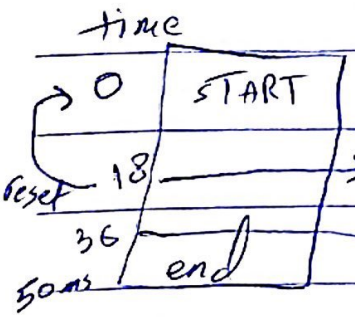


{ Cycle length = $\frac{18\,ms}{256}$

Delay of WDT = 18ms * prescale

Example:- assume WDT is enabled from config. word and PSA = ∅ (not Prescaled)

delay of WDT = 18ms * 1 = 18 ms (1 when no Presc)



→ clear WDT (to prevent reset)

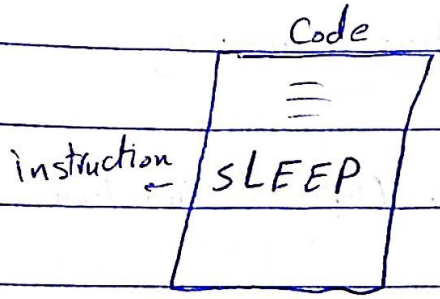
→ before 36ms clear WDT

Note:- max delay of WDT when Presc = 128

↳ 2.3 seconds

Lecture 18

- How to enter sleep mode ?



- How to exit from sleep mode ?

1) interrupt flag = 1 while its enabled, regardless of GIE.
   if GIE = 0 (continue from next instruction), GIE = 1 (~~reset~~ from interrupt)

2) $\overline{MCLR} = 0$ (return to 0000)

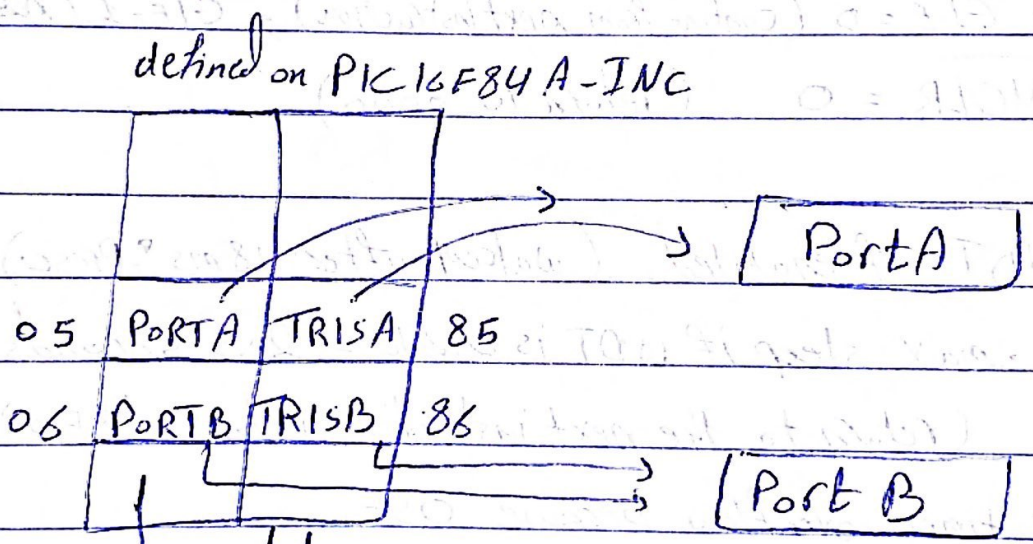
3) WDT if enabled (wakeup after 18ms * Presc)
   - max sleep if WDT is enable = 2.3 seconds
   (return to the next instruction after SLEEP)

except timed overflow because OSF.

Chapter 3

Note :- memory mapped I/O → each I/O has a memory location mapped to it.

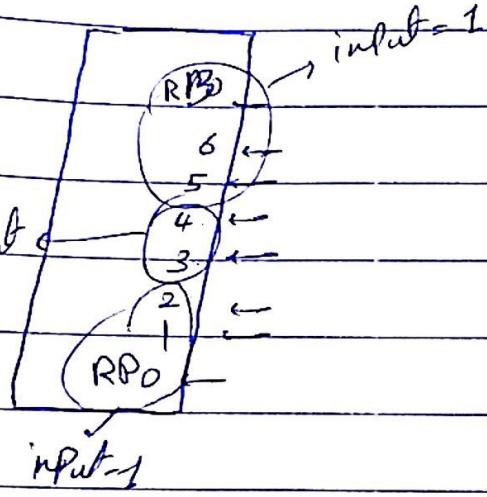
- to deal with any I/O device → through (speedal function registers)
- to connect any I/O device → on Port A (5-bits) and Port B (8-Bits), if each I/O device is single bit max # of devices = 13 due to 13 pins.



ex:- if pin 4 on Port B is input → TRIS(B)(4) = 1

- Pins are independent and half duplex.

example slide 7




```
BSF STATUS, RP0      , switch to bank 1 because TRISB
MOVLW B'10000111'
MOVWF TRISB
BCF STATUS, RP0
MOVF PORTB, 0
```

- we need to know what is connected to the input and its state to determine its final value. (connected switches or sensors on inputs)

Example 1 slide 8:-

```
BSF STATUS, RP0
MOVLW 0x00
MOVWF TRISB
BCF STATUS, RB0
MOVLW 0xAA
MOVWF PortB
```

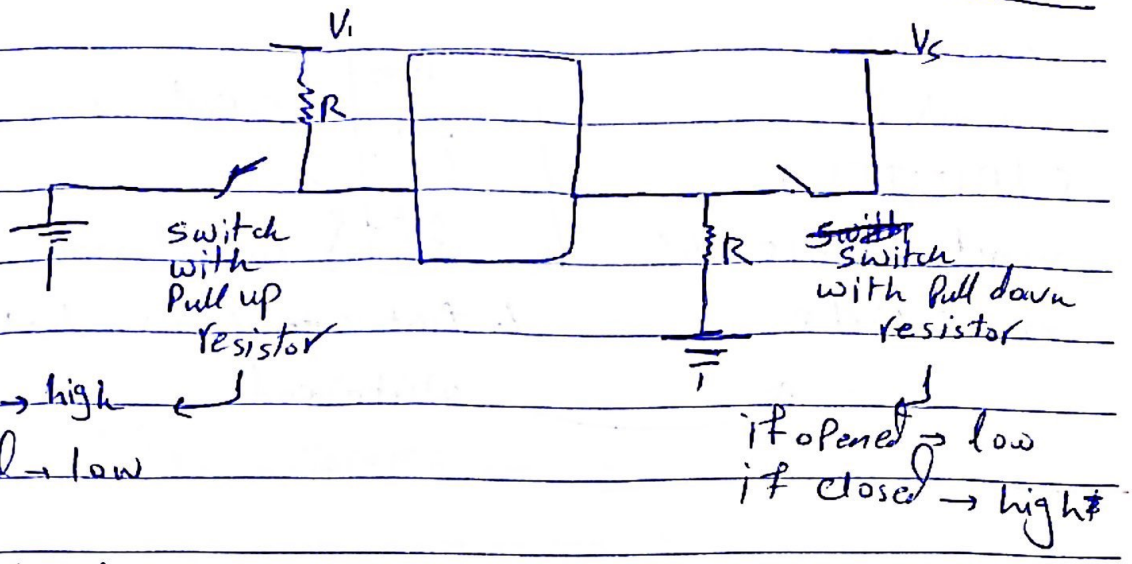
- Connect led on outputs for example, when 1 - led is on)

→

Example 2 slide 8 :-

```
BSF   STATUS, RP0
MOVLW B'XXX11111'
MOVWF TRISA
BCF   STATUS, RP0
MOVF  PortA, 0   → read
MOVWF 0X0D       → save
```

lecture 19



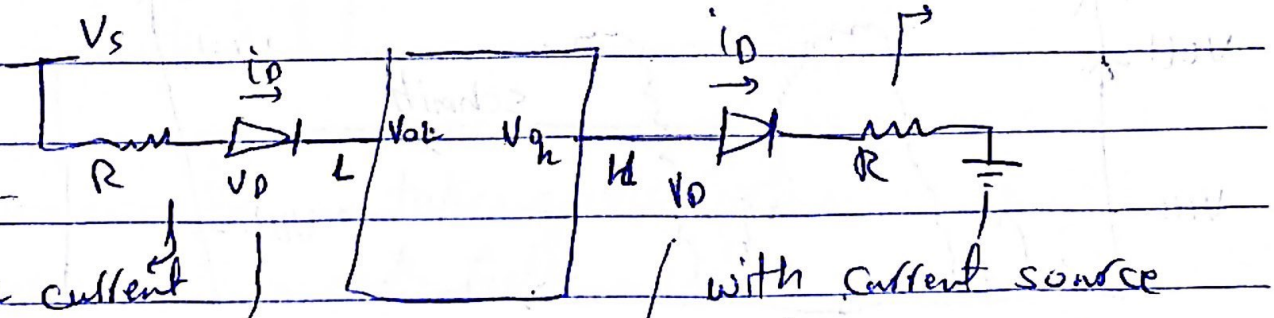
if open → high
if closed → low

if opened → low
if closed → high

- why to add resistor ?
  - 1) current limiting
  - 2) to inforce the ~~input~~ input to connect to the ground inorder to avoid noise.

