MICROCONTROLLER
SYSTEMS & DESIGN

A Comprehensive Engineering Approach to Embedded Architecture, Hardware Design, and Peripheral Interfacing.

Dr. Nabil Karami

Advanced Engineering Faculty & Senior PCB Designer

LO1



Week 1

Lecture

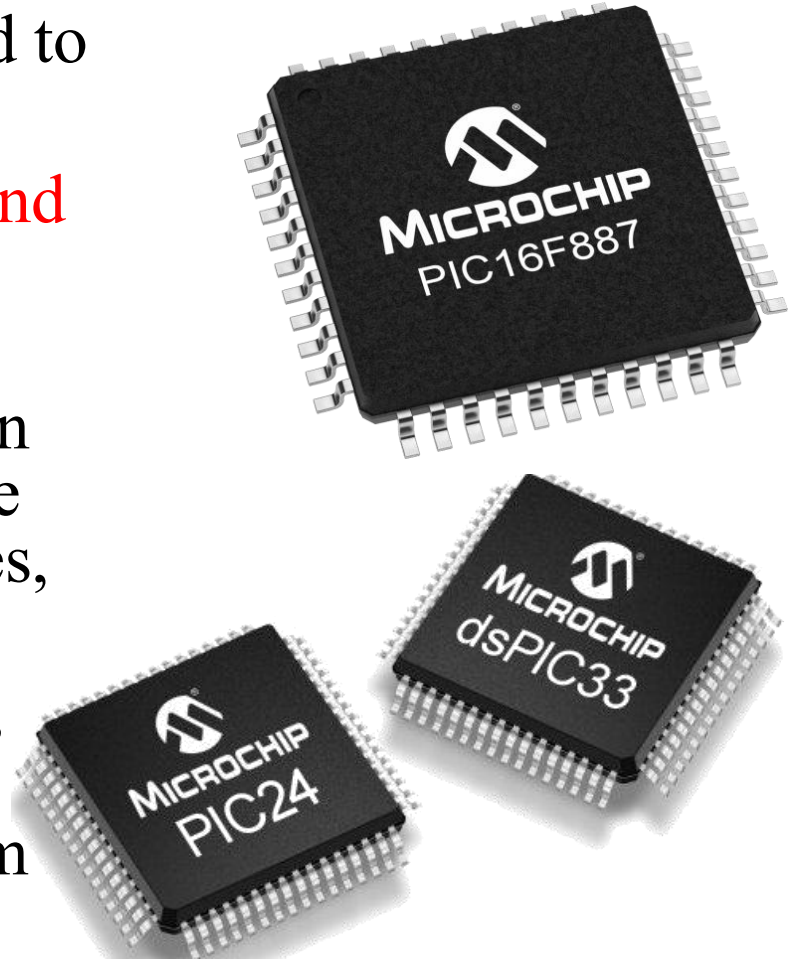
Lecture 1 / LO1

General Overview



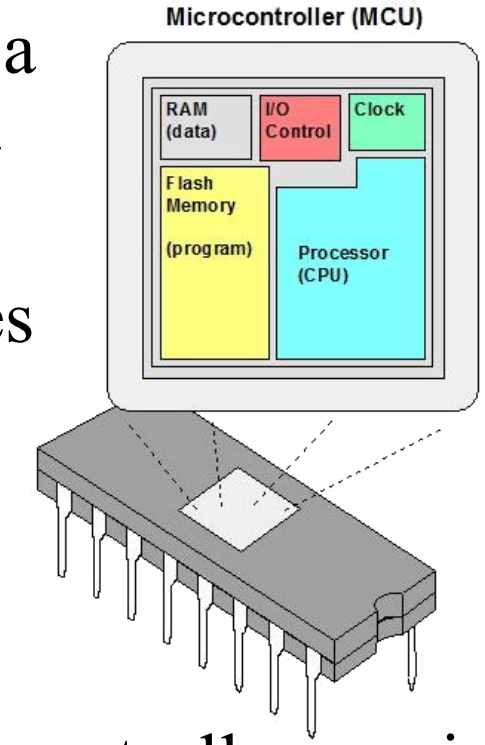
What is a Microcontroller !

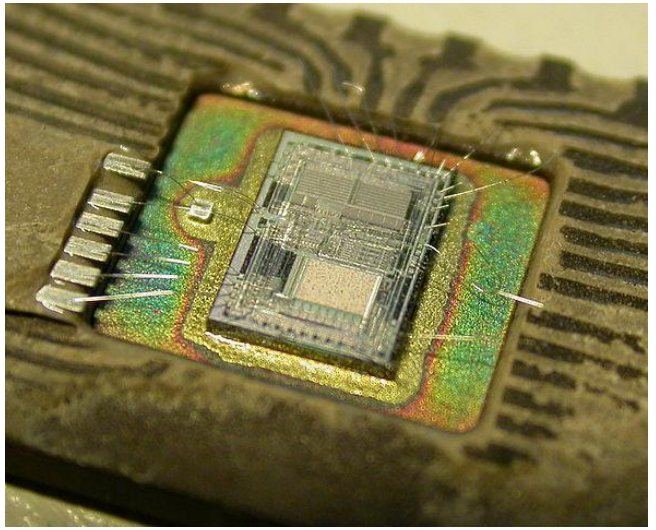
- A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system.
- A typical microcontroller includes a processor, memory and input/output (I/O) peripherals on a single chip.
- Sometimes referred to as an embedded controller or microcontroller unit (MCU), microcontrollers are found in vehicles, robots, office machines, medical devices, mobile radio transceivers, vending machines and home appliances, among other devices.
- They are essentially simple miniature personal computers (PCs) designed to control small features of a larger component, without a complex front-end operating system (OS).



How do microcontrollers work?

- A microcontroller is embedded inside of a system to control a singular function in a device.
- It does this by interpreting data it receives from its I/O peripherals using its central processor.

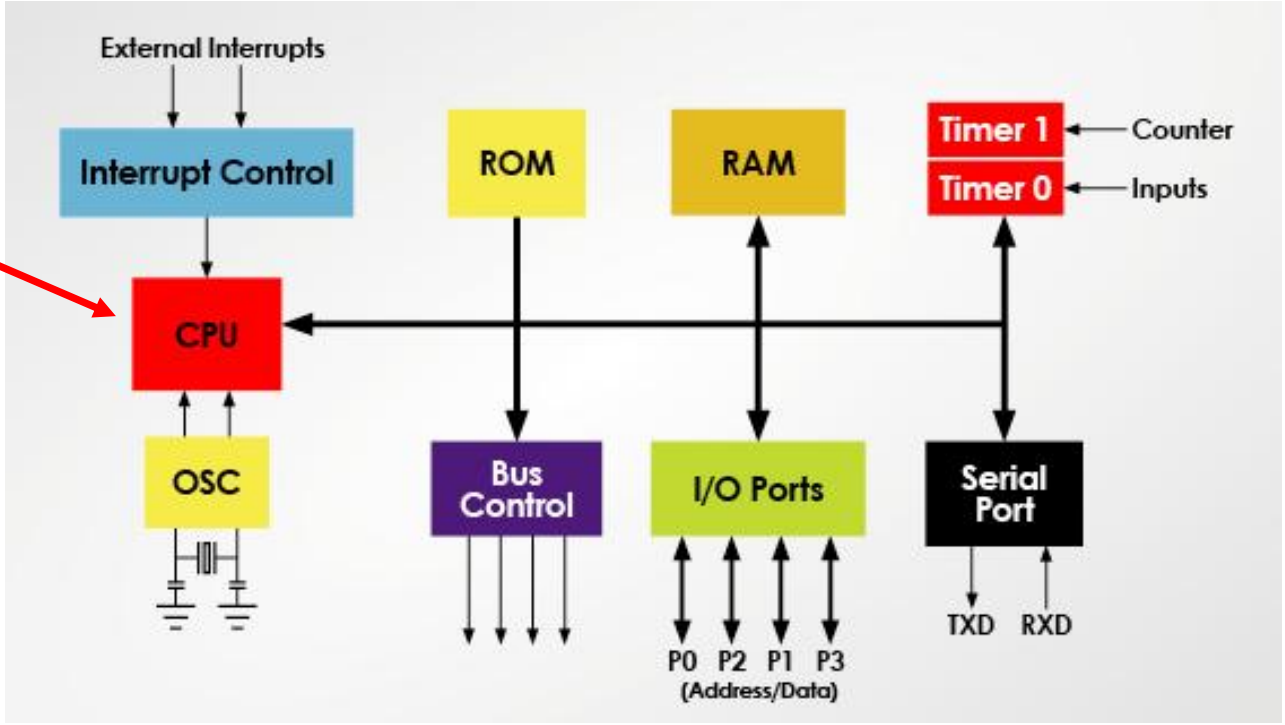




The temporary information that the microcontroller receives is stored in its data memory, where the processor accesses it and uses instructions stored in its program memory to decipher and apply the incoming data.
It then uses its I/O peripherals to communicate and enact the appropriate action.

What are the elements of a microcontroller?

- **The processor (CPU):** A processor can be thought of as the brain of the device.
- It processes and responds to various instructions that direct the microcontroller's function.
- This involves performing basic arithmetic, logic and I/O operations.
- It also performs data transfer operations, which communicate commands to other components in the larger embedded system.

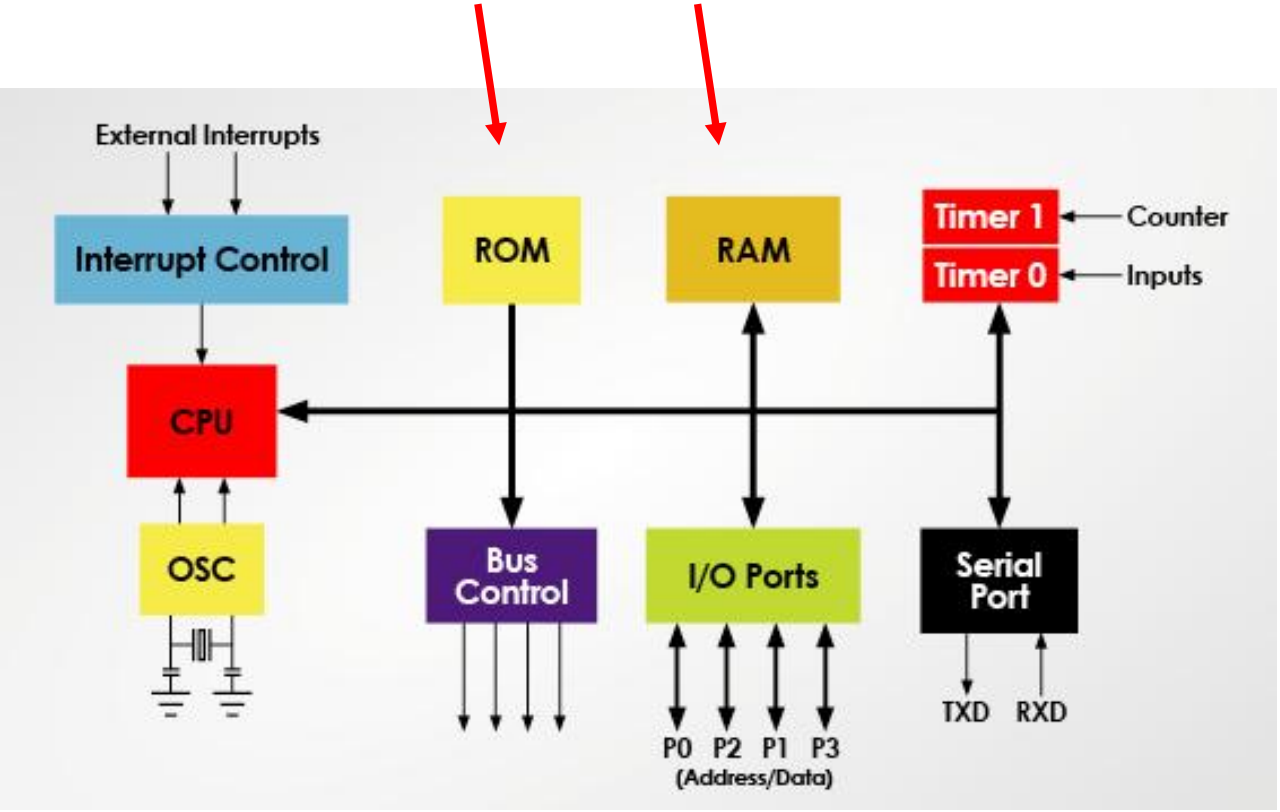


What are the elements of a microcontroller?

Memory: A microcontroller's memory is used to store the data that the processor receives and uses to respond to instructions that it's been programmed to carry out. A microcontroller has two main memory types:

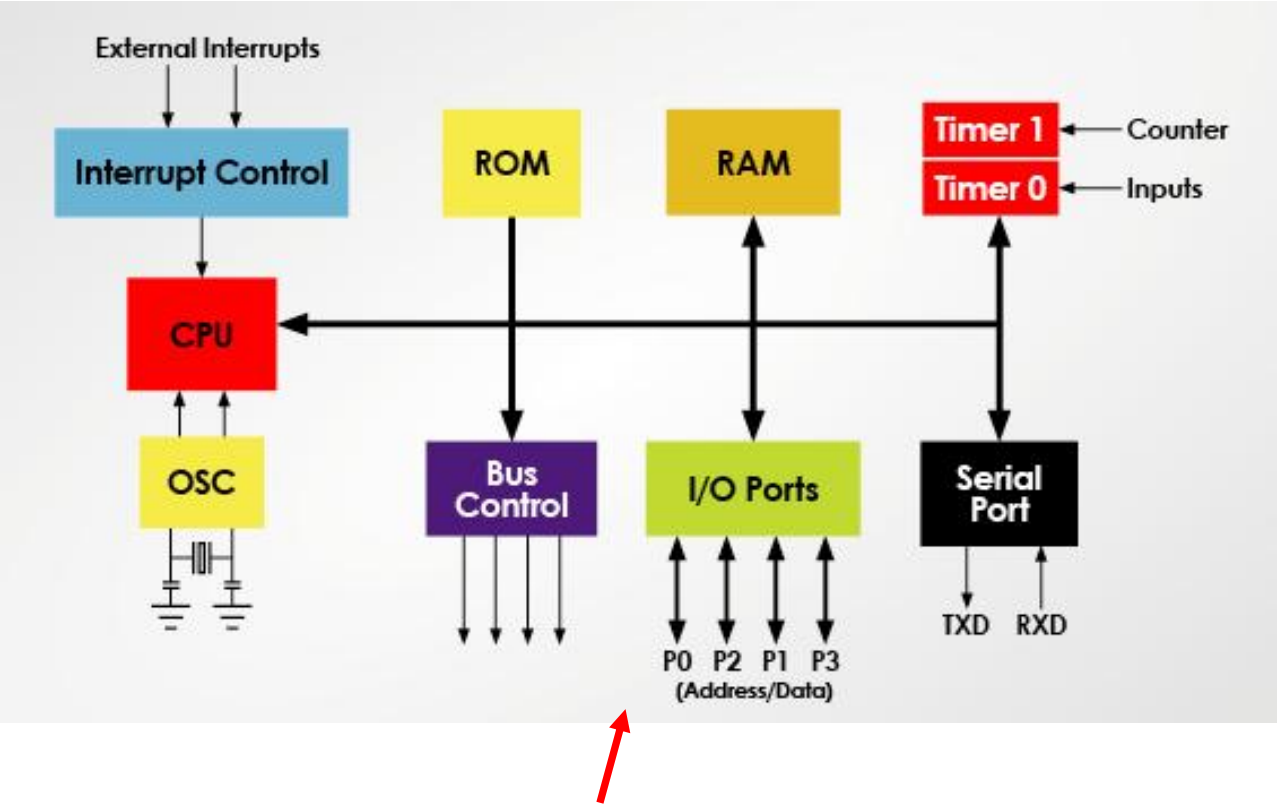
a. Program memory, which stores long-term information about the instructions that the CPU carries out. Program memory is non-volatile memory, meaning it holds information over time without needing a power source.

b. Data memory, which is required for temporary data storage while the instructions are being executed. Data memory is volatile, meaning the data it holds is temporary and is only maintained if the device is connected to a power source.



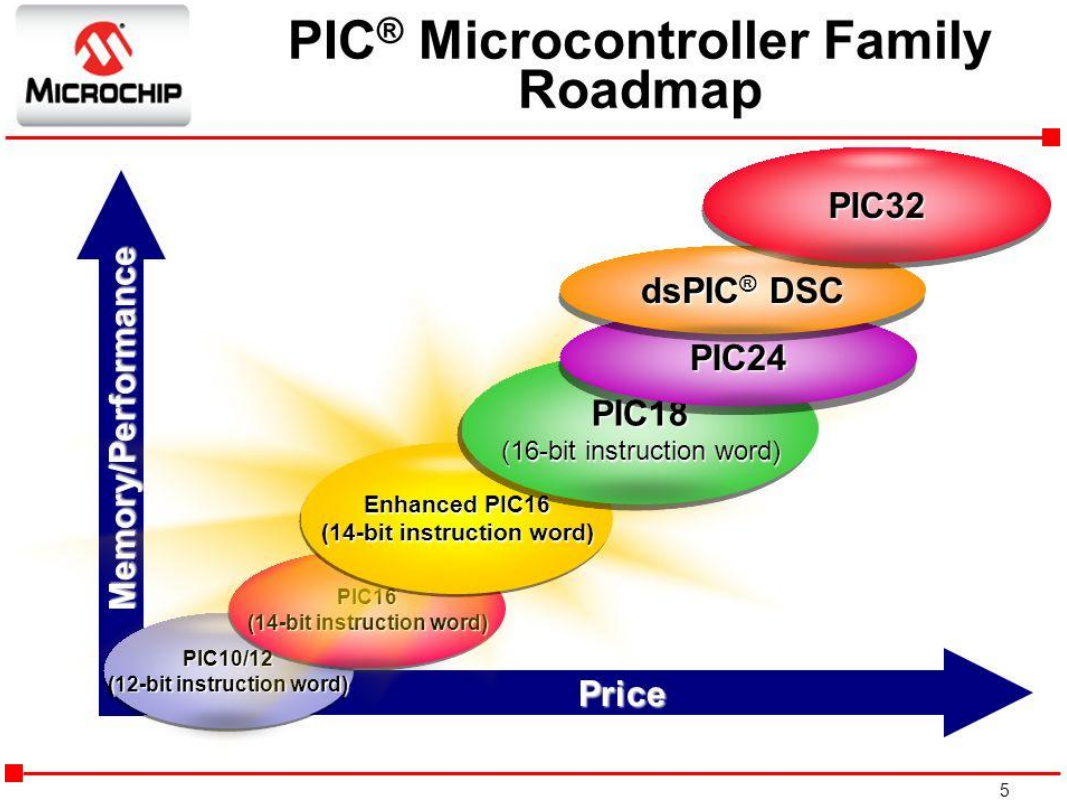
What are the elements of a microcontroller?

- I/O peripherals: The input and output devices are the interface for the processor to the outside world.
- The input ports receive information and send it to the processor in the form of binary data. The processor receives that data and sends the necessary instructions to output devices that execute tasks external to the microcontroller.
- While the processor, memory and I/O peripherals are the defining elements of the microprocessor, there are other elements that are frequently included.
- The term *I/O peripherals* itself simply refers to supporting components that interface with the memory and processor.
- There are many supporting components that can be classified as peripherals.



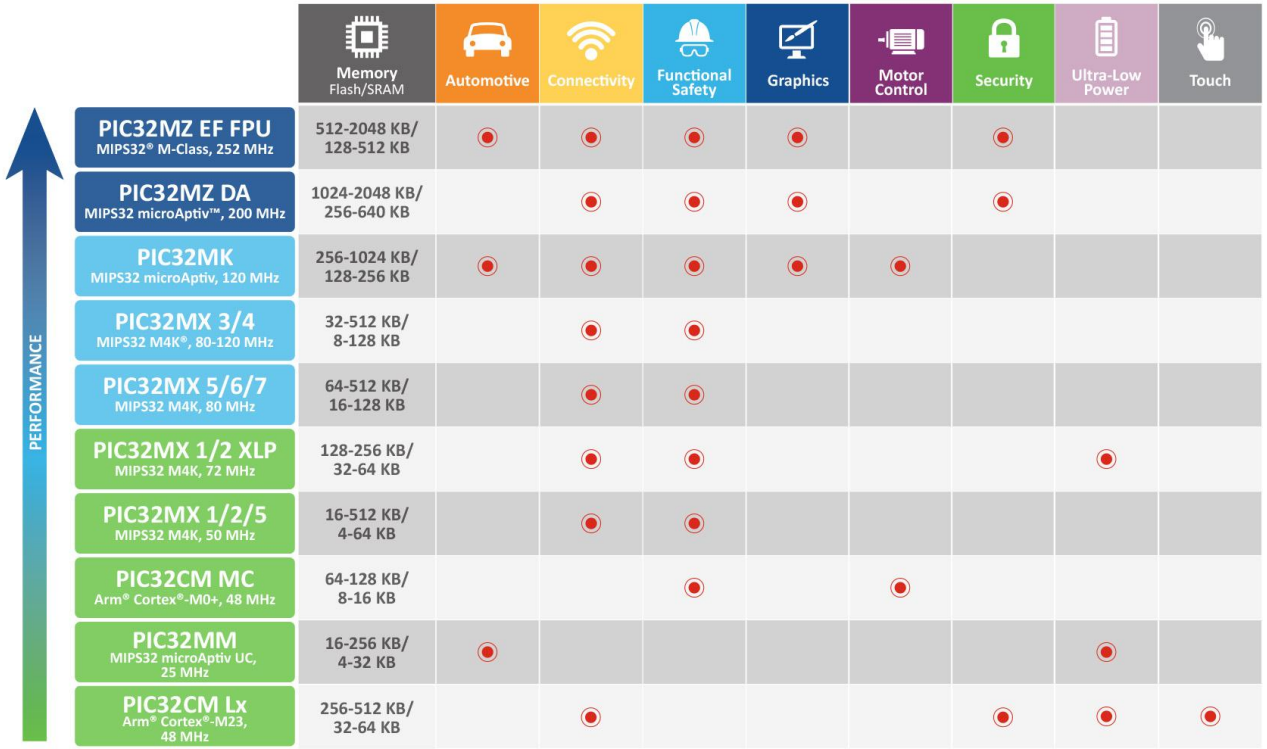
Microcontroller features

- A microcontroller's processor will vary by application. Options range from the simple 4-bit, 8-bit or 16-bit processors to more complex 32-bit or 64-bit processors.
- Generally, microcontrollers are designed to be readily usable without additional computing components because they are designed with sufficient onboard memory as well as offering pins for general I/O operations, so they can directly interface with sensors and other components.



Microcontroller applications

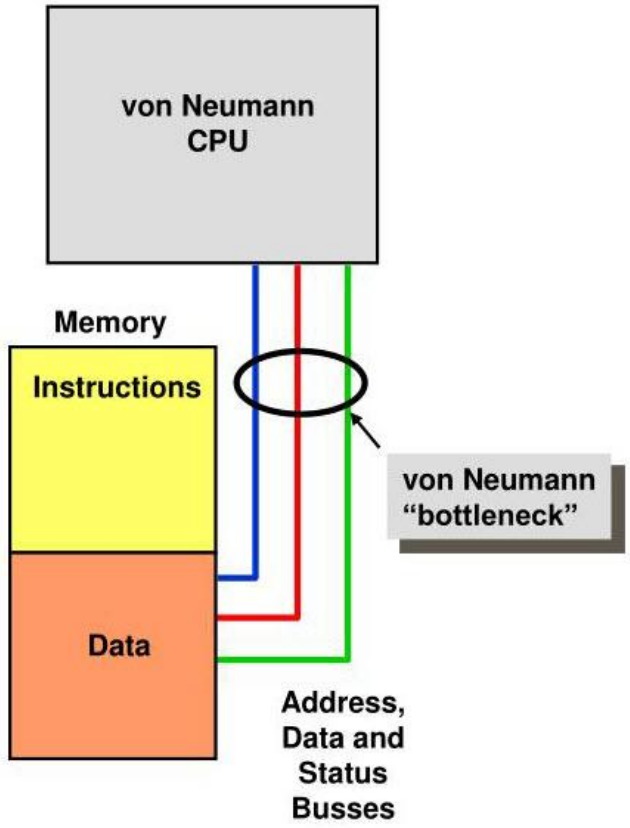
- Microcontrollers are used in multiple industries and applications, including in the home and enterprise, building automation, manufacturing, robotics, automotive, lighting, smart energy, industrial automation, communications and internet of things (IoT) deployments.

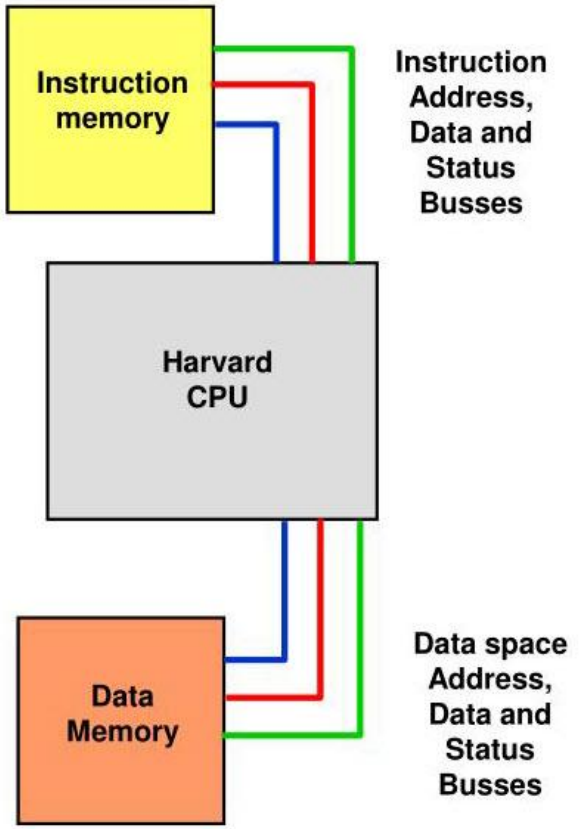


- The simplest microcontrollers facilitate the operation of electromechanical systems found in everyday convenience items, such as ovens, refrigerators, toasters, mobile devices, video game systems, televisions and lawn-watering systems.

Microcontroller Architecture

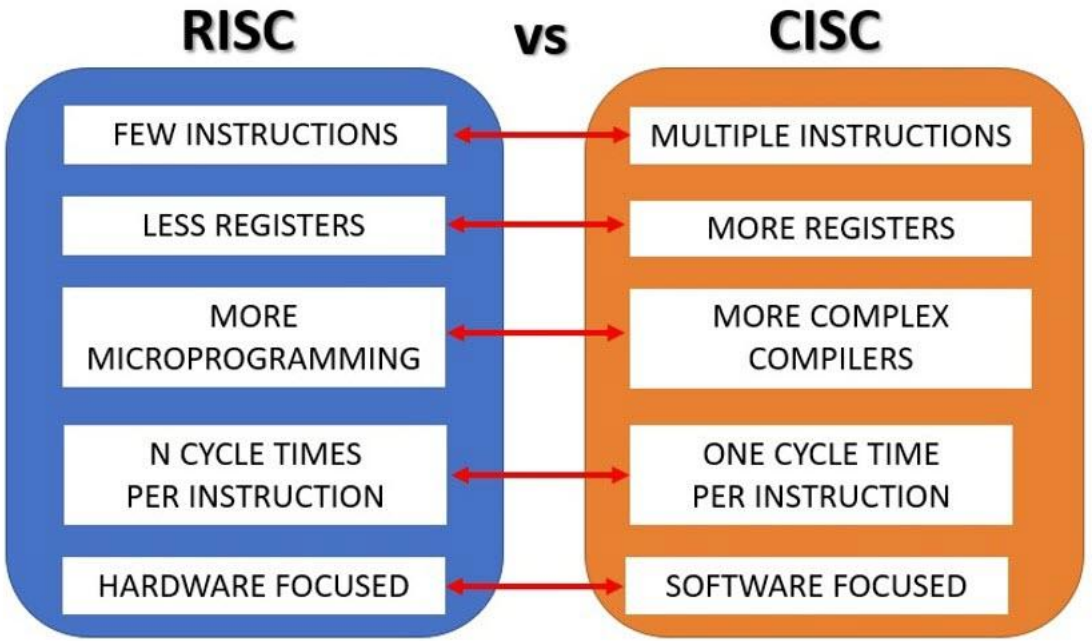
- Microcontroller architecture can be based on the Harvard architecture or von Neumann architecture, both offering different methods of exchanging data between the processor and memory.
- With a Harvard architecture, the data bus and instruction are separate, allowing for simultaneous transfers.
- With a Von Neumann architecture, one bus is used for both data and instructions.





Microcontroller Architecture

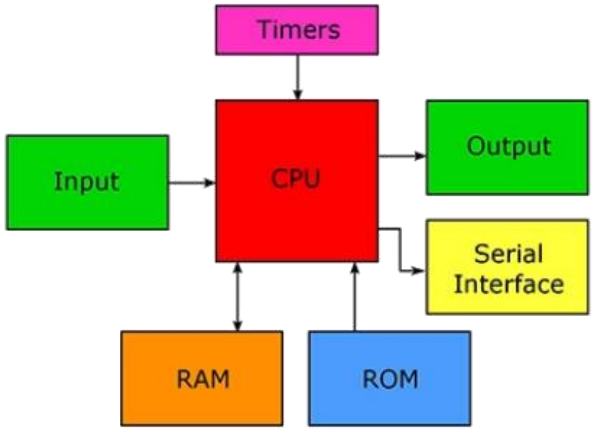
- Microcontroller processors can be based on complex instruction set computing (CISC) or reduced instruction set computing (RISC).
- CISC generally has around 80 instructions while RISC has about 30, as well as more addressing modes, 12-24 compared to RISC's 3-5.
- When they first became available, microcontrollers solely used assembly language. Today, the C programming language is a popular option. Other common microprocessor languages include Python and JavaScript.
- The PIC microcontroller is based on the Harvard with RISC architecture



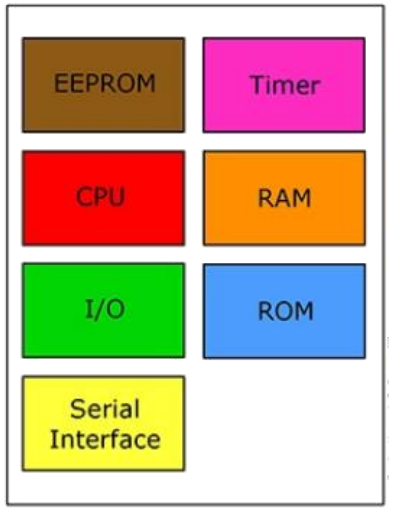
Microcontrollers vs. microprocessors

- Microcontrollers can be said to function usefully on their own, with a direct connection to sensors and actuators, where,
- Microprocessors are designed to maximize compute power on the chip, with internal bus connections (rather than direct I/O) to supporting hardware such as RAM and serial ports.
- Simply put, coffee makers use microcontrollers; desktop computers use microprocessors.
- Microcontrollers are less expensive and use less power than microprocessors.
- Microprocessors do not have built-in RAM, read-only memory (ROM) or other peripherals on the chip, but rather attach to these with their pins.
- A microprocessor can be considered the heart of a computer system, whereas a microcontroller can be considered the heart of an embedded system.

Microprocesser: CPU and several supporting chips.



Microcontroller: CPU on a single chip.





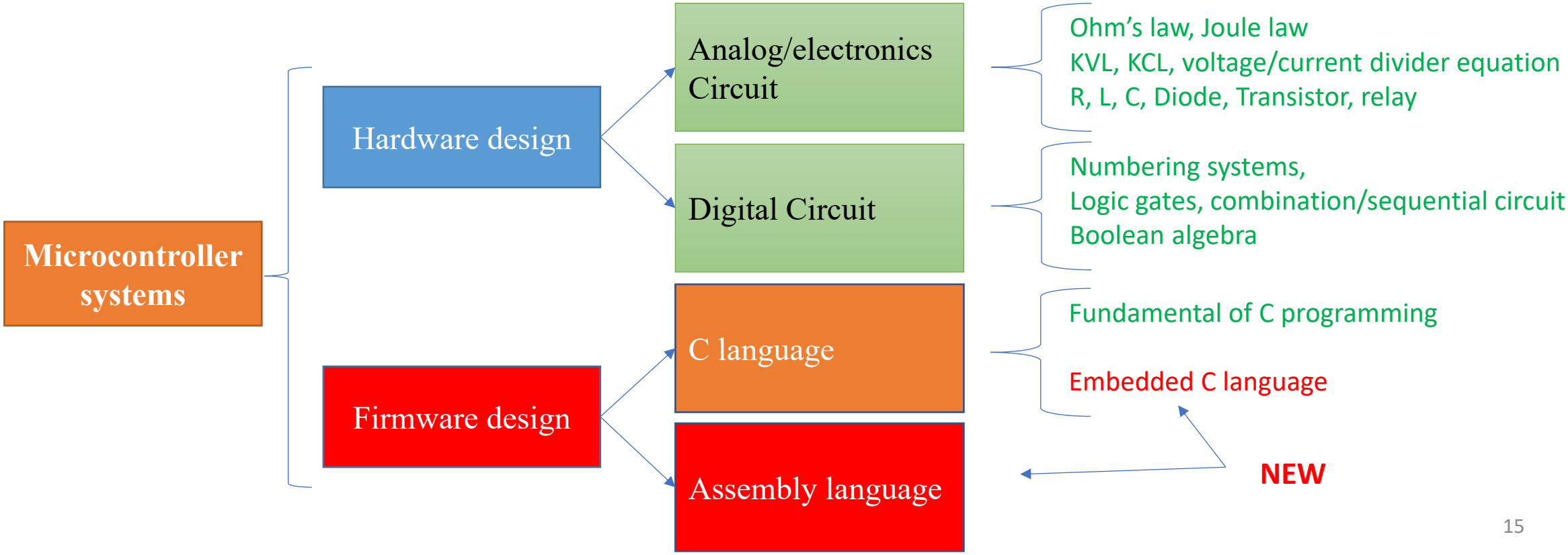
Choosing the right microcontroller

- There are a number of technology and business considerations to keep in mind when choosing a microcontroller for a project.
- Beyond cost, it is important to consider the maximum speed, amount of RAM or ROM, number or types of I/O pins on an MCU, as well as power consumption and constraints and development support. Be sure to ask questions such as:
  - What hardware peripherals are required?
  - Are external communications needed?
  - What architecture should be used?
  - What sort of community and resources are available for the microcontroller?
  - What is the market availability of the microcontroller?

Skills/knowledge required to build a Microcontroller based system

- To build a microcontroller based system you **MUST** be familiar with

1. Hardware circuit design (analog and digital circuits design)
2. Firmware Programming languages (Assembly and/or C language)





Skills/knowledge required to build a Microcontroller based system

Thus, in this course you will learn how to design a Firmware using assembly and C languages to perform a specific task.

- An assembly language is a type of low-level programming language that is intended to communicate directly with a computer's hardware.
- Assembly languages are designed to be readable by humans.
- Assembly language is a necessary bridge between the underlying hardware of a processor/controller and the higher-level programming languages—such as Python or JavaScript—in which modern software programs are written.



Week 1

Lecture

Lecture 2 / LO1

Numbering Systems and Logic operations



Numbering Systems

Three main numbering systems are required to build/understand low level coding… numbers are mainly described in 3 different formats such as

1: Decimal numbers: Base 10 [0, 1, 2, 3, …., 9], total of 10 digits

the last digit is 9 which is the base – 1 = 10-1 = 9

2: Binary numbers: Base 2 [0, 1], total of 2 digits

the last digit is 1 which is the base – 1 = 2-1 = 1

3: Hexadecimal numbers: Base 16 [0, 1, 2, 3…..15], total of 16 digits

the last digit is 15 which is the base – 1 = 16-1 = 15



Numbering Systems (Decimal numbers)

- Decimal numbers are used by programmer due to its simplicity (Human being count number on their 10 fingers)

A simple decimal number is a number between 0 and 9

After 9, the decimal number becomes a combined number made of two digits

After 99, the decimal number becomes a combined number made of three digits … etc.

Each digit is represented by its value multiplied by its base weight (position)

For example, 125 is written as 1 x100 + 2 x10 + 5 x1

where 100, 10, and 1 are the base weights of the numbers1, 2 and 5, respectively

Numbering Systems (Decimal numbers)

The number **125** can be written as

$$1 \times 100 + 2 \times 10 + 5 \times 1$$

Weights

$$1 \times 10^{2} + 2 \times 10^{1} + 5 \times 10^{0}$$

Base



Numbering Systems (Binary numbers)

- Binary numbers are used by the microcontroller (they represent True or false state of current, i.e. 1 or 0 availability of current)

A simple binary number is a number made of 0 or 1 (it is called **bit**)

After 1, the binary number becomes a combined number made of two digits

After 11, the binary number becomes a combined number made of three digits… etc.

Each digit is represented by its value multiplied by its base weight (position)

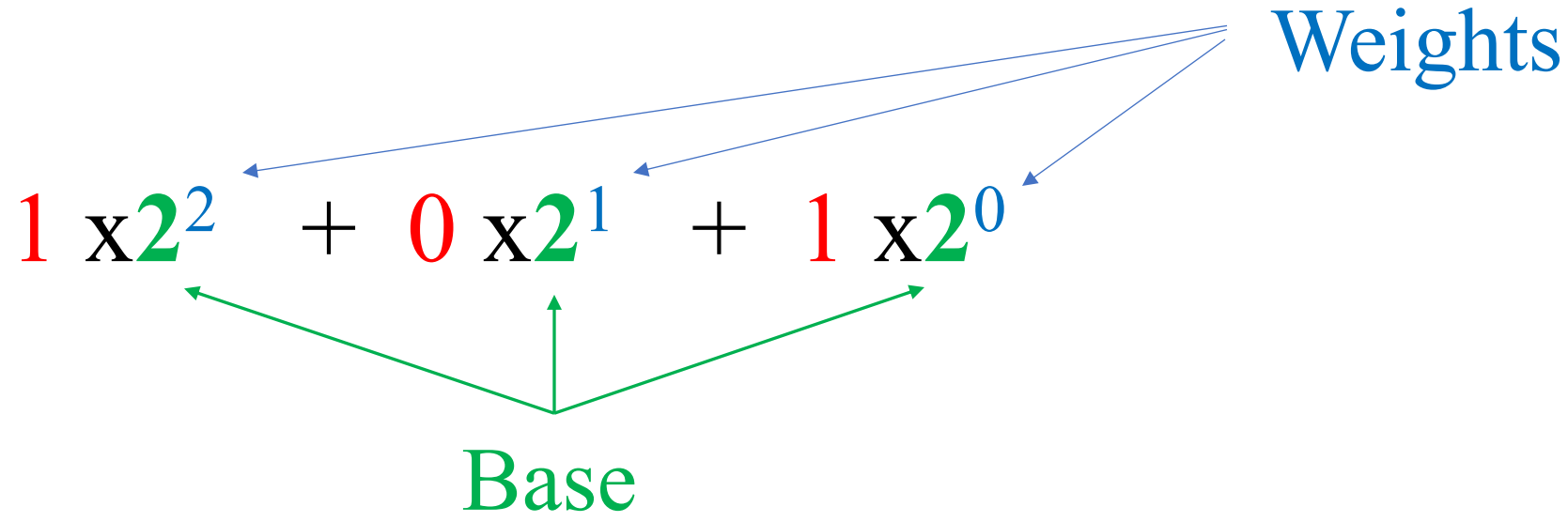
For example, 101 is written as 1 x4 + 0 x2 + 1 x1

where 4, 2, and 1 are the base weights of the numbers 1, 0 and 1, respectively

Numbering Systems (Binary numbers)

The number **101** can be written as

$$1 \times 4 + 0 \times 2 + 1 \times 1$$

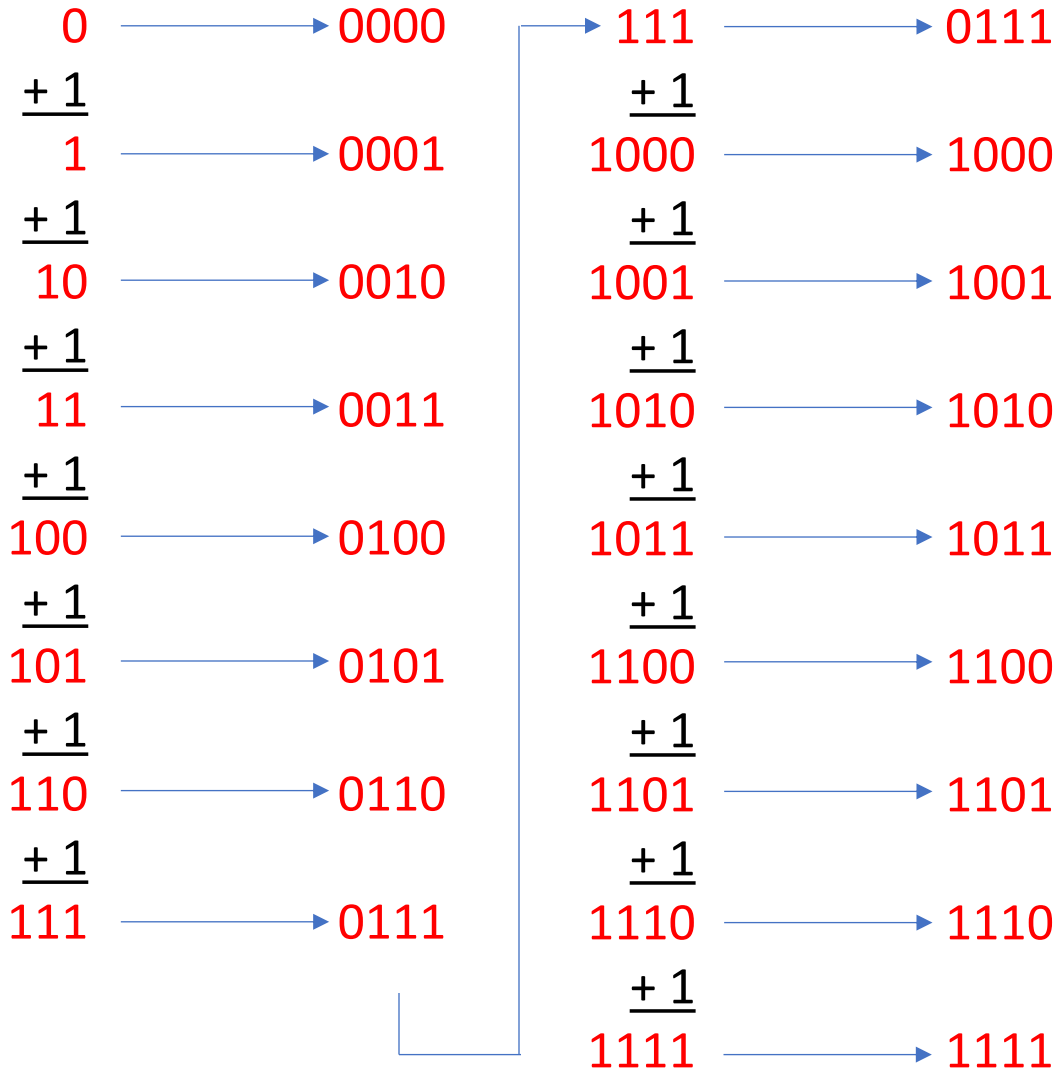


Numbering Systems (Binary numbers)

- It is important to remember the 16 first binary numbers...this will reduce the programming complexity and time.

- Consider that
  - 0+0 = 0
  - 0+1 = 1
  - 1+0 = 1
  - 1+1 = 10

- Thus the first 16 binary numbers are generated from the 16 sequential increments





Numbering Systems (Binary numbers)

- A **BIT** is a binary digit, the smallest increment of data on a computer.
- Bits are usually assembled into a group of **4 bits** to form a **Nibble**
- Also assembled into a group of **8 bits** to form a **BYTE**. A byte contains enough information to store a single ASCII character, like “A, h @, 1, 6, !".
- If bits are assembled in a group of **more than 8 bits** then it will form a **WORD**

- For example 1010,0001 is a BYTE

  11,0010,0001 is a WORD of 10 bits

  0010,0000,1111 is a WORD of 12 bits



Numbering Systems (Hexadecimal numbers)

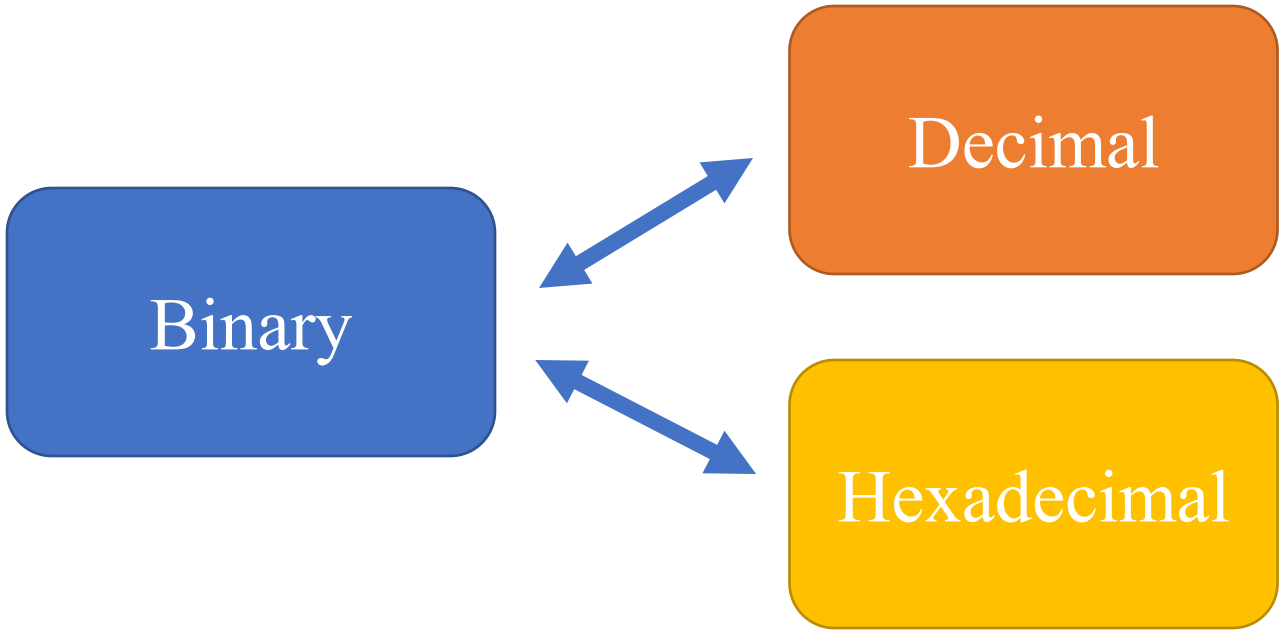
- Hexadecimal numbers are used to easily read binary numbers.
- A simple Hexadecimal number is a number made of 0 to 15
- In order not to confuse between decimal and hexadecimal numbers, the hex numbers made of two digits are replaced by letters.
- Thus the simple hex numbers are 0, 1, ….9, A, B, C, D, E, F, where A to F represents 10 to 15.
- After F, the hex number becomes a combined number made of two digits
- After FF, the hex number becomes a combined number made of three digits… etc.

- Usually, Hex numbers are used to express the 4-bit nibbles for simplicity

- For example
  - 1010,1001 in binary is A9 in Hexadecimal
  - 11,0010,0001 in binary is 321 in Hexadecimal
  - 0010,0000,1111 in binary is 20F in Hexadecimal

Conversion between numbering Systems

- During programming, it will be required to make conversions between different numbering systems.
- You have to learn how to make **mental** conversion from binary ↔ decimal and from binary ↔ hexadecimal



- DON'T do direct conversion from hex to decimal and vice-versa. Always pass through binary



Conversion from Binary to Decimal

- Having for example a binary number 1101,0110. To convert it to decimal, follow the steps

1101,0110

$= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$= 1 \times 128 + 1 \times 64 + 0 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1$

= 214 in decimal

You have to remember $2^7=128$, $2^6=64$, $2^5=32$, $2^4=16$, $2^3=8$, $2^2=4$, $2^1=2$, $2^0=1$



Conversion from Decimal to Binary

- Having for example a decimal number 214. To convert it to binary, follow the steps

214

Keep in mind the following numbers 128 64 32 16 8 4 2 1
And prepare the binary number b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

Does 214 contain 128 ? Yes → replace b_7 by 1 → calculate 214-128 = 86
Does 86 contain 64? Yes → replace b_6 by 1 → calculate 86-64 = 22
Does 22 contain 32? No → replace b_5 by 0
Does 22 contain 16? Yes → replace b_4 by 1 → calculate 22-16 = 6
Does 6 contain 8? Yes → replace b_3 by 0
Does 6 contain 4? Yes → replace b_2 by 1 → calculate 6-4 = 2
Does 2 contain 2? Yes → replace b_1 by 1 → calculate 2-2 = 0
Does 0 contain 1? No → replace b_0 by 0



Conversion from Hex to Binary and vice-versa

- As we already know the first 16 values in binary, Hexadecimal and decimal, as shown in the table, we can use our memory to make direct conversion from/to binary to hexadecimal

- For Example: 0101 0000 in binary is 50 in Hex

  1111 1111 in binary is FF in Hex

- Or for example FAB3 in Hex is

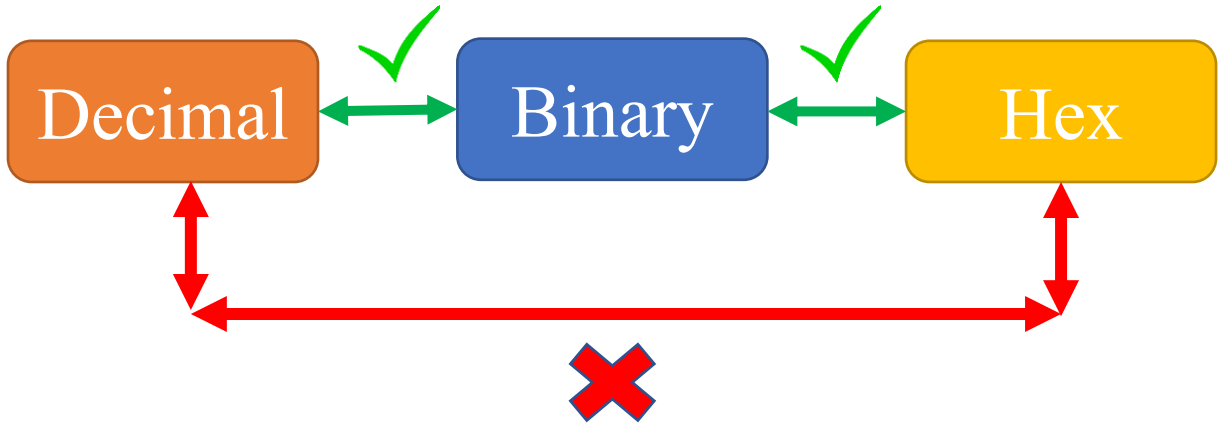
  1111,1010,1011,0011 in Binary

Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Conversion from Hex to Decimal and vice-versa

- In case we need to convert from decimal to Hex or from Hex to decimal for a value below 16, the table can still be used.

- But in case we need to convert a bigger value from/to decimal/hex, It is highly recommended to pass through binary.



Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Conversion from Hex to Decimal and vice-versa

- Example: Convert decimal 120 to Hex

- Step1: From decimal to binary
  - 120 contains 64 → $b_6 = 1$, 120-64 = 56
  - 56 contains 32 → $b_5 = 1$, 56-32=24
  - 24 includes 16 → $b_4 = 1$, 24-16 = 8
  - 8 includes 8 → $b_3 = 1$, 8-8=0
  - All other b_x are zeros
  - Thus the binary value will be 0111,1000

- Step 2: From binary to Hex (using table)

  0111,1000 = 78 in Hex

- Thus decimal 120 is 78 in Hex

Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Conversion from Hex to Decimal and vice-versa

- Example: Convert Hex C7 to decimal

- Step1: From Hex to binary (From table)

  C7 = 1100,0111

- Step 2: From binary to Decimal

  the number 1100,0111 has

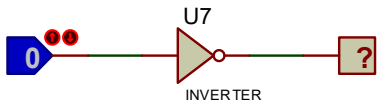
  128 + 64 + 4 + 2 + 1 = 199

- Thus Hex C7 is 199 in decimal

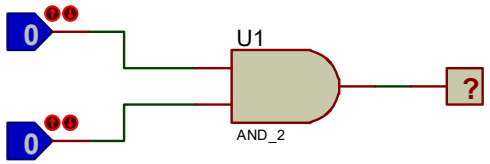
Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Logic Gates/Operations

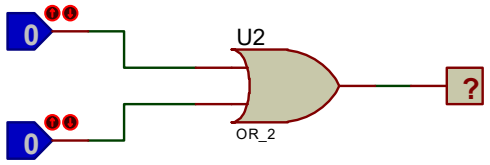
- Logic gates are the basic building blocks of any digital system. It is an electronic circuit having one or more than one input and only one output. The relationship between the input and the output is based on a **certain logic**. Based on this, logic gates are named as AND gate, OR gate, NOT gate etc.



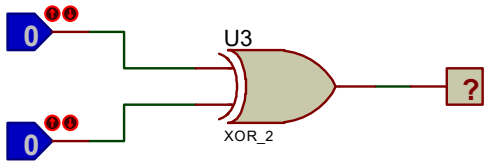
NOT gate	
A	OUT
0	1
1	0



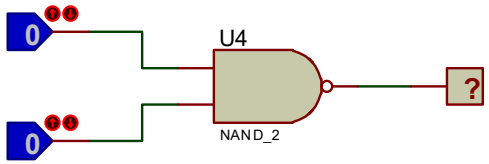
AND gate		
A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1



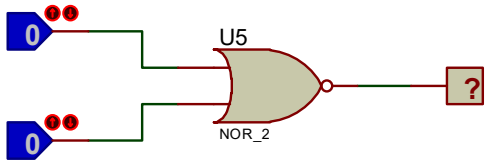
OR gate		
A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1



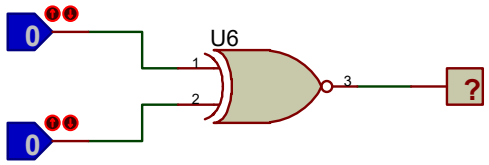
XOR gate		
A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0



NAND gate		
A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0



NOR gate		
A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0



XNOR gate		
A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	1



Logic Operations

- In 8-bit microcontroller, the main logic operations are AND, OR and XOR.
- The operations are made as byte with another byte, i.e. 8 bits with another 8 bits.

- Example, ANDing 1010,0001 with 11011100 is

Byte 1	1	0	1	0	0	0	0	1
Byte 2	1	1	0	1	1	1	0	0
Byte 1 AND Byte 2	1	0	0	0	0	0	0	0

Logic Operations

- Example, ORing 1010,0001 with 11011100 is

Byte 1	1	0	1	0	0	0	0	1
Byte 2	1	1	0	1	1	1	0	0
Byte 1 OR Byte 2	1	1	1	1	1	1	0	1

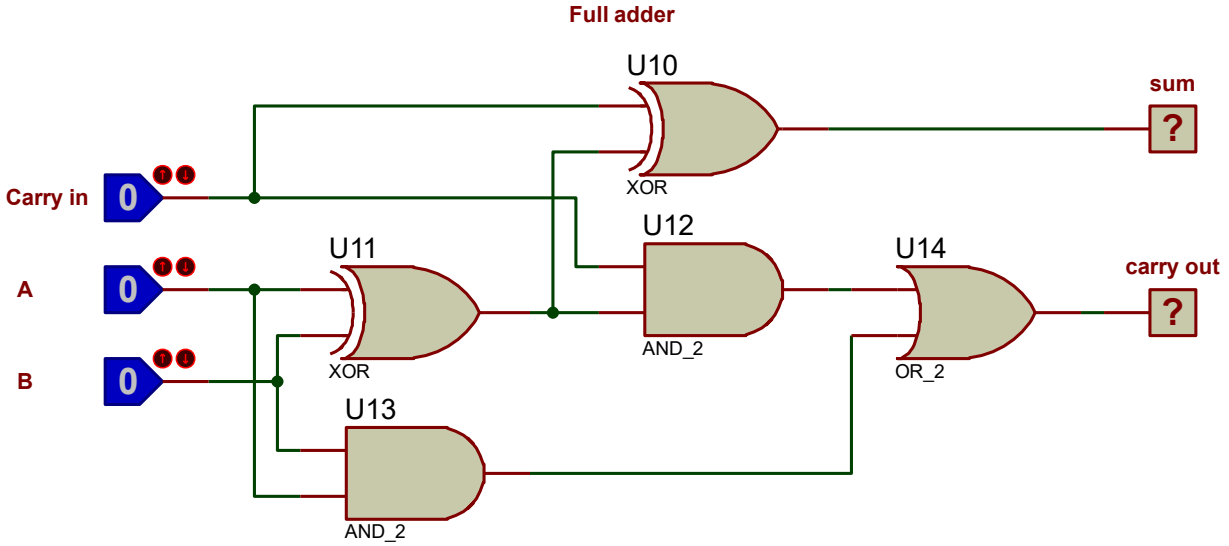
- Example, XORing 1010,0001 with 11011100 is

Byte 1	1	0	1	0	0	0	0	1
Byte 2	1	1	0	1	1	1	0	0
Byte 1 XOR Byte 2	0	1	1	1	1	1	0	1

Arithmetic Operations

- Arithmetic operations are such Addition, Subtraction, Multiplication, Division…

- For a 1bit added, the circuit and the truth table are as follow



Carry In	A	B	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Arithmetic Operations (Addition)

- Microcontroller can also solve 8-bit arithmetic operations like Addition and Subtraction (Advanced microcontroller can also make Multiplication/Division)

- Example, ADDing 1010,0001 with 11011100 is

Start from the LSB

	Carry								
Byte 1		1	0	1	0	0	0	0	1
Byte 2		1	1	0	1	1	1	0	0
Byte 1 + Byte 2	**1**	0	1	1	1	1	1	0	1



Arithmetic Operations (Subtraction)

- The PIC microcontroller can do Subtraction but with a special technique. The Subtraction is based on Addition and thus reducing the complexity/cost of the chip. Thus no real Subtractor circuit inside the chip.
- The method is called two's complement.
- For Example: Byte1 – Byte2 is achieved as

  Step 1: do the complement of Byte2

  Step 2: Add 1 to the result

  Step 3: Then do simple addition of Byte1 with the result of step 2

- We will do 3 examples of different values of Byte1 and Byte2 and check what will happen to the Carry bit

Arithmetic Operations (Subtraction)

- Let's assume that Byte1 = 60 and Byte2 = 50 (case of Byte1 > Byte2)

	Byte1 = 60		0	0	1	1	1	1	0	0
	Byte2 = 50		0	0	1	1	0	0	1	0
Step 1	**Byte2 is flipped= 205**		1	1	0	0	1	1	0	1
Step 2	**Add 1**		0	0	0	0	0	0	0	1
	result = 206		1	1	0	0	1	1	1	0
Step 3	**Add to Byte1 = 60**		0	0	1	1	1	1	0	0
	Byte1 + Byte2 = 266	1	0	0	0	0	1	0	1	0

1

Carry is 1

10

Value is positive

Arithmetic Operations (Subtraction)

- Let's assume that Byte1 = 60 and Byte2 = 60 (case of Byte1 = Byte2)

	Byte1 = 60		0	0	1	1	1	1	0	0
	Byte2 = 60		0	0	1	1	1	1	0	0
Step 1	**Byte2 is flipped= 195**		1	1	0	0	0	0	1	1
Step 2	**Add 1**		0	0	0	0	0	0	0	1
	result = 196		1	1	0	0	0	1	0	0
Step 3	**Add to Byte1 = 60**		0	0	1	1	1	1	0	0
	Byte1 + Byte2 = 256	1	0	0	0	0	0	0	0	0

1

Carry is 1

0

Value is zero

Arithmetic Operations (Subtraction)

- Let's assume that Byte1 = 50 and Byte2 = 60 (case of Byte1 < Byte2)

	Byte1 = 50		0	0	1	1	0	0	1	0
	Byte2 = 60		0	0	1	1	1	1	0	0
Step 1	**Byte2 is flipped= 195**		1	1	0	0	0	0	1	1
Step 2	**Add 1**		0	0	0	0	0	0	0	1
	result = 196		1	1	0	0	0	1	0	0
Step 3	**Add to Byte1 = 50**		0	0	1	1	0	0	1	0
	Byte1 + Byte2 = -10	**0**	1	1	1	1	0	1	1	0

0

Carry is 0

246

The result is a number that represents a negative number of 246-256 = -10



Arithmetic Operations (Subtraction)

- So, as a conclusion on the Subtraction operations

- If Byte1 ≥ BYTE2, Carry = 1
- If Byte1 < BYTE2, Carry = 0

- IMPORTANT NOTE: Other microcontrollers may have another subtraction methods and the **Carry bit** may vary compared to the PIC subtraction.



Test your knowledge

1. Convert from Binary to Decimal the following numbers
   - B’11010001’
   - B’11111111’
   - B’0111111’
   - B’1000000’
2. Convert from Decimal to Binary the following numbers
   - D’128’
   - D’255’
   - D’127’
   - D’16’
3. Convert from Hex to Decimal the following numbers
   - H’AB’
   - H’FF’
   - H’1F
   - H’7F’



4. Convert from Decimal to Hex the following numbers
   - D’128’
   - D’127’
   - D’255’
   - D’16’
5. Convert from Hex to Binary the following numbers
   - H’11’
   - H’7F’
   - H’FF’
   - H’AC’
6. Convert from Binary to Hex the following numbers
   - B’11001111’
   - B’11111’
   - B’10101010’
   - B’01010101’

Test your knowledge



7. Calculate D’128’ AND D’127’
8. Calculate H’AF’ OR B’10100001’
9. Calculate D’128’ – H’1F’ using two’s complement

Test your knowledge



Week 1

Lab 0 / LO1

Installing MPLAB



PIC programming languages

- PIC can be programmed using the official languages provided by the manufacturer or using third-party languages.
- Official languages: Assembly language, C language (Free tools)
- Third-party languages: mikroC, Flowcode, PICBASIC, mikroBasic, mikroPascal and many others (paid or partially free)
- In this course we will focus only on the official languages

PIC programming software

- The official software used to write assembly or C languages is MPLABX IDE

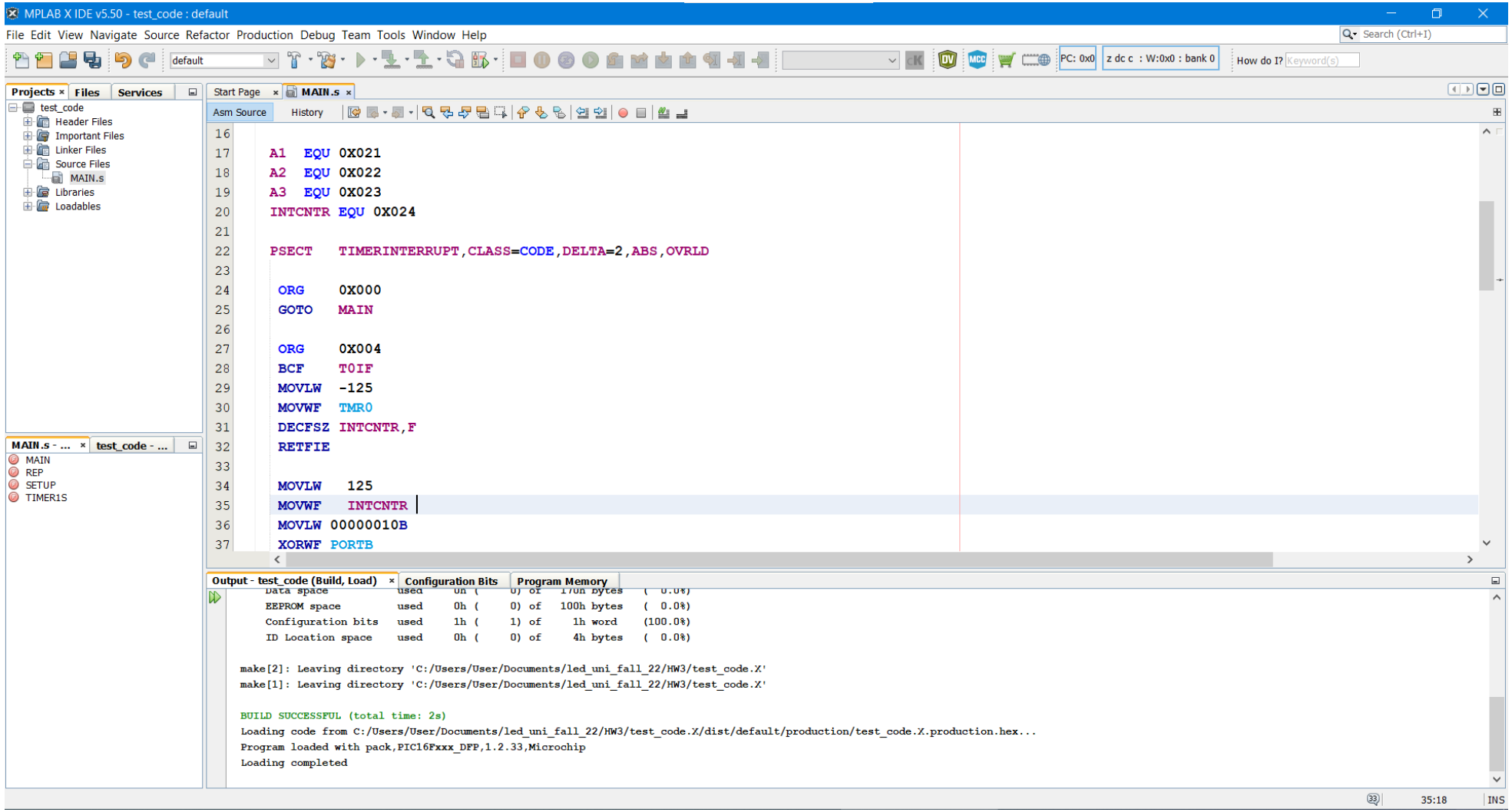
- IDE means an Integrated Development Environment: a software application that helps programmers develop software code efficiently.
- It increases developer productivity by combining capabilities such as software editing, building, testing, and packaging in an easy-to-use application.

PIC programming software

- MPLABX IDE needs a compiler/assembler to convert the assembly or C codes into machine language.
- What is a compiler: It is a translator that convert the C coding into a 0/1 files ready to be deployed on the chip.
- What is an assembler: It is a translator that converts the assembly coding into a 0/1 files ready to be deployed on the chip
- What is a machine language: It is the set of binary values that will be loaded in the PIC Flash memory. It is generated by the assembler and the compiler.
- Lately, Microchip created a compiler called XC8 used to translate the C and assembly codes for the midrange family

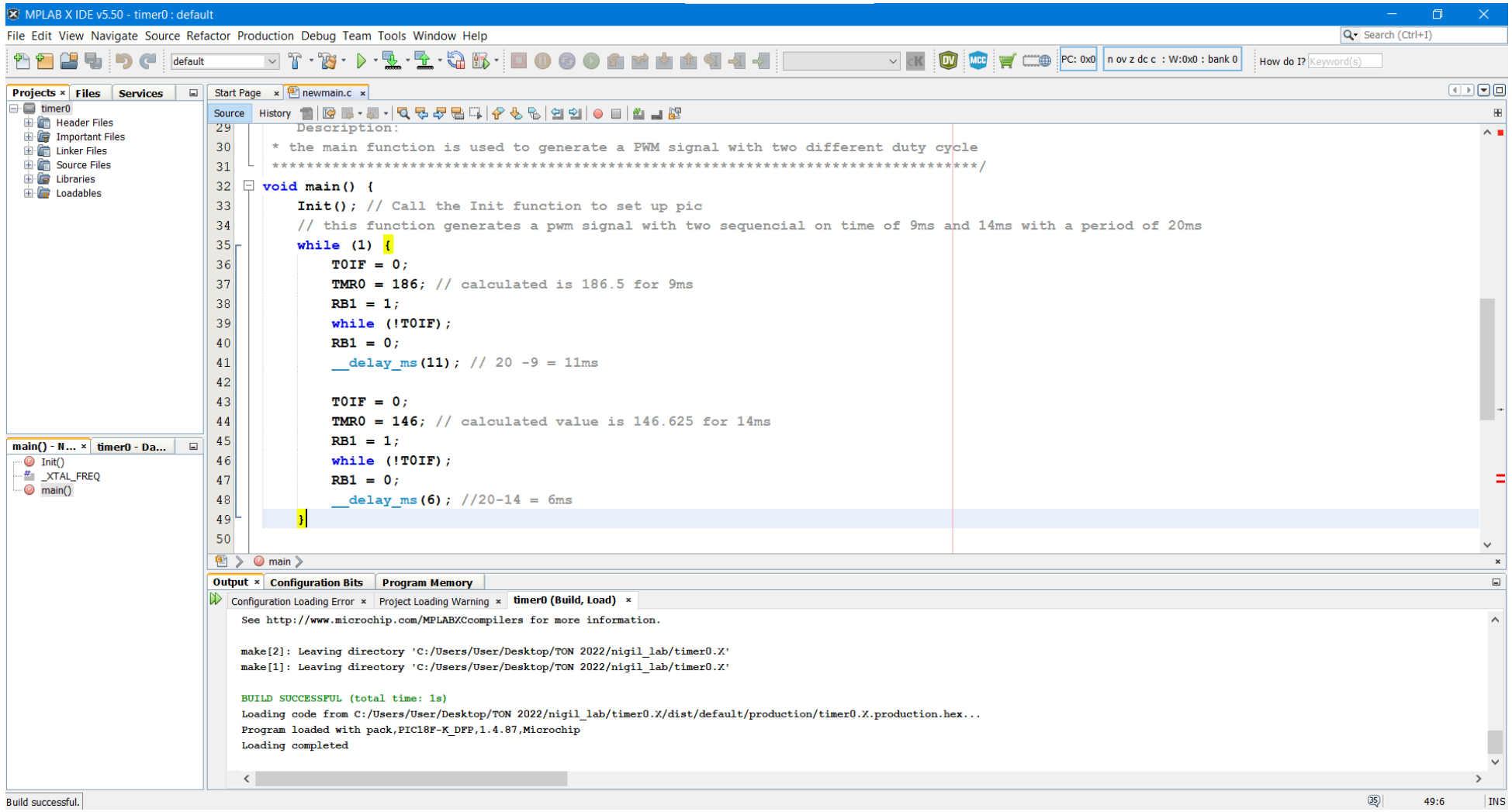
PIC programming software

Demo code written in assembly language



PIC programming software

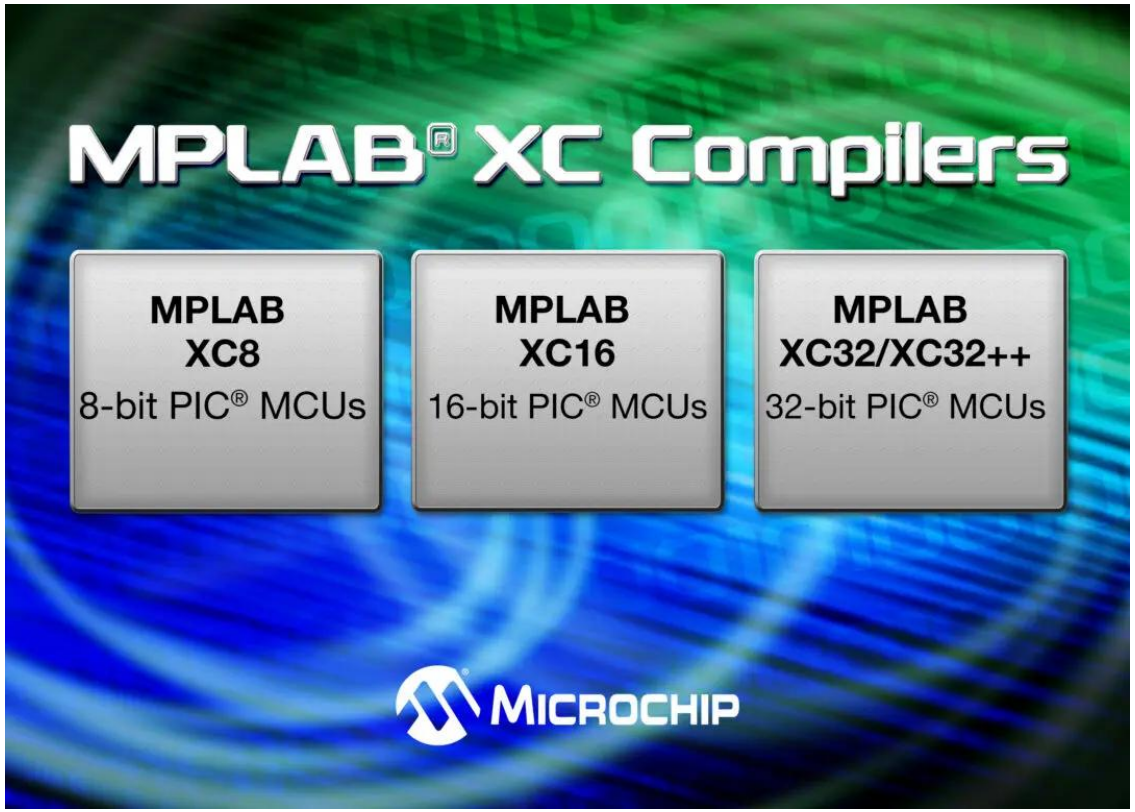
Demo code written in C language



PIC programming software

Microchip also designed other compilers

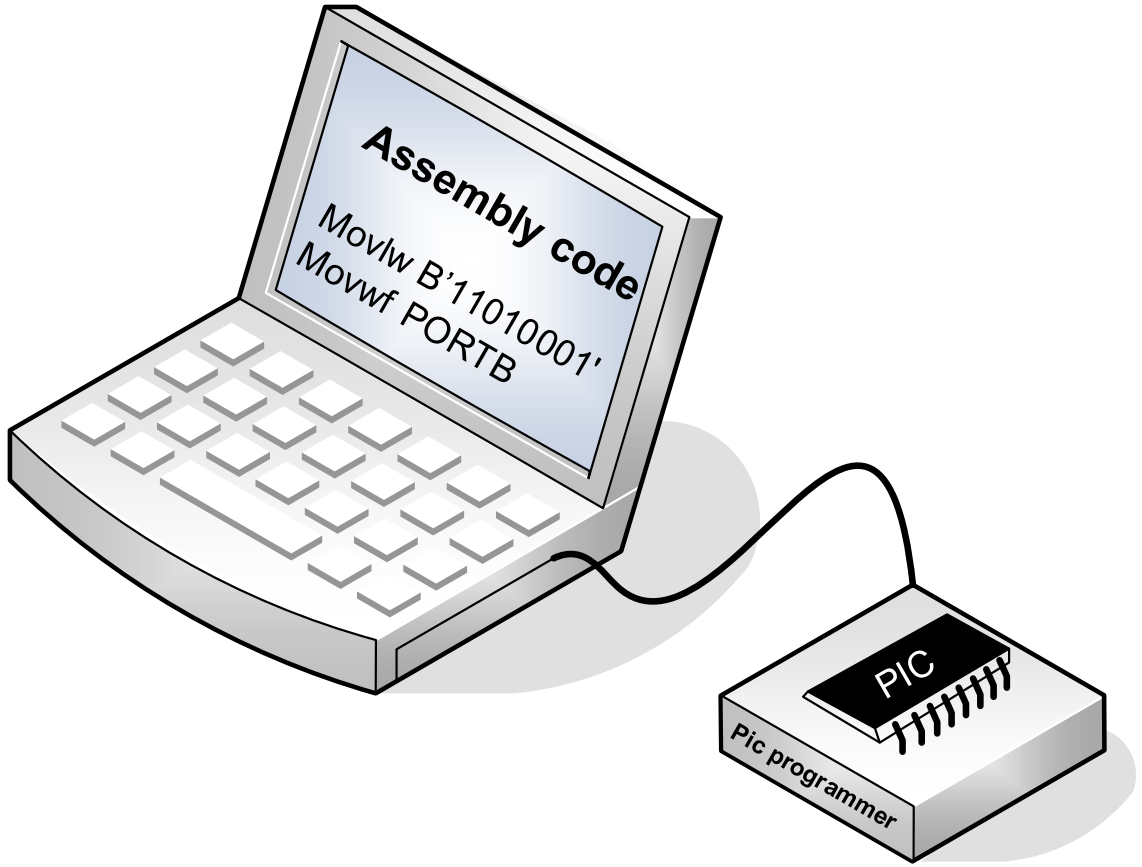
- for the 16bit MCU like the PIC24 family
- and other compilers for 32bit MCU like PIC32 families



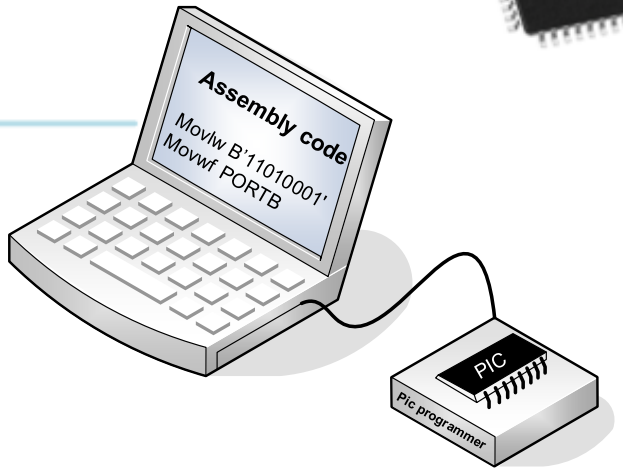
PIC Programming tools

- After writing and compiling a code, it is time to move the binary file to the PIC flash memory. This can be done using a PIC programmer

- Again we have lot of choices
  - Official programming tools
  - Third-party programming tool



PIC Programming hardware



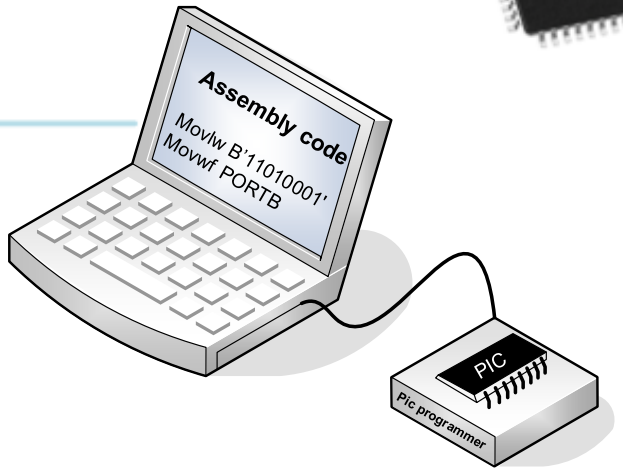
Official programmer

- The MPLAB PICkit Programmer allows fast and easy debugging and programming of PIC, dsPIC, AVR, SAM and CEC flash microcontrollers (MCUs) and microprocessors (MPUs), using the powerful graphical user interface of MPLAB X Integrated Development Environment (IDE)



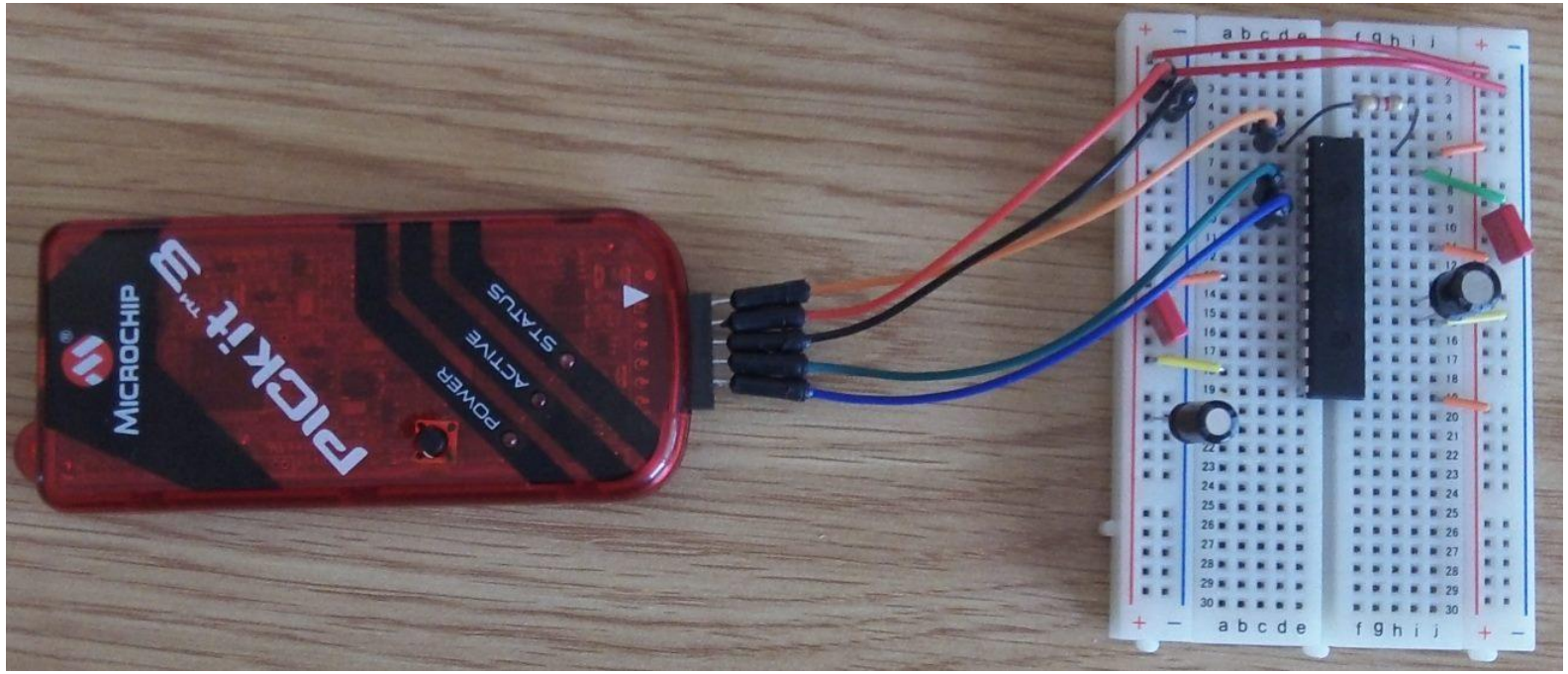


PIC Programming hardware



PICKit can be connected to the microcontroller through 5 wires only, which represents the programming bus.

The PIC is already placed on the breadboard with the necessary components

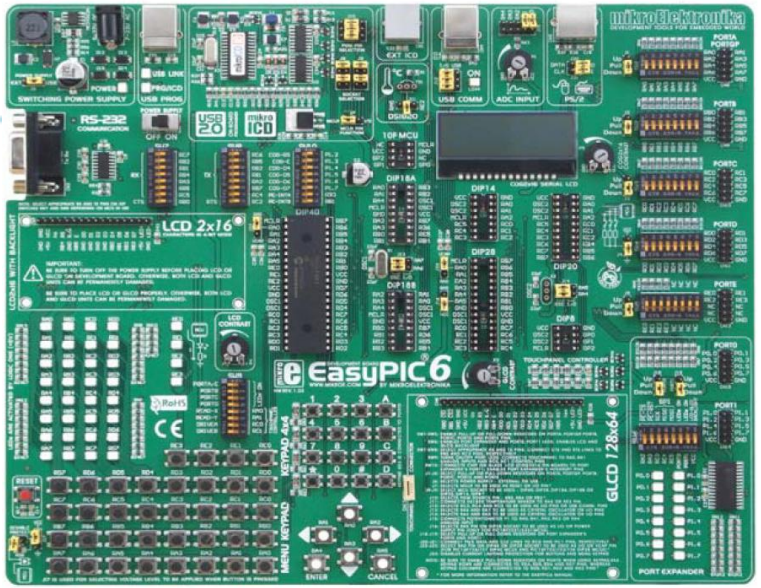


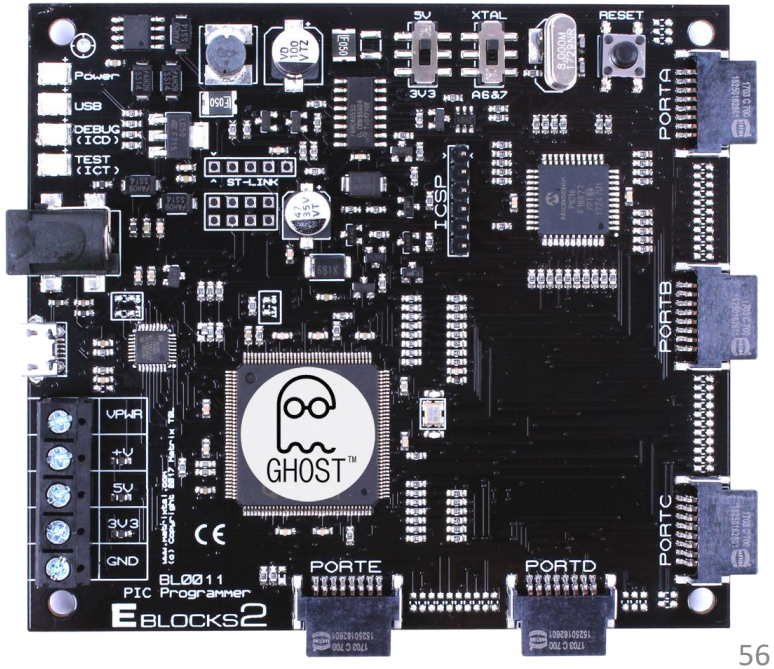
PIC development tools

In order to reduce the hardware complexity, Microchip and other third party companies designed Development tools.

These tools can be easily attached to the programmer or it can be designed to be a programmer and a development tool at the same time.

Development boards have some on board switches, Push buttons, Leds, LCD and many other sensor/actuators.







Week 2

Lecture

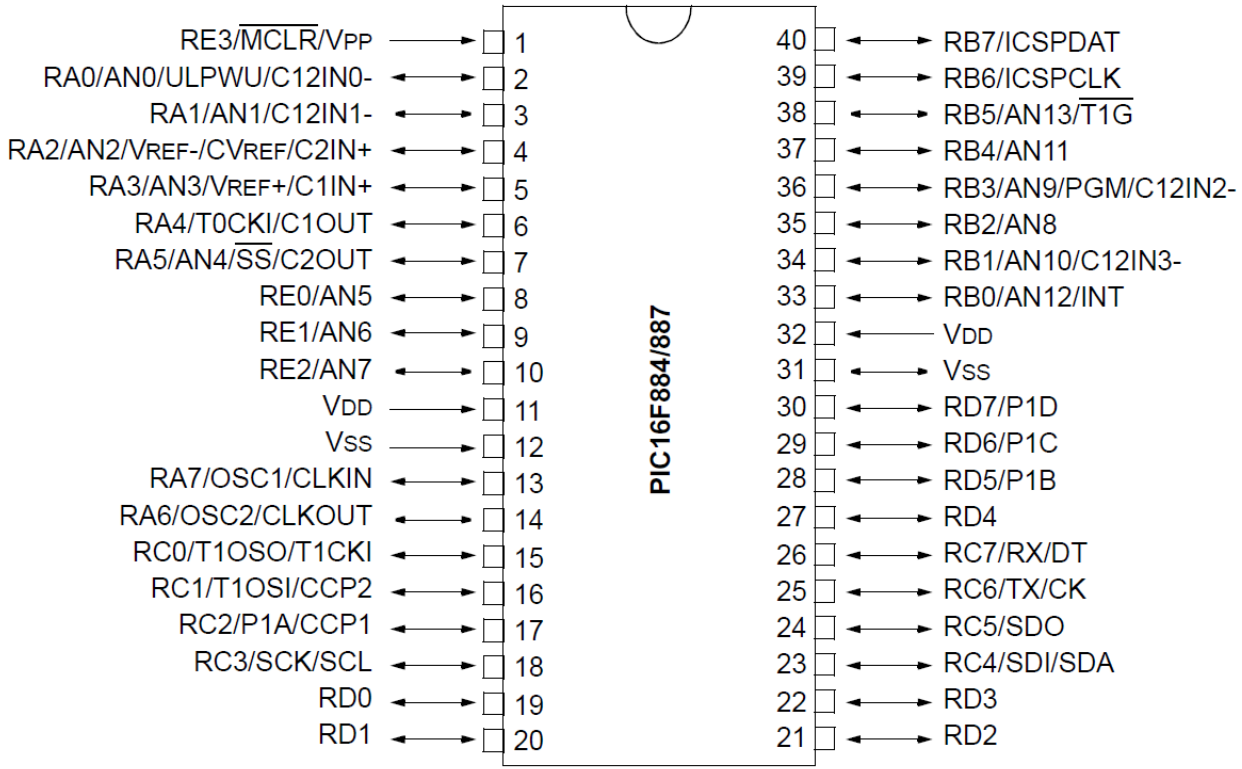
Lecture 3 / LO1

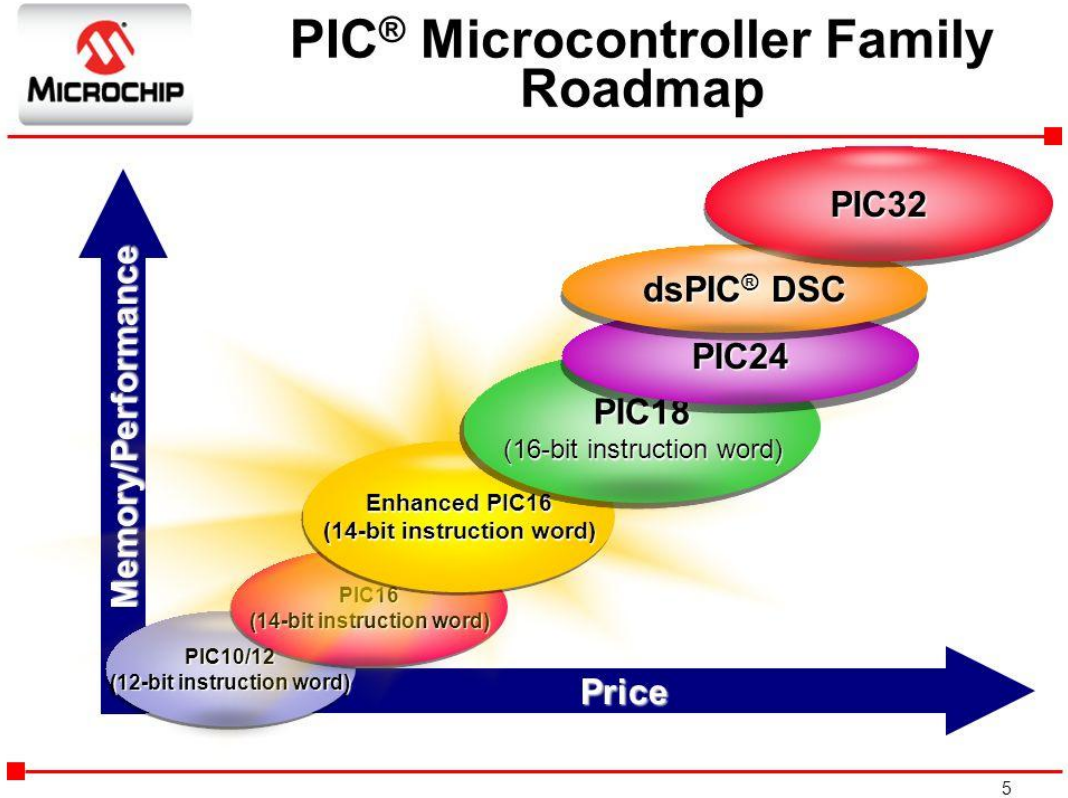
Overview on the PIC16F887



What is the microcontroller PIC16F887

- The PIC16F887 is one of the devices manufactured by Microchip.
- This PIC is in the midrange family **14bit instruction word**
- It is an **8bit** Microcontroller







What is the microcontroller PIC16F887

The most important specifications of this chip are

1: 35 input/output pins
2: Flash memory (Program memory) of 8KWord
3: Ram of 368 bytes
4: EEPROM of 256 bytes
5: 8 internal crystals (oscillators)
6: 14 Analog to Digital channels
7: Three timers
8: Max running frequency of 20Mhz
9: Wide operating voltage (from 2 to 5.5 volts)

- From now on, you have to use the PIC datasheet for extra information.