Pull up resistor 0-5 → closed → low (0-3)
↳ closed → high (4-5)

Pull down resistor 6-8 → open → low



with current sink

$R = \frac{V_s - V_{OL} - I_D R_{oh}}{i_D}$

$V_{oh} = V_D + iR + I_D R_{oh}$

with current source ↳ the PIC is the source

$R = \frac{V_{on} - V_D}{i_D}$ (max R to make LED works) and Protect it

$-I_D R_{oh}$

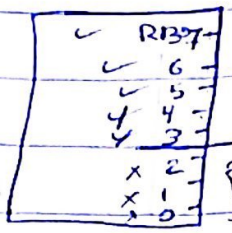
Example :-

→ bank 1

CLRF TRISB

movlw B'11100111'

movwf PortB



↳ first 3 are on, last 5 are off.

write code

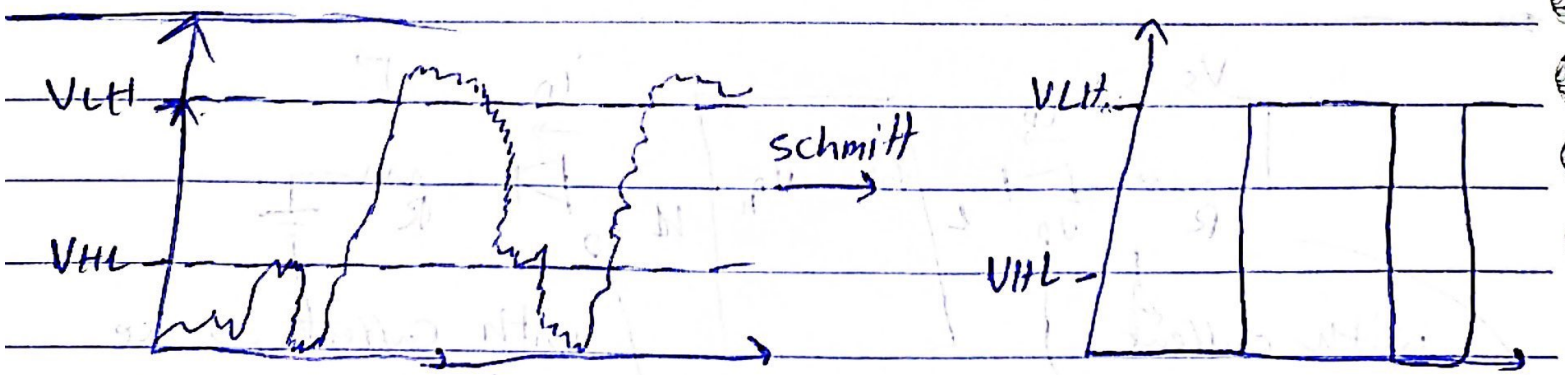
- schmitt Trigger input has two thresholds

VHL VLH

VHL → if initially the output is H, to convert into low, the input voltage must go below VHL

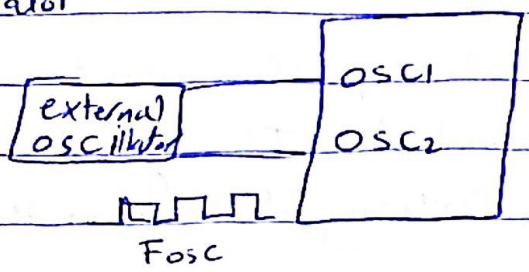
VLH → if initially the output is L, to convert into high, the input voltage must go above the VLH

(VHL < VLH)



advantages → 1) cancel noise 2) fast switching

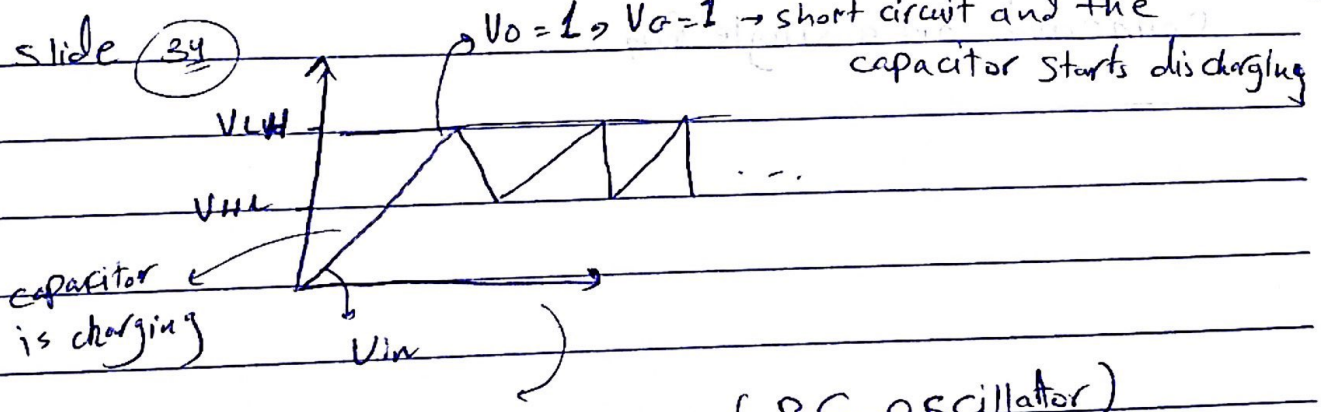
- The oscillator



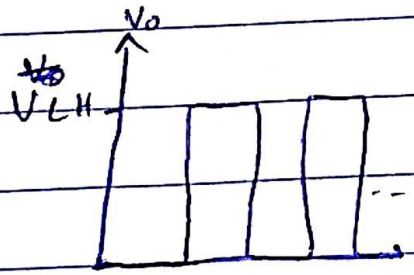
- 2 types of OSC.
  1) Resistor - capacitor
  2) Crystal or Ceramic.

remember :- in n-MosFet → if $V_G = H$ → short circuit.
& $V_G = L$ → open circuit.

slide (34)



(RC oscillator)



- crystal oscillator → takes small current → oscillation → generates electromagnetic field → high voltage → inverter (low) which decline the oscillation process and the electromagnetic field.

in RC oscillator → frequency depends on the RC constant (T)

in Crystal oscillator → frequency depends on the quartz

- in crystal (the crystal oscillator) is connected between the two pins (OSC1, OSC2), with 2 capacitors in order to cancel noise and fast stabilization

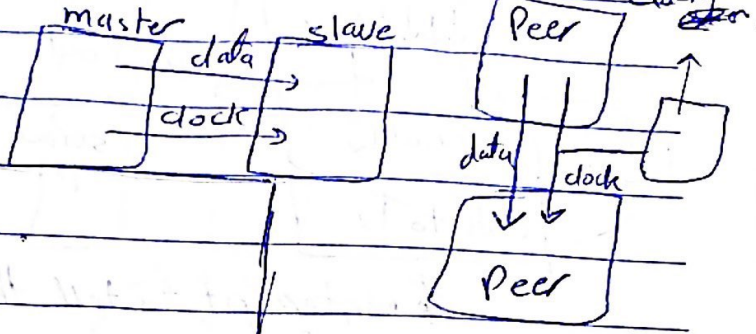
- in the RC oscillator, the n-mosfet and the schmitt trigger are inside the PIC, R and C are connected on a single Pin

Chapter 10

Lecture 20

- we have challenges with 2 solutions:-

1) synchronous
   - the clock is shared



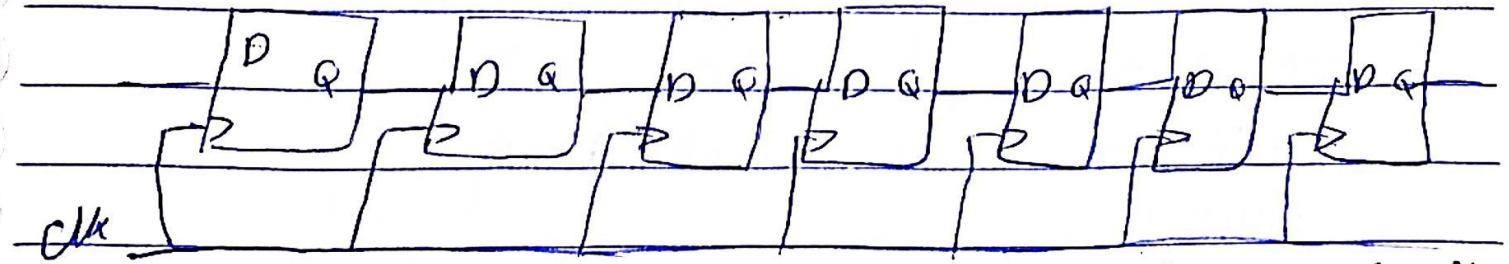
2) ~~Ay~~ Asynchronous

→ 2 clocks need to be ~~re~~ resynchronized through resynchronize after each word using stop and start (stop and start could be more than 1-bit)