What's inside the Chip

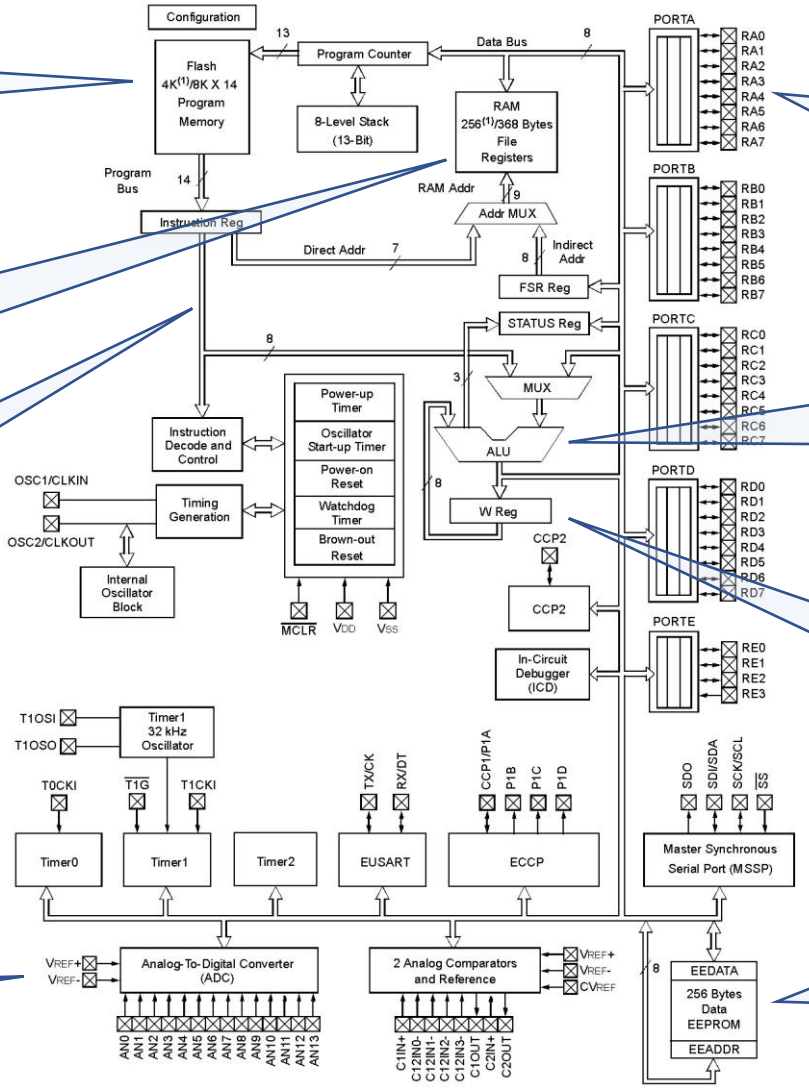
Flash Memory

RAM

Bus

3 timers

A2D converter



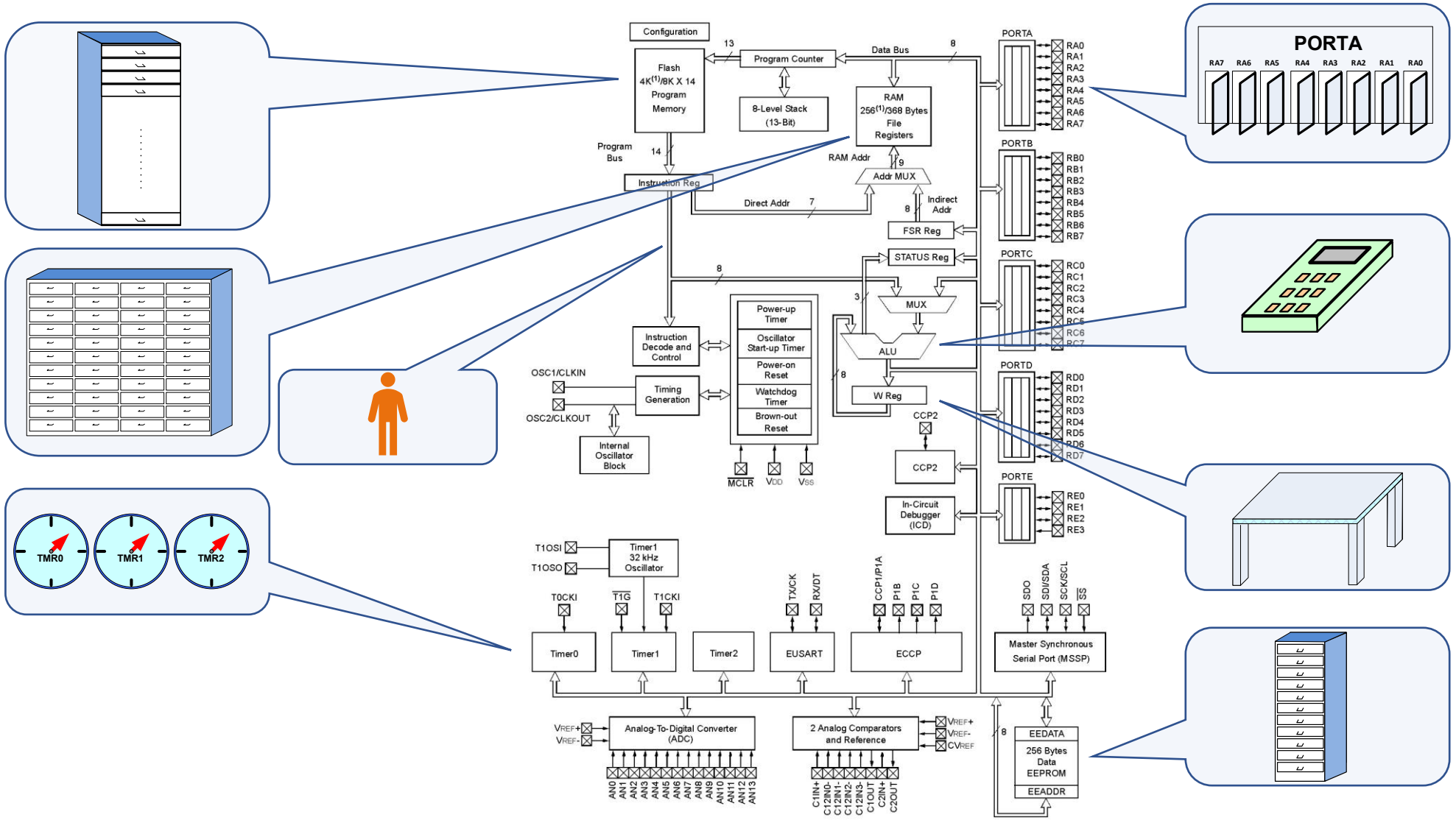
PORTS

Arithmetic Logic Unit

Working register

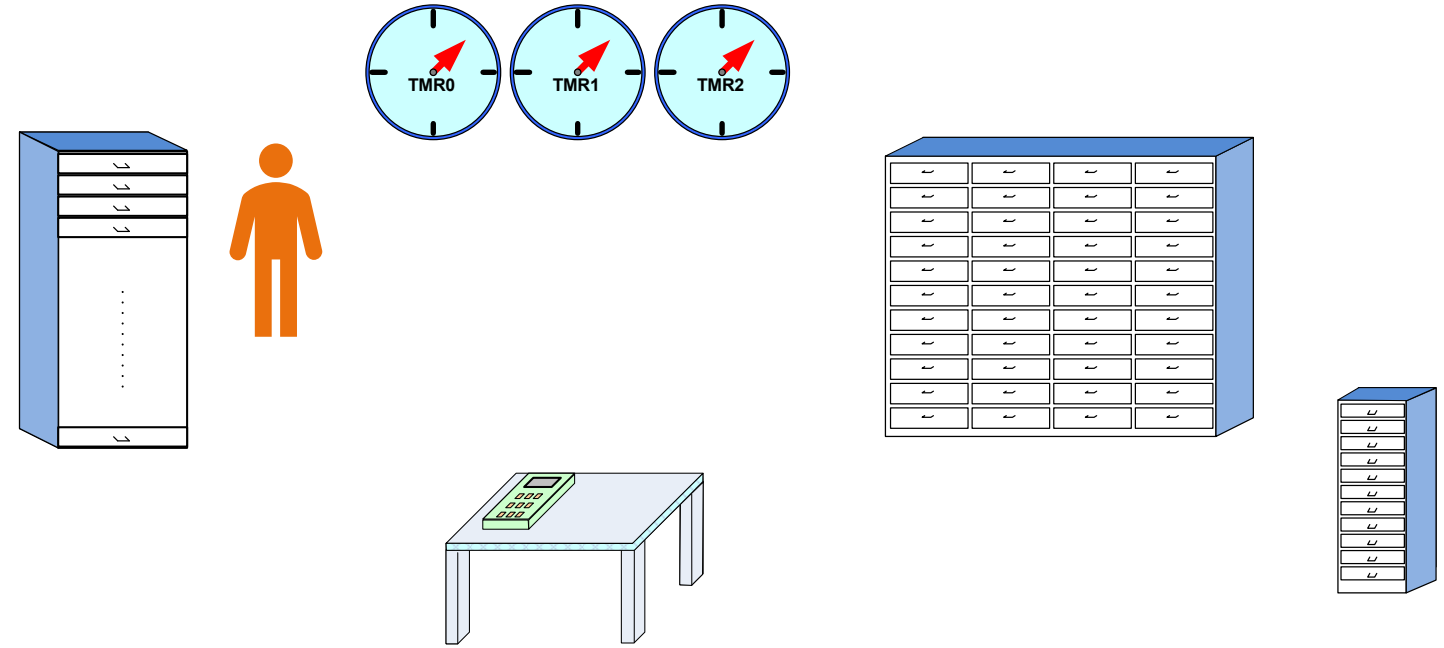
EEPROM

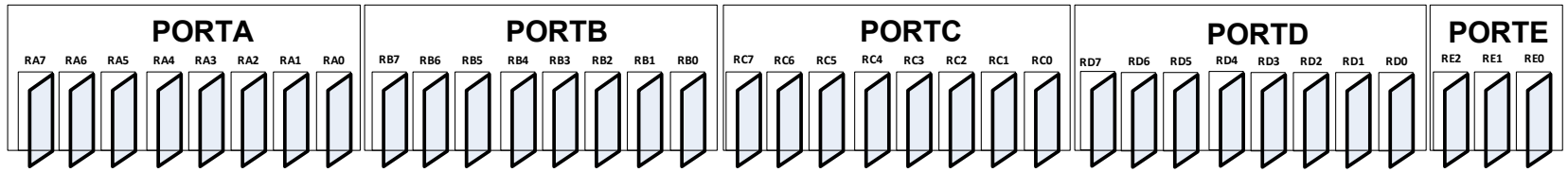
What's inside the Chip



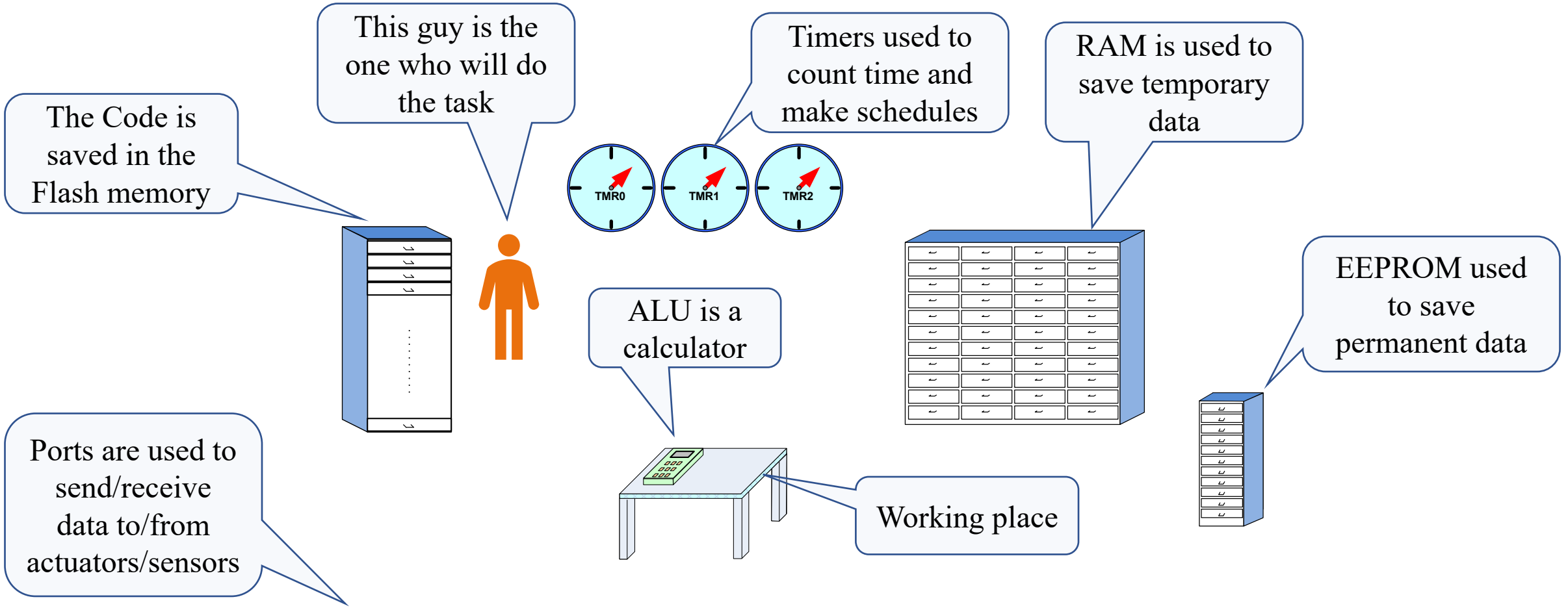
What's inside the Chip

In order to program any microcontroller, you have to be aware of the hardware structure of the chip. Just imagine that you have the following items inside the PIC



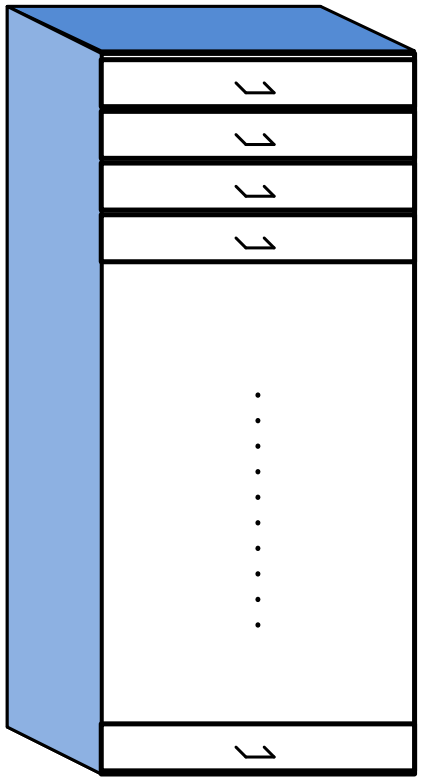


What's inside the Chip



What's inside the Chip (Flash Memory)

- The Flash memory is like an Armoire with a set of drawers. It is the location in which the machine language code is saved.
- The machine language is the translated version of the code written in Assembly or C languages…



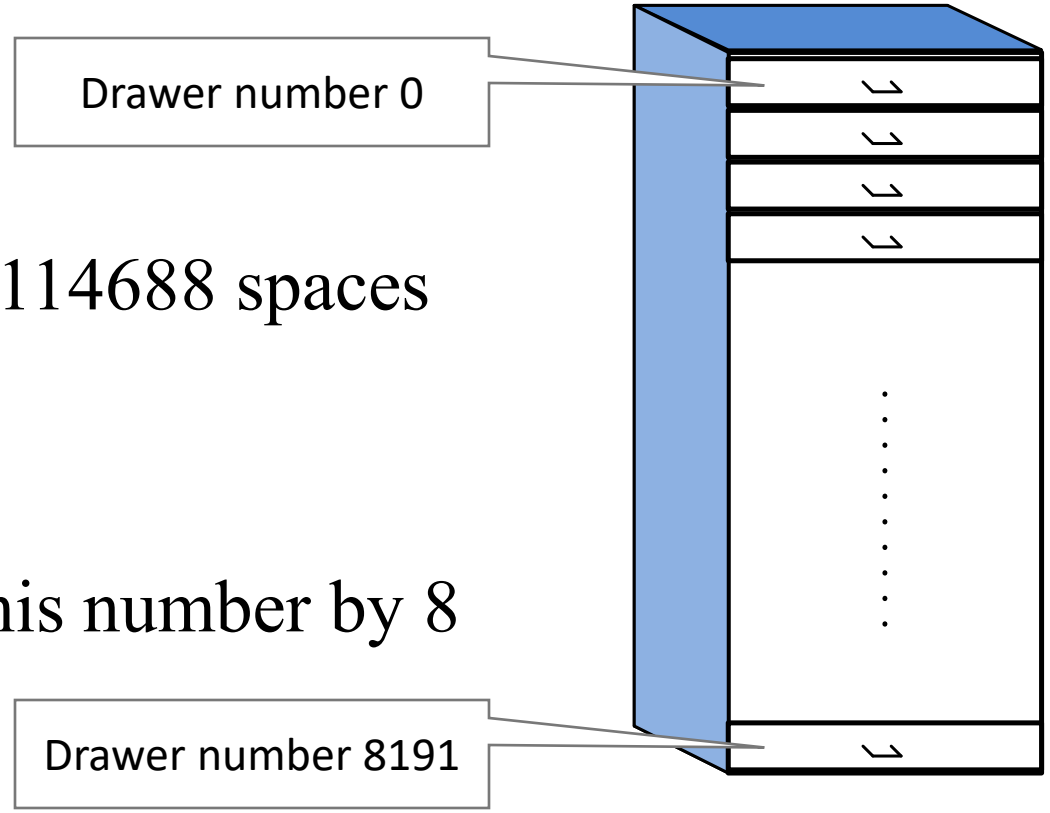
- The Flash memory holds the instruction required to make a special process. Every instruction is saved in a drawer.
- Like for example, how to make a pizza, how to control the temperature of a room or any other process.

- Since this memory is used to save the operating program, it is also called program memory

- NB: this memory is programmed using a special programmer

What's inside the Chip (Flash Memory)

- The Armoire of the PIC16F887 has 8192 drawers (8K)
- Each drawer can hold 14bits (spaces)
- So, it is called 8KWord of 14 bits



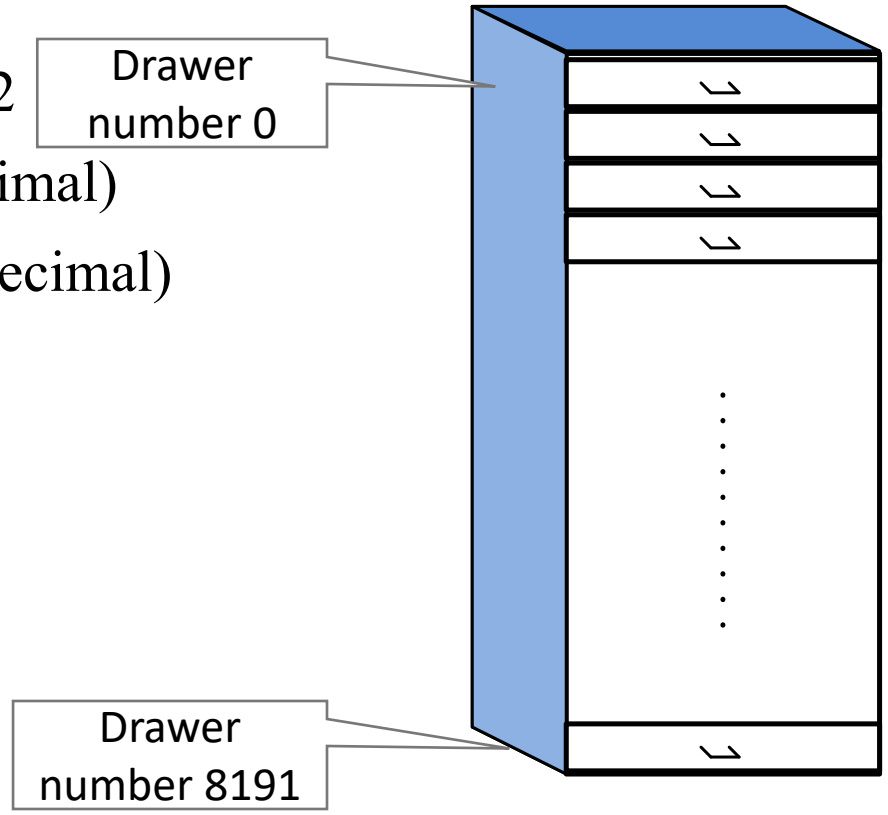
- The total number of spaces is 8192 x 14 = 114688 spaces
- Or more precisely, 114688 bits

- To know what is the byte size, we divide this number by 8

- 114688/8 = 14336 Bytes = 14Kbytes

What's inside the Chip (Flash Memory)

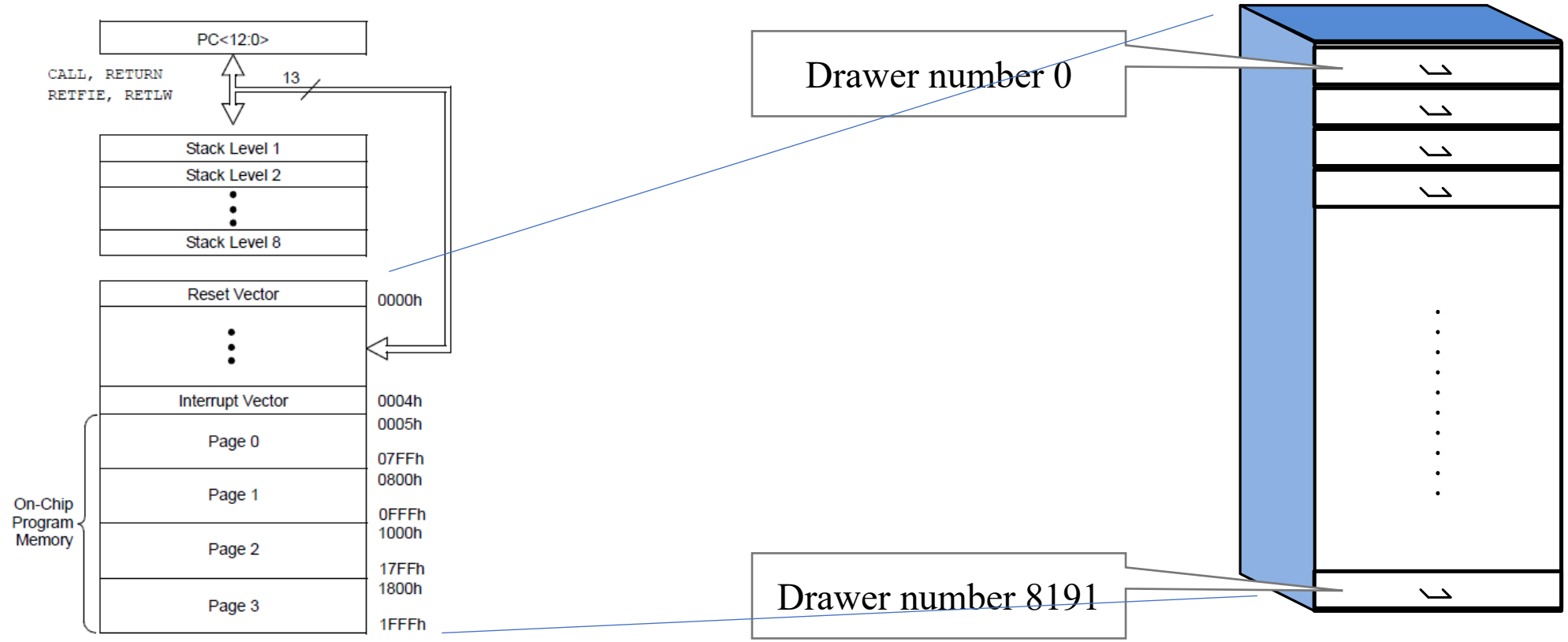
- As mentioned before, the Armoire of the PIC16F887 has 8192 drawers (8K)
- Each drawer has an address (0, 1, 2…8191)
- We need 13 address lines to access these memories. 2^{13} =8192
- The address of the first drawer is 0,0000,0000,0000 (0 in decimal)
- The address of the last drawer is 1,1111,1111,1111 (8191 in decimal)



- It is better to describe the address in Hex format, so,
- The address of the first drawer will be 0000 in Hex
- The address of the last drawer will be 1FFF in Hex

What's inside the Chip (Flash Memory)

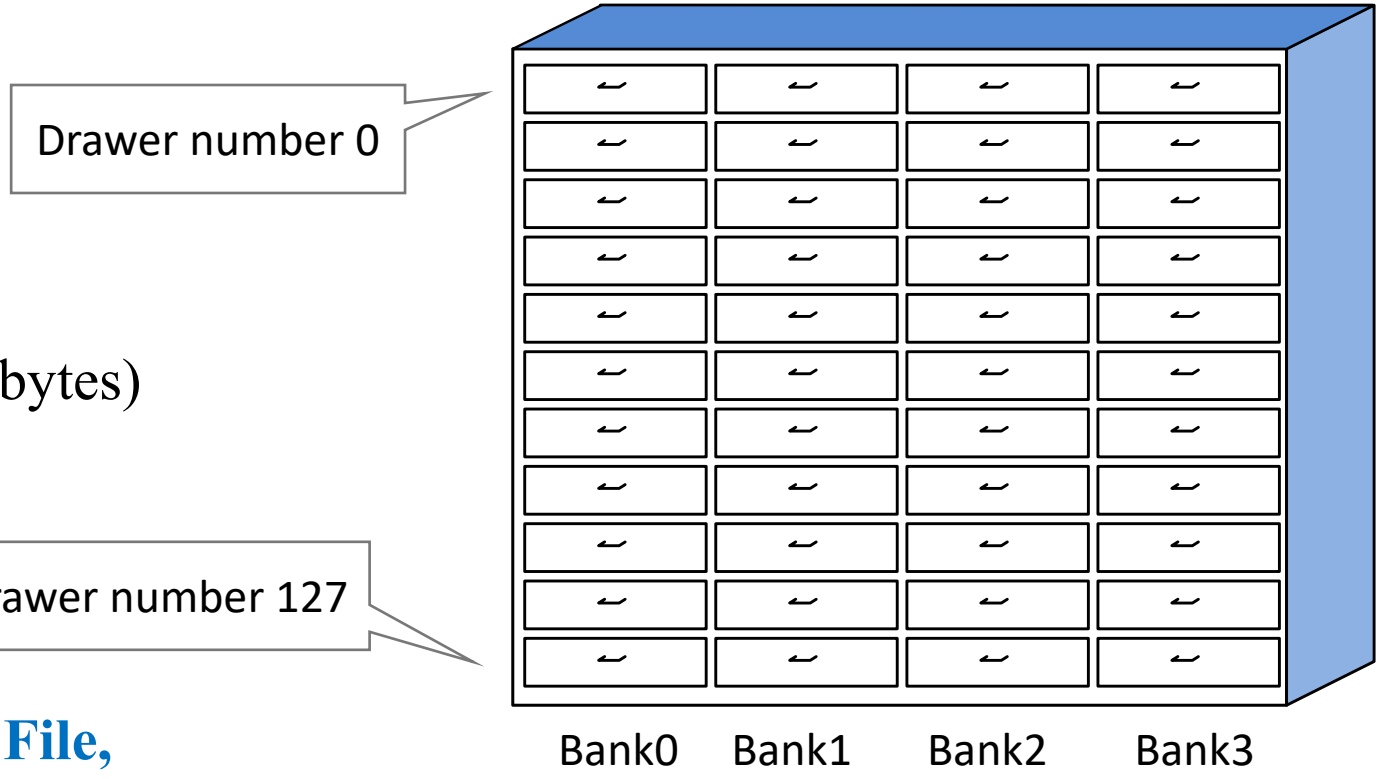
- As per the datasheet, the flash memory starts by a reset vector, in which the program will start from address zero when the chip is powered up.
- Also the flash memory is made of 4 pages (will be discussed later)



Flash memory (from datasheet)

What's inside the Chip (RAM)

- The RAM is also like an Armoire with drawers. It is made of 4 banks called bank0, Bank1, Bank2, and Bank3.
- Each bank has 128 drawers (512 in total)
- Each drawer can hold 8 bits



- Some of the drawers are not accessible
- And some others are duplicated
- Only 368 drawers are free to use, i.e., 368 bytes)

The data inside the drawers is volatile

NB: Drawer is an acronym to Register, or File, or Location memory

What's inside the Chip (RAM)

- Some of the drawers are used for the "settings" of the PIC, called Special Function Registers (SFR)
- They have special name and they should be carefully loaded by special values
- Others registers are free to use. They can be named by any name, and loaded by any value, thus called General Purpose Register (GPR)
- The value that can fit into the registers ranges from 0 to 255. (8 bit register)

A memory location is called FILE or Register



What's inside the Chip (RAM)

- Each bank has 128 drawers, thus in total 512.
- Some are taken (SFR)
- And some others are free to use (GPR)

- The total free to use registers are
  1. 96 in bank 0 (starting from address 20H)
  2. 80 in bank 1 (starting from address A0)
  3. 96 in bank 2 (starting from address 110H)
  4. 96 in bank 3 (starting from address 190H)
- Total of 368 free space

The registers already named are special

The empty spaces are the general registers

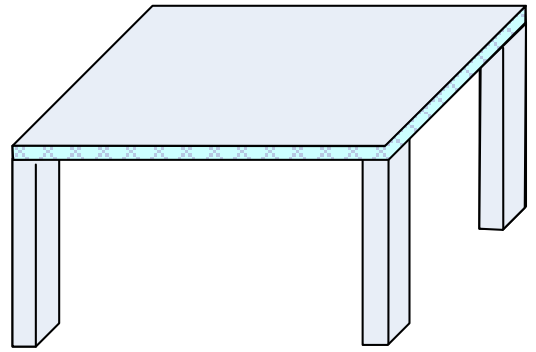
RAM (from datasheet)

Bank 0	File Address	Bank 1	File Address	Bank 2	File Address	Bank 3	File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers 16 Bytes	116h	General Purpose Registers 16 Bytes	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h–6Fh	General Purpose Registers 80 Bytes	A0h–EFh	General Purpose Registers 80 Bytes	120h–16Fh	General Purpose Registers 80 Bytes	1A0h–1EFh
	70h–7Fh	accesses 70h-7Fh	F0h–FFh	accesses 70h-7Fh	170h–17Fh	accesses 70h-7Fh	1F0h–1FFh

Unimplemented data memory locations, read as '0'.
Note 1: Not a physical register.
2: PIC16F887 only.

What's inside the Chip (W register)

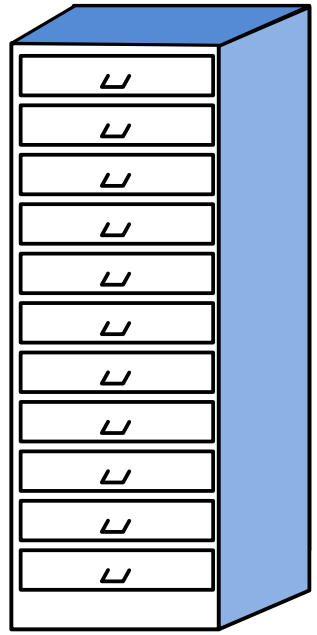
- The Working register is the place in which the task is done.
- It is a temporary location to place some values taken from/to the RAM of taken from the Flash memory



- The table can fit only one byte
- Thus the value that can be loaded is from 0 to 255

What's inside the Chip (EEPROM)

- EEPROM is also an Armoire for not volatile values.
- It is made of 256 drawers with 8bit capacity.

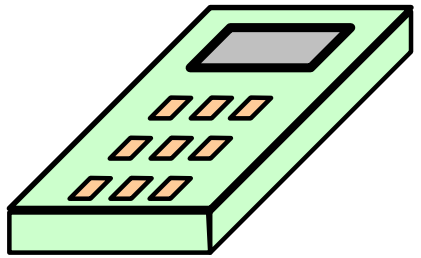


- EEPROM is the abbreviation of Electrical Erasable Programable Read-Only Memory

- This memory can be used to save some important data during run-time and keep them saved even at power off

What's inside the Chip (ALU)

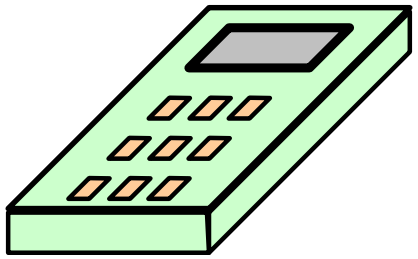
- The ALU is the Arithmetic/Logic unit. It is like a calculator that can do:
  - Arithmetic operations (Addition, Subtraction…)
  - Logic operations (ANDing, ORing, XORing…)



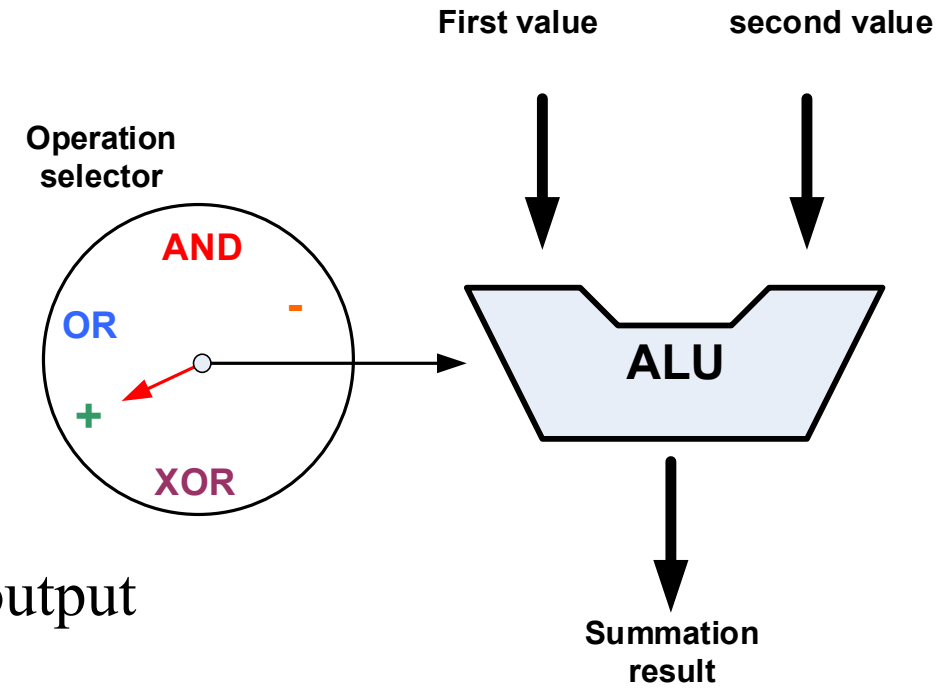
- In the PIC midrange family, the ALU cannot perform multiplication or division. Thus if we need to perform complicated operations, we have to make it based on the addition/subtraction operations using programming skills.

What's inside the Chip (ALU)

- This calculator differ from the traditional one (the Casio for example) and works as follows:
  1. Enter the first value
  2. Enter the second value
  3. Select the operation

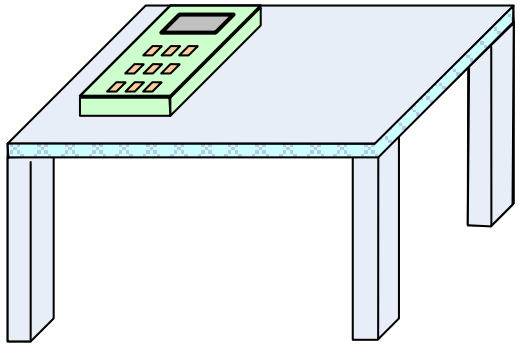


- For example: 3 + 4 is done as follow
  1. Enter 3
  2. Enter 4
  3. Select addition
     - The result will be fetched out on the calculator output

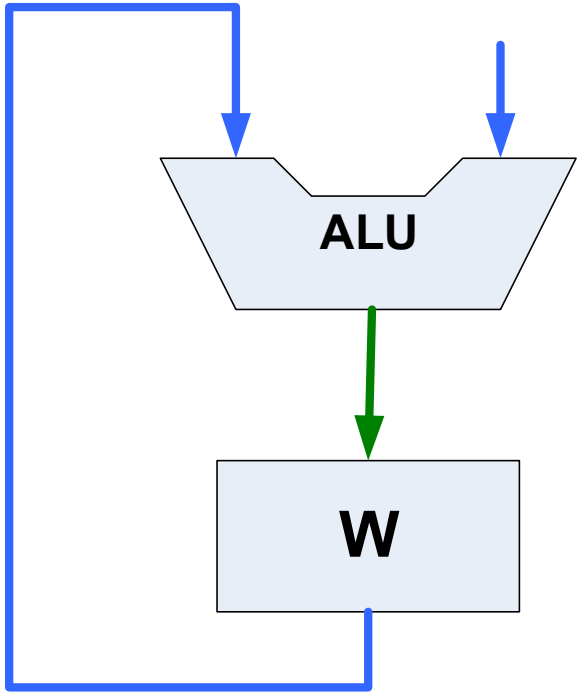


What's inside the Chip (ALU)

- You have to imagine now that the value appearing on the calculator will be placed on the table, and vice-versa, the value placed on the table is already entered as first operand on the calculator

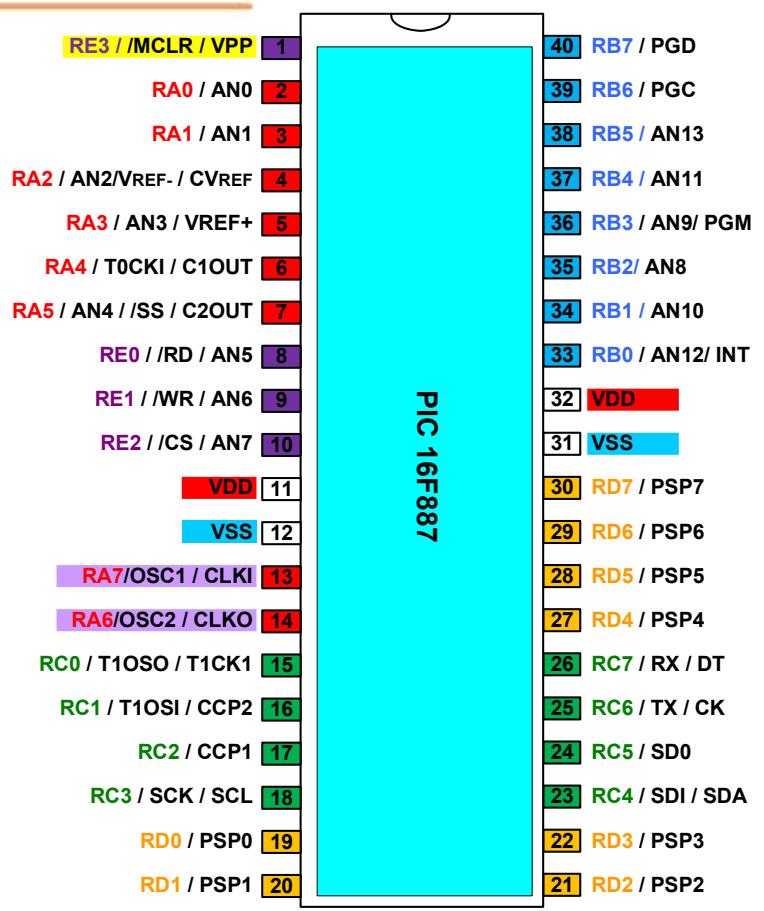


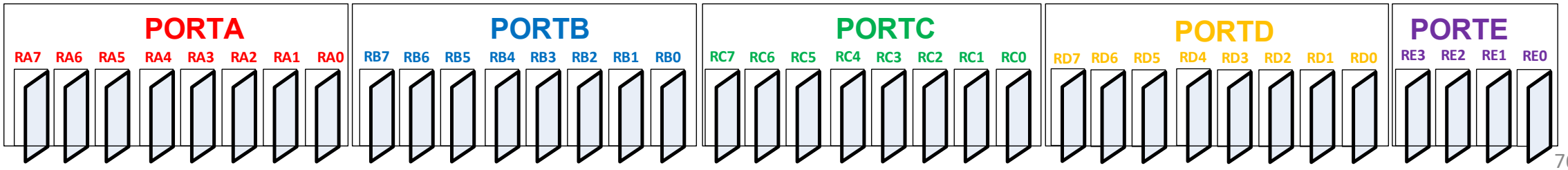
- In other words, the operation 3 + 4 leads to 7
  - 7 will be placed on the W register
  - also entered to the calculator
  - And is ready for another operation



What's inside the Chip (PORTS)

- Any chip has pins or PORTS used to exchange data with the external environment. For example reading data from sensors or writing to actuators.
- These are called the input/output pins
- The PIC16F887 has 40 pins. Only 36 are called PORTS. Other pins are for power supply.
- PORTS are usually grouped by 8 pins



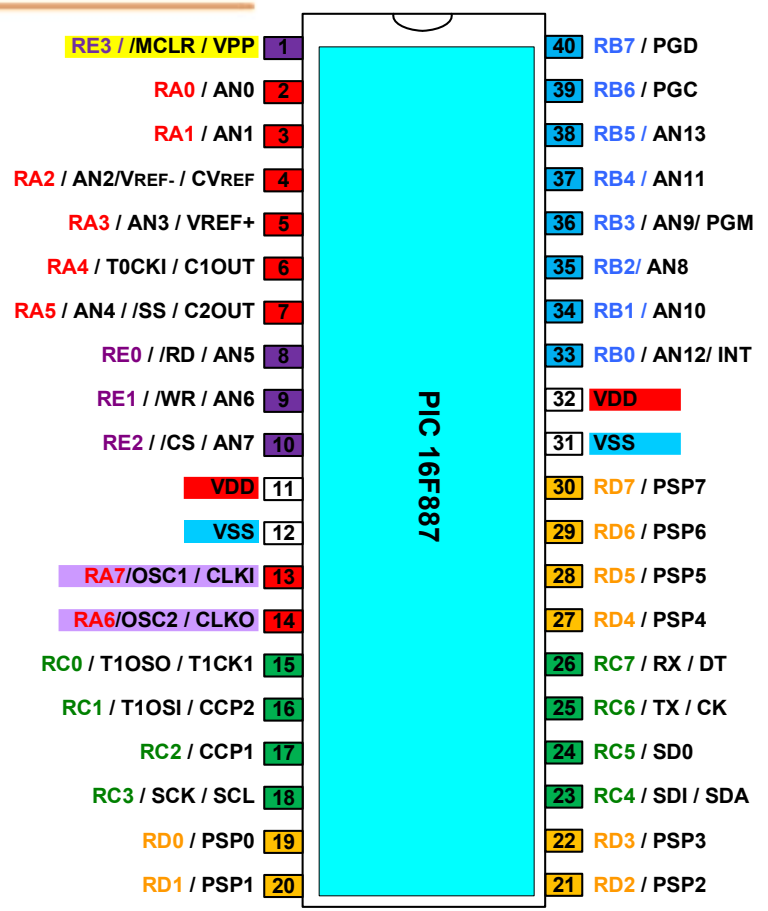


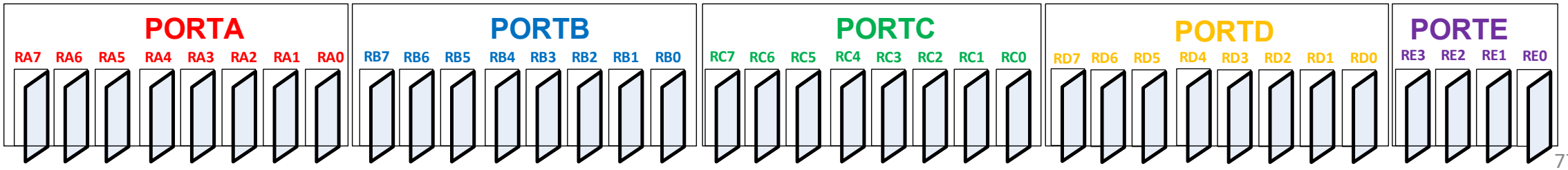
What's inside the Chip (PORTS)

PIC16F887 has

- PORTA: 8 pins
- PORTB: 8pins
- PORTC: 8pins
- PORTD: 8pins
- PORTE: 4 pins

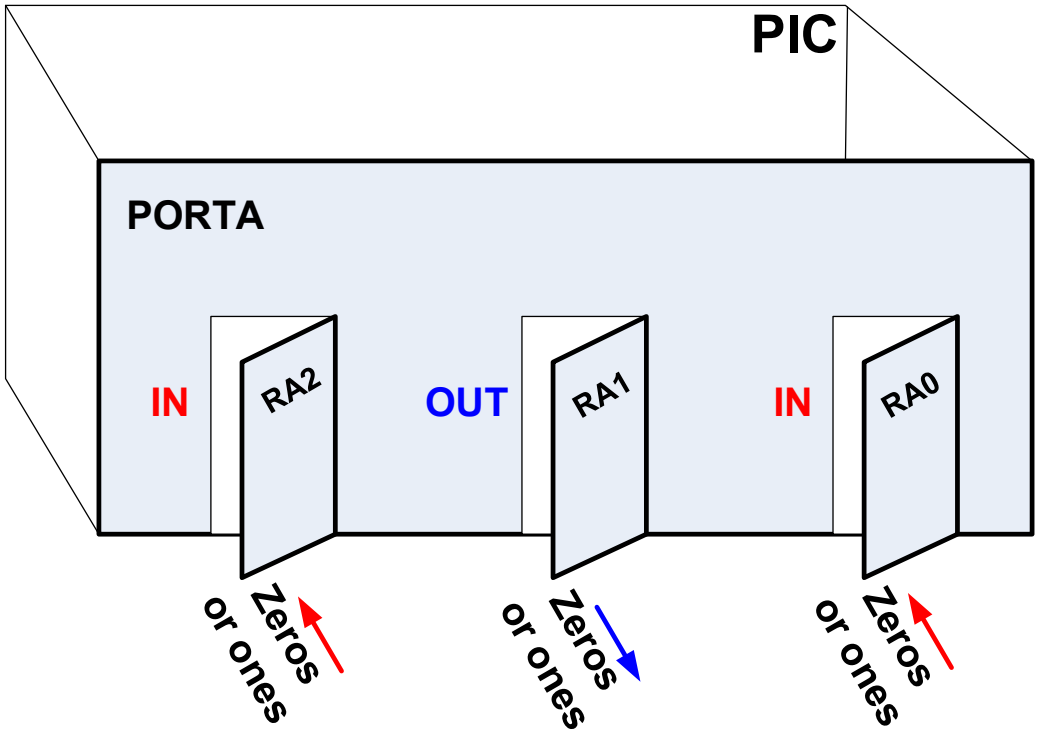
TOTAL of 36 input/output pins





What's inside the Chip (PORTS)

- Any pin can be configured as input or output.
- Input means that the PIC is reading data (0 or 1) from a sensor.
- Output means that the PIC is sending data (0 or 1) to an actuator
- The PIC reads the data in digital form. i.e., TTL logic level
  - TTL level means 0/5 volts
  - Logic 0 is 0 volt
  - Logic 1 is 5 volts



NB: in some new technologies, Logic 1 is 3.3V



Test your knowledge

1. Assume that we are dealing with a PIC having 4kWord of 14 bit Flash memory. Calculate the total number of bits and the size in byte.
2. How many address lines are required to address a 64kByte memory
3. Having a USB flash memory of 4GByte, how many address lines are required to access all the memory locations
4. What is the range of addresses of each bank in RAM of the PIC16F887
5. What is the abbreviation of SFR and GPR
6. What is the maximum clocking speed of the PIC16F887
7. How many pins and how many I/O pins the PIC16F887 has
8. What is the maximum value that can be placed in W register
9. If a microcontroller memory has 14-bits address lines, hence the address range in hexadecimal will be_____.
10. The _____ memory of the PIC16F887 has four banks.



Week 2

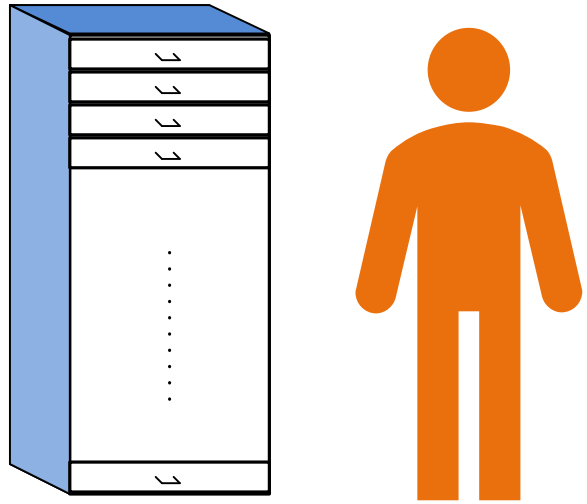
Lecture

Lecture 4 / LO1

Assembly Language

Assembly Language

- The assembly language for the PIC16F887 is made of 35 different instructions
- An instruction is an order to be executed by the guy
- The instructions are placed in the Flash memory
- Each instruction in a drawer (i.e., memory location)



- The guy opens the first drawer, reads the instruction and then executes it.
- When finished, he moves automatically to the next drawer and executes it….

Assembly Language

Instructions in assembly

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS						
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff	C, DC, Z	1, 2
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff	Z	1, 2
CLRF	f	Clear f	1	00 0001 lfff ffff	Z	2
CLRW	–	Clear W	1	00 0001 0xxx xxxx	Z	
COMF	f, d	Complement f	1	00 1001 dfff ffff	Z	1, 2
DECF	f, d	Decrement f	1	00 0011 dfff ffff	Z	1, 2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00 1011 dfff ffff		1, 2, 3
INCF	f, d	Increment f	1	00 1010 dfff ffff	Z	1, 2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00 1111 dfff ffff		1, 2, 3
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff	Z	1, 2
MOVF	f, d	Move f	1	00 1000 dfff ffff	Z	1, 2
MOVWF	f	Move W to f	1	00 0000 lfff ffff		
NOP	–	No Operation	1	00 0000 0xx0 0000		
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff	C	1, 2
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff	C	1, 2
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff	C, DC, Z	1, 2
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff		1, 2
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff	Z	1, 2
BIT-ORIENTED FILE REGISTER OPERATIONS						
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff		1, 2
BSF	f, b	Bit Set f	1	01 01bb bfff ffff		1, 2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff		3
LITERAL AND CONTROL OPERATIONS						
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk	C, DC, Z	
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk	Z	
CALL	k	Call Subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	–	Clear Watchdog Timer	1	00 0000 0110 0100	$\overline{TO}$, $\overline{PD}$	
GOTO	k	Go to address	2	10 1kkk kkkk kkkk		
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z	
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	–	Return from interrupt	2	00 0000 0000 1001		
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	–	Return from Subroutine	2	00 0000 0000 1000		
SLEEP	–	Go into Standby mode	1	00 0000 0110 0011	$\overline{TO}$, $\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk	C, DC, Z	
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z	

Every drawer in the flash memory is 14 bit wide

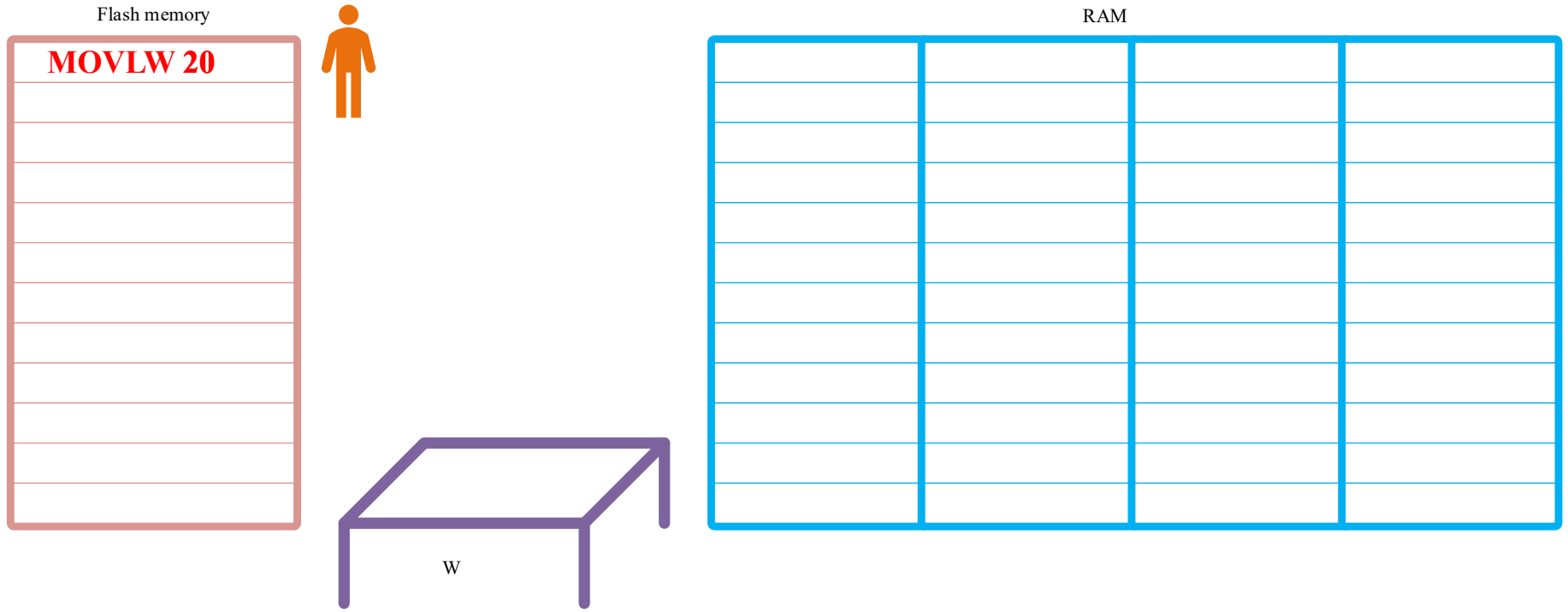
Machine language Equivalent (generated by the compiler/assembler

Cycle required to be executed

Lecture

Assembly Language (MOVLW)

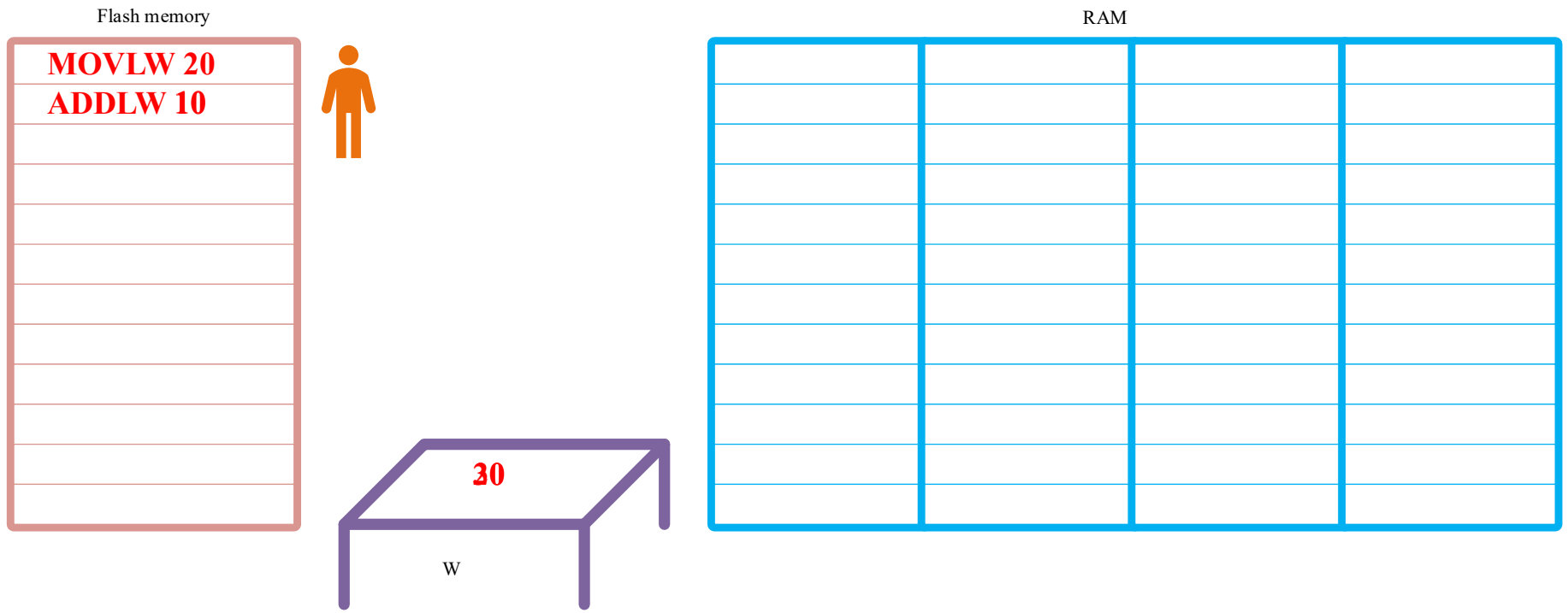
MOVLW	Move Literal to W
Syntax	MOVLW k
Operands	k is a literal value of 8 bit value (range from 0 to 255)
Operation	Copy k to W register
Example	**MOVLW 20**



Assembly Language (ADDLW)

ADDLW	Add Literal to W
Syntax	ADDLW k
Operands	k is a literal value of 8 bit value (range from 0 to 255)
Operation	Add k to W and place on W (W = k+W)
Example	**ADDLW 10**

Arithmetic Operation

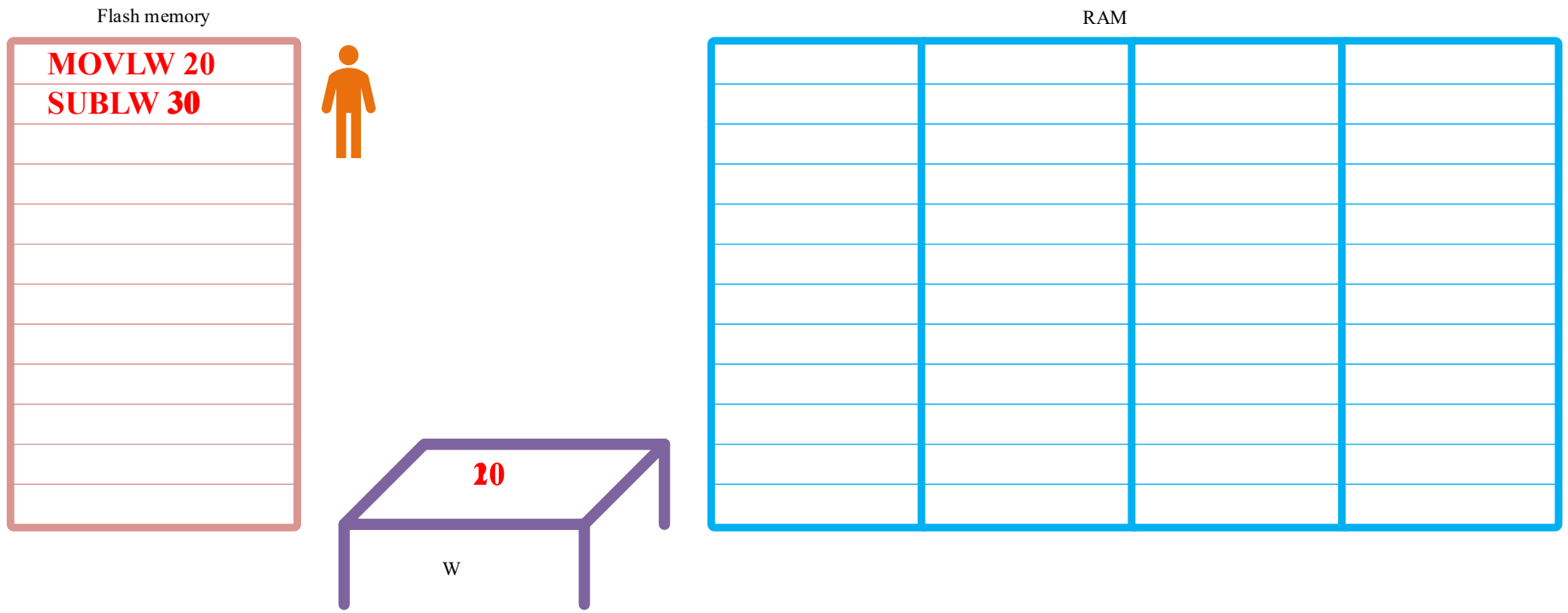


Click to play

Assembly Language (SUBLW)

SUBLW	Subtract W from Literal
Syntax	SUBLW k
Operands	k is a literal value of 8 bit value (range from 0 to 255)
Operation	Subtract W from k and place on W (W = k-W)
Example	**SUBLW 10**

Arithmetic Operation



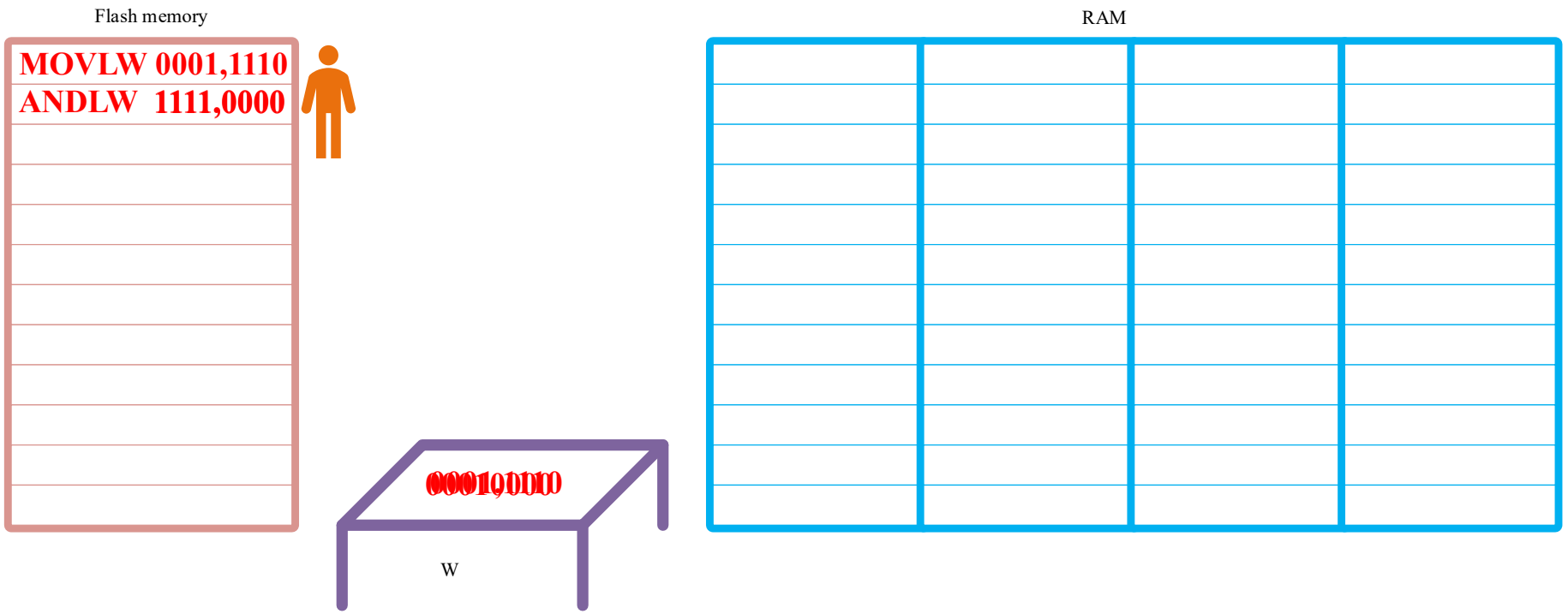
Click to play

Assembly Language (ANDLW)

ANDLW	AND Literal with W
Syntax	ANDLW k
Operands	k is a literal value of 8 bit value (range from 0 to 255)
Operation	AND k with W and place on W (W = k AND W)
Example	**ANDLW 11110000**

Logic Operation

Old W	0	0	0	1	1	1	1	0
K	1	1	1	1	0	0	0	0
New W	0	0	0	1	0	0	0	0



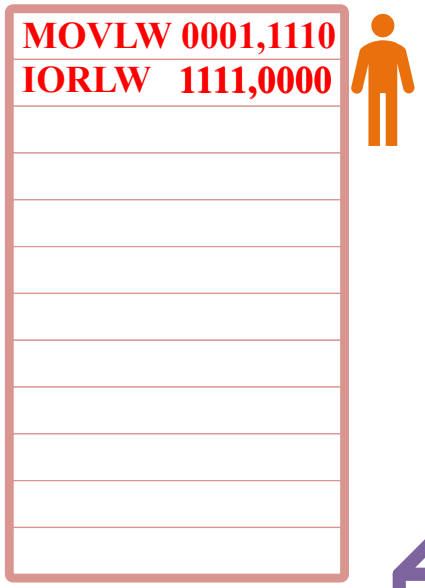
Assembly Language (IORLW)

IORLW	Inclusive OR Literal with W
Syntax	IORLW k
Operands	k is a literal value of 8 bit value (range from 0 to 255)
Operation	OR k with W and place on W (W = k OR W)
Example	**IORLW 11110000**

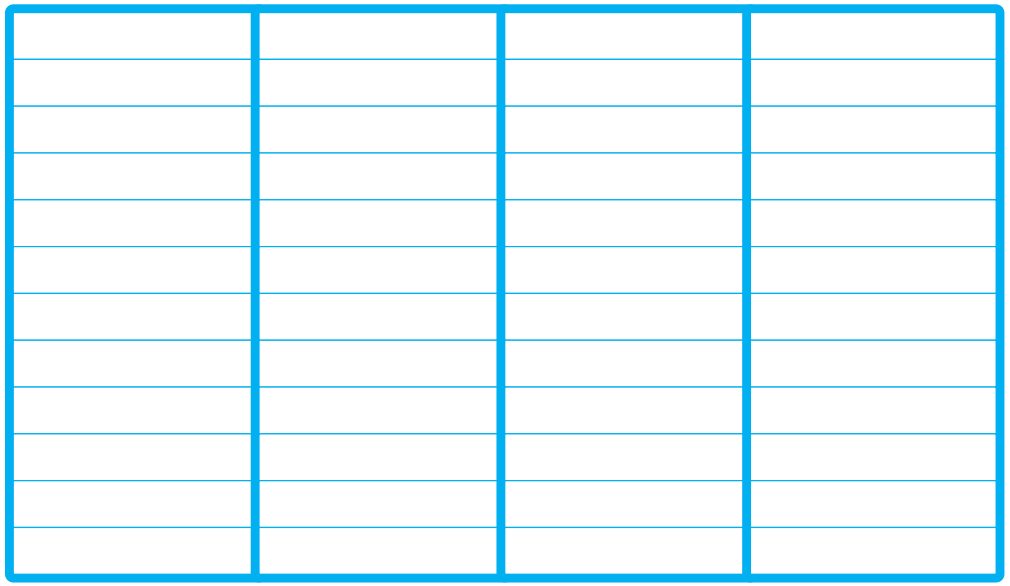
Logic Operation

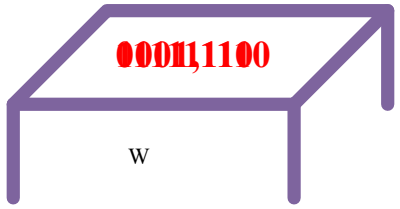
Old W	0	0	0	1	1	1	1	0
K	1	1	1	1	0	0	0	0
New W	1	1	1	1	1	1	1	0

Flash memory



RAM





Click to play

Assembly Language (XORLW)

XORLW	XOR Literal with W
Syntax	XORLW k
Operands	k is a literal value of 8 bit value (range from 0 to 255)
Operation	XOR k with W and place on W (W = k XOR W)
Example	**XORLW 11110000**

Logic Operation

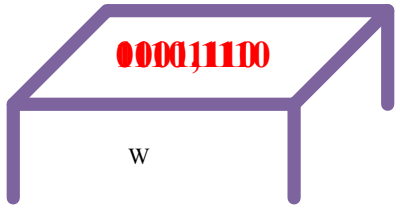
Old W	0	0	0	1	1	1	1	0
K	1	1	1	1	0	0	0	0
New W	1	1	1	0	1	1	1	0

Flash memory

MOVLW 0001,1110
XORLW 1111,0000



RAM



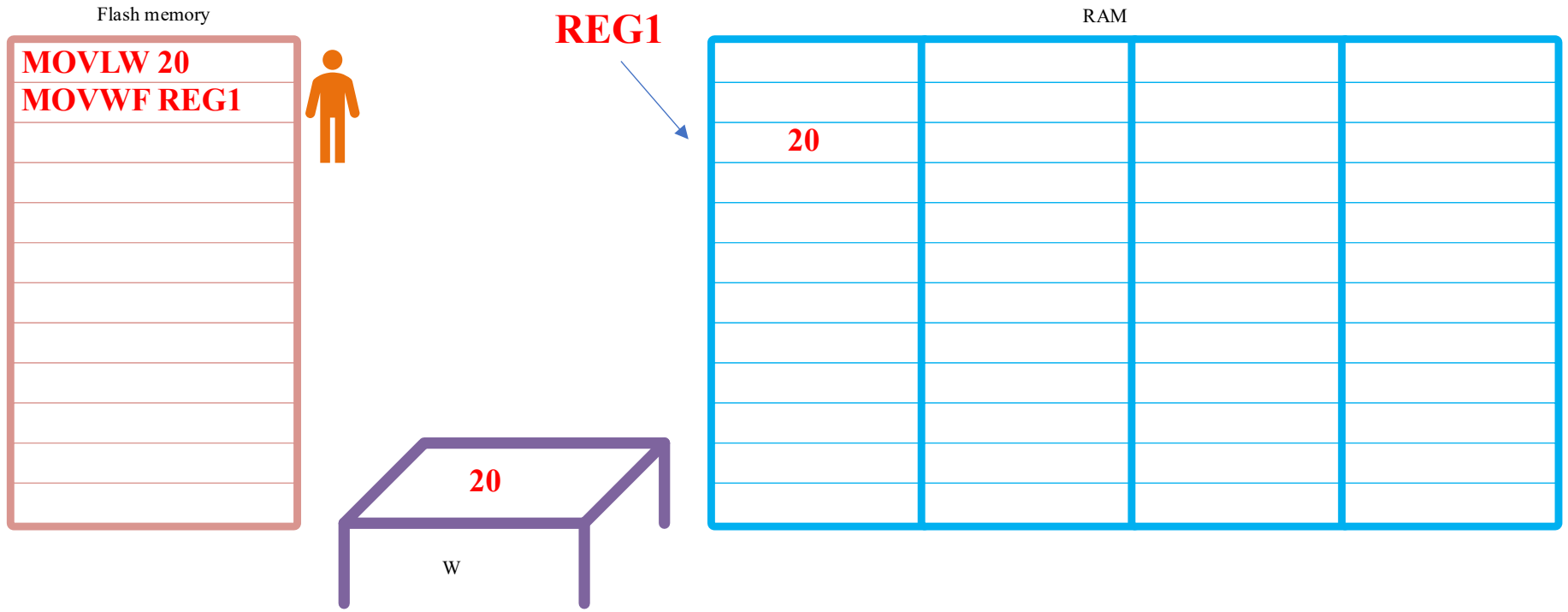
Click to play



Assembly Language (MOVWF)

MOVWF	Move W to File
Syntax	MOVWF f
Operands	f is a file located in RAM (0<f<127)
Operation	(W) to (f) i.e., the content of W is moved to the file
Example	**MOVWF REG1**

W is the register
(W) Is the content of the register

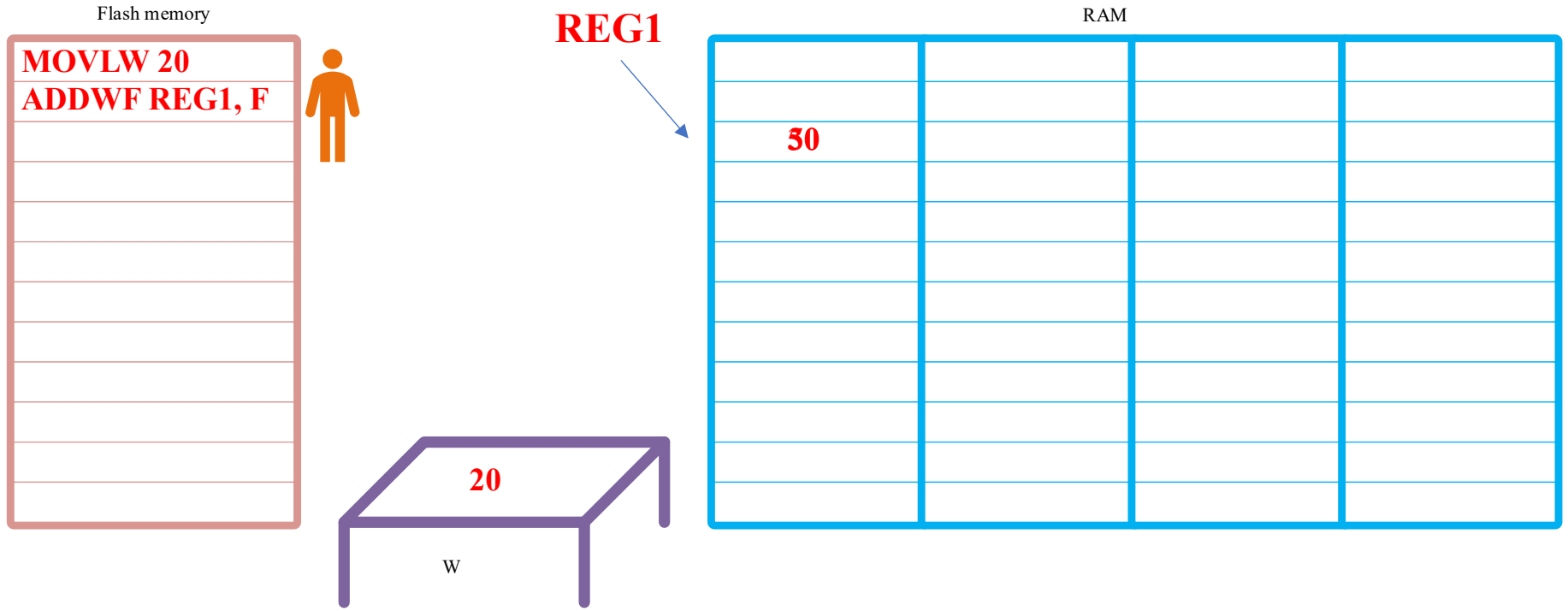


Click to play

Assembly Language (ADDWF)

ADDWF	ADD W and File
Syntax	ADDWF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(destination) = (W)+(f)
Example	**ADDWF REG1, F**

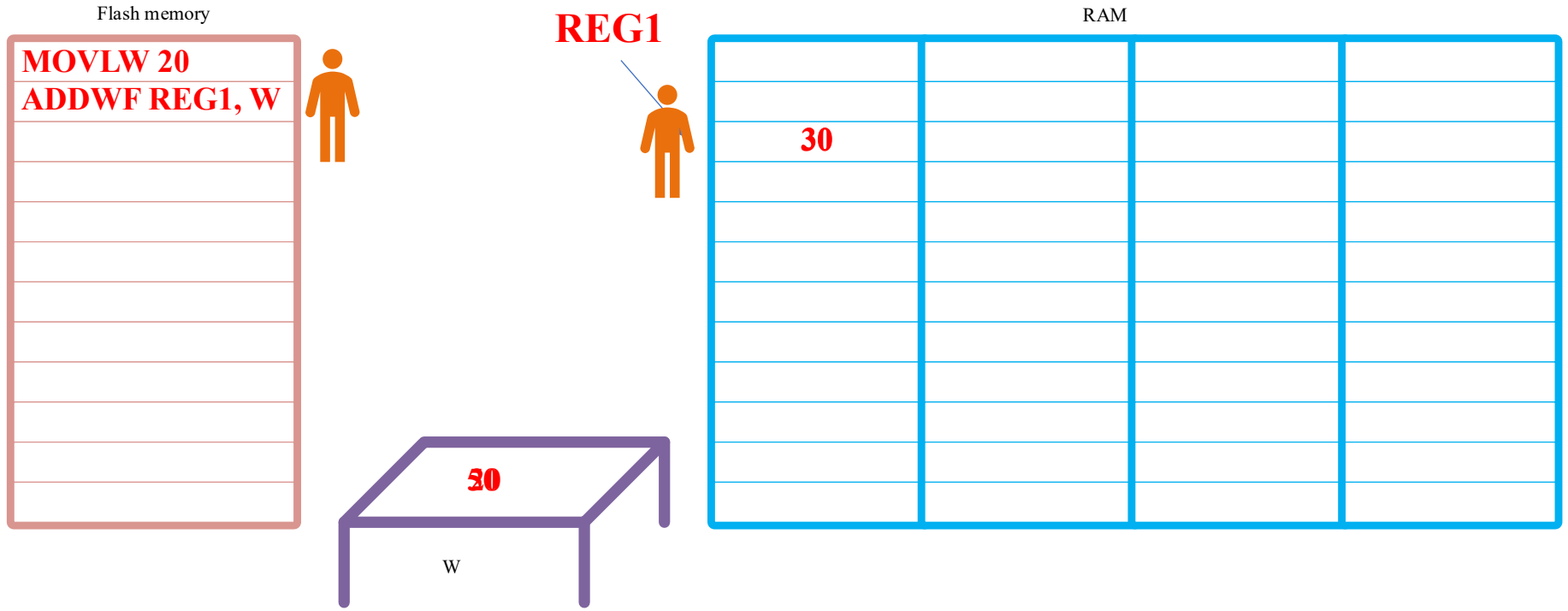
Arithmetic Operation



Assembly Language (ADDWF)

ADDWF	ADD W and File
Syntax	ADDWF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(destination) = (W)+(f)
Example	**ADDWF REG1, W**

Arithmetic Operation

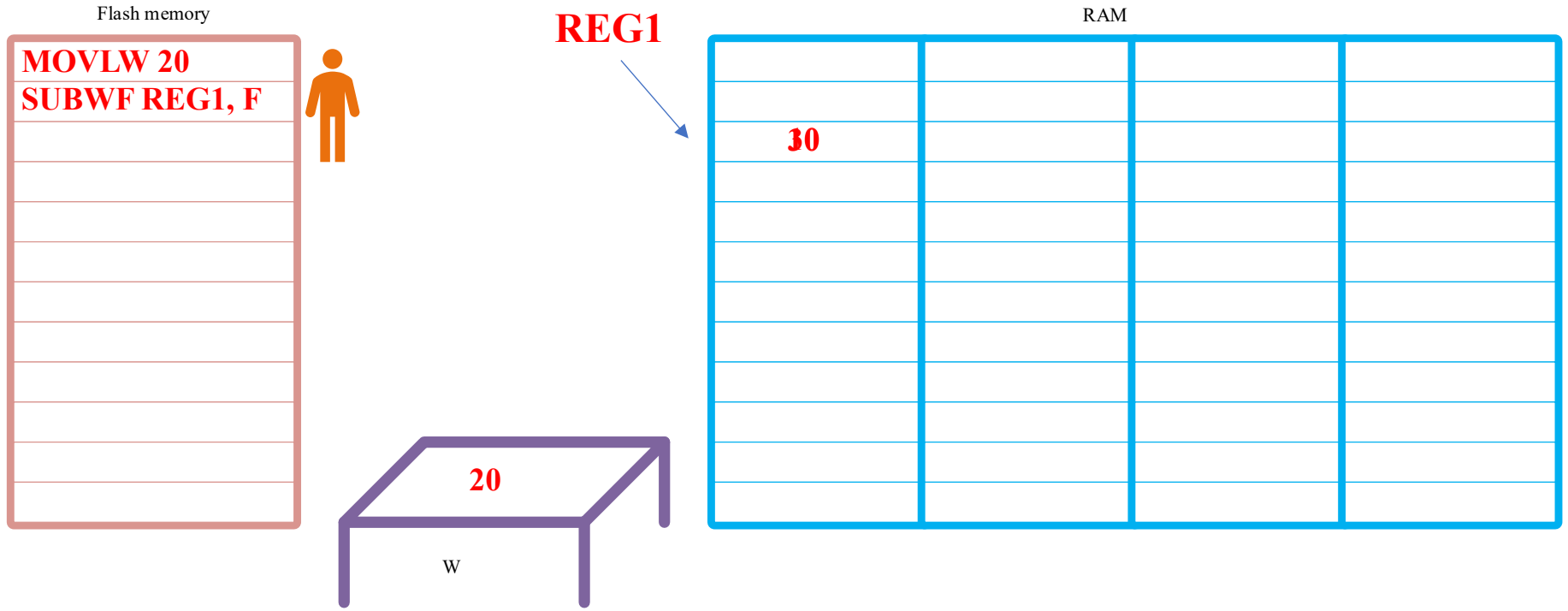


Click to play

Assembly Language (SUBWF)

SUBWF	SUB W from File
Syntax	SUBWF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(destination) = (f)-(W)
Example	**SUBWF REG1, F**

Arithmetic Operation

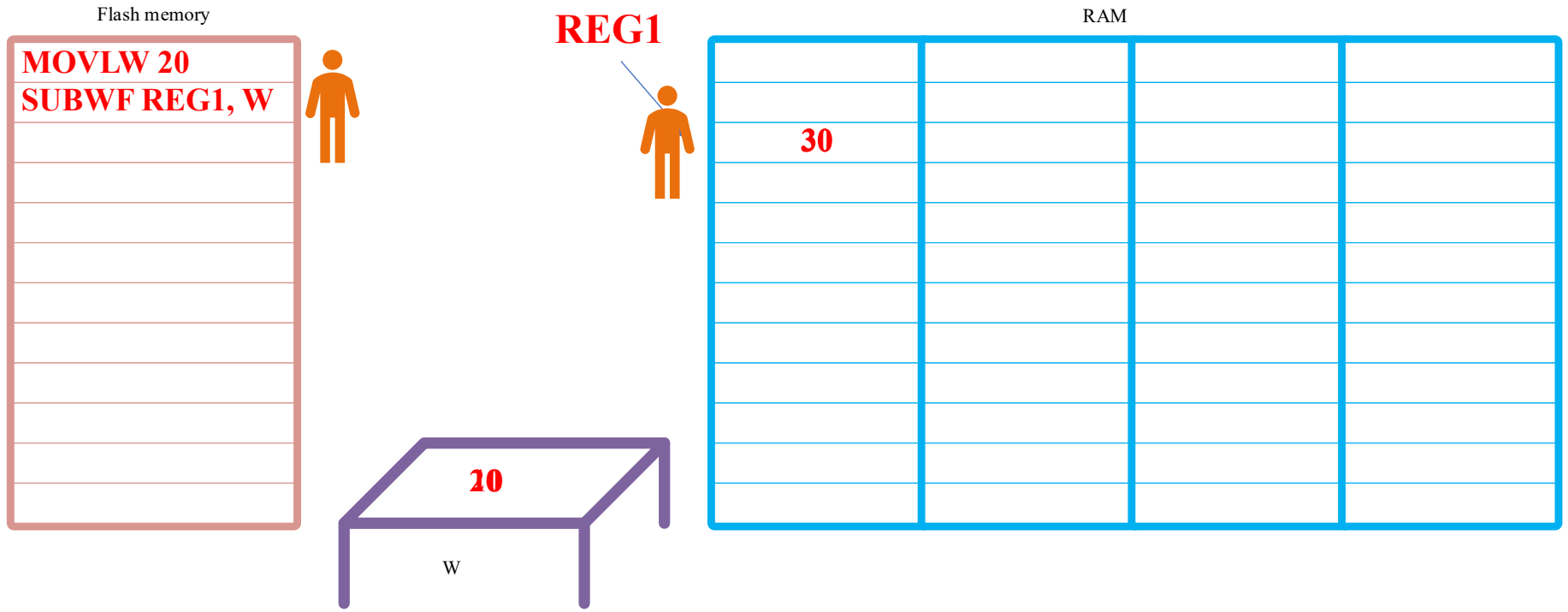


Click to play

Assembly Language (SUBWF)

SUBWF	SUB W from File
Syntax	SUBWF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(destination) = (f)-(W)
Example	**SUBWF REG1, W**

Arithmetic Operation



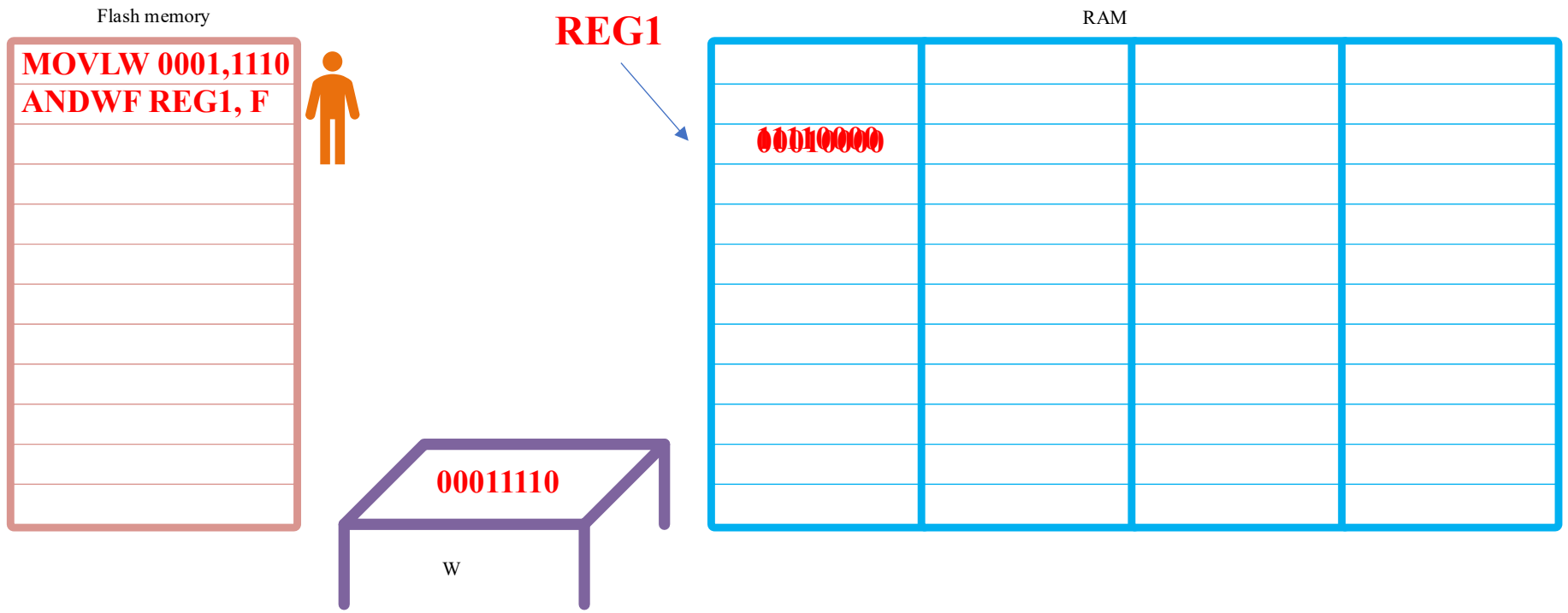
Click to play

Assembly Language (ANDWF)

ANDWF	AND W with File
Syntax	ANDWF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(destination) = (f) AND (W)
Example	**ANDWF REG1, F**

Arithmetic Operation

W	0	0	0	1	1	1	1	0
Old F	1	1	1	1	0	0	0	0
New F	0	0	0	1	0	0	0	0



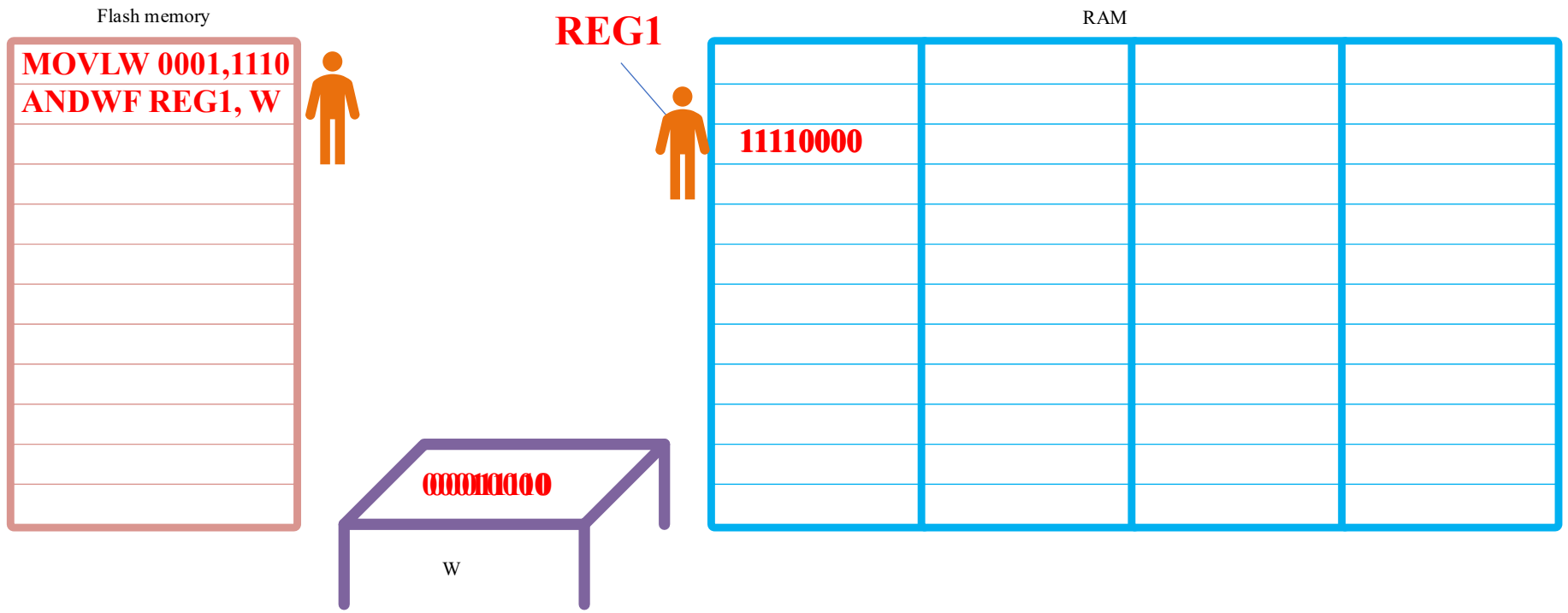
Click to play

Assembly Language (ANDWF)

ANDWF	AND W with File
Syntax	ANDWF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(destination) = (f) AND (W)
Example	**ANDWF REG1, W**

Arithmetic Operation

W	0	0	0	1	1	1	1	0
F	1	1	1	1	0	0	0	0
New W	0	0	0	1	0	0	0	0



Click to play



Assembly Language (IORWF, XORWF)