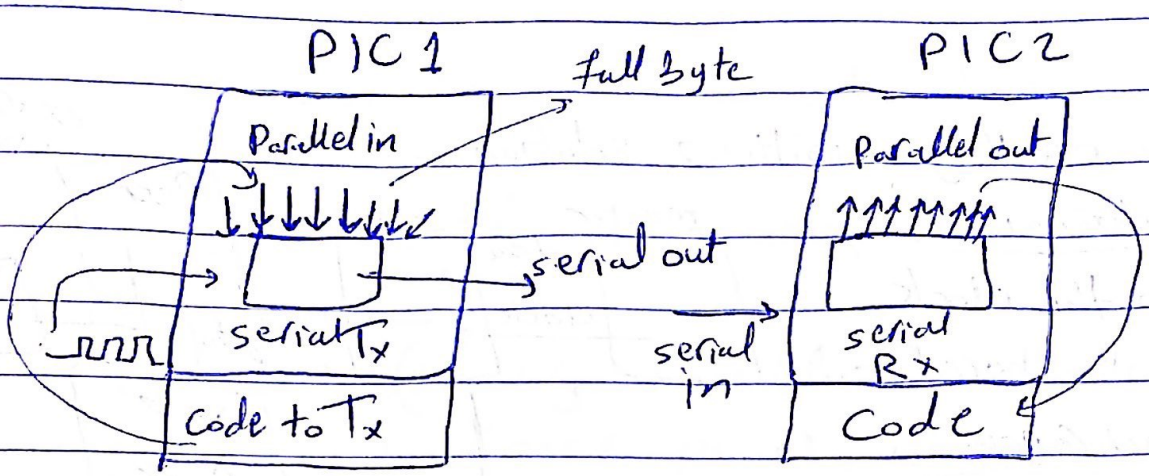
data in Asynchronous is formated according to protocol.

- synchronous is faster than Asynchronous due to less overhead, as well as its simpler.

- serial Port ≡ shift register → shift the bit on each cycle



input → output after 7 cycles (# of D-flipflops = 7)



- interrupt to tell that Transmisson is finished to send the next byte (TXIF = 1)
- " " " " the byte is received to read it (RCIF)

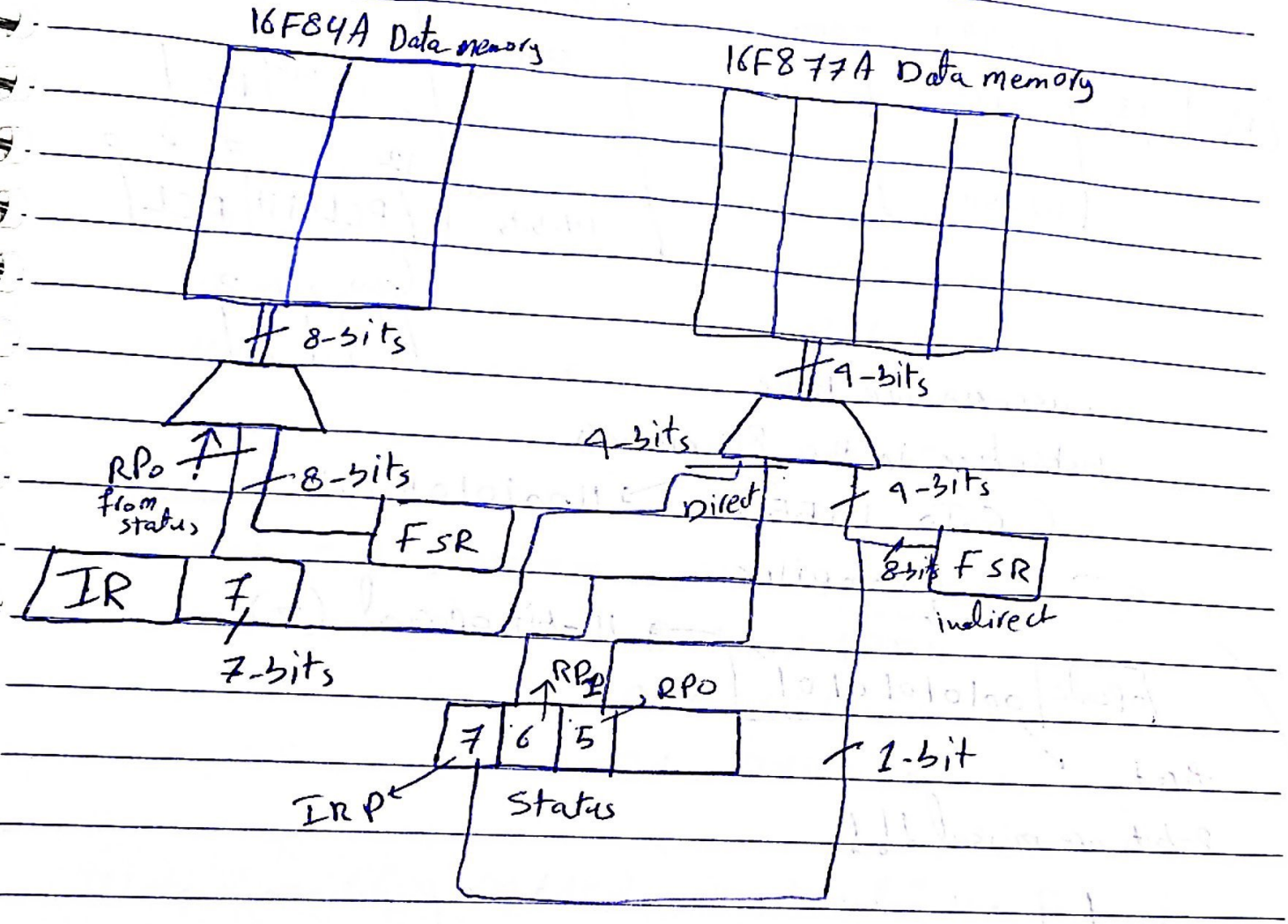
↓↓

These interrupts are related to 16F877A (it has more interrupt types)

it has the serial Port

Note. 16F877A is within the same family of the 16F84A

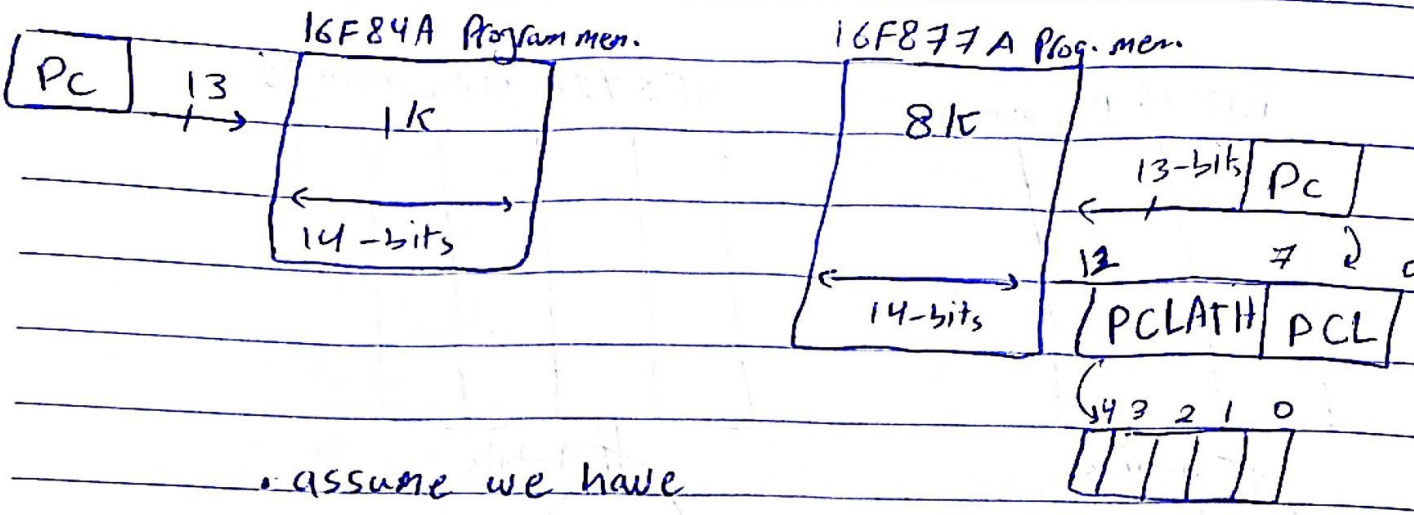
- Data memory in 16F84A → 2 banks
- Data memory in 16F877A → 4 banks



Note :- max size of FSR = 8-bits

example:- write code to read the value in address 197 to w-reg using both direct and indirect.

Direct →	Indirect →
BSF STATUS, 5	BSF STATUS, IRP
BSF STATUS, 6	Movlw 97
MOVF 17, 0	MOVWF FSR
	MOVF INDF, 0.



- assume we have instruction in the 8k memory

(GoTo 1955) → 11001010101 01

Compiler

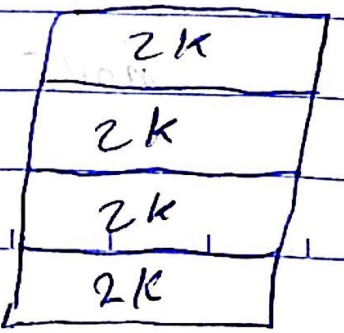
opcode | 00101010101 → 11-bit operand (k).

first 2-bits are missed!!!

↳ so, divide the programe memory in the 16F877A into Pages (horizantally) → we need 4-Pages each page is 2K → then we need (2-bits) to determine the Page. (bits 12, 11 in the PC)

↳ through the (PCLATH) (bits 4, 3 in the PCLATH)

Page selection:
BSF PCLATH,4
BSF PCLATH,3
GoTo 155



ex: read address 0x1AE

1 1 0 1 0 1 1 1 0

Direct

```
BSF STATUS,6
BSF STATUS,5
MOVF 0x2E,0x7FF
```

Indirect

```
BSF STATUS,7
MOVLW 0xAE
MOVWF FSR
MOVF INDF,0
```

- ex: Call 0X1955 → 11001 01010 (Page 5)

```
BSF PCLATH,4
BSF PCLATH,3
Call 0x155
```

Note: Modifying PCLATH will not modify the PC until (call) instructions, that's why it is called (PC LATch high)
Goto

Note: stack size (word size in the stack) = size of the PC.

- due to there are more I/O's in 16F877A, # of interrupt types increases.
- we have 11 new interrupt types.

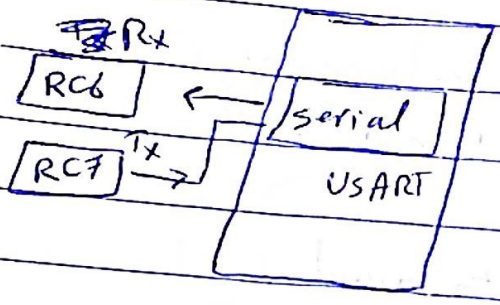
- 4 Conditions for interrupt to occure in 16F877A:
  1) GIE is enabled
  2) PEIE is enabled
  3) local enable = 1
  4) Flag is enabled

- To deal with IIOs in PIC 16F877A we have 9 special function registers.

1) TXSTA
2) TXREG
   ↳ mainly for Tx (transmission)
3) RCSTA
4) RCREG
   ↳ mainly for Rx (receiving)
5) INTCON
6) PTE1
7) PTR1
   ↳ for interrupts
8) SPBR
   ↳ for baud rate
9) TRISC
   ↳ determine RC6 and RC7 output, input

- TXSTA, RCSTA → for control
- TXREG, RCREG → for Data
- INTCON has GIE, PEIE
- PIE1 → has the new enables
- PIR1 → has the new flags.
- SPBR → same value for Tx and Rx.

Note:- PEIE instead of EEIE in INTCON



slide 22

- TSR Register can't be accessed, it doesn't have an address. (write in Tx register, and TXEN and SPEN should be enabled)
- the byte transfers from Tx register when the TSR Register is empty.
- When the byte is received in Rx Reg an interrupt occurs.

so

- To send a byte, write it in TXREG
- If TSR is empty, TXREG is moved to TSR.
  ⇒ TXIF = 1, if GIE, PEIE and TXIE are = 1
  ↳ interrupt
- transmission to start ⇒ 2 enables: SPEN and TXEN.
- 1st is LSB, and Last is MSB
- baud rate = F(SPBRG, BRGH, FOSC)
- When TSR is empty ⇒ TRMT = 1
- if TX9 = 1 ⇒ the serial Port will send additional 9th-bit with each word. usually Parity. →
  (stored TX9D)

- TXIF = is read only = 1 when TXREG is empty
  // // // = 0 when TXREG is full.

- we don't write directly on TSR, to avoid

slide 27 lecture 22

OERR = 1 : when the stop bit of 3rd word is received before reading the previous 2 bytes
→ 1) 3rd byte is lost
2) reception is stopped

FERR = 1 : if received stop bit = 0

- RCIF → is read only
  ↳ = 1 : when RCREG at least has 1 word
  ↳ = 0 : when RCREG is empty

MOVF RCREG, 0 (if RCREG != 1, still)
↳ MOVF RCREG, 0

OERR ⇒ read only, as long as it is equal to 1, there is no reception

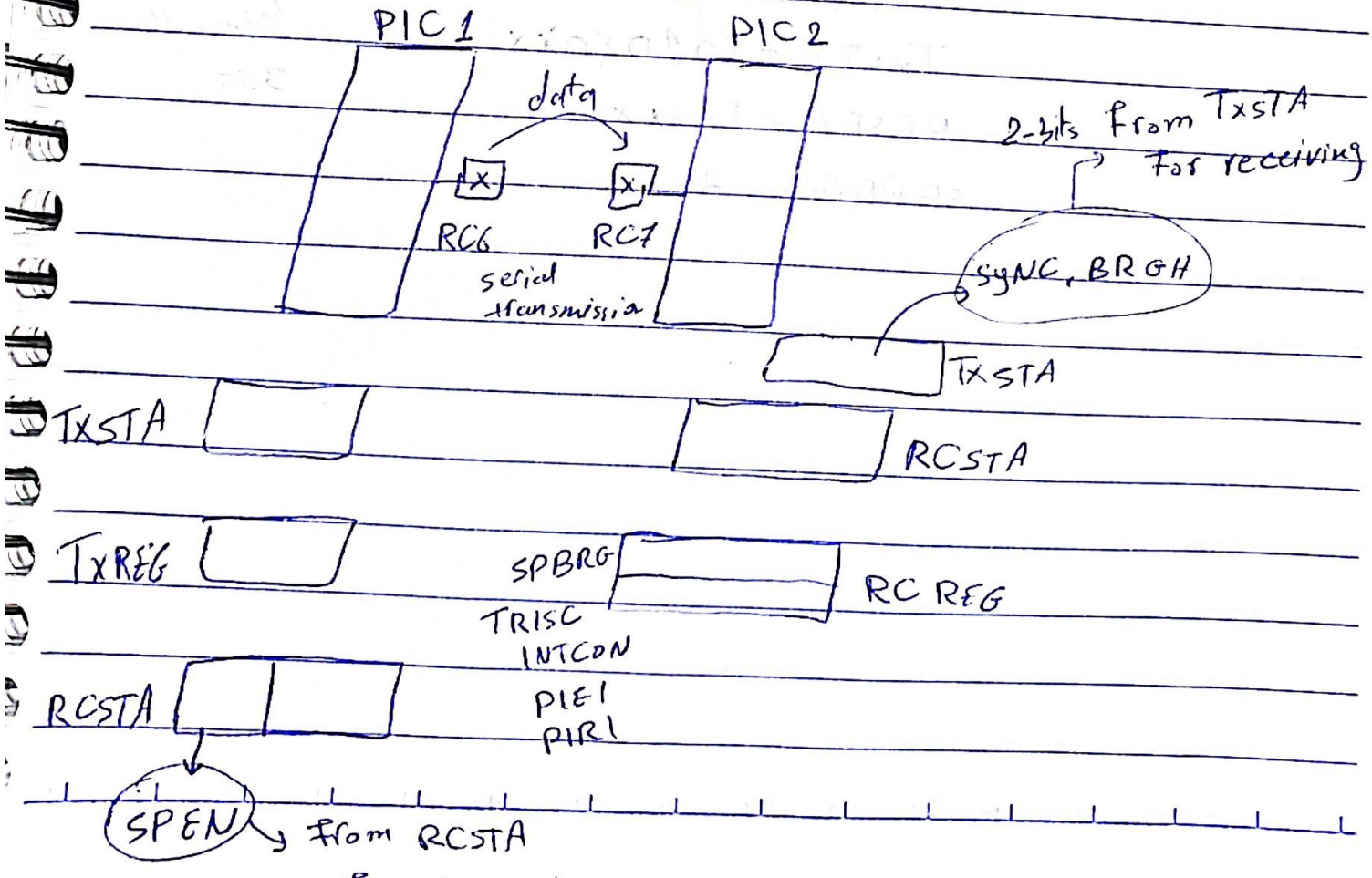
To solve it :-

1) read 2 bytes
2) CREN = 0 ⇒ OERR = 0
3) CREN = 1

avoid :-

GIE = 1
PEIE = 1
PCIE = 1

ISR movf RCREG, 0
=
retfie



/ /

- Baud rate in Asynchronous

↳ = Fosc

if BRGH=1 ← 16 + (1+SPBRG)

or

if BRGH=0 ← 64

64 → low speed

16 → high speed

→ used Polling (try it with interrupts)