IORWF	Inclusive OR W with File
Syntax	IORWF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(destination) = (f) OR (W)
Example	**IORWF REG1, W IORWF REG1, F**

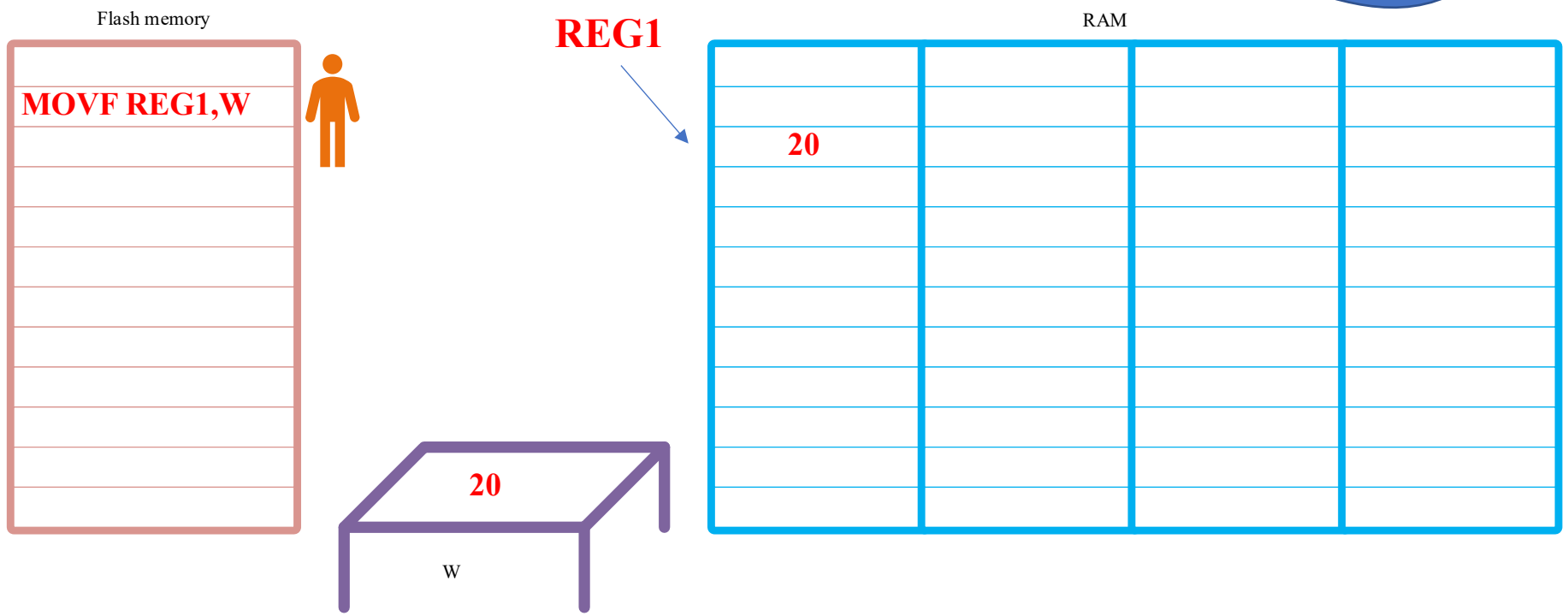
XORWF	XOR W with F
Syntax	XORWF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	destination = f XOR W
Example	**XORWF REG1, W XORWF REG1, F**

Arithmetic Operations

Assembly Language (MOVF)

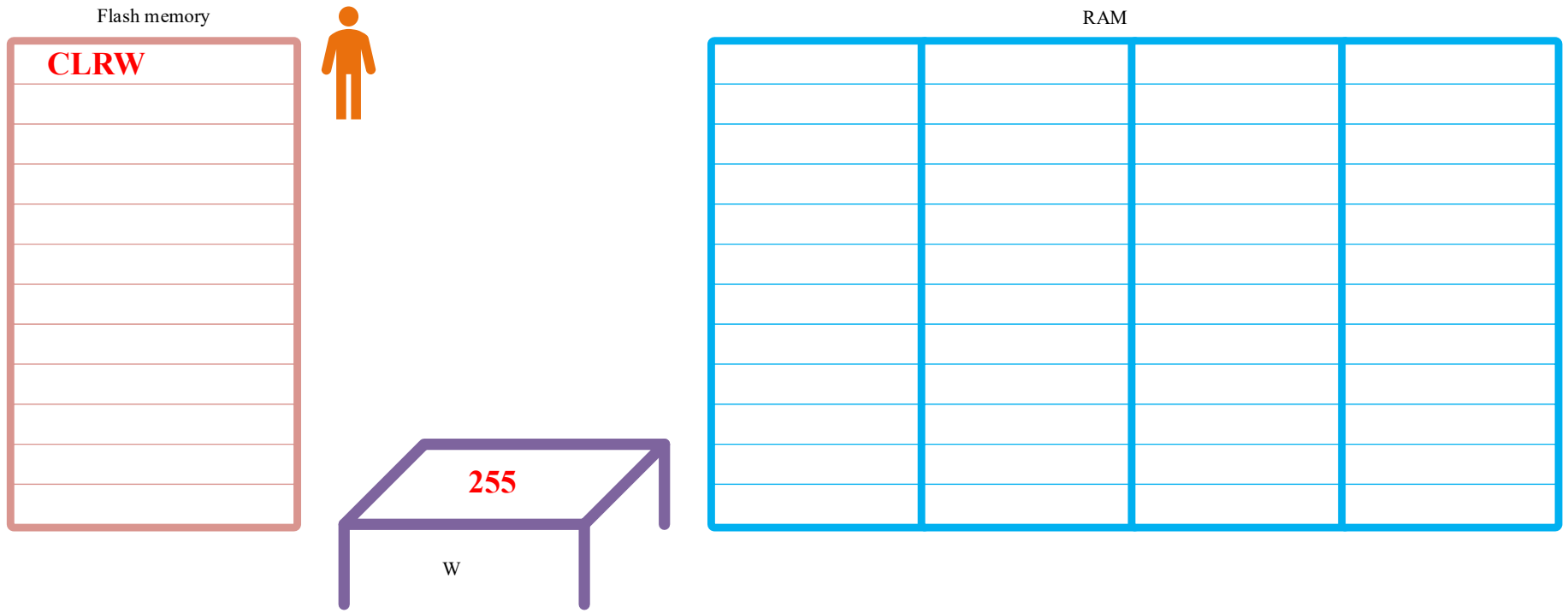
MOVF	Move File to destination
Syntax	MOVF f, d
Operands	f is a file located in RAM (0<f<127) , (d is W or F)
Operation	(f) to (destination)
Example	**MOVF REG1, W**

If d = 0,destination is W register.
If d = 1, the destination is file register 'f' itself.
d = 1 is useful to test a file register since status flag Z is affected. (To be discussed later)



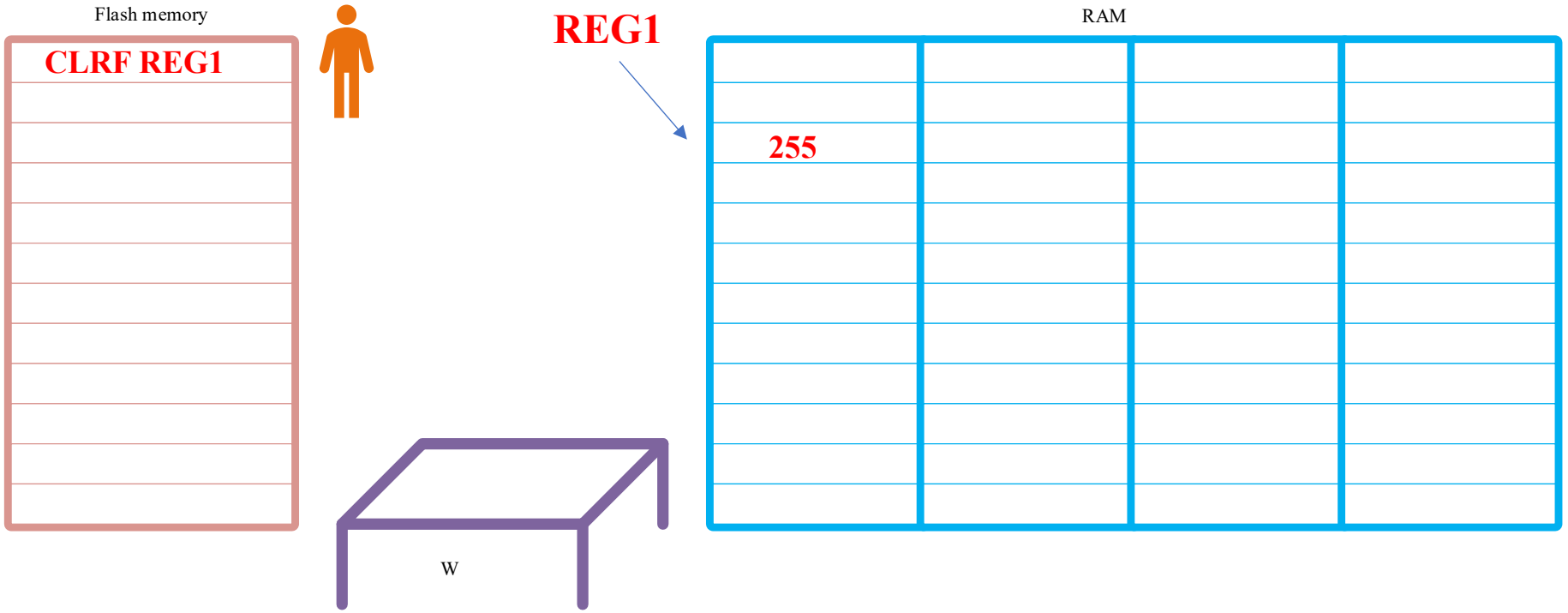
Assembly Language (CLRW)

CLRW	Clear W register
Syntax	CLRW
Operands	
Operation	(W) = 0
Example	**CLRW**



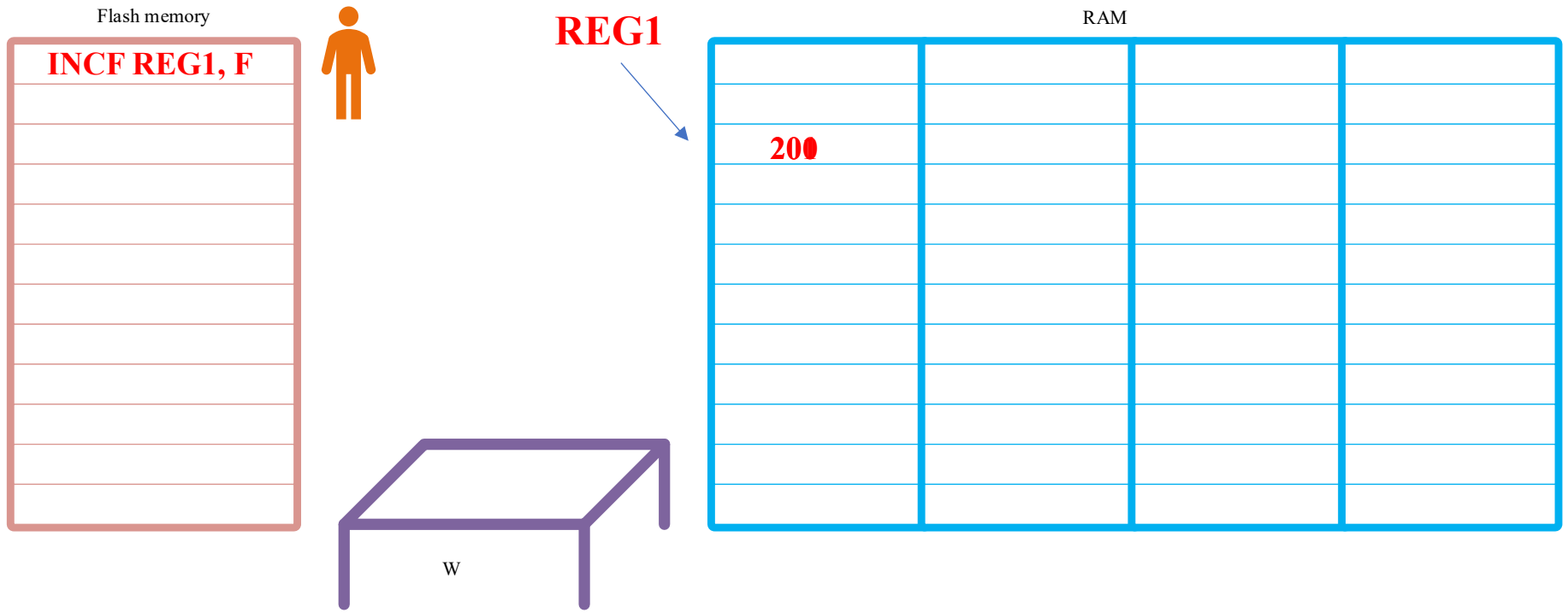
Assembly Language (CLRF)

CLRF	Clear file
Syntax	CLRF f
Operands	f is a file located in RAM (0<f<127)
Operation	(f) = 0
Example	**CLRF REG1**



Assembly Language (INCF)

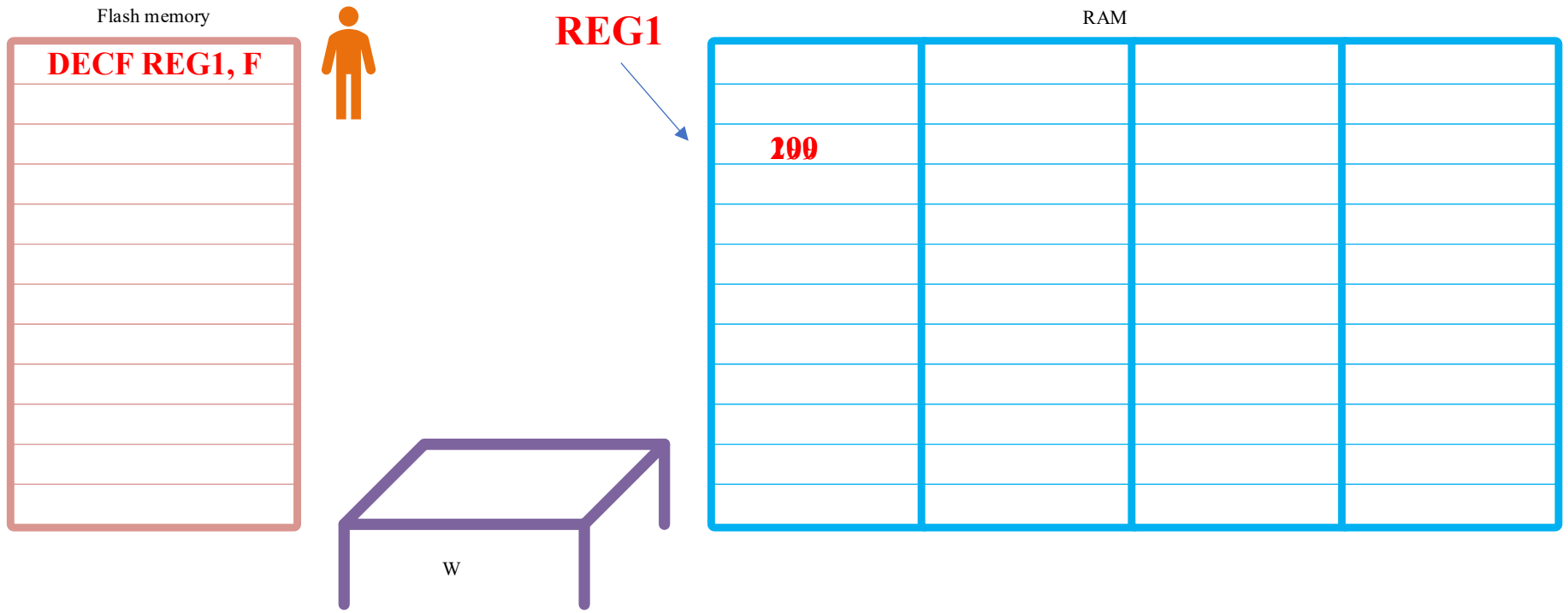
INCF	Increment file
Syntax	INCF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(d) = (f) + 1
Example	**INCF REG1, F INCF REG1, W**



Click to play

Assembly Language (DECF)

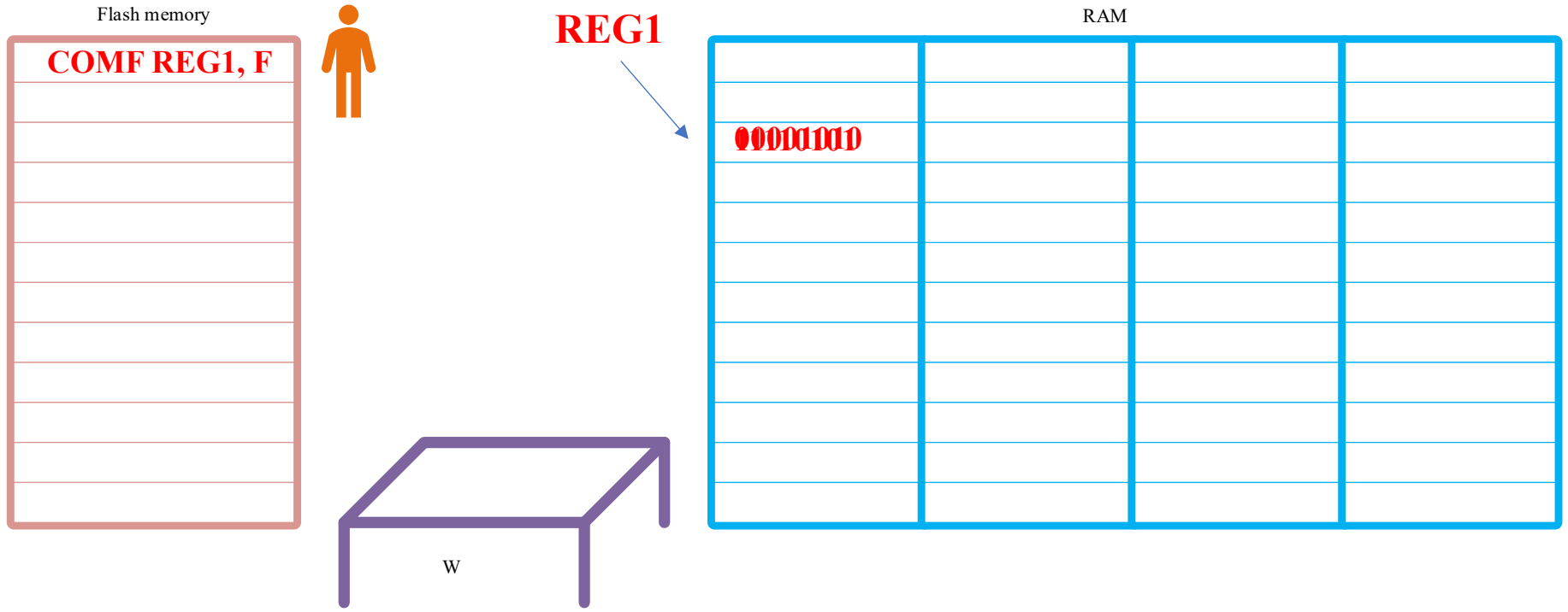
DECF	Decrement file
Syntax	DECF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(d) = (f) – 1
Example	**DECF REG1, F DECF REG1, W**



Click to play

Assembly Language (COMF)

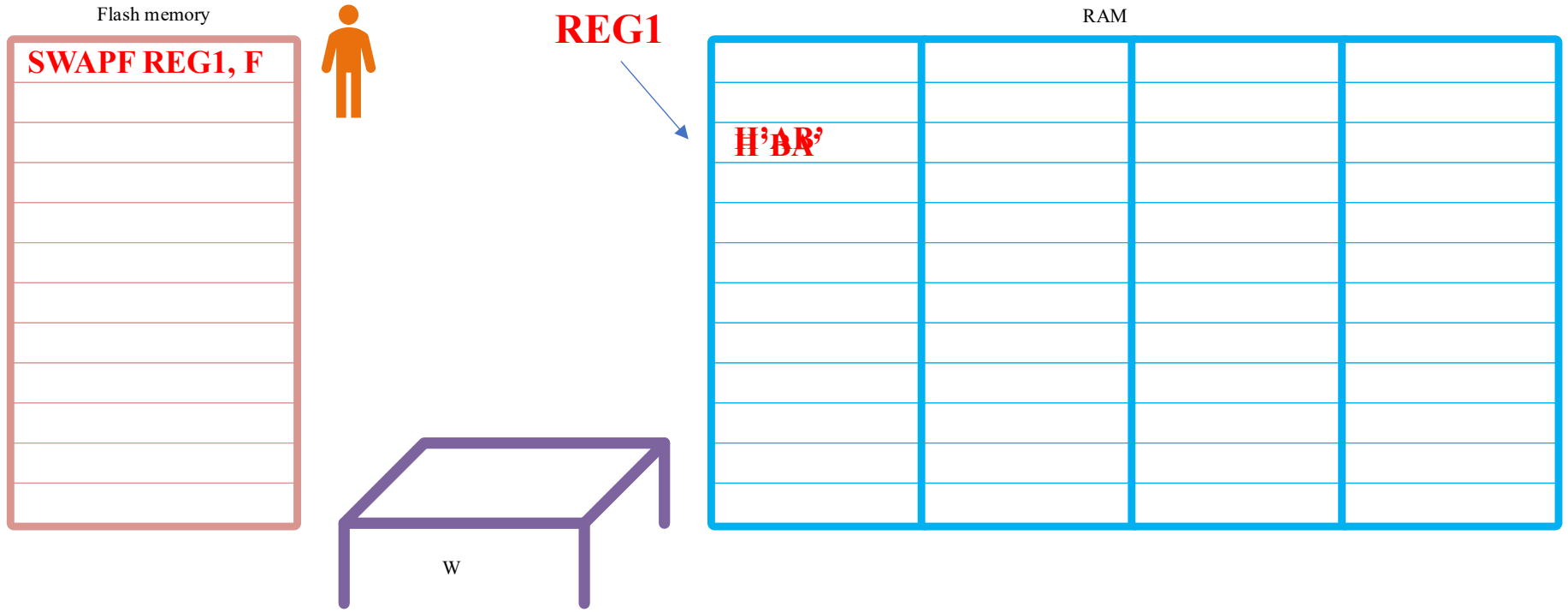
COMF	Complement file
Syntax	COMF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(d) = $(\overline{f})$ (every bit in f is flipped, i.e., 1→0 0→1)
Example	**COMF REG1, F COMF REG1, W**



Click to play

Assembly Language (SWAPF)

SWAPF	Swap nibbles in file
Syntax	SWAPF f, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(f<3:0>) → (destination<7:4>),(f<7:4>) →(destination<3:0>)
Example	**SWAPF REG1, F SWAPF REG1, W**

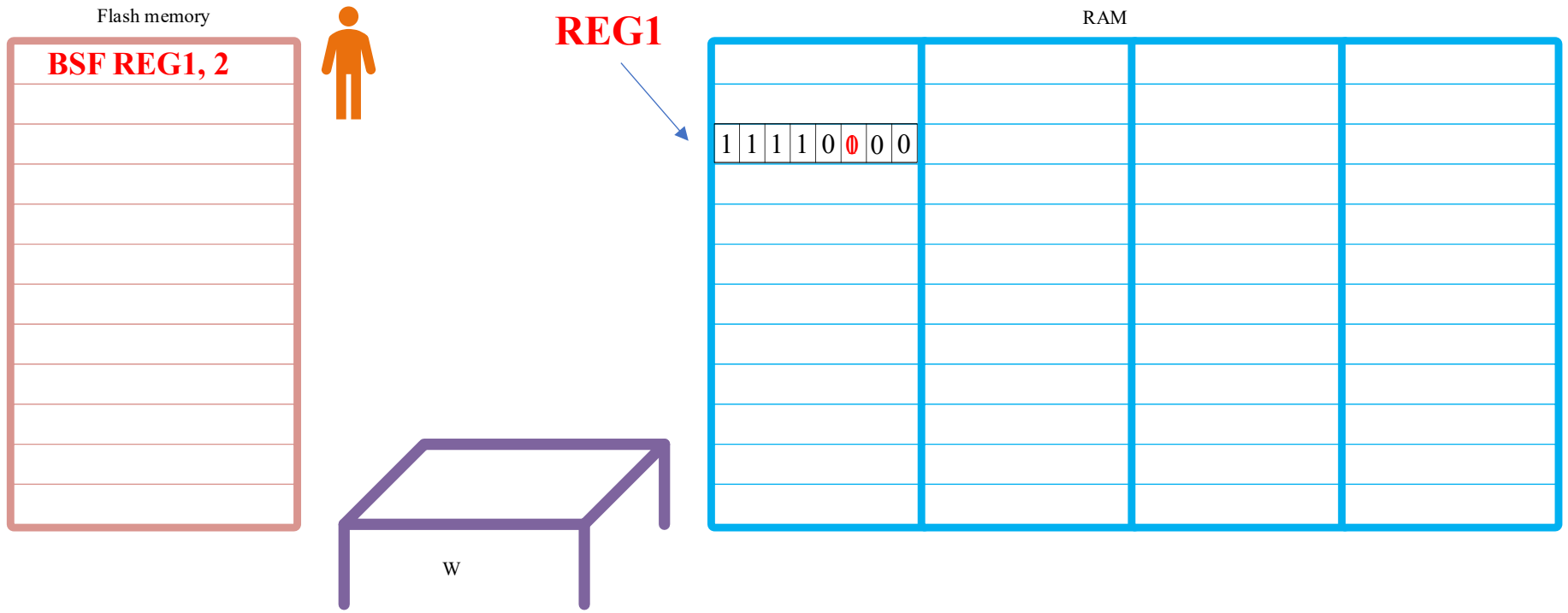


Click to play

Assembly Language (BSF)

BSF	Bit set file
Syntax	BSF file, bit#
Operands	f is a file located in RAM (0<f<127), bit# from 0 to 7
Operation	1 → (f<bit#>)
Example	**BSF REG1, 2**

b7	b6	b5	b4	b3	b2	b1	b0

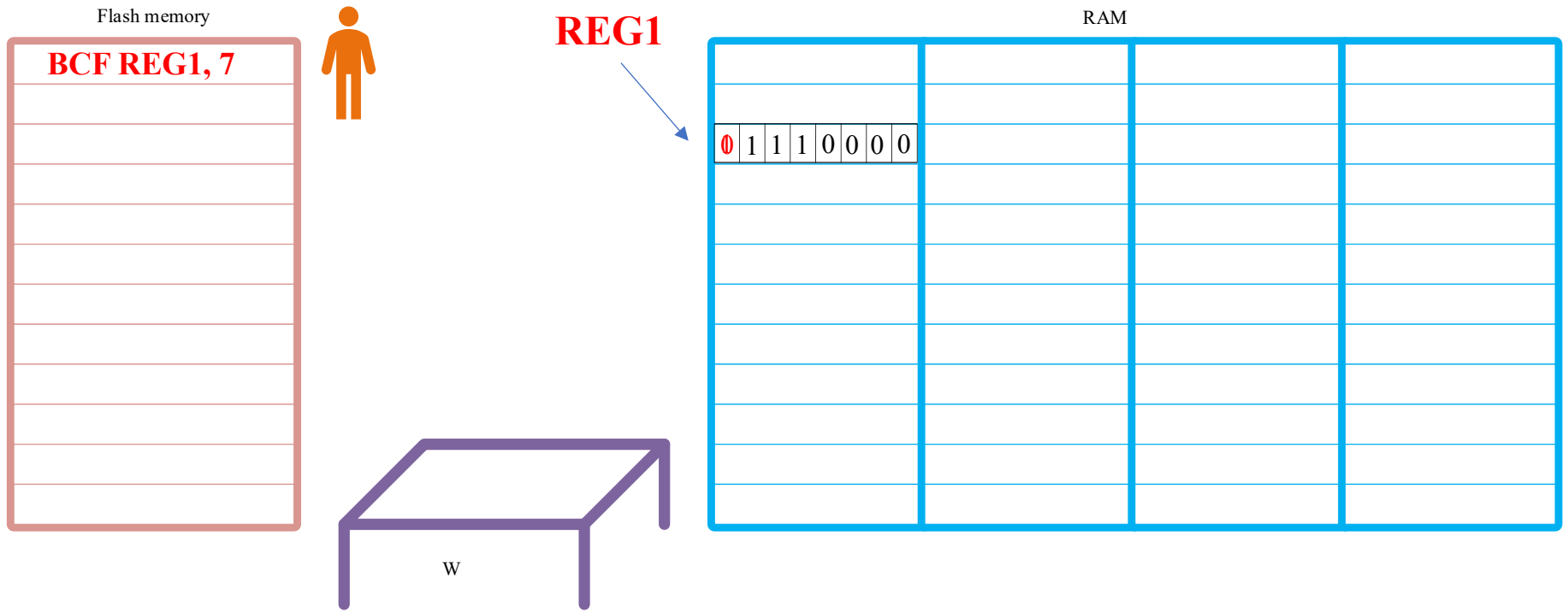


Click to play

Assembly Language (BCF)

BCF	Bit clear file
Syntax	BCF file, bit#
Operands	f is a file located in RAM (0<f<127), bit# from 0 to 7
Operation	0 → (f<bit#>)
Example	**BCF REG1, 7**

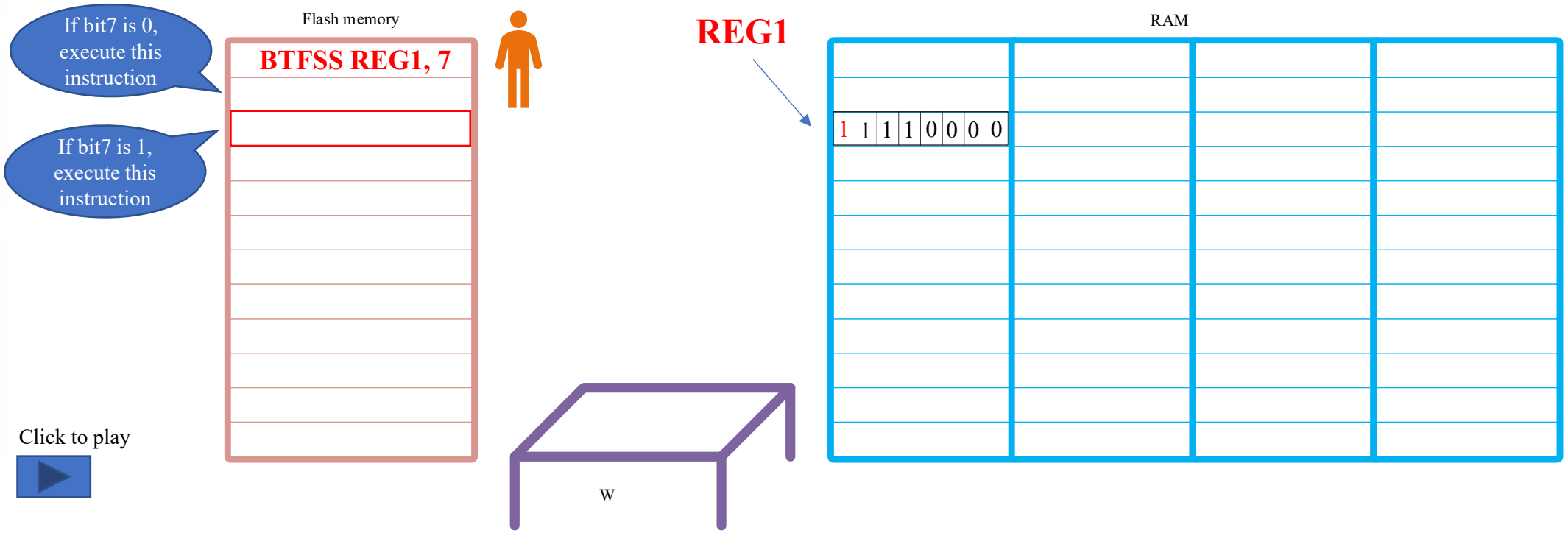
b7	b6	b5	b4	b3	b2	b1	b0



Assembly Language (BTFSS)

BTFSS	Bit test f and skip if set
Syntax	BTFSS file, bit#
Operands	f is a file located in RAM (0<f<127), bit# from 0 to 7
Operation	Skip one line if (f<b>) = 1
Example	**BTFSS REG1, 7**

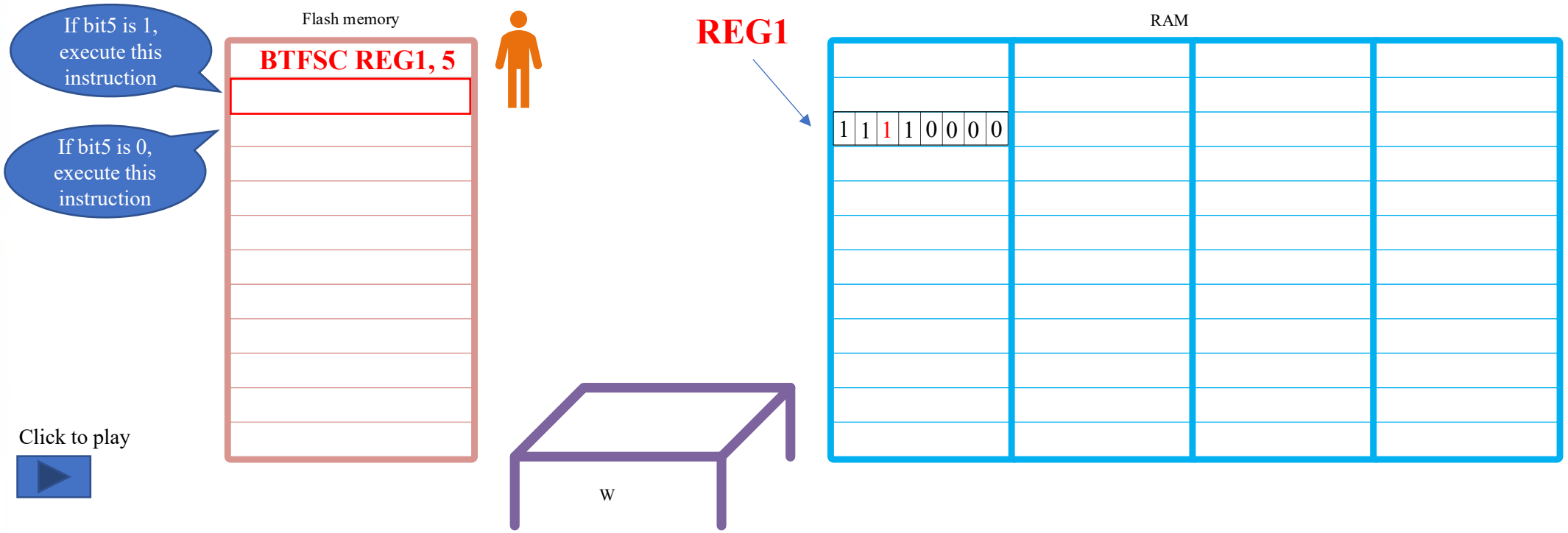
b7	b6	b5	b4	b3	b2	b1	b0



Assembly Language (BTFSC)

BTFSC	Bit test f and skip if clear
Syntax	BTFSC file, bit#
Operands	f is a file located in RAM (0<f<127), bit# from 0 to 7
Operation	Skip one line if (f<b>) = 0
Example	**BTFSC REG1, 5**

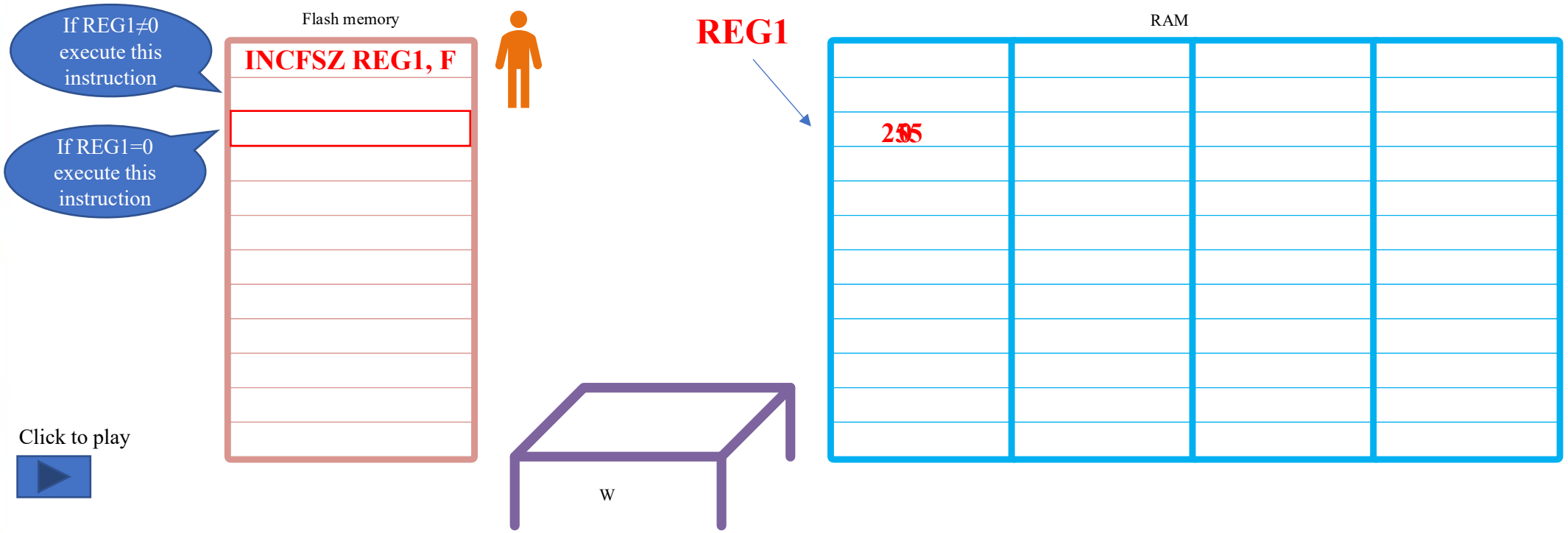
b7	b6	b5	b4	b3	b2	b1	b0



Assembly Language (INCFSZ)

INCFSZ	Increment file and skip is zero
Syntax	INCFSZ file, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(f) + 1 → (destination), skip if result = 0
Example	**INCFSZ REG1, F**

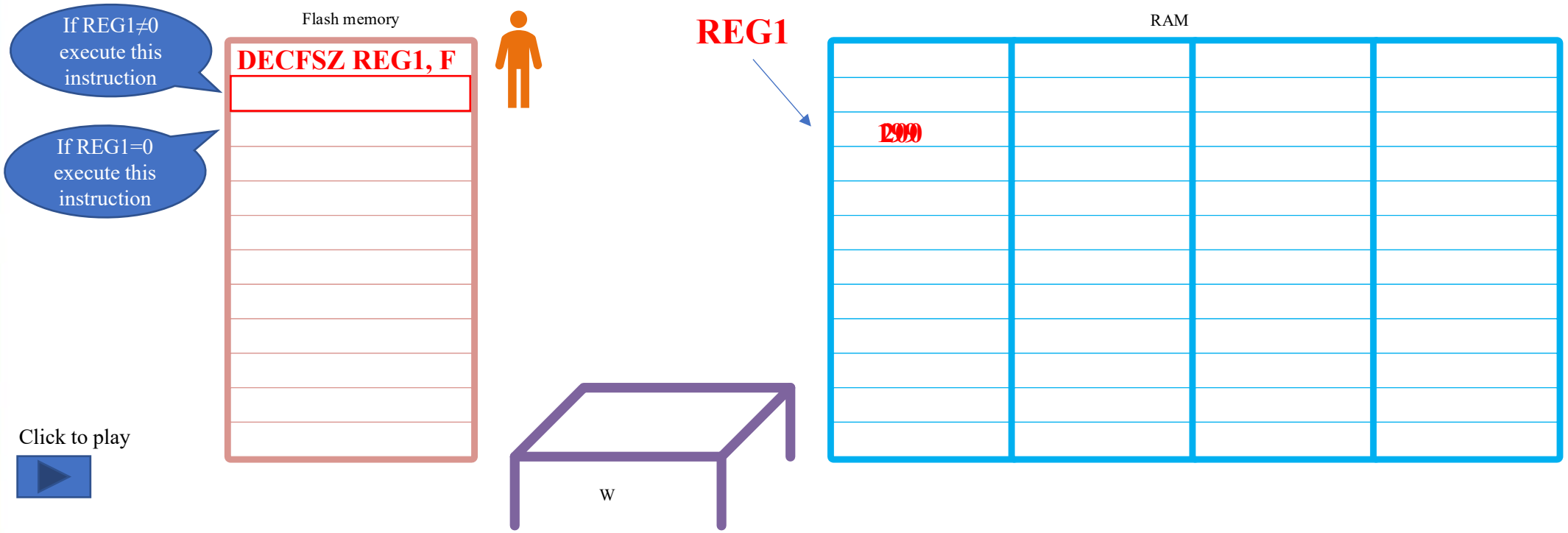
In a byte, the biggest value is 255
Thus 255 + 1 = 256 = 0



Assembly Language (DECFSZ)

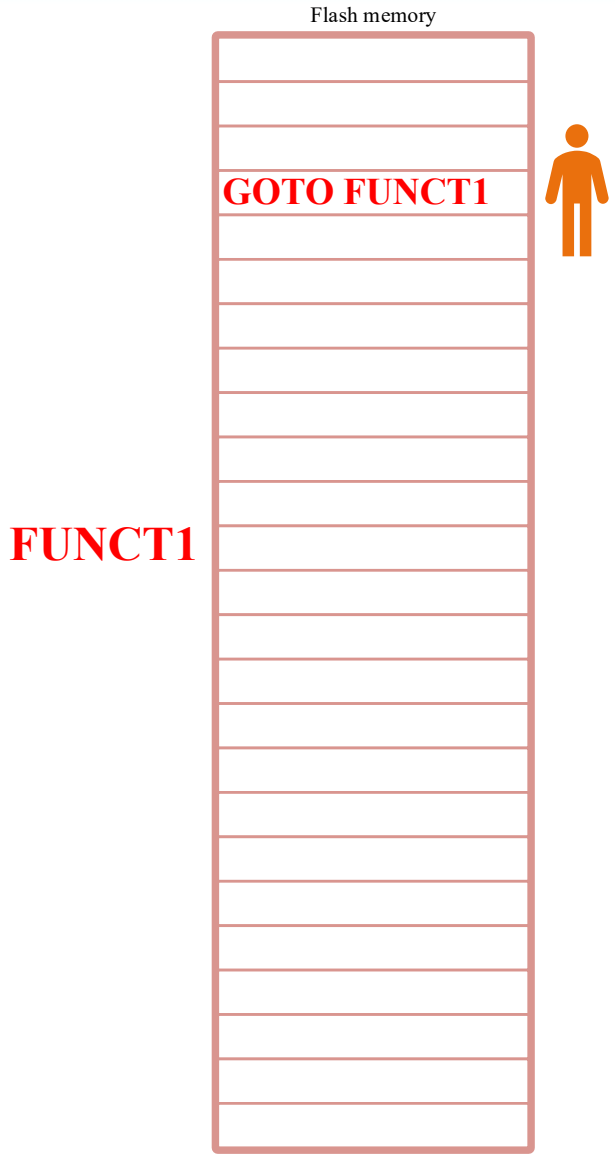
DECFSZ	Decrement file and skip is zero
Syntax	DECFSZ file, d
Operands	f is a file located in RAM (0<f<127), (d is W or F)
Operation	(f) - 1 → (destination), skip if result = 0
Example	**DECFSZ REG1, F**

In a byte, the biggest value is 255
Thus 255 + 1 = 256 = 0



Assembly Language (GOTO)

GOTO	Unconditional branch
Syntax	GOTO k
Operands	k is a Flash address (0<k<2047)
Operation	k → PC<10:0>
Example	**GOTO SUBROUTINE1**



Click to play

Assembly Language (CALL)

CALL	Call Subroutine
Syntax	CALL k
Operands	k is a Flash address (0<f<2047)
Operation	k → PC<10:0>
Example	**CALL SUBROUTINE1**

Stack memory is the place in which the address of the Flash memory as saved.
8 levels of stack are available

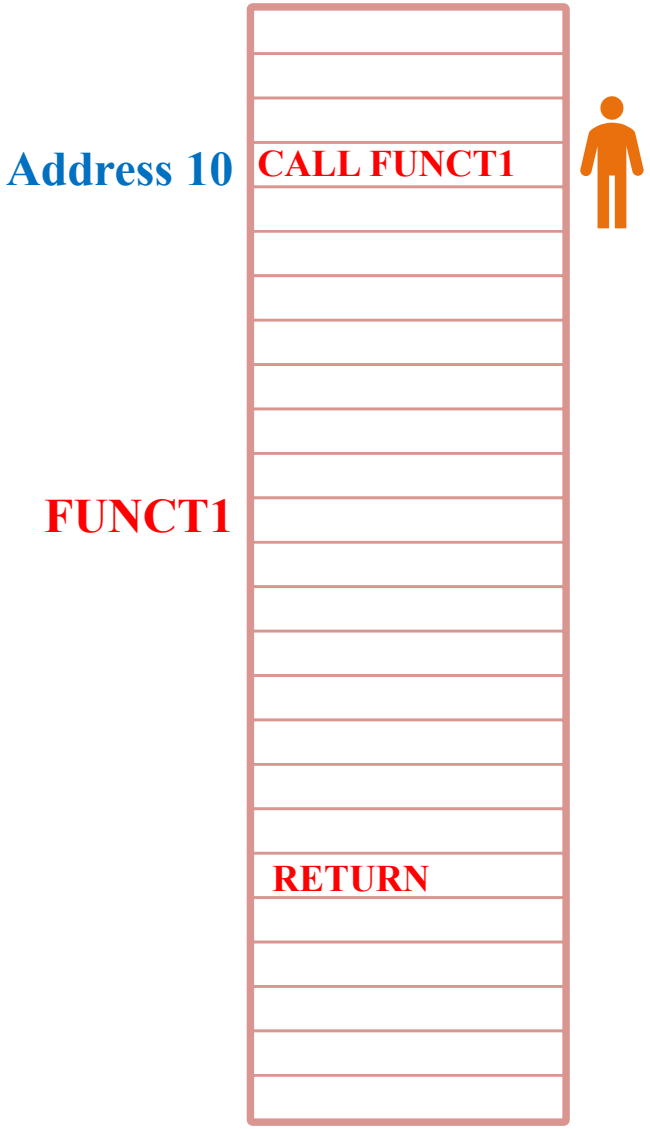
Stack memory



Click to play



Flash memory



NB: a CALL function Pushes an address+1 onto the stack memory

NB: a Call function requires a RETURN function to pop up the flash address.
The Stack memory follows the sequence of Last-in, first-out

Assembly Language (CALL)

CALL	Call Subroutine
Syntax	CALL k
Operands	k is a Flash address (0<f<2047)
Operation	k → PC<10:0>
Example	**CALL SUBROUTINE1**

Stack memory is the place in which the address of the Flash memory as saved.
8 levels of stack are available

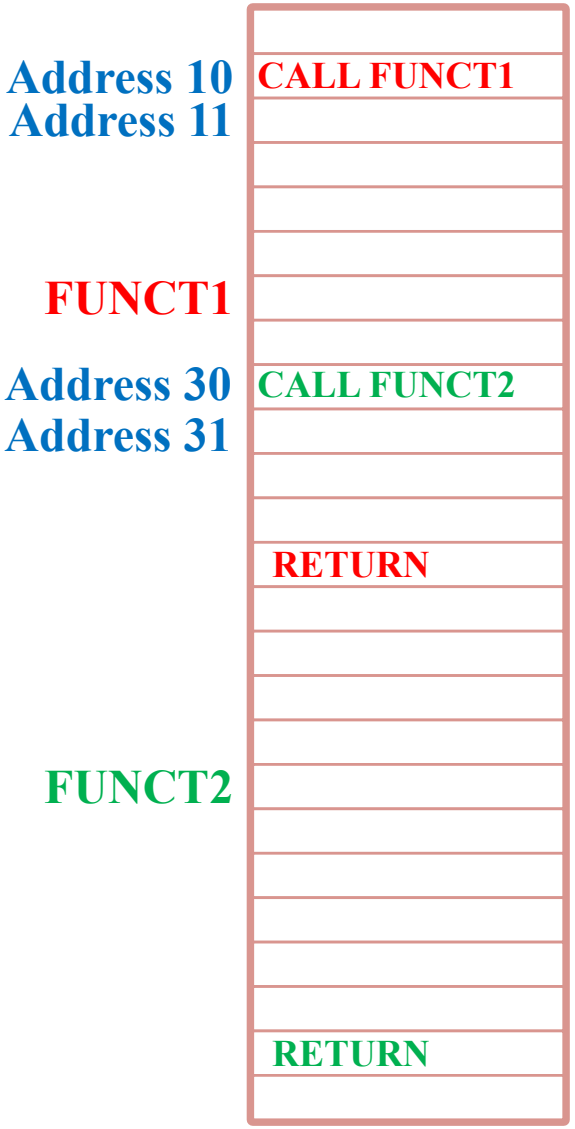
Stack memory



Click to play



Flash memory



NB: a CALL function Pushes the address+1 onto the stack memory

This is a "CALL function" inside a function.
MAX 8 nested CALL are allowed

NB: a Call function requires a RETURN function to pop up the flash address.

Example 1: Adding two numbers

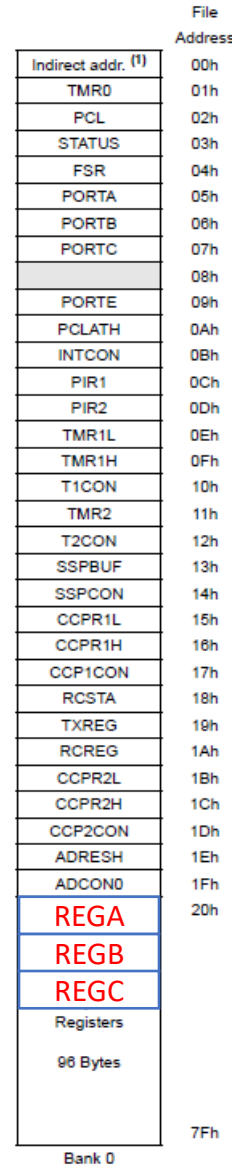
- Assume we want to add two numbers and save the result in a register.
- REGC = REGA + REGB, where REGA = 25 and REGB = 30

To do that, first we have to allocate 3 memory locations in RAM for the 3 registers.

This can be done in assembly as

```
REGA   EQU 0X020
REGB   EQU 0X021
REGC   EQU 0x022
```

3 memory locations are used at addresses 0x20, 0x21 and 0x22



Example 1: Adding two numbers

- The next step consists of loading the values in REGA and REGB. (REGA = 25 and REGB = 30)

```
MOVLW   25      ; move literal 25 to W
MOVWF   REGA    ; move the content of W to REGA

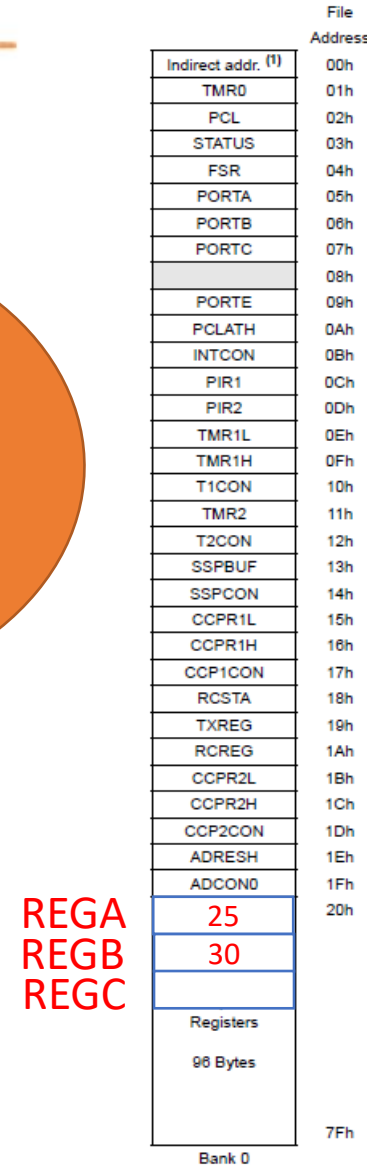
MOVLW   30      ; move literal 30 to W
MOVWF   REGB    ; move the content of W to REGA
```

Check the instructions
MOVLW
MOVWF
From the previous slides

NB: W register is a temporary place to put values…

There is no way to move literal from Flash to RAM directly

i.e., there is no instruction called MOVLF 25, REGA in PIC16



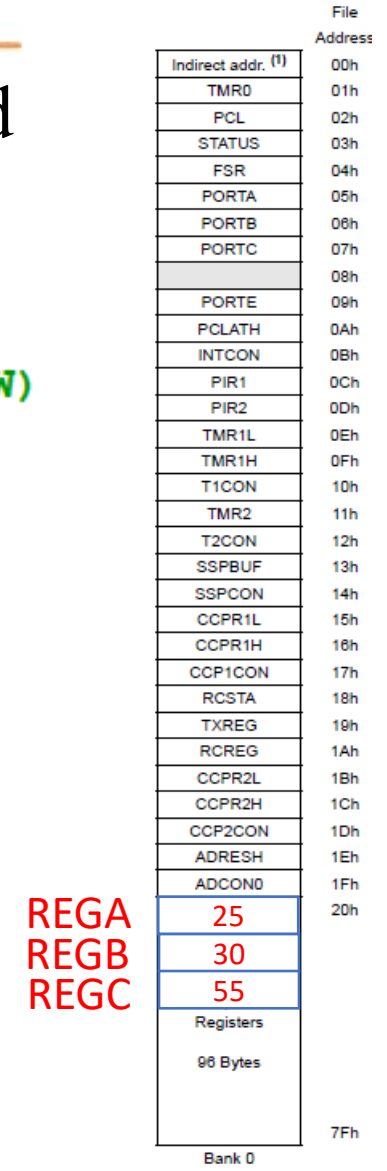
Example 1: Adding two numbers

- The next step consists of making the summation of REGA and REGB, then save the result in REGC

```
MOVF    REGA, W ; move the content of REGA to W
ADDWF   REGB, W ; ADD it with the content of REGB (the result is in W)
MOVWF   REGC    ; then move the W content to REGC
```

NB: W register is a temporary place to put values…

Remember that the calculator is placed on the table, thus the addition operation should be done on W

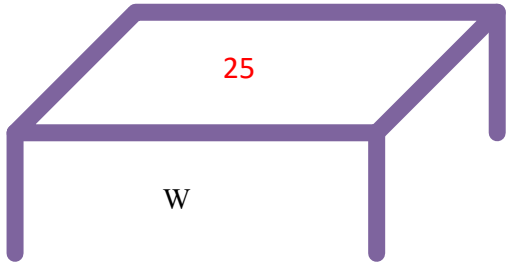


Example 1: Adding two numbers

Flash memory

```
MOVLW   25
MOVWF   REGA
MOVLW   30
MOVWF   REGB
MOVF    REGA, W
ADDWF   REGB, W
MOVWF   REGC
```





RAM

REGA	25			
REGB	30			
REGC	55			

Click to play





Basic Elements of Assembly Language

- From the previous example, the code includes lot of elements other than the instructions. The Basic elements of assembly language are:
  - Labels
  - Mnemonic
  - Operands
  - Directives
  - Comments

Basic Elements of Assembly Language

Labels

- A Label is a textual designation (generally an easy-to-read word) for a line in a program, or section of a program where the micro can jump to - or even the beginning of set of lines of a program.
- It can also be used to execute program branching (such as Goto .....)
- In the new Assembler is pic-as, the labels should be terminated by ":"

```
MAIN:
        CALL    SETUP
REP:
        MOVLW   11111111B
        MOVWF   PORTA
        CALL    DELAY_1S
        CLRF    PORTA
        CALL    DELAY_1S
        GOTO    REP

SETUP:
        CLRF    PORTA
        CLRF    PORTB
        CLRF    PORTC
        CLRF    PORTD
        CLRF    PORTE
```



Basic Elements of Assembly Language

Mnemonic

Mnemonic is the command by itself. The assembly language of the PIC midrange family is made of 35 mnemonics.

```
MAIN:
        CALL    SETUP
REP:
        MOVLW   11111111B
        MOVWF   PORTA
        CALL    DELAY_1S
        CLRF    PORTA
        CALL    DELAY_1S
        GOTO    REP

SETUP:
        CLRF    PORTA
        CLRF    PORTB
        CLRF    PORTC
        CLRF    PORTD
        CLRF    PORTE
```



Basic Elements of Assembly Language

Operands

Operands are the instruction elements for the instruction is being executed. They are usually

1. Registers (PORTA, PORTB…)
2. Variables (any defined register in GFR)
3. Constant, or number represented in:
   - Decimal (Digits 0 to 9 followed by D, d or nothing, Example 23D)
   - Hexadecimal (Digits 0 to 9, A to F preceded by 0x or followed by H or h
   - Binary (Digits 0 and 1 followed by B. 01010101B
   - Ascii (any ascii letter between ‘ ‘, Example ‘A’

```
MAIN:
     CALL    SETUP
REP:
     MOVLW   11111111B
     MOVWF   PORTA
     CALL    DELAY_1S
     CLRF    PORTA
     CALL    DELAY_1S
     GOTO    REP

SETUP:
     CLRF    PORTA
     CLRF    PORTB
     CLRF    PORTC
     CLRF    PORTD
     CLRF    PORTE
```



Basic Elements of Assembly Language

Directives

A directive is similar to an instruction, but unlike an instruction it is independent on the microcontroller model, and represents a characteristic of the assembly language itself.

Directives are usually given purposeful meanings via variables or registers.

```
; CONFIG2
  CONFIG   BOR4V = BOR40V        ; Brown-out Reset Se
  CONFIG   WRT = OFF             ; Flash Program Memo

REGA   EQU 0X020
REGB   EQU 0X021
REGC   EQU 0x022

PSECT BLABLA,CLASS=CODE, DELTA=2,ABS

   ORG     0          ; upload the below code from res

   MOVLW   'A'        ; move literal 25 to W
   MOVWF   REGA       ; move the content of W to REGA

   MOVLW   30         ; move literal 30 to W
   MOVWF   REGB       ; move the content of W to REGA
```



Basic Elements of Assembly Language

Comments

Comment is a series of words that a programmer writes to make the program more clear and legible.

It is placed after an instruction, and must start with a semicolon ";".

```
; CONFIG2
  CONFIG   BOR4V = BOR40V        ; Brown-out Reset Se
  CONFIG   WRT = OFF             ; Flash Program Memo

REGA   EQU 0X020
REGB   EQU 0X021
REGC   EQU 0x022

PSECT BLABLA,CLASS=CODE, DELTA=2,ABS

    ORG      0          ; upload the below code from res

    MOVLW    'A'        ; move literal 25 to W
    MOVWF    REGA       ; move the content of W to REGA

    MOVLW    30         ; move literal 30 to W
    MOVWF    REGB       ; move the content of W to REGA
```

Lecture



Test your knowledge

1. What is the total number of assembly instruction used to program the PIC16F887
2. In the instruction MOVLW, what is L ?
3. MOVLW instruction is actually used to move or copy the L to W ?
4. What is the result of swapping H'AB'
5. What is the result of swapping B'11010001'
6. What is the result of swapping D'127'
7. What is the result of COMF D'127'
8. Reg1 = 128, what is the result of Reg1 after "DECF REG1, W"
9. Reg1 = 128, what is the result of W after "INCFSZ Reg1, F"



Test your knowledge

10. What is the size of the stack memory of the PIC16F887
11. What is the machine code of the instruction BTFSS 0x3A, 5 in 14-bit binary representation?
12. Having W = B'11110000'. What is the result of IORLW B'00001111'.
13. You are given the following PIC16F887 assembly instructions SWAP Reg1, 1. If Reg1 = 0xF3, what is the right answer when this function is executed?
14. Given an assembly language instruction COMF REG1,0 and the current values of REG = 0x13 and W = 0xC2. What will be the values of the register after the instruction?
15. The Stack memory follows the sequence of_____.



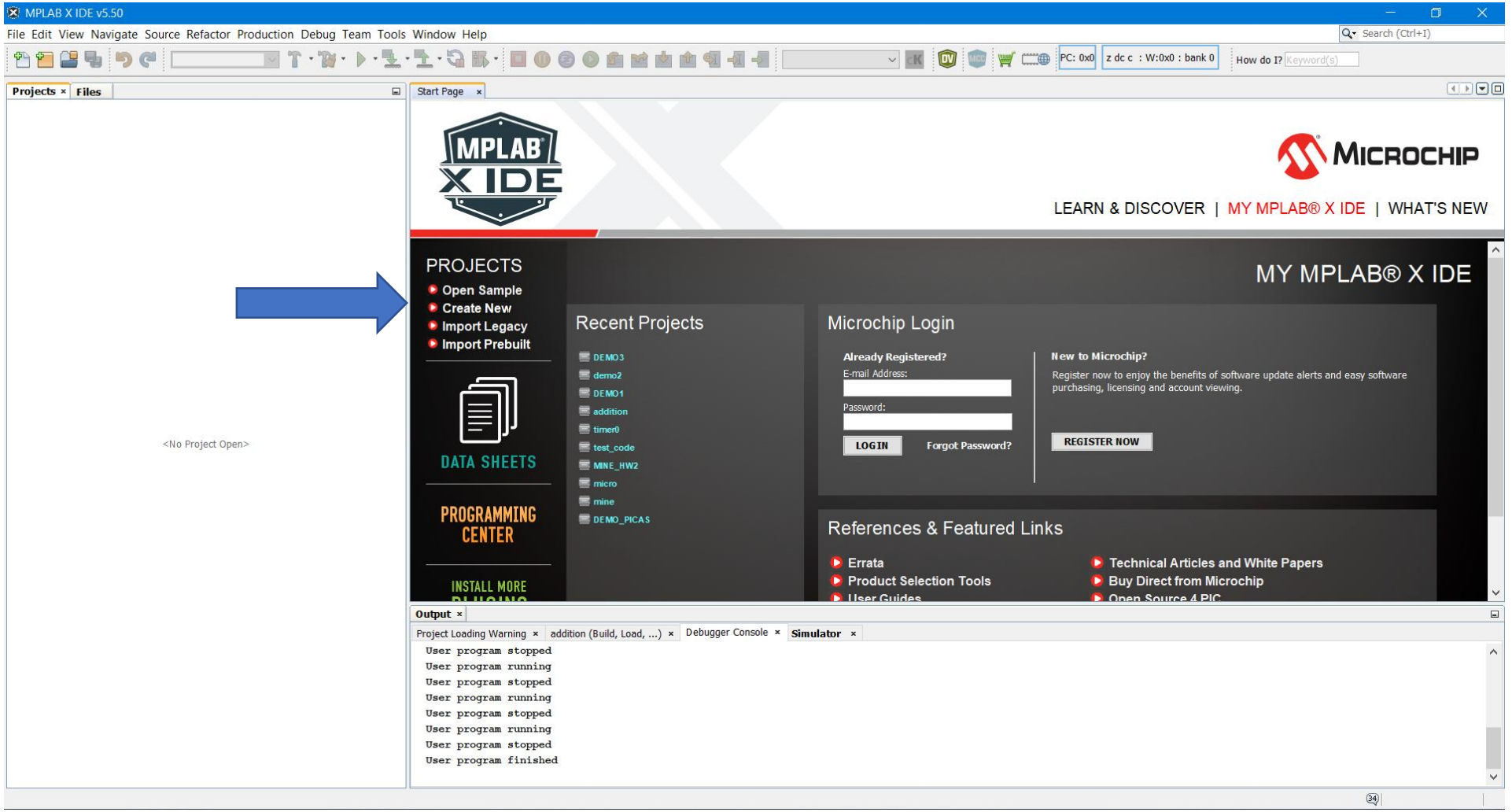
Week 2

Lab 1 / LO1

MPLABX and Assembler Code debugging

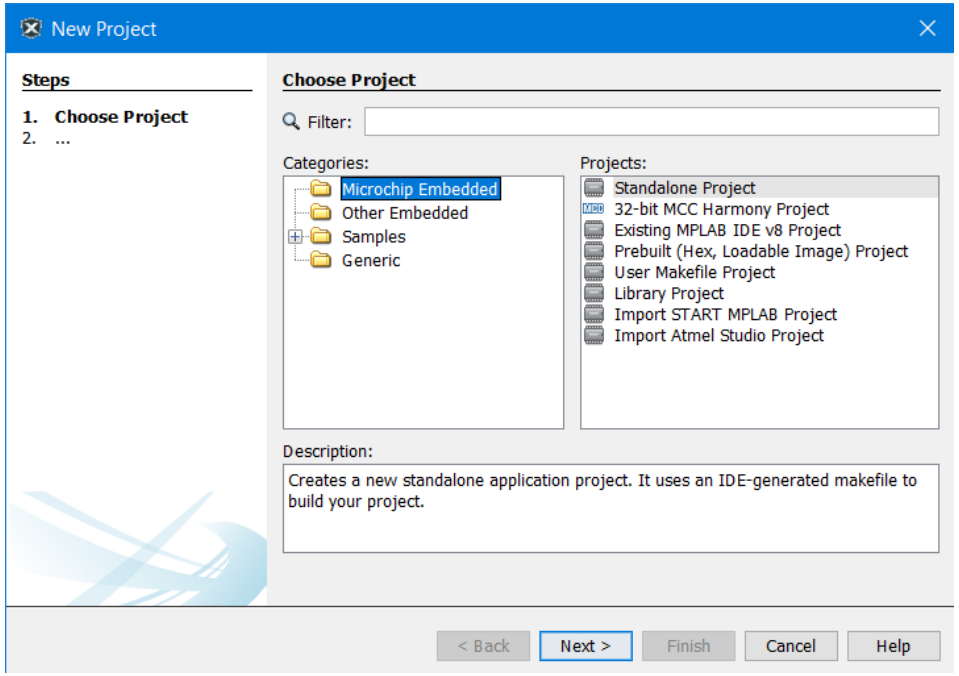
Starting with MPLABX (Activity 1)

Step 1: Create new project

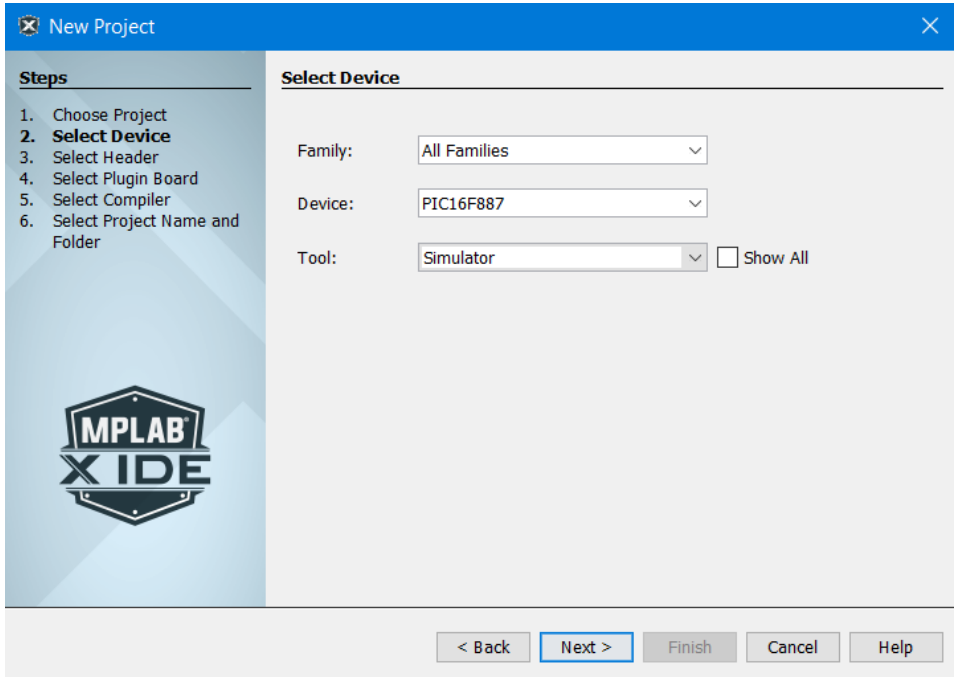


Starting with MPLABX (Activity 1)

Step 2: Select standalone Project



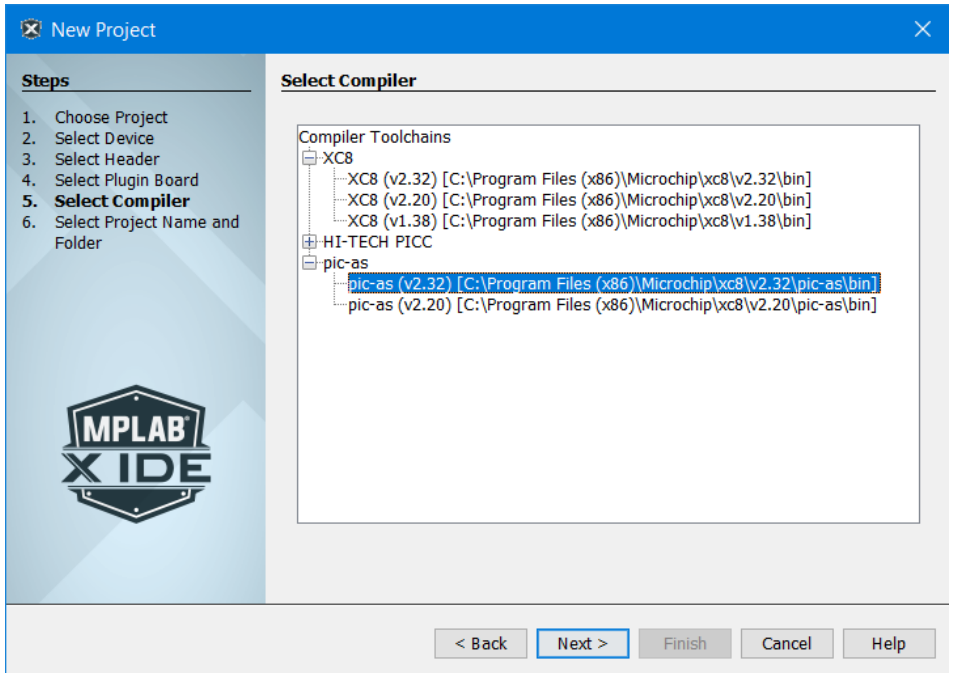
Step 3: Select PIC16887 and select the simulator tool



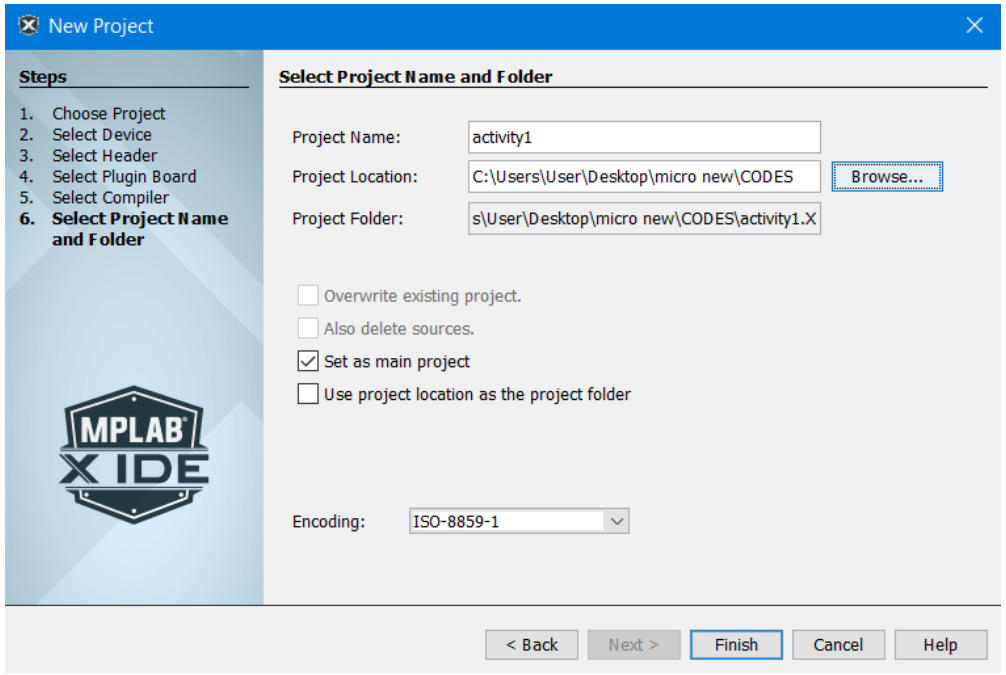
Lab

Starting with MPLABX (Activity 1)

Step 4: Select the pic-as compiler

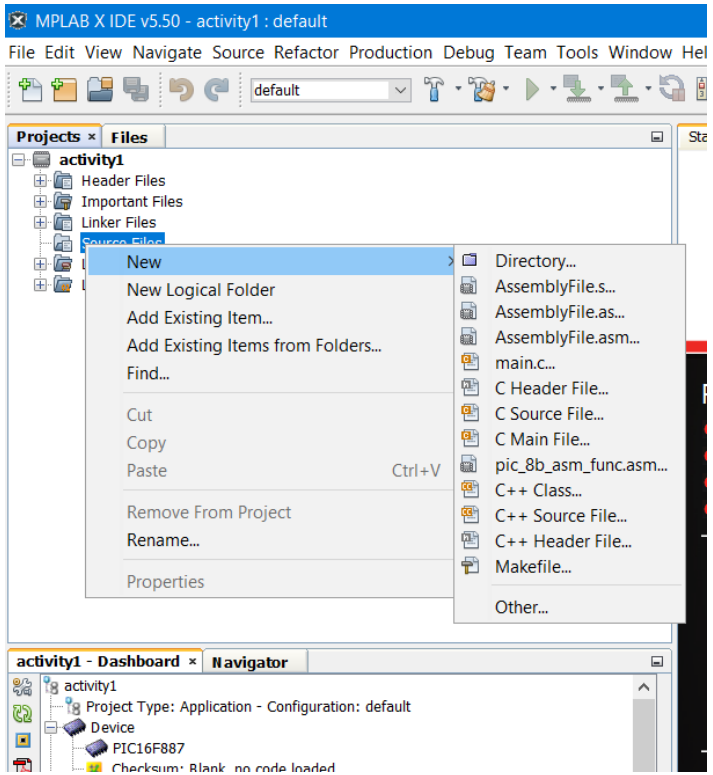


Step 5: Name the project as activity1

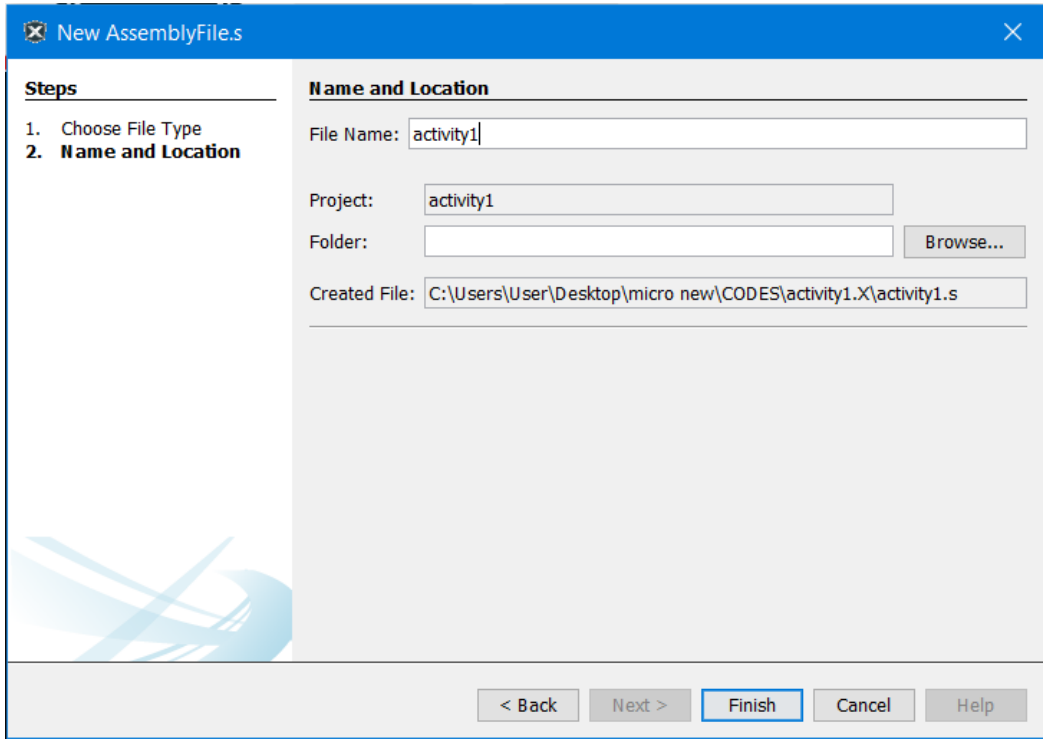


Starting with MPLABX (Activity 1)

Step 6: Right click on Source Files and select AssemblyFiles.s

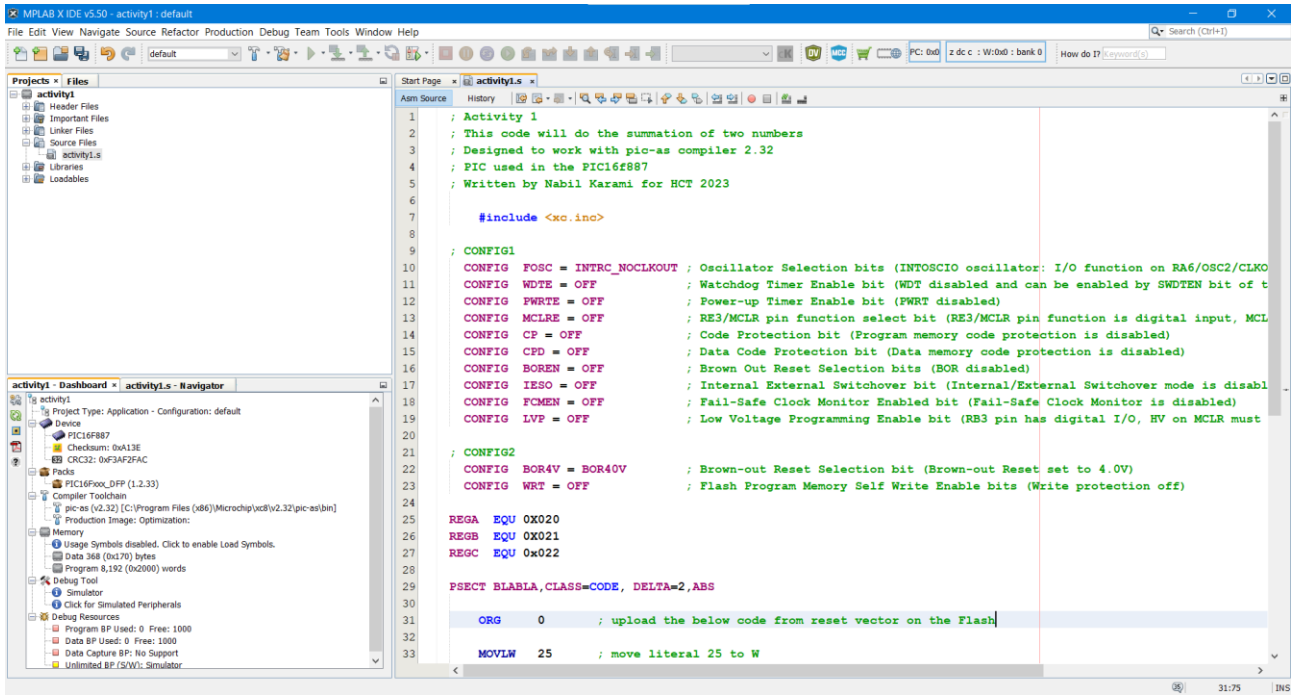


Step 7: Name the file as activity1



Lab

Starting with MPLABX (Activity 1)



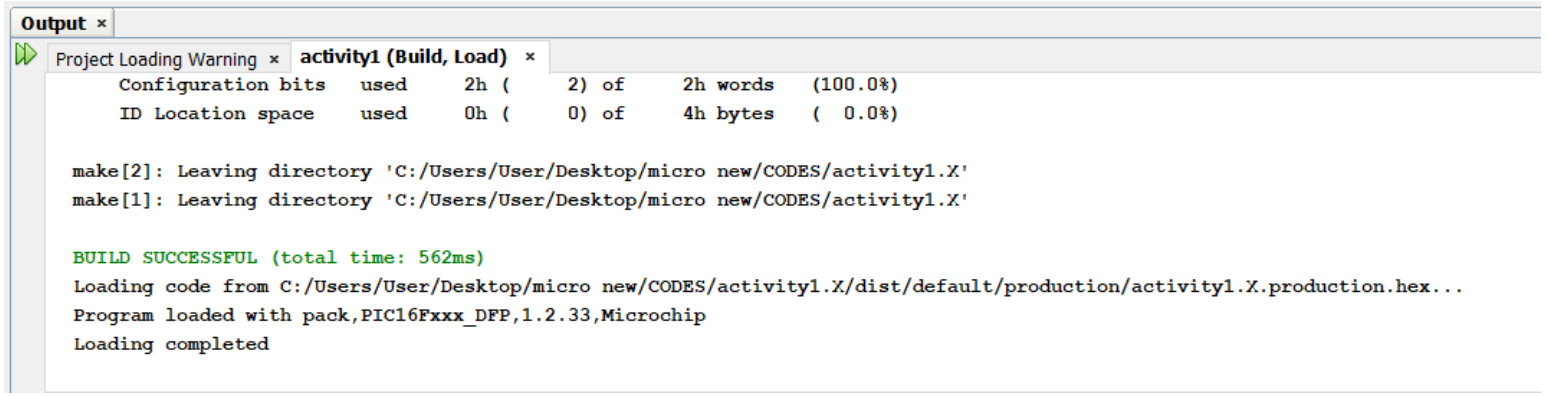
Step 8: Paste the Activity code and then click on “Build main project”



or use function key F11

Step 9: You must see the below message on the output window

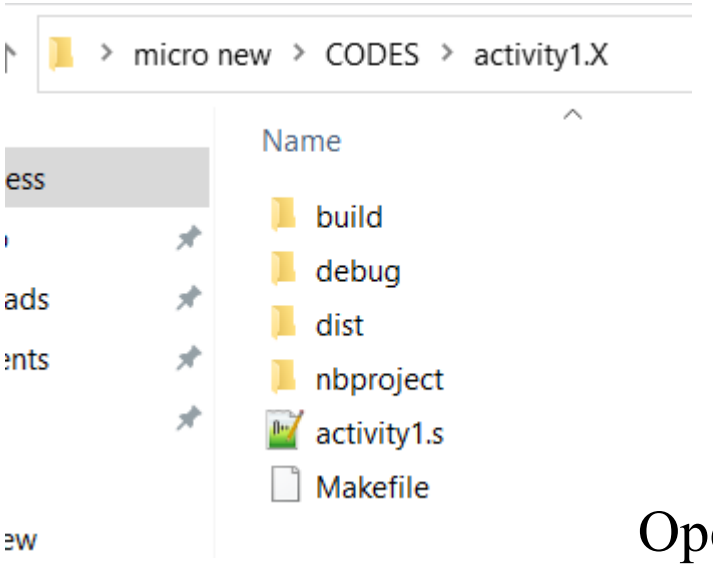
BUILD SUCCESSFUL



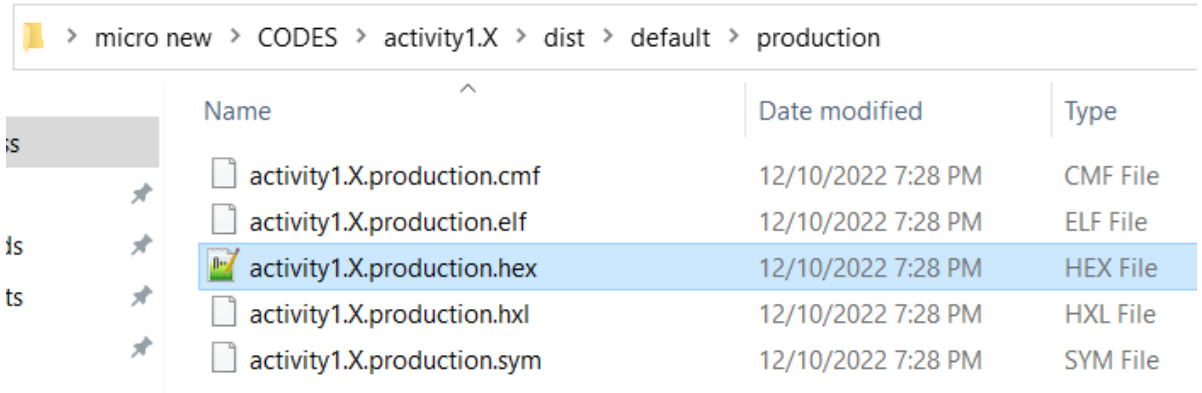
Lab

Starting with MPLABX (Activity 1)

BUILD SUCCESSFUL mean that a list of files is created in the destination folder



The Hex file is the machine language file generated by the compiler. We need this file to upload it to the Flash memory of the PIC using a programmer



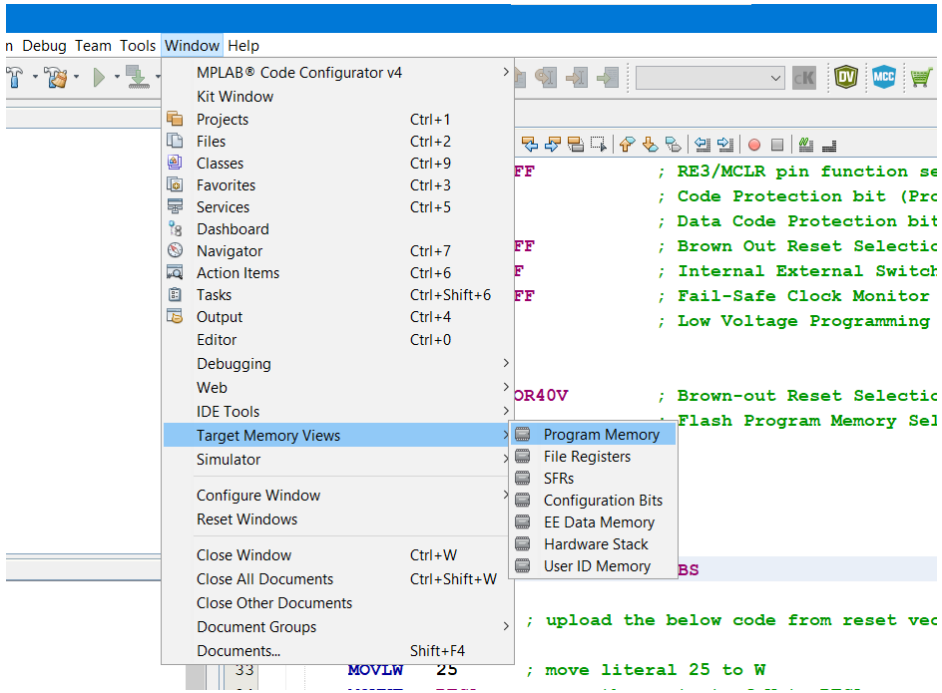
Open the Hex file in WordPad or Notepad and you must see the Hexadecimal numbers representing the code written in assembly

```
:10000000193A0001E30A10020082107A20007 28F7
:04400E00D420FF3F7C
:00000001FF
```

Lab

Starting with MPLABX (Activity 1)

To see how the code is written in Flash memory, go to Windows and click on the Program Memory



The instructions are listed in the Flash one by one starting Address 0

	Line	Address	Opcode	Label	DisAssy
➪	1	0000	3019		MOVLW 0x19
	2	0001	00A0		MOVWF 0x20
	3	0002	301E		MOVLW 0x1E
	4	0003	00A1		MOVWF 0x21
	5	0004	0820		MOVF 0x20, W
	6	0005	0721		ADDWF 0x21, W
	7	0006	00A2		MOVWF 0x22
	8	0007	2807		GOTO 0x7
	9	0008	3FFF		ADDLW 0xFF
	10	0009	3FFF		ADDLW 0xFF

```
MOVLW   25
MOVWF   REGA

MOVLW   30
MOVWF   REGB

MOVF    REGA, W
ADDWF   REGB, W
MOVWF   REGC

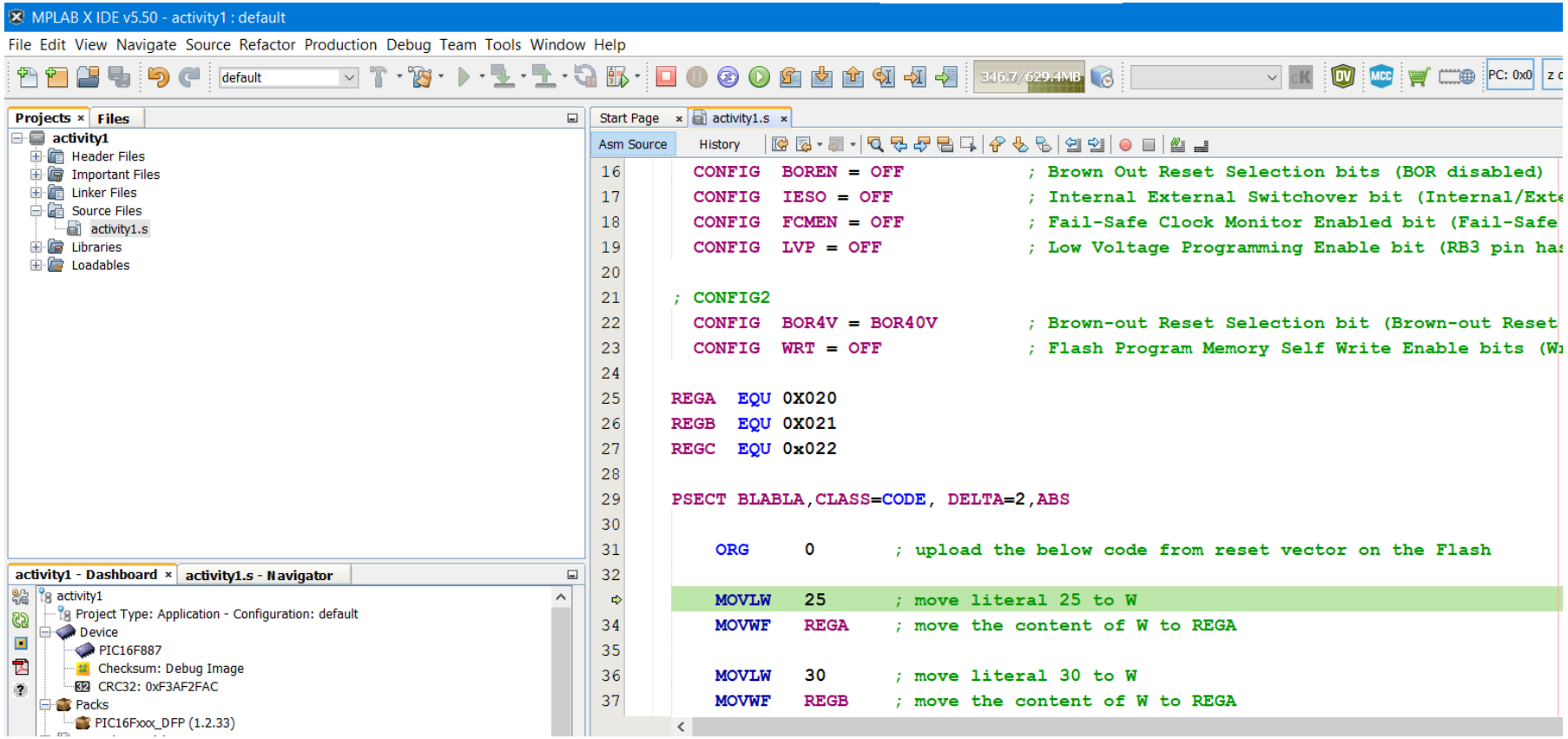
GOTO    $
```

The free spaces in Flash will be filled by ADDLW 0xFF
The opcode of ADDLW 0xFF is 11,1111,1111,1111
i.e., free spaces are filled by 1s

Starting with MPLABX (Activity 1)

Click on Debug main project button 

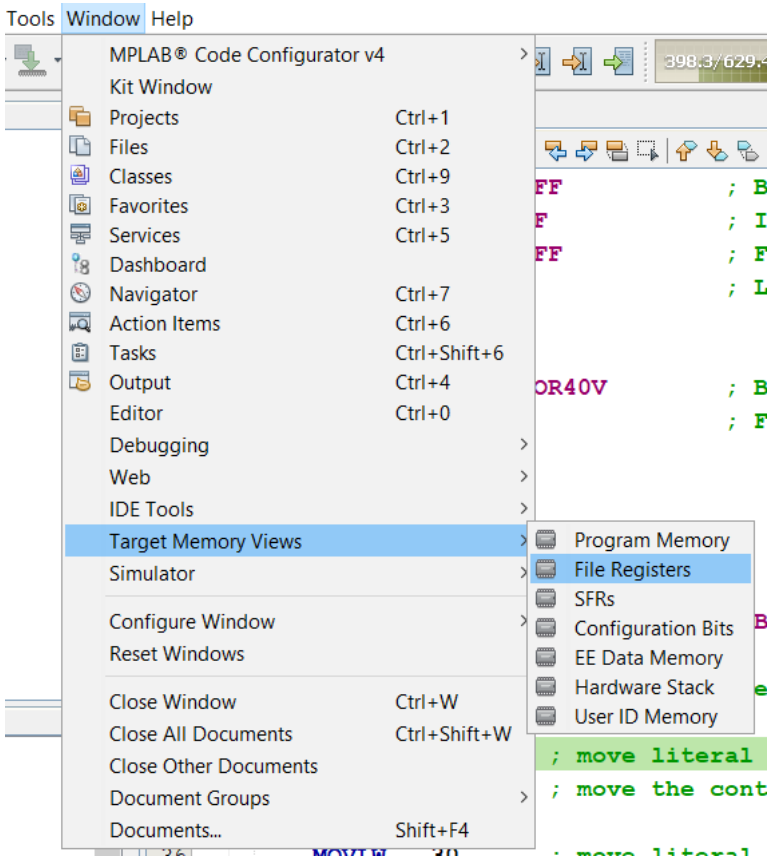
The debugger starts and focus on the reset vector



Starting with MPLABX (Activity 1)

Click on Debug main project button 

The debugger starts and focus on the reset vector



REGA=0 REGB=0 REGC=0

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
000	00	00	00	18	00	00	00	00	00	00	00	00	00	00	00	00	....
010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
080	00	FF	00	18	00	FF	FF	FF	FF	0F	00	00	00	00	10	60	....`
090	00	00	FF	00	00	FF	00	00	02	00	00	00	00	01	00	00	....