Example slide 33:-

Fosc = 20MHz

baud rate = 9.6kbps , $9.6 = \frac{20MHz}{16\ (or\ 64)\ (1+SPBRG)}$

addresses 40, 41, 42

no Parity

TRISC 6=0 ← no Parity

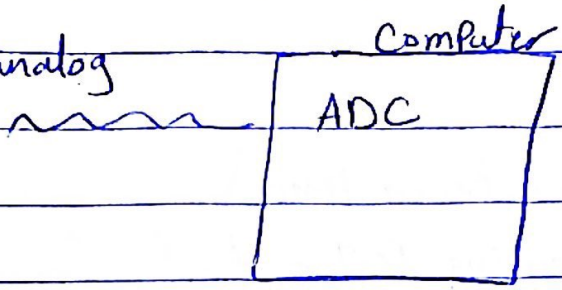
TxSTA = X010X0XX

RCSTA = 1XXXXXXX

SBPRG = 31

SPBRG = 31 ← less than 305

Chapter 11 | lecture 23



- Computer only understand the digital signal.

amplitude
- Analog :- takes infinite # of values in a range
- Digital : takes limited # of values in a range

time
- discrete :- read the value every specific period
- ~~Continuing~~ Continuous :- at any time, the value may change

10-bits ADC → 2^{10} levels (values)

↳ each sample is represented as 10-bits value

infinite analog | ADC | limited digital

(loss error due to converting from infinite to limited range)
→ so analog is more accurate.

- ADC is 2 steps :-

1) sampling :- Discrete in time

sampling freq = $\frac{1}{t_s}$

$f_s \uparrow \uparrow$ accuracy

2) Quantization :- discrete in amplitude

rounding ← number of bits (n) → 2^n digital levels.

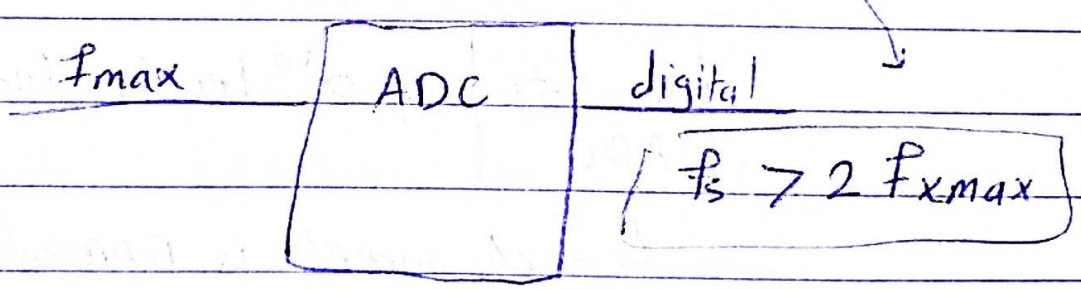
$\uparrow f_s$	⊕ more accuracy
	⊖ Power
$n \uparrow$	⊖ storage
	⊖ bandwidth
	⊖ cost
	⊖ speed

tradeoff

⊕ : advantages

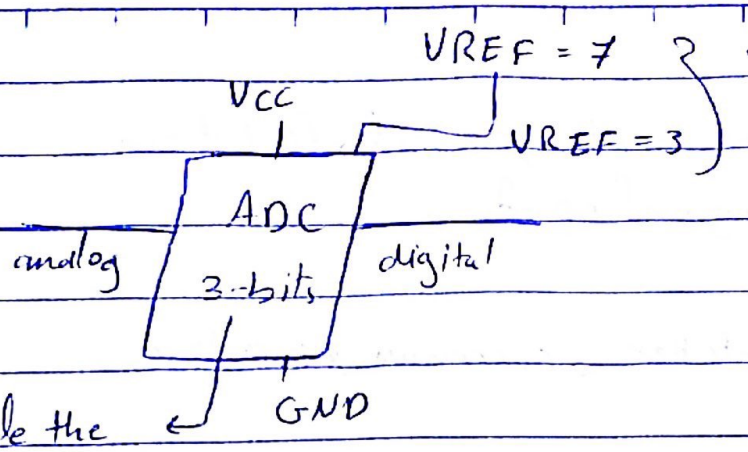
⊖ : disadvantage

So



if, $f_s < 2 f_{max}$,

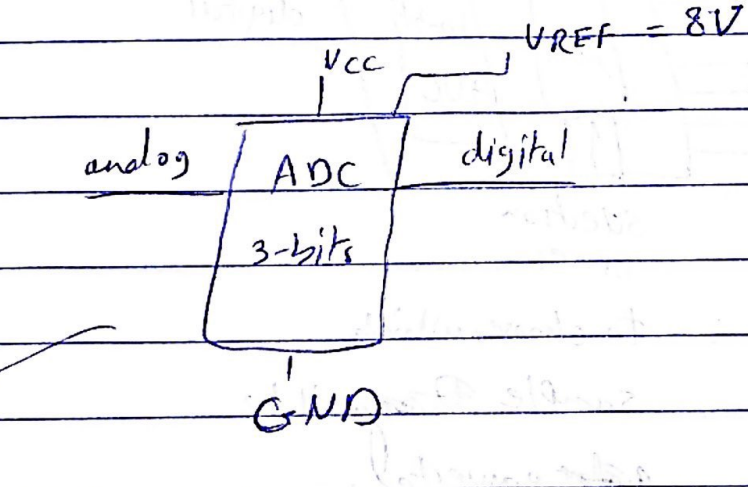
↳ aliasing in error conversion

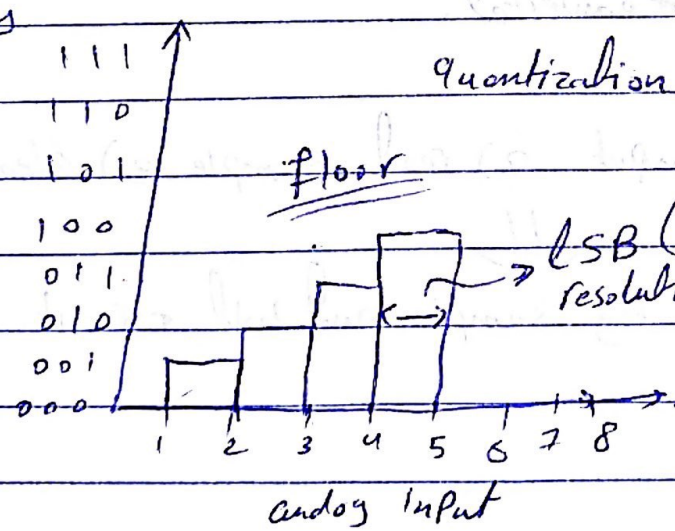


input between 7-3

divide the range into 8 levels / each analog input → 3-bits value

VRE :- range of acceptable input



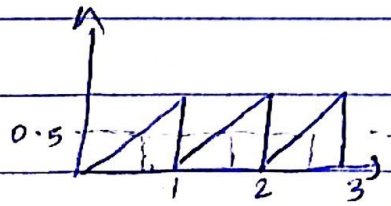


quantization characterestics graph

floor

LSB (lest sign. bit voltage)
resolution ↳ change in analog input that will change the output level

error = input - output



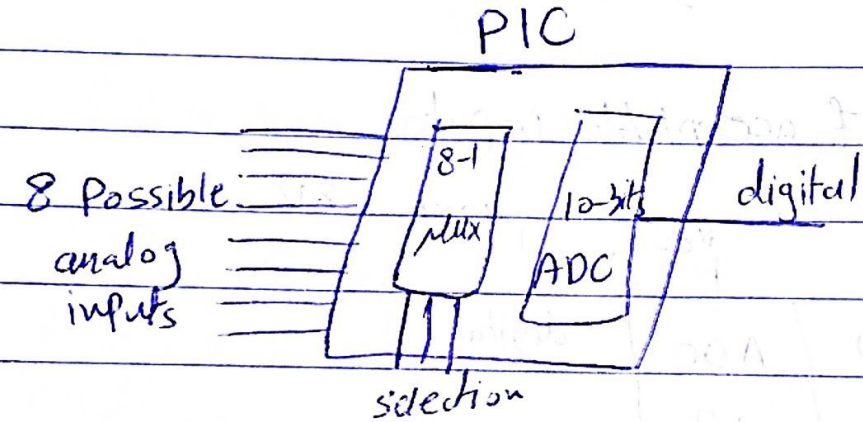
$$\cdot LSB = \frac{V_r}{2^n}$$

$$\text{max error} = \frac{1}{2} LSB = \frac{V_r}{2^n}$$

if used round function $\text{max error} = \frac{V_r}{2^{n+1}}$

- in ADC there is always a tradeoff between accuracy and speed.

- in microcontroller → successive ~~approx~~ ADC



to choose which sample ~~to be~~ will be ~~other~~ converted.

Sampling :-

1) hold input 2) read sample 3) release input.

⇓

by sample and hold circuit.

lecture 24

- sample and hold circuit → hold the input, read sample, convert it, release input (do this more and more

- in sample and hold circuit → Vc tracks Vs (when switch is closed)

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

after period of time $V_C \approx V_S$

- There is no path for discharging → so, if the switch is open → lose a little from the capacitance (voltage (Vc) is appro. fixed)

- to hold the input → open the switch
- to release the input → close the switch
- switch is controlled in the code.

Q) when to open the switch?

ans when $V_0 \approx V_{in}$

assume, I want to wait until $V_C = 0.9 V_S$ ⇒ 10% error

↳ then $t = 2.3\tau$

- max error $= \frac{1}{2} LSB = \frac{V_r}{2^{n+1}}$

$$V_0 = V_{in} - \frac{V_i}{2^{n+1}} \longrightarrow t = -\ln \frac{1}{2^{n+1}} * \tau$$

↑bits ↑accuracy ↓speed

* 10 bits ADC → each analog sample → 10 levels

PIC16F87xA

TRISA, TRISB → to determine inputs or outputs

↓ then

we have to determine if its digital or Analog through PCFG(3:0)

- Voltage references
  - ↳ internal (V_{DD} and V_{SS})
  - ↳ external (RA3 and RA2)

  } through PCFG (3:0)

external → more accurate if range of the input is not V_{SS} to V_{DD}

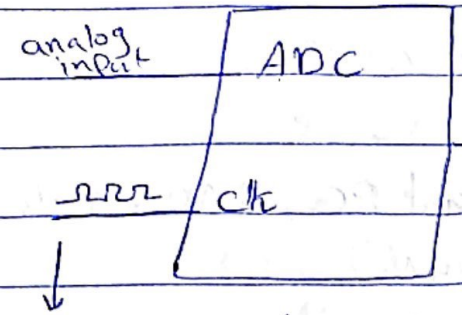
if the signal between (0-1) → RA3 = 1, RA2 = 0

Channel selection lines → determine the ~~Conversion~~ Conversion

Controlling the ADC:-

1) Switch on the ADC on AD on bit in ADCon0 (0)
   ↳ if no analog input → turn off the ADC, to prevent consuming Power.
2) Quantization speed = $(2+n)T_{AD}$

remember: Acq speed → $-\ln \frac{1}{2^{n+1}} \times \tau$



TAD → must be ≥ 1.6 μs (micro) for correct conv.

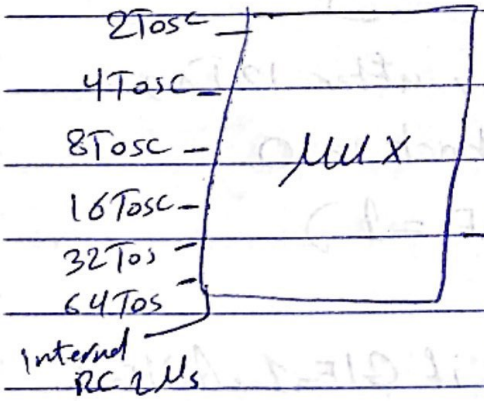
* X bits quantization needs (2+x) TAD cycles for correct quantization

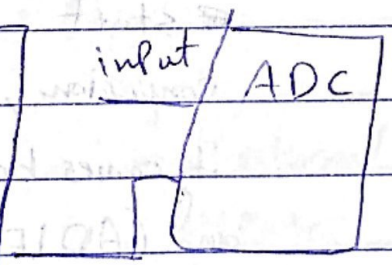
# of bits ⟹ # of bits + 2 TAD

↓

ex:- 10 bits ⟹ 12 TAD

if TAD = 1.6 → more accurate, faster.





* PIC is always 10-bits ADC

eg. if Fosc = 1MHz, what is the fastest quantization for PIC (10 bits ADC)?

$T_{OSC} = \frac{1}{1MHz} = 1\mu s$ → TAD = 2Tosc ⟹ 2 μs

⟹ quant. time = 12 * 2 μs = 24 μs

eg. IF FOSC = 100KHz

TOSC = 10µs

Choose internal RC, TAD = 2µs

⇒ TConv = 24µs

3) Configuring inputs and voltage references. → throug PCFG

Analog | digital | internal | external.

4) ~~channel~~ channel selection → channel selection lines (2:0)

5) start conversion (there is a bit called GO/$\overline{\text{DONE}}$ = 1)

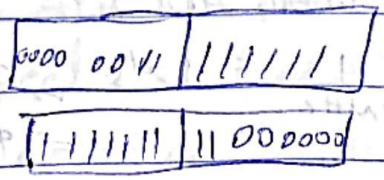
- GO/$\overline{\text{DONE}}$, set by software

↓ start conversion, after 12TAD it comes back to 0 and (ADIF = 1)

↓

interrupt if GIE=1, ADIE=1, PEIE=1

6) read result in ADRESH and ADRESL



→ right justified

→ left justified

if you're interested in 8 bit result

Controlling the ADC in the code

1) Configure ADC (steps from 1 to 4)
2) wait 7.6 T (Acq. time) → you have to write code for it
3) set Go/Done bit
4) wait 12 TAD (Quantization time) → you don't have to write it
5) Read the result. (read digital out)

slide 22 — lecture 25

- Single analog input, internal voltage reference?
  - PCFG = 1110
  - Analog input must be connected to RA0 (AN0)
  - CHS 2:0 = 000
  - C/R = 1/0

C/R
- # of analog inputs
- # of external voltage

- to wait 7.6 T (Acq. time) we have to write a code.

Temp coef + 7.6 . R . C + Constant time

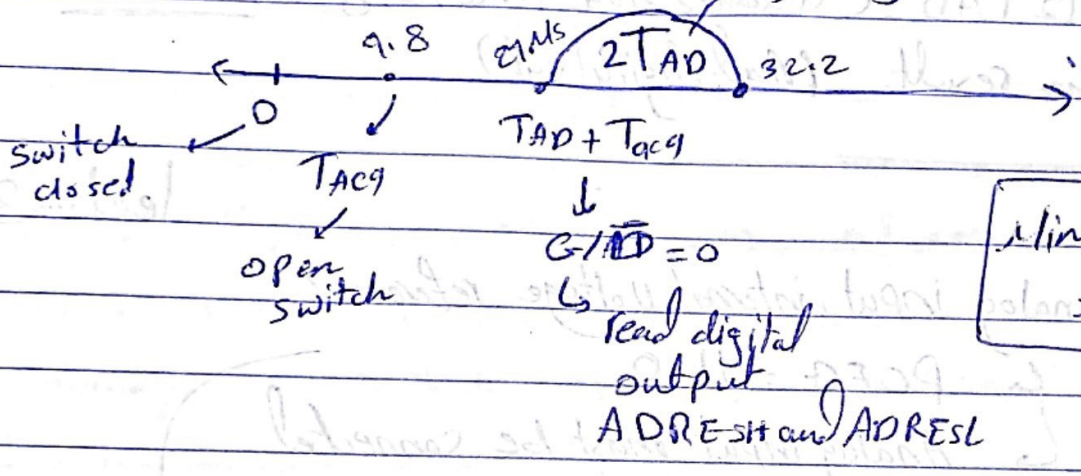
- Temp coef: 0.05 μs for each degree > 25°C
- R: Rs + Rss + Ric
- C: CHOLD

(Temp - 25) * 0.05 μs

If less than 25, consider it = 0

- $T_{acq} = 2\mu s + (Temp - 25) \times 0.05\mu s + 7.6 \times (R_S + R_{IC} + R_{SS}) \times C_{Hold}$

- $T_{quant} = 12 * T_{AD}$



minimum T_{AD} = 1.6

- each sample in repetitive sampling requires = 32.2µs

$$f_s = \frac{1}{32.2\mu s} = 31 kSps$$

f_{max} = 15.5kHz

ADC 31kSps

Example slide 33

- single analog input (RA Ø)
  - $T_{AD} = 8T_{OSC}$ → X because T_{AD} will be 0.4 → it should be ~~8Tosc~~ 32 Tosc
  - internal voltage ref.
  - $F_{OSC} = 20\,MHz$
  - $V_{DD} = 5$ Volt, $R_{SS} = 7\,k\Omega$, $R_{IC} = 1k\Omega$
  - $C_{Hold} = 120\,pF$
  - $R_S = 0$
  - Temp = 25

$$T_{acq} = 2\times10^{-6} + 7.6\times(1+7+0)\times10^{3}\times120\times10^{-12} + 6$$