NB: Numbers are represented in HEX

Starting with MPLABX (Activity 1)

Click on Step over button



And check how the code runs

The simulator must freeze at GOTO $

Now you can check again the register values

```
28
29  PSECT BLABLA,CLASS=CODE, DELTA=2,ABS
30
31      ORG     0       ; upload the below code from reset vector on the Flash
32
33      MOVLW   25      ; move literal 25 to W
34      MOVWF   REGA    ; move the content of W to REGA
35
36      MOVLW   30      ; move literal 30 to W
37      MOVWF   REGB    ; move the content of W to REGA
38
39      MOVF    REGA, W ; move the content of REGA to W
40      ADDWF   REGB, W ; ADD it with the content of REGB (the result is in W)
41      MOVWF   REGC    ; then move the W content to REGC
42
        GOTO    $       ; do nothing
44
```

REGA=25

REGB=30

REGC=55

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
000	00	00	07	1A	00	00	00	00	00	00	00	00	00	00	00	00	....
010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
020	19	1E	37	00	00	00	00	00	00	00	00	00	00	00	00	00	..7..... .....
030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
080	00	FF	07	1A	00	FF	FF	FF	FF	0F	00	00	00	00	10	60	....`
090	00	00	FF	00	00	FF	00	00	02	00	00	00	00	01	00	00	....
0A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

End of LO1

LO2



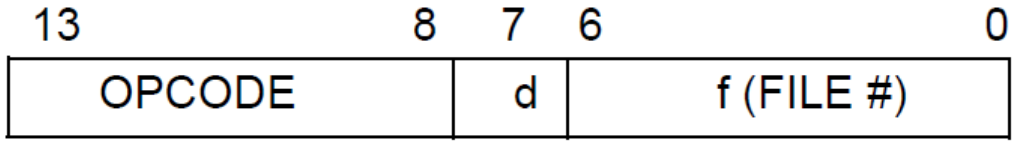
Week 3

Lecture 5 / LO2

RAM Direct Accessing
PIC clocking systems

Accessing the RAM

- It was clear from the instruction set that the size of the opcode is 14 bits.
- The instructions that deals with files like MOVWF, ADDWF, ANDWF… are given as



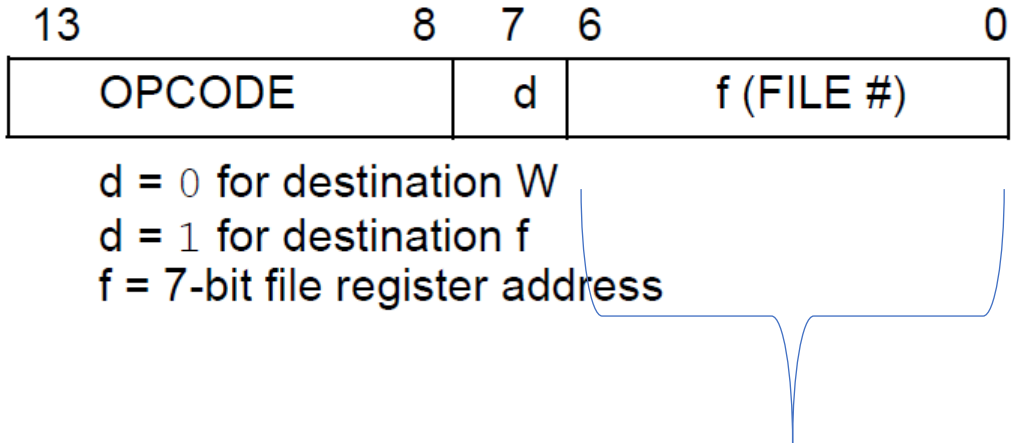
d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address

Space available in each memory location in Flash

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C, DC, Z	1, 2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1, 2
CLRF	f	Clear f	1	00	0001	lfff	ffff	Z	2
CLRW	–	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1, 2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1, 2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1, 2, 3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1, 2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1, 2, 3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1, 2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1, 2
MOVWF	f	Move W to f	1	00	0000	lfff	ffff		
NOP	–	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1, 2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1, 2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C, DC, Z	1, 2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1, 2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1, 2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1, 2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1, 2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C, DC, Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call Subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	–	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$, $\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	–	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	–	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	–	Go into Standby mode	1	00	0000	0110	0011	$\overline{TO}$, $\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C, DC, Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Accessing the RAM

- This means that the instruction is coded in 6 bits (from 8 to 13)
- The destination bit takes one bit size
- And the file address takes only 7 bits !! (from 0 to 6)
- 7 bits means that the address that can be accessed are from 000,0000 to 111,1111, i.e., from 0 to 127



We can only talk to 128 registers !!!!
What about the others registers in the RAM ??

Accessing the RAM

- Now if we look to the RAM, we can see that the addresses of the registers start from 0 to 511 (0 to 1FF, i.e., 9 bits)
- This means that the addresses range are 0,0000,0000 to 1,1111,1111

- The addresses of:
  - Bank 0 are from 0 to 127
  - Bank 1 are from 128 to 255
  - Bank 2 are from 256 to 383
  - Bank 3 are from 384 to 511

Bank 0	File Address	Bank 1	File Address	Bank 2	File Address	Bank 3	File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers 16 Bytes	116h	General Purpose Registers 16 Bytes	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h 3Fh 40h 6Fh	General Purpose Registers 80 Bytes	A0h EFh	General Purpose Registers 80 Bytes	120h 16Fh	General Purpose Registers 80 Bytes	1A0h 1EFh
accesses 70h-7Fh	70h 7Fh	accesses 70h-7Fh	F0h FFh	accesses 70h-7Fh	170h 17Fh	accesses 70h-7Fh	1F0h 1FFh

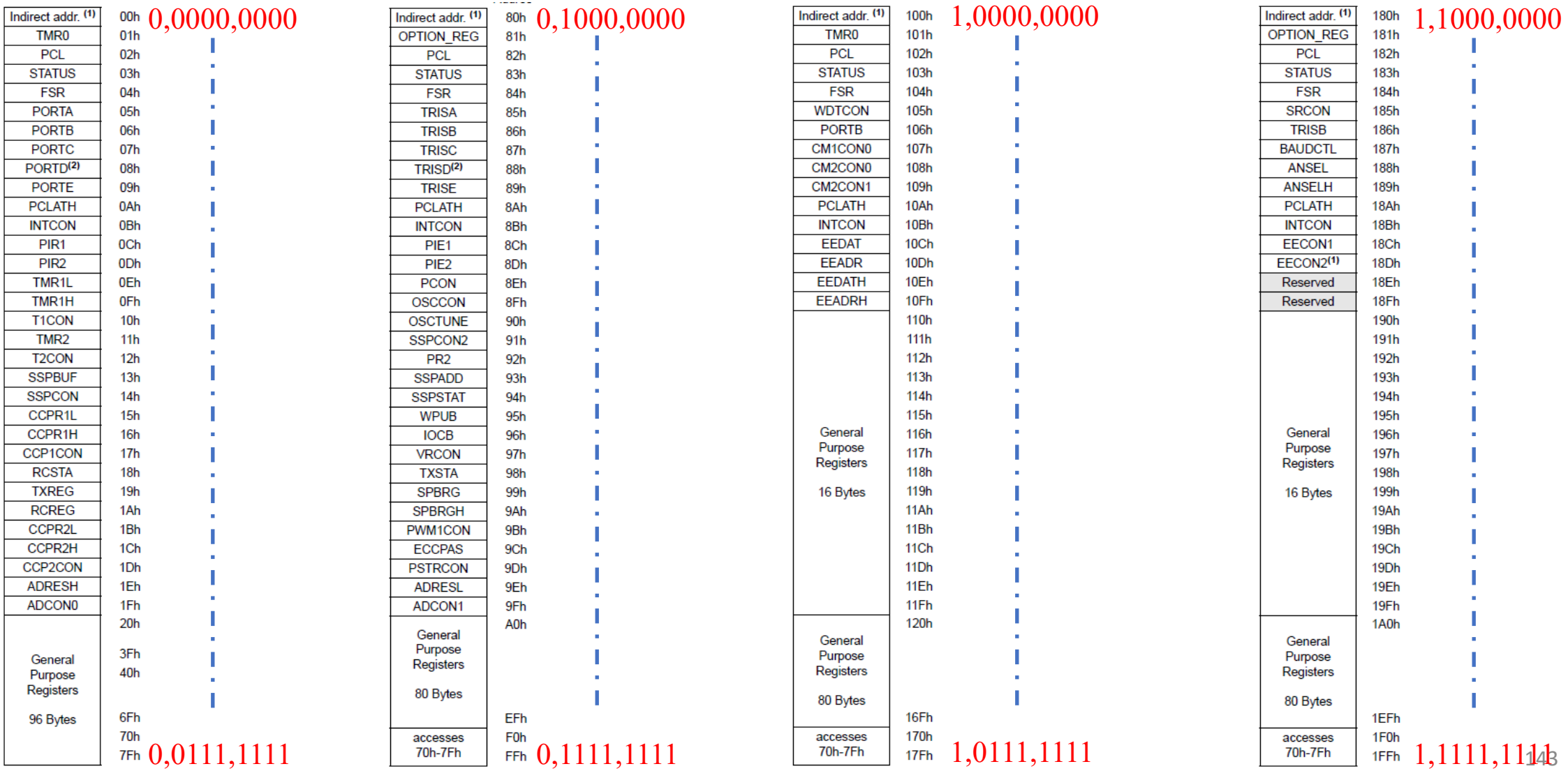
Accessing the RAM

- Also, if we focus on the addresses of the registers having the same rows, like for example PORTD, TRISD, CM2CON0 and ANSEL, we can conclude that they have the same least 7 bits

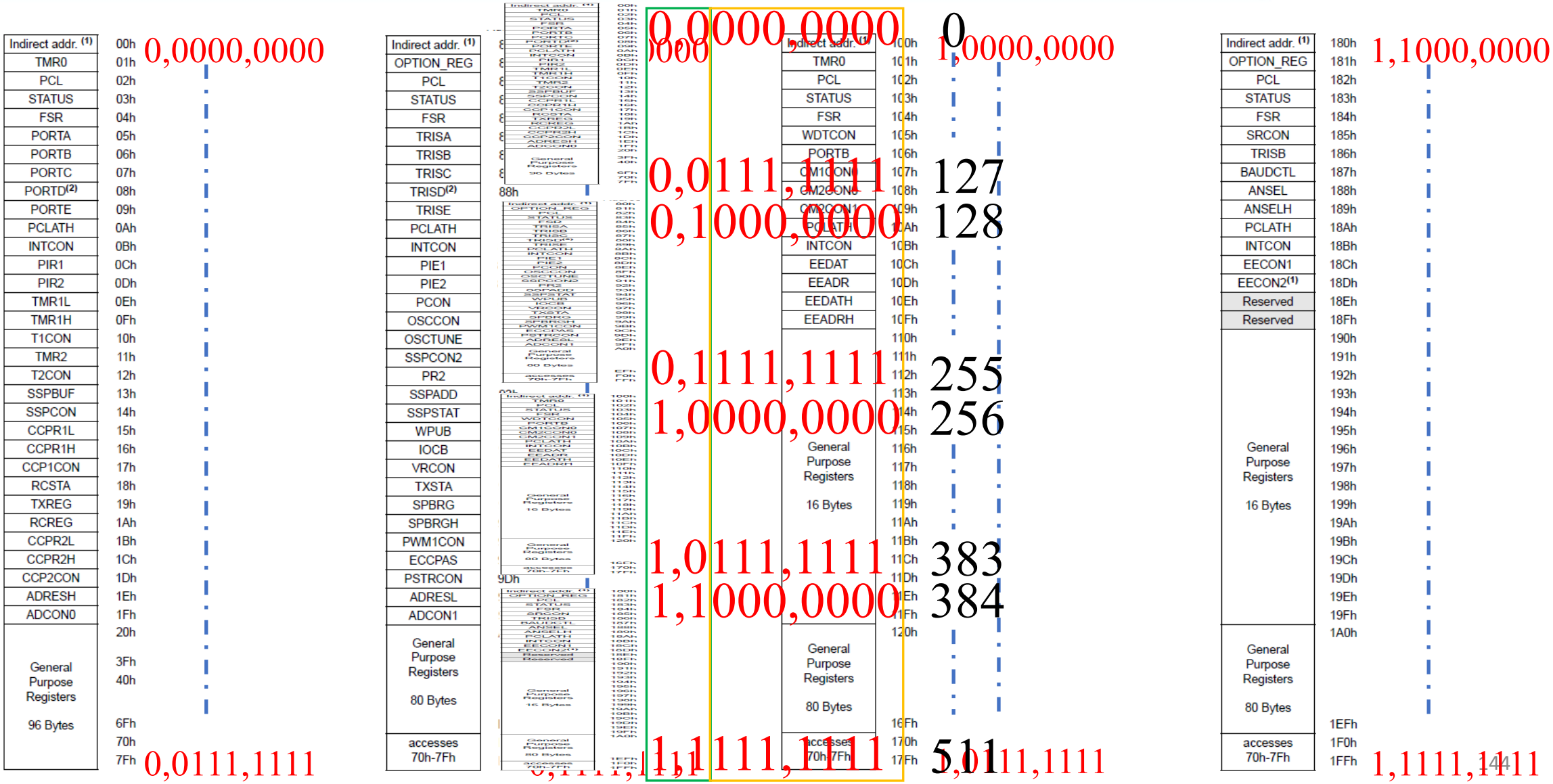
- PORTD: 08H = 0,0000,1000
- TRISD: 88H = 0,1000,1000
- CM2CON0: 108H = 1,0000,1000
- ANSEL: 188H = 1,1000,1000

	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers	116h	General Purpose Registers	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h	16 Bytes	119h	16 Bytes	199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h 3Fh 40h 6Fh	General Purpose Registers 80 Bytes	A0h EFh	General Purpose Registers 80 Bytes	120h 16Fh	General Purpose Registers 80 Bytes	1A0h 1EFh
	70h 7Fh	accesses 70h-7Fh	F0h FFh	accesses 70h-7Fh	170h 17Fh	accesses 70h-7Fh	1F0h 1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

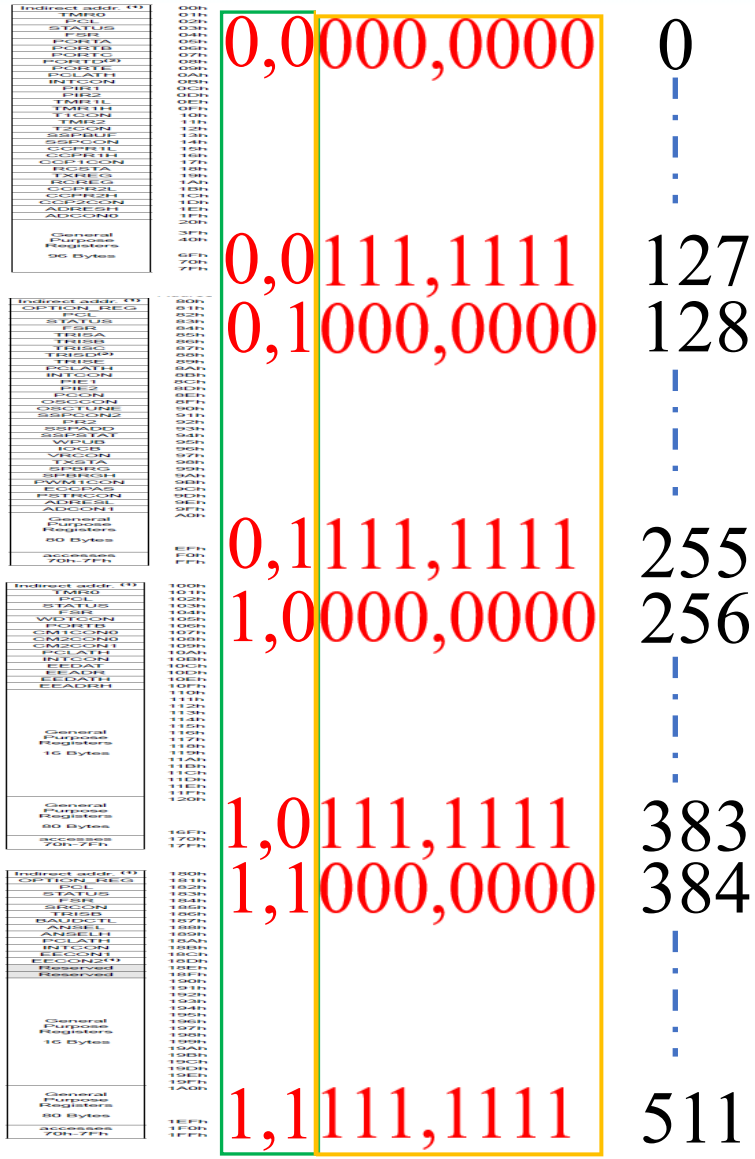
Accessing the RAM



Accessing the RAM



Accessing the RAM



For the 7 least bits, same values are repeated

Same addresses

Accessing the RAM

So, we can conclude that if we write for example

MOVLW 20 → MOVLW 20
MOVWF TRISD → MOVWF 0001000B

- Is same as writing

MOVLW 20 → MOVLW 20
MOVWF CM2CON0 → MOVWF 0001000B

- and same as

MOVLW 20 → MOVLW 20
MOVWF ANSEL → MOVWF 0001000B

Same least 7 bits

The most significant 2 bits cannot fit inside the opcode

	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers 16 Bytes	116h	General Purpose Registers 16 Bytes	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h – 7Fh	General Purpose Registers 80 Bytes	A0h – EFh	General Purpose Registers 80 Bytes	120h – 16Fh	General Purpose Registers 80 Bytes	1A0h – 1EFh
		accesses 70h-7Fh	F0h – FFh	accesses 70h-7Fh	170h – 17Fh	accesses 70h-7Fh	1F0h – 1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

Actually, the 20 will be written in PORTD not in TRISD, neither in CM2CN0 and neither in ANSEL.

In fact, only registers in bank0 are accessible

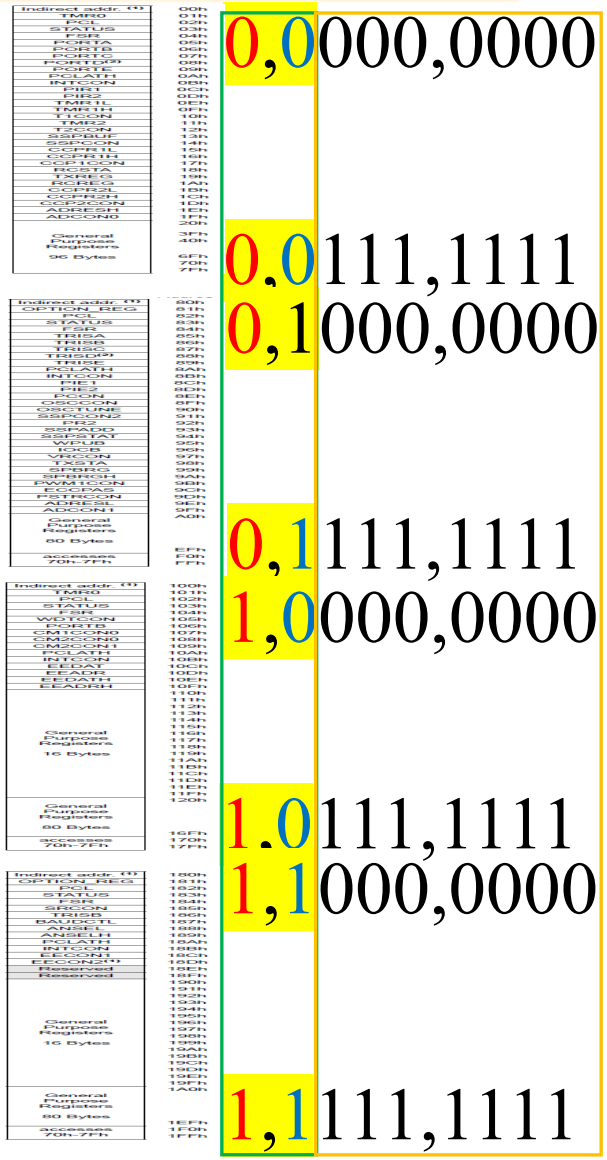
Accessing the RAM

How to overcome this case and have access to other registers in the other banks (1, 2 ,and 3) !!

The answer is we have to write the most 2 significant bits of the register address before accessing the register.

The 2 MSbits should be written in two bits called RP1 and RP0

- PORTD: 08H = 0,0000,1000
- TRISD: 88H = 0,1000,1000
- CM2CON0: 108H = 1,0000,1000
- ANSEL: 188H = 1,1000,1000



Accessing the RAM

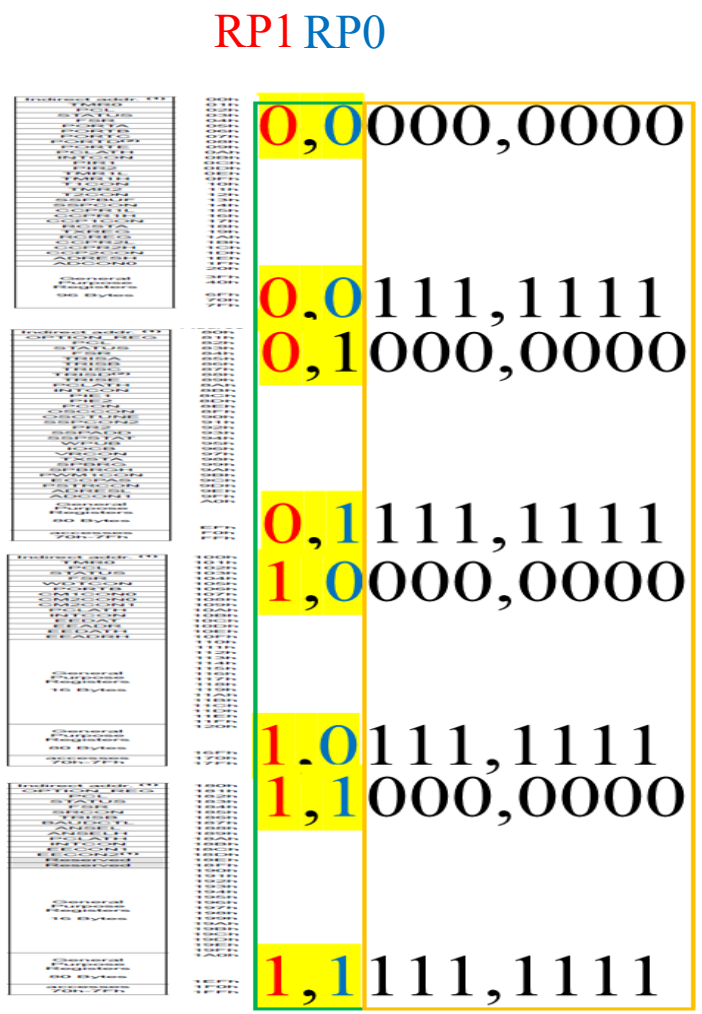
Thus to write on a register in

- bank 0, we have to make RP1=0 and RP0=0
- bank 1, we have to make RP1=0 and RP0=1
- bank 2, we have to make RP1=1 and RP0=0
- bank 3, we have to make RP1=1 and RP0=1

BANK	RP1	RP0
0	0	0
1	0	1
2	1	0
3	1	1

RP1 and RP0 represent the KEYS of the banks

Now the question is where are the RP1 and RP0 and how to write on these bits !!



Accessing the RAM

- RP1 and RP0 are located in the RAM itself, more precisely in a register called STATUS.
- BUT! How to accesses the KEYS in a LOCKED bank ?
- STATUS register is available in the 4 banks and thus it is accessible whatever are RP1 and RP0.
- STATUS register includes 8 bits. RP1 and RP0 are bits #6 and 5, respectively.

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
03h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C

	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers 16 Bytes	116h	General Purpose Registers 16 Bytes	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h 3Fh 40h 6Fh 70h 7Fh	General Purpose Registers 80 Bytes accesses 70h-7Fh	A0h EFh F0h FFh	General Purpose Registers 80 Bytes accesses 70h-7Fh	120h 16Fh 170h 17Fh	General Purpose Registers 80 Bytes accesses 70h-7Fh	1A0h 1EFh 1F0h 1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

The remaining bits will be discussed later

Accessing the RAM

To conclude, writing on registers of different BANKS will be like,

```
BCF     RP1
BCF     RP0     ; ACCESS TO BANK 0
MOVLW   20
MOVWF   PORTD

BCF     RP1
BSF     RP0     ; ACCESS TO BANK 1
MOVLW   20
MOVWF   TRISD

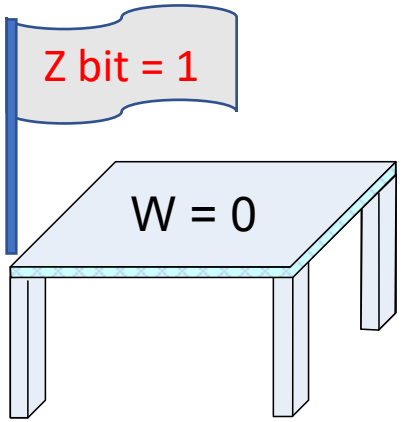
BSF     RP1
BCF     RP0     ; ACCESS TO BANK 2
MOVLW   20
MOVWF   CM2CON0

BSF     RP1
BSF     RP0     ; ACCESS TO BANK 3
MOVLW   20
MOVWF   ANSEL
```

	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers	116h	General Purpose Registers	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h	16 Bytes	119h	16 Bytes	199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
	20h		A0h		120h		1A0h
General Purpose Registers 96 Bytes	3Fh 40h 6Fh 70h 7Fh	General Purpose Registers 80 Bytes accesses 70h-7Fh	EFh F0h FFh	General Purpose Registers 80 Bytes accesses 70h-7Fh	16Fh 170h 17Fh	General Purpose Registers 80 Bytes accesses 70h-7Fh	1EFh 1F0h 1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

Status Register

- The STATUS register, shown below, contains:
  - the arithmetic status of the ALU (Z, DC and C bits) these bits are called flags
  - the Reset status
  - the bank select bits for data memory (GPR and SFR)
- The Z bit is used to check if the W=0 or not. Z = 1 when W = 0
- This bit is very important in comparing values (equal or not)



Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
03h	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C

bit 2 **Z:** Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC:** Digit Carry/$\overline{\text{Borrow}}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions)(1)
1 = A carry-out from the 4th low-order bit of the result occurred
0 = No carry-out from the 4th low-order bit of the result

bit 0 **C:** Carry/$\overline{\text{Borrow}}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions)(1)
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

- The carry bit (C bit) is used to check if an arithmetic operation is leading to a value more than 255 (more than 8 bits) or a negate value

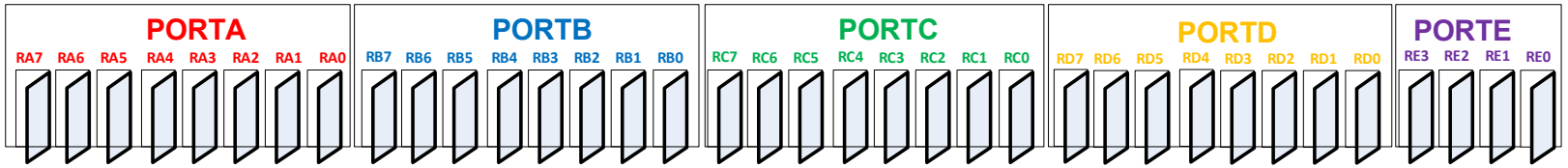
Check the Arithmetic Operations (Subtraction) slide

Lecture

Dealing with PORTS

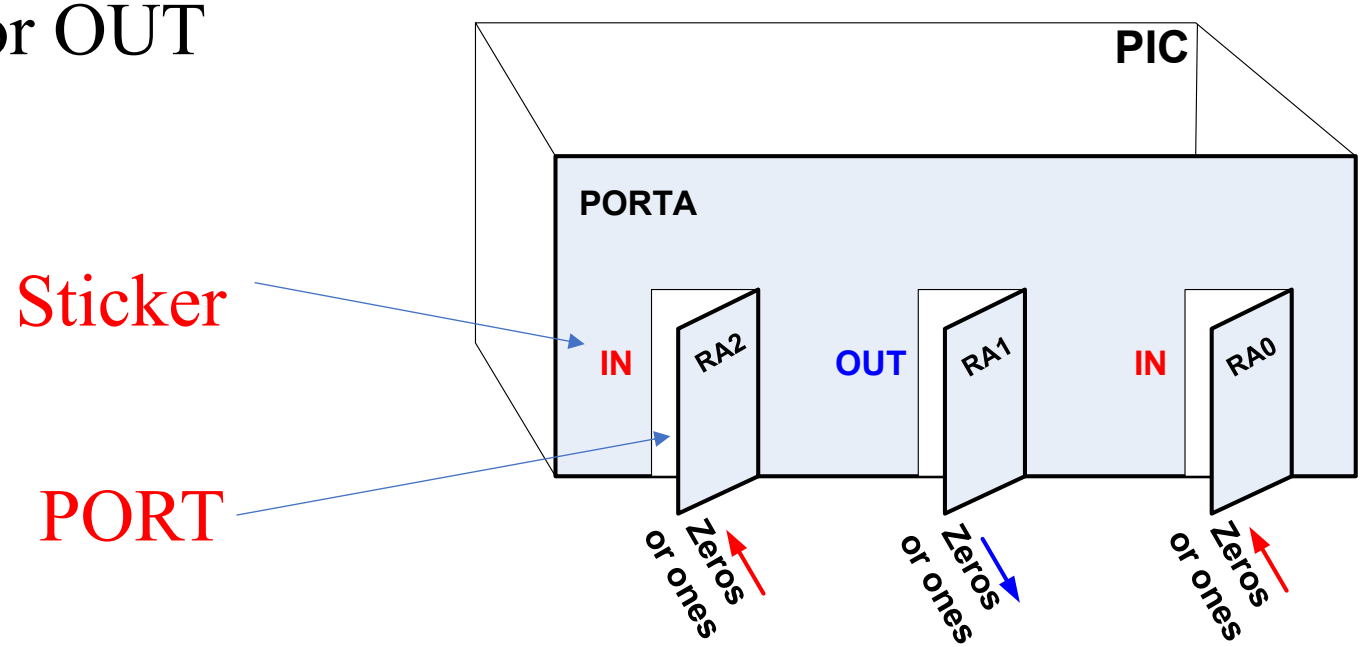
- Remember from the previous slides that the PIC16F887 has 36 I/O pins distributed on 5 ports..
- These ports are mapped in RAM
- "Mapped" means that accessing the ports is achieved by accessing their registers in RAM
- These registers are readable and writable, i.e., we can write values on the ports and we can read values from ports, thus called I/O ports.

	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers 16 Bytes	116h	General Purpose Registers 16 Bytes	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h – 6Fh	General Purpose Registers 80 Bytes	A0h – EFh	General Purpose Registers 80 Bytes	120h – 16Fh	General Purpose Registers 80 Bytes	1A0h – 1EFh
	70h – 7Fh	accesses 70h-7Fh	F0h – FFh	accesses 70h-7Fh	170h – 17Fh	accesses 70h-7Fh	1F0h – 1FFh
Bank 0		Bank 1		Bank 2		Bank 3	



Dealing with PORTS

- PORTS **cannot** be Input and Output at the same time.
- We have to select the direction first
- This is done by writing on the TRIS registers located in BANK1.
- TRIS are like the stickers. They define the PORTS to be IN or OUT

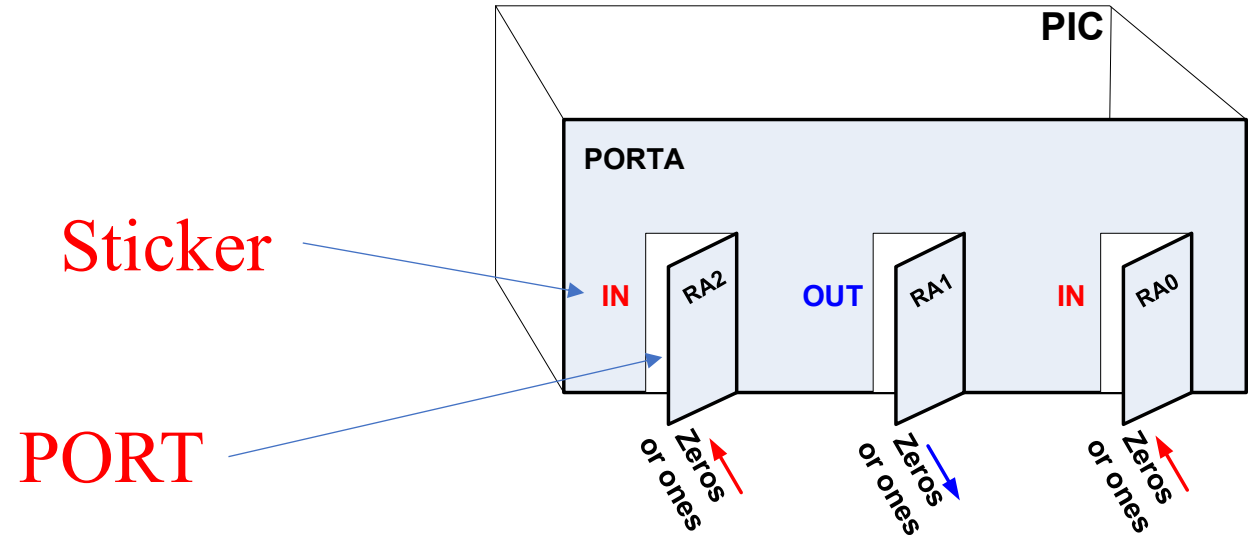


Bank 0	File Address	Bank 1	File Address	Bank 2	File Address	Bank 3	File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers 16 Bytes	116h	General Purpose Registers 16 Bytes	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h–6Fh	General Purpose Registers 80 Bytes	A0h–EFh	General Purpose Registers 80 Bytes	120h–16Fh	General Purpose Registers 80 Bytes	1A0h–1EFh
	70h–7Fh	accesses 70h-7Fh	F0h–FFh	accesses 70h-7Fh	170h–17Fh	accesses 70h-7Fh	1F0h–1FFh

Dealing with PORTS

- TRIS = 0 means that the PORT is OUTPUT
- TRIS = 1 means that the PORT is INPUT

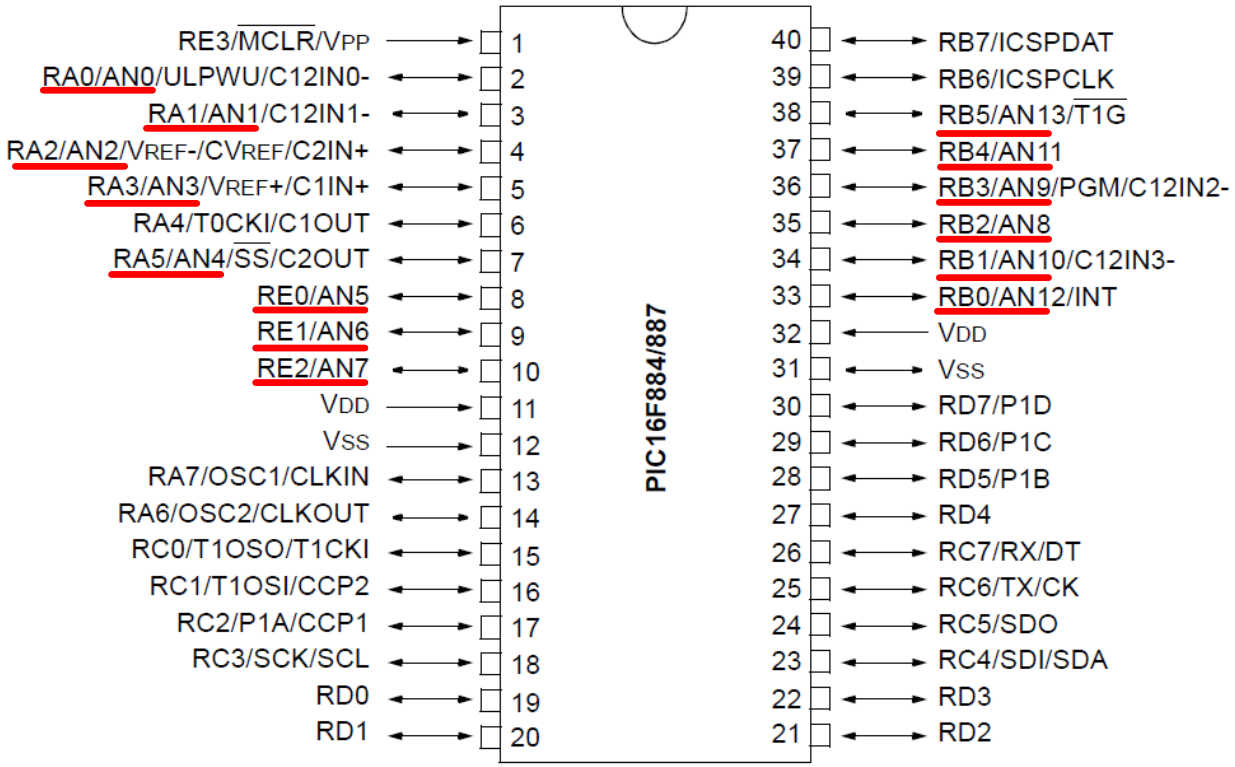
- PORT configured as OUTPUT can deliver 0 or 1 logic, i.e., 0 or 5 volts
- PORT configured as INPUT can read 0 or 1 logic, i.e., 0 or 5 volts
- All PORTS are digital ports



	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers 16 Bytes	116h	General Purpose Registers 16 Bytes	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h – 6Fh	General Purpose Registers 80 Bytes	A0h – EFh	General Purpose Registers 80 Bytes	120h – 16Fh	General Purpose Registers 80 Bytes	1A0h – 1EFh
	70h – 7Fh	accesses 70h-7Fh	F0h – FFh	accesses 70h-7Fh	170h – 17Fh	accesses 70h-7Fh	1F0h – 1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

Dealing with PORTS

- Some of the PORTS like PORTA, PORTB and PORTE can also be analog input ports. i.e., a voltage value between 0 and 5 can also be recognized by the PIC.
- This can be configured by the two analog select registers ANSEL and ANSELH located on BANK3. 1 means analog, 0 means digital.



	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers 16 Bytes	116h	General Purpose Registers 16 Bytes	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h – 7Fh	General Purpose Registers 80 Bytes	A0h – EFh	General Purpose Registers 80 Bytes	120h – 16Fh	General Purpose Registers 80 Bytes	1A0h – 1EFh
Bank 0		Bank 1		Bank 2		Bank 3	



Dealing with PORTS

To conclude, in RAM we have

- 5 port registers for reading and writing
- 5 tris registers for defining the direction of the pins
- 2 analog-select register to select the mode of PORTA, PORTB and PORTE to be analog or digital

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
05h	PORTA(3)	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
06h	PORTB(3)	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
07h	PORTC(3)	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0
08h	PORTD(3,4)	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0
09h	PORTE(3)	—	—	—	—	RE3	RE2(4)	RE1(4)	RE0(4)
85h	TRISA	TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0
86h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0
87h	TRISC	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0
88h	TRISD(3)	TRISD7	TRISD6	TRISD5	TRISD4	TRISD3	TRISD2	TRISD1	TRISD0
89h	TRISE	—	—	—	—	TRISE3	TRISE2(3)	TRISE1(3)	TRISE0(3)
188h	ANSEL	ANS7(2)	ANS6(2)	ANS5(2)	ANS4	ANS3	ANS2	ANS1	ANS0
189h	ANSELH	—	—	ANS13	ANS12	ANS11	ANS10	ANS9	ANS8

	File Address		File Address		File Address		File Address
Indirect addr. (1)	00h	Indirect addr. (1)	80h	Indirect addr. (1)	100h	Indirect addr. (1)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	WDTCON	105h	SRCON	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	CM1CON0	107h	BAUDCTL	187h
PORTD(2)	08h	TRISD(2)	88h	CM2CON0	108h	ANSEL	188h
PORTE	09h	TRISE	89h	CM2CON1	109h	ANSELH	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDAT	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2(1)	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	0Fh	OSCCON	8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h	OSCTUNE	90h		110h		190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h	WPUB	95h		115h		195h
CCPR1H	16h	IOCB	96h	General Purpose Registers 16 Bytes	116h	General Purpose Registers 16 Bytes	196h
CCP1CON	17h	VRCON	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	SPBRGH	9Ah		11Ah		19Ah
CCPR2L	1Bh	PWM1CON	9Bh		11Bh		19Bh
CCPR2H	1Ch	ECCPAS	9Ch		11Ch		19Ch
CCP2CON	1Dh	PSTRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers 96 Bytes	20h	General Purpose Registers 80 Bytes	A0h	General Purpose Registers 80 Bytes	120h	General Purpose Registers 80 Bytes	1A0h
	3Fh 40h						
	6Fh		EFh		16Fh		1EFh
	70h	accesses 70h-7Fh	F0h	accesses 70h-7Fh	170h	accesses 70h-7Fh	1F0h
	7Fh		FFh		17Fh		1FFh
Bank 0		Bank 1		Bank 2		Bank 3	



Speed of Instruction Execution

- From the activity, we saw the compiler executing the code line by line. But what would be the execution speed in reality !.

- The speed on instruction execution depends on the oscillator frequency.
- Microcontrollers/microprocessors never work without oscillator.
- The oscillator generates the clocking signal for the sequential logic circuit of the chip (remember the clock signal of flipflops in digital circuit course)

- The PIC16F887 is designed to get clock from external generators or it can generates its own clocking signals.

- From the PIC datasheet, the maximum clocking frequency is 20Mhz.

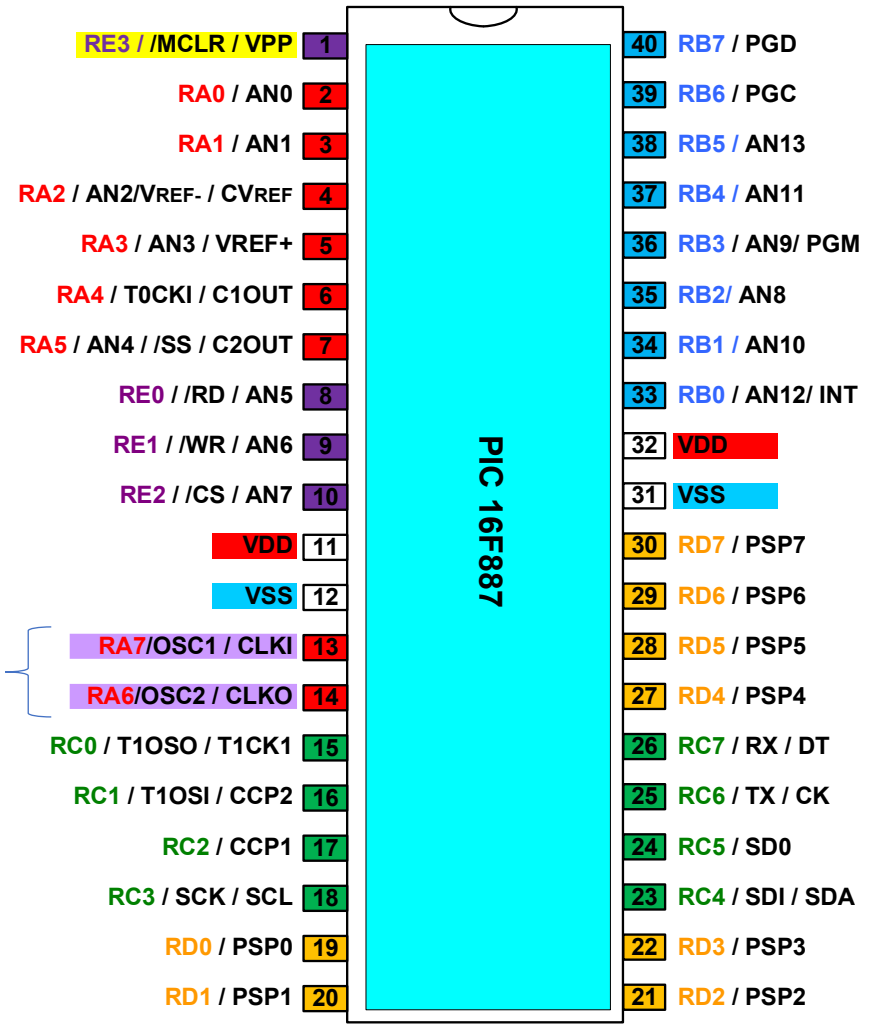
Speed of Instruction Execution

External clock generators rely on external circuitry for the clock source.

The clock signal can be generated from:

- Oscillator modules (EC mode),
- Quartz crystal resonators or ceramic resonators (LP, XT and HS modes)
- Resistor-Capacitor (RC) mode circuits.

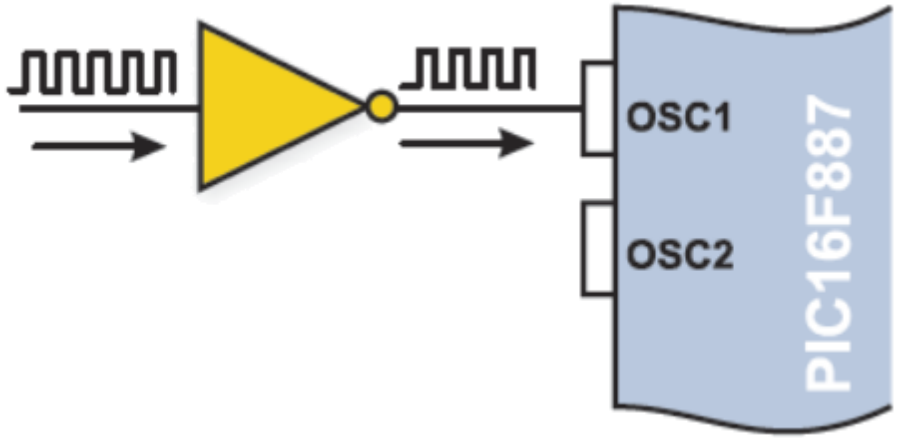
Oscillator is to be connected here



Speed of Instruction Execution

- The External Clock (EC) mode allows an externally generated logic level as the system clock source. When operating in this mode, an external clock source is connected to the OSC1 input and the OSC2 is available for general purpose I/O.

Signal generated by any oscillator circuit like "555 timer" of any combination/sequential circuit



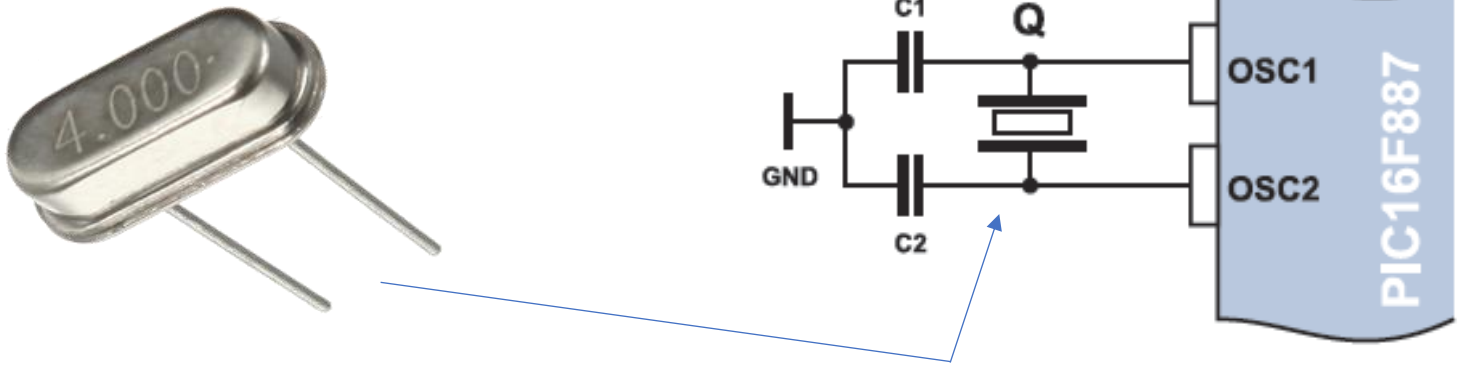
The instruction execution speed is ¼ of the signal frequency i.e., if the signal is 1kHz, the instruction speed is achieved at 250 Hz, i.e., 250 instructions are achieved in 1 second



Speed of Instruction Execution

The LP, XT and HS modes support the use of quartz crystal resonators or ceramic resonators connected to OSC1 and OSC2. The mode selects a low, medium or high gain setting of the internal inverter amplifier to support various resonator types and speed.

- **LP** Oscillator mode: Used for low crystal frequency
- **XT** Oscillator mode: Used for medium crystal frequency
- **HS** Oscillator mode: Used for high crystal frequency.



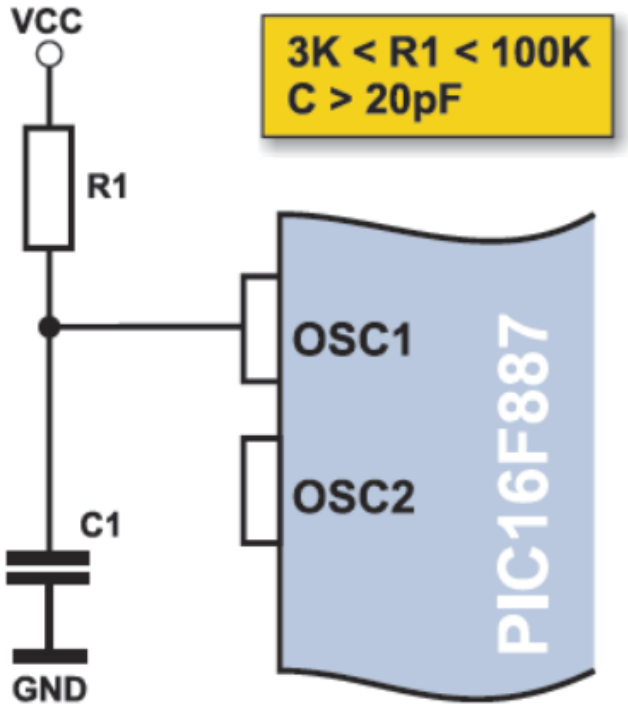
Mode	Frequency	C1, C2
LP	32 KHz	33pF
	200 KHz	15pF
XT	200 KHz	47-68 pF
	1 MHz	15 pF
	4 MHz	15 pF
HS	4 MHz	15 pF
	8 MHz	15-33 pF
	20 MHz	15-33 pF

The instruction execution speed is ¼ of the crystal frequency i.e., if the signal is 4MHz, the instruction speed is achieved at 1MHz i.e., 1000000 instructions are achieved in 1 second

Speed of Instruction Execution

The external Resistor-Capacitor (RC) modes support the use of an external RC circuit. This allows the designer maximum flexibility in frequency choice while keeping costs to a minimum when clock accuracy is not required.

- There are two modes: RC and RCIO.
  - In RC mode, the RC circuit connects to OSC1. OSC2/CLKOUT outputs the RC oscillator frequency divided by 4. This signal may be used to provide a clock for external circuitry, synchronization, calibration, test or other application requirements.
  - In RCIO mode, the RC circuit is connected to OSC1. OSC2 becomes an additional general purpose I/O pin.



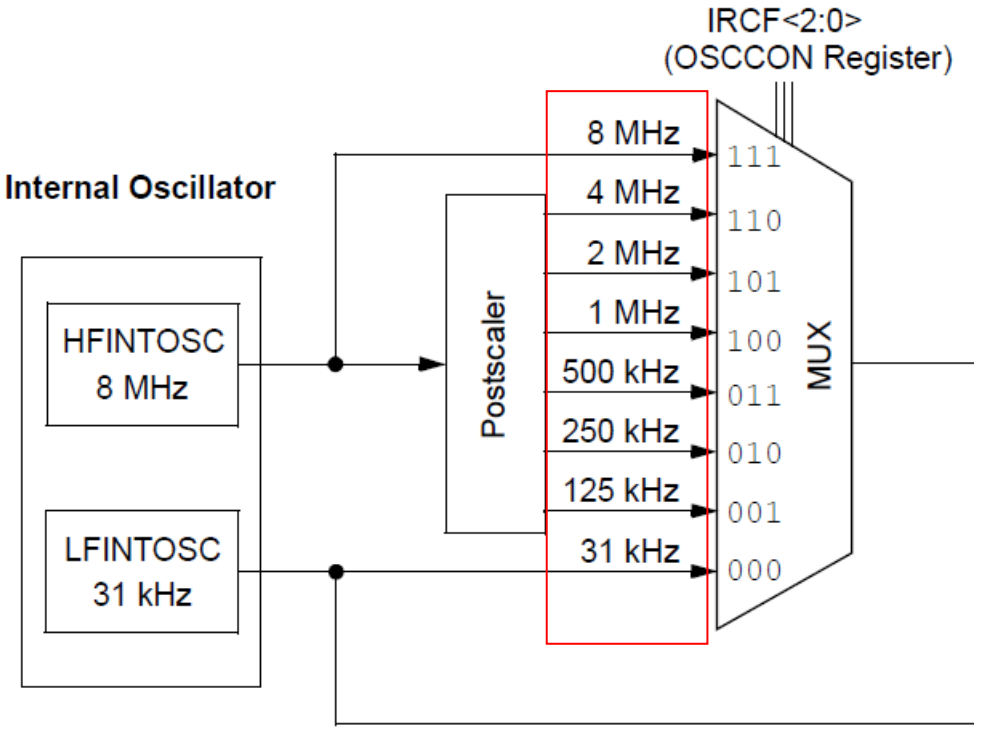
Speed of Instruction Execution

The PIC16F887 is also able to generate its own clocking signals due to two internal oscillators called:

- High speed internal oscillator: HSINTOSC
- Low speed internal oscillator: LSINTOSC

A frequency divider circuit is used to divide the HSINTOSC into 6 different frequencies.

Thus, in total, 8 different internal frequencies can be used depends on our requirement.



Speed of Instruction Execution

The selection of the clocking source (internal or external) is done using the CONFIGURATION bits (to be discussed later). In case internal oscillator is used, the selection of the frequency will be selected using OSCCON register available in BANK1. Default is 4MHz.

REGISTER 4-1: OSCCON: OSCILLATOR CONTROL REGISTER

U-0	R/W-1	R/W-1	R/W-0	R-1	R-0	R-0	R/W-0
—	IRCF2	IRCF1	IRCF0	OSTS(1)	HTS	LTS	SCS
bit 7							bit 0

bit 7 **Unimplemented:** Read as '0'

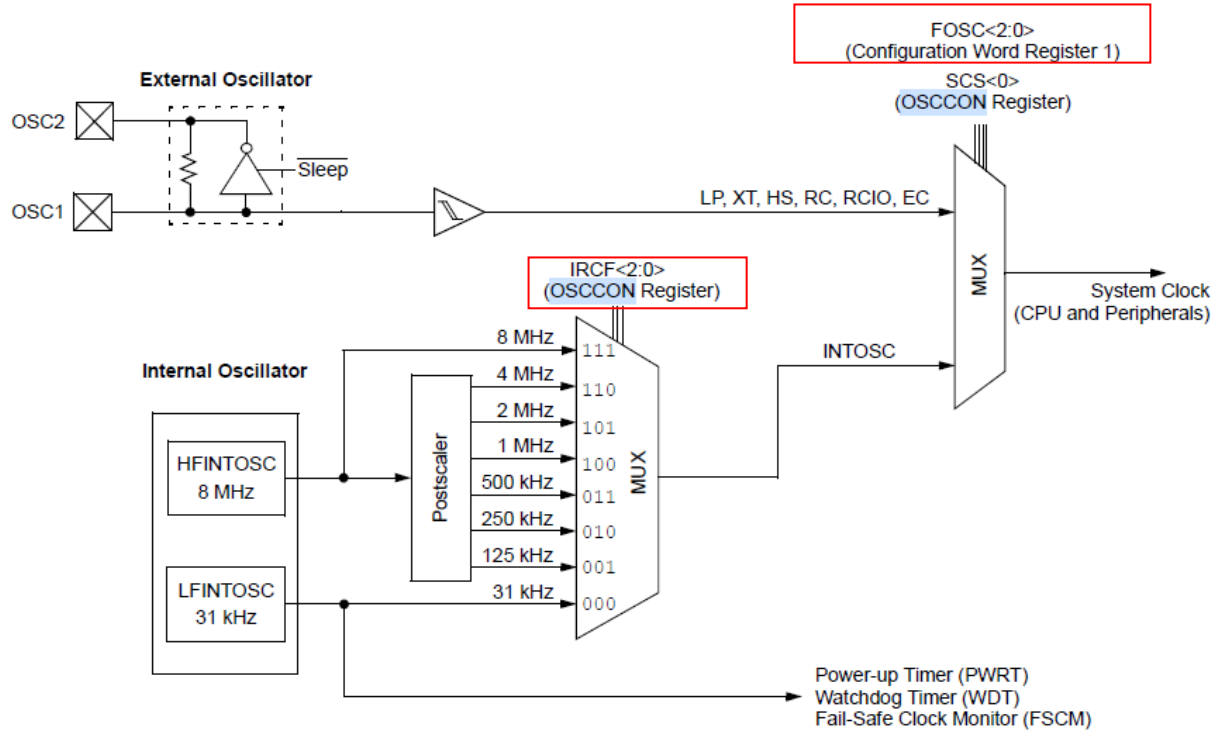
bit 6-4 **IRCF<2:0>:** Internal Oscillator Frequency Select bits
111 = 8 MHz
110 = 4 MHz (default)
101 = 2 MHz
100 = 1 MHz
011 = 500 kHz
010 = 250 kHz
001 = 125 kHz
000 = 31 kHz (LFINTOSC)

bit 3 **OSTS:** Oscillator Start-up Time-out Status bit(1)
1 = Device is running from the clock defined by FOSC<2:0> of the CONFIG1 register
0 = Device is running from the internal oscillator (HFINTOSC or LFINTOSC)

bit 2 **HTS:** HFINTOSC Status bit (High Frequency – 8 MHz to 125 kHz)
1 = HFINTOSC is stable
0 = HFINTOSC is not stable

bit 1 **LTS:** LFINTOSC Stable bit (Low Frequency – 31 kHz)
1 = LFINTOSC is stable
0 = LFINTOSC is not stable

bit 0 **SCS:** System Clock Select bit
1 = Internal oscillator is used for system clock
0 = Clock source defined by FOSC<2:0> of the CONFIG1 register



Speed of Instruction Execution

Execution speed calculation.

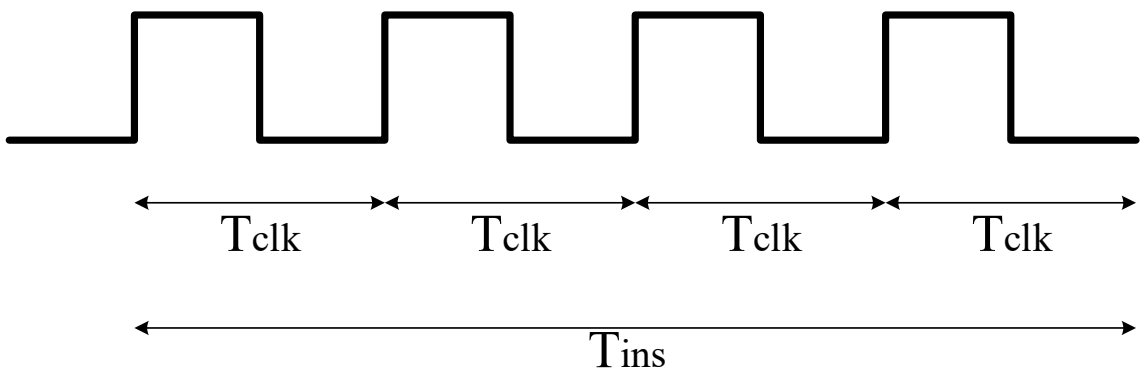
The internal or the external clock generators produce a square wave at predefined frequencies. This is called the oscillating frequency Fosc or clocking frequency Fclk

Thus the clock period Tclk can be calculated as Tclk = 1/Fclk

Each instruction needs at least 4 clocks to be totally executed. This is called Tins.

Thus Tins = 4 x Tclk, and Fins = Fclk/4

Clock signal generated by the crystal, RC circuit, oscillator or resonator



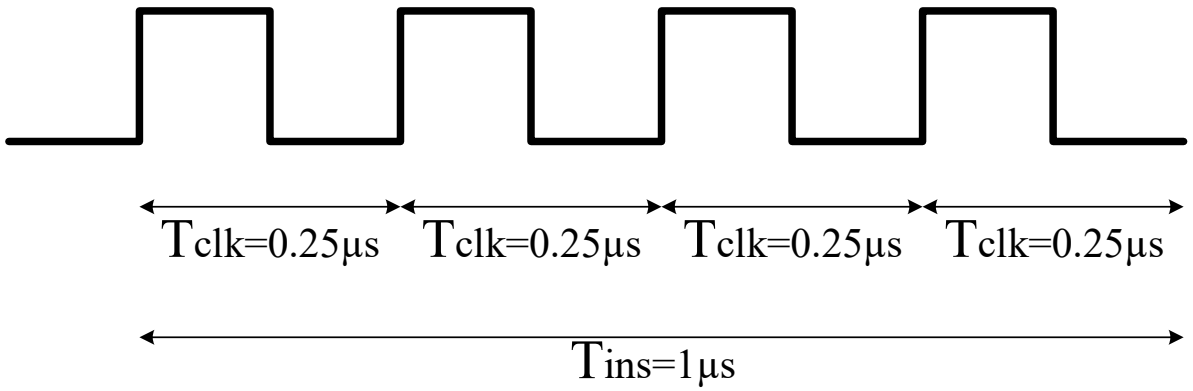
Speed of Instruction Execution

Execution speed calculation.

For example, if the crystal oscillator used is 4Mhz (This is Fosc or Fclk)

- Tclk = 1/4000000 = 0.25µs
- Tins = 4xTclk = 1µs (time to execute one instruction)
- Fins = Fosc/4 = 1Mhz (Speed of execution of instructions, i.e., 1000,000 instructions/s)
  (or 1 Million Instructions per Second, i.e. 1MIPS)

- Million Instructions per second is a measure of the execution speed of the computer.
- The measure approximately provides the number of machine instructions that could be executed in a second by a computer.

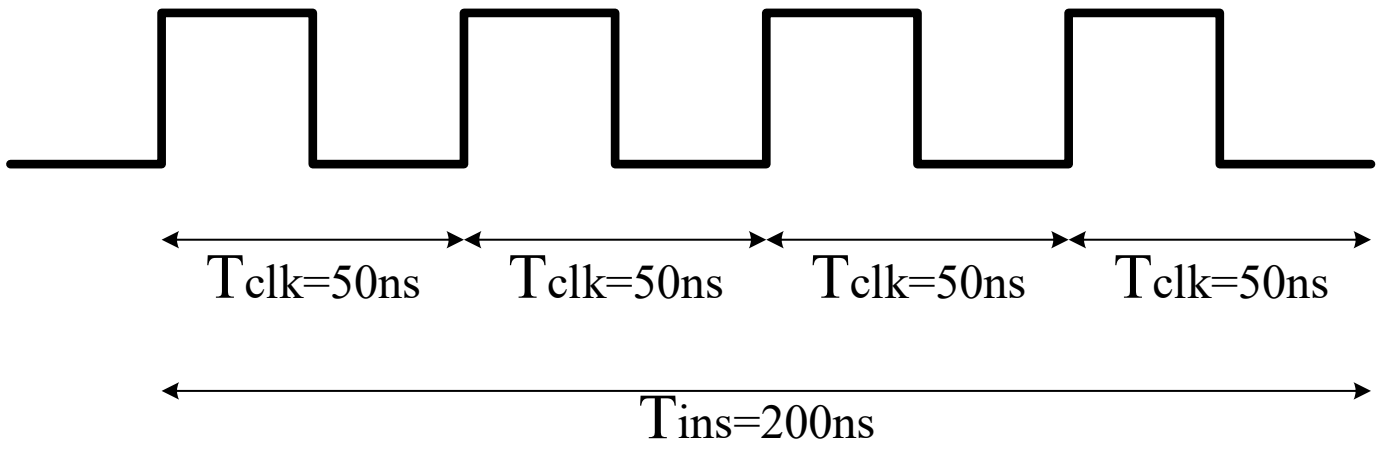


Speed of Instruction Execution

Execution speed calculation.

The PIC16F887 can run at a maximum clocking of 20Mhz. Thus,

- $T_{clk} = 1/20,000,0000 = 0.05\mu s = 50ns$ (nano is 10^{-9})
- $T_{ins} = 4xT_{clk} = 200ns$ (time to execute one instruction)
- $F_{ins} = F_{osc}/4 = 5Mhz$ (Speed of execution of instructions, i.e., 5000,000 instructions/s)
  (or 5 Million Instructions per Second, i.e. 5MIPS)



Speed of Instruction Execution

Also, from the datasheet, it can be seen from the table that some instructions need 2 cycle to be fully executed. i.e., double time of regular instructions

These instructions are the jump instructions (branching) like CALL, GOTO, RETURN, RETFIE, RETLW

Or the instructions that are conditional branching like INCFSZ, DECFSZ, BTFSS, BTFSC) only in case of branching

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS						
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff	C, DC, Z	1, 2
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff	Z	1, 2
CLRF	f	Clear f	1	00 0001 lfff ffff	Z	2
CLRW	–	Clear W	1	00 0001 0xxx xxxx	Z	
COMF	f, d	Complement f	1	00 1001 dfff ffff	Z	1, 2
DECF	f, d	Decrement f	1	00 0011 dfff ffff	Z	1, 2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00 1011 dfff ffff		1, 2, 3
INCF	f, d	Increment f	1	00 1010 dfff ffff	Z	1, 2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00 1111 dfff ffff		1, 2, 3
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff	Z	1, 2
MOVF	f, d	Move f	1	00 1000 dfff ffff	Z	1, 2
MOVWF	f	Move W to f	1	00 0000 lfff ffff		
NOP	–	No Operation	1	00 0000 0xx0 0000		
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff	C	1, 2
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff	C	1, 2
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff	C, DC, Z	1, 2
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff		1, 2
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff	Z	1, 2
BIT-ORIENTED FILE REGISTER OPERATIONS						
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff		1, 2
BSF	f, b	Bit Set f	1	01 01bb bfff ffff		1, 2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff		3
LITERAL AND CONTROL OPERATIONS						
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk	C, DC, Z	
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk	Z	
CALL	k	Call Subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	–	Clear Watchdog Timer	1	00 0000 0110 0100	$\overline{TO}$, $\overline{PD}$	
GOTO	k	Go to address	2	10 1kkk kkkk kkkk		
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z	
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	–	Return from interrupt	2	00 0000 0000 1001		
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	–	Return from Subroutine	2	00 0000 0000 1000		
SLEEP	–	Go into Standby mode	1	00 0000 0110 0011	$\overline{TO}$, $\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk	C, DC, Z	
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z	

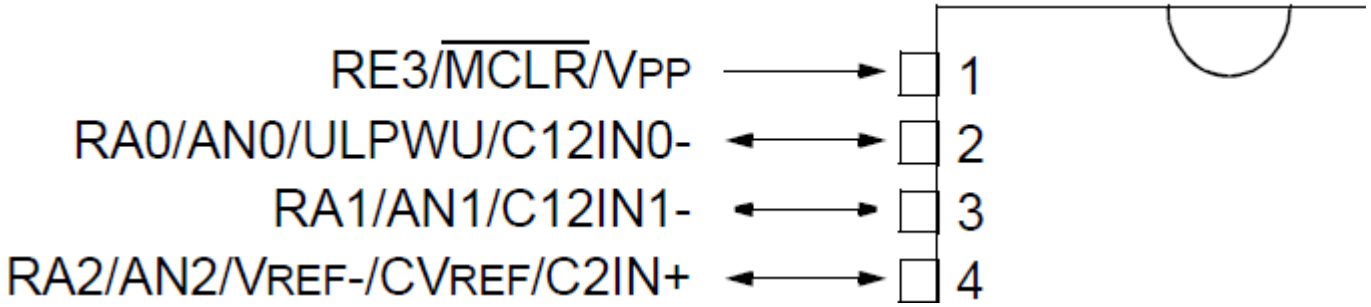
Chip reset

- All microcontrollers have a reset pin used to reset the chip and restart the code from the beginning. This is mainly used in case of trouble in execution due to a mistake in coding or due to environmental noise around the PIC (electric or electromagnetic noise).

In PIC16F887, the reset pin in pin #1

The PIN is called Master CLeaR and is active low, i.e., it will reset the PIC if a logic 0 is applied.

Thus, to allow PIC to work, a logic 1 level should be applied.





Chip reset

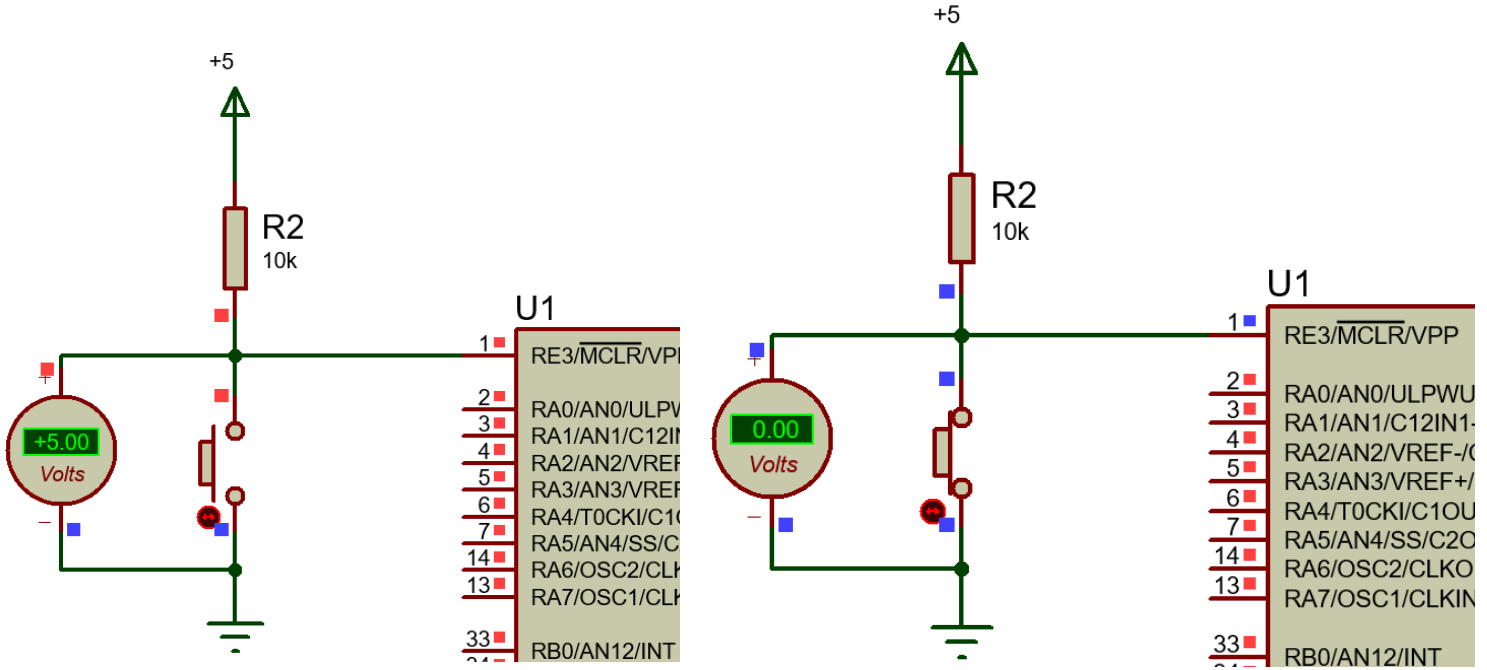
- From the previous activity, the MCLR is disactivated in the CONFIGURATION bits and the PIN is used as RE3 digital input.

```
CONFIG  MCLRE = OFF           ; RE3/MCLR pin function select bit (RE3/MCLR pin function is digital input, MCLR internally tied to VDD)
```

- In case we want to use this PIN, we should turn on this bit and connect a special circuit to reset the chip when required.

The main reset circuit is made of a push button and a pull-up resistor.
The role of the pull-up resistor is to pull the logic level of the PIN to 5 volt. (the PIC will be executing the code normally)

When the push button is pressed, the PIN will be grounded (0 logic) and the PIC is reset.

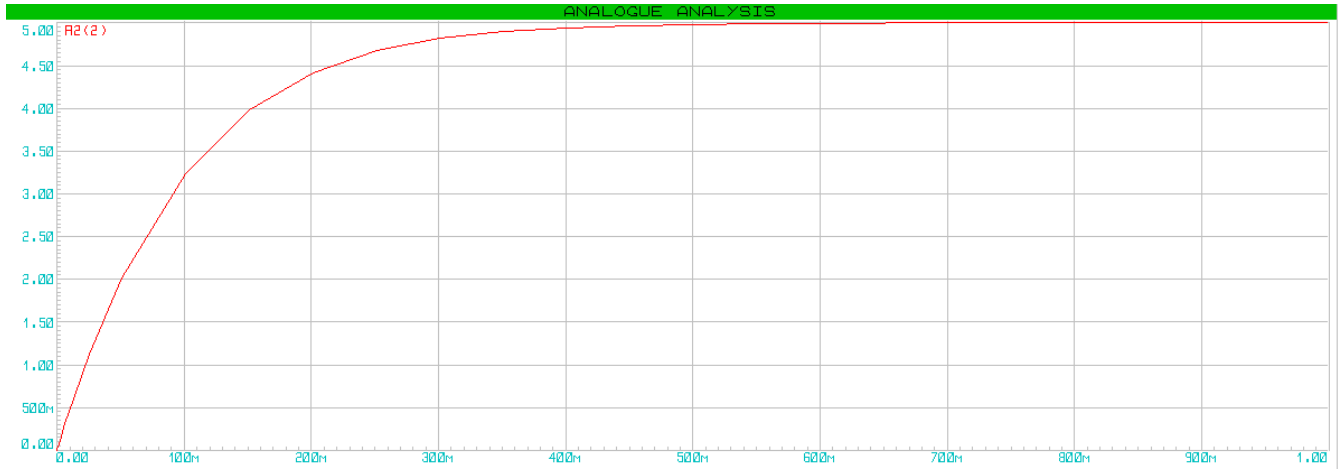


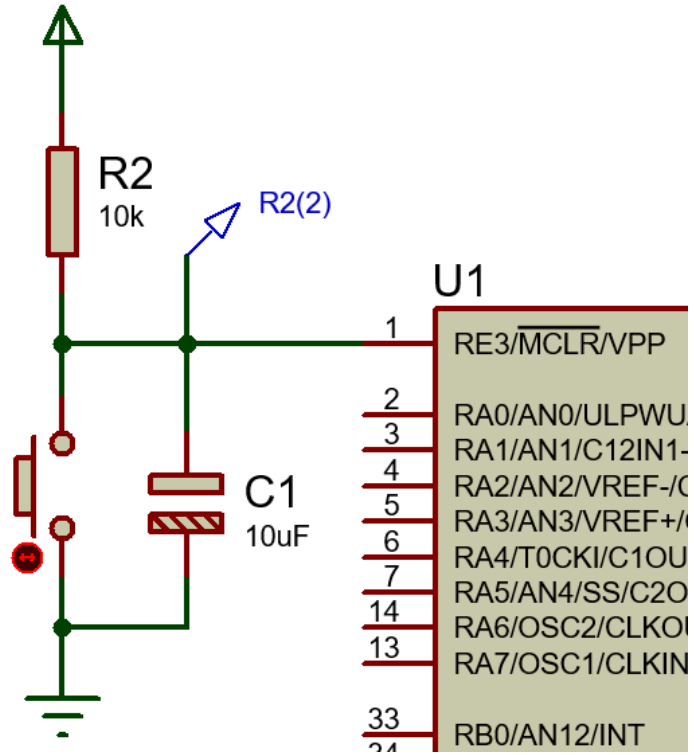
Chip reset

Sometimes, at startup, the PIC generate unexpected results as the code is not running correctly. This is due to a non-stable oscillation of the crystal oscillator at power-up.

The manufacturer recommend adding a capacitor across the push button to make a slower starting up of the chip. The voltage across the capacitor increases exponentially from 0 to 5V

The required time to reach 5 volt is $5RC = 5\tau$, unit in second



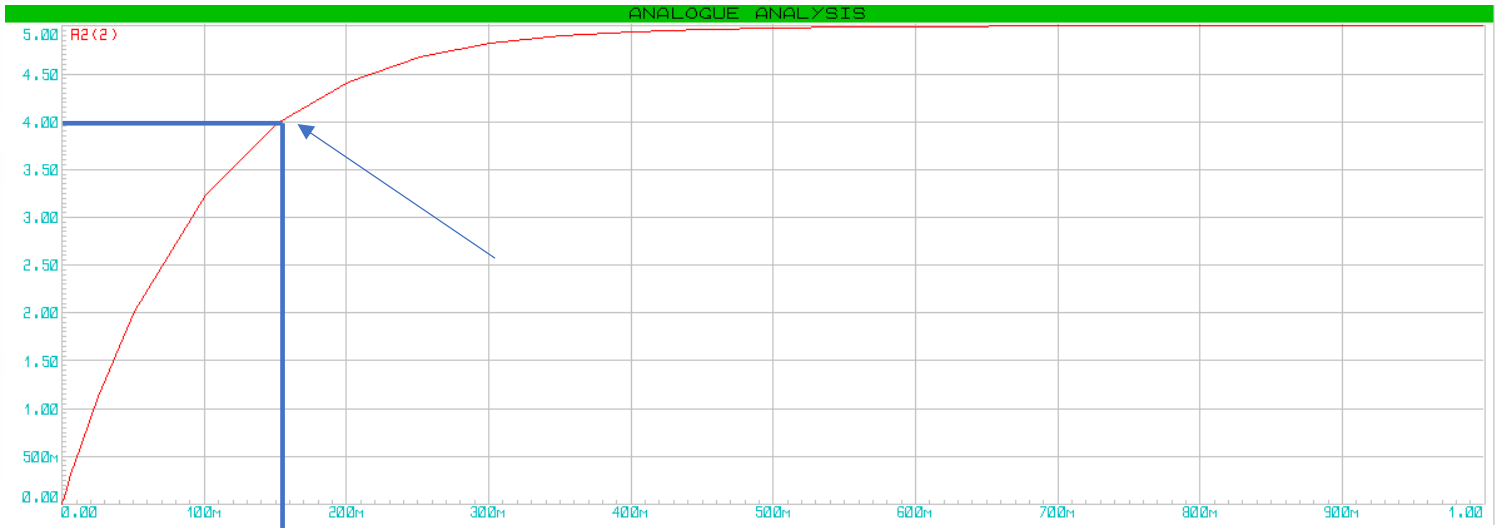


Chip reset

From the datasheet, the MCLR pin is considered HIGH at 0.8VDD which is 0.8x5 = 4V.

The voltage occurs after 150ms from power up. This time is enough to get accurate oscillation from the crystal.

Thus a resistor of 10k and a capacitor of 10uF are good choices to slow down the PIC startup by 150ms.



	VIH	Input High Voltage	
		I/O ports:	
D040		with TTL buffer	2.0
D040A			0.25 VDD + 0.8
D041		with Schmitt Trigger buffer	0.8 VDD
D042		MCLR	0.8 VDD
D043		OSC1 (XT and LP modes)	1.6
D043A		OSC1 (HS mode)	0.7 VDD
D043B		OSC1 (RC mode)	0.9 VDD



Test your knowledge

1. How many bits are allocated for the destination bit
2. How many address lines are required to access a 512 byte RAM
3. What should be RP0 and RP1 to access bank 2
4. PIC resets when a logic level 1 is applied on /MCLR (T/F)
5. A PIC is connected to a 16Mhz crystal, what is the instruction time ?
6. If the instruction "MOVLW" takes 0.2µs to be executed, what would be the oscillator frequency
7. If the instruction "GOTO" takes 0.4µs to be executed, what would be the oscillator frequency
8. How many internal oscillator the PIC16F887 has and what are the frequencies that can be used.
9. Why Microchip recommend to connect a capacitor to the reset circuit
10. What is the default frequency of the internal oscillator of the PIC
11. What is the name of the register used to select between the internal oscillator frequencies
12. What should be TRIS to configure the PORT as input
13. What is the instruction time if the used quartz is 4 kHz
14. What should be assigned to TRISB to have: RB0 = input , RB4 = output.



Week 3

Lecture

Lecture 6 / LO2

Writing on PORTS
Activity 2 Preparation



Assembly code structure (template code)

- Assembly code includes the following points with the same order :
  1. Code description
  2. Including the xc.inc library
  3. defining the configuration bits
  4. writing the interrupt code (to be discussed later)
  5. writing the main code
  6. writing the setup function
  7. write any other function



Assembly code structure (template code)

- **Code description**

At the beginning of each code, it is required to add come comments about your code task. This will help you understand the code objective for later revision.

```
1    ; Template code to be used as reference
2    ; This code will ........
3    ; Designed to work with pic-as compiler 2.32
4    ; PIC used is the PIC16f887
5    ; Written by ........
6
```

Assembly code structure (template code)

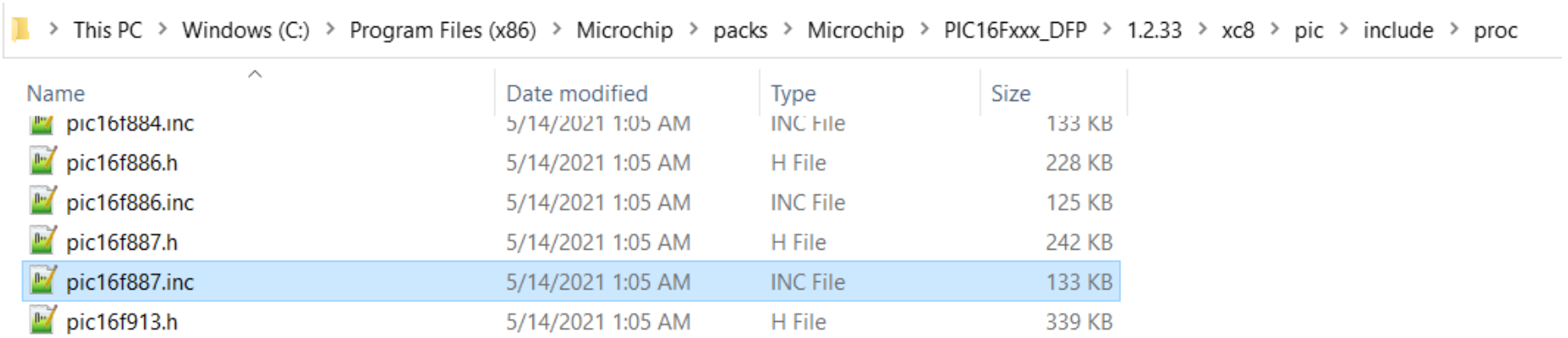
- **Including the PIC library**

Including the PIC register library is essential in assembly programming. The file is called xc.inc. The MPLAB look for the required library file of the chosen chip.

```
6
7       #include <xc.inc>
8
```

This file is actually located in C:\Program Files (x86)\Microchip\packs\Microchip\PIC16Fxxx_DFP\1.2.33\xc8\pic\include\proc

This file includes all the register definition and the configuration-bit values

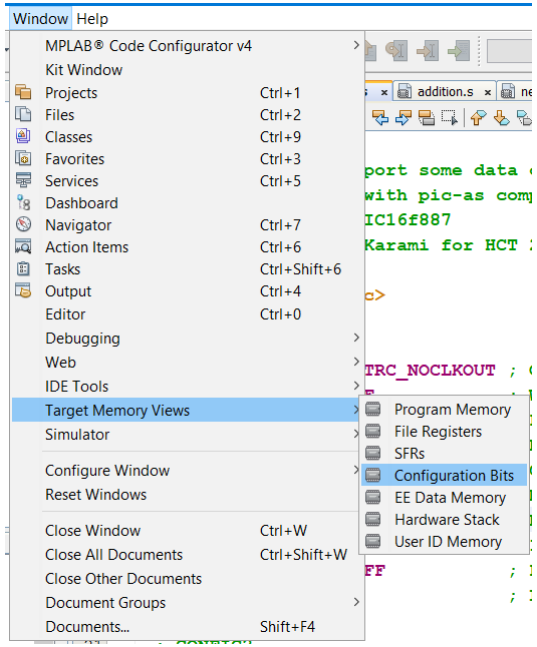


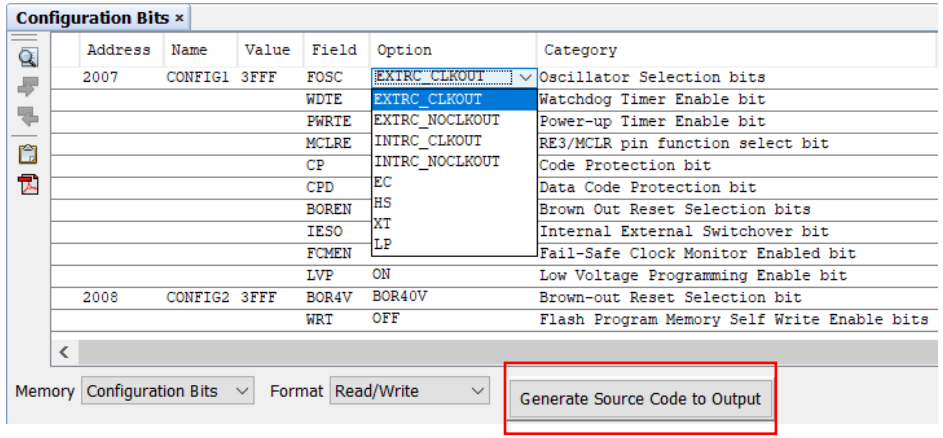
Assembly code structure (template code)

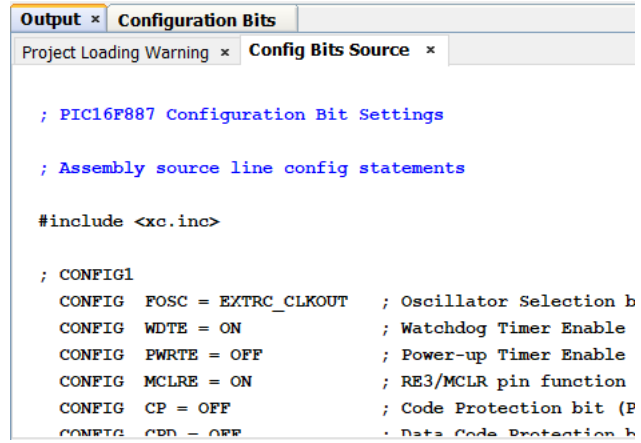
- **Configuration bits**

Configuration bits are the settings of the PIC that includes the way to programming it, the method of protecting the internal code, the method of clocking the chip and many other settings.

Configuration bits can be generated automatically by MPLABX as shown









Assembly code structure (template code)

- **Configuration bits**

Most common configurations bit are shown.

The internal oscillator is selected. No watch-dog timer used , MCLR pin is not used, code is not protected….

```
    #include <xc.inc>

; CONFIG1
  CONFIG  FOSC = INTRC_NOCLKOUT ; Oscillator Selection bits (INTOSCIO oscillator: I/O function on RA6/OSC2/CLKOUT pin, I/O function on RA7/
  CONFIG  WDTE = OFF            ; Watchdog Timer Enable bit (WDT disabled and can be enabled by SWDTEN bit of the WDTCON register)
  CONFIG  PWRTE = OFF           ; Power-up Timer Enable bit (PWRT disabled)
  CONFIG  MCLRE = OFF           ; RE3/MCLR pin function select bit (RE3/MCLR pin function is digital input, MCLR internally tied to VDD)
  CONFIG  CP = OFF              ; Code Protection bit (Program memory code protection is disabled)
  CONFIG  CPD = OFF             ; Data Code Protection bit (Data memory code protection is disabled)
  CONFIG  BOREN = OFF           ; Brown Out Reset Selection bits (BOR disabled)
  CONFIG  IESO = OFF            ; Internal External Switchover bit (Internal/External Switchover mode is disabled)
  CONFIG  FCMEN = OFF           ; Fail-Safe Clock Monitor Enabled bit (Fail-Safe Clock Monitor is disabled)
  CONFIG  LVP = OFF             ; Low Voltage Programming Enable bit (RB3 pin has digital I/O, HV on MCLR must be used for programming)

; CONFIG2
  CONFIG  BOR4V = BOR40V        ; Brown-out Reset Selection bit (Brown-out Reset set to 4.0V)
  CONFIG  WRT = OFF             ; Flash Program Memory Self Write Enable bits (Write protection off)
```



Assembly code structure (template code)

- **Main code**

The MAIN code is the place we write the main task of the code.

```
25      PSECT BLABLA,CLASS=CODE, DELTA=2,ABS ; this line is required in the new assembler pic-as
26
27          ORG     0           ;reset vector
28          GOTO    MAIN
29
30          ORG     0X4         ; interrupt vector
31          RETFIE
32
33
34      MAIN:
35          CALL    SETUP
36
37                                  ; write your code here
38
39          GOTO    $               ; do nothing
40
```

Assembly code structure (template code)

- **SETUP function**

The SETUP function is the one used to defines the modes of the PORTS and the internal oscillator frequency… and many more to be discussed later.

In this function, we usually jumps between banks to make the required setup.

For example, to configure the TRIS and OSCCON we must jump to BANK1

To configure the analog/digital pins (ANSEL, ANSELH) we must jump to BANK3.

Finally, we jump back to BANK0

```
41  SETUP:                  ; by default we are in BANK0
42      CLRF    PORTA       ; clear port at the beginning
43      CLRF    PORTB       ; clear port at the beginning
44      CLRF    PORTC       ; clear port at the beginning
45      CLRF    PORTD       ; clear port at the beginning
46      CLRF    PORTE       ; clear port at the beginning
47
48      BSF     RP0
49      BCF     RP1     ; access to BANK 1, RP1 = 0, RP0 = 1
50      MOVLW   01100000; configuring OSCCON register to select 4Mhz oscillator
51      MOVWF   OSCCON
52      MOVLW   10101010B
53      MOVWF   TRISA
54      ; ADD ALL OTHER CONFIGURATION RELATED TO BANK 1
55
56      BSF     RP0
57      BSF     RP1     ; access to BANK 2, RP1 = 1, RP0 = 0
58      ; ADD ALL OTHER CONFIGURATION RELATED TO BANK 2
59
60      BSF     RP0
61      BSF     RP1     ; access to BANK 3, RP1 = 1, RP0 = 1
62      CLRF    ANSEL   ; PORTA and PORTE are digital
63      CLRF    ANSELH  ; PORTB is digital
64      ; ADD ALL OTHER CONFIGURATION RELATED TO BANK 3
65
66      BCF     RP0     ; access to BANK 0, RP1 = 0, RP0 = 0
67      BCF     RP1
68      RETURN
```



Activity 2: Using PORTS to display data

- In this example, we will learn how to write values on the PORTS. We will be using assembly language to configure the PORTS as outputs and then write some data on.
- Also, we will be using Proteus software for the simulation.
- Proteus is a paid software able to simulate schematic for microcontrollers also design professional PCB (Printed circuit board) with 3D view.

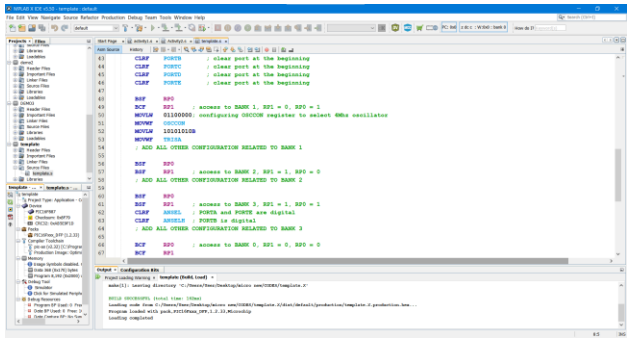
- This Activity will be implemented in the next Lab

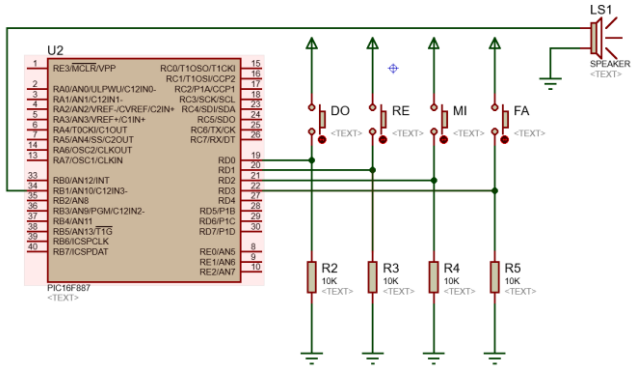
Thus the main task is to send 1111,1111 to all ports and check if they are all illegible to send 5 volts on the output.

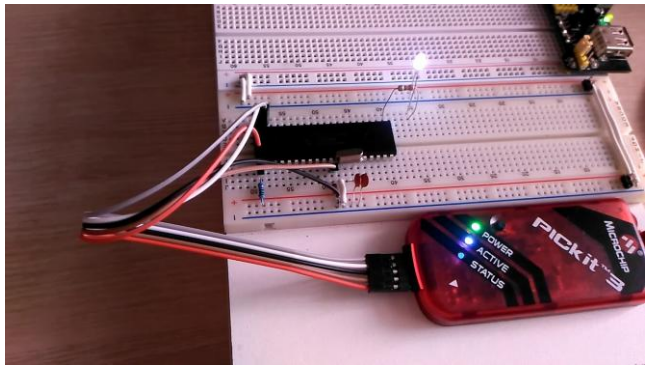
Activity 2: Using PORTS to display data

IMPORTANT NOTE

- From now on, you have to consider that designing a control circuit is achieved by
  1. Writing code (in class)
  2. Simulation (in class and/or lab)
  3. Implementation (in lab)









Activity 2: Using PORTS to display data

- **Step 1: Writing assembly code**

- Any code will start by some descriptive comment, include of the library file and the configuration bits.

- The major configuration bit are configure to:
  - Use internal oscillator
  - No watch dog timer
  - Power up timer is not used
  - Mclr pin is not used
  - No code protection required
  - No power fail detection required
  - Low voltage programming is off

```
; Activity 2
; This code will export some data on the PORTS
; Designed to work with pic-as compiler 2.32
; PIC used is the PIC16f887
; Written by Nabil Karami for HCT 2023

    #include <xc.inc>

; CONFIG1
  CONFIG  FOSC = INTRC_NOCLKOUT ; Oscillator Sel
  CONFIG  WDTE = OFF            ; Watchdog Timer
  CONFIG  PWRTE = OFF           ; Power-up Timer
  CONFIG  MCLRE = OFF           ; RE3/MCLR pin f
  CONFIG  CP = OFF              ; Code Protectio
  CONFIG  CPD = OFF             ; Data Code Prot
  CONFIG  BOREN = OFF           ; Brown Out Rese
  CONFIG  IESO = OFF            ; Internal Exter
  CONFIG  FCMEN = OFF           ; Fail-Safe Cloc
  CONFIG  LVP = OFF             ; Low Voltage Pr

; CONFIG2
  CONFIG  BOR4V = BOR40V        ; Brown-out Rese
  CONFIG  WRT = OFF             ; Flash Program
```

Activity 2: Using PORTS to display data

- **Step 1: Writing assembly code**

- The main code consists of sending 11111111 on the ports.
- This is achieved by placing the value on the W register first (line37) then move the W content to all ports.
- In case no other duties are required from the chip, it is required to freeze the code by the GOTO $ (line 44).
- GOTO $ means keep executing this line address

```
25  PSECT BLABLA,CLASS=CODE, DELTA=2,ABS ; this line is rec
26
27      ORG     0       ;reset vector
28      GOTO    MAIN
29
30      ORG     0X4     ; interrupt vector
31      RETFIE
32
33
34  MAIN:
35      CALL    SETUP
36
37      MOVLW   11111111B   ; put 11111111 on W register
38      MOVWF   PORTA       ; turn on all pins of the port
39      MOVWF   PORTB       ; turn on all pins of the port
40      MOVWF   PORTC       ; turn on all pins of the port
41      MOVWF   PORTD       ; turn on all pins of the port
42      MOVWF   PORTE       ; turn on all pins of the port
43
44      GOTO    $           ; do nothing
```

Activity 2: Using PORTS to display data

- **Step 1: Writing assembly code**

- The SETUP function is used to configure some important registers.
- First we clear all ports (lines 47 to 51). Preventive step to remove all junk from the RAM.
- OSCCON configures the 4Mhz internal crystal (since we are using the internal oscillator in the CONFIG)
- All TRIS are cleared to make PORTS output.
- ANSEL and ANSELH are cleared to make PORTA, E and B as digital pins

```
46  SETUP:                  ; by default we are in BANK0
47      CLRF    PORTA       ; clear port at the beginning
48      CLRF    PORTB       ; clear port at the beginning
49      CLRF    PORTC       ; clear port at the beginning
50      CLRF    PORTD       ; clear port at the beginning
51      CLRF    PORTE       ; clear port at the beginning
52
53      BSF     RP0
54      BCF     RP1         ; access to BANK 1, RP1 = 0, RP0 = 1
55      MOVLW   01100000B   ; configuring OSCCON
56      MOVWF   OSCCON      ;register to select 4Mhz oscillator
57      CLRF    TRISA   ; Make the port OUTPUT
58      CLRF    TRISB   ; Make the port OUTPUT
59      CLRF    TRISC   ; Make the port OUTPUT
60      CLRF    TRISD   ; Make the port OUTPUT
61      CLRF    TRISE   ; Make the port OUTPUT
62
63      BSF     RP0
64      BSF     RP1     ; access to BANK 3, RP1 = 1, RP0 = 1
65      CLRF    ANSEL   ; PORTA and PORTE are digital
66      CLRF    ANSELH  ; PORTB is digital
67
68      BCF     RP0     ; access to BANK 0, RP1 = 0, RP0 = 0
69      BCF     RP1
70      RETURN
```



Activity 2: Using PORTS to display data

- **Step 1: Writing assembly code**

- Now we need the hex file to be generated by MPLAB. This file will be later used to be uploaded on the Flash memory of the PIC using a programmer. Also it will be used for the Proteus simulation.
- Click on F11, or Shift F11 to build the code. The Hex file is generate successfully

```
Memory Summary:
    Program space        used    22h (    34) of  2000h words   (  0.4%)
    Data space           used     0h (     0) of   170h bytes   (  0.0%)
    EEPROM space         used     0h (     0) of   100h bytes   (  0.0%)
    Configuration bits   used     2h (     2) of     2h words   (100.0%)
    ID Location space    used     0h (     0) of     4h bytes   (  0.0%)

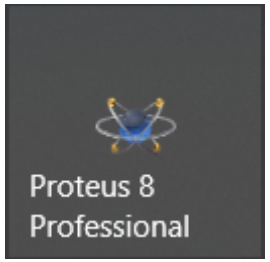
make[2]: Leaving directory 'C:/Users/User/Desktop/micro new/CODES/Activity2.X'
make[1]: Leaving directory 'C:/Users/User/Desktop/micro new/CODES/Activity2.X'

BUILD SUCCESSFUL (total time: 558ms)
```

Activity 2: Using PORTS to display data

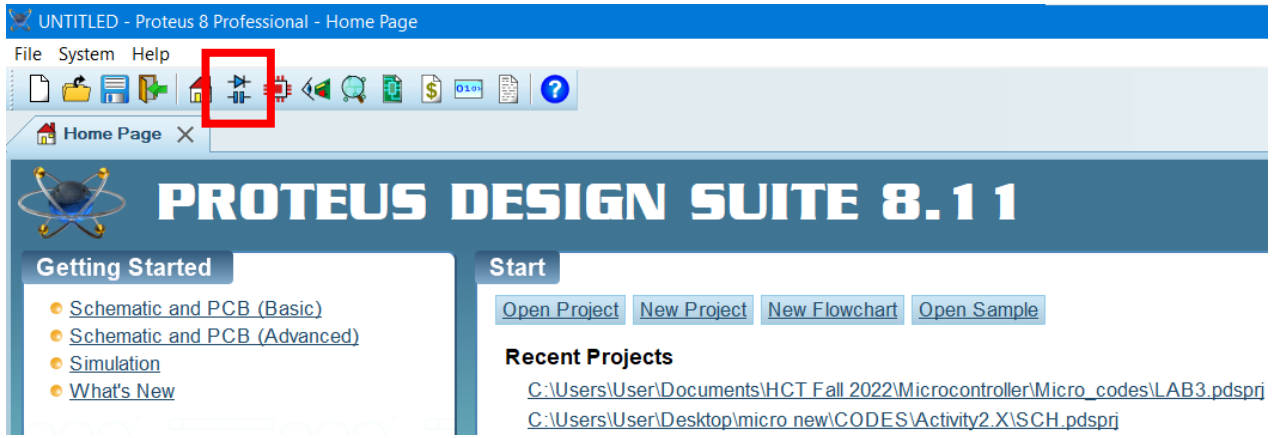
- **Step 2: Checking the Code (Simulation)**

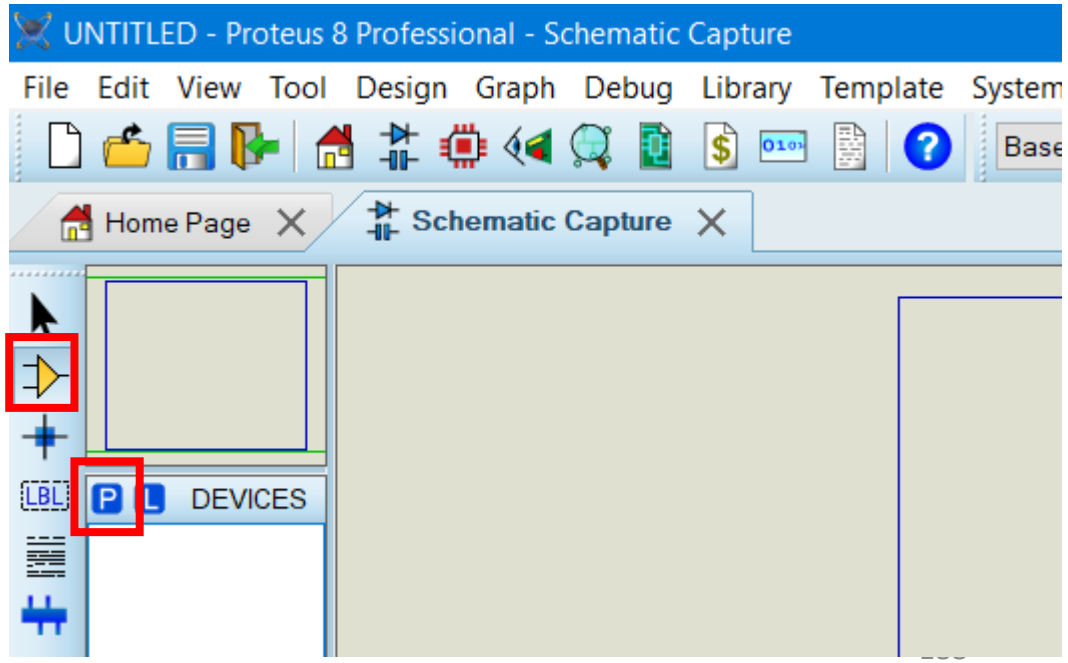
- After writing the code, and before making any hardware test, it is required to simulate the code and check if it is running as expected.
- We can simulate on MPLAB (as done in Activity 1) or we can use schematic based software to check the result.
- PROTEUS is one of the best software used to simulate microcontroller and processor also any analog/digital circuit



Activity 2: Using PORTS to display data

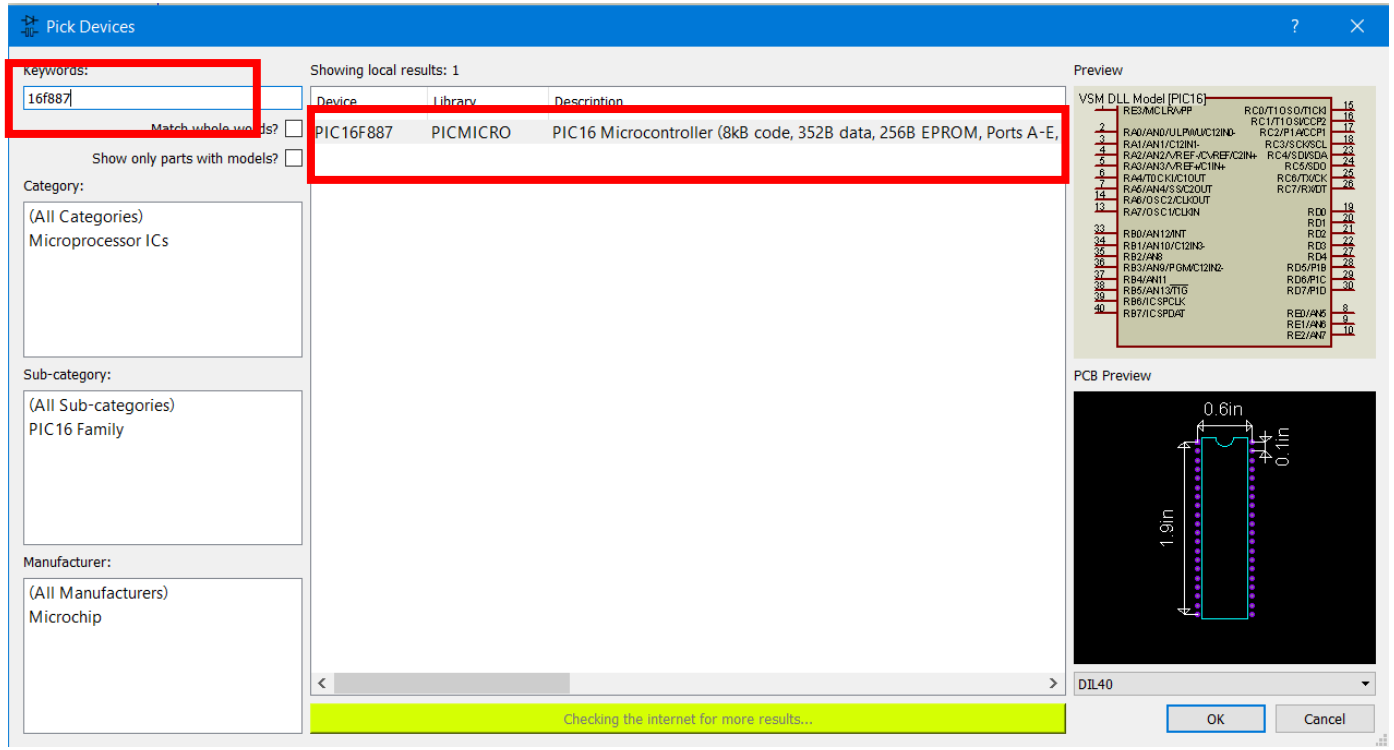
- **Step 2: Checking the Code (Simulation)**
- Open Proteus, and click on schematic capture icon to create a new schematic. Then click on the component icon and select P to Pick a device.





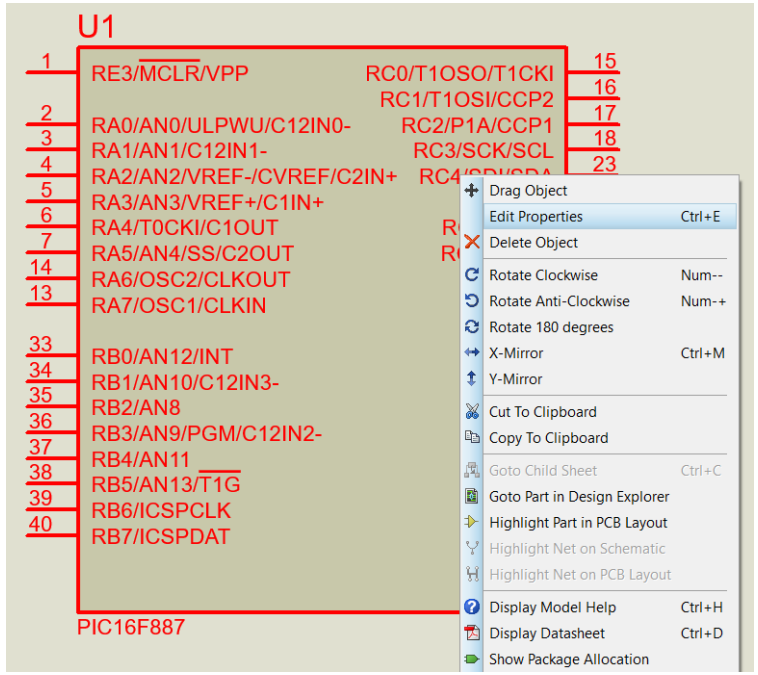
Activity 2: Using PORTS to display data

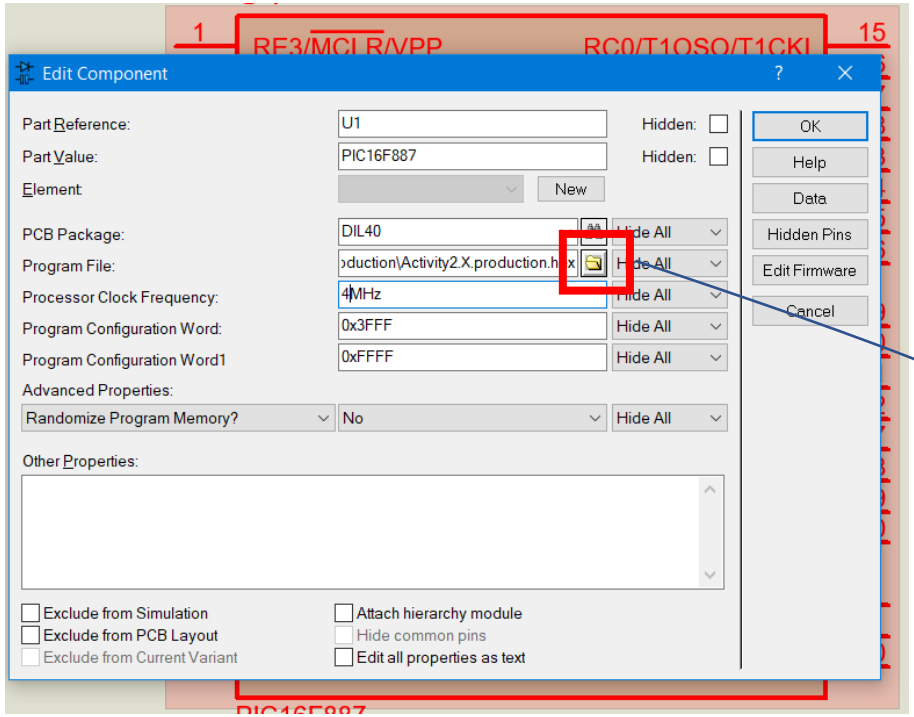
- **Step 2: Checking the Code (Simulation)**
- Write PIC16F887 in keywords and search for it in the library. Double click on the found result to add this component to the list. No other device will be needed for the moment.



Activity 2: Using PORTS to display data

- **Step 2: Checking the Code (Simulation)**
- Place the PIC in the workspace area, right click and edit the properties. Select the Hex file generated by MPLAB (available in the same project folder) and select clock frequency 4Mhz.





..\..\..\..\Activity2.X\dist\default\production\Activity2.X.production.hex

Lecture

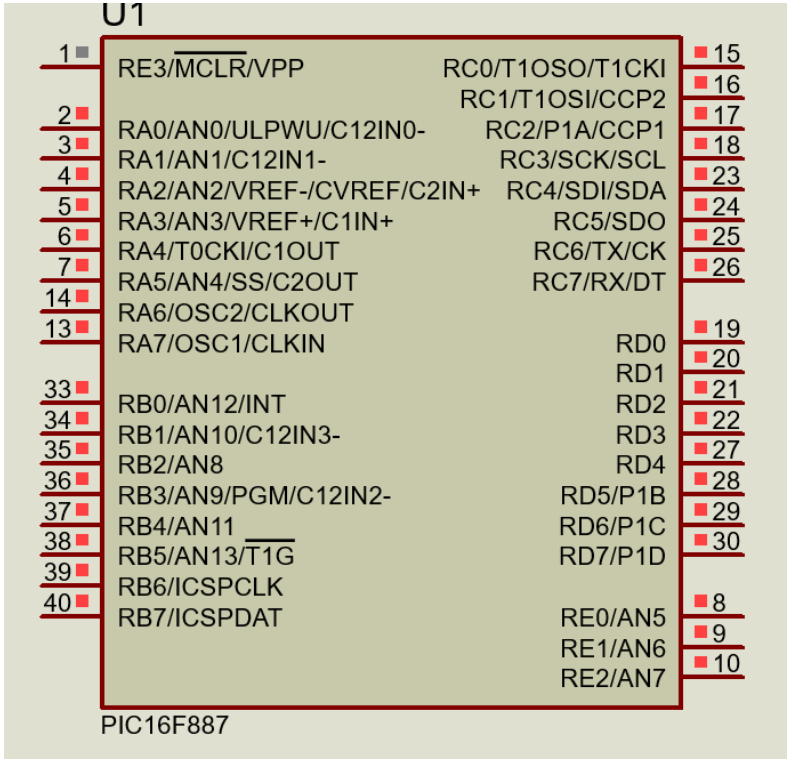
Activity 2: Using PORTS to display data

- **Step 2: Checking the Code (Simulation)**
- Click on the PLAY button and check the results



- Observation: The pins are labeled in RED. Means that 5 volt are generated by the pins (i.e., 1 logic).
- All pins are ON unless RE3.
- This PIN, as per the datasheet, is only INPUT pin

- RE3/MCLR/VPP is a special pin that can be used for 3 purposes.
  - Input pin RE3
  - Master CLeaR pin (Will be discussed later)
  - VPP pin for PIC programmer interfacing.



Read only pin

REGISTER 3-13: PORTE: PORTE REGISTER

U-0	U-0	U-0	U-0	R-x	R/W-x	R/W-x	R/W-x
—	—	—	—	RE3	RE2	RE1	RE0
bit 7							bit 0

Activity 2: Using PORTS to display data

- **Step 3: Hardware implementation**
- In order to see the logic levels on the PIC pins, the easiest way is to connect a LED on the pins. We will design and implement only one LED circuit. (no need to connect 35 LEDs).

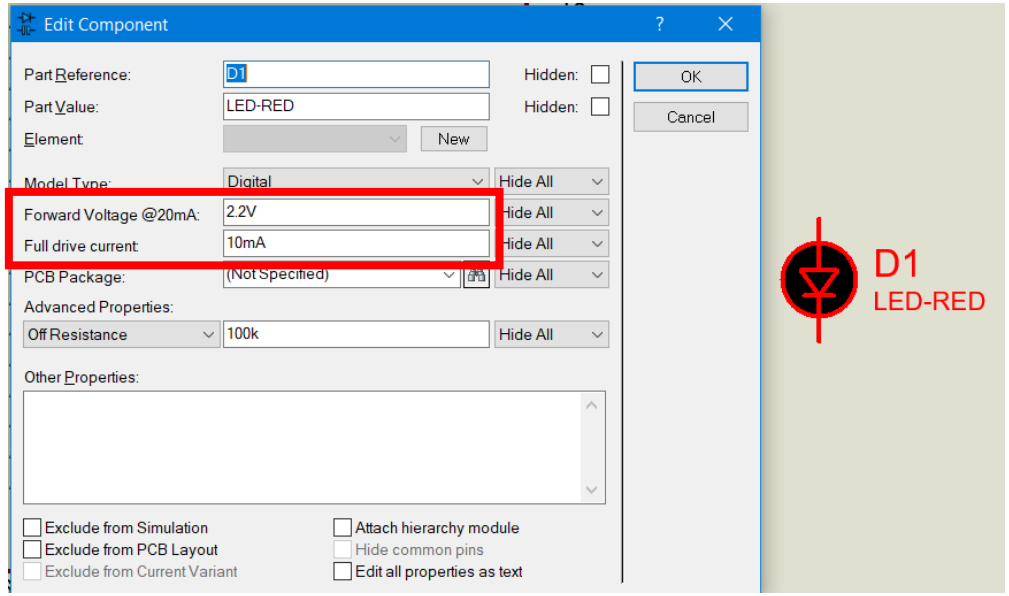
- As mentioned before, 1 logic means 5 volts, i.e., it is considered as a voltage source (battery) of 5 volts.



The LED is similar to a regular diode with a forward voltage drop of **2.2V** (not 0.6 volt as regular diode).

This means the LED cannot be connected to the source directly, otherwise it will be burned.

NOTE: forward voltage changes with type of LED and color.



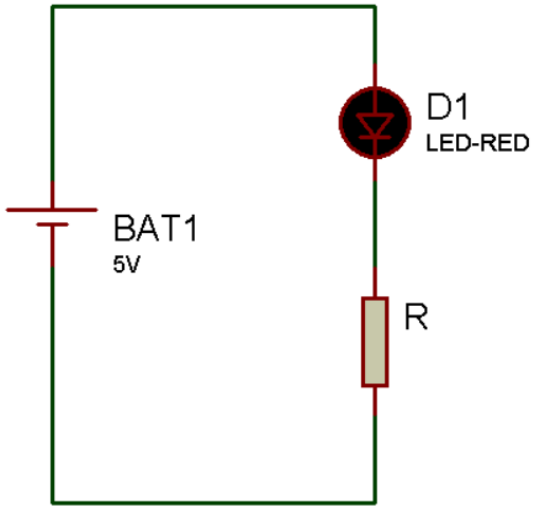
Activity 2: Using PORTS to display data

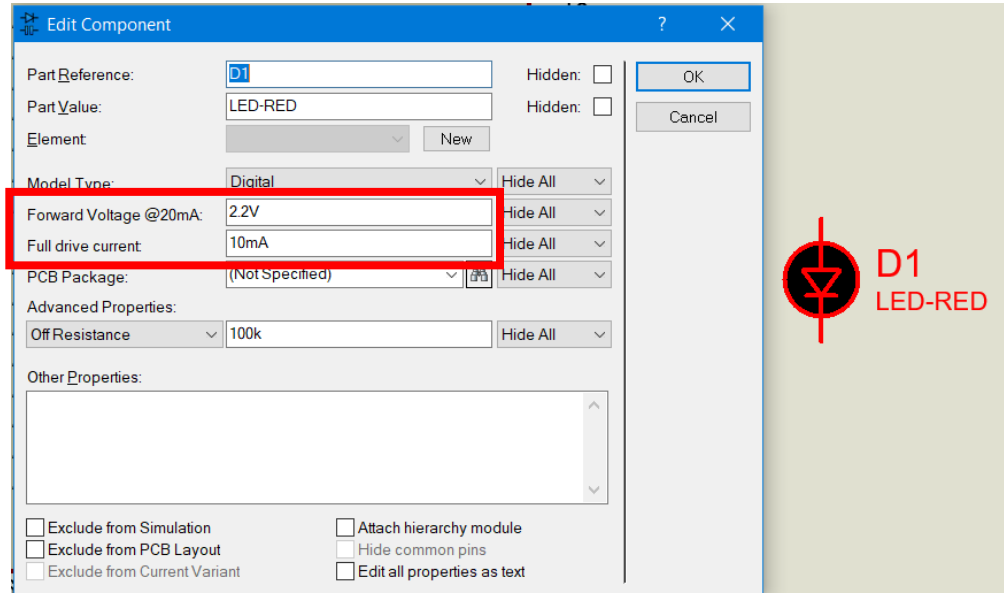
- **Step 3: Hardware implementation**
- A red led needs 2.2V to generate the optimum brightness. At this voltage, the current flowing will be 10mA. (Important note: the max current a pin can deliver is 20mA)
- Thus we need to connect a resistor in series with the source and the LED to drop the voltage to 2.2 volt.

Using KVL, the voltage across resistor
$V_R = V_{in} - V_{led} = 5 - 2.2 = 2.8V$

The Resistor current is same as led current (series connection), so $I_R = 10mA$

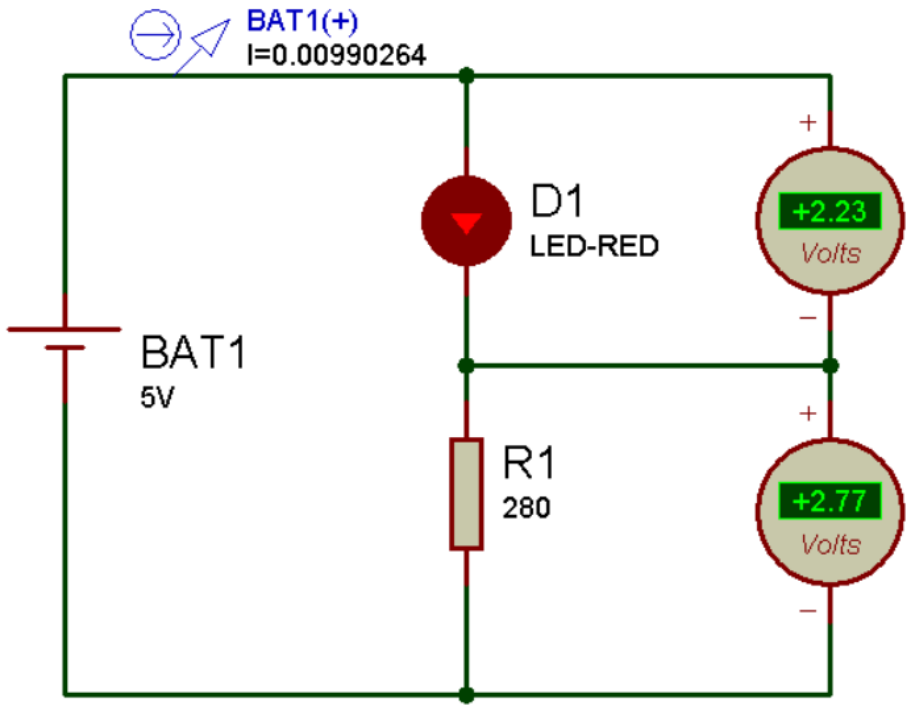
Using ohm's law, the resistor R is given as:
$R = V_R / I_R = 2.8/10mA = 280\ \Omega$

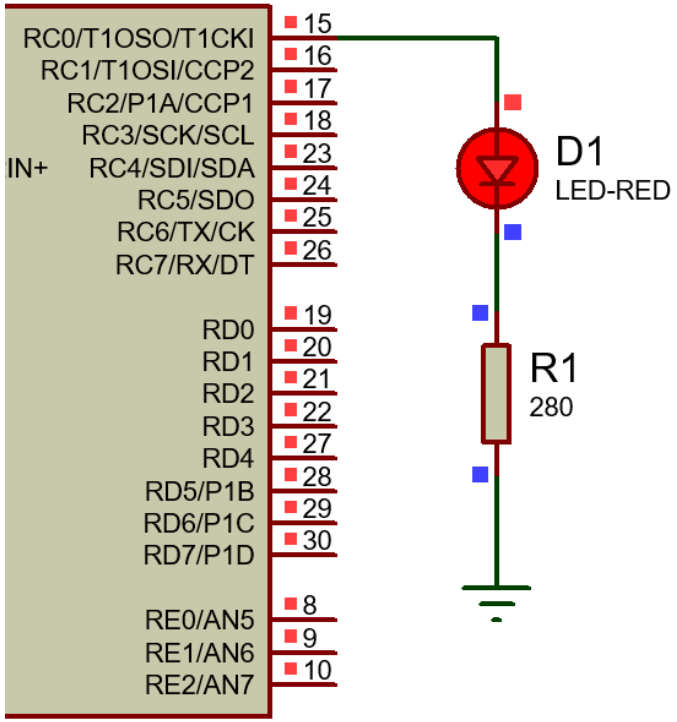


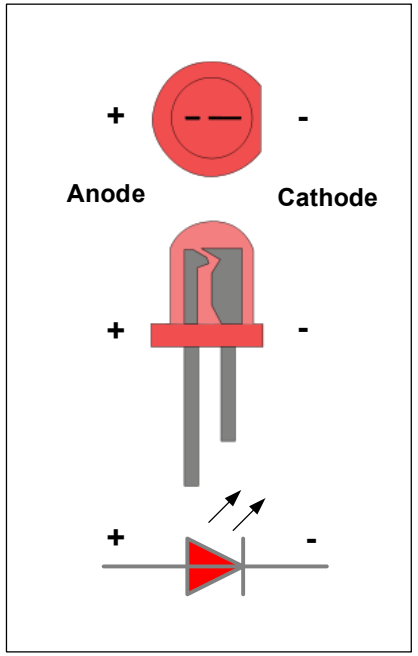


Activity 2: Using PORTS to display data

- **Step 3: Hardware implementation**
- The simulation result of the circuit is showing the same result as the one calculated.
- Thus, applying the same circuit on the PIC output will turn on the led.

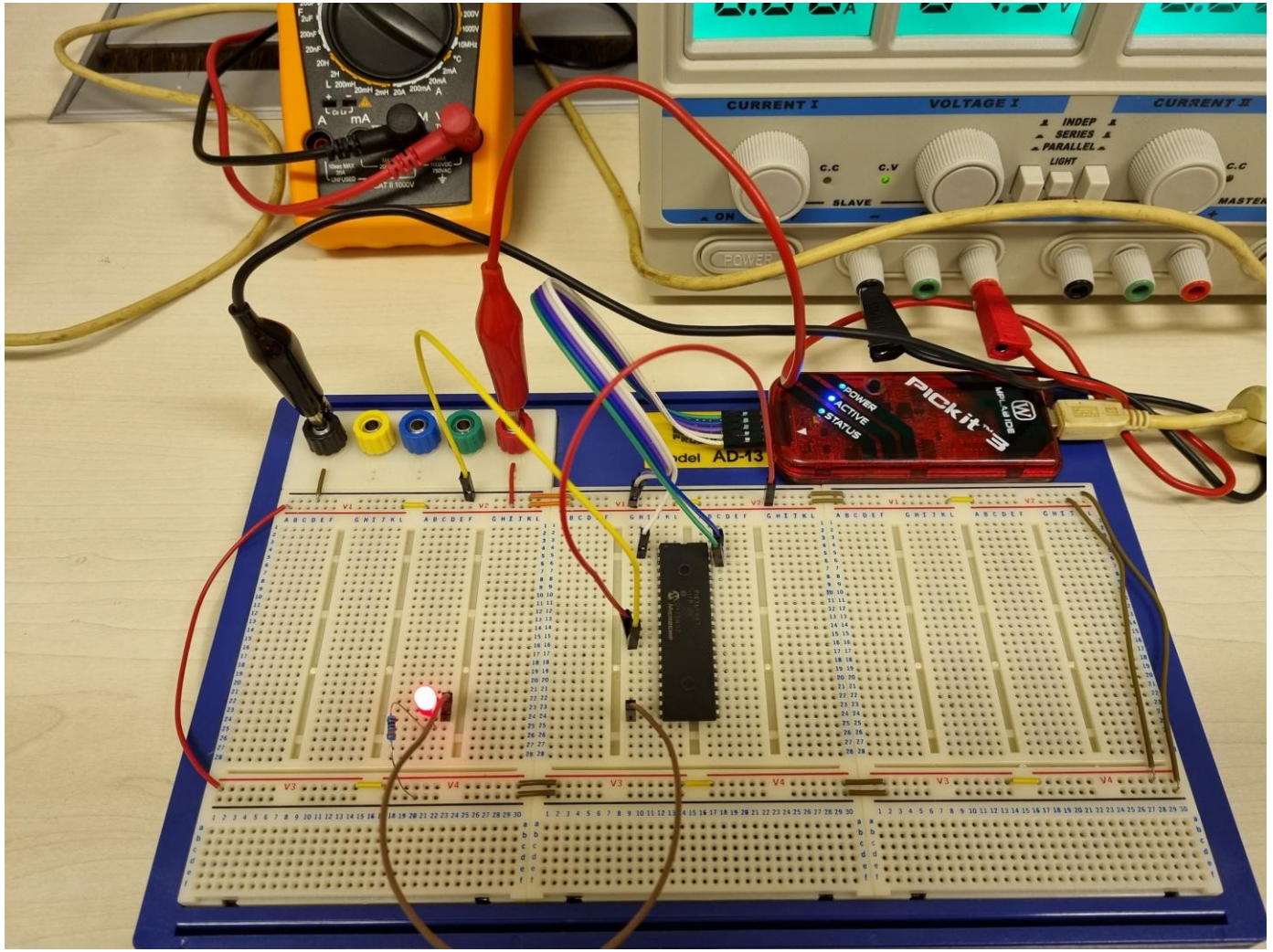






Activity 2: Using PORTS to display data

- **Step 3: Hardware implementation**
- The PIC is placed on a breadboard, supplied by an external power source of 5 volts at pins 11 (VCC, positive supply pin) and 12 (VSS, ground supply pin).
- The programmer used is the PIC KIT 3 connected to MCLR, VCC, VSS, RB6 (clock data line), and RB7 (data line).





Test your knowledge

1. Calculate the series resistor that should be connected to an ultra bright led with 3V, 30 mA connected to the pin RC0. Is it allowed to connect such led to the PIC ? Why ?
2. Which configuration is better; connect the resistor before the led or after ?
3. State three different ways to identify the positive and negative pins of a LED
4. How many ports in PIC16F887 can be used as output
5. How many ports in PIC16F887 can be used as input
6. Open the datasheet and check the role of every bit used in the configuration bits; in CONFIG1 and CONFIG2
7. CONFIG words are located in RAM (T/F)



Week 3

Lab 2 / LO2

Hardware implementation of Activity 2



Implementation of Activity 2

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. PIC programmer
  4. Breadboard wires (male-male)

- Required components:
  1. PIC 16F887
  2. Led x1
  3. Resistor from 200 to 470 ohm x 1



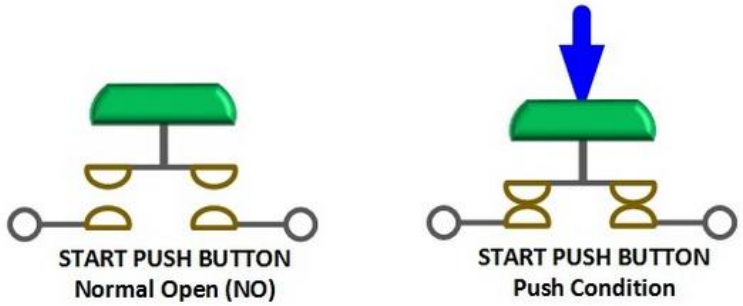
Week 4

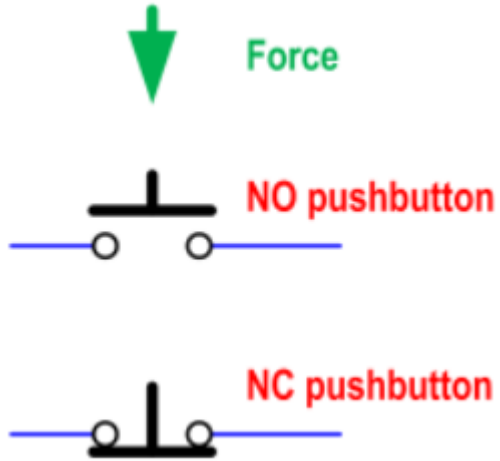
Lecture 7 / LO2

Push Button interfacing
Reading inputs

Push button interface with PIC

- A Push button switches rely on a simple in-out actuation mechanism.
- They can be employed to break (off) or initiate (on) a circuit.

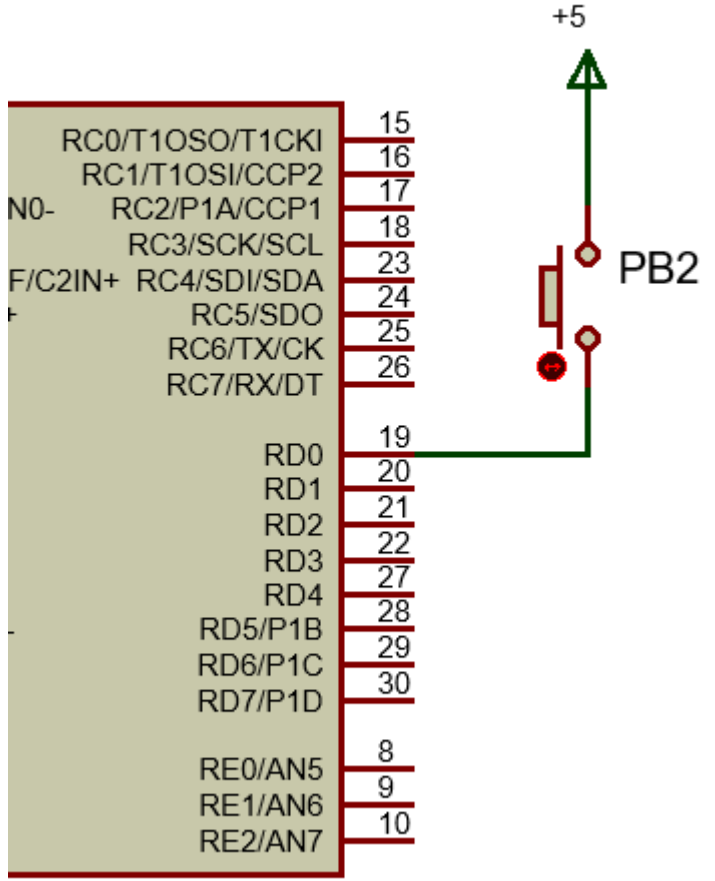




- Push button switches can be classified as being
  - Normally Open (NO) or
  - Normally Closed (NC).
- Normally Open ("OFF" position) switches complete the circuit when actuated,
- while Normally Closed ("ON" position) switches break the circuit when actuated.

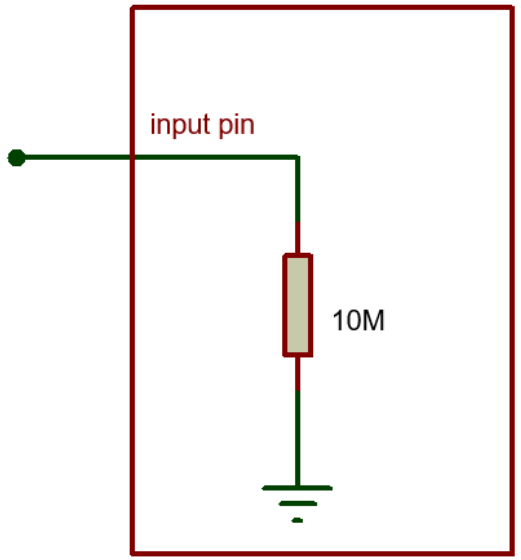
Push button interface with PIC

- PB is used to send command to the PIC to run certain process
- It is a digital device, thus the PIC read it as logic 0 or 1.
- As per the figure, it is clear that when the PB is pressed, a logic 1 is applied on the pin. But what if it is not! What will be the logic level ! 0 or 1 logic (0V or 5V) ?!
- In fact, when the push button is open, the RD0 is not connected to anything.
- In electronics, the input pin has a high input “impedance”, i.e., as if a very high resistor is internally connected to ground

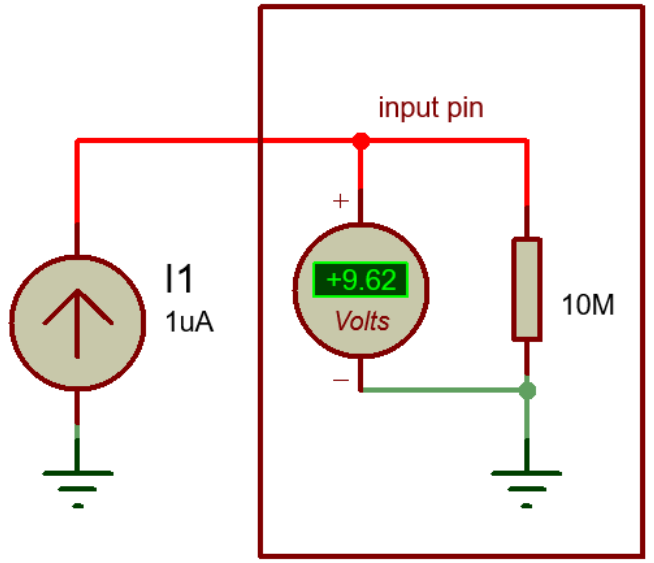


Push button interface with PIC

- This means that if a noise surrounding the wire produces only 1μA, a voltage V = IR will be generated across the pin.
- For example, if R = 10M, the produced V = $10x10^{-6} * 1x10^{6}$ = 10V. This voltage represent a logic 1 as if the PB is pressed.



- Thus it is not allowed to keep the input pin "FLOATING"
- We have to pull the logic level up or down using an external resistor, depending on the circuit.

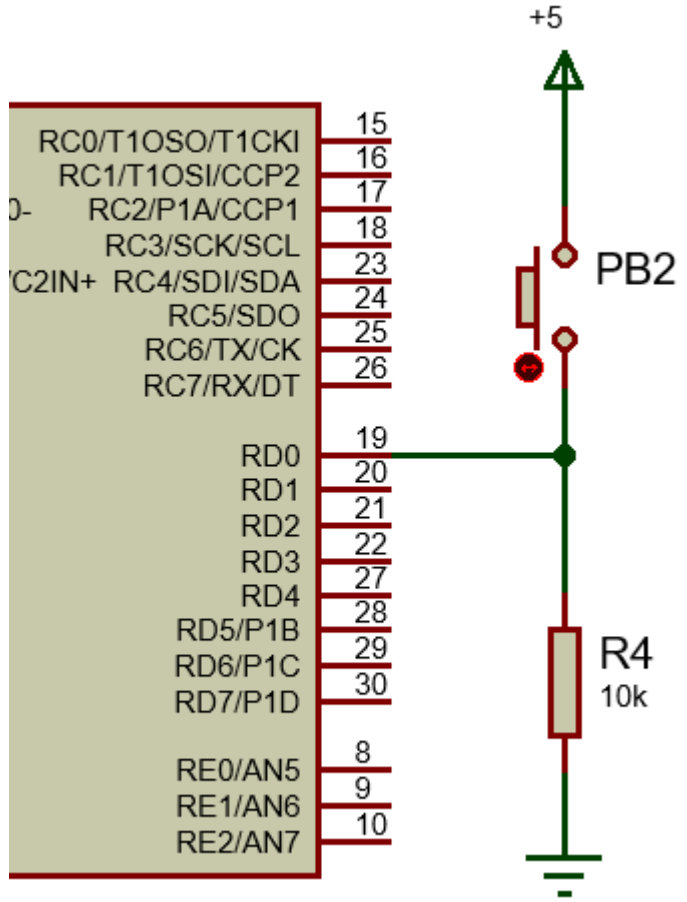


Push button interface with PIC

- Back to our circuit, a pull-down resistor here is used to pull the logic level to zero in case the push button is not pressed.

- What is the best value of R4 ?!
- The Role of R4 is do dominate the effect of noisy current, so not to generate a logic level 1 (a voltage above 3 is considered 1 logic… depend on the chip in use).

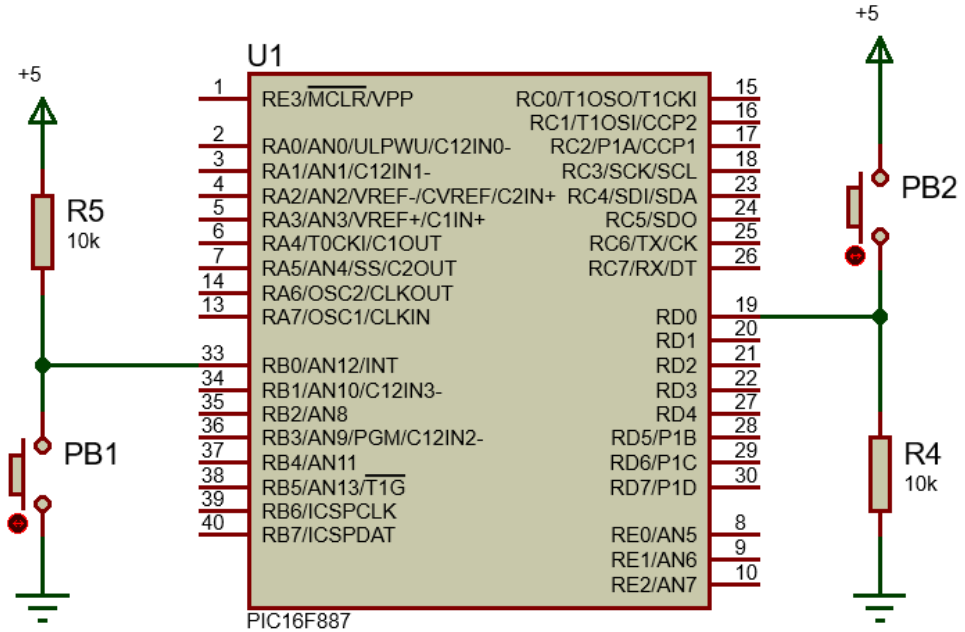
- On the other side the pull down should not be very low, otherwise a large current will flow in this resistor at every button press.
- Values between 1k to 47k are good choices



Push button interface with PIC

- Also, PB can be used to deliver zero logic when pressed (Case of PB1).
- This PB requires a pull-up resistor similar to the one used for PB2.

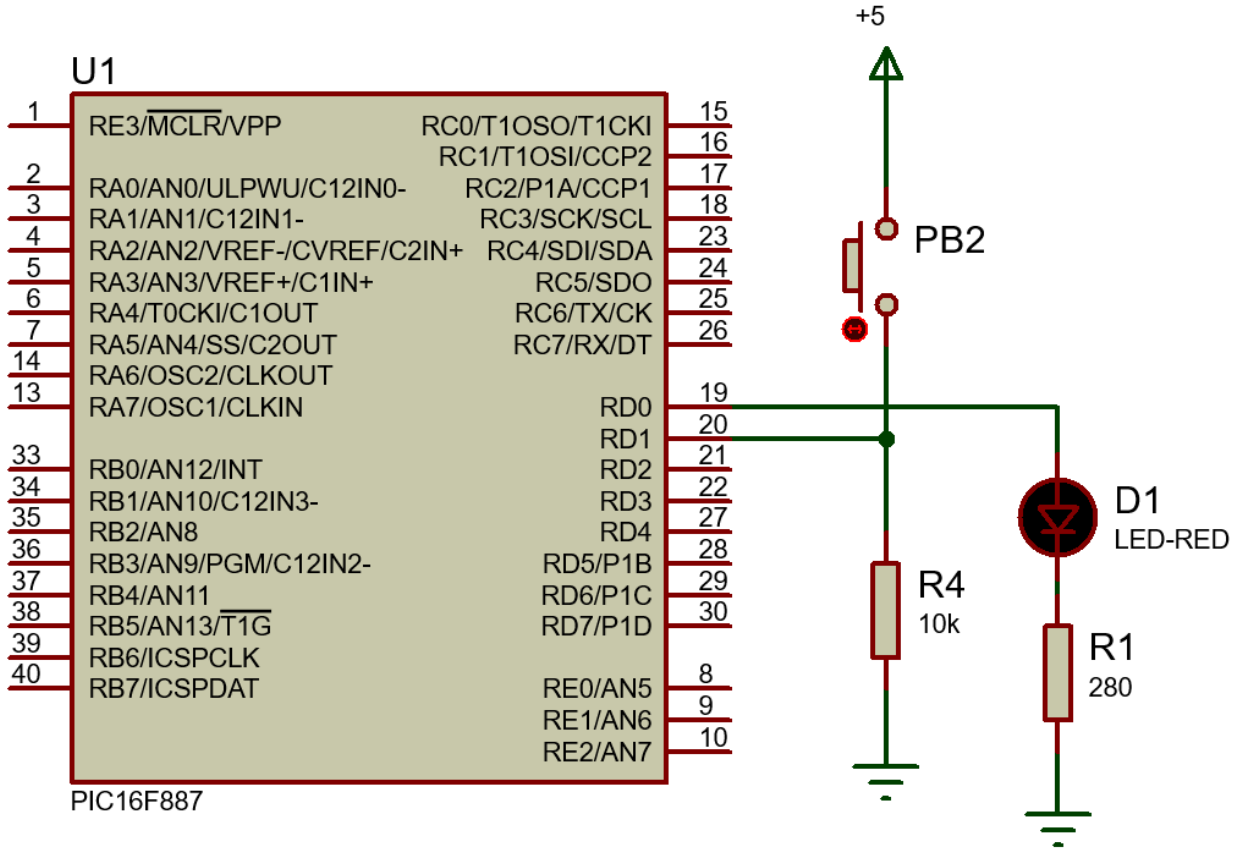
- As a summary,
  - PB1 is ACTIVE LOW and required a pull-up resistor
  - PB2 is ACTIVE HIGH and requires a pull-down resistor



Activity 3: Using Push buttons

- In this Activity, we will be using a NO Push Button connected to RD1.

- In this Activity, we will be using a push button to turn on a Led.
- The task is to test the Push button and turn ON the LED when pressed. Nothing else is required for the moment.



Activity 3: Using Push buttons

- When a push button is connected to a PIC, the pin should be configured as INPUT (TRIS = 1)
- The task of this Activity is to test the Push button and turn ON the LED when pressed.
- At line 37, the instruction GOTO $-1 is used. This is used to go up one line.
- In general, GOTO $±n is used to jump down(+) or up(-) for 'n' instructions

```
27      ORG     0       ;reset vector
28      GOTO    MAIN
29
30      ORG     0X4     ;interrupt vector
31      RETFIE
32
33  MAIN:
34      CALL    SETUP
35  REPEAT:
36      BTFSS   RD1         ; TEST IF PB IS PRESSED
37      GOTO    $-1         ; NO, RECHECK
38      BSF     RD0         ; YES, TURN ON LED
39      GOTO    REPEAT      ; REPEAT CODE
40
41  SETUP:                  ; by default we are in BANK0
42      CLRF    PORTD       ; clear port at the beginning
43      BSF     RP0         ; RP1 IS BY DEFAUT 0
44      MOVLW   01100000B   ; configuring OSCCON
45      MOVWF   OSCCON      ; register to select 4Mhz oscillator
46      MOVLW   00000010B
47      MOVWF   TRISD
48      BCF     RP0         ; access to BANK 0, RP0 = 0 , RP1 IS BY DEFAULT 0
49      RETURN
```

Activity 3: Using Push buttons

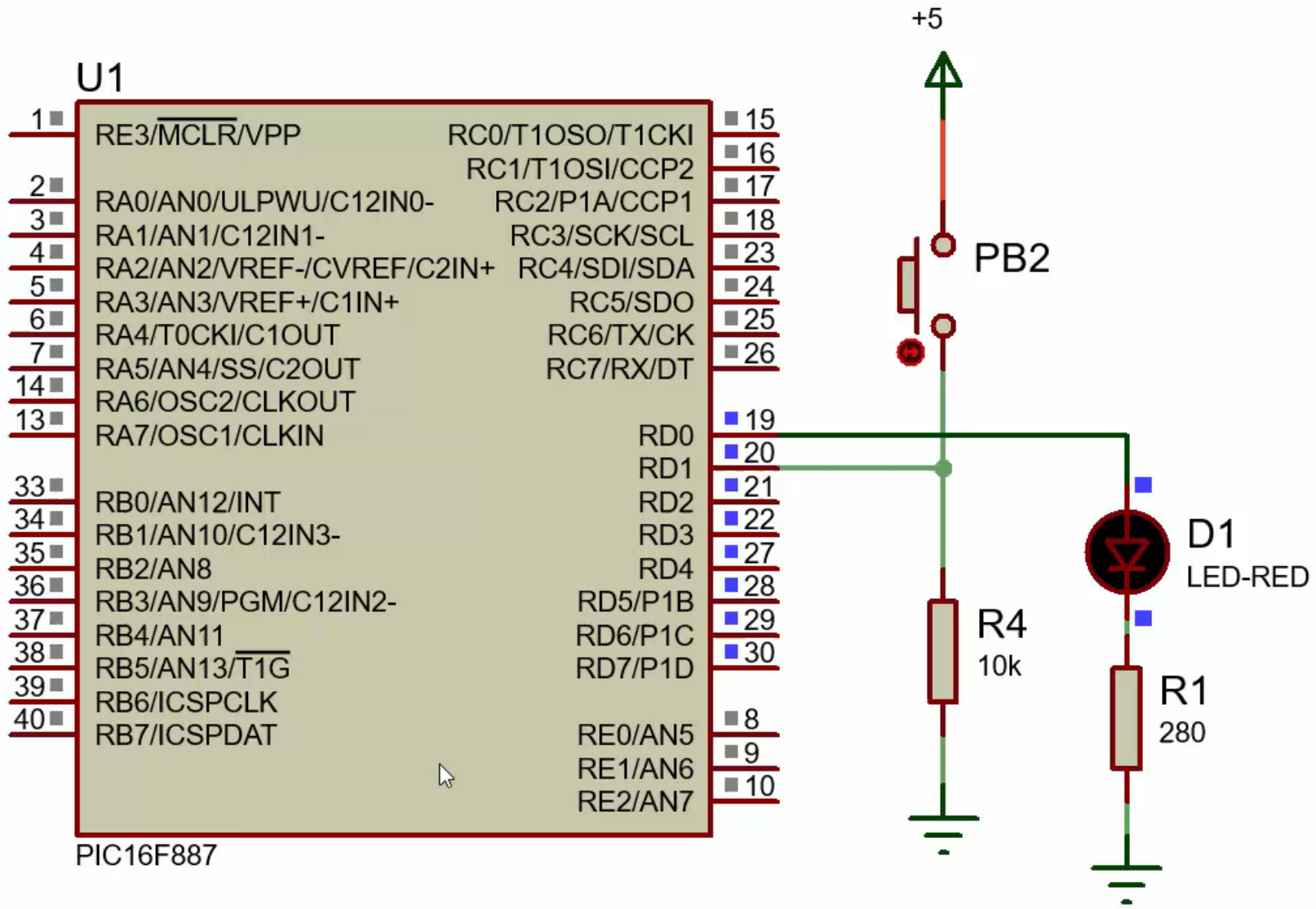
- To enhance our coding skills, the directive #define is used.
- This allows to replace the pin number by a significant name.
- In this example for example

"#define LED RD0" means that we can replace RD0 by the word LED as shown in line 42.

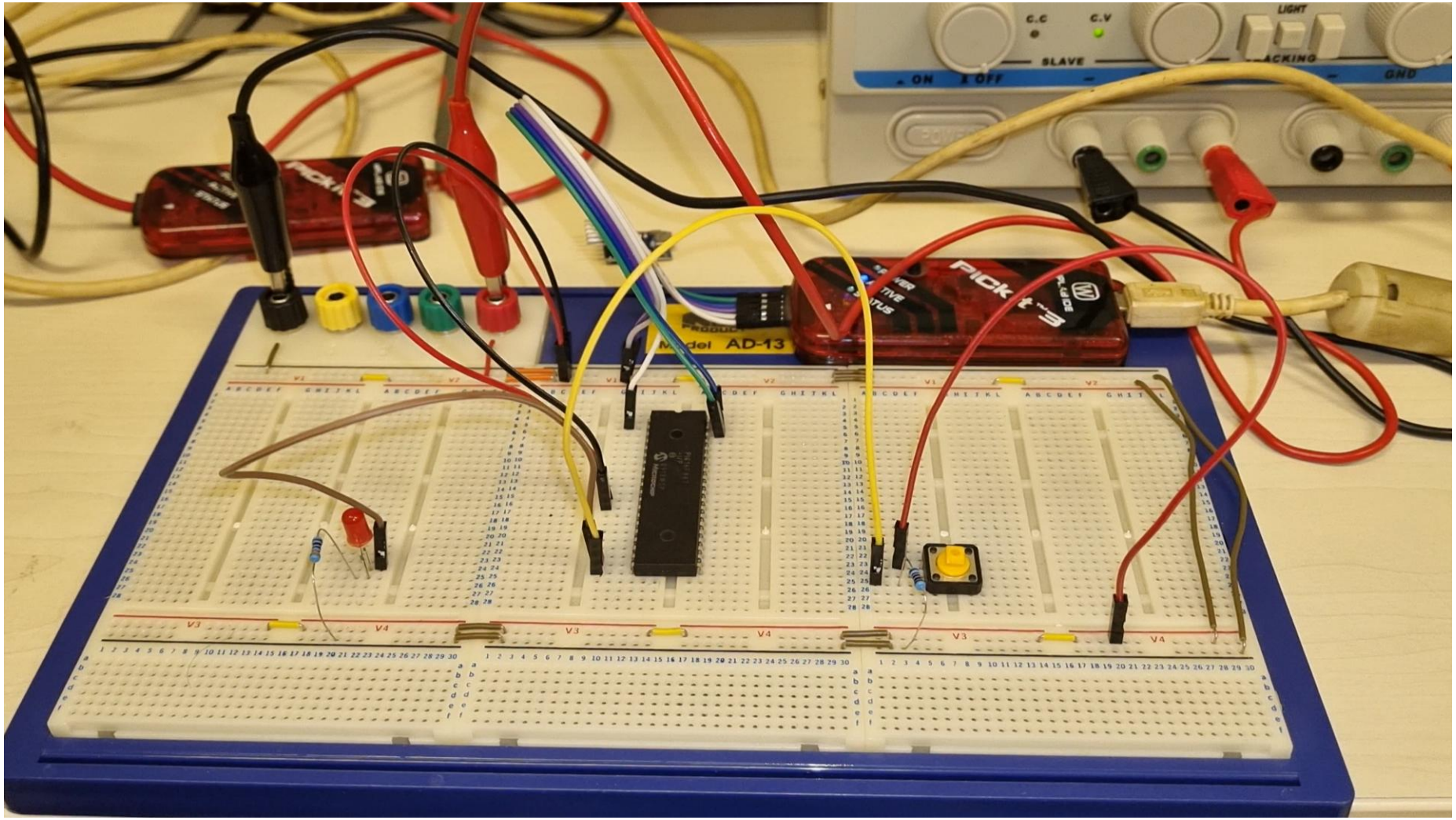
And at line 40, we are testing the PB which is the same as testing RD1.

```
27      #define LED RD0
28      #define PB  RD1
29
30        ORG     0         ;reset vector
31        GOTO    MAIN
32
33        ORG     0X4       ;interrupt vector
34        RETFIE
35
36
37    MAIN:
38        CALL    SETUP
39    REPEAT:
40        BTFSS   PB              ; TEST IF PB IS PRESSED
41        GOTO    $-1             ; NO, RECHECK
42        BSF     LED             ; YES, TURN ON LED
43        GOTO    REPEAT          ; REPEAT CODE
```

Activity 3: Using Push buttons



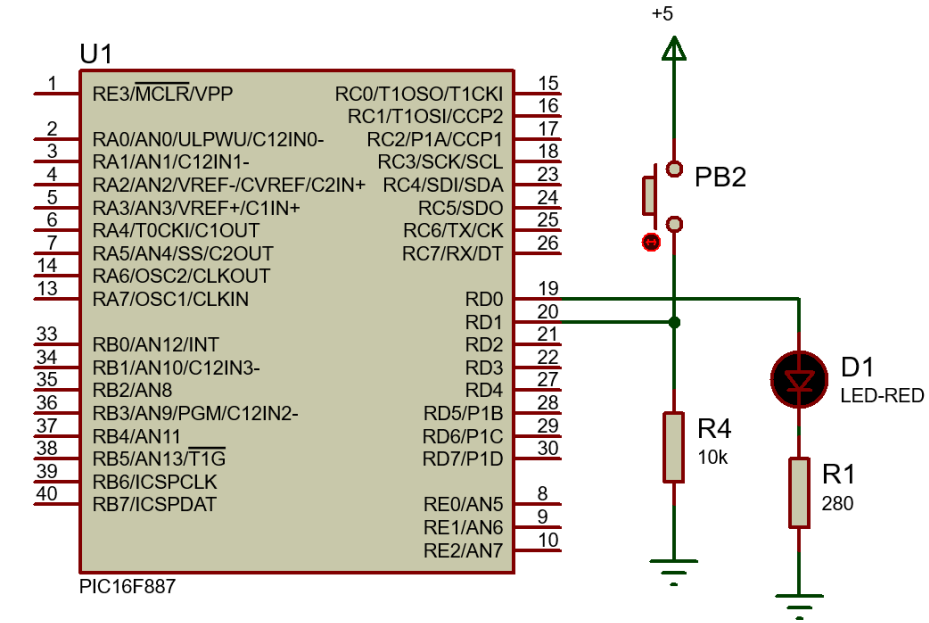
Activity 3: Using Push buttons



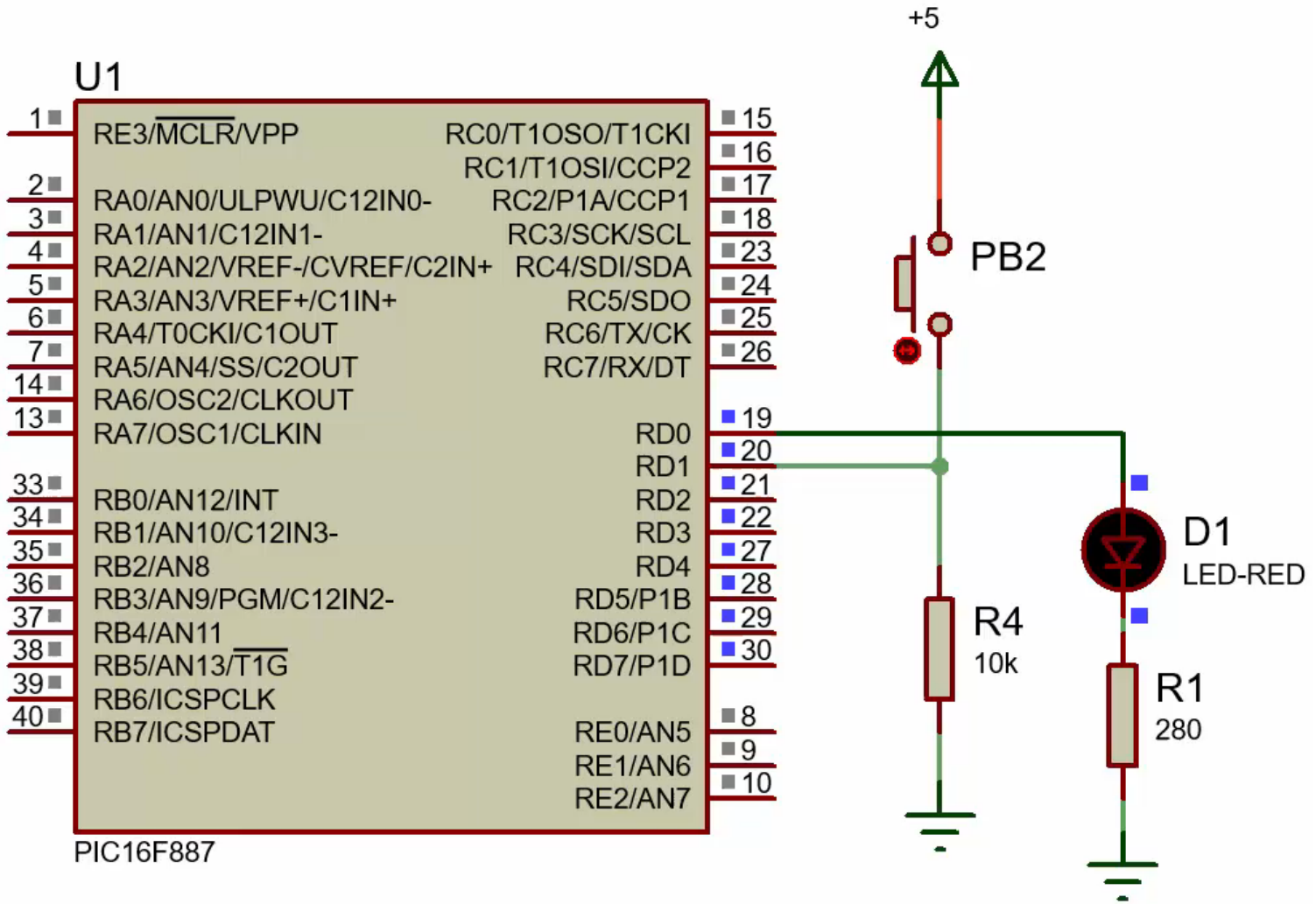
Activity 4: Using Push buttons

- In this Activity, we will modify the code to turn on the LED ON when the PB is pressed, and OFF when released.
- Only the main code is modified as shown.

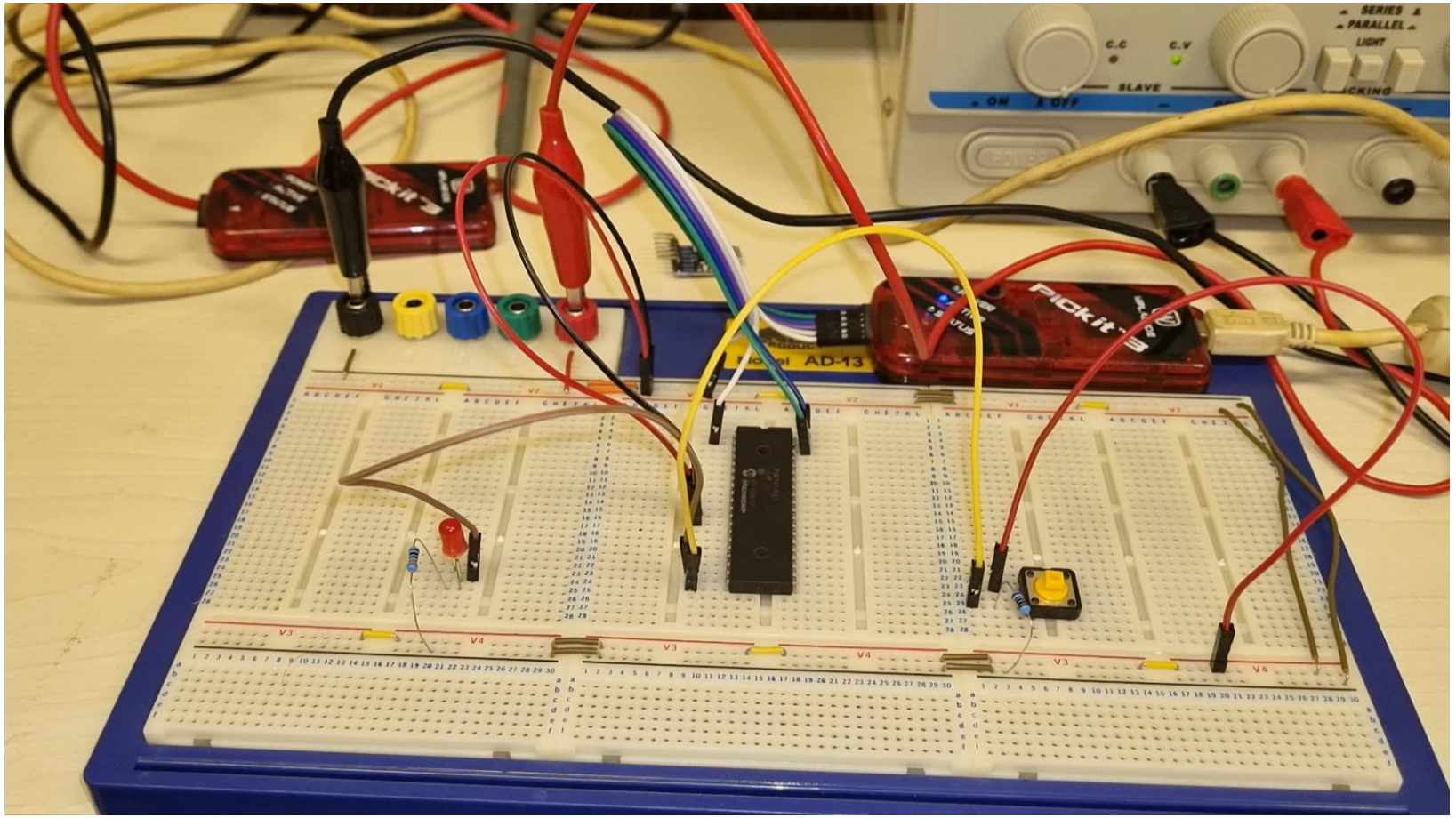
```
38      MAIN:
39          CALL    SETUP
40      REPEAT:
41          BTFSS   PB          ; TEST IF PB IS PRESSED
42          GOTO    $-1         ; NO, RECHECK
43          BSF     LED         ; YES, LED ON
44
45          BTFSC   PB          ; CHECK IF PB IS RELEASED
46          GOTO    $-1         ; NO, RECHECK
47          BCF     LED         ; YES, TURN OFF LED
48          GOTO    REPEAT      ; REPEAT CODE
```



Activity 4: Using Push buttons



Activity 4: Using Push buttons

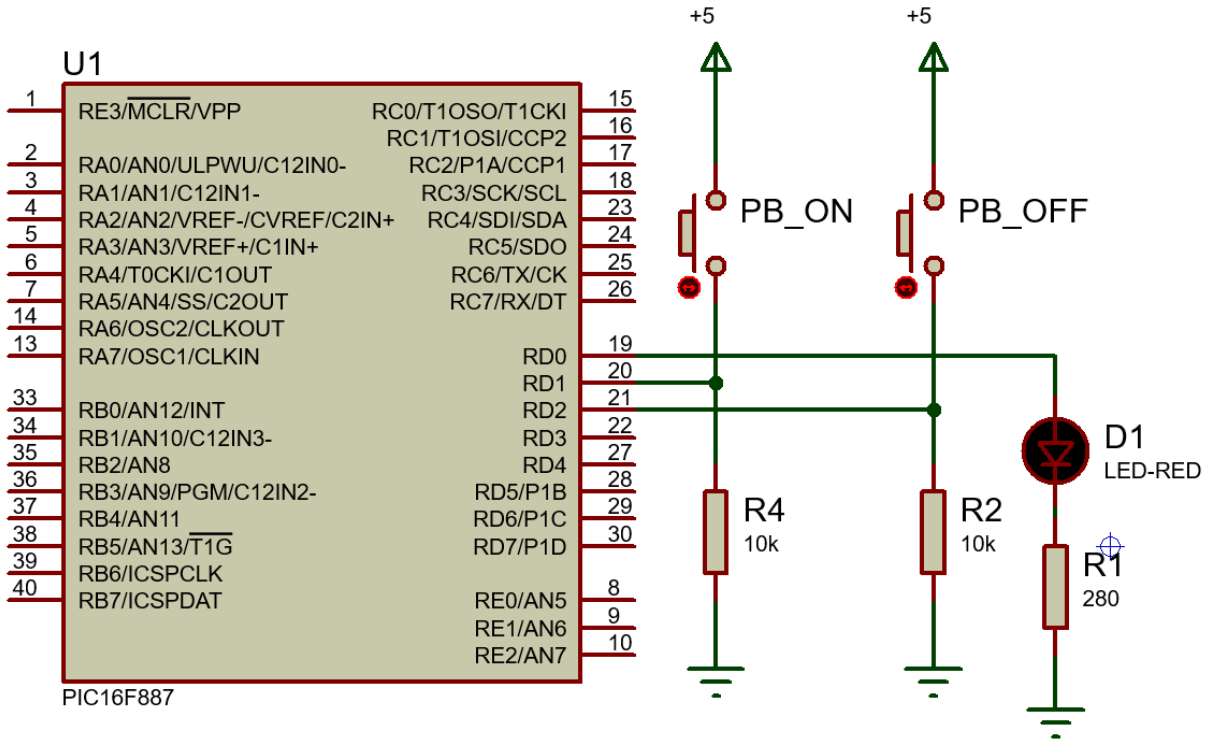


Activity 5: Using Push buttons

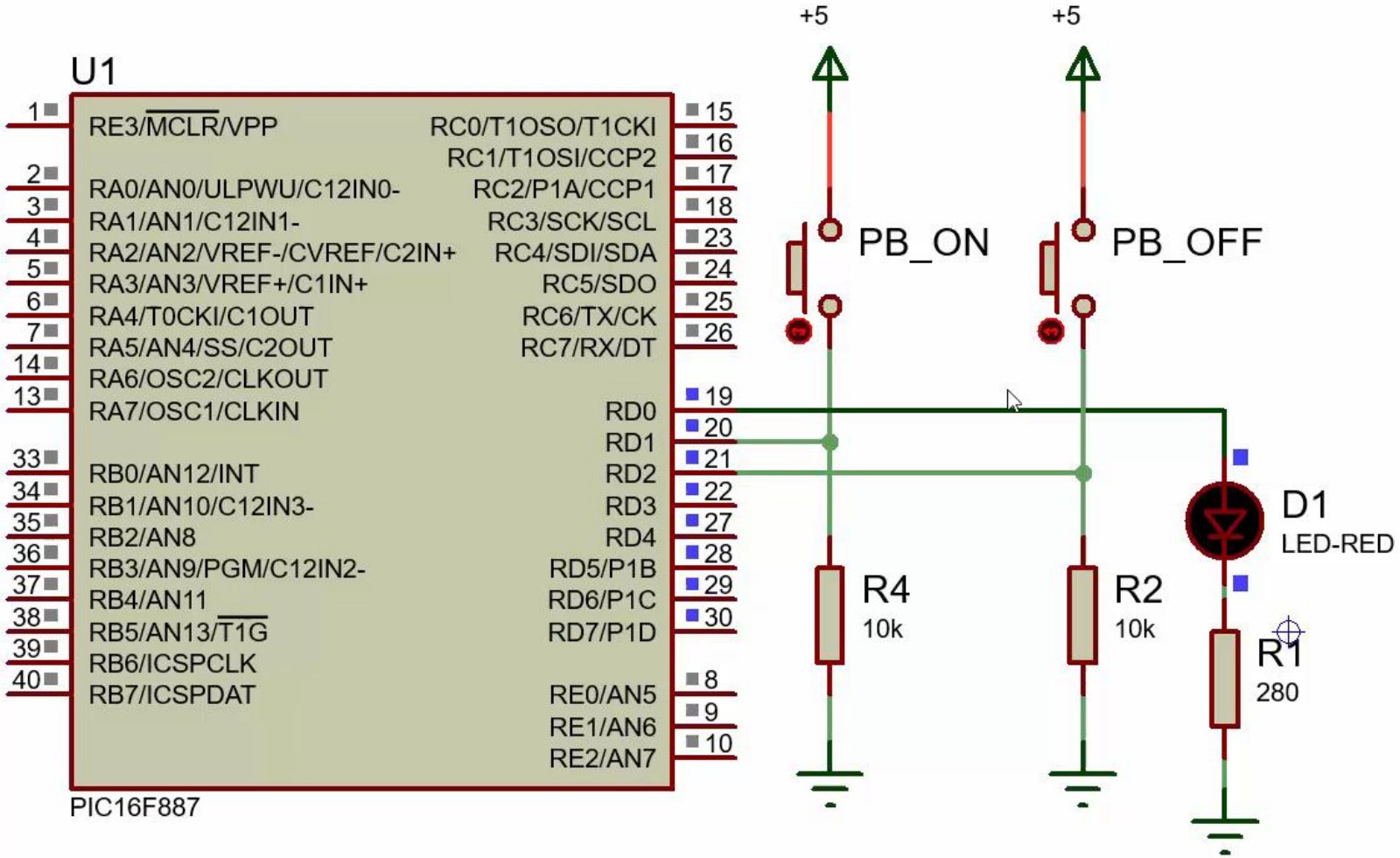
- In this Activity, we will add another PB. One is used to turn ON the LED and the other is used to turn OFF the LED. PBs are defines as RD1 and RD2. Of course both pins are configure as inputs.

```
28      #define LED         RD0
29      #define PB_ON       RD1
30      #define PB_OFF      RD2
```

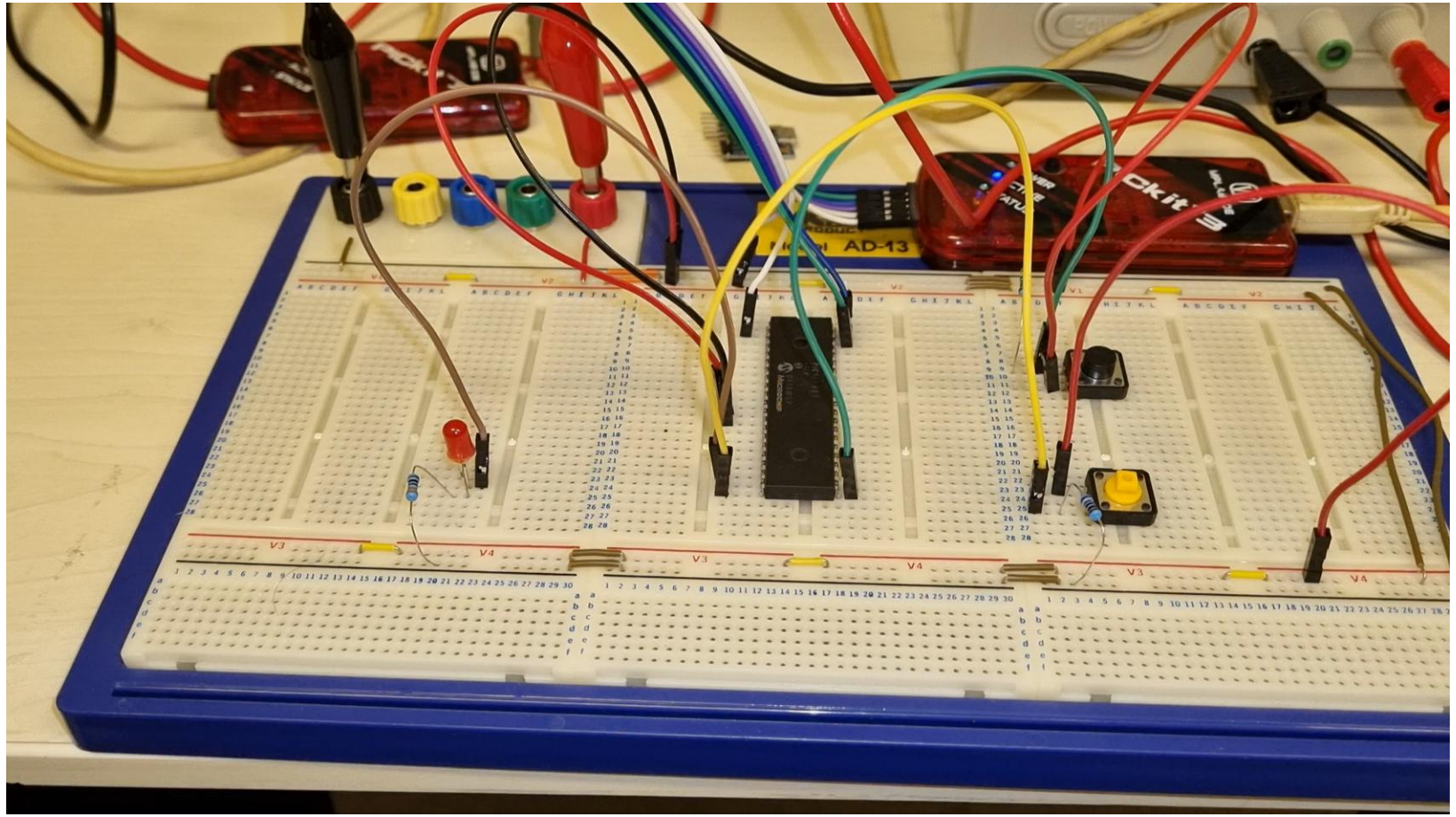
```
39  MAIN:
40      CALL    SETUP
41  REPEAT:
42      BTFSS   PB_ON    ; TEST PB_ON IF PRESSES
43      GOTO    $-1      ; NO, RECEHCK
44      BSF     LED      ; YES, TURN ON LED
45
46      BTFSS   PB_OFF   ; TESH PB_OFF IF PRESSED
47      GOTO    $-1      ; NO, RECHCK
48      BCF     LED      ; YES, TURN OFF LED
49      GOTO    REPEAT    ; REPEAT THE CODE
```



Activity 5: Using Push buttons



Activity 5: Using Push buttons





Test your knowledge

1. What is the optimal value of a pull up or pull-down resistor ?
2. Calculate the current flowing in a 1Ω resistor used as a pull-down when a active high push button is pressed (assume 5v supply voltage). What is your observation of using too small pull-down resistor ?
3. Open the datasheet and check the register called WPUB. What is it role, advantage and disadvantage ?



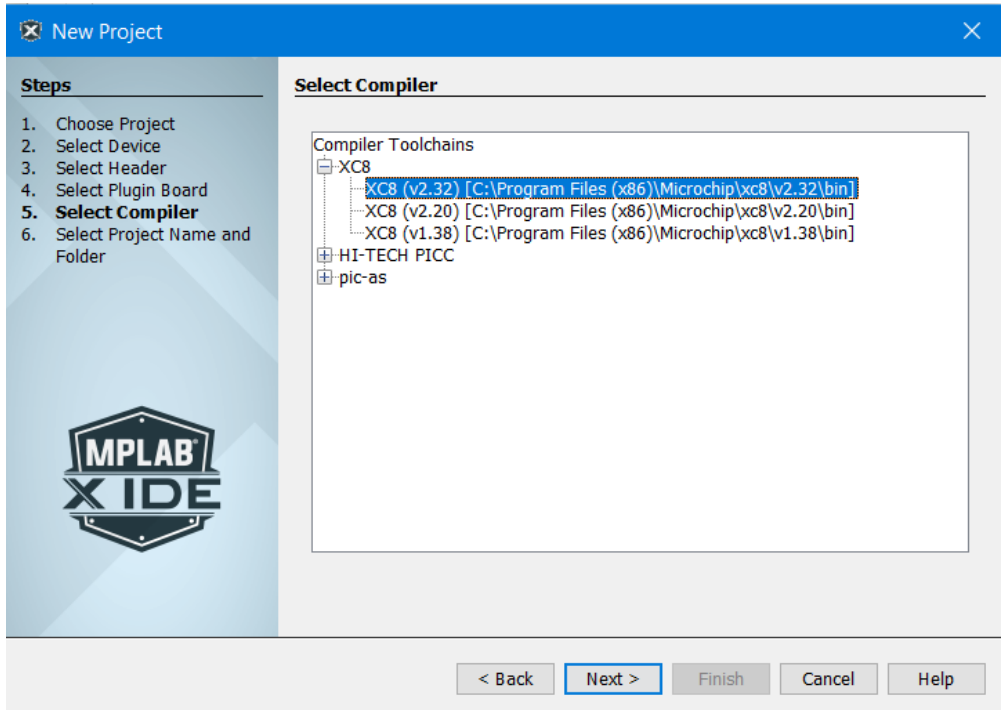
Week 4

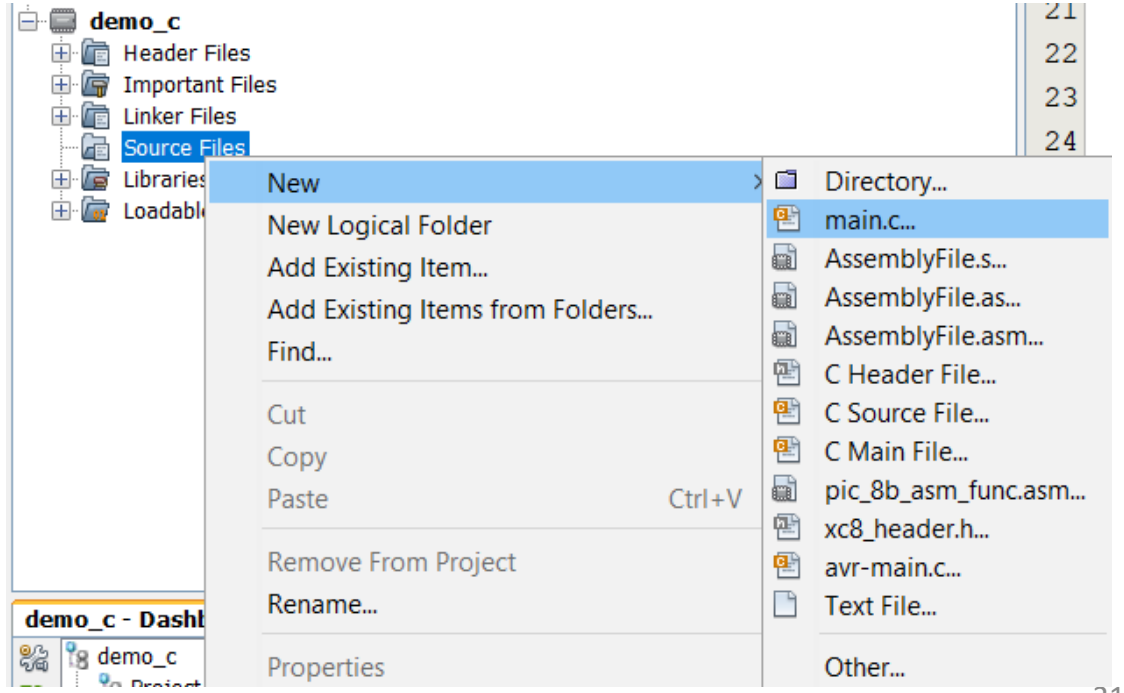
Lecture 8 / LO2

C language Using MPLABX XC8 compiler

Creating C project in MPLABX

- Creating C project steps are similar to the assembly project. Here we have to select the C compiler and not the pic-as.
- Also we need to create a C file to write our code. We can use the template main.c from the list.