$8T_{OSC} = 10\,\mu s$

ADCON Ø = (01)(000)(x1) → RAØ, ON, GIO

ADCON1 = (10) xx 1110

(10) → Right

TRISA = xxxx xxx1

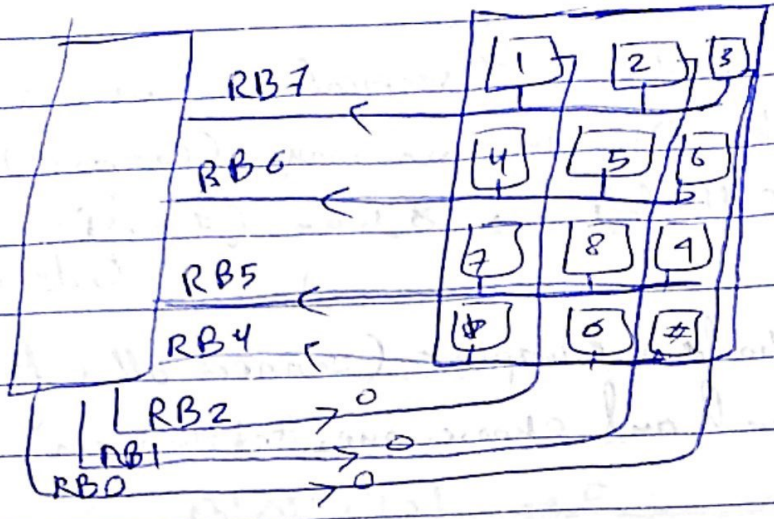
Chapter 8

Keypad :- is 12 push buttons connected to 7-pins :-

4 → each row is connected to pin o
3 → each column is connected to pin o

. To read the pressed button

1) read row
   ↳ 1) make row pins inputs
   2) make column pins outputs
   3) output zero on column pins
   4) read row pins.
2) read column
   1) make row pins output
   2) make column pins input
   3) output ø on row pins
   4) read column pins.



- each row and each column connected on a single pin

read row

row pins input → { movlw B'1111 0000' / movwf TRISB

column pins output

output 0 on column pin → CLRF PortB

read row → 1101 → r_2

read column → 110 → c_2 → 4

- To deal with PortB change, we need to clear the flag (RBIF) first.

- Press any button → interrupt. (Port B change)

3* row Index + column Index → Press <12

- 7 segments display → 7 segments each one is a LED connected to the PIC in two ways (common Anode and common Cathode) → 8 bins (7 + dp)
  ↳ decimal Point)

- Common cathode is simpler (connect all cathodes with the ground and choose one to turn on)

- to show a number on 7 segment display (6)

bank → movlw 00
movwf PortB → PortB is output

bank switching → movlw B'0111 1101'
movwf PortB.

---

2 seven segments display → 16 pins
4 seven segments display → 32 Pins → too much

assume we want to display #53

using 2, 7 segment displays

loop BSF PortA,1 → enable first segment (7 display)
BCF PortA,0 → disable second segment
movlw B'0110 1101' → 5
movwf PortB
delay 5ms
BCF PORTA,1
BSF PortA,0
Movlw B'0100 1111' → 3
MOVWF PortB
delay 5ms
GoTo loop

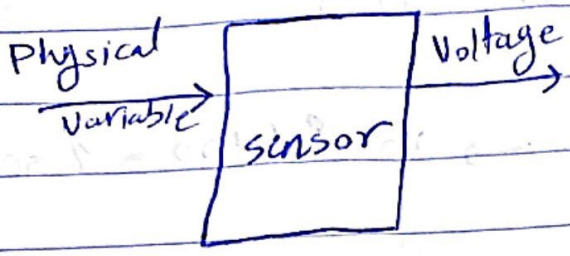
→ in one second 100 times
↳ we'll not notice that they turn off and on every 10ms

/ /

- to show 53 for one second

```
        movlw D'100'        → 10ms * 100 = 1 second
        movwF Counter
loop    BSF  PortA,1
        BCF  PortA,0
        movlw B'01101101' → 5
        movwF PortB       → PortB output
        delay 5ms
        BCF  PortA,1
        BSF  PortA,0
        movlw B'01001111' → 3
        movwF PortB
        delay 5ms
        decfsz Counter,1
        GoTo loop
```

Lecture 27



- sensor is connected to the PIC as an input, it could be analog or digital, we need to know the range.

- Ligh-dependent Resistors → type of sensors → analog

slide 28 $V_o = 5 * \frac{RLDR}{RLDR + 10 \times 10^{-3}}$

↳ depends on the illumination

- optical object Sensing composed of:
  1) IRLED → Vanode > Vcathod, ↳ IR light
  2) Photo transistor → if no light, it will be open circuit
  3) plastic housing → Passes only unseen ligh

Digital → used in digital doors usually

• Ultrasonic object Sensor → analog

sends ultrasonic pulses → receives ~~echo~~ echo
• mesures the time between the ultrasonic pulses and the echo.

① DC — Voltage → [motor] → Physical

• rotational direction → current direction
• rotational speed → strength of current

② • stepper motors

↑ steps ↑ rotates smoother

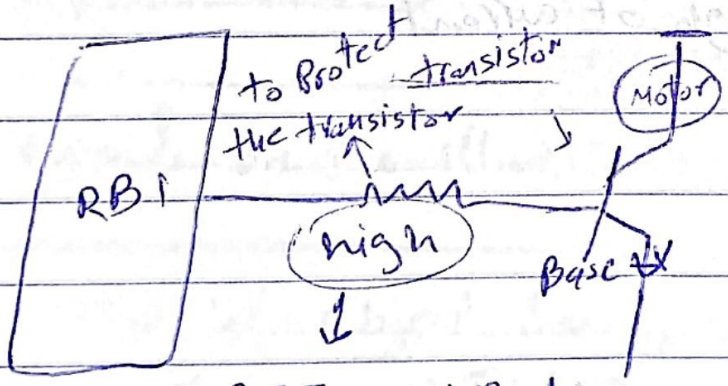
③ Servo motors

• rotates 180°
• PWM → according to pulse width → modulation the angular position
0° → PW = 1.25 ms →