Creating C project in MPLABX

- When created, the file appears as shown. The xc.h is included (in pic-as, the file is called xc.inc), also a main function is created.

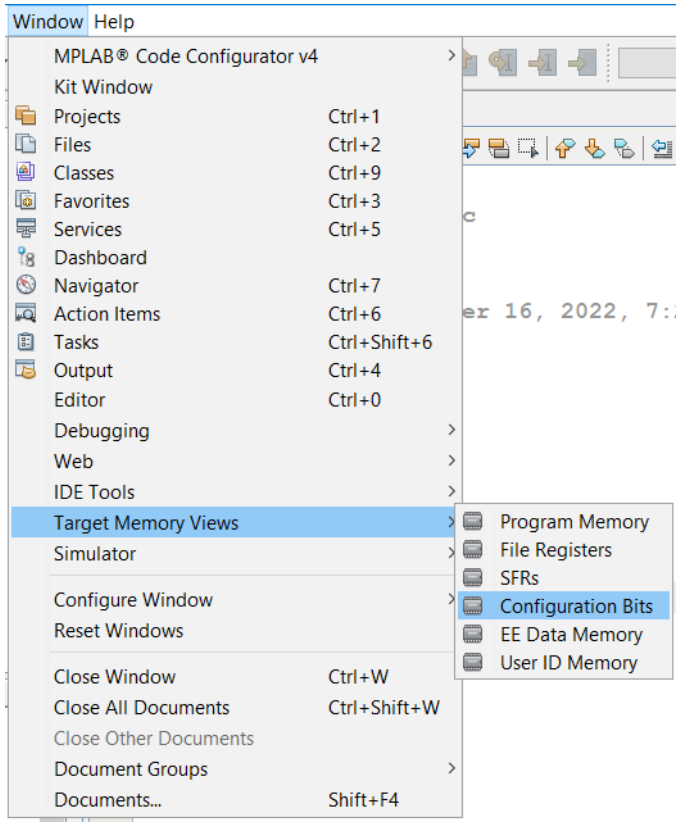
newmain1.c ×

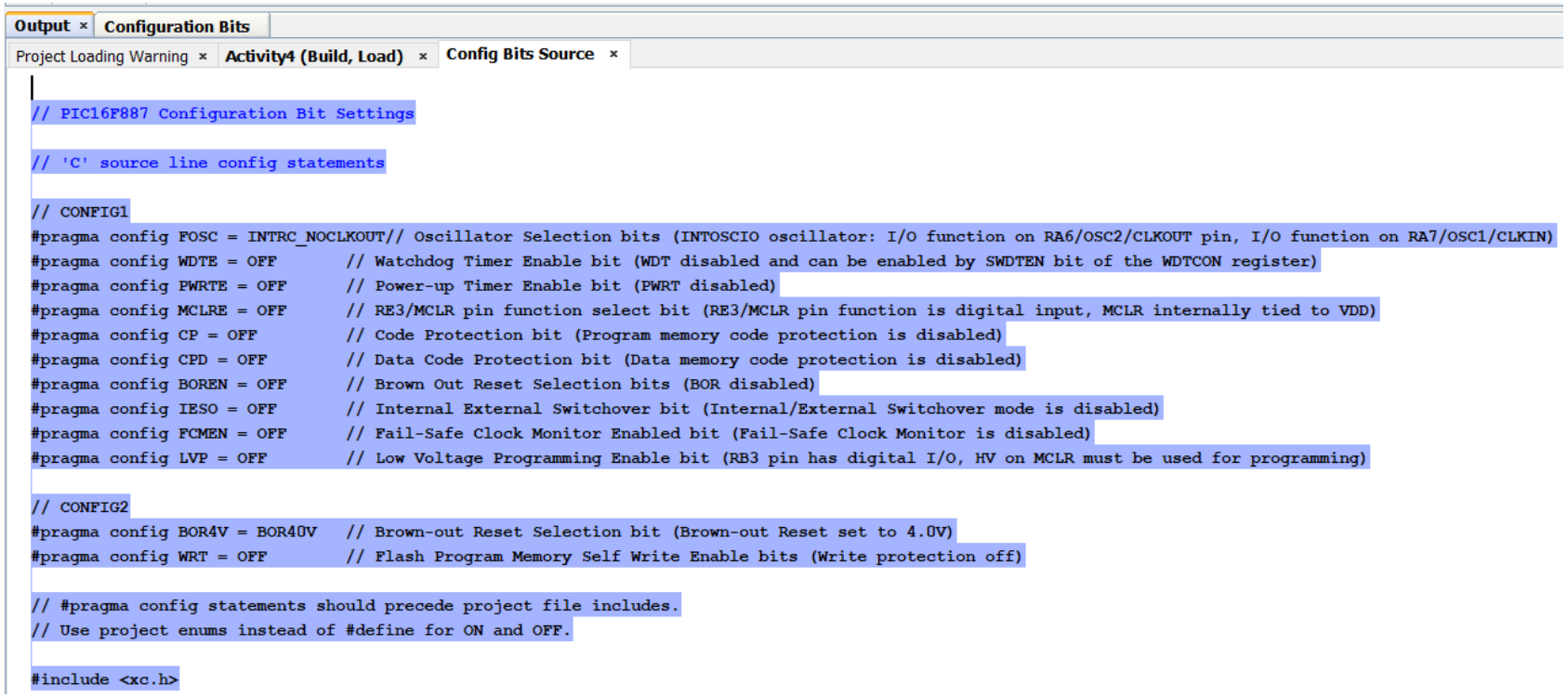
Source | History

```
1  /*
2   * File:   newmain1.c
3   * Author: User
4   *
5   * Created on December 16, 2022, 7:25 PM
6   */
7
8
9  #include <xc.h>
10
11 void main(void) {
12     return;
13 }
```

Creating C project in MPLABX

- Here also we need to generate the configuration bits. This step is exactly similar to the one done previously in assembly. Copy the code and paste it in the main file.







Creating C project in MPLABX

- Your code should appear like shown
- Only one line is required to define the crystal frequency as: #define _XTAL_FREQ 4000000, In case of 4MHz internal or external crystal.

(This will be required later for delay functions)

```
1   #pragma config FOSC = INTRC_NOCLKOUT// O
2   #pragma config WDTE = OFF       // Watch
3   #pragma config PWRTE = OFF      // Power
4   #pragma config MCLRE = OFF      // RE3/M
5   #pragma config CP = OFF         // Code
6   #pragma config CPD = OFF        // Data
7   #pragma config BOREN = OFF      // Brown
8   #pragma config IESO = OFF       // Inter
9   #pragma config FCMEN = OFF      // Fail-
10  #pragma config LVP = OFF        // Low V
11
12  // CONFIG2
13  #pragma config BOR4V = BOR40V   // Brown
14  #pragma config WRT = OFF        // Flash
15
16  // #pragma config statements should prec
17  // Use project enums instead of #define
18
19  #include <xc.h>
20
21  #define _XTAL_FREQ 4000000
23  void main(void) {
24      return;
25  }
```

Activity 5C: Start coding with C using XC8

- In this Activity, we will repeat the Activity 5 but in C language. The activity is about controlling LED with 2 PBs.

- **Step 1: Create the setup function**

```
void setup(){}

void main(void) {
    setup();
}
```

This is the function
It starts by void because it will not return any value
And has empty () because it will not take any argument

Note: Place the function before the main function

This is how we call the function in C



Activity 5C: Start coding with C using XC8

- **Step 2: writing the setup function**
- In C language, no need to define the banks. The compiler will take care of switching between banks.

- The binary values in assembly like 01100000B will be 0b01100000 in C

C format

```
void setup(){
    PORTD = 0;
    OSCCON = 0b01100000;
    TRISD = 0b00000110;
}
```

Assembly format

```
SETUP:
    CLRF    PORTD
    BSF     RP0
    MOVLW   01100000B
    MOVWF   OSCCON
    MOVLW   00000110B
    MOVWF   TRISD
    BCF     RP0
    RETURN
```



Activity 5C: Start coding with C using XC8

- **Step 3: Creating the infinite loop**
- In C language, the infinite loop is achieved using while(1). We used GOTO repeat to keep on repeating the code indefinitely.

C format

```
void main(void) {
    setup();

    while(1){


    }

}
```

Assembly format

```
MAIN:
    CALL    SETUP
REPEAT:


    GOTO    REPEAT    ; REPEAT THE CODE
```

Activity 5C: Start coding with C using XC8

- **Step 4: Creating the infinite loop Reading PB and tacking decision**
- In C language, the if condition is written as if(condition) {result of condition}
- We can remove the {} if the result of condition is one instruction

C format

```
while(1){
    if (pb_on ==1) led = 1;
    if (pb_off ==1)led = 0;
}
```

Assembly format

```
REPEAT:
    BTFSS   PB_ON
    GOTO    $-1          } if condition
    BSF     LED          result of condition

    BTFSS   PB_OFF
    GOTO    $-1          } if condition
    BCF     LED          result of condition
    GOTO    REPEAT
```



Activity 5C: Start coding with C using XC8

Complete C code

```
/* Activity 5 in C
 This code will turn on an LED when ON-PB is pressed
 and OFF when OFF-PB is pressed
 Designed to work with XC8 compiler 2.32
 PIC used is the PIC16f887
 Written by Nabil Karami for HCT 2023 */

#pragma config FOSC = INTRC_NOCLKOUT// Oscillator Sel
#pragma config WDTE = OFF       // Watchdog Timer Ena
#pragma config PWRTE = OFF      // Power-up Timer Ena
#pragma config MCLRE = OFF      // RE3/MCLR pin funct
#pragma config CP = OFF         // Code Protection bi
#pragma config CPD = OFF        // Data Code Protecti
#pragma config BOREN = OFF      // Brown Out Reset Se
#pragma config IESO = OFF       // Internal External
#pragma config FCMEN = OFF      // Fail-Safe Clock Mo
#pragma config LVP = OFF        // Low Voltage Progra

// CONFIG2
#pragma config BOR4V = BOR40V   // Brown-out Reset Se
#pragma config WRT = OFF        // Flash Program Memo
```

```
#include <xc.h>

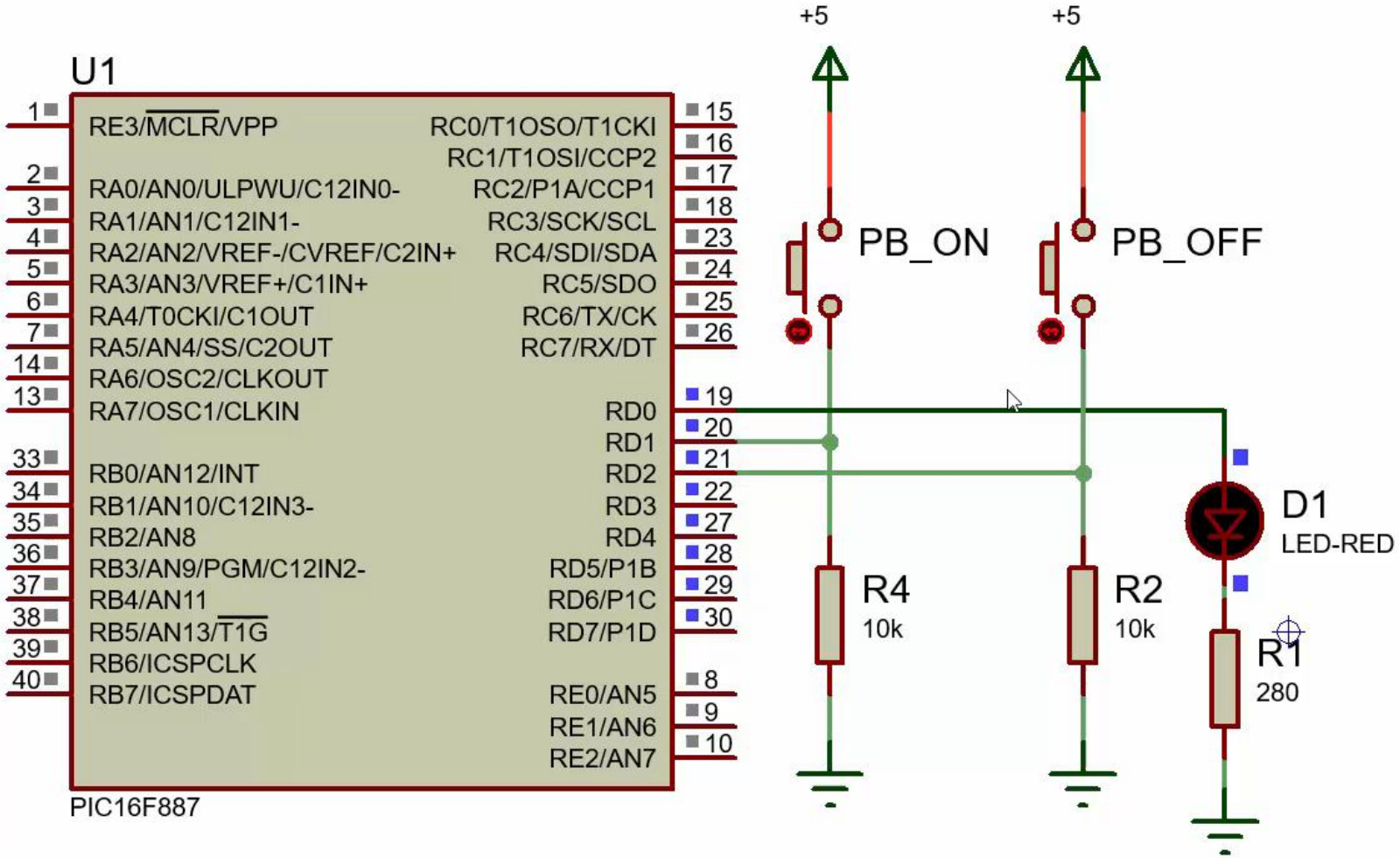
#define _XTAL_FREQ 4000000  // used crystal is 4MHz

#define led RD0             // define led as RD0
#define pb_on RD1           //
#define pb_off RD2          //

void setup(){
    PORTD = 0;              // clear portd
    OSCCON = 0b01100000;    // select 4MHz oscillator
    TRISD = 0b00000110;     // define in and out
}

void main(void) {
    setup();
    while(1){                   // infinite loop
        if (pb_on ==1) led = 1; // test pb, led on if pressed
        if (pb_off ==1)led = 0; // test pb, led off if pressed
    }
}
```

Activity 5C: Start coding with C using XC8



Remarks concerning assembly and C

- As shown previously that the assembly and C versions are identical, but the generated hex files are not the same.

Hex file generate from C code

```
:060000000A128A11FC2F18
:100FBA00831203138801603083160313 8F000630EF
:100FCA00880008000A128A11DD270A128A11831280
:100FDA00031388 1CF12FF22FF32F0814081DF62F84
:100FEA00F72FEC2F0810EC2F0A128A11002883 0120
:060FFA000A128A11E72F24
:04400E00D420FF3F7C
:00000001FF
```

Hex file generated from assembly code

```
:020000000528D1
:10000800090000D20881C0628081408 1D0928081056
:10001800062888018316603 08F00063088008312 16
:02002800 0800CE
:04400E00D420FF3F7C
:00000001FF
```

Remark 1: The C compiler converts the C file into a machine language without optimization (Maybe the paid version will optimize the code)

Lecture

Remarks concerning assembly and C

- **Remark 2: The flash memory shows that the code in C does not starts at address 0 and we have no control on defining the location of the code in the memory**

Hex file generate from C code

Line	Address	Opcode	Label	DisAssy
2012	07DB	3FFF		ADDLW 0xFF
2013	07DC	3FFF		ADDLW 0xFF
2014	07DD	1283		BCF STATUS, 0x5
2015	07DE	1303		BCF STATUS, 0x6
2016	07DF	0188		CLRF PORTD
2017	07E0	3060		MOVLW 0x60
2018	07E1	1683		BSF STATUS, 0x5
2019	07E2	1303		BCF STATUS, 0x6
2020	07E3	008F		MOVWF TMR1H
2021	07E4	3006		MOVLW 0x6
2022	07E5	0088		MOVWF PORTD
2023	07E6	0008		RETURN
2024	07E7	120A		BCF PCLATH, 0x4
2025	07E8	118A		BCF PCLATH, 0x3
2026	07E9	27DD		CALL 0x7DD
2027	07EA	120A		BCF PCLATH, 0x4
2028	07EB	118A		BCF PCLATH, 0x3
2029	07EC	1283		BCF STATUS, 0x5
2030	07ED	1303		BCF STATUS, 0x6
2031	07EE	1C88		BTFSS PORTD,...
2032	07EF	2FF1		GOTO 0x7F1
2033	07F0	2FF2		GOTO 0x7F2
2034	07F1	2FF3		GOTO 0x7F3
2035	07F2	1408		BSF PORTD, 0x0
2036	07F3	1D08		BTFSS PORTD,...
2037	07F4	2FF6		GOTO 0x7F6
2038	07F5	2FF7		GOTO 0x7F7
2039	07F6	2FEC		GOTO 0x7EC
2040	07F7	1008		BCF PORTD, 0x0
2041	07F8	2FEC		GOTO 0x7EC
2042	07F9	120A		BCF PCLATH, 0x4
2043	07FA	118A		BCF PCLATH, 0x3
2044	07FB	2800		GOTO 0x0
2045	07FC	0183		CLRF STATUS
2046	07FD	120A		BCF PCLATH, 0x4
2047	07FE	118A		BCF PCLATH, 0x3
2048	07FF	2FE7		GOTO 0x7E7

Code starts at location 0x7DD !!!

Code starts at location 0x00 as defined by the ORG in directive

Hex file generated from assembly code

Line	Address	Opcode	Label	DisAssy
1	0000	2805		GOTO 0x5
2	0001	3FFF		ADDLW 0xFF
3	0002	3FFF		ADDLW 0xFF
4	0003	3FFF		ADDLW 0xFF
5	0004	0009		RETFIE
6	0005	200D		CALL 0xD
7	0006	1C88		BTFSS PORTD,...
8	0007	2806		GOTO 0x6
9	0008	1408		BSF PORTD, 0x0
10	0009	1D08		BTFSS PORTD,...
11	000A	2809		GOTO 0x9
12	000B	1008		BCF PORTD, 0x0
13	000C	2806		GOTO 0x6
14	000D	0188		CLRF PORTD
15	000E	1683		BSF STATUS, 0x5
16	000F	3060		MOVLW 0x60
17	0010	008F		MOVWF TMR1H
18	0011	3006		MOVLW 0x6
19	0012	0088		MOVWF PORTD
20	0013	1283		BCF STATUS, 0x5
21	0014	0008		RETURN



Remarks concerning assembly and C

Final conclusion

- **C is easier compared to the assembly language,**
- **Assembly has direct control on registers.**

- **In C, we cannot predict how much time the code will take to be executed**
- **In Assembly, we know how the lines are executed and how much time each instruction needs to be executed.**



Delay function in C and Assembly

- Delay function is usually used in program to slow down some process. As we already know that the speed of execution of instruction is very high due to the crystal high frequency.
- In C language there is already a ready made delay functions that can be called directly. The original delay function in C is _delay(n) where n is the number of instruction count.

- This is an in-line function that is expanded by the code generator.
- When called, this routine expands to an in-line assembly delay sequence. The sequence will consist of code that delays for the number of instruction cycles that is specified as the argument. The argument must be a constant expression.



Delay function in C and Assembly

- It is often more convenient to request a delay in time-based terms, rather than in cycle counts.
- The macros __delay_ms(x) and __delay_us(x) are provided to meet this need.
- These macros convert the time-based request into instruction cycles that can be used with _delay(n).

```
__delay_ms(x) // request a delay in milliseconds
__delay_us(x) // request a delay in microseconds
```

Delay function in C and Assembly

- If we write a code in C that only includes a delay function, we can conclude the assembly format of the code that makes the same delay.
- For a small value in the delay function (100 for example), the code is converted to a for loop in assembly with one variable

```
void main(void) {
    _delay(100);
    while (1);
}
```

Code converted to

Line	Address	Opcode	Label	DisAssy
2033	07F0	3FFF		ADDLW 0xFF
2034	07F1	3FFF		ADDLW 0xFF
2035	07F2	3FFF		ADDLW 0xFF
2036	07F3	3FFF		ADDLW 0xFF
2037	07F4	3021		MOVLW 0x21
2038	07F5	00F0		MOVWF 0x70
2039	07F6	0BF0		DECFSZ 0x70, F
2040	07F7	2FF6		GOTO 0x7F6
2041	07F8	2FF8		GOTO 0x7F8
2042	07F9	120A		BCF PCLATH, 0x4
2043	07FA	118A		BCF PCLATH, 0x3
2044	07FB	2800		GOTO 0x0
2045	07FC	0183		CLRF STATUS
2046	07FD	120A		BCF PCLATH, 0x4
2047	07FE	118A		BCF PCLATH, 0x3
2048	07FF	2FF4		GOTO 0x7F4
2049	0800	3FFF		ADDLW 0xFF

MOVLW 0x21 takes 1 cycle
MOVWF 0x70 takes 1 cycle
} 2 cycles

DECFSZ 0x70, F takes 1 cycle
GOTO $-1 takes 2 cycles

These are repeated 33-1 times (0x21-1), i.e., 32x3 = 96 cycles

The last DECFSZ (when the value reaches 0) will jump and make 2 cycles, thus in total 98

Total of 100 cycles



Delay function in C and Assembly

- The C compiler did the calculation of the value that be loaded in address 70 so that the entire function takes 100 cycles

- Since the register 0x70 is an 8 bit register, the max value that can be loaded in is 255 (0xFF).
- If we redo the calculation, we can see that a function with one variable can take max 766 cycles

```
MOVLW 0xFF      takes 1 cycle
MOVWF 0x70      takes 1 cycle
DECFSZ 0x70, F  takes 1 cycle
GOTO $-1        takes 2 cycles
```

2 cycles

These are repeated 255 times , i.e., 254x3 = 762 cycles
The last DECFSZ (when the value reaches 0) will jump and make 2 cycles, thus in total 764

Total of 766 cycles

Delay function in C and Assembly

And this is proven when writing delay 766 in C and got the assembly equivalent code

```
void main(void) {
    _delay(766);
    while (1);
}
```

2037	07F4	30FF		MOVLW 0xFF
2038	07F5	00F0		MOVWF 0x70
2039	07F6	0BF0		DECFSZ 0x70, F
2040	07F7	2FF6		GOTO 0x7F6
2041	07F8	2FF8		GOTO 0x7F8

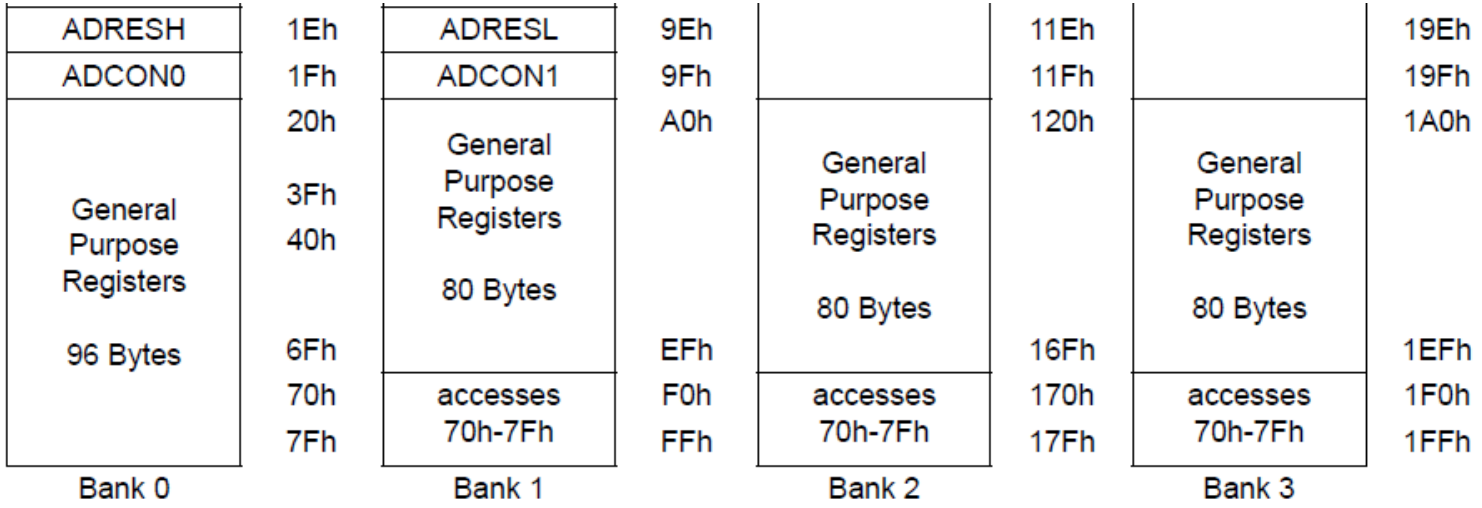
If the value in delay is higher than 766, the C compiler will create two or more cascaded **for loops** in assembly using two or more variables

Delay function in C and Assembly

Important note: why the compiler is using address 0x70 and not any other register in GFR ?!!

2037	07F4	30FF		MOVLW 0xFF
2038	07F5	00F0		MOVWF 0x70
2039	07F6	0BF0		DECFSZ 0x70, F
2040	07F7	2FF6		GOTO 0x7F6
2041	07F8	2FF8		GOTO 0x7F8

The answer is found in the RAM. The address 0x70 to 0x7F (16 registers) are accessible in all banks, which means that the compiler is using these registers are GLOBAL VARIABLE



Delay function in C and Assembly

Let's increase now the delay to 1000,000 and check what would be the assembly equivalent code

```
void main(void) {
    _delay(1000000);
    while (1);
}
```

	2025	07E8	3FFF		ADDLW 0xFF
	2026	07E9	3FFF		ADDLW 0xFF
	2027	07EA	3FFF		ADDLW 0xFF
	2028	07EB	3006		MOVLW 0x6
	2029	07EC	00F2		MOVWF 0x72
	2030	07ED	3013		MOVLW 0x13
	2031	07EE	00F1		MOVWF 0x71
	2032	07EF	30AD		MOVLW 0xAD
	2033	07F0	00F0		MOVWF 0x70
	2034	07F1	0BF0		DECFSZ 0x70, F
	2035	07F2	2FF1		GOTO 0x7F1
	2036	07F3	0BF1		DECFSZ 0x71, F
	2037	07F4	2FF1		GOTO 0x7F1
	2038	07F5	0BF2		DECFSZ 0x72, F
	2039	07F6	2FF1		GOTO 0x7F1
	2040	07F7	2FF8		GOTO 0x7F8
	2041	07F8	2FF8		GOTO 0x7F8

Loading 0x6 in register 0x72

Loading 0x13 in register 0x71

Loading 0xAD in register 0x70

First for loop

Second for loop

Third for loop

For big delay value, the compiler is creating 3 for loop with 3 variables



Delay function in C and Assembly

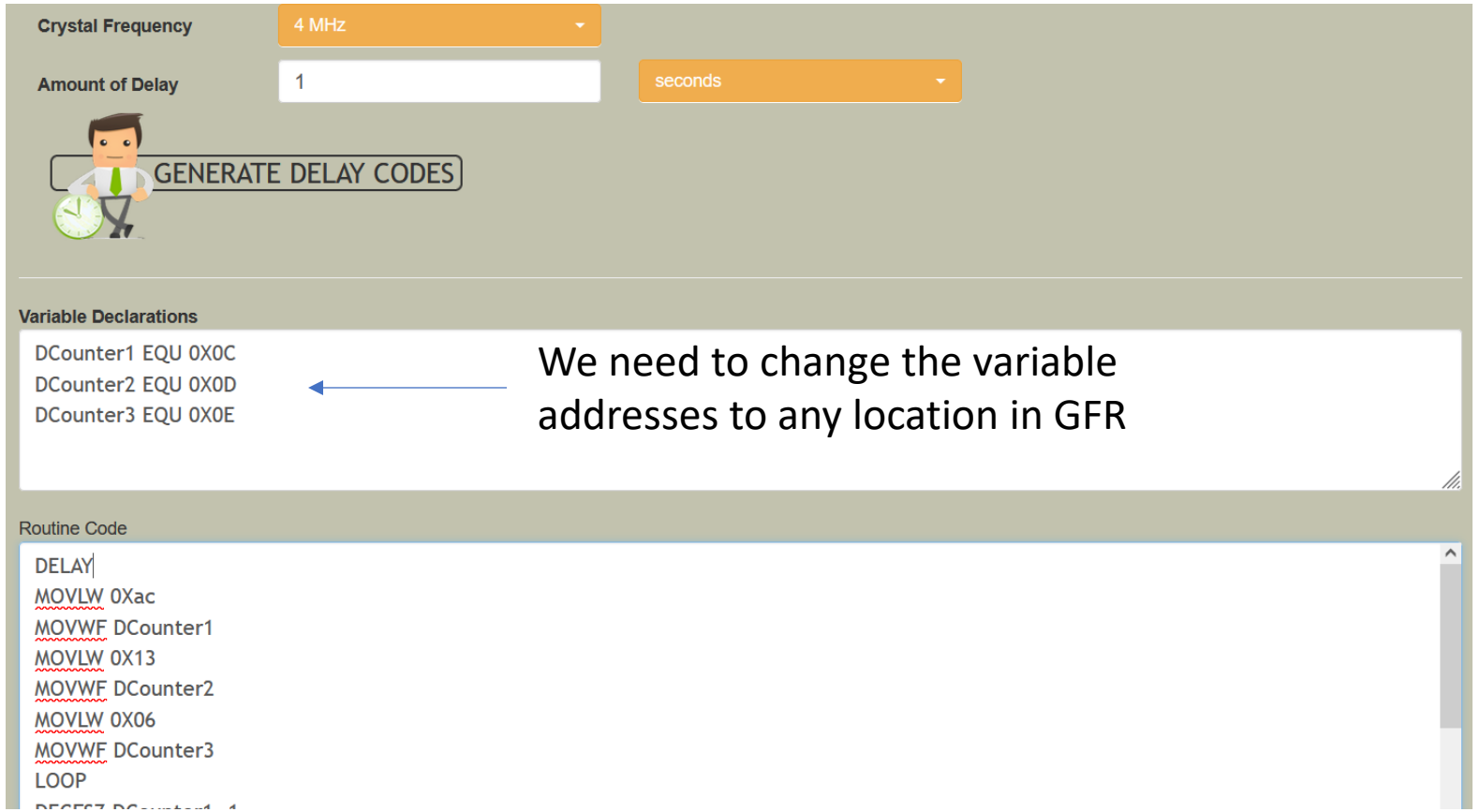
- After knowing the format of the delay function in assembly, the question now is what should be loaded in 0x70, 0x71 and 0x72 to have a particular delay time!

- The answer is that the variables are calculated by iterations and it is very hard to make it by hand. The compiler is making 256 * 256 * 256 iterations to find the nearest values of variables that lead to the desired time delay.

- Lot of online calculators can do the job and generate assembly code ready to be paste in your code… check this link www.onlinepiccompiler.com/delayGeneratorENG.php

Delay function in Assembly

- For example, the online calculator generates a code for a 1 second delay, ready to use





Data type in C

- The MPLAB XC8 compiler supports integer data types with 1, 2, 3 and 4 byte sizes as well as a single bit type.
- All integer values are represented in little endian format with the Least Significant Byte (LSB) at the lower address.

TABLE 5-3: INTEGER DATA TYPES

Type	Size (bits)
bit	1
signed char	8
unsigned char	8
signed short	16
unsigned short	16
signed int	16
unsigned int	16
signed short long	24
unsigned short long	24
signed long	32
unsigned long	32
signed long long	32
unsigned long long	32

TABLE 5-4: RANGES OF INTEGER TYPE VALUES

Symbol	Meaning	Value
CHAR_BIT	bits per char	8
CHAR_MAX	max. value of a char	127
CHAR_MIN	min. value of a char	-128
SCHAR_MAX	max. value of a signed char	127
SCHAR_MIN	min. value of a signed char	-128
UCHAR_MAX	max. value of an unsigned char	255
SHRT_MAX	max. value of a short	32767
SHRT_MIN	min. value of a short	-32768
USHRT_MAX	max. value of an unsigned short	65535
INT_MAX	max. value of an int	32767
INT_MIN	min. value of a int	-32768
UINT_MAX	max. value of an unsigned int	65535
SHRTLONG_MAX	max. value of a short long	8388607
SHRTLONG_MIN	min. value of a short long	-8388608
USHRTLONG_MAX	max. value of an unsigned short long	16777215
LONG_MAX	max. value of a long	2147483647
LONG_MIN	min. value of a long	-2147483648
ULONG_MAX	max. value of an unsigned long	4294967295
LLONG_MAX	max. value of a long long	2147483647
LLONG_MIN	min. value of a long long	-2147483648
ULLONG_MAX	max. value of an unsigned long long	4294967295

- If no signedness is specified in the type, then the type will be signed except for the char types which are always unsigned.
- Signed values are stored as a two's complement integer value



Data type in C

- For both float and double values, the 24-bit format is the default.
- The options --FLOAT=24 and --DOUBLE=24 can also be used to specify this explicitly.
- The 32-bit format is used for double values if the --DOUBLE=32 option is used and for float values if --FLOAT=32 is used.
- Variables can be declared using the float and double keywords, respectively, to hold values of these types.
- Floating-point types are always signed and the unsigned keyword is illegal when specifying a floating-point type.
- Types declared as long double will use the same format as types declared as double.
- All floating-point values are represented in little endian format with the LSB at the lower address.

TABLE 5-5: FLOATING-POINT DATA TYPES

Type	Size (bits)	Arithmetic Type
`float`	24 or 32	Real
`double`	24 or 32	Real
`long double`	same as `double`	Real



Test your knowledge

1. Where is the syntax mistake in `if (PORTD = 20) PORTC = 20;`
2. What is the role of `while(true);`
3. How many cycles are required by `DECFSZ Reg1, F` if Reg1=20
4. How many cycles are required by `DECFSZ Reg1, F` if Reg1=1
5. Having Reg1 = 0, what would be its value after `DECFSZ Reg1, F`
6. Go through web and check what is the Big Endian Versus Little Endian Byte Ordering



Week 4

Lab 3 / LO2

Blinking LED in C and Assembly



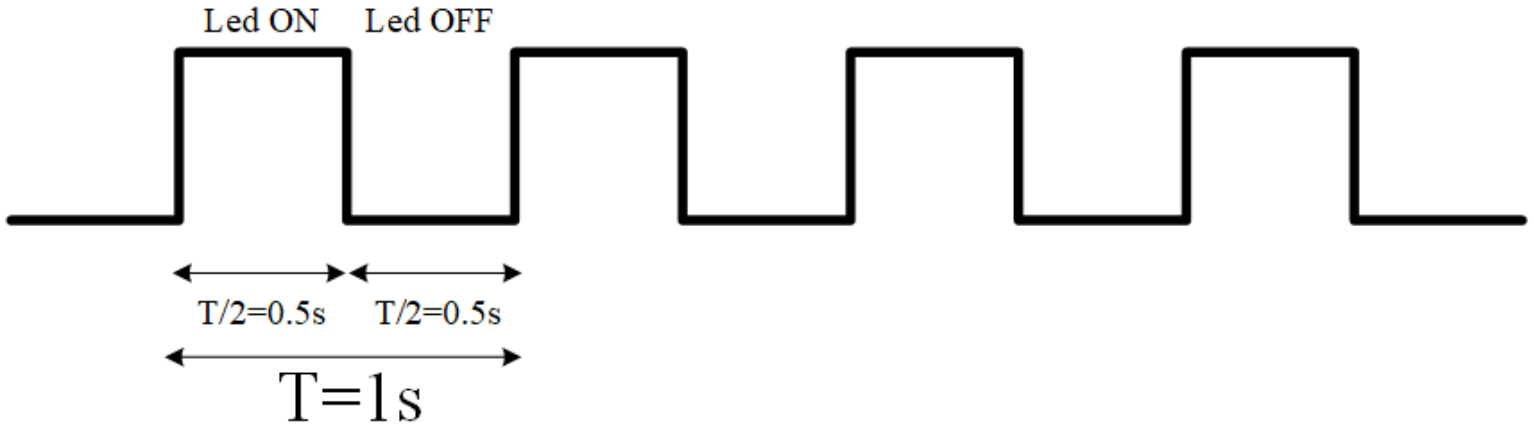
Activity 6: Blinking Led

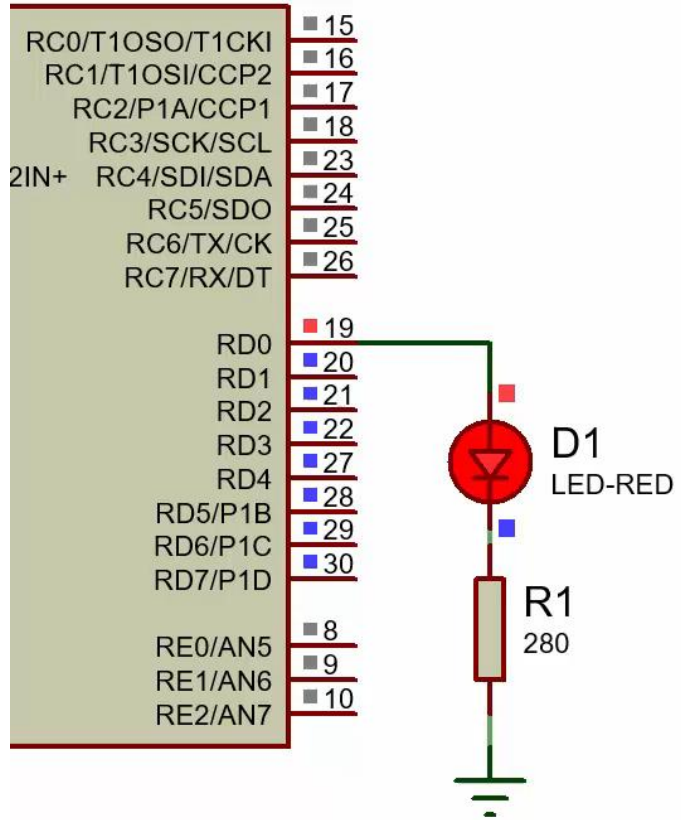
- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. PIC programmer
  4. Power supply 5V
  5. Breadboard wires (male-male)

- Required components:
  1. PIC 16F887
  2. Led x1
  3. Resistor from 200 to 470 ohm x 1

Activity 6: Blinking Led

- In this activity, we will blink a led with a frequency of 1Hz, i.e., Period of 1s.
- Blink means ON and OFF
- The ON and the OFF time are T/2, i.e., 500ms
-





Activity 6: Blinking Led

- The 3 variables are called A1, A2 and A3 and are assigned to addresses 0x70, 71 and 72, respectively.
- The directive EQU is used to declare the variables, i.e., to allocate a name to the address, so it will be easy to call the address by its name. the Delay is generated from the link and modified to fit our code

Variable declaration

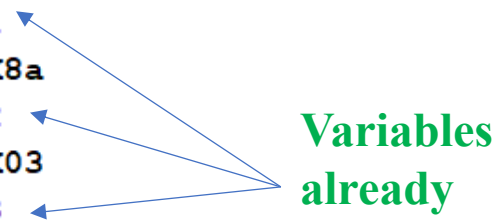
```
A1 EQU 0X70
A2 EQU 0X71
A3 EQU 0X72
```

Main function

```
MAIN:
    CALL    SETUP
REPEAT:
    BSF     LED
    CALL    DELAY_500MS
    BCF     LED
    CALL    DELAY_500MS
    GOTO    REPEAT
```

Delay function

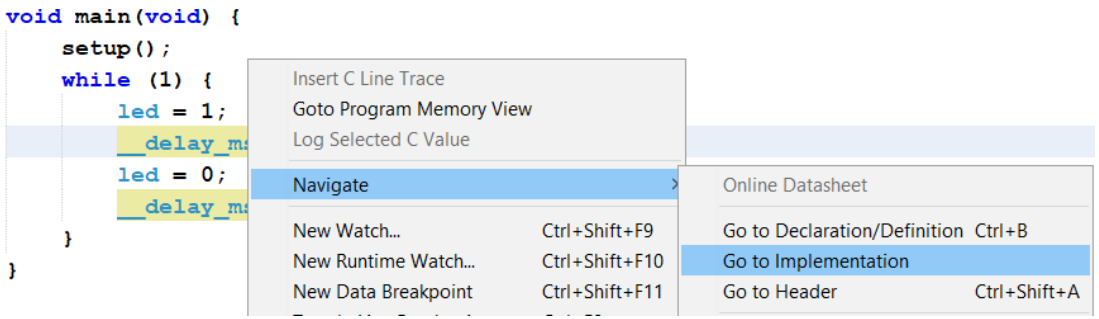
```
DELAY_500MS:
    MOVLW   0X54
    MOVWF   A1
    MOVLW   0X8a
    MOVWF   A2
    MOVLW   0X03
    MOVWF   A3
    DECFSZ  A1, 1
    GOTO    $-1
    DECFSZ  A2, 1
    GOTO    $-3
    DECFSZ  A3, 1
    GOTO    $-5
    NOP
    RETURN
```



Lab

Activity 6C: Blinking Led

- The same code is now written in C.
- The delay functions used here are the time-based delay function __delay_ms(). This requires the definition of the crystal frequency.
- You can check the C code of this function by right clicking on the function and navigating to the implementation. This will open the builtins.h file




```c
#include <xc.h>

#define _XTAL_FREQ 4000000  // used crystal is 4MHz
#define led RD0

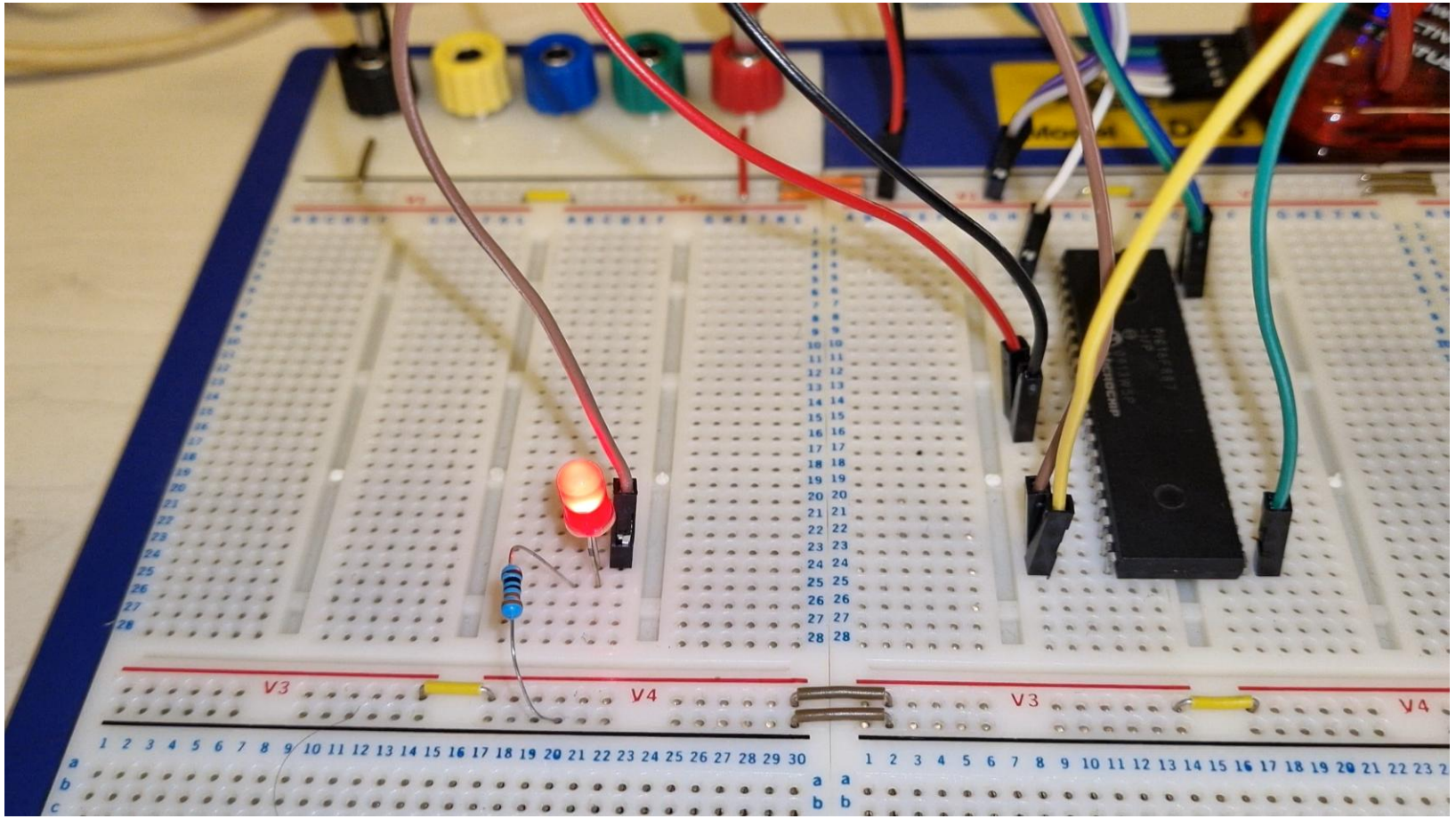
void setup() {
    PORTD = 0; // clear portd
    OSCCON = 0b01100000; // select 4MHz oscillator
    TRISD = 0b00000000; // define port as out
}

void main(void) {
    setup();
    while (1) {
        led = 1;
        __delay_ms(500);
        led = 0;
        __delay_ms(500);
    }
}
```

```c
#define __delay_ms(x) _delay((unsigned long)((x)*(_XTAL_FREQ/4000.0)))
```

The __delay_ms(x) depends on the _delay() function multiplied by x and the 4000000/4000 which is 1000
Thus a __delay_ms(500) will call a _delay(500,000)

Activity 6C: Blinking Led



Lab



Week 5

Lecture

Lecture 9 / LO2

Reading PORTS and Comparing data

Activity 7: Reading PORTS and comparing results

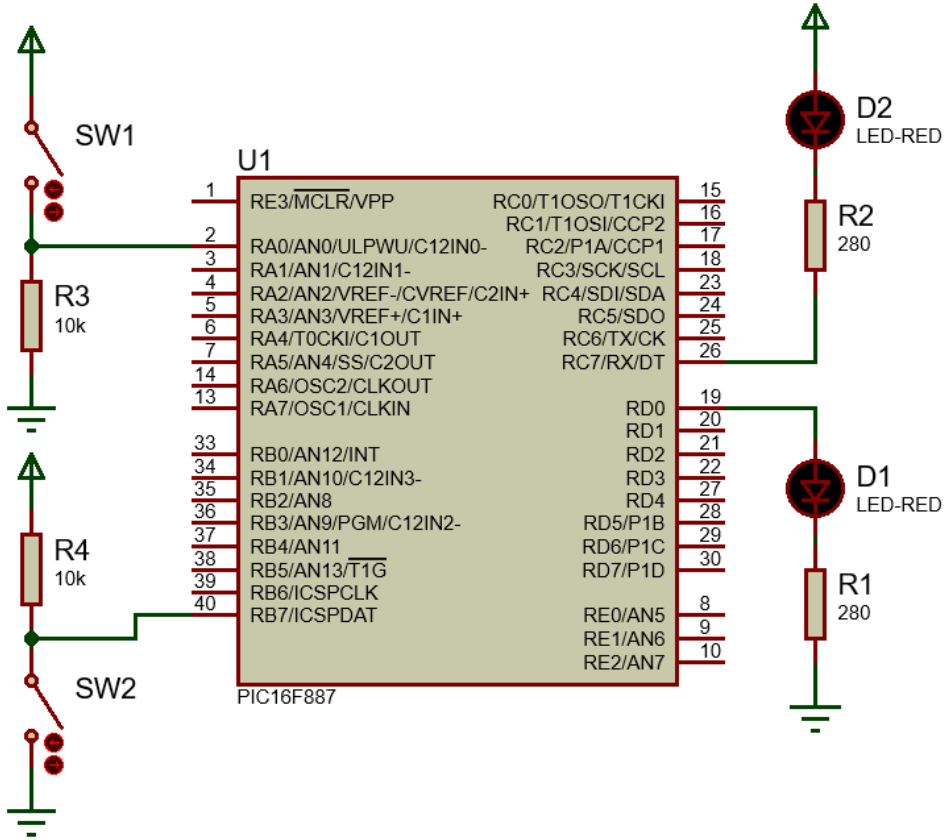
- In this activity, we will learn how to read data available on PORTs and compare them to predefined values to achieve a specific task.
- Assume for example that we want our PIC to read 4 switches and control 3 LED based on the table below

SW3	SW2	SW1	SW0	Led2	Led1	Led0
pressed	Not pressed	Not pressed	pressed	ON	ON	ON
Not pressed	pressed	pressed	pressed	ON	OFF	OFF
pressed	pressed	Not pressed	Not pressed	OFF	ON	ON
else				OFF	OFF	OFF

- The first step is to assign switches to pins. We will use PORTD for switches and PORTC for Leds

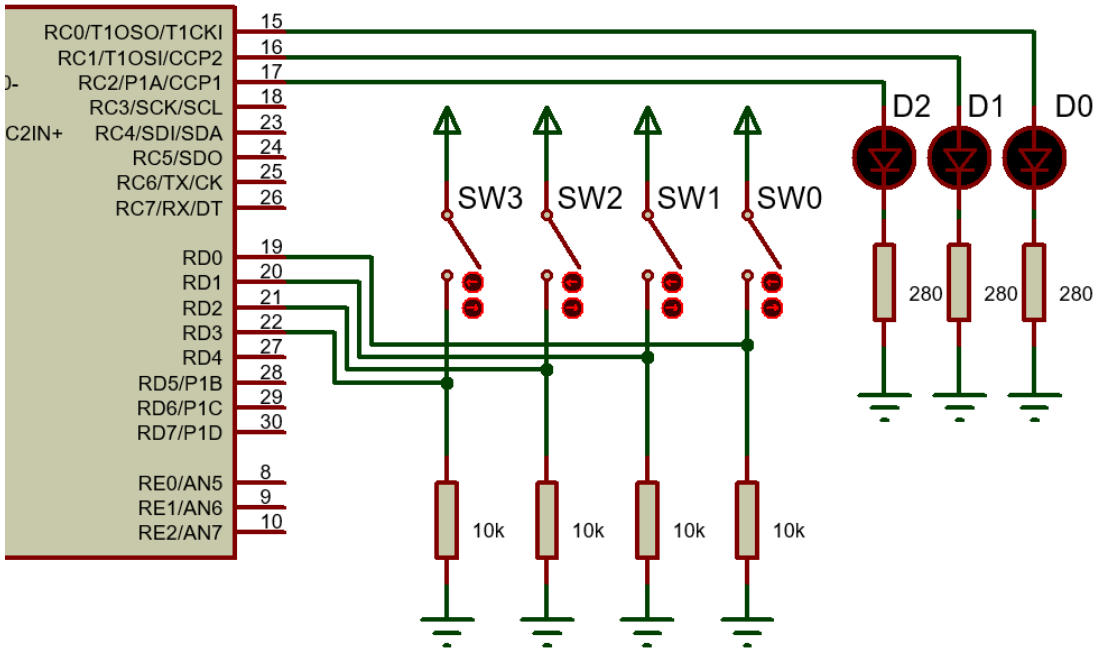
Activity 7: Reading PORTS and comparing results

- Step2: We have to know how the switches are connected (Active high or active low ?)
  - Active high means when pressed, the logic level read is 1, and 0 when released (case of SW1) (more logical action)
  - Active low means when pressed, the logic level is 0, and 1 when released (case of SW2)
- Step 3: We have also to know how the Leds are connected (Logic 1 turns ON or OFF the Leds)?
  - Led D2 turns ON when the pin RC7 is cleared, and OFF when set
  - Led D1 turns ON when RD0 is set and OFF when cleared (more logical connection)



Activity 7: Reading PORTS and comparing results

- It is assumes that we selected the more logic configuration of switches and leds and therefore the table is updated as
- Pressed is replaced by 1 logic
- Not pressed is replaced by 0 logic
- ON is replaced by 1 logic
- OFF is replaced by 0 logic



SW3 RD3	SW2 RD2	SW1 RD1	SW0 RD0	Led2 RC2	Led1 RC1	Led0 RC0
1	0	0	1	1	1	1
0	1	1	1	1	0	0
1	1	0	0	0	1	1
else				0	0	0



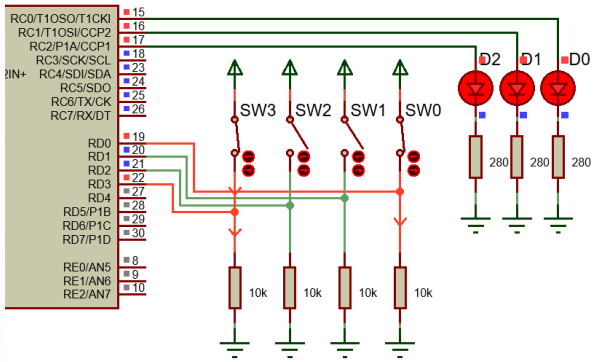
Activity 7: Reading PORTS and comparing results

```
27      PORTD_TEMP EQU 0X20

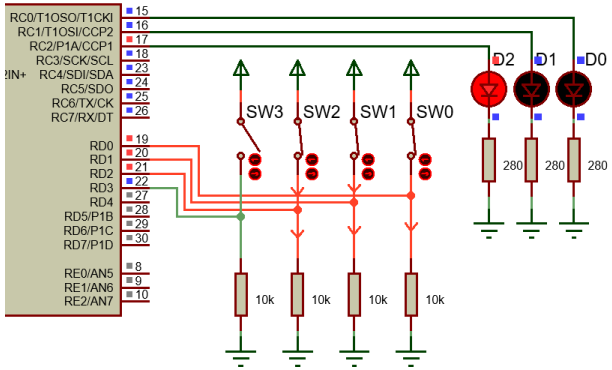
34  MAIN:
35      CALL    SETUP
36  REPEAT:
37      MOVF    PORTD, W
38      ANDLW   00001111B   ; MASKING ONLY THE 4 LEAST BITS
39      MOVWF   PORTD_TEMP  ; AND SAVING IN A TEMPORARY FILE
40      CALL    READ_SWITCHES
41      MOVWF   PORTC       ; PUT THE RETURNED VALUE ON PORTC
42      GOTO    REPEAT      ; REPEAT CODE
44  READ_SWITCHES:
45      MOVF    PORTD_TEMP, W
46      XORLW   00001001B   ; COMPARE WITH 1001
47      BTFSC   ZERO
48      RETLW   00000111B   ; CASE 1, RETURN THIS VALUE TO THE MAIN CODE
49
50      MOVF    PORTD_TEMP, W
51      XORLW   00000111B   ; COMPARE WITH 0111
52      BTFSC   ZERO
53      RETLW   00000100B   ; CASE 2, RETURN THIS VALUE TO THE MAIN CODE
54
55      MOVF    PORTD_TEMP, W
56      XORLW   00001100B   ; COMPARE WITH 1100
57      BTFSC   ZERO
58      RETLW   00000011B   ; CASE 3, RETURN THIS VALUE TO THE MAIN CODE
59
60      RETLW   00000000B   ; CASE OF ELSE, RETURN THIS VALUE TO THE MAIN CODE
```

- In the main code, the setup is called to configure input and outputs
- Then PORTD is read and ANDed with 00001111 to keep in the least 4 bits and remove the most 4 bits. This is called masking the required bits.
- Then it is saved in a temporary file to be used later
- In the READ_SWITCHES function, the temporary file is compared to case 1, if are equal, (by testing the Zero bit), the led values are placed on W and the code returns to the main code. This is done using RETLW (L →W + return)
- If not equal, case 2 is tested….
- If non of the 3 cases are achieved, the last RETLW is used to mimic the else condition

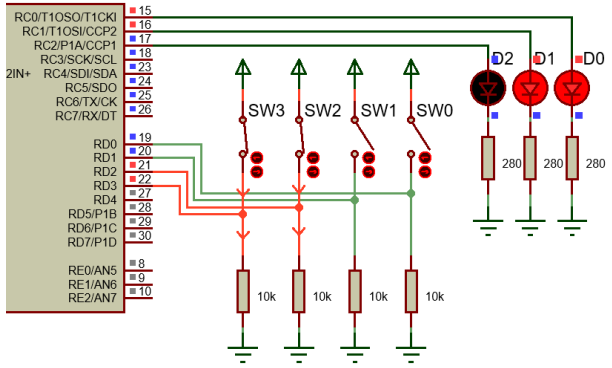
Activity 7: Reading PORTS and comparing results



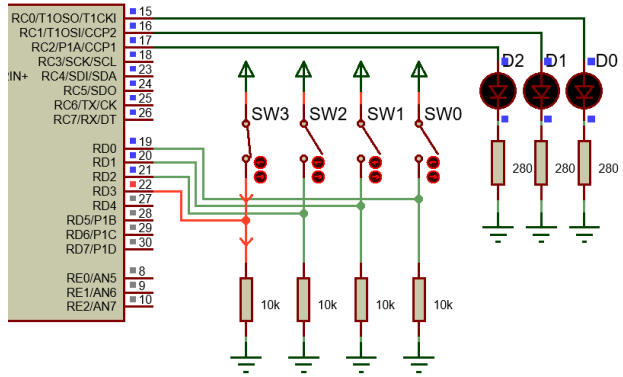
Case 1



Case 2



Case 3



Case else

SW3 RD3	SW2 RD2	SW1 RD1	SW0 RD0	Led2 RC2	Led1 RC1	Led0 RC0
1	0	0	1	1	1	1
0	1	1	1	1	0	0
1	1	0	0	0	1	1
else				0	0	0



Activity 7C: Reading PORTS and comparing results in C

- The same Activity is also made using C. At line 26, the variable is declared as char.
- At line 39, the bitwise AND is used to remove the most 4 bits and keep on the lowest 4 bits.

```
26    char PORTD_TEMP;
27
28    void setup() {
29        PORTD = 0;                // clear portd
30        PORTC = 0;
31        OSCCON = 0b01100000;      // select 4MHz oscillator
32        TRISD = 0b11111111;       // define port as out
33        TRISC = 0b00000000;
34    }
35
36    void main(void) {
37        setup();
38        while (1) {
39            PORTD_TEMP = PORTD & 0b00001111; // masking only the 4 lowest bits
40            if (PORTD_TEMP == 0b00001001) PORTC = 0b00000111;        // case 1
41            else if (PORTD_TEMP == 0b00000111) PORTC = 0b00000100;   // case 2
42            else if (PORTD_TEMP == 0b00001100) PORTC = 0b00000011;   // case 3
43            else PORTC = 0b00000000;                                 // case else
44        }
45    }
```



1. Redo the assembly code above following the below table

SW2 RB2	SW1 RB1	SW0 RB0	Led2 RA2	Led1 RA1	Led0 RA0
0	1	1	0	1	1
1	1	0	1	1	0
1	0	1	1	1	1
else			0	0	0

NB: Note that the PORTB and PORTA are analog by default

Test your knowledge



Week 5

Lecture 10 / LO2

Interrupt

Interrupt

- In general, an interrupt is an event that interrupts a main event
- Imagine that you are reading a book, and suddenly you got a call. This call is interrupting your reading.
- What we do exactly is
  1. Stop reading
  2. Memorizing what was read so far
  3. Answer the call
  4. Return back to continue our reading from the same location
- The PIC is able to do the same thing in the same manner.
- When an interrupt event happens, the PIC
  1. stops the main code,
  2. saves the last executed instruction address in stack,
  3. answer the interrupt event,
  4. then return back to the main code



Interrupt

- We, as human, have 5 senses. If we don't hear, we will not be able to get interrupt from phone call. Thus interrupt is based on senses. Similarly, the PIC16F887 has 17 sources of interrupt.

1. External interrupt
2. Interrupt on change
3. Interrupt on Timer zero
4. EEPROM write interrupt
5. Interrupt on Timer 1
6. Interrupt on Timer 2
7. Interrupt on A2D conversion
8. Interrupt on USART receive
9. Interrupt on USART transmit
10. Interrupt on Master synchronous Serial port
11. Int on Capture/Compare/ PWM (CCP1)
12. Int on Capture/Compare/ PWM (CCP2)
13. Oscillator fail interrupt
14. Comparator 1 interrupt
15. Comparator 2 interrupt
16. Bus collision interrupt
17. Low power wake-up interrupt

The first 3 are considered the **major** interrupt sources. The remaining are called **peripheral** interrupt sources



Interrupt

- The interrupt is an event that happens in an unexpected time, but has priority over the main routine.
- The phone call happens in an expected time. We don't check our phone every second, otherwise we won't be able to read. But we have to be prepared for a call at any time.

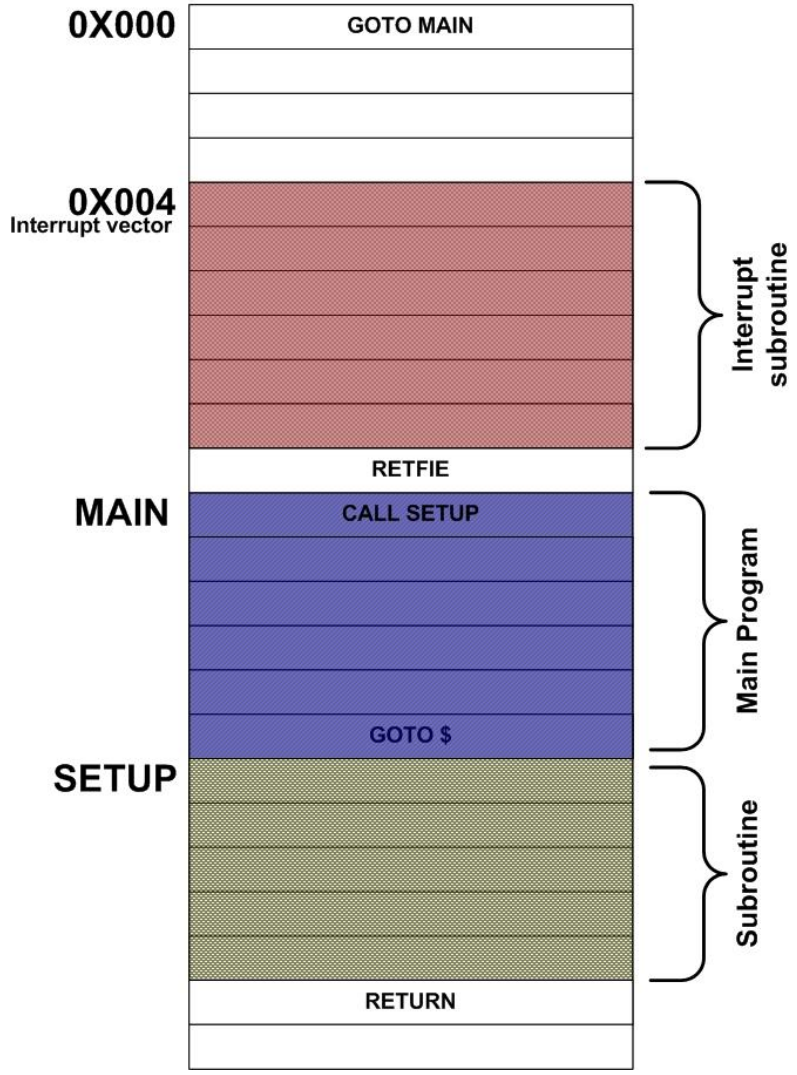
- Similarly, We have to program the PIC to respond to the interrupt event using a special mechanism.

General Mechanism of an Interruption

- We can say that interrupt subroutine is almost similar to any subroutine that you call using CALL instruction.
- The difference is that interrupt subroutine is called by the event itself.
- Therefore, the location of that subroutine should be known before the occurrence of the event, in order to be reached.
- Assume for example that a PIC is dedicated to control an industrial machine and an anti-fire system. When the fire alarm generates an interrupt event, the PIC program counter leaves the machine control process and jump automatically to the program that defied fire. Therefore, this program should be placed in a defined place in order to be treated.

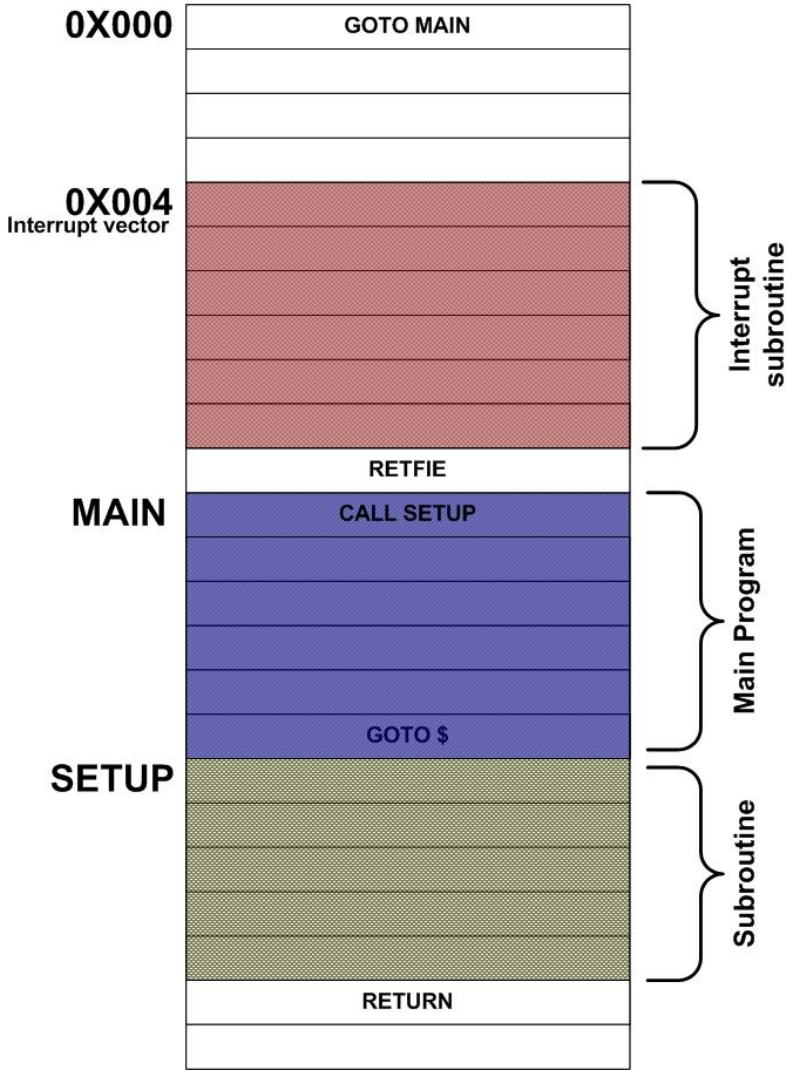
General Mechanism of an Interruption

- The PIC16 family has a fixed place on which it jumps in case of interrupt.
- This point is located in the flash memory, in its fifth address (address number 0x4).
- The address number 0x4 represents the Interrupt Service Routine (ISR) vector, i.e., the beginning of the interrupt function.
- As any subroutine is ended by a RETURN instruction, the interrupt subroutine is also ended by a RETFIE.
- RETFIE is exactly as RETURN but it **re-enables** the global interrupt enable (GIE bit).



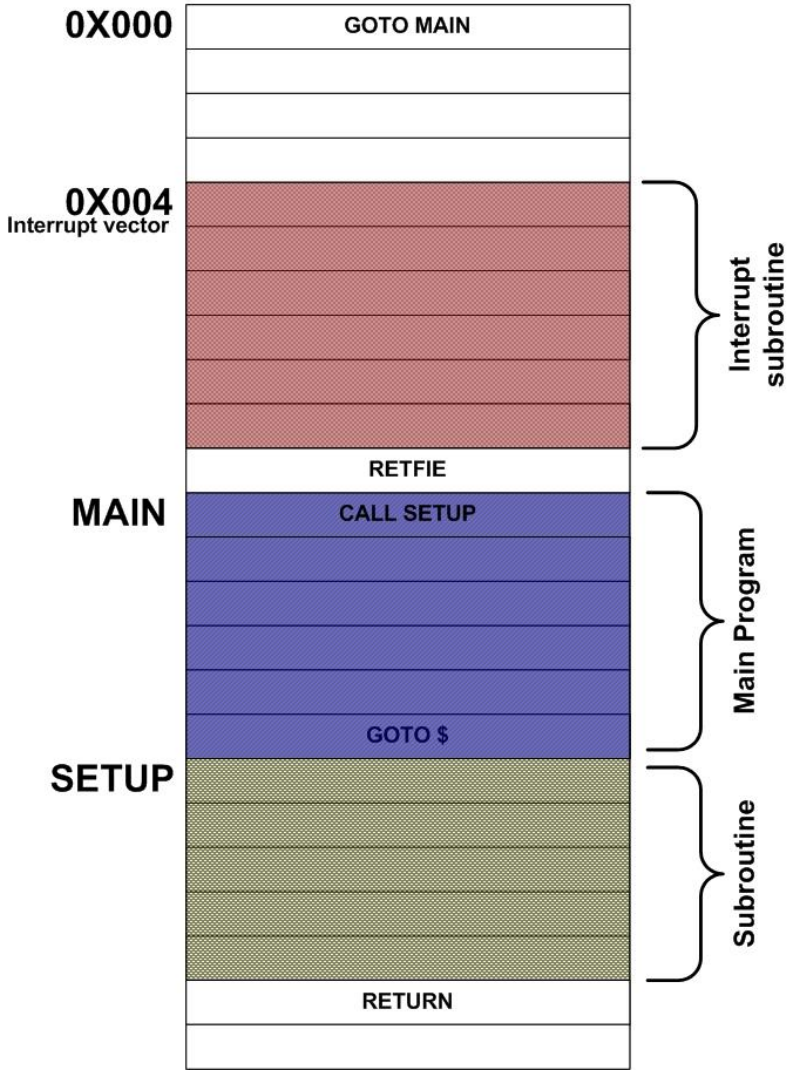
General Mechanism of an Interruption

- The Global Interrupt Enable (GIE) is the bit that allows the PIC to answer any interrupt event.
- If GIE = 0, interruption will not happen, even if the source of interrupt exists. In other word, the PIC will not treat the anti-fire subroutine even if fire exists.
- When starting the process of an interrupt subroutine, the GIE bit is cleared automatically, which means that no new interrupt happened during the process of interrupt subroutine.
- The RETFIE instruction located at the end of the subroutine will **re-enable** the GIE in order to monitor again the interrupt sources.



General Mechanism of an Interruption

- One of the weakness in the PIC16 family is that all interrupts generated by the 17 sources (if enabled) have the same interrupt vector 0x04. In other words all interrupt processes should be written starting address 0x04.
- Thus, to identify what is the source of interrupt, a flag should be tested first.
- For example, if we have two interrupts (one from the fire sensor and another from a water leakage sensor) both will interrupt the main code and jump to 0x04. Here, the flag of the interrupt should be tested to know the origin interrupt.



Interrupt registers

- The 17 sources of interrupt are managed through 3 registers. The main register is called INTCON. It includes the Global Interrupt Enable (GIE) bit, The Peripheral enable bit (PEIE), and three major interrupt enable bits with their flags.
- PEIE is the enable bit of all peripheral sources that are enabled through PIE1 and PIE2 registers

REGISTER 2-3: INTCON: INTERRUPT CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE(1)	T0IF(2)	INTF	RBIF
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **PEIE:** Peripheral Interrupt Enable bit
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **T0IE:** Timer0 Overflow Interrupt Enable bit
1 = Enables the Timer0 interrupt
0 = Disables the Timer0 interrupt

bit 4 **INTE:** INT External Interrupt Enable bit
1 = Enables the INT external interrupt
0 = Disables the INT external interrupt

bit 3 **RBIE:** PORTB Change Interrupt Enable bit(1)
1 = Enables the PORTB change interrupt
0 = Disables the PORTB change interrupt

bit 2 **T0IF:** Timer0 Overflow Interrupt Flag bit(2)
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF:** INT External Interrupt Flag bit
1 = The INT external interrupt occurred (must be cleared in software)
0 = The INT external interrupt did not occur

bit 0 **RBIF:** PORTB Change Interrupt Flag bit
1 = When at least one of the PORTB general purpose I/O pins changed state (must be cleared in software)
0 = None of the PORTB general purpose I/O pins have changed state



Interrupt registers

- PIE1 and PIE2 includes the Enable bits of the remaining 14 peripheral interrupt sources.

REGISTER 2-4: PIE1: PERIPHERAL INTERRUPT ENABLE REGISTER 1

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

REGISTER 2-5: PIE2: PERIPHERAL INTERRUPT ENABLE REGISTER 2

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
OSFIE	C2IE	C1IE	EEIE	BCLIE	ULPWUIE	—	CCP2IE
bit 7							bit 0

- The flags of these sources are available in two registers PIR1 and PIR2.

REGISTER 2-6: PIR1: PERIPHERAL INTERRUPT REQUEST REGISTER 1

U-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
—	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

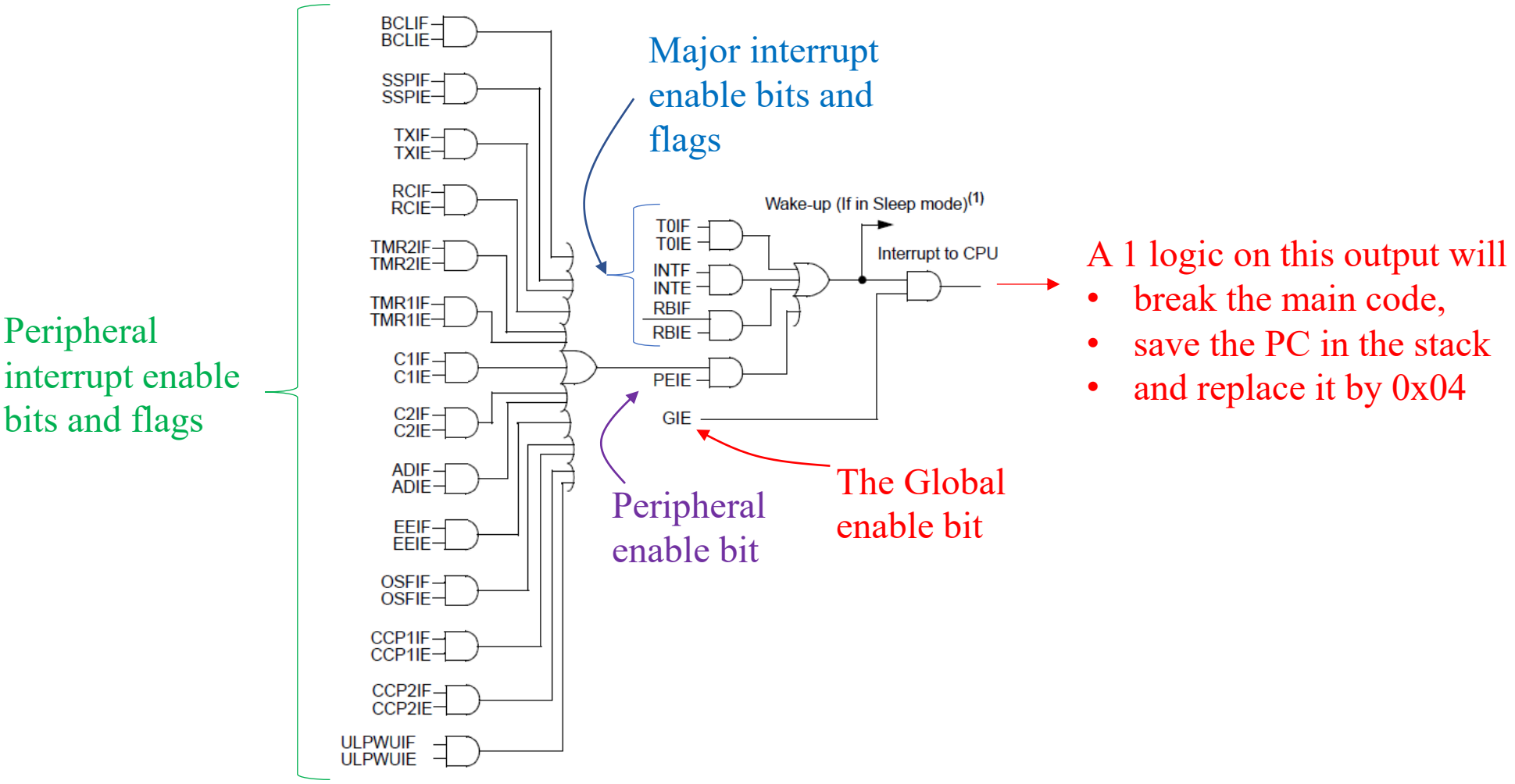
REGISTER 2-7: PIR2: PERIPHERAL INTERRUPT REQUEST REGISTER 2

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
OSFIF	C2IF	C1IF	EEIF	BCLIF	ULPWUIF	—	CCP2IF
bit 7							bit 0

All enable and flag bits are aligned

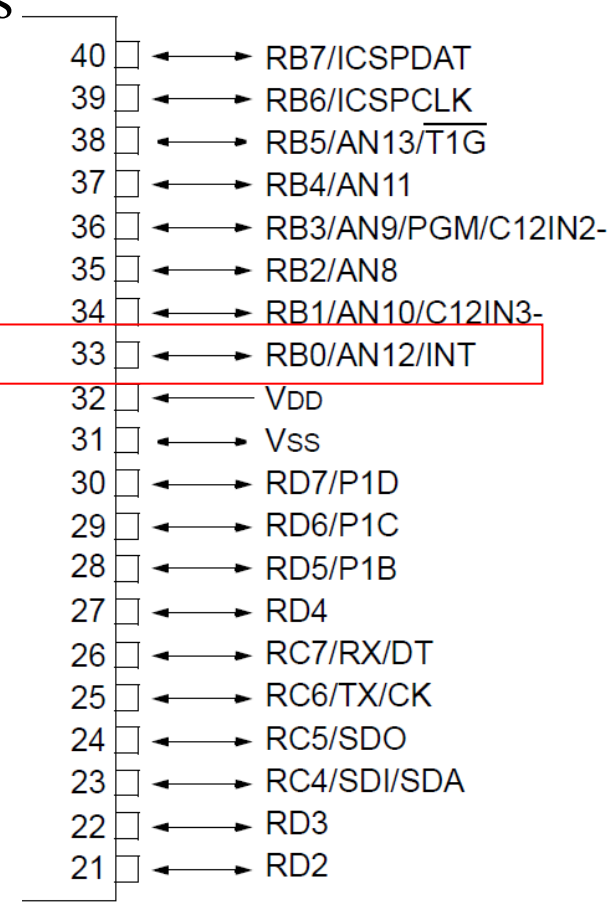
Interrupt logic circuit

- The internal schematic of the interrupt circuit is presented below.



Activity 8: Interrupt on RB0

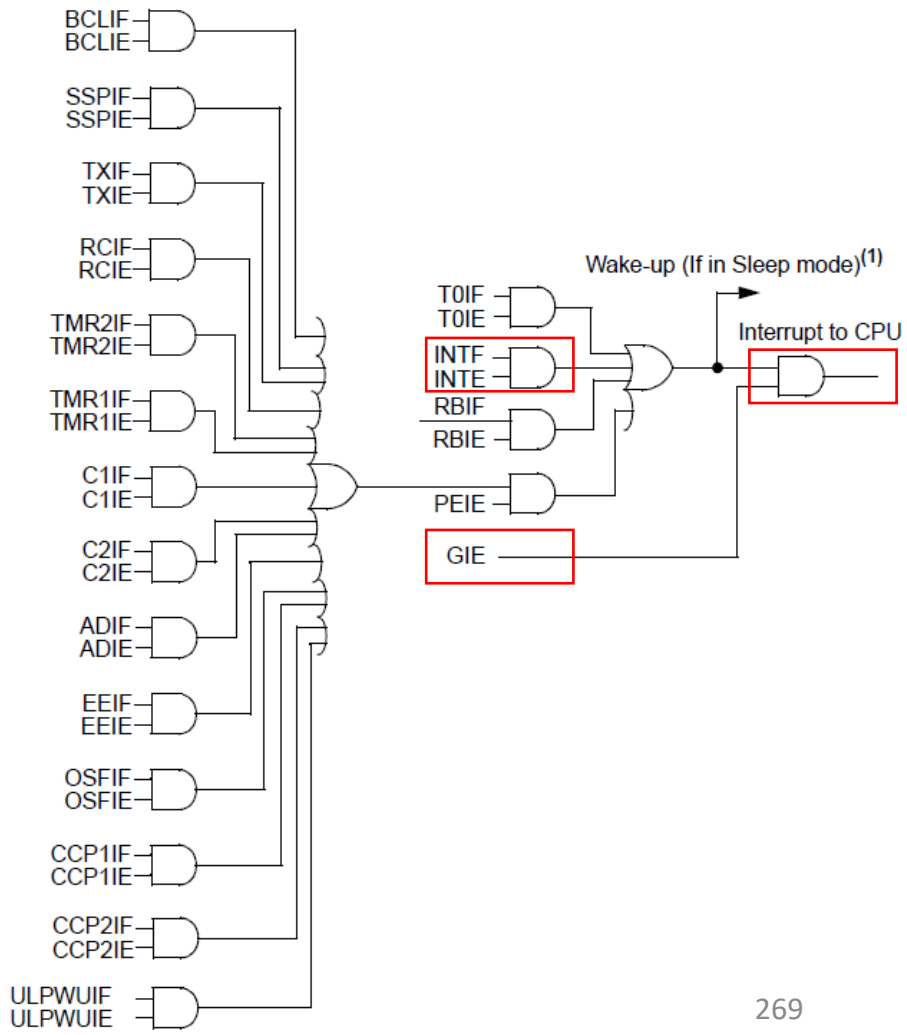
- In this activity, we will apply interrupt on RB0/INT. This is an external interrupt, i.e., the event is triggered when an external pulse is applied on this pin.
- For the moment, we will assume that the pulse has a rising edge.
- 4 conditions are required to have interrupt on this pin
  1. RB0 should be digital input
  2. Enable this interrupt source INTE = 1
  3. Enable the global interrupt GIE = 1
  4. Wait for external pulse (Rising edge by default)
- Thus we have to prepare the 3 first conditions and wait for the 4th to happen from a push button or from a sensor.



Activity 8: Interrupt on RB0

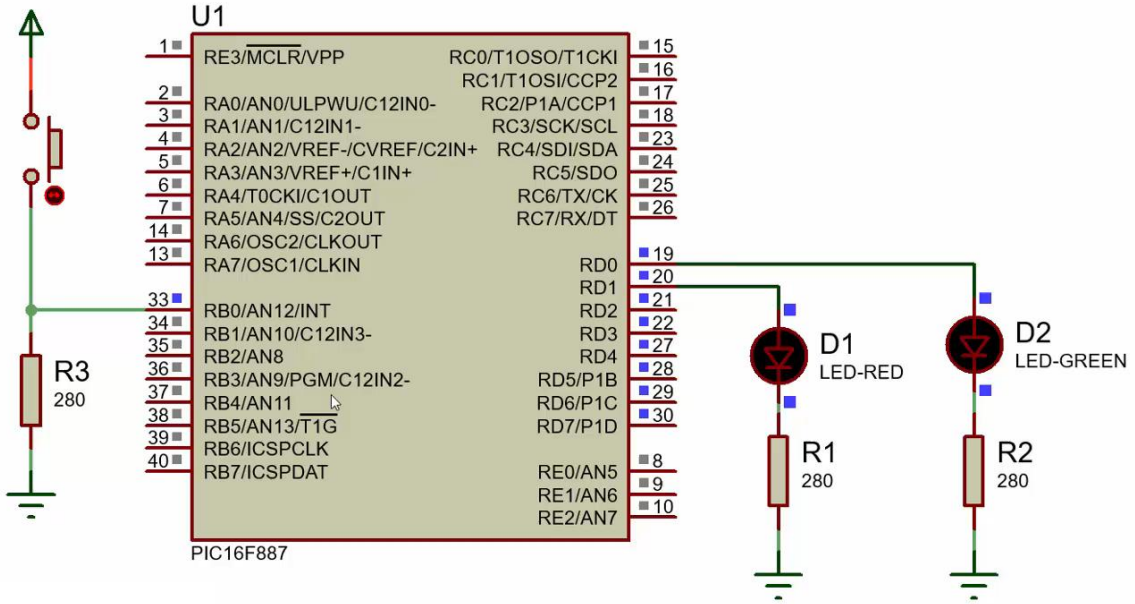
- When all these conditions are true, a flag INTF is set (INTF=1) automatically.
- Having INTE = 1, INTF = 1 and GIE = 1, the CPU will generate an interrupt event.

- The interrupt event is described as:
  1. Leaving the main code
  2. Disabling GIE bit (automatically)
  3. Going to address 0x04 (The interrupt vector)
  4. Execute the code of the interrupt service routine
  5. Return back to the main code



Activity 8: Interrupt on RB0

- In this activity, we will blink a red led in the main code with a frequency of 1Hz, and we will connect a push button on RB0 to create a pulse. When the interrupt event is triggered we will turn on a green led.
- Remember that we have to setup the 3 conditions first
  1. RB0 should be input (using TRISB0)
  2. Enable this interrupt source INTE = 1
  3. Enable the global interrupt GIE = 1



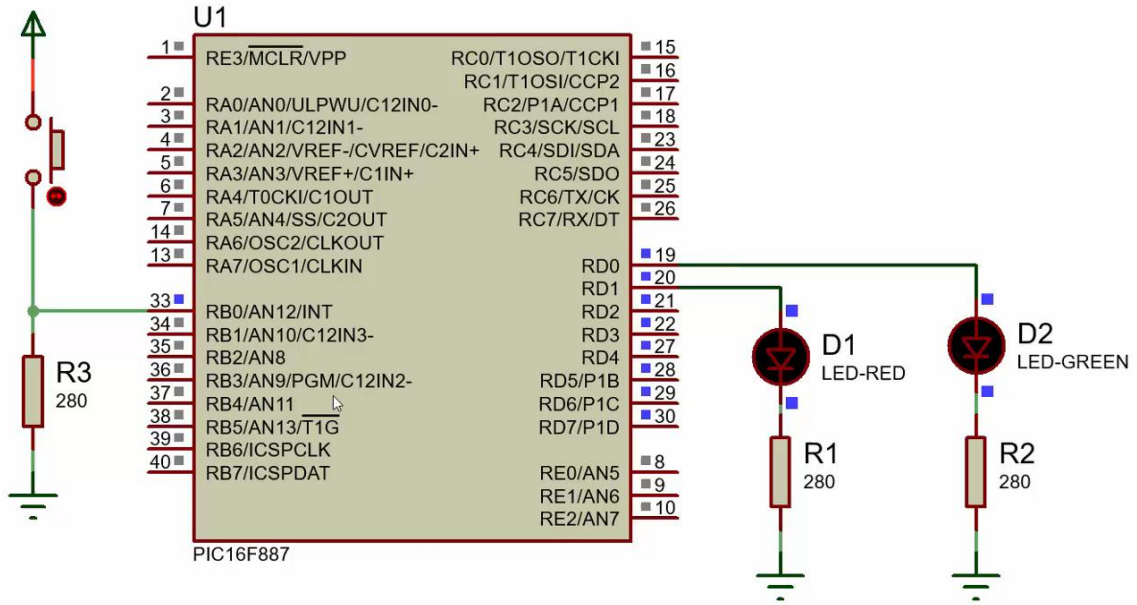
REGISTER 2-3: INTCON: INTERRUPT CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE(1)	T0IF(2)	INTF	RBIF
bit 7							bit 0

Activity 8: Interrupt on RB0

- When an external pulse is applied through the push button, the INTF will be set.
- And the Program counter (PC) is loaded by 0x04
- At this particular address, we have to write code to activate the green led.

- NOTE: the push button is not tested using BTFSS or BTFSC. The event will happen automatically



REGISTER 2-3: INTCON: INTERRUPT CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE(1)	T0IF(2)	INTF	RBIF
bit 7							bit 0

Activity 8: Interrupt on RB0

- RB0 is set to input in the SETUP function. Then GIE and INTE are set at line 43 and 44.
- The ISR starts at line 37 which is at address 0x04. First thing to do **every time** is to clear the flag.
- At the end of the ISR, the instruction RETIFE is used. This will reactivate GIE because it was automatically disactivated when jumping to ISR. This means that no interrupt is allowed inside the ISR. **(No interrupt on interrupt)**

```
34        ORG     0             ; reset vector
35        GOTO    MAIN
36        ORG     0X4           ; interrupt vector
37        BCF     INTF          ; clear the flag first
38        BSF     LED_GREEN     ; turn on green led
39        RETFIE
40
41    MAIN:
42        CALL    SETUP
43        BSF     GIE           ; required condition
44        BSF     INTE          ; required condition
45    REPEAT:                   ; blink red led
46        BSF     LED_RED
47        CALL    DELAY_500MS
48        BCF     LED_RED
49        CALL    DELAY_500MS
50        GOTO    REPEAT        ; REPEAT CODE
```

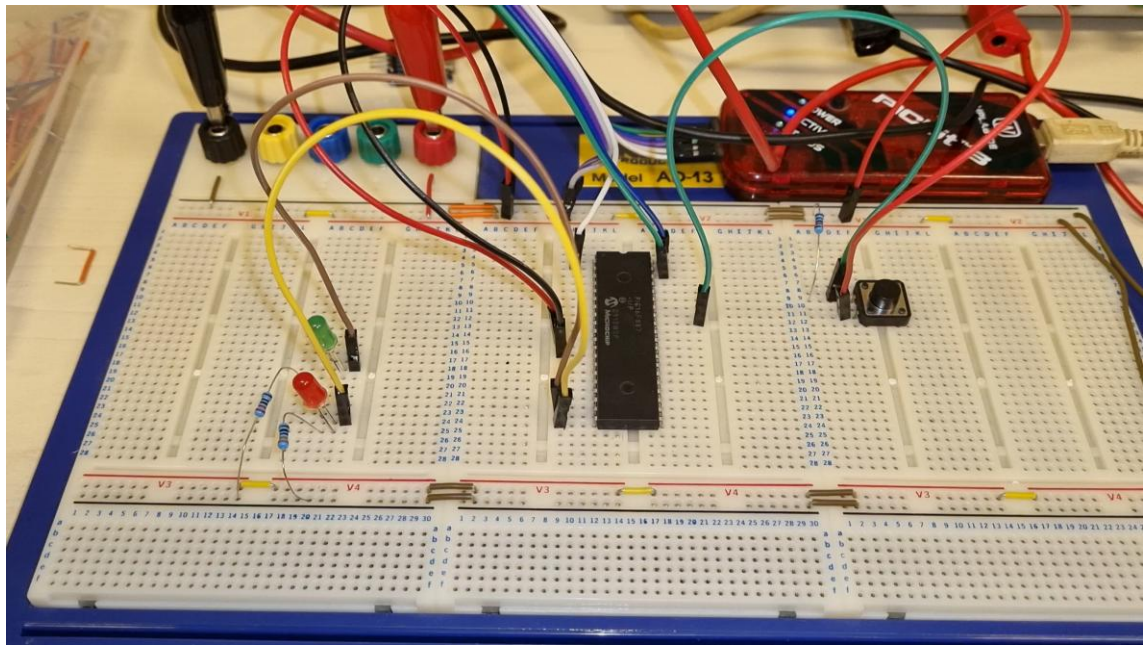
This function is executed when the PB is pressed

```
52    SETUP:                    ; by default we are in BANK0
53        CLRF    PORTD         ; clear port at the beginning
54        BSF     RP0           ; RP1 IS BY DEFAUT 0
55        MOVLW   01100000B     ; configuring OSCCON register
56        MOVWF   OSCCON        ; to select 4Mhz oscillator
57        MOVLW   11111111B     ; RB0 is input (required condition)
58        MOVWF   TRISB
59        MOVLW   00000000B     ; portD is out
60        MOVWF   TRISD
61        BSF     RP1
62        CLRF    ANSELH
63        BCF     RP0           ; access to BANK 0, RP0 = 0 , RP1 = 0
64        BCF     RP1
65        RETURN
```

Activity 8 C: Interrupt on RB0

- The same Activity is also designed in C.
- The __interrupt() function is the replacement of the ORG 0x04.
- my_isr_routine() can be replaced by any name

This function is executed when the PB is pressed



```
28  void __interrupt() my_isr_routine() {
29      INTF = 0; // clear the flag
30      led_green = 1; // led on
31  }
32
33  void setup() {
34      PORTD = 0; // clear port
35      PORTB = 0;
36      OSCCON = 0b01100000; // select 4MHz oscillator
37      TRISB = 0b11111111; // define port as in
38      TRISD = 0b00000000; // and as out
39      ANSELH = 0; // PORTB digital
40  }
41
42  void main(void) {
43      setup();
44      GIE = 1;
45      INTE = 1;
46      while (1) {
47          led_red = 1;
48          __delay_ms(500);
49          led_red = 0;
50          __delay_ms(500);
51      }
52  }
```

Activity 8 C: Interrupt on RB0

- After doing this activity in assembly and C, the generated hex files are compared as below.
- It is clearly obvious that the C compiler in doing hidden instructions.
- In fact the Compiler is programmed to generate extra instructions that may be used in more advanced task. In this activity, the main task is just to blink and the ISR task is just to turn ON the led, and no need for the extra instructions.
- The extra instructions will be treated in the next activities

Hex file generated by the C compiler

```
:060000000A128A110C280F
:10000800FE0030EF0000A08F1000A128A114D28BA
:10001800A128A110F2883010A128A1113280A1258
:10002800A113E200A128A118B170B1683120313AA
:10003800881403030F4008A30F3005530F200F20BD4
:10004800232F30B2328F40B23282A2883120313CD
:10005800881003030F4008A30F3005530F200F20BB8
:10006800332F30B3328F40B33283A281A280A12BA
:10007800A110C2883120313880186016030831 6C5
:10008800313F00FF3086008801831603178901 48
:10009800808B10831203130814710 8A00700E6D
:0800A8008300FE0E7E0E09002C
:04400E00D420FF3F7C
:00000001FF
```

Hex file generated by the PIC-as assembler

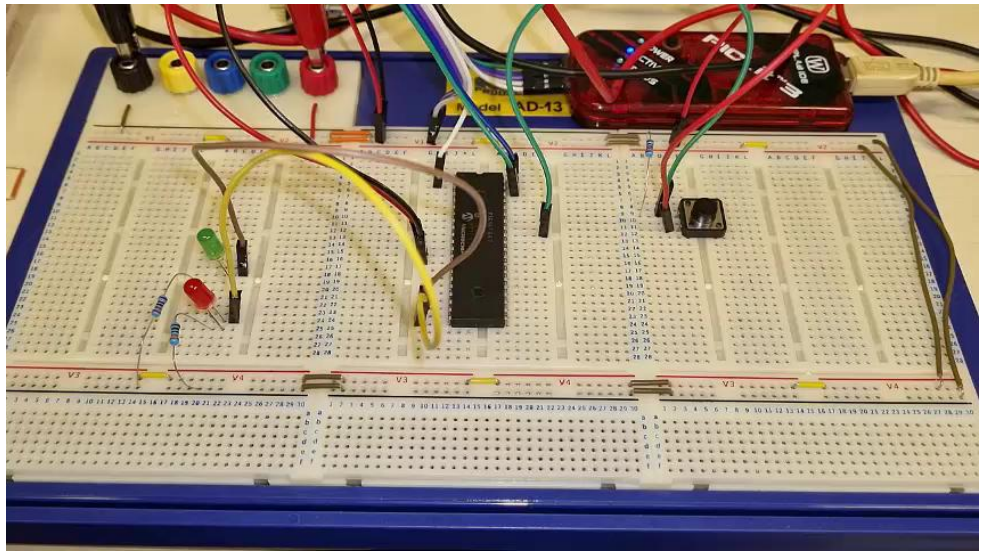
```
:020000000828CE
:10000800 8B10081488150900102 08B170B168814FC
:10001800 1D2088101D200B2888018316603 08F0052
:10002800FF308600003088000317890183120313 0C
:10003800080054 30F0008A30F1000330F200F00B71
:0E004800232 8F10B2328F20B232800000800C8
:04400E00D420FF3F7C
:00000001FF
```

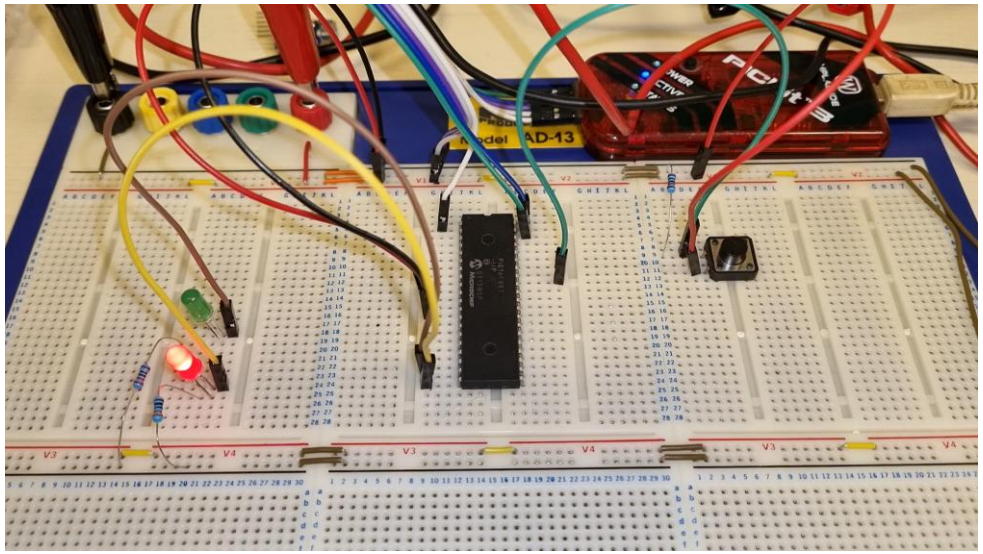
Activity 9: Interrupt on RB0/ Effect off common variables

- Activity 9 is a modified version of Activity 8. The only difference is to turn off the green led after 500ms. Thus when the PB is pressed, the led will turn on for 500ms then off. The same code is written in assembly and C.

```
ORG     0X4           ; interrupt vector
BCF     INTF          ; clear the flag first
BSF     LED_GREEN     ; turn on green led
CALL    DELAY_500MS   ; FOR 500MS
BCF     LED_GREEN     ; THEN OFF
RETFIE
```

```
void __interrupt() my_isr_routine() {
    INTF = 0; // clear the flag
    led_green = 1; // led on
    __delay_ms(500); // for 500 ms
    led_green = 0; // then off
}
```





Activity 9: Interrupt on RB0/ Effect off common variables

- It is clear from the videos that the two similar codes lead to different results. The C code is showing a logical output compared to the assembly code that freezes for about 50 second somewhere in the loop.

```
ORG     0X4           ; interrupt vector
BCF     INTF          ; clear the flag first
BSF     LED_GREEN     ; turn on green led
CALL    DELAY_500MS   ; FOR 500MS
BCF     LED_GREEN     ; THEN OFF
RETFIE
```

```
void __interrupt() my_isr_routine() {
    INTF = 0; // clear the flag
    led_green = 1; // led on
    __delay_ms(500); // for 500 ms
    led_green = 0; // then off
}
```

- This malfunctioning of the assembly code is due to the common delay function that is called from the main code and from the interrupt code.
- When the PB is pressed, the delay function is interrupted with arbitrary values of A1, A2, A3 that are the variables of the delay function. In the ISR, the same delay is used and the variables are loaded with new values and decremented until reaching zeros (the condition that terminated the delay function). When returning back to the main code, more precisely to the main code, the variables will be loaded by different values of the one before interrupt
- Thus, as a conclusion, the SHARED VARIABLES are the reason if the mal functioning of the program.

Activity 9: Interrupt on RB0/ Effect off common variables

- The C compiler is smart by creating different variables for the delay function used in the main code and the delay used in the ISR.
- The assembly code is then modified to have two delay functions with different variables.

```
29      A1 EQU 0X70
30      A2 EQU 0X71
31      A3 EQU 0X72
32      B1 EQU 0X73
33      B2 EQU 0X74
34      B3 EQU 0X75
```

```
39      ORG     0               ; reset vector
40      GOTO    MAIN
41      ORG     0X4             ; interrupt vector
42      BCF     INTF            ; clear the flag first
43      BSF     LED_GREEN       ; turn on green led
44      CALL    DELAY_B_500MS           ; FOR 500MS
45      BCF     LED_GREEN       ; THEN OFF
46      RETFIE
47
48  MAIN:
49      CALL    SETUP
50      BSF     GIE             ; required condition
51      BSF     INTE            ; required condition
52  REPEAT:                     ; blink red led
53      BSF     LED_RED
54      CALL    DELAY_A_500MS
55      BCF     LED_RED
56      CALL    DELAY_A_500MS
57      GOTO    REPEAT          ; REPEAT CODE
```

```
75  DELAY_A_500MS:
76      MOVLW   0X54
77      MOVWF   A1
78      MOVLW   0X8a
79      MOVWF   A2
80      MOVLW   0X03
81      MOVWF   A3
82      DECFSZ  A1, 1
83      GOTO    $-1
84      DECFSZ  A2, 1
85      GOTO    $-3
86      DECFSZ  A3, 1
87      GOTO    $-5
88      NOP
89      RETURN
```

```
91  DELAY_B_500MS:
92      MOVLW   0X54
93      MOVWF   B1
94      MOVLW   0X8a
95      MOVWF   B2
96      MOVLW   0X03
97      MOVWF   B3
98      DECFSZ  B1, 1
99      GOTO    $-1
100     DECFSZ  B2, 1
101     GOTO    $-3
102     DECFSZ  B3, 1
103     GOTO    $-5
104     NOP
105     RETURN
```



1. What is the difference between RETFIE and RETURN
2. The Interrupt vector address is at the 5th memory starting from address 0 (T/F)
3. In the ISR, the GIE is automatically disabled (T/F)
4. What should be cleared manually at the beginning of the ISR ?

Test your knowledge



Week 5

Lab 4 / LO2

Interrupt on RB0



Activity 8 C: Interrupt on RB0

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. PIC programmer
  4. Power supply 5V
  5. Breadboard wires (male-male)

- Required components:
  1. PIC 16F887
  2. Led x1
  3. Push button for breadboard
  4. Resistor from 200 to 470 ohm x 1
  5. Resistor from 1k to 10k ohm x 1

End of LO2

LO3



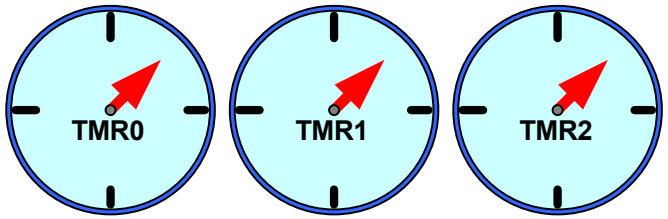
Week 6

Lecture 11 / LO3

Timers/ Timer 0

Timers

- The PIC16F887 has three timers, called Timer 0, Timer 1 and Timer 2.
- You can imagine these timers as wall clock with only one arrow.
- Timer 0 and Timer 2 count from 0 to 255 (same as clock counts from 0 to 23), i.e., these timers are 8bits
- Timer 1 counts from 0 to 65535, i.e., 16bits

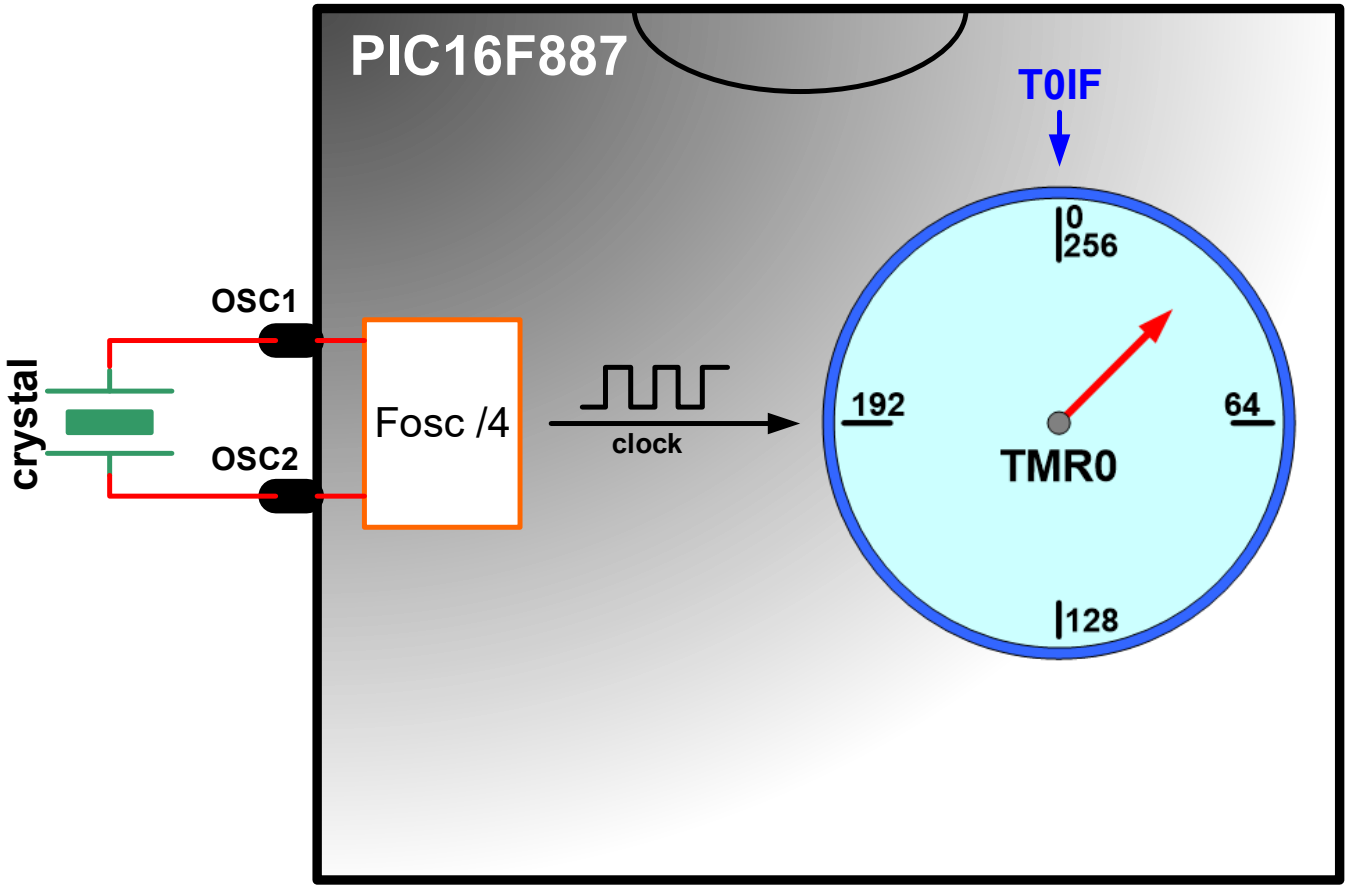


- Count means counting the pulses of the oscillator, thus counting the time as we know the period of the pulses.

Device	Program Memory	Data Memory		I/O	10-bit A/D (ch)	ECCP/ CCP	EUSART	MSSP	Comparators	Timers 8/16-bit
	Flash (words)	SRAM (bytes)	EEPROM (bytes)							
PIC16F887	8192	368	256	35	14	1/1	1	1	2	2/1

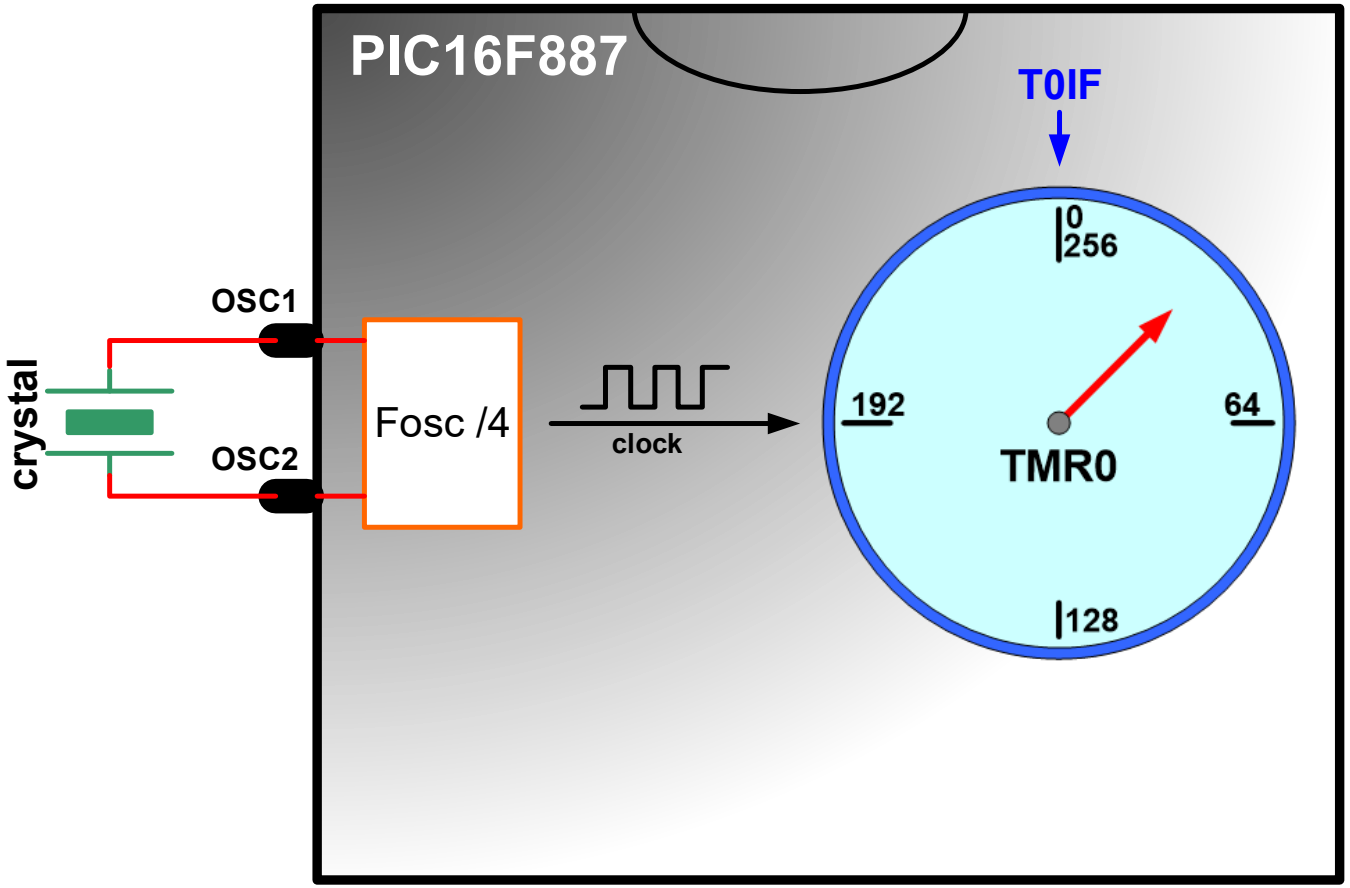
Timer Zero

- Timer0 is an 8-bit counter. It counts from 0 to 255.
- Actually, timer0 counts the pulses generated by the internal or external oscillator, divided by 4, i.e., Each four clocks increment the timer by 1.
- In other words, the speed of TMR0 is four times less than the oscillator speed.
- If the used oscillator is 4 MHz, the timer0 runs at 1 MHZ.
- This means that the timer increments every 1µs



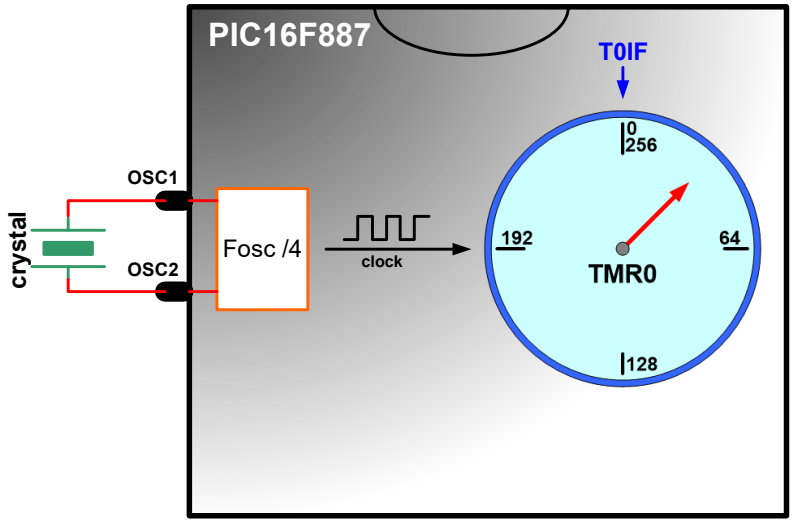
Timer Zero

- When the arrow reaches the position of 256 (timer overflow), a flag is raised automatically. This is like an alarm in a real clock.
- This flag is called Timer0 interrupt flag or T0IF
- This flag can be used with or without interrupt.
- This flag should be cleared in software every time is raised.



Timer Zero

- Unfortunately, Timer0 has no special configuration register to control it.
- The register used to configure this timer is called OPTION_REG which is also used for different setup.
- Only bit 0 to bit 5 are used to configure the timer. Other bits have another duties (to be treated later).
- Also, this timer cannot be stopped. It has no ON/OFF control line. It is always running whenever the PIC is powered.



REGISTER 5-1: OPTION_REG: OPTION REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 5 **T0CS:** TMR0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (Fosc/4)

bit 4 **T0SE:** TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

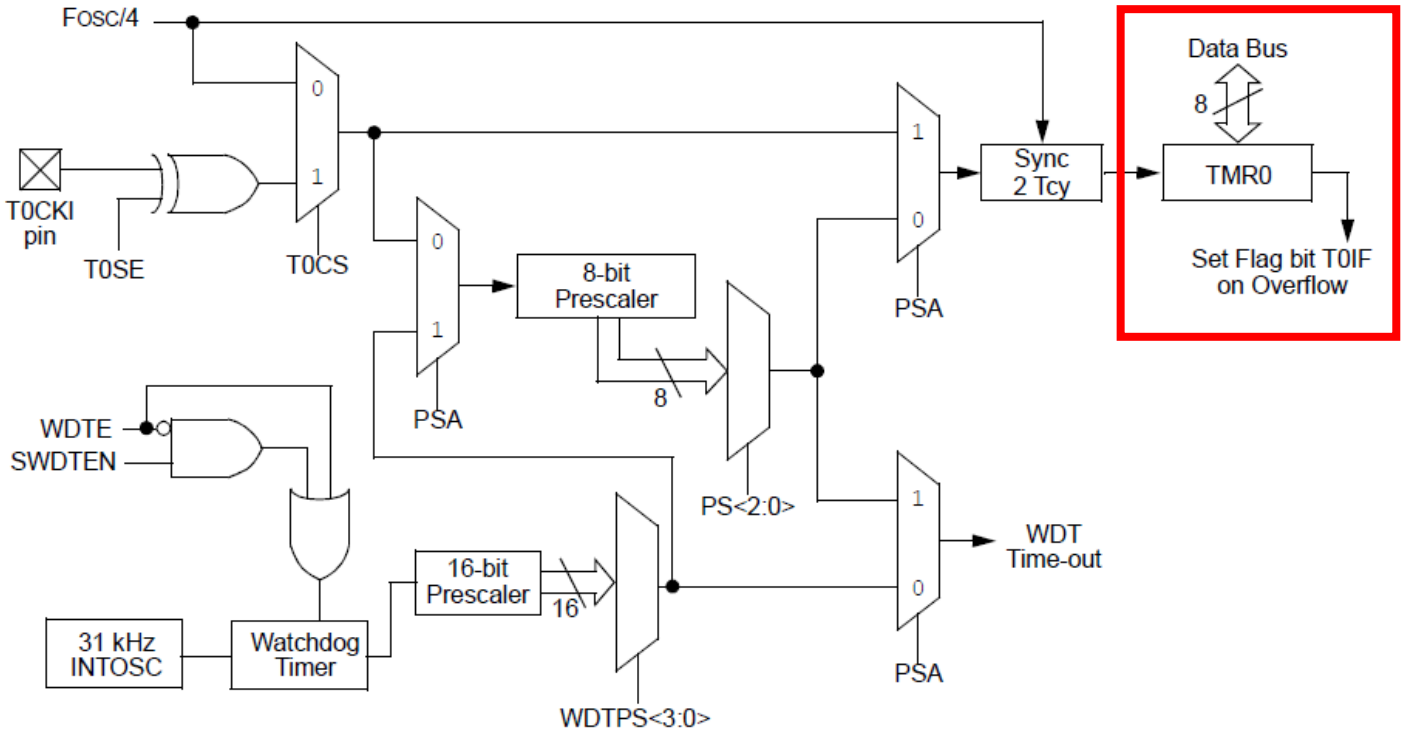
bit 3 **PSA:** Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

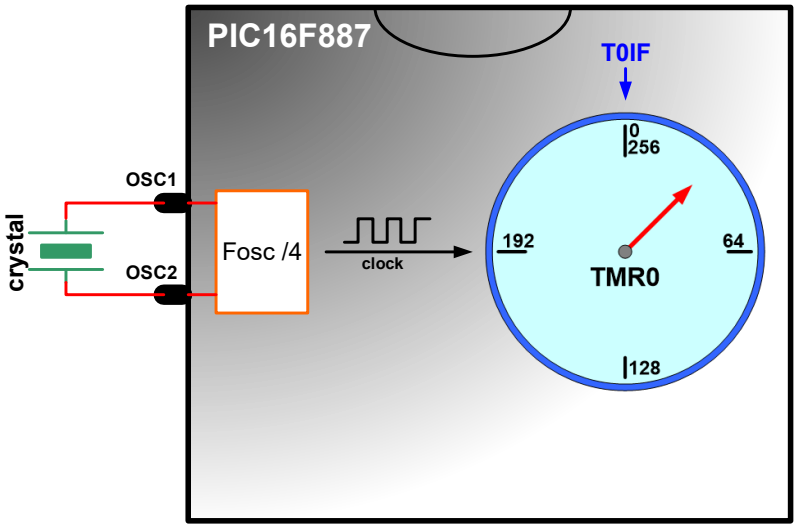
bit 2-0 **PS<2:0>:** Prescaler Rate Select bits

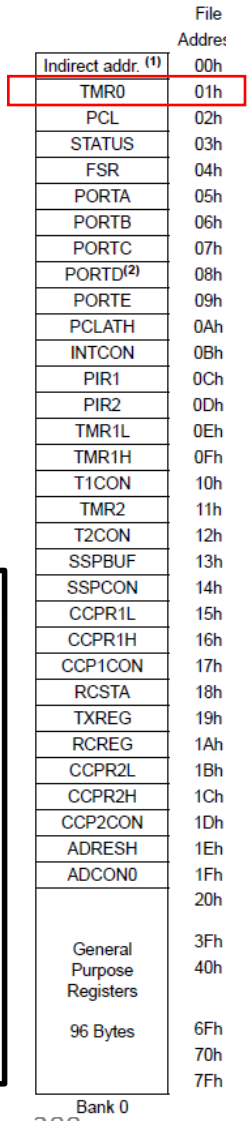
BIT VALUE	TMR0 RATE	WDT RATE
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Timer Zero

- In reality, timer0 is nothing more than an auto-incrementing register called TMR0 located in bank 0 of the RAM. When the TMR0 overflows, a carry bit is set… which is called flag (T0IF)

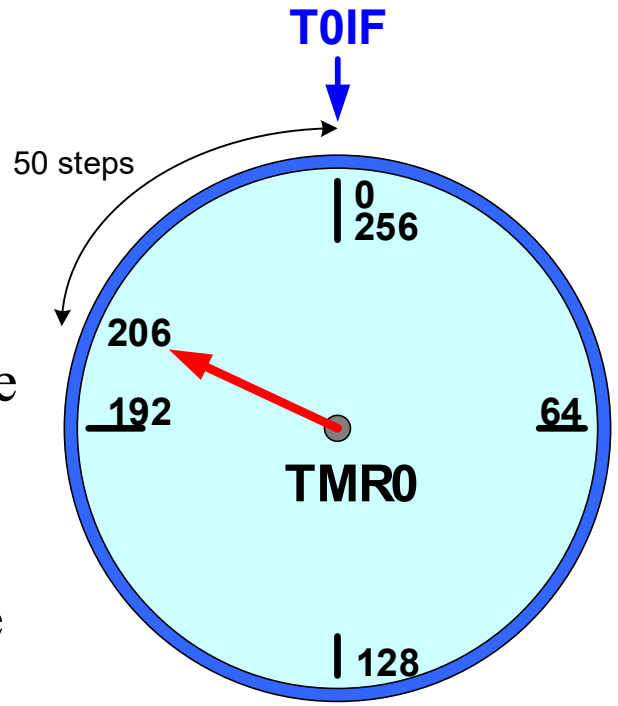






Activity 10: Signal generation using Timer 0

- In this activity, we will be using Timer0 to generate a square wave signal on RD0 at a frequency of 10Khz.
- For F=10Khz, T = 1/10000 = 10^{-4} s = 100µs, i.e., 50µs ON and 50µs OFF.
- Thus it is required to tune our timer to generate a flag (the alarm) after 50µs after which we toggle the RD0 state.
- In this example, we will assume that the oscillator is running at 4Mhz, hence the timer is incremented every 1µs
- Therefore, to count 50µs , it is required to place the timer arrow at 256-50 which is 206. After 1µs the arrow will be at 207… and so on after 50µs , the alarm will be triggered.

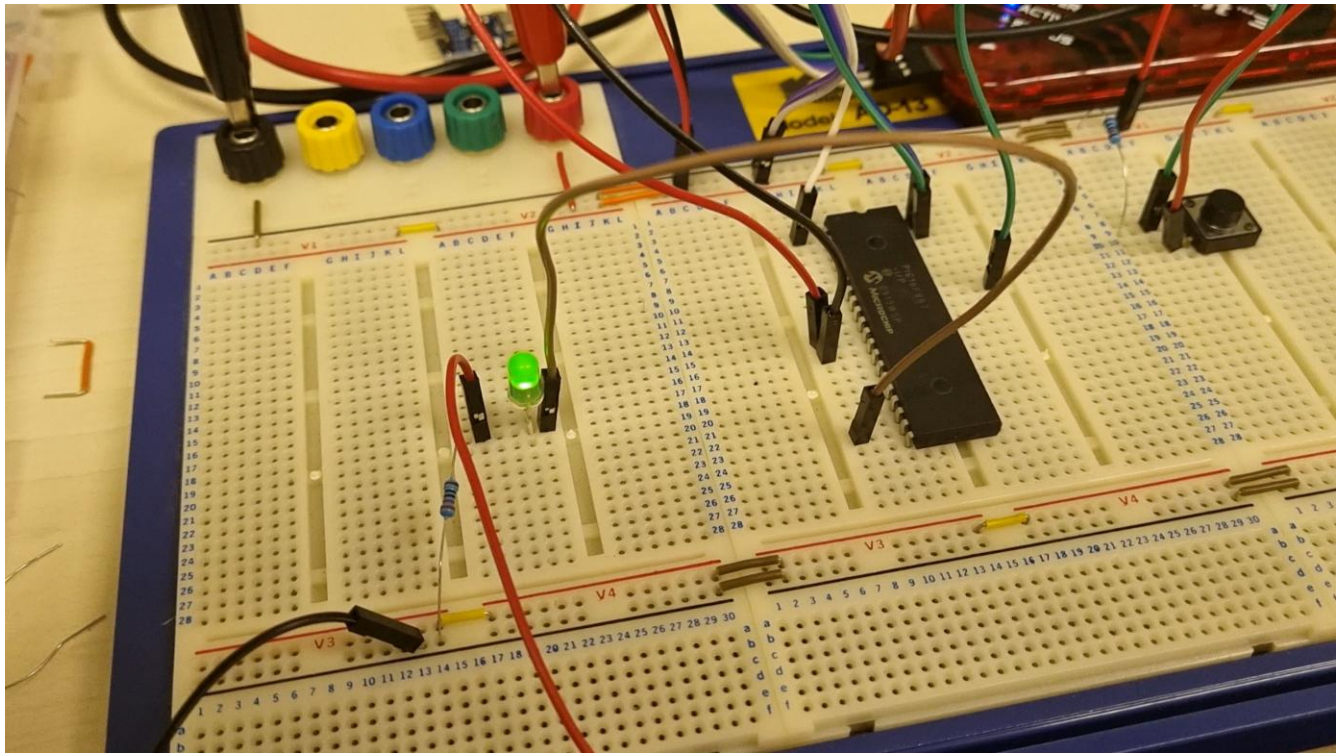


Thus the register TMR0 should be loaded by 256 - ”the required time”

And as 256 = 0 , TMR0 could be loaded by 0 - ”the required time” = - ”the required time”

Activity 10: Signal generation using Timer 0

- Based on the experimental results, the led is not showing any blinking effect. In fact it is blinking but human eyes cannot detect more than 20 ON/OFF per second (20Hz)
- The second remark is that the measured frequency is not 10KHz as expected. This is due to some instruction execution time loss inside the loop. The TMR0 is tuned experimentally to be -39



Activity 10: Signal generation using Timer 0

- **The code works as follow:**
- 1: In the setup, at line 52, the OPTION_REG (in bank 1) is loaded by xx011000. the first 2 bits are not dedicated for timer 0 setup. The other bits will be explained later
- 2: T0IF is cleared at the beginning , 3: TMR0 register is loaded by -39; 4: The code keeps on testing the alarm (T0IF), if set, the RD0 is toggle using the XOR gate. The code is repeated to generate the next state

REGISTER 5-1: OPTION_REG: OPTION REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 5 **T0CS:** TMR0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (FOSC/4)

bit 4 **T0SE:** TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA:** Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

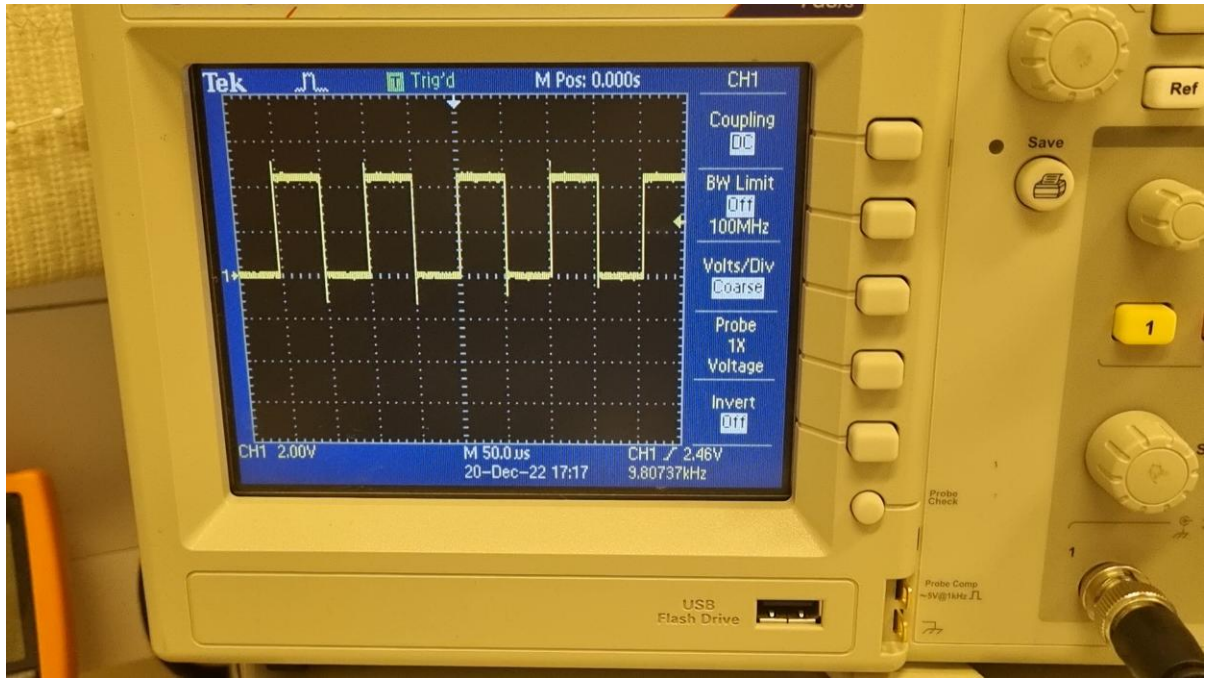
bit 2-0 **PS<2:0>:** Prescaler Rate Select bits

BIT VALUE	TMR0 RATE	WDT RATE
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

```
33  MAIN:
34      CALL    SETUP
35  REPEAT:                 ; blink red led
36      BCF     T0IF        ; clear the flag
37      MOVLW   -39         ; load TMR0 with 217 to count 39 steps
38      MOVWF   TMR0
39      BTFSS   T0IF        ; wait the flag
40      GOTO    $-1
41      MOVLW   00000001B   ; toggling RD0
42      XORWF   PORTD, F    ; using XOR
43      GOTO    REPEAT      ; REPEAT CODE
44
45  SETUP:                  ; by default we are in BANK0
46      CLRF    PORTD       ; clear port at the beginning
47      BSF     RP0         ; RP1 IS BY DEFAUT 0
48      MOVLW   01100000B   ; configuring OSCCON register
49      MOVWF   OSCCON      ; to select 4Mhz oscillator
50      MOVLW   00000000B   ; portD is out
51      MOVWF   TRISD
52      MOVLW   11011000B;  ; no prescaller used
53      MOVWF   OPTION_REG
54      BCF     RP0         ; access to BANK 0, RP0 = 0 , RP1 = 0
55      RETURN
```

Activity 10: Signal generation using Timer 0

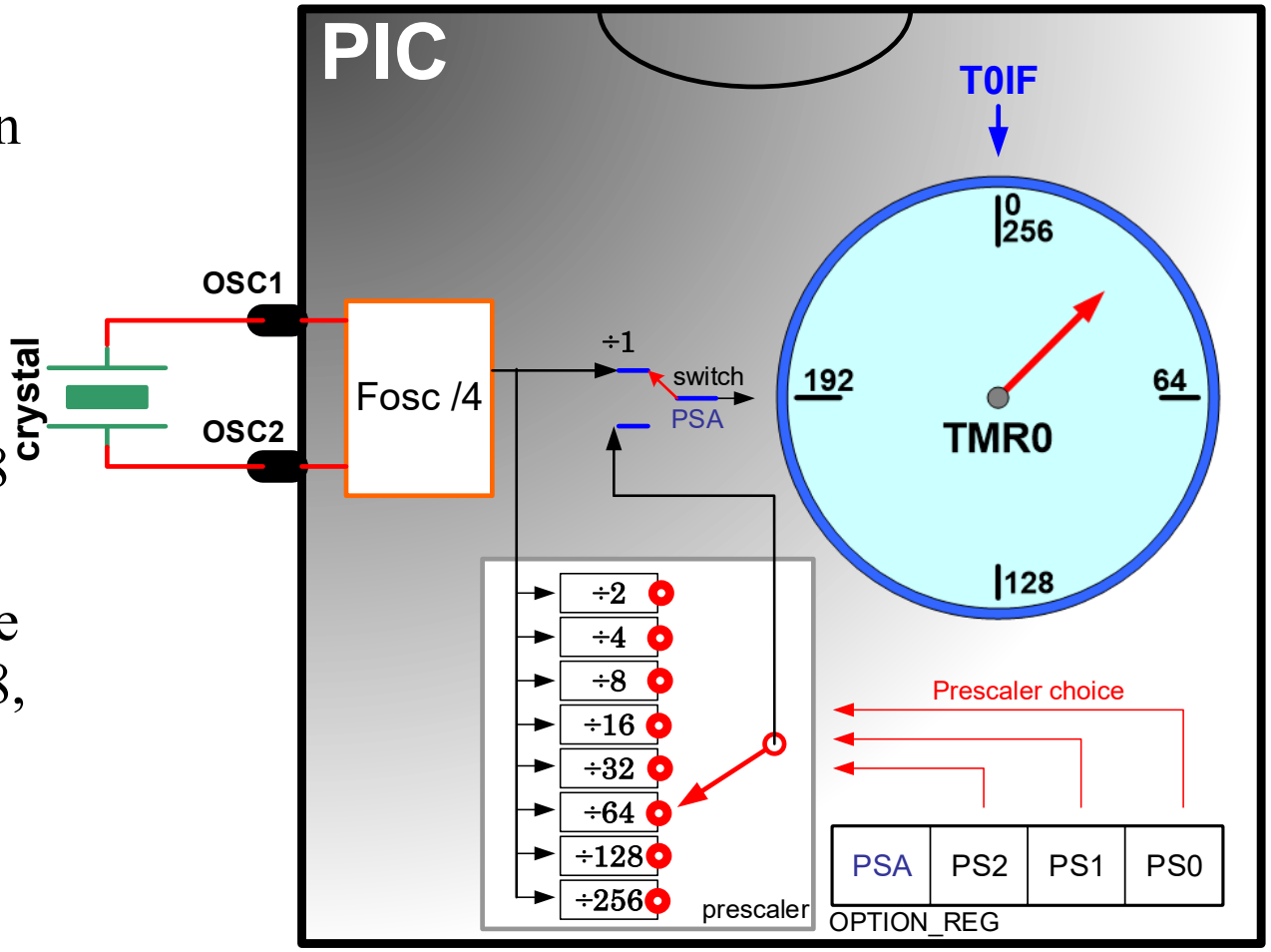
- As seen in the code that some instructions are involved in the loop and thus contributed in few microseconds loss. The TMR0 is loaded by 39 to compensate this loss, although the result is not perfect at high frequency.




```
33  MAIN:
34      CALL    SETUP
35  REPEAT:                 ; blink red led
36      BCF     T0IF        ; clear the flag
37      MOVLW   -39         ; load TMR0 with 217 to count 39 steps
38      MOVWF   TMR0
39      BTFSS   T0IF        ; wait the flag
40      GOTO    $-1
41      MOVLW   00000001B   ; toggling RD0
42      XORWF   PORTD, F    ; using XOR
43      GOTO    REPEAT      ; REPEAT CODE
44
45  SETUP:                  ; by default we are in BANK0
46      CLRF    PORTD       ; clear port at the beginning
47      BSF     RP0         ; RP1 IS BY DEFAUT 0
48      MOVLW   01100000B   ; configuring OSCCON register
49      MOVWF   OSCCON      ; to select 4Mhz oscillator
50      MOVLW   00000000B   ; portD is out
51      MOVWF   TRISD
52      MOVLW   11011000B;  ; no prescaller used
53      MOVWF   OPTION_REG
54      BCF     RP0         ; access to BANK 0, RP0 = 0 , RP1 = 0
55      RETURN
```

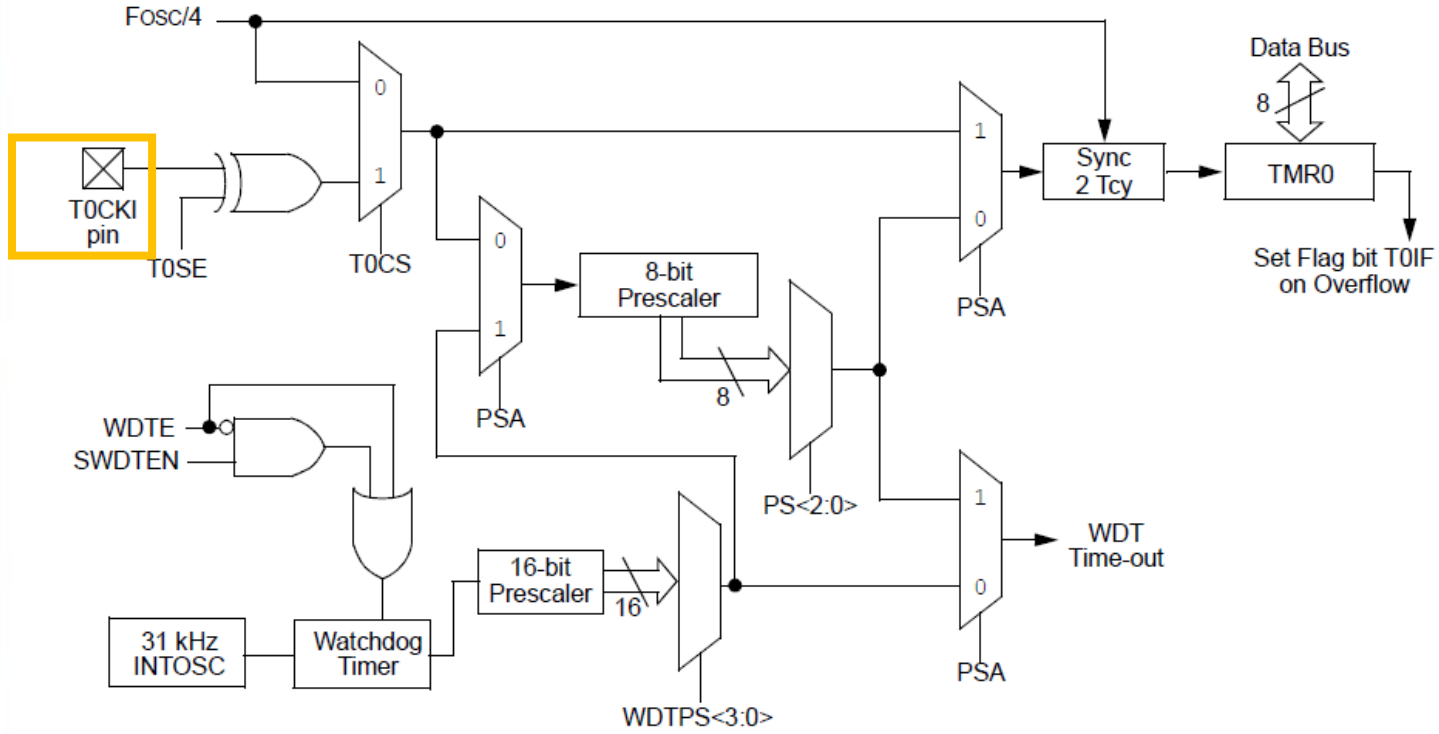
Activity 10 C: Signal generation using Timer 0

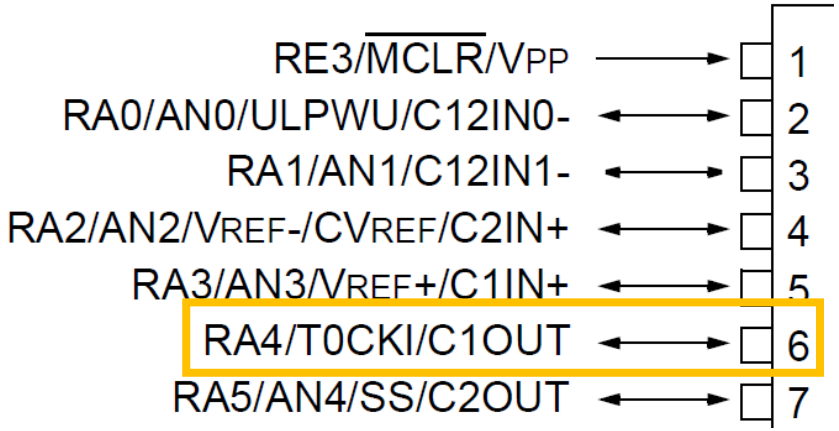
- Now we will apply the same scenario using C but we will use the extra options of the timer 0.
- As shown before that the maximum value that can be loaded in the timer is 256, i.e., the maximum time that can be counted is 256µs (in case of 4MHz oscillator).
- Timer 0 can be slowed down using a prescaler; a circuit that divide the oscillator frequency using 8 cascaded flipflops.
- The prescaler has 8 possible settings. It can divide the oscillator frequency by 2, 4, 8, 16, 32, 64, 128, 256, depends on PS2, PS1 and PS0 bits.
- If it required to get the oscillator signal without division, a switch (bit) called PSA is used to connect the timer 0 directly to the oscillator pulses (of course ¼ of the frequency)



Activity 10 C: Signal generation using Timer 0

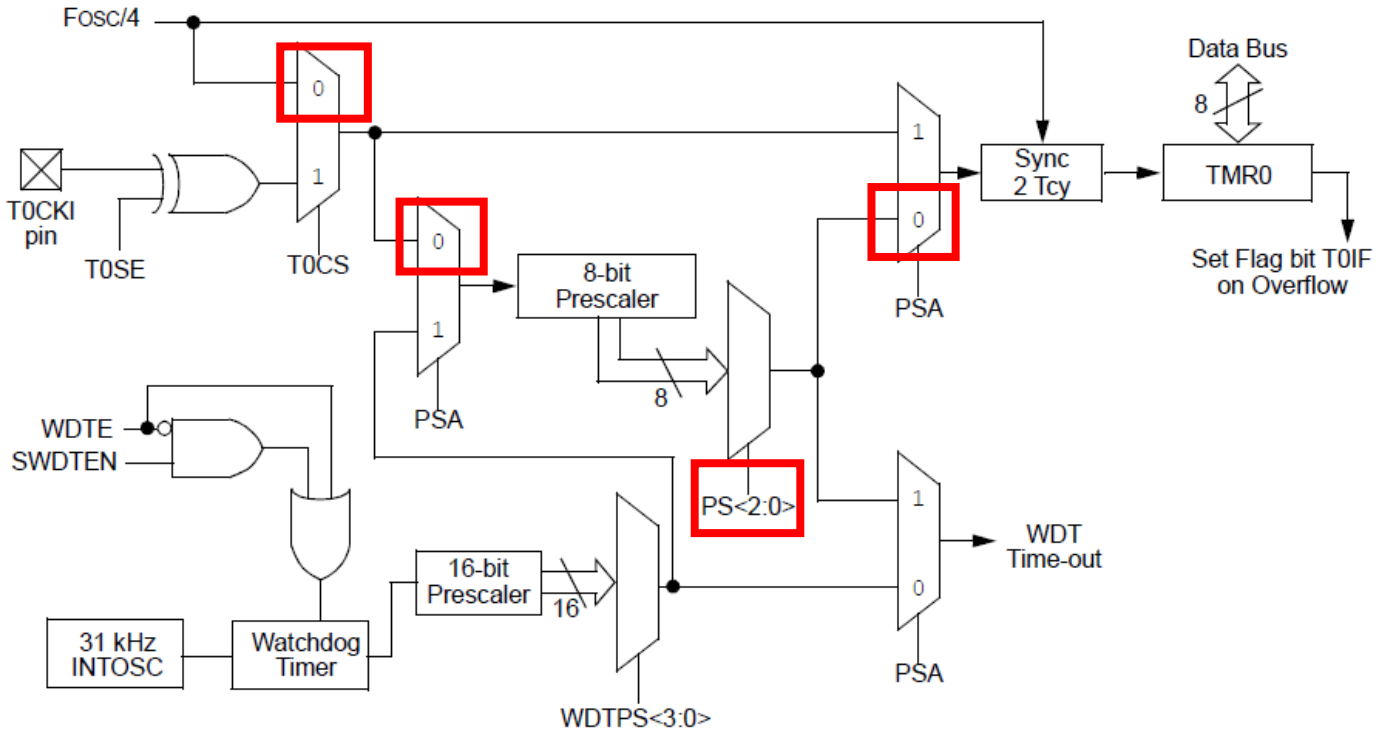
- Timer0 has much more options and the real block diagram is shown below.
- Other than timer, Timer0 can be also used as counter. It can count the pulses applied on pin RA4/T0CKI (Timer zero clock input). Also the counter increment can be selected on high-to-low transition on T0CKI pin or low-to-high transition.
- Furthermore, the prescaler of timer0 (if not used for timer0) can be used for another timer called watch dog timer. Check datasheet for more info. NB: the watch dog timer is usually disabled in the configuration bits in our codes





Activity 10 C: Signal generation using Timer 0

- Thus, we can now understand the bits of OPTION_REG. In this activity, we will use the timer0 as timer, so T0CS =0, we don't care about T0SE as Timer0 is a timer. We need to use prescaler (PSA = 0) and select PS2,PS1, PS0 for the prescaler division value



REGISTER 5-1: OPTION_REG: OPTION REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 5 **T0CS:** TMR0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (Fosc/4)

bit 4 **T0SE:** TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA:** Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0 **PS<2:0>:** Prescaler Rate Select bits

BIT VALUE	TMR0 RATE	WDT RATE
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

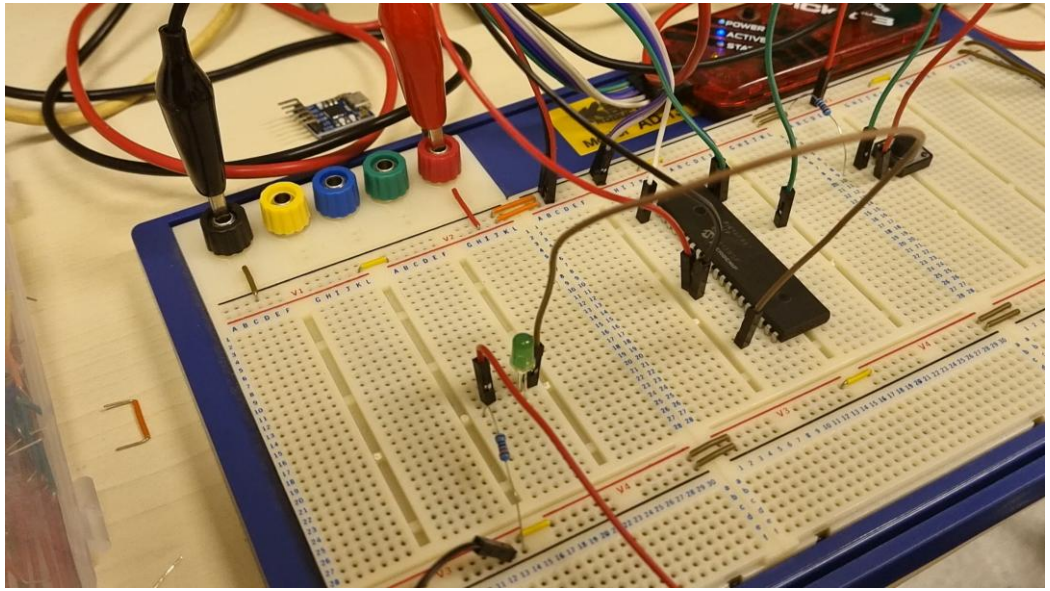
Activity 10 C: Signal generation using Timer 0

- In this activity, we need to generate a signal at 10Hz, i.e., 100ms (50ms ON, 50ms OFF)
- In fact we need the flag to be set after 50ms.
- This flag will set after 1µs x Prescaler x "time taken by timer0" = 1µs x Prescaler x (256-TMR0) (case of 4Mhz oscillator).
- For a general oscillator, the equation will be time = Tins x Prescaler x (256-TMR0)
- In this case 1µs x Prescaler x (256-TMR0) =50000µs
- For example if Prescaler = 256, (256-TMR0) = 50000/256 ≈ 195, so TMR0 = -195 or 61

Activity 10 C: Signal generation using Timer 0

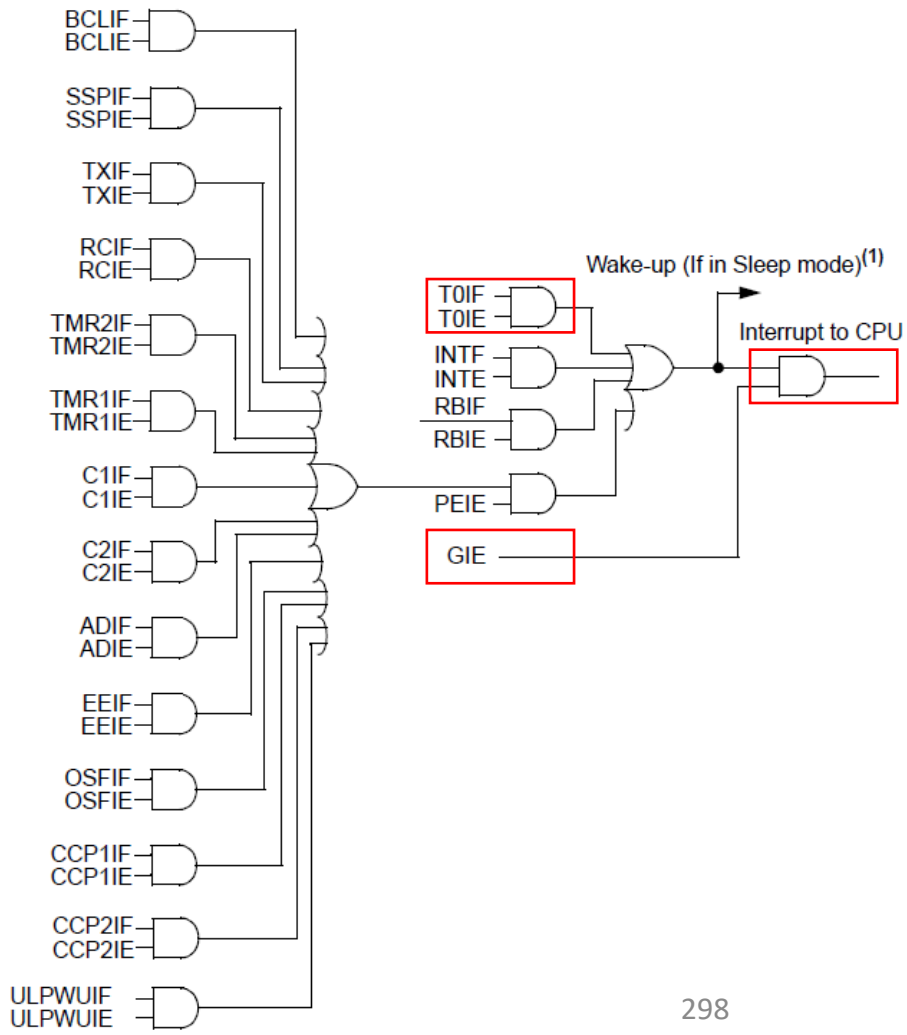
- Option_reg is tuned at line 31 with a prescaler of 256 (PS2=1, PS1=1, PS0=1)
- The toggling is achieved using the "!"

```
25    #define _XTAL_FREQ 4000000  // used crystal is 4MHz
26
27    void setup() {
28        PORTD = 0; // clear port
29        OSCCON = 0b01100000; // select 4MHz oscillator
30        TRISD = 0b00000000; // and as out
31        OPTION_REG = 0b11010111; // prescaler 256
32    }
33
34    void main(void) {
35        setup();
36        while (1) {
37            T0IF = 0;         //clear the flag
38            TMR0 = -195;      // load TMR0 by 61 to make 195 steps
39            while (!T0IF);    // wait T0IF to be 1
40            RD0 = !RD0;       // toggle RD0
41        }
42    }
```



Interrupt on Timer zero

- As shown in the previous activities, the code keeps on checking the timer flag to take decision.. Thus the code is pending and no other process can be done. This scenario is similar to the delay function in which no other process can be done.
- Thus, timer zero is best used with interrupt. The flag will trigger the interrupt event every time the timer overflows. In the meanwhile the main task can be processed.



- To activate the interrupt 3 conditions are required
  1. GIE = 1
  2. T0IE = 1
  3. Timer overflow => T0IF is triggered

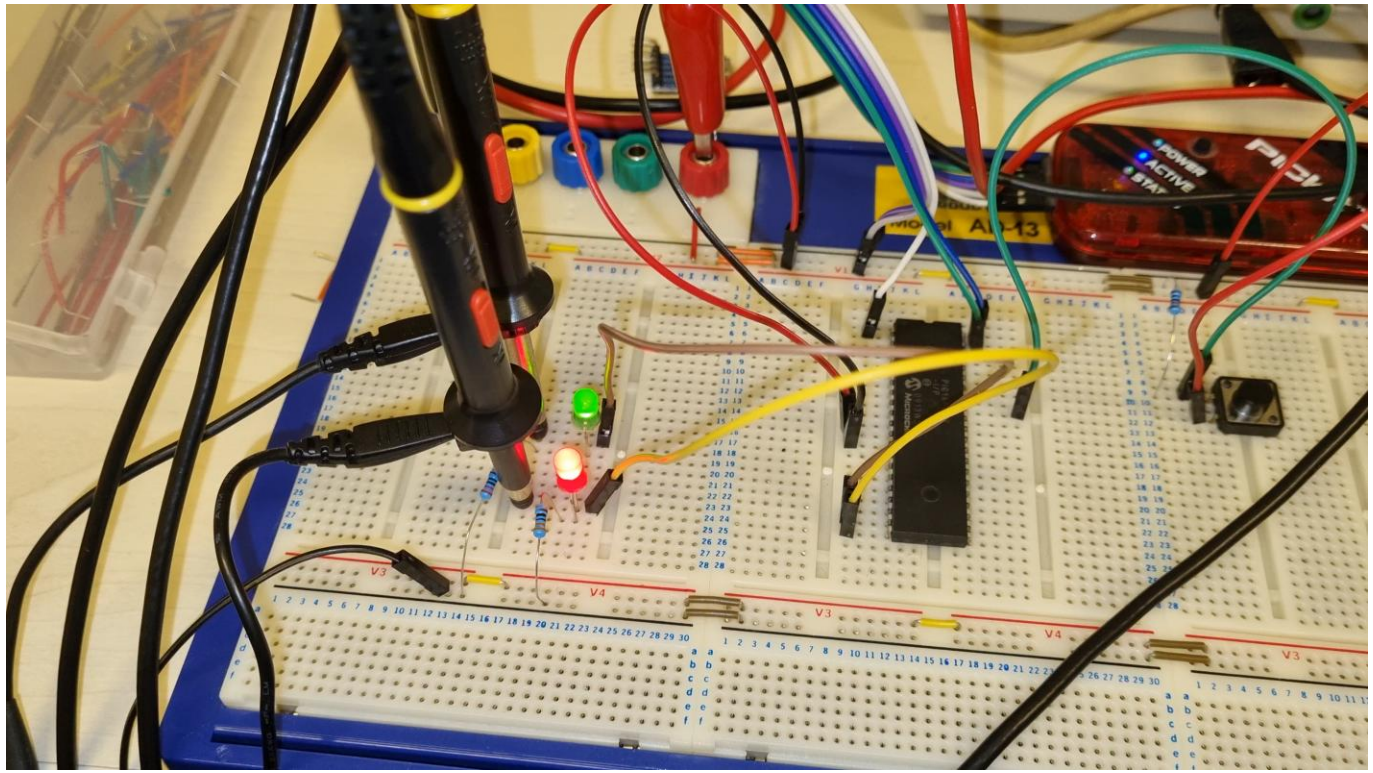
  When these conditions are satisfied, the PC switches to 0x04 (Interrupt vector)

Lecture

Activity 11: Multi tasking using Timer0 interrupt

- In this activity, we'll design a multitasking process using Timer0 interrupt
- The first task is located in the main code in which we'll blink a led at 1Hz.
- At the same time, will configure Timer0 to blink another led in the ISR at 10Hz.

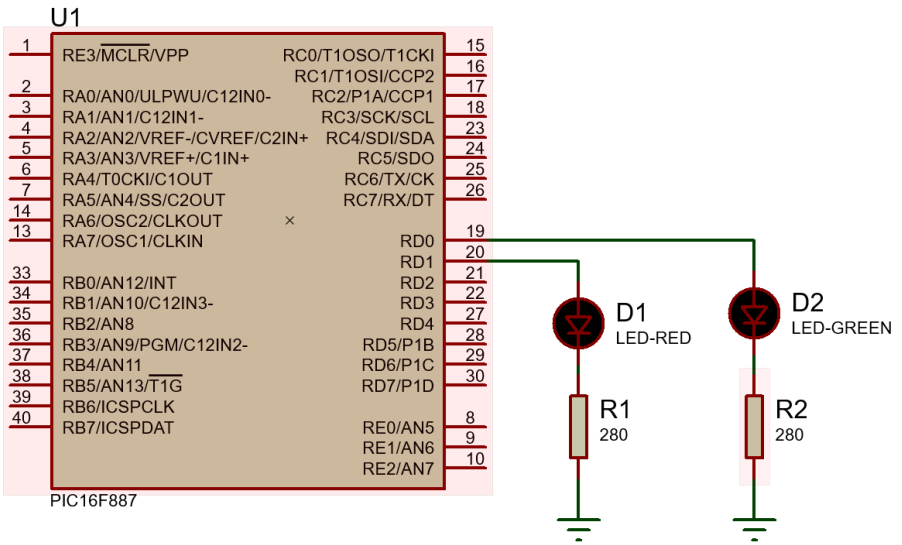
Multitasking in a computer is based on a scheduler which is based on timer trigger events.
For example when running internet explorer, media player, office application or any other applications on your PC, the Processor, due to the operating system, schedule the processes based on their priority. You have the feeling that all applications are running at the same time but in fact, every application is operating in a time frame.



Activity 11: Multi tasking using Timer0 interrupt

- Two conditions are set in the SETUP function; GIE =1, T0IE = 1 (line 61, 62).
- One condition is missing is the timer overflow. No need to check the flag here. The interrupt event is triggered whenever the flag is raised .

```
51    SETUP:                    ; by default we are in BANK0
52        CLRF    PORTD         ; clear port at the beginning
53        BSF     RP0           ; RP1 IS BY DEFAUT 0
54        MOVLW   01100000B     ; configuring OSCCON register
55        MOVWF   OSCCON        ; to select 4Mhz oscillator
56        MOVLW   00000000B     ; portD is out
57        MOVWF   TRISD
58        MOVLW   11010111B;    ; 256 prescaller used
59        MOVWF   OPTION_REG
60        BCF     RP0           ; access to BANK 0, RP0 = 0 , RP1 = 0
61        BSF     GIE           ; enable global int
62        BSF     T0IE          ; enable timer0 int
63        RETURN
```



Activity 11: Multi tasking using Timer0 interrupt

- The first task is located in the main code. It consists of blinking a led connected to RD1.
- The second task is located in the ISR. it is called every 50ms to flip the led connected to RD0 and return back to the main code.
- This is the appropriate application of timers with interrupt in which is mainly used for multitasking process.

```
35      ORG     0X4             ; interrupt vector
36      BCF     T0IF
37      MOVLW   -195            ; reload TMR0 by 61 to count 159 steps
38      MOVWF   TMR0
39      MOVLW   00000001B       ; toggling RD0
40      XORWF   PORTD, F        ; using XOR
41      RETFIE
```

```
43  MAIN:
44      CALL    SETUP
45      BCF     T0IF            ; clear first the flag
46      MOVLW   -195            ; load TMR0 by 61 to count 159 steps
47      MOVWF   TMR0
48  REPEAT:                     ; blink red led
49      MOVLW   00000010B       ; toggle RD1
50      XORWF   PORTD
51      CALL    DELAY_500MS
52      GOTO    REPEAT          ; REPEAT CODE
```



Test your knowledge

1. It is required to generate a signal using Timer0 With a period of 1ms, find the appropriate values of prescaler and TMR0 if the oscillator is 8MHz. Try to find the most accurate combination (integer number)
2. Timer 0 can be used as timer only (T/F)
3. Timer0 without prescaler can count from 0 to ____
4. Timer 0 with its prescaler can count _____cycles
5. It is required to make a delay of 1s using timer0 ? Is it possible? Why?
6. What is the value that TMR0 register should be loaded to generate an interrupt after 100 microseconds? Assume Prescaler is 256 and the Xtal=4Mhz
7. A square wave 500 Hz signal has to be generated using timer 0 interrupt. Find the value of TMR0 and its prescaler for a PIC running at 16Mhz



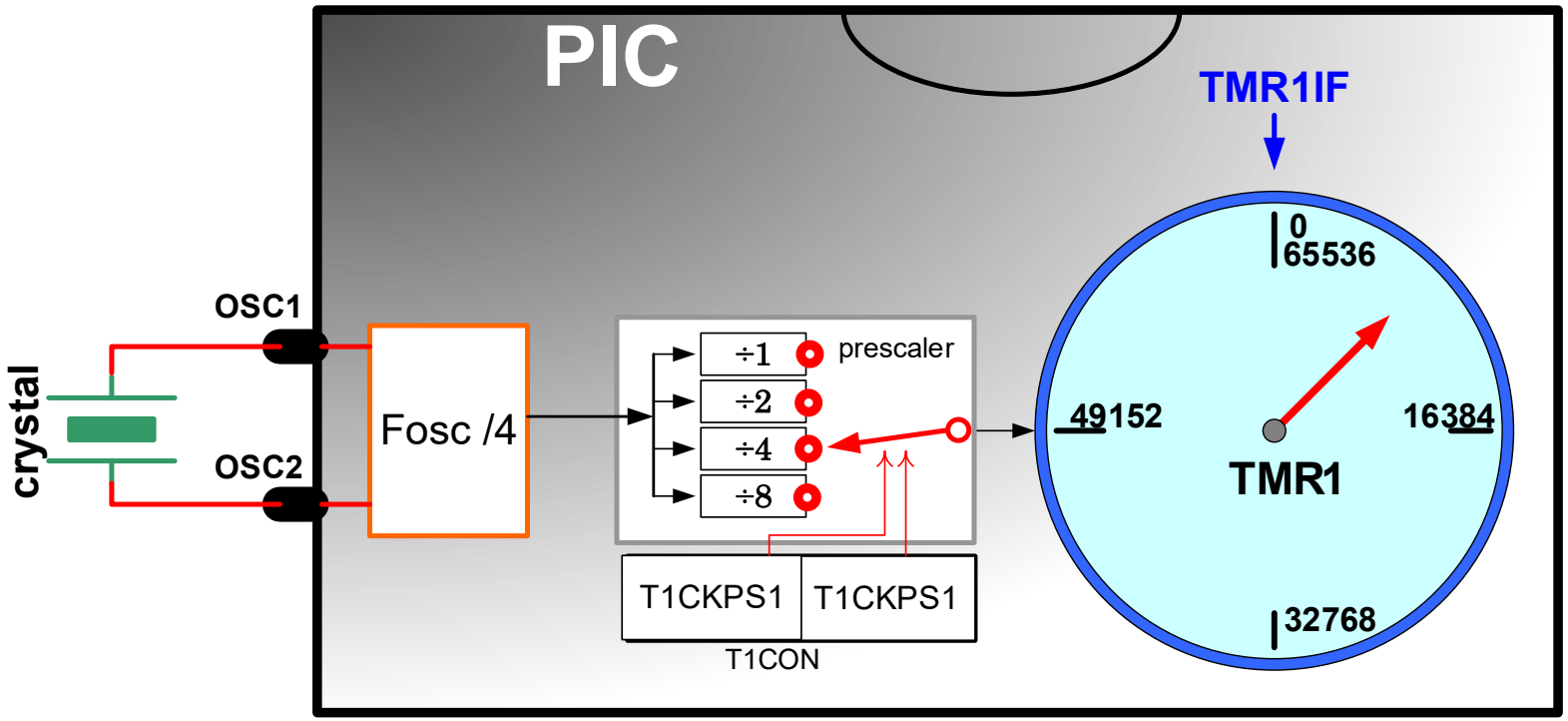
Week 6

Lecture 12 / LO3

Timers/ Timer 1

Timer 1

- Similar to timer0, timer1 is also able to count time and external pulses, thus it is also used as timer and counter.
- Timer1 is a 16bit counter, i.e., the range is from 0 to 65535 (2^{16}) with a max 1/8 prescaler. It is controlled by a special register called T1CON.
- Because of the 2 byte size, Timer1 is divided into 2 registers in RAM, called TMR1L and TMR1H.



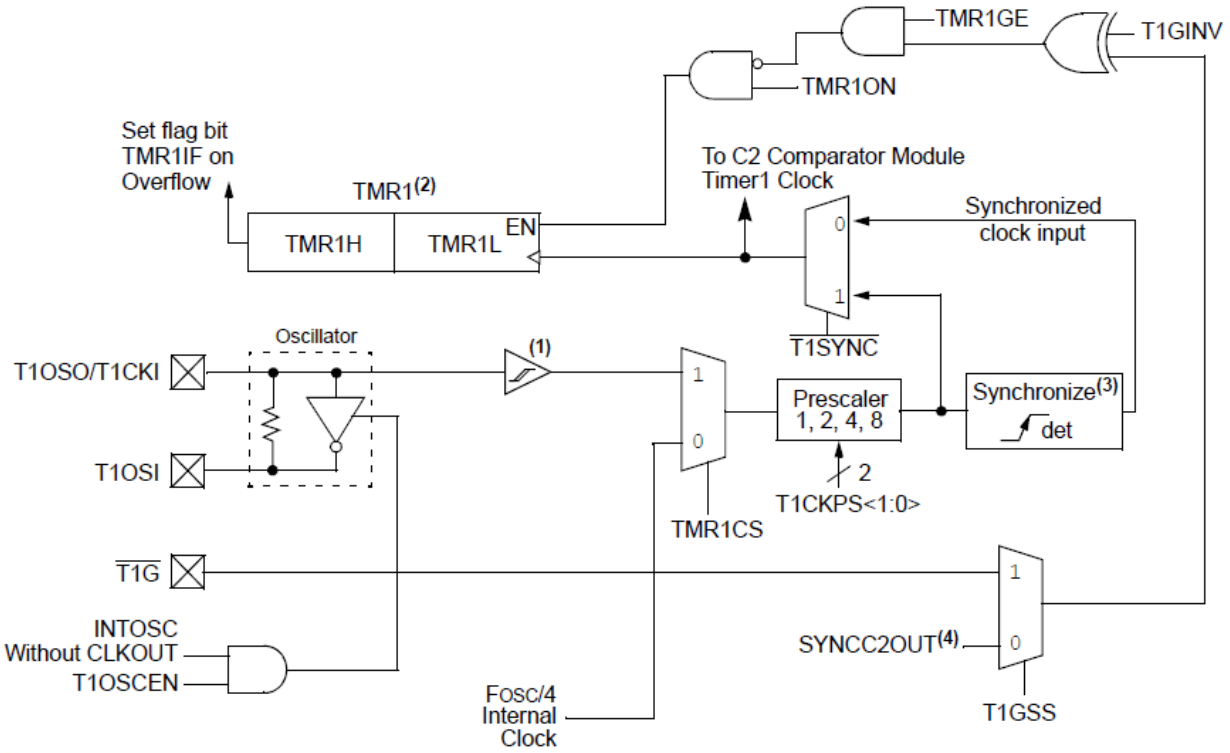
	File Address
Indirect addr. (1)	00h
TMR0	01h
PCL	02h
STATUS	03h
FSR	04h
PORTA	05h
PORTB	06h
PORTC	07h
PORTD(2)	08h
PORTE	09h
PCLATH	0Ah
INTCON	0Bh
PIR1	0Ch
PIR2	0Dh
TMR1L	0Eh
TMR1H	0Fh
T1CON	10h

Timer 1

- The Timer1 Control register (T1CON), is used to control Timer1 and select the various features of the Timer1 module such as timer or counter. In this course we are mostly concerned in timer mode.

REGISTER 6-1: T1CON: TIMER1 CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
T1GINV(1)	TMR1GE(2)	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0



bit 7 **T1GINV:** Timer1 Gate Invert bit(1)
1 = Timer1 gate is active-high (Timer1 counts when gate is high)
0 = Timer1 gate is active-low (Timer1 counts when gate is low)

bit 6 **TMR1GE:** Timer1 Gate Enable bit(2)
If TMR1ON = 0:
This bit is ignored
If TMR1ON = 1:
1 = Timer1 counting is controlled by the Timer1 Gate function
0 = Timer1 is always counting

bit 5-4 **T1CKPS<1:0>:** Timer1 Input Clock Prescale Select bits
11 = 1:8 Prescale Value
10 = 1:4 Prescale Value
01 = 1:2 Prescale Value
00 = 1:1 Prescale Value

bit 3 **T1OSCEN:** LP Oscillator Enable Control bit
1 = LP oscillator is enabled for Timer1 clock
0 = LP oscillator is off

bit 2 **$\overline{\text{T1SYNC}}$:** Timer1 External Clock Input Synchronization Control bit
TMR1CS = 1:
1 = Do not synchronize external clock input
0 = Synchronize external clock input
TMR1CS = 0:
This bit is ignored. Timer1 uses the internal clock

bit 1 **TMR1CS:** Timer1 Clock Source Select bit
1 = External clock from T1CKI pin (on the rising edge)
0 = Internal clock (Fosc/4)

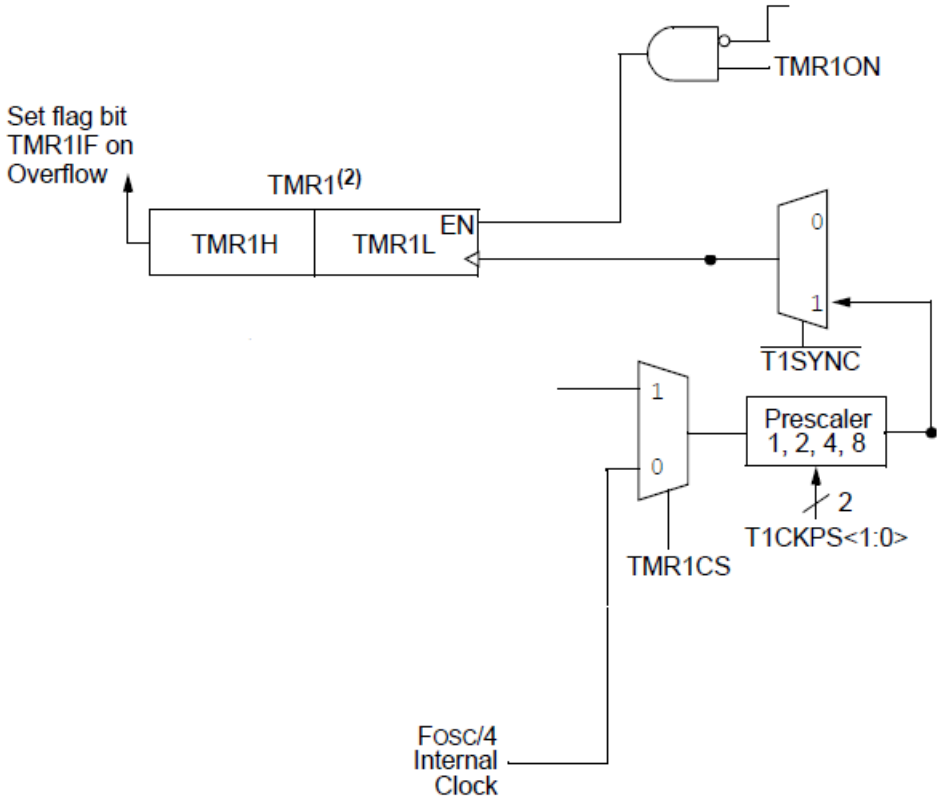
bit 0 **TMR1ON:** Timer1 On bit
1 = Enables Timer1
0 = Stops Timer1

Timer 1

- In timer mode, the block diagram can be simplified as shown and some bit are canceled. We are concerned in the prescaler, the on/off control bit and the clock source bit.

REGISTER 6-1: T1CON: TIMER1 CONTROL REGISTER

		R/W-0	R/W-0			R/W-0	R/W-0
		T1CKPS1	T1CKPS0			TMR1CS	TMR1ON
bit 7							bit 0



bit 7 **T1GINV:** Timer1 Gate Invert bit(1)
1 = Timer1 gate is active-high (Timer1 counts when gate is high)
0 = Timer1 gate is active-low (Timer1 counts when gate is low)

bit 6 **TMR1GE:** Timer1 Gate Enable bit(2)
If TMR1ON = 0:
This bit is ignored
If TMR1ON = 1:
1 = Timer1 counting is controlled by the Timer1 Gate function
0 = Timer1 is always counting

bit 5-4 **T1CKPS<1:0>:** Timer1 Input Clock Prescale Select bits
11 = 1:8 Prescale Value
10 = 1:4 Prescale Value
01 = 1:2 Prescale Value
00 = 1:1 Prescale Value

bit 3 **T1OSCEN:** LP Oscillator Enable Control bit
1 = LP oscillator is enabled for Timer1 clock
0 = LP oscillator is off

bit 2 **T1SYNC:** Timer1 External Clock Input Synchronization Control bit
TMR1CS = 1:
1 = Do not synchronize external clock input
0 = Synchronize external clock input
TMR1CS = 0:
This bit is ignored. Timer1 uses the internal clock

bit 1 **TMR1CS:** Timer1 Clock Source Select bit
1 = External clock from T1CKI pin (on the rising edge)
0 = Internal clock (Fosc/4)

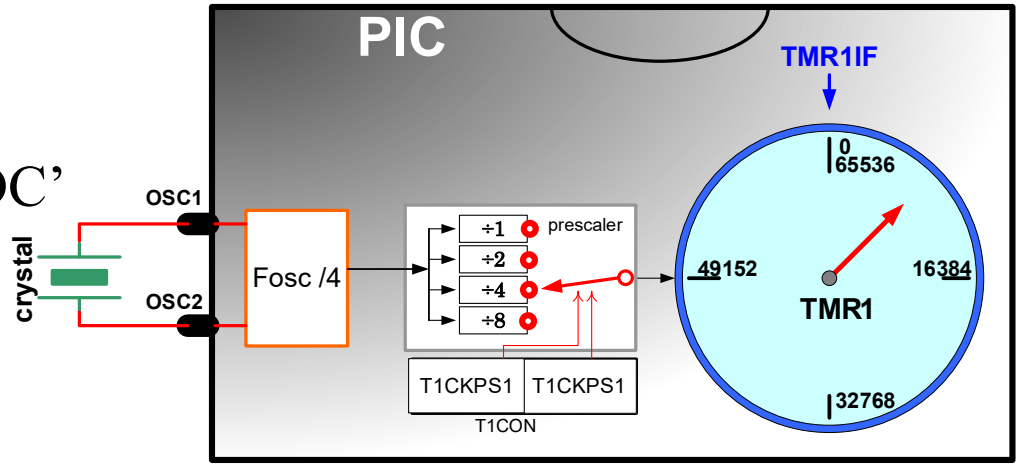
bit 0 **TMR1ON:** Timer1 On bit
1 = Enables Timer1
0 = Stops Timer1

Activity 12: Signal generation using Timer 1

- One of the advantage of Timer1 compared to timer0 is the timing range. Timer0 with its prescaler can count up to 256x256 = 65536 cycle (i.e., 65536 µs if the oscillator is 4MHz). Whereas, timer1 with its prescaler can count up to 65536 x 8 = 524288 cycles (or 524288 µs for 4MHz oscillator).
- In this Activity, we are interested in generating a signal of 1Hz (500ms ON/ 500ms OFF) for a PIC running at 4MHz
- The task is to find the prescaler and the TMR1 values to generate a flag after 500ms
- Time = Tins x Prescaler x (65536-TMR1) = 500,000 µs
- For a prescaler = 8 and Tins = 1 µs

  65536-TMR1 = 500000/(8x1µs)

→ TMR1 = 3036 (value in decimal) corresponds to H'0BDC'

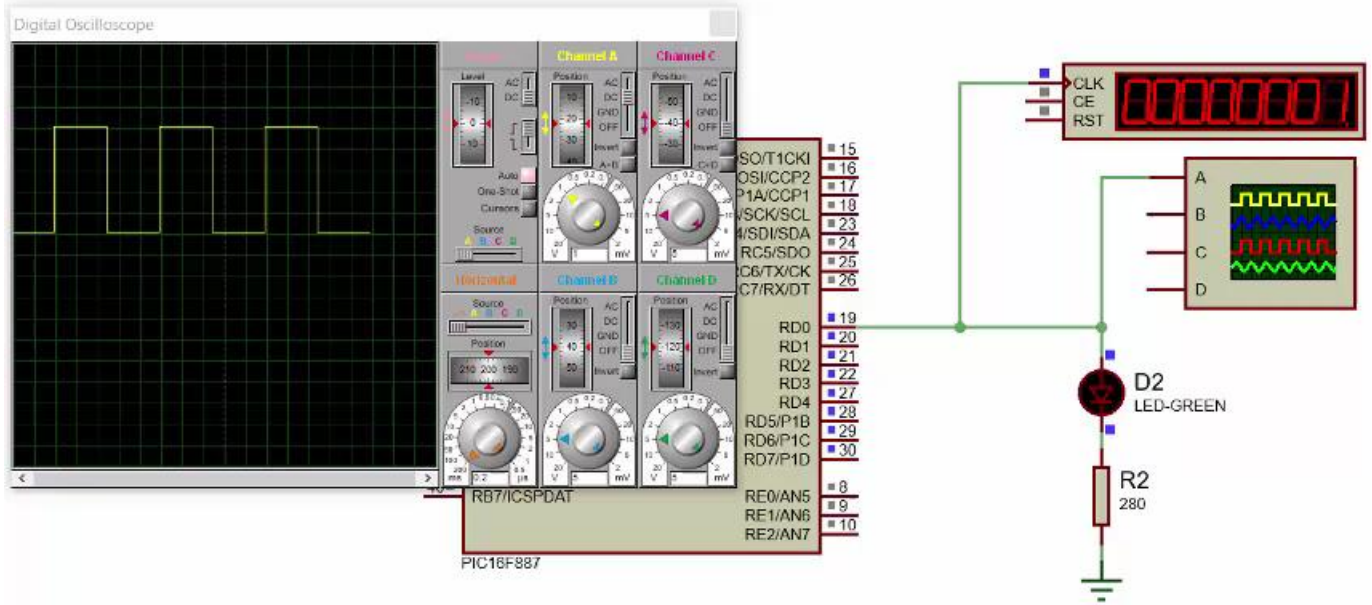


Thus we have to load TMR1L by 0xDC

and TMR1H by 0x0B

Activity 12: Signal generation using Timer 1

- First the T1CON is configured to set the prescaler to 8 and turn ON the timer.
- Next the values are loaded in TMR1L and TMR1H as already calculated.
- We keep on testing the flag TMR1IF which is located in PIR1 register, If set, we toggle the led.




```
33      MAIN:
34          CALL    SETUP
35          BCF     TMR1IF      ; CLEAR FLAG BEFORE STARTING
36          MOVLW   00110001B
37          MOVWF   T1CON       ; TIMER IS ON WITH PRESCALER 8
38      REPEAT:
39          BCF     TMR1IF
40          MOVLW   0x0B        ; load 3036 in TMR1
41          MOVWF   TMR1H       ; which is 0BDC in hex
42          MOVLW   0xDC
43          MOVWF   TMR1L
44          BTFSS   TMR1IF      ; wait the flag located in PIR1
45          GOTO    $-1
46          MOVLW   00000001B   ; toggle RD0
47          XORWF   PORTD
48          GOTO    REPEAT      ; REPEAT CODE
```



Test your knowledge

1. It is required to generate a signal using Timer1 With a period of 1ms, find the appropriate values of prescaler and TMR1 if the oscillator is 20MHz. Try to find the most accurate combination (integer number)
2. Timer1 can be used as timer only (T/F)
3. Timer1 without prescaler can count from 0 to ____
4. The TMR1 with its prescaler can count _____cycles
5. It is required to make a delay of 1s using timer1 ? Is it possible? Why? What should be changes in the circuit to make it possible?



Week 6

Lab 5 / LO3

Multitasking using timer0 Application with DC motor



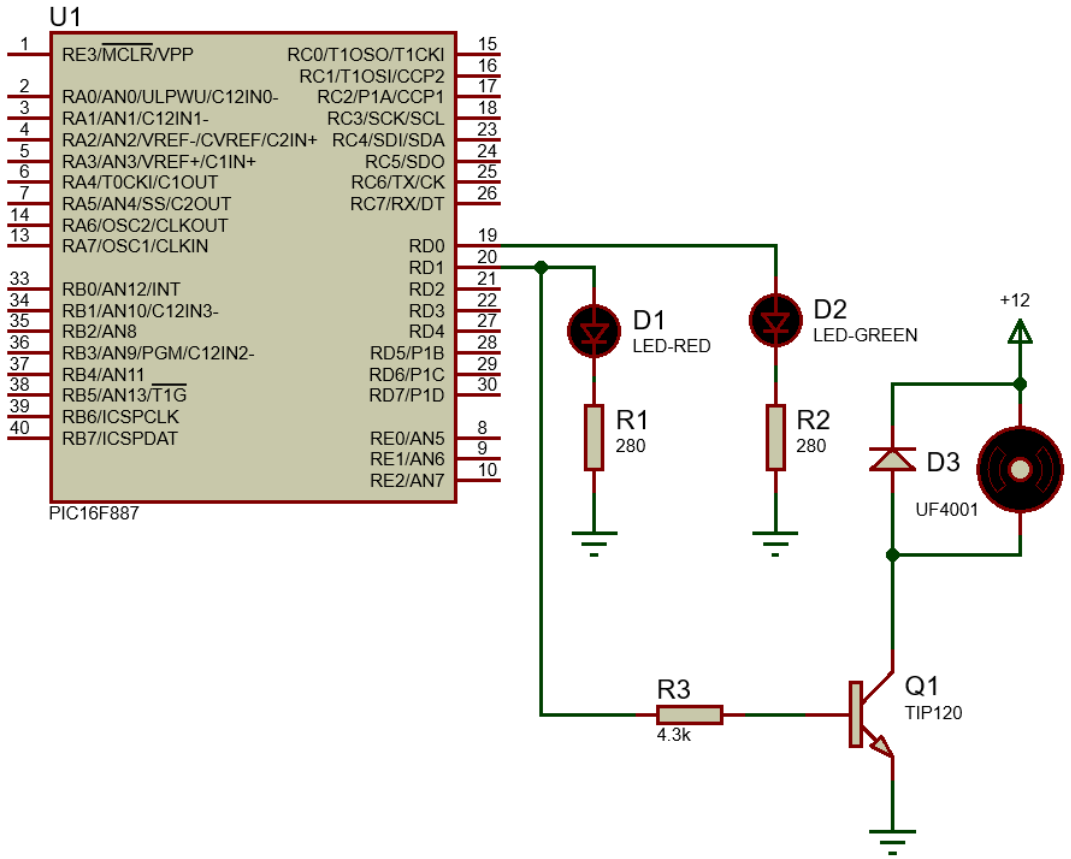
Activity 11: Multi tasking using Timer0 interrupt

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. PIC programmer
  4. Oscilloscope 2 channels
  5. Power supply 5V and 12V
  6. Breadboard wires (male-male)

- Required components:
  1. PIC 16F887
  2. Led x2
  3. Resistor from 200 to 470 ohm x 2
  4. Resistor from 1k to 10k x 1
  5. NPN transistor TIP120 or any equivalent (or use 2n2222 in case of small motor)
  6. Fast recovery diode UF4001 or any equivalent (use any regular diode if you don't have)
  7. Any DC motor (measure the motor current before choosing the transistor)

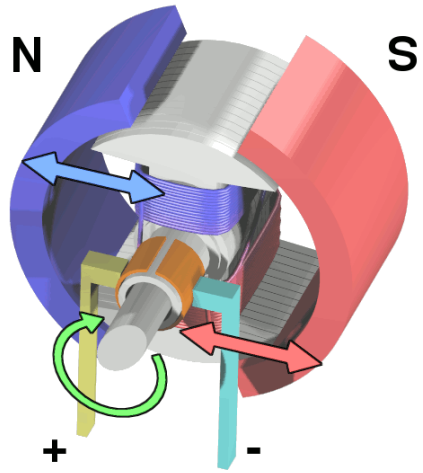
Activity 11: Multi tasking using Timer0 interrupt

- To make the design more attractive, a DC motor is added to the circuit and connected to RD1 (the low speed led).