Pulse width for $(55^\circ) = 1.25 + \frac{1.5 - 1.25}{90} \times 55$

for 90° → BSF PortB,1
delay 1.5 ms
BCF PortB,1
delay 18.5 ms
GoTo loop

Simple DC interfacing

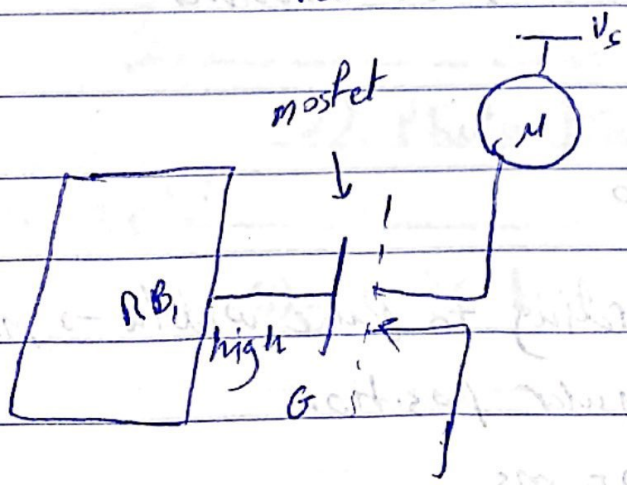


V_s → external voltage source controls the transistor

high → the motor works

low → open circuit

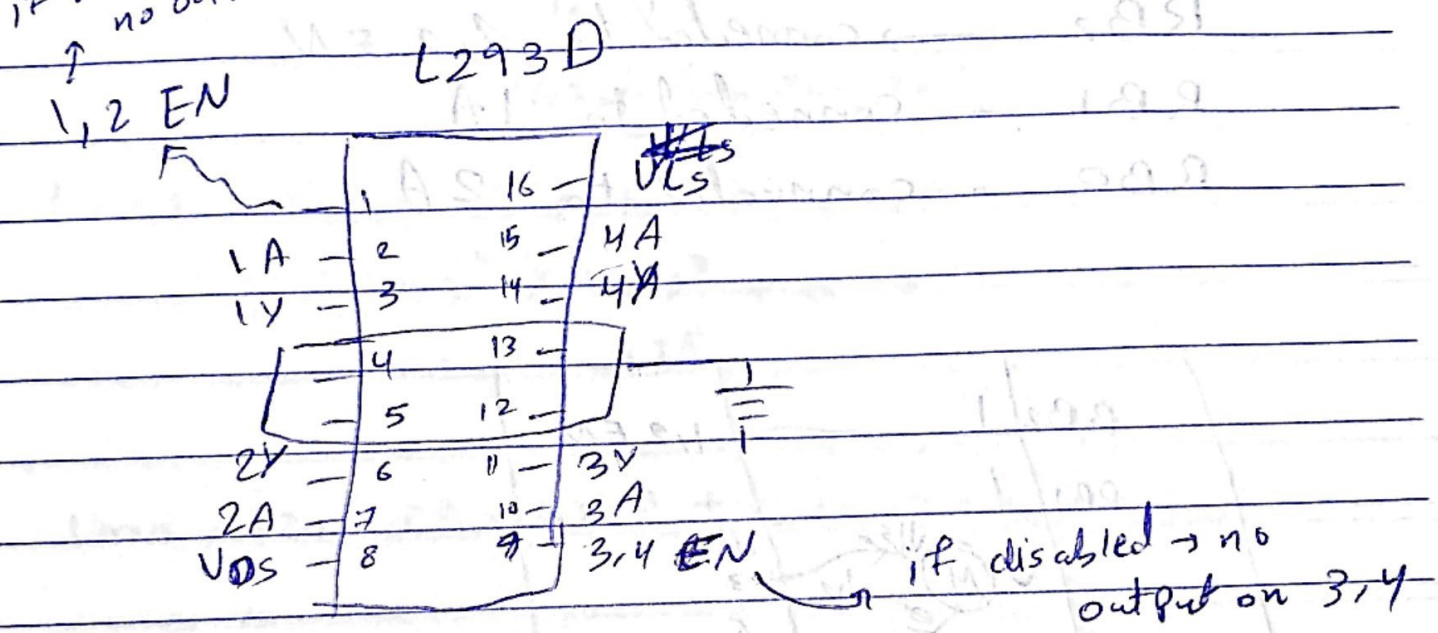
BSF PortB,1



no need for transistor in mosfet because no current through the Gate.

. DC switching allows driving loads with current flowing in one direction

. to make the DC motors rotates in two directions → Use H-bridge.



A → input

Y → output

VLS : Voltage logic level

H → VLS

L → 0

VDS → enable for the chip

↳ 0 => disabled.

. VDS > VLS

→

example: PortB(2) => 0 → motor is off

o.w &-

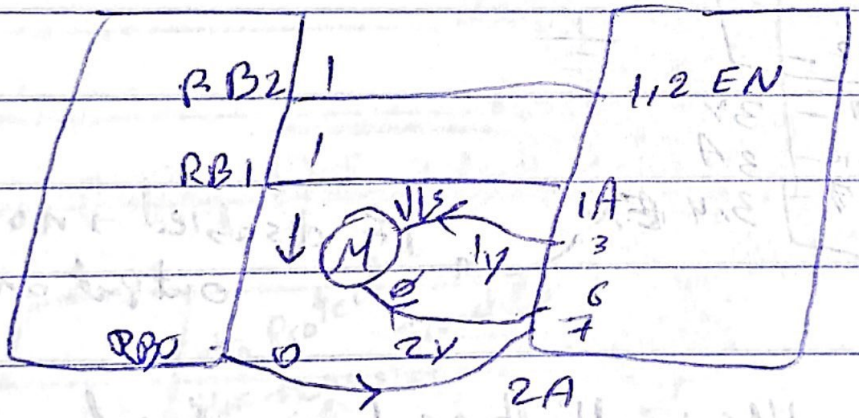
RB1 = 1 RB0 = 0 ↓

RB1 = 0 RB0 = 1 ↑

RB2 ⟶ connected to 1,2 EN

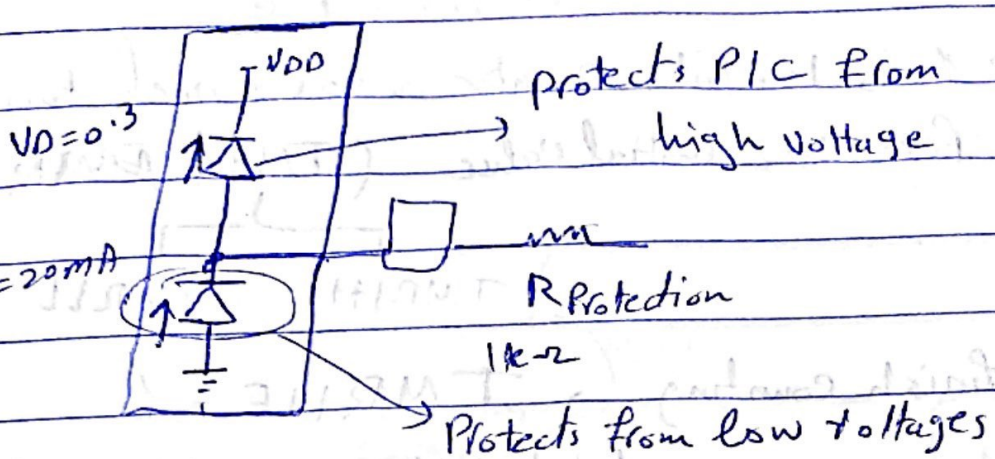
RB1 → connected to 1A

RB0 → connected to 2A



$V_{LS} < V_{OS}$

Lecture 28



$$V_{max} = I_{max} \times R_{prot} + V_D + V_{DD}$$
$$= 20\times10^{-3} \times 1\times10^{3} + 0.3 + 5$$
$$= 25.3\ V$$

$$V_{min} = -(20\times10^{-3} \times 1\times10^{3} + 0.3) = -20.3$$

- Filter → Cancels the noise
- schmitt trigger → fast switching

- we use filter with schmitt trigger to cancel bouncing in order to avoid the undefined region.
  - by using hardware
- cancel bouncing using software by (delay ≥ bouncing time)

Chapter 9

- Timer 1 is 16-bits counter → so we need two registers for the initial value (TMR1 ~~TMR~~)
  - TMR1H
  - TMR1L
- when finish counting → TMR1IF = 1
  - ↳ interrupt when GIE = 1
    - TMR1IE = 1
    - PEIE = 1
- TMR1 has on/off bit. (able / disable the counter)
- if the clock is external then we need synchronized.
  - ↳ values
- Presc (1/2/4/8) only
- RC0 → used for counting (connected with switch or push
- RC1 → connect with external OSC. with freq = 200 kHz

$$\text{delay of TMR1} = \frac{4}{Fosc} \times \text{Prescale} \times (2^{16} - IN)$$

$$\text{max delay of TMR1} = \frac{4}{Fosc} \times 8 \times 2^{16} = 8 \times \text{Timer0 delay}$$

- if IN = 264

0000 0001 | 0000 1000

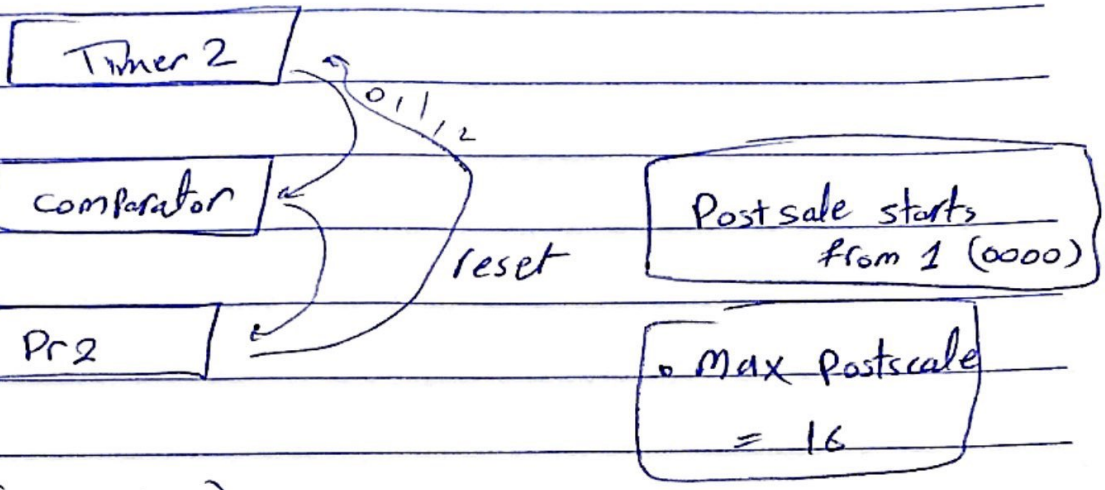
TMR1H | Timer1L

• Timer2 → 8-bit counter

↓

It has on/off bit

↳ 0, ... value in PR2 then reset



• Prescale = (1, 4, 16)

• only one source osc. → internal (no external clock)

• if Postscale = X → when TMR2 = PR2 X times

TMR2IF → interrupt

• delay of TMR2 = $\frac{4}{Fosc}$ * Presc * Postscale * (PR2+1)

↓

after → interrupt.

• delay to reset = $\frac{4}{Fosc}$ * Presc * (PR2+1)

• Max delay of TMR2 = $\frac{4}{Fosc}$ * 16 * 16 * (256) = TMR0 Max delay

Good Luck!