Activity 11: Multi tasking using Timer0 interrupt

- The design of the brushed DC motor is quite simple. A permanent magnetic field is created in the stator by either of two means:
  1. Permanent magnets
  2. Electro-magnetic windings
- If the field is created by permanent magnets, the motor is said to be a "permanent magnet DC motor" (PMDC).
- If created by electromagnetic windings, the motor is often said to be a "shunt wound DC motor" (SWDC).
- Today, because of cost-effectiveness and reliability, the PMDC motor is the motor of choice for applications involving fractional horsepower DC motors, as well as most applications up to about three horsepower.











Activity 11: Multi tasking using Timer0 interrupt



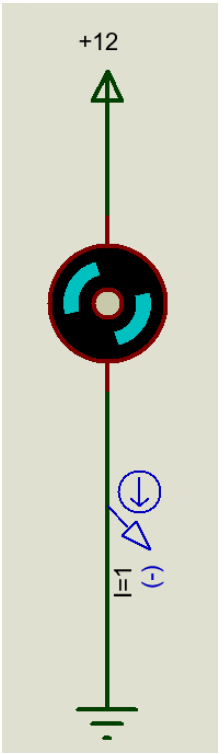
How to design an amplifier circuit to drive a motor

- Assume that we have a 12V DC motor with internal resistance of 12Ω.
- The current required by the motor will be

$$I_m = \frac{V_m}{R_m} = \frac{12}{12} = 1A$$

- This current cannot be supplied by the Arduino pin and thus a transistor amplifier circuit is required.

- The transistor will be used to amplify the Arduino current to be able to drive the motor.



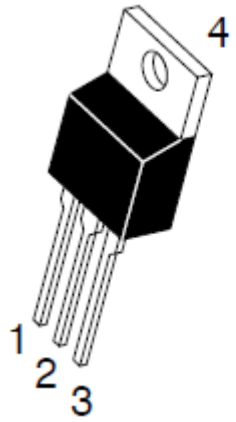
Activity 11: Multi tasking using Timer0 interrupt

How to design an amplifier circuit to drive a motor

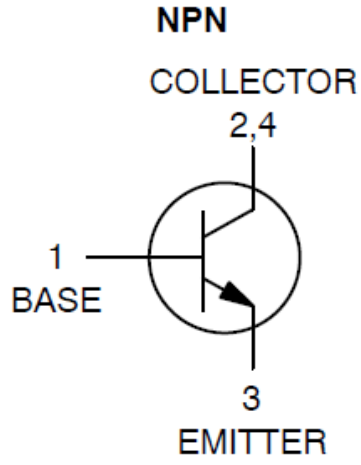
- The first thing to consider is the current of the transistor
- In this case, we are using the TIP120 (Darlington transistor) that has a collect current of 5A continuous

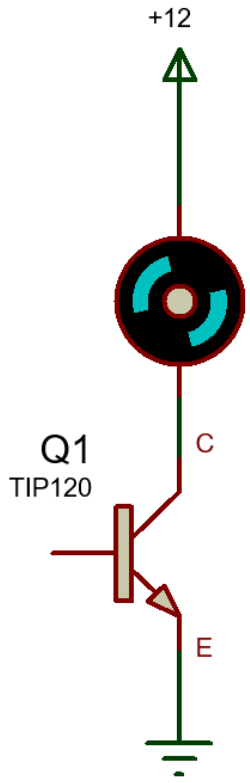
I_C	Collector Current (DC)	5	A
I_{CP}	Collector Current (Pulse)	8	A

h_{FE}	* DC Current Gain	V_{CE} = 3V,I_C = 0.5A V_{CE} = 3V, I_C = 3A	1000 1000



TO–220
CASE 221A
STYLE 1





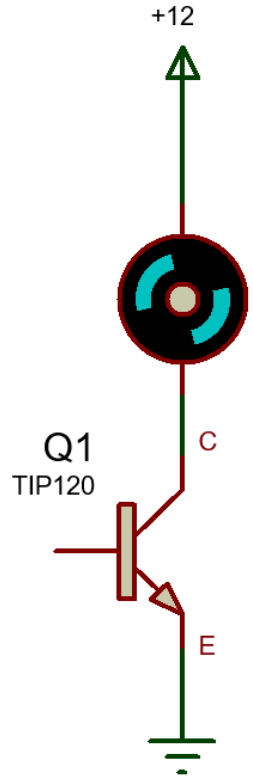
Activity 11: Multi tasking using Timer0 interrupt

How to design an amplifier circuit to drive a motor

- The saturation current required is the motor current I_m which is the collector current I_C
- Knowing I_C, we can calculate the base current I_B using

$$I_B = \frac{I_C}{\beta} = \frac{1}{1000} = 0.001A = 1mA$$

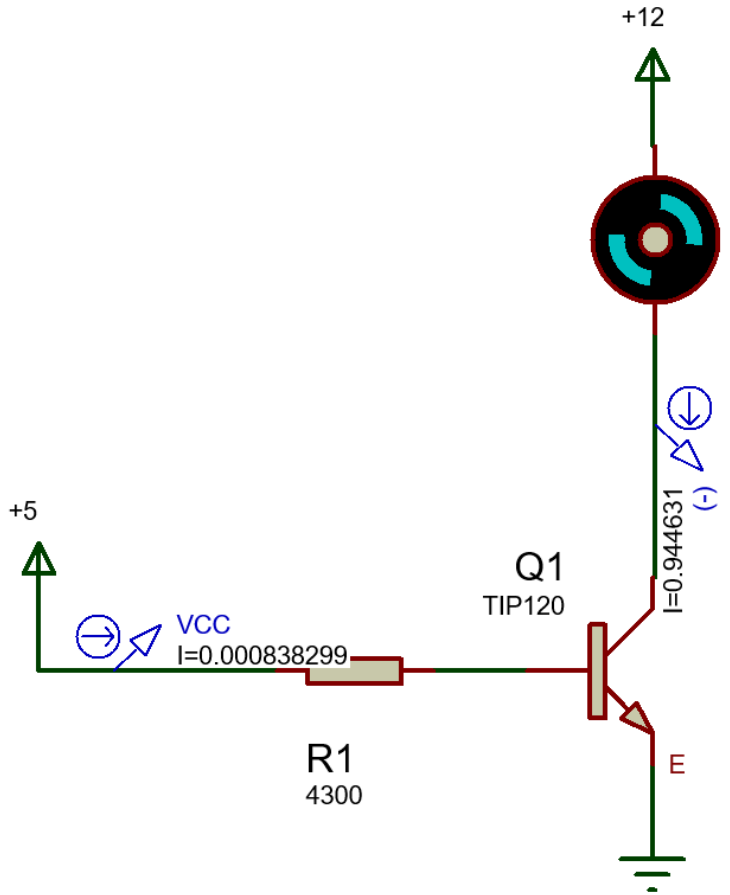
- With *beta* is the grain in current of the transistor (also called *hef*)
- NOTE: the base current will be delivered from the PIC and should be less that 20mA (as recommended by the PIC Datasheet)



Activity 11: Multi tasking using Timer0 interrupt

How to design an amplifier circuit to drive a motor

- Knowing IB, we need now to find the series resistor between the PIC pin and the base of the transistor. This is required to drop the PIC voltage (5 volt) to the VBE of the transistor which is around 0.7 volt (voltage across the BE pins which is a PN junction)



- The value of the resistor can be calculated using KVL and Ohm's law as:

$$R = \frac{V_{PIC} - V_{BE}}{I_B} = \frac{5 - 0.7}{0.001} = 4300\Omega = 4.3k\Omega$$

Activity 11: Multi tasking using Timer0 interrupt

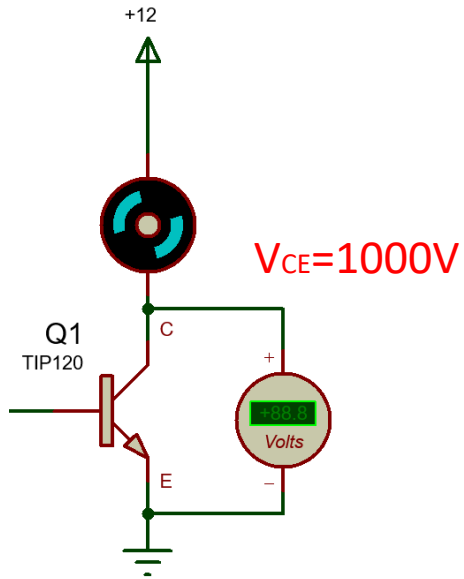
How to design an amplifier circuit to drive a motor

- The circuit need enhancement to protect the transistor for the Inductor current of the motor.

- Assume that the motor is normally running and a 1A is flowing through the transistor.
- When switching it off by sending the BCF instruction, the current should be zero (logical analysis)
- But the fast switching will generate a high voltage from the motor that can be modeled as:

- $V_M = V_L = L\frac{di_M}{dt} = L\frac{\Delta i}{\Delta t}$
- Taking some numerical values for L=1mH , ΔiM=1A, and Δt=1µs

- $V_M = V_L = 1000V$ which represents the voltage across the CE pins of the transistor.
- The VCE voltage of the transistor will be given in the datasheet



V_{CEO}	Collector-Emitter Voltage : TIP120	60	V

Activity 11: Multi tasking using Timer0 interrupt

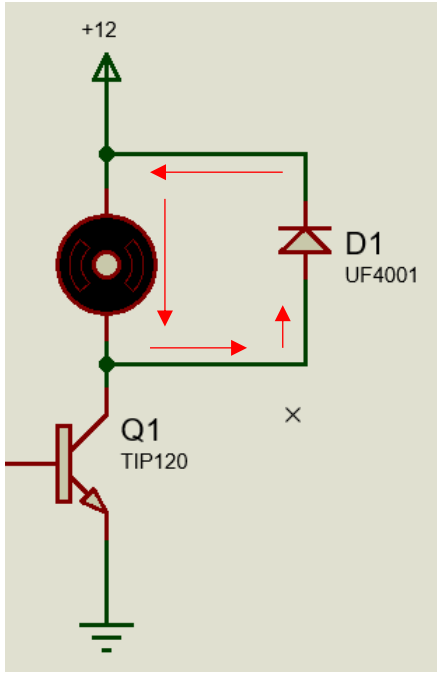
How to design an amplifier circuit to drive a motor

- Thus it is required to find solution for the high voltage generated by the inductor of the motor when switched off.
- It can be seen from the equation $V_M = V_L = L\frac{di_M}{dt} = L\frac{\Delta i}{\Delta t}$ that the only parameter that can be controlled is time

- Thus increasing Δt will decrease the voltage VL
- This can be done by using a fast diode (fast recovery diode)
- The role of the diode is to open a path for the current to circulate inside the Motor winding (made of inductor and resistance)

- The current will drop slowly until the energy is fully dissipated inside the resistance of the motor

The diode here is called free-wheeling diode





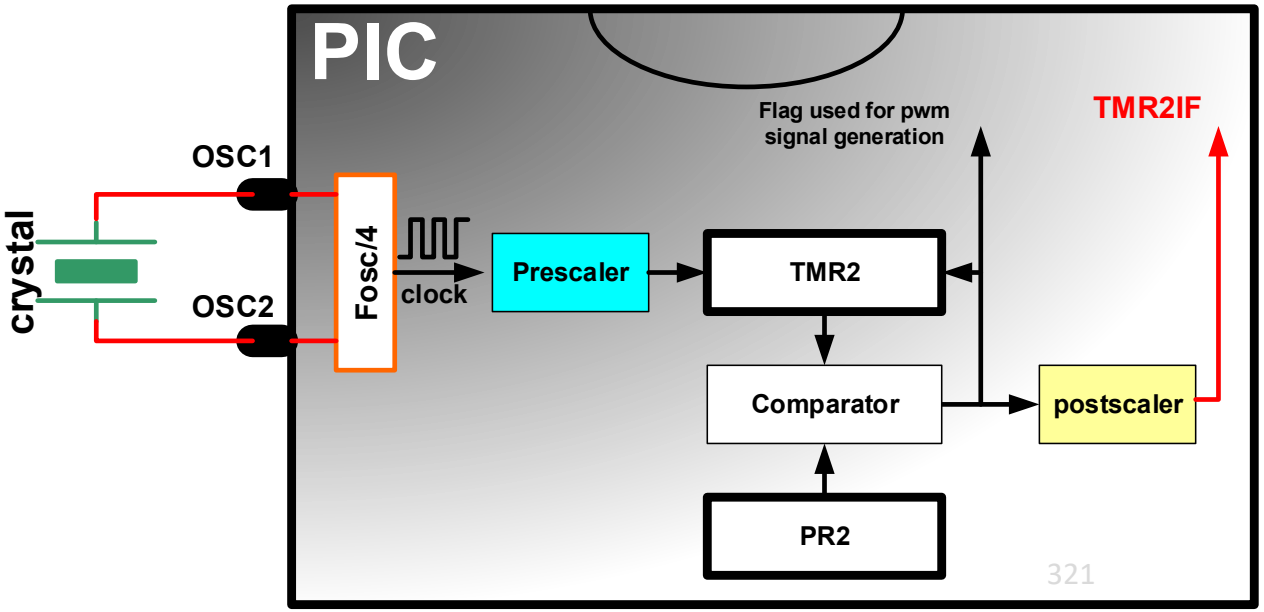
Week 7

Lecture 13 / LO3

Timers/ Timer 2/PWM

Timer 2

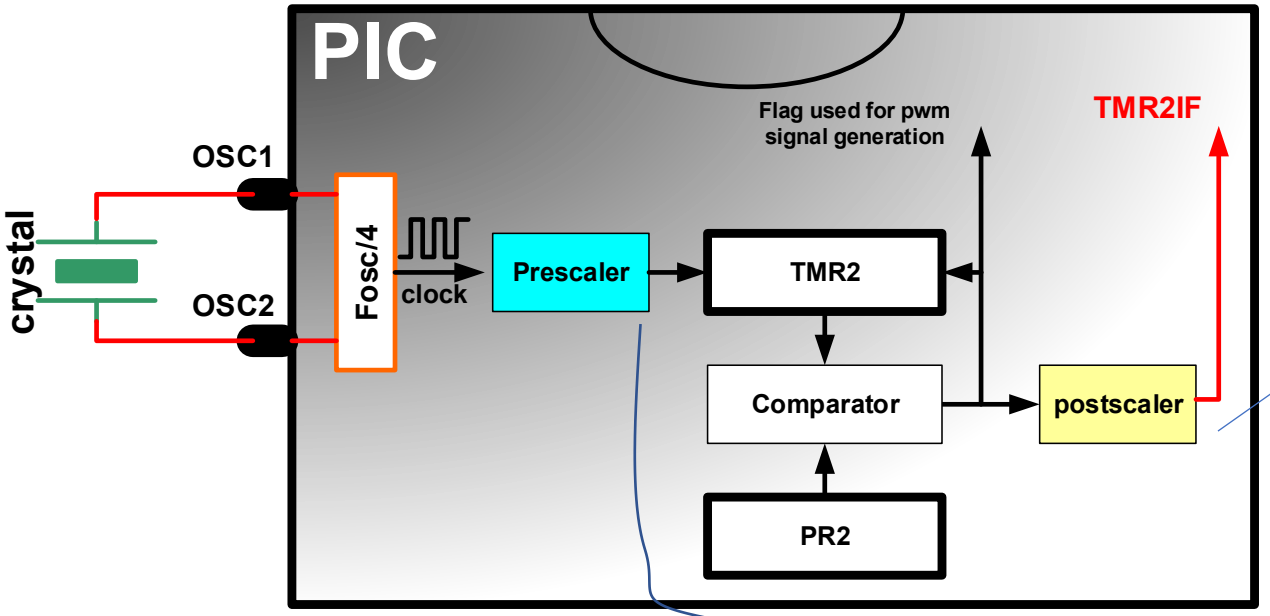
- Timer2 is an 8bit timer with a prescaler and a postscaler. Unlike Timer0 and timer1, timer2 does not have the ability to count external pulses. It is used for timing only.
- The clock input to the Timer2 module is the system instruction clock (FOSC/4). The clock is fed into the Timer2 prescaler, which has prescale options of 1:1, 1:4 or 1:16.
- The output of the prescaler is then used to increment the TMR2 register.
- The values of TMR2 and PR2 are constantly compared to determine when they match.
- TMR2 will increment from 00h until it matches the value in PR2.
- When a match occurs, two things happen:
  - TMR2 is reset to 00h on the next increment cycle
  - The Timer2 postscaler is incremented
- The match output of the Timer2/PR2 comparator is then fed into the Timer2 postscaler.
- The postscaler has postscale options of 1:1 to 1:16 inclusive.
- The output of the Timer2 postscaler is used to set the TMR2IF interrupt flag bit in the PIR1 register.



Timer 2

REGISTER 7-1: T2CON: TIMER2 CONTROL REGISTER

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0



bit 7 **Unimplemented:** Read as '0'

bit 6-3 **TOUTPS<3:0>:** Timer2 Output Postscaler Select bits

0000 = 1:1 Postscaler
0001 = 1:2 Postscaler
0010 = 1:3 Postscaler
0011 = 1:4 Postscaler
0100 = 1:5 Postscaler
0101 = 1:6 Postscaler
0110 = 1:7 Postscaler
0111 = 1:8 Postscaler
1000 = 1:9 Postscaler
1001 = 1:10 Postscaler
1010 = 1:11 Postscaler
1011 = 1:12 Postscaler
1100 = 1:13 Postscaler
1101 = 1:14 Postscaler
1110 = 1:15 Postscaler
1111 = 1:16 Postscaler

bit 2 **TMR2ON:** Timer2 On bit

1 = Timer2 is on
0 = Timer2 is off

bit 1-0 **T2CKPS<1:0>:** Timer2 Clock Prescale Select bits

00 = Prescaler is 1
01 = Prescaler is 4
1x = Prescaler is 16

- The total number of cycle that the timer2 module can count is 16x256x16 = 65535. Which is same as timer0 module. Thus timer2 is mainly used for PWM signal generation

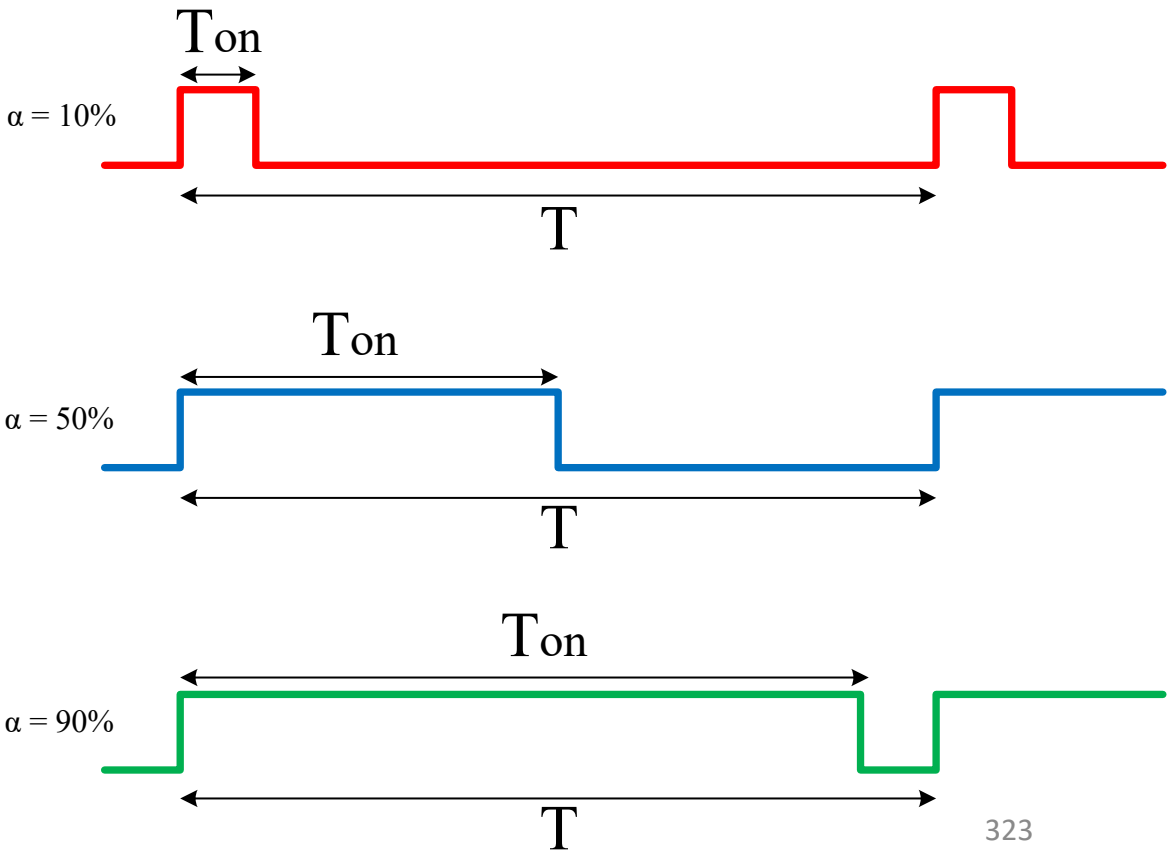
PWM signal generation

- A PWM signal or Pulse Width Modulation is a digital signal with fixed period and variable duty cycle.
- The duty cycle (α) is the positive time over the total period.

$$\alpha = \frac{T_{on}}{T}$$

- The TTL signal duty cycle is 50%, which means the positive part is half of the period. In other words, the time of the logic 1 is equal to the time of the logic 0.
- Any PIC can generate such as signal by software.
- This can be done by setting and clearing any pin for a delay time. The example below show the assembly of a PWM signal with a period of 1s with a duty cycle equal to 100ms/1000ms or 10%.

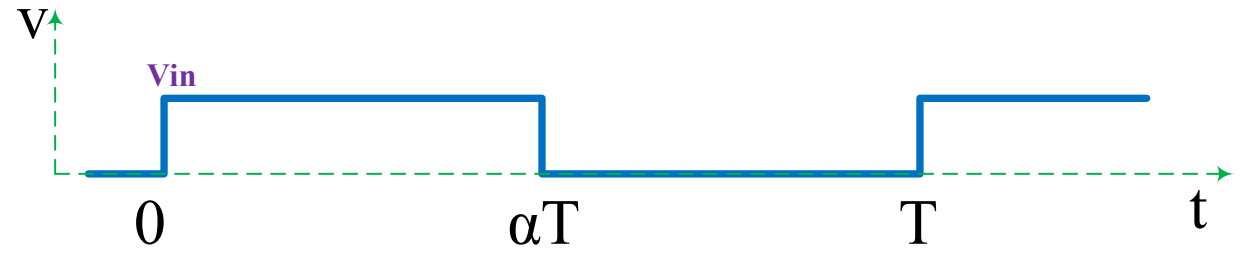
```
REPEAT:
   BSF      RD0
   CALL     DELAY_100MS
   BCF      RD0
   CALL     DELAY_900MS
   GOTO     REPEAT
```



PWM signal generation

- Why PWM signal is so important !! In fact when this signal is applied on a Led at high frequency, the intensity of light can be controlled depends on the signal average, i.e., on the duty cycle.
- Same for motors, the PWM signal can control the speed of the shaft. Also in Power electronics, this signal is usually used in converters/inverters. As well in communication systems, PWM is used in modulation.

- The average value can be calculated using the integration of the signal over period



- $V_{avg} = \frac{1}{T}\int_0^T V_{in}dt = \frac{1}{T}\left[\int_0^{\alpha T} V_{in}dt + \int_{\alpha T}^{T} V_{in}dt\right] = \alpha V_{in}$ with $0 \leq \alpha \leq 1$

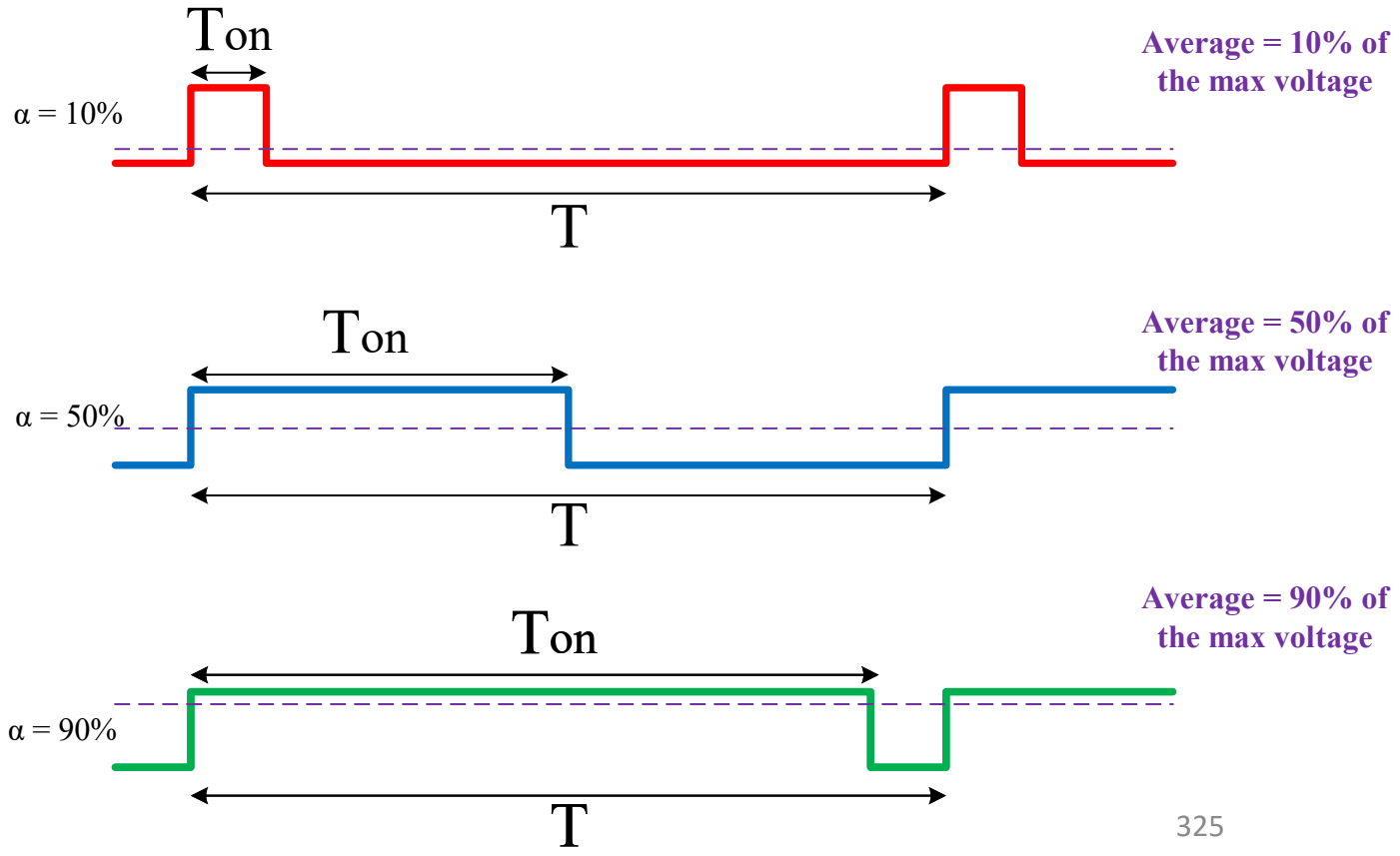
$\int_{\alpha T}^{T} V_{in}dt = 0$

- Thus we can say that the average voltage is part of the input voltage

PWM signal generation

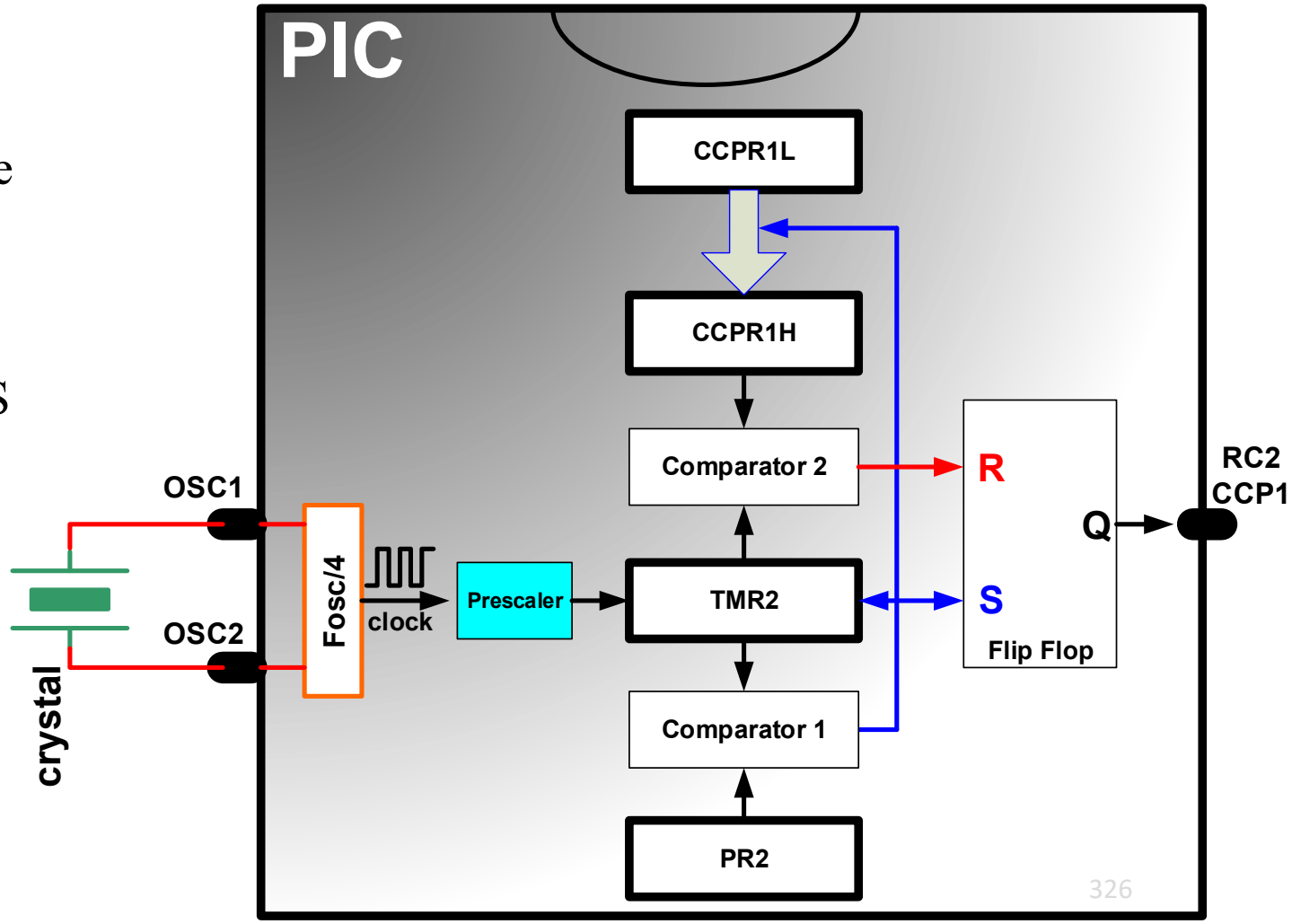
- In PIC16F887, the generation of that signal is made by hardware. This means that this PIC contains a module dedicated for this job.
- All what is required is to configure the period and the duty cycle, and the module generates signal automatically.

- The module is called CCP or Capture/Compare/PWM module.
- The PIC contains two modules.
- The CCP module can do many tasks. It can Capture the signal time, Compare times, and generates a PWM signal.
- The generation of a PWM signal is related to the TIMER2 and its PRESCALER that define the period, also, related to CCPR1L register that defines the duty cycle.
- The POSTSCALER of TIMER2 is not used in PWM generation.



PWM signal generation

- The block diagram of the CCP module is merged with the timer2 module (without the postscaler)
- The block diagram shows 2 comparators; the first one compare TMR2 with PR2 and the second compares TMR2 with CCPR1H.
- Both comparators generates events to the RS flipflop connected to the output pin RC2/CCPR1L
- The whole module is controlled by
  - T2CON
  - CCP1CON
  - CCPR1L
  - PR2



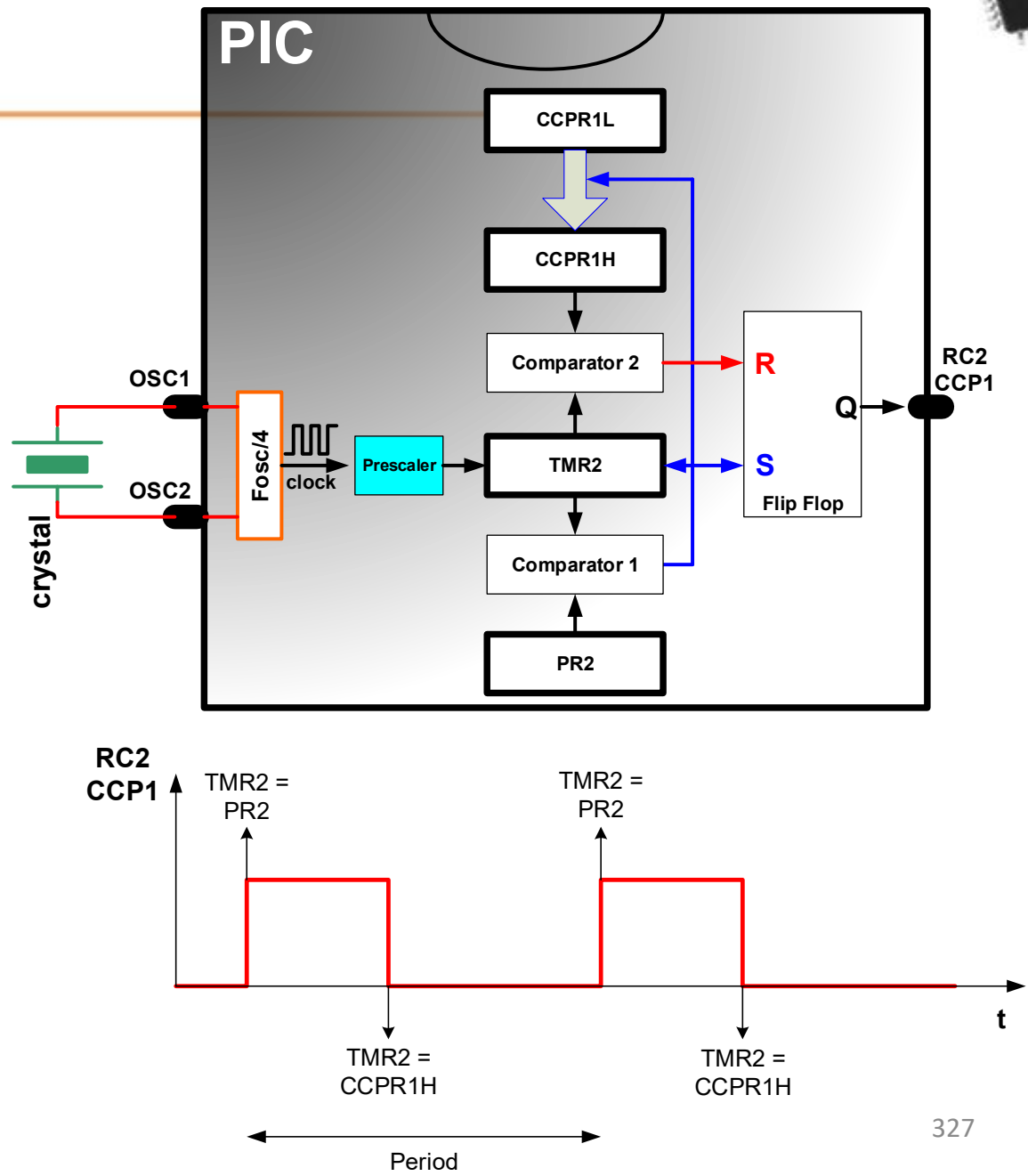
PWM signal generation

- **PWM Generation Mechanism**

- The TMR2 is incremented on every prescaler overflow.
- When TMR2 =PR2 value, the first comparator does the following
  1. Set the RS-flip-flop, (RC2=1)
  2. clear TMR2, and
  3. Fetch the CCPR1L value to CCPR1H.

- This time represents the signal period defined by the equation:

$$\text{Period } T = (PR2 + 1) \times T_{ins} \times prescaler$$



PWM signal generation

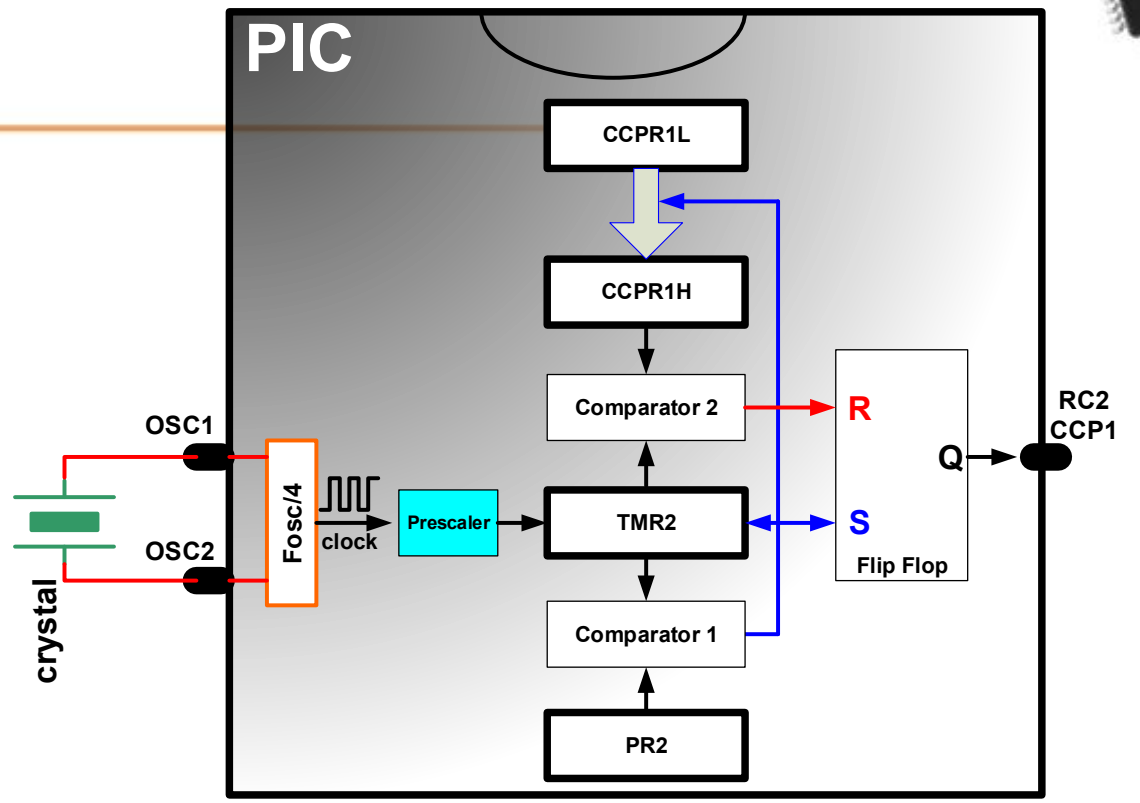
- **PWM Generation Mechanism**

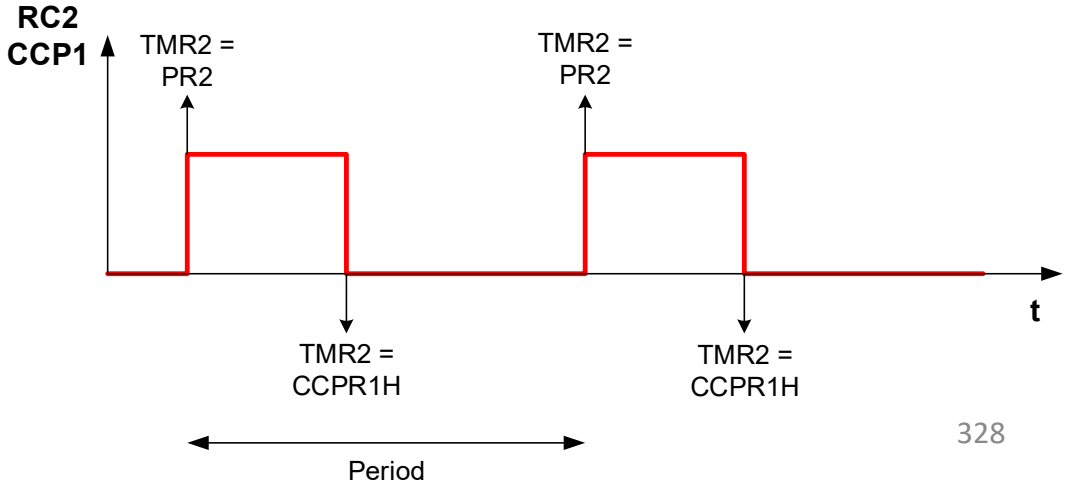
- When TMR2 becomes equal to the CCPR1H, the second comparator resets the CCP pin via the RS-flip-flop (RC2=0).

- The pulse width is given by the formula:

Pulse width $T_{on} = CCPR1L \times T_{ins} \times prescaler$

- In fact, the TMR2 reaches the value of CCPR1H before reaching the value of PR2 since the signal period is bigger than its positive portion. Therefore, to understand the scenario of the flip-flop, start by its reset position where the TMR2 value is equal to CCPR1H.





Lecture

PWM signal generation

- **PWM Generation Mechanism**

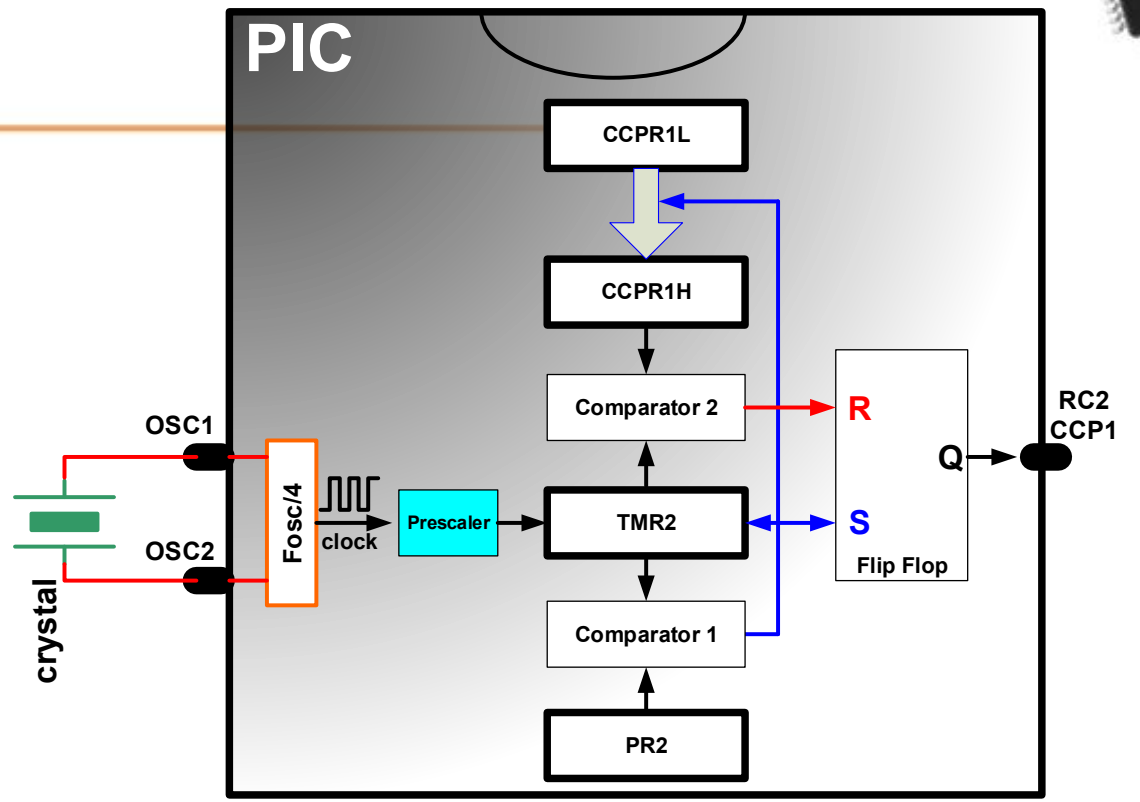
Having $T = (PR2 + 1) \times T_{ins} \times prescaler$

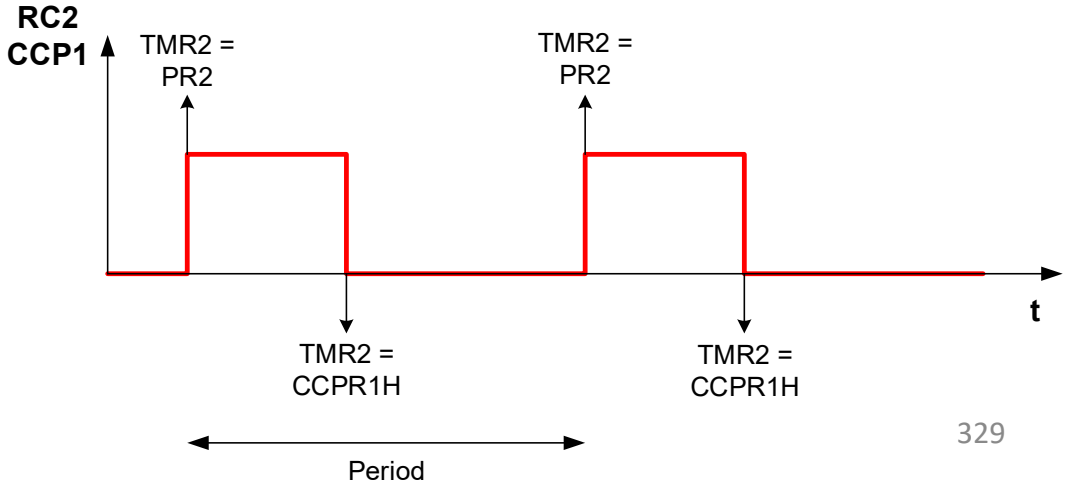
And

$T_{on} = CCPR1L \times T_{ins} \times prescaler$

The duty cycle $\alpha = \frac{CCPR1L}{PR2+1}$

- NB1: The CCPR1L is 10 bit wide, here is used in 8 bit mode.
- NB2: the PIC has two PWM modules, in which the timer2 module is common for both. Thus the PIC is able to generate 2 PWM signal but with same period.







PWM signal generation

REGISTER 11-1: CCP1CON: ENHANCED CCP1 CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
P1M1	P1M0	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0
bit 7							bit 0

bit 7-6 **P1M<1:0>:** PWM Output Configuration bits
If CCP1M<3:2> = 00, 01, 10:
xx = P1A assigned as Capture/Compare input; P1B, P1C, P1D assigned as port pins
If CCP1M<3:2> = 11:
00 = Single output; P1A modulated; P1B, P1C, P1D assigned as port pins
01 = Full-Bridge output forward; P1D modulated; P1A active; P1B, P1C inactive
10 = Half-Bridge output; P1A, P1B modulated with dead-band control; P1C, P1D assigned as port pins
11 = Full-Bridge output reverse; P1B modulated; P1C active; P1A, P1D inactive

bit 5-4 **DC1B<1:0>:** PWM Duty Cycle Least Significant bits
Capture mode:
Unused.
Compare mode:
Unused.
PWM mode:
These bits are the two LSbs of the PWM duty cycle. The eight MSbs are found in CCPR1L.

bit 3-0 **CCP1M<3:0>:** ECCP Mode Select bits
0000 = Capture/Compare/PWM off (resets ECCP module)
0001 = Unused (reserved)
0010 = Compare mode, toggle output on match (CCP1IF bit is set)
0011 = Unused (reserved)
0100 = Capture mode, every falling edge
0101 = Capture mode, every rising edge
0110 = Capture mode, every 4th rising edge
0111 = Capture mode, every 16th rising edge
1000 = Compare mode, set output on match (CCP1IF bit is set)
1001 = Compare mode, clear output on match (CCP1IF bit is set)
1010 = Compare mode, generate software interrupt on match (CCP1IF bit is set, CCP1 pin is unaffected)
1011 = Compare mode, trigger special event (CCP1IF bit is set; CCP1 resets TMR1 or TMR2
1100 = PWM mode; P1A, P1C active-high; P1B, P1D active-high
1101 = PWM mode; P1A, P1C active-high; P1B, P1D active-low
1110 = PWM mode; P1A, P1C active-low; P1B, P1D active-high
1111 = PWM mode; P1A, P1C active-low; P1B, P1D active-low

REGISTER 11-2: CCP2CON: CCP2 CONTROL REGISTER

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DC2B1	DC2B0	CCP2M3	CCP2M2	CCP2M1	CCP2M0
bit 7							bit 0

bit 7-6 **Unimplemented:** Read as '0'

bit 5-4 **DC2B<1:0>:** PWM Duty Cycle Least Significant bits
Capture mode:
Unused.
Compare mode:
Unused.
PWM mode:
These bits are the two LSbs of the PWM duty cycle. The eight MSbs are found in CCPR2L.

bit 3-0 **CCP2M<3:0>:** CCP2 Mode Select bits
0000 = Capture/Compare/PWM off (resets CCP2 module)
0001 = Unused (reserved)
0010 = Unused (reserved)
0011 = Unused (reserved)
0100 = Capture mode, every falling edge
0101 = Capture mode, every rising edge
0110 = Capture mode, every 4th rising edge
0111 = Capture mode, every 16th rising edge
1000 = Compare mode, set output on match (CCP2IF bit is set)
1001 = Compare mode, clear output on match (CCP2IF bit is set)
1010 = Compare mode, generate software interrupt on match (CCP2IF bit is set, CCP2 pin is unaffected)
1011 = Compare mode, trigger special event (CCP2IF bit is set, TMR1 is reset and A/D conversion is started if the ADC module is enabled. CCP2 pin is unaffected.)
11xx = PWM mode.



Activity 13 C: Timer 2/PWM application

- In this Activity, we will generate two PWM signals with different duty cycles. As previously mentioned that these signal cannot be generated at different frequency because the timer2 module is common for both CCP modules.

- We will assume that the used oscillator is 4Mhz, the PR2 = 255 (highest value) and the timer 2 prescaler is 16, the period is then calculated as:

$$T = (PR2 + 1) \times T_{ins} \times prescaler = 256 \times 1\mu s \times 16 = 4096\mu s$$

- The frequency will be 1/T = 1000000/4096 = 244.14 Hz

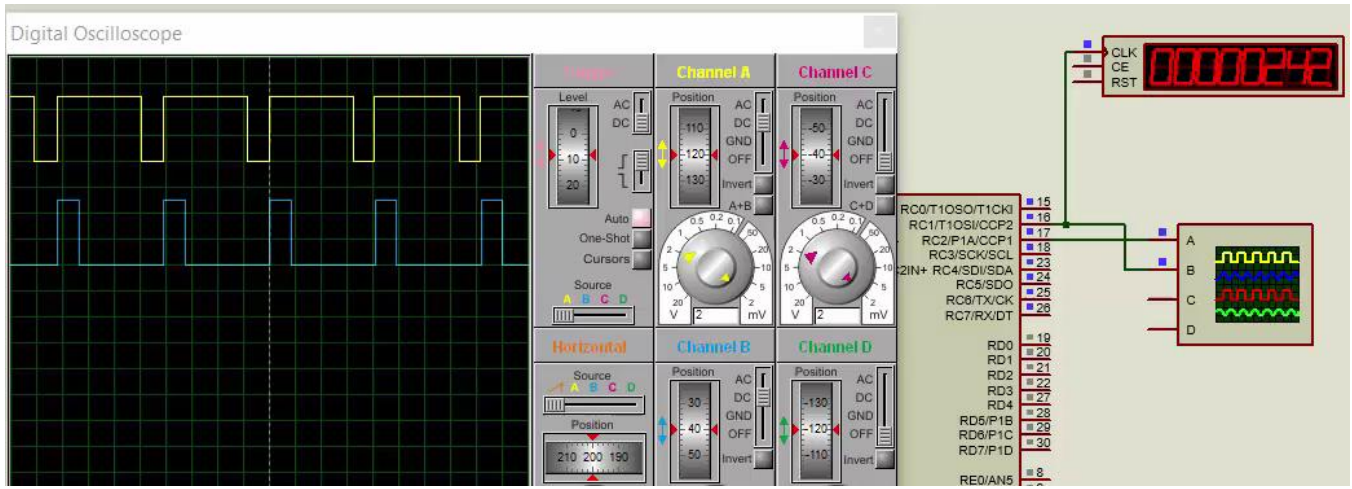
- Changing CCPR1L and CCPR2L from 0 to 255 will change the Ton time from 0 to 4095µs, i.e. from 0 to 100% duty cycle.

- In this activity we'll change CCPR1L from 0 to 255 and CCPR2L from 255 to 0 with 10ms step.

Activity 13 C: Timer 2/PWM application

- In the setup function, PR2, Timer2, CCPxCON and TRISC are configured.

```
23  #include <xc.h>
24
25  #define _XTAL_FREQ 4000000  // used crystal is 4MHz
26
27  void setup() {
28      PORTC = 0; // clear port
29      OSCCON = 0b01100000; // select 4MHz oscillator
30      TRISC = 0b00000000; // and as out
31      PR2 = 255;          // define the period
32      T2CON = 0b00000111; // prescaler 16, timer 2 on
33      CCP1CON = 0b00001100;   // CCP1 in PWM mode
34      CCP2CON = 0b00001100;   // CCP2 in PWM mode
35  }
36
37  void main(void) {
38      setup();
39      CCPR1L = 0;     // starting from 0% duty cycle
40      CCPR2L = 255;   // starting from 100% duty cycle
41      while (1) {
42          CCPR1L++;   // increment DC
43          CCPR2L--;   // decrement DC
44          __delay_ms(10); // wait 10ms
45      }
46  }
```





1. Calculate the PR2 value in order to have f = 500 Hz, using an 8MHz crystal and a prescaler 16.
2. Having a 10MHz crystal and a prescaler of 1, calculate the best frequency of the PWM signal (best frequency happens at PR2 = 255)
3. What is the average voltage of a pulsating 5 volt signal at 500Hz with T_{on}=500µs
4. It is required to have 3V from a 12 volt source. The PWM signal has 1000Hz, calculate PR2 and CCPR1L of a PIC running at 4Mhz. Select the best prescaler.
5. What to put in PR2 to generate a flag after 500 microseconds from the timer 2 module of a PIC running at 1Mhz. Assume that Prescaler is 1 and Postscaler is 5.



Week 7

Lab 6 / LO3

Timer 2/PWM



Activity 13 C: Timer 2/PWM application

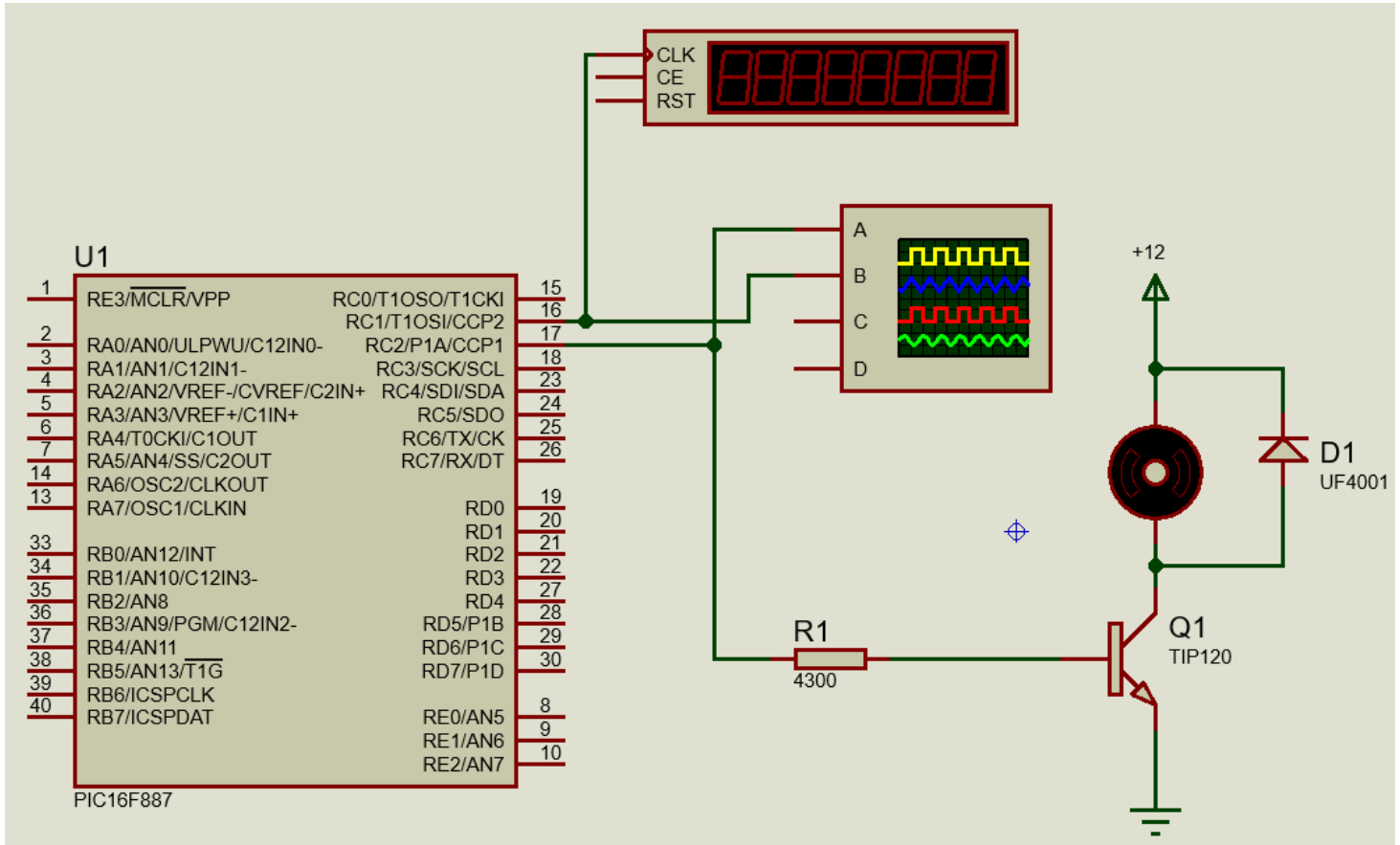
- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. PIC programmer
  4. Oscilloscope 2 channels
  5. Power supply 5V and 12V
  6. Breadboard wires (male-male)

- Required components:
  1. PIC 16F887
  2. Resistor from 1k x 2
  3. NPN transistor TIP120 or any equivalent (or use 2n2222 in case of small motor)
  4. Fast recovery diode UF4001 or any equivalent (use any regular diode if you don't have)
  5. Any DC motor (measure the motor current before choosing the transistor)

Activity 13 C: Timer 2/PWM application

- Same code as Activity 13C but with different components

  The PWM signal with different duty cycle is usually applied on DC motors. In this activity you will see the effect of duty cycle on the speed of the motor.



End of LO3

LO4



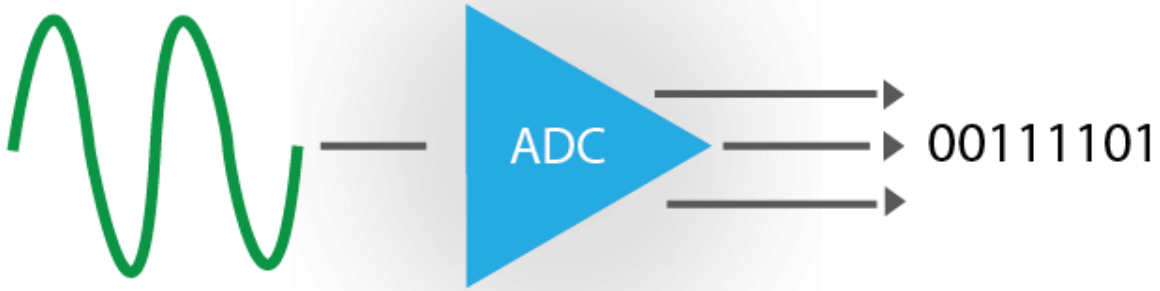
Week 9

Lecture 15 / LO4

Analog To Digital Conversion

Analog to Digital Conversion

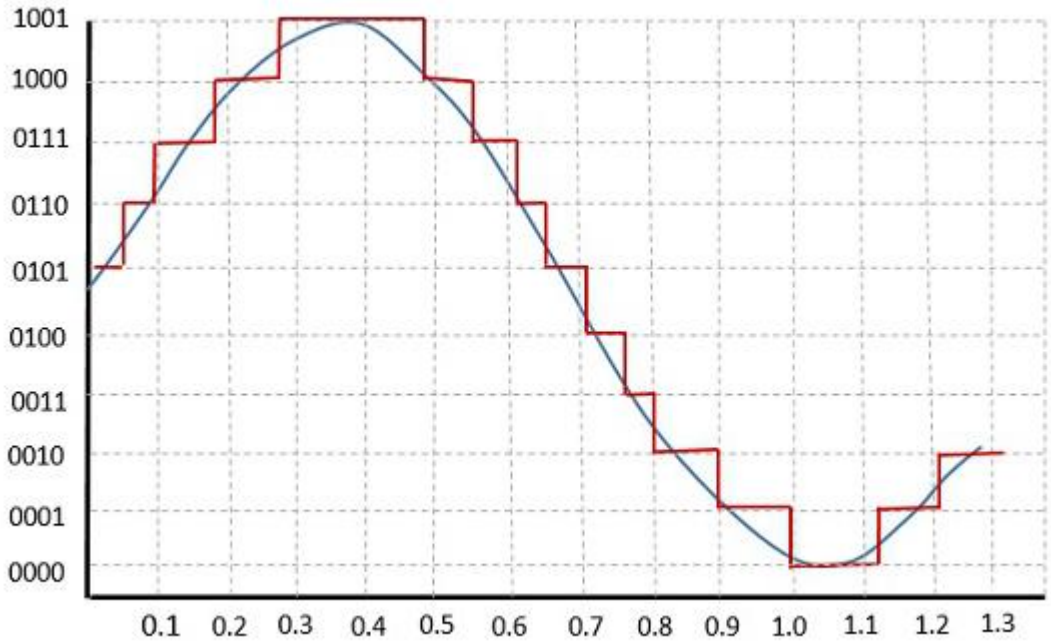
- Analog-to-digital conversion (ADC) is an electronic process in which a continuously variable, or analog, signal is changed into a multilevel digital signal without altering its essential content.
- An analog-to-digital converter changes an analog signal that's continuous in terms of both time and amplitude to a digital signal that's discrete in terms of both time and amplitude.
- The analog input to a converter consists of a voltage that varies among a theoretically infinite number of values.
- Examples are sine waves, the waveforms representing human speech and the signals from a conventional television camera.
- The output of the analog-to-digital converter has defined levels or states. The number of states is almost always a power of two -- that is, 2, 4, 8, 16, etc. The simplest digital signals have only two states and are called binary. All whole numbers can be represented in binary form as strings of ones and zeros.



Analog to Digital Conversion

- Two things to be considered when converting an analog signal into digital
  1. Quantization
  2. Sampling rate

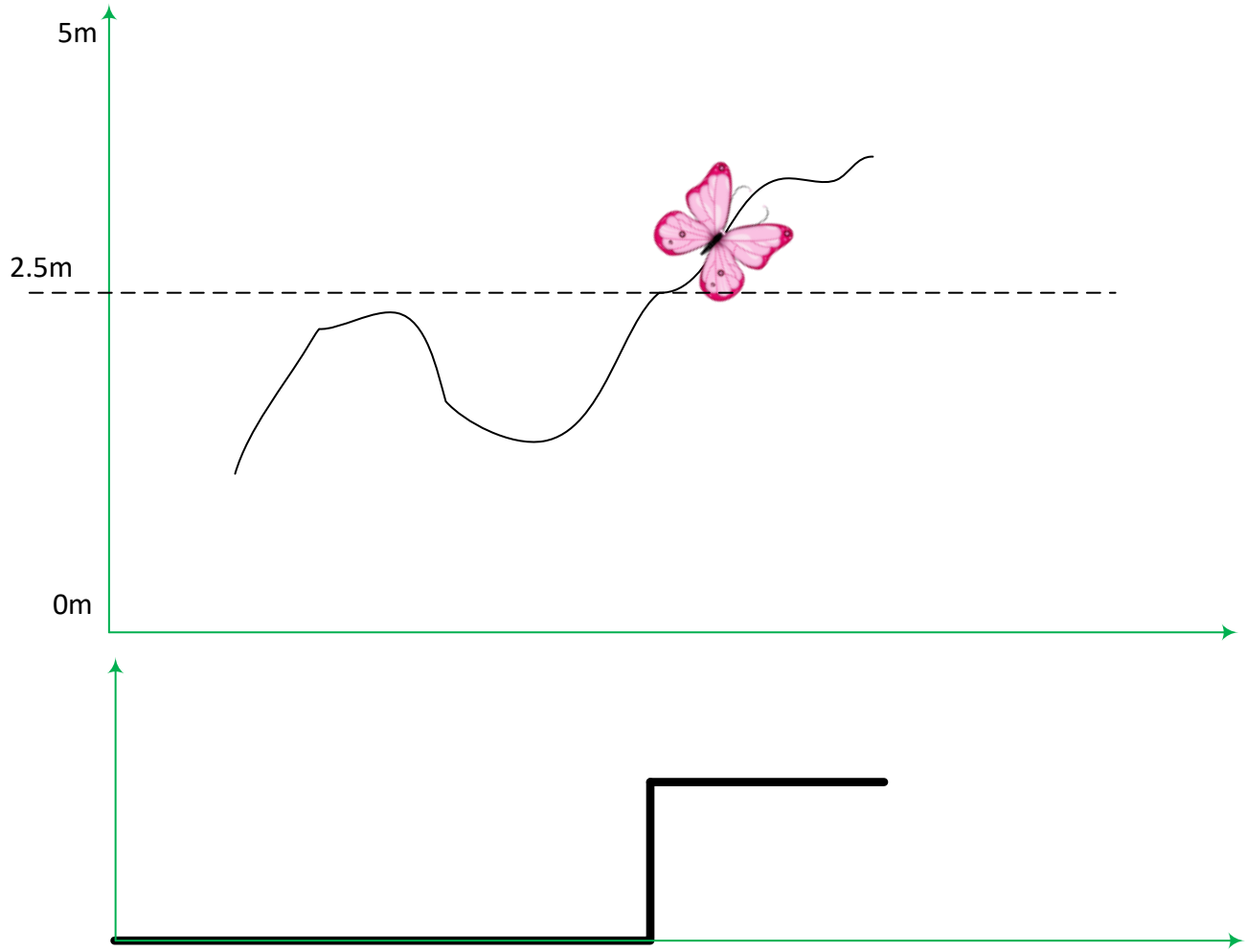
- Quantization is the measure of quantity (voltage) of the analog signal. Higher quantization accuracy requires higher resolution and number of bits of the A/D converter. (These represent the segments on the Y axis- usually voltage)
- Sampling rate is how much shots can the A/D converter takes of the analog signal. Highest sample per sec rate assure a best recovery of information included in the analog signal.(These represent the segments on the X axis- usually time)



Analog to Digital Conversion

Quantization

- Assume for example that a butterfly is moving in a room of 5m height. At one moment I asked a kid to describe me the position of the butterfly in the space… the silly answer will be HIGH or LOW.
- Thus the silly description of the analog position of the butterfly is in its digital form HIGH or LOW.
- High means above the reference located virtually in between the 0 and max height.
- Low means below the reference
-
- This is actually an analog to digital conversion using 1 bit.
- 1 bit can be 0 (LOW) or 1 (HIGH)

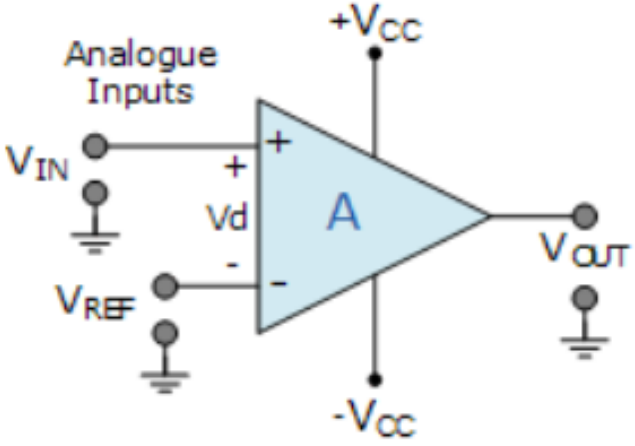


Analog to Digital Conversion

- Actually, the opamp in comparator mode is the native analog to digital convertor with 1 bit.

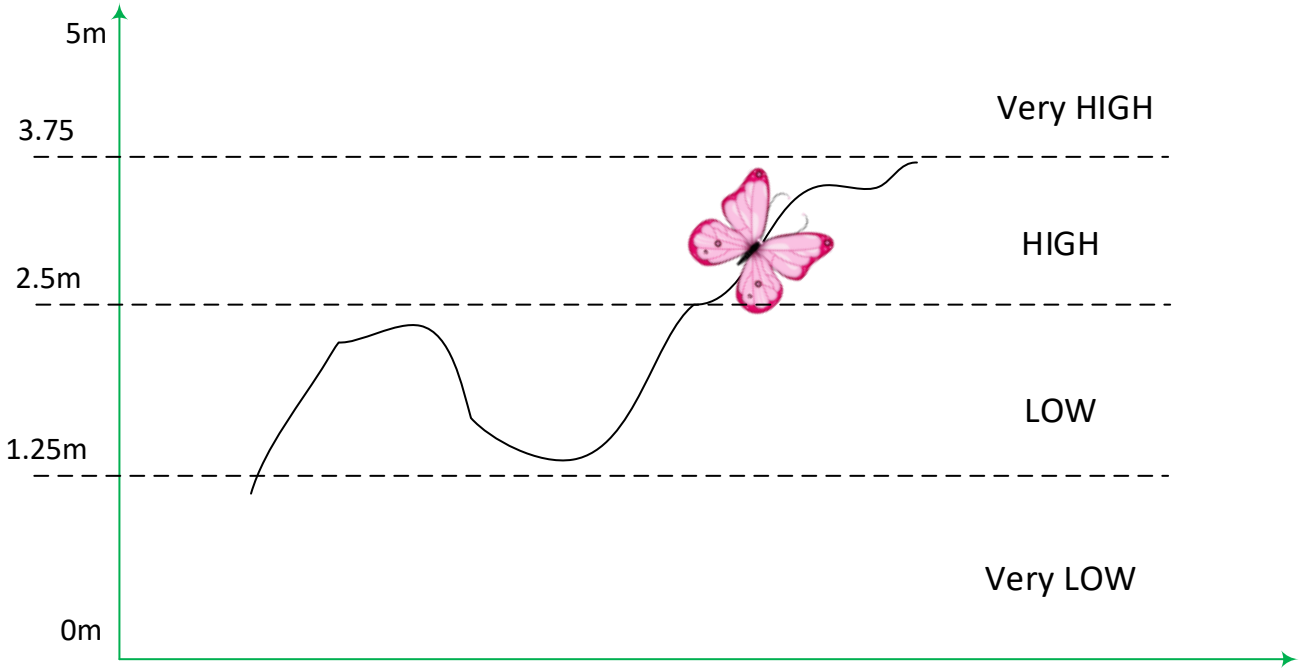
- The comparator compares (as the kid) the voltage on V+ (position of the butterfly) with the reference voltage applied on V-. The Output is HIGH or LOW depends on who's bigger

- If V+ > Vref => output is high
- If V+ < Vref => output is low



Analog to Digital Conversion

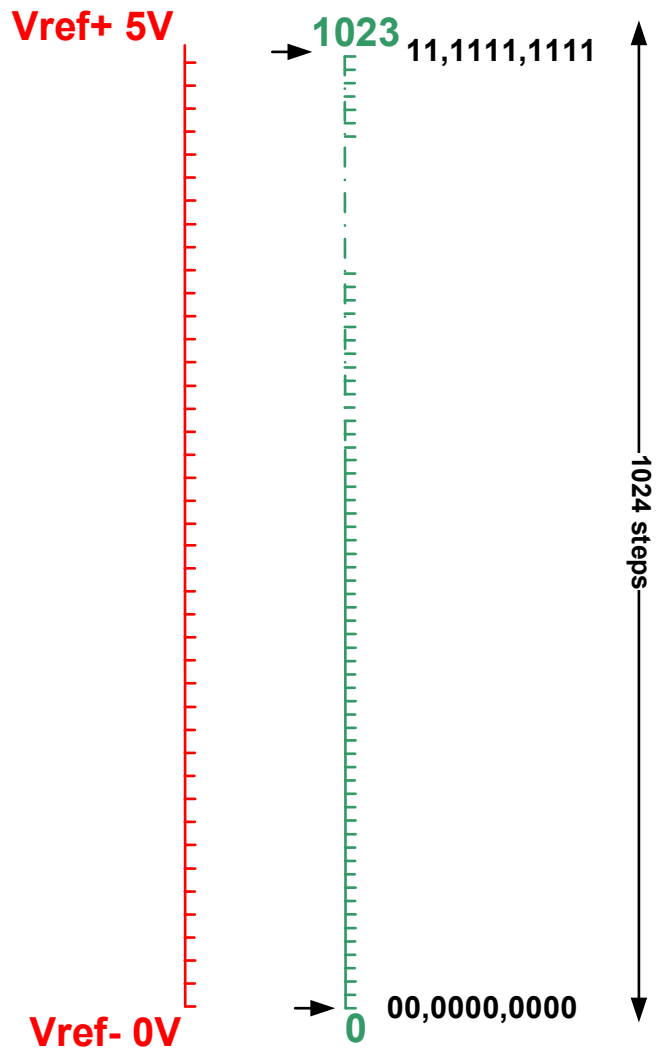
- For a better assumption now, we'll divide the room into 4 portion through 3 references.
  - From 0 to 1.25m is very low
  - From 1.25 to 2.5m is low
  - From 2.5 to 3.75 is high
  - From 3.75 to 5 is very high
- Thus the position is described in 2 bits which can be:
  1. 00 for very low
  2. 01 for low
  3. 10 for high
  4. 11 for very high



We can conclude that the precision increases as the number of bits increases. Thus, the resolution of the ADC is the number of bits it uses to digitize the input samples. For an n bit ADC the number of discrete digital levels that can be produced is 2^n. Thus, a 2 bit digitizer can resolve 2^2 or 4 levels.

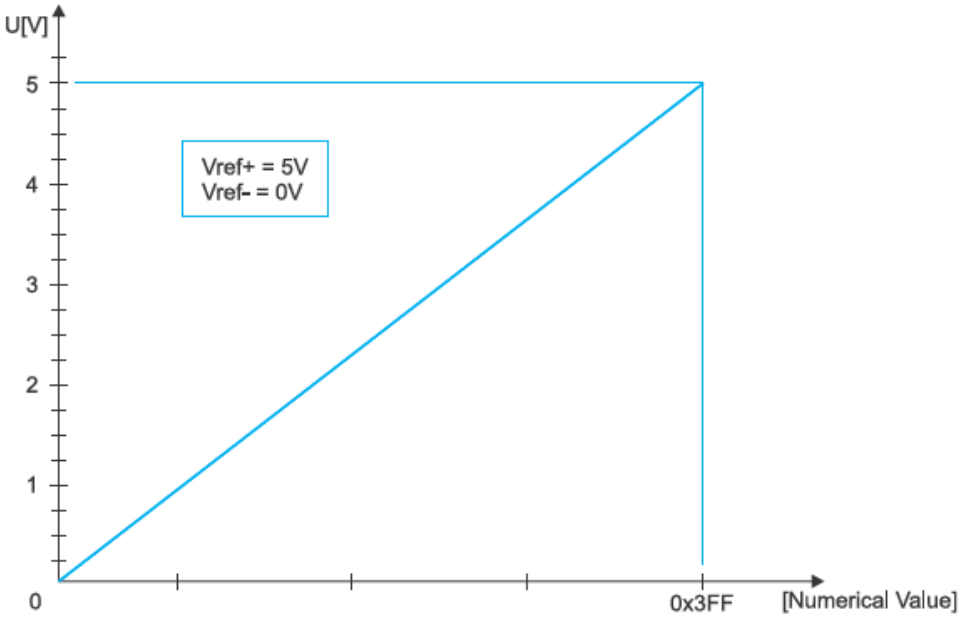
Analog to Digital Conversion

- The PIC16F887 ADC resolution is 10 bits, i.e., it can recognize the analog signal with 1024 different level.
- Assuming for now that the range of voltage is from 0 to 5, the digital value range will be from 0 to 1023, or in binary between 00,0000,0000 to 11,1111,1111.
- Thus the ADC can recognize a variation of 5/1024 = 4.88mV change in the analog signal.
- Usually, the border of the voltages is called V_{ref}^- and V_{ref}^+. We are assuming for now that they are 0 and 5.



Analog to Digital Conversion

- The PIC16F887 also has 14 analog inputs. They enable the microcontroller to precisely measure its voltage and convert it into numerical value, i.e. digital format.
- One of the most important analog modules within the microcontroller is an A/D converter which has the following features:
  - Conversion is performed by applying the method of successive approximation;
  - There are 14 separate analog inputs connected to the microcontroller port pins;
  - The A/D converter converts an analog input signal into a 10-bit binary number;
  - The resolution, i.e. the quality of conversion may be adjusted to various needs by selecting voltage references Vref- and Vref+.



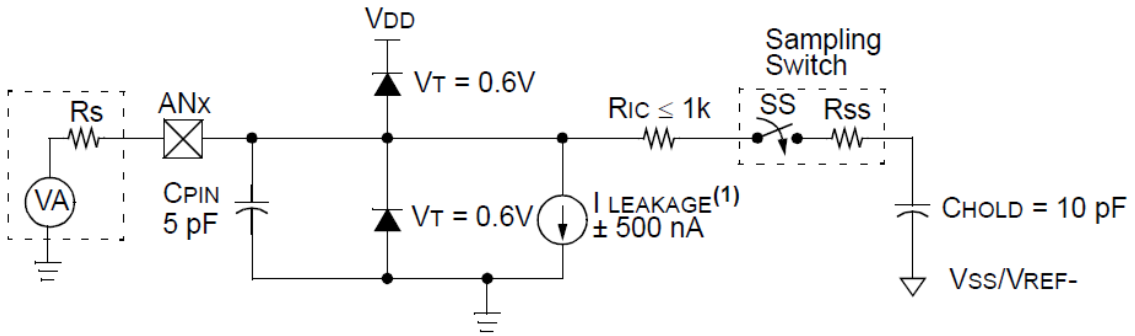
Analog to Digital Conversion

- **Sampling**

- Sampling rate is the speed of reading the information from the signal. Usually, sampling is the time required to take a copy of the signal, save it in a holding capacitor then converting it into digital. The second shot cannot be proceeded until the previous shot is converted.
- This scenario is similar to the diaphragm of a camera. It opens to take a shot, then closes to start the image processing. The next image cannot be taken until the image is completely processed.



- The circuit model of the holding circuit is given on the right. The SS switch closed to charge the capacitor Chold. The time required to charge it depends on its value and the series resistors connected to the source. This time is called acquisition time and it is about 20µs.

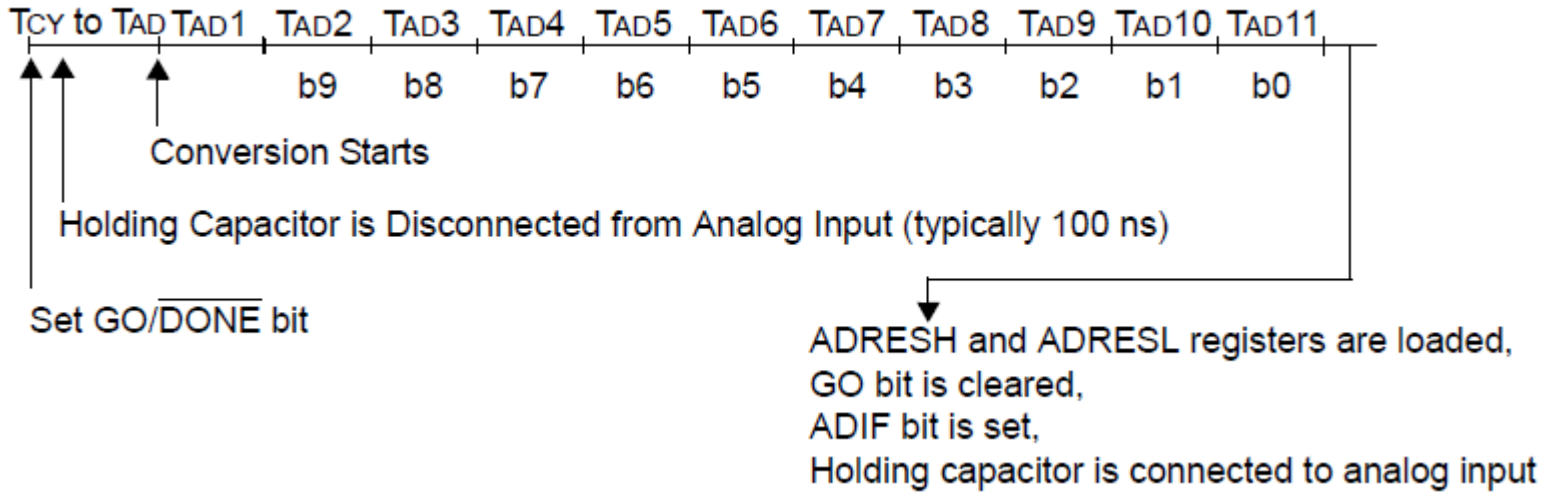


Analog to Digital Conversion

- **Quantazation**

- After taking the shot, it is time to process the voltage, i.e., find its quantity using the sets of comparator inside the A2D convertor.

- The time required to convert 1 bit is called Tad.
- Since the ADC is 10 bits, it needs 10Tad to make the conversion. Also 2Tad are required to open the SS switch and prepare the conversion. So in total 12Tad.
- The minimum time for 1Tad is 1.6µs, thus in total quantization time is 19.2µs



Analog to Digital Conversion

- **Sampling**

- The manufacturer provides the below table for different Tad at different oscillator frequency.
- It is clear that the minimum time for 4MHz oscillator is 2µs.

- As a result, the total time required to take a shot and make the process is the sum of the acquisition time and the quantization time $= 20 + 2x12 \approx 44$ µs / sample
- This means that ideally the PIC is able to read almost 22700 samples in 1 second. Based on the Nyquist rate criteria, the PIC is able to read a signal with a maximum frequency of 11kHz

TABLE 9-1: ADC CLOCK PERIOD (TAD) Vs. DEVICE OPERATING FREQUENCIES (VDD ≥ 3.0V)

ADC Clock Period (TAD)		Device Frequency (Fosc)			
ADC Clock Source	ADCS<1:0>	20 MHz	8 MHz	4 MHz	1 MHz
Fosc/2	00	100 ns(2)	250 ns(2)	500 ns(2)	2.0 µs
Fosc/8	01	400 ns(2)	1.0 µs(2)	2.0 µs	8.0 µs(3)
Fosc/32	10	1.6 µs	4.0 µs	8.0 µs(3)	32.0 µs(3)
FRC	11	2-6 µs(1,4)	2-6 µs(1,4)	2-6 µs(1,4)	2-6 µs(1,4)

Legend: Shaded cells are outside of recommended range.



Analog to Digital Conversion

- **Sampling**

Nyquist Sampling Rate

- In signal processing, the Nyquist rate is the minimum sampling rate required to accurately rebuild the original analog signal from its digital representation.
- To avoid errors, the rate at which you sample a signal must exceed twice the highest meaningful frequency contained within the signal.

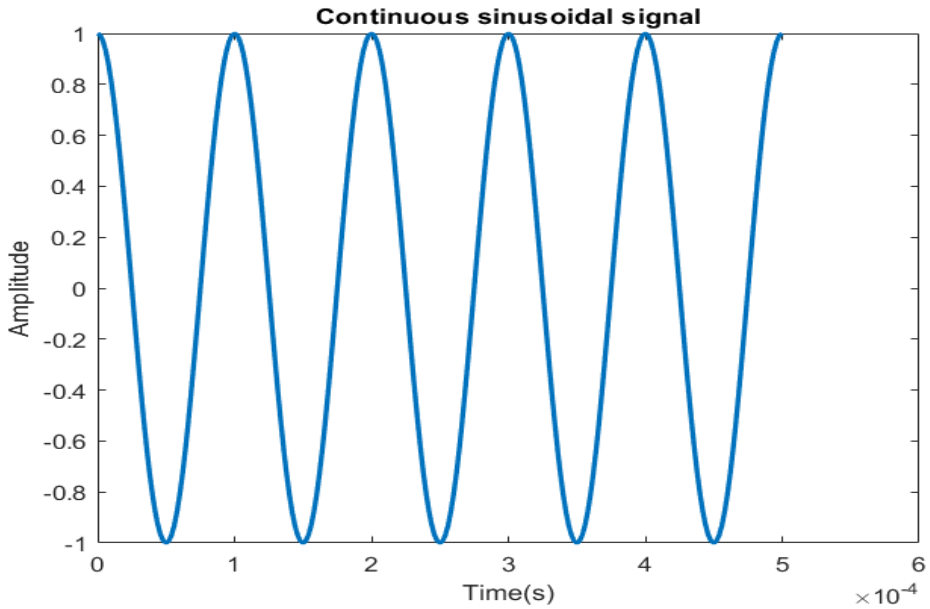
$$F_{sampling} > 2F_{max}$$

- Or, if the sampling rate is fixed (*Fsampling*), then the fastest signal that can be sampled must have a maximum meaningful frequency less than half *Fsampling* :

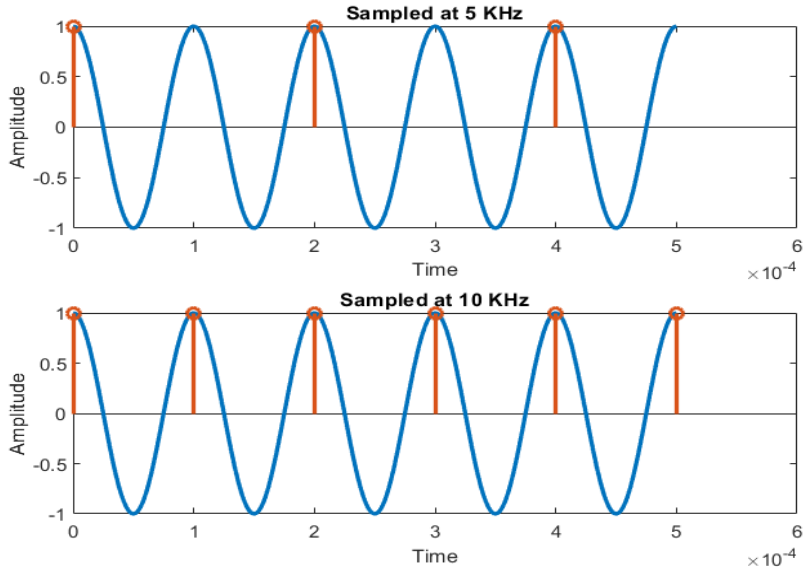
$$F_{max} < \frac{F_{sampling}}{2}$$

Analog to Digital Conversion

- **Sampling**



Here is a **10 KHz signal**



If we sample the signal at 5 KHz and 10 KHz, the signal will look like as shown below (the brown points)

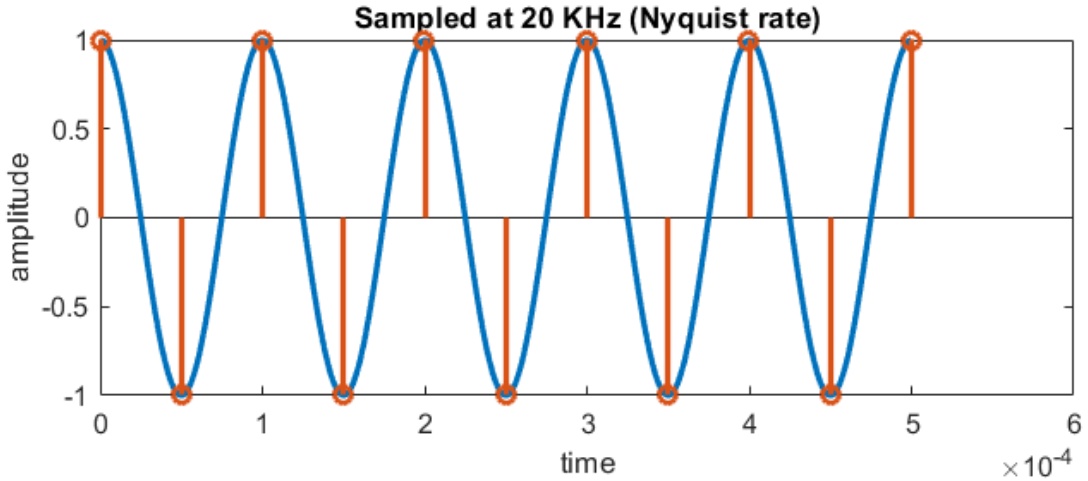
It is clear that we can't get any useful information from the sampled signal.

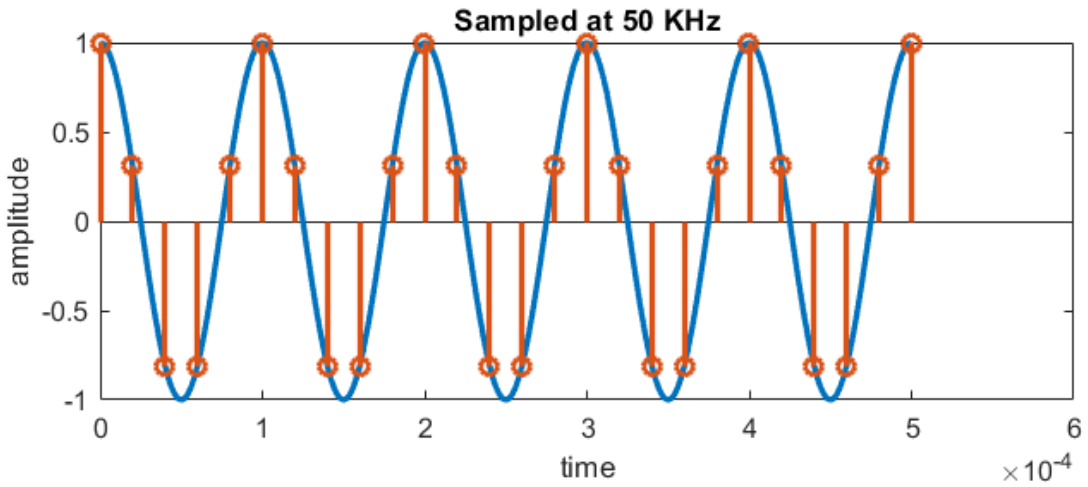
Analog to Digital Conversion

- **Sampling**

Now, to get some information about the signal, Nyquist said that we must sample the signal at least **2 x max signal frequency**. Now, here are the signals sampled at the **Nyquist frequency** and more than it:

It can be seen that by sampling at the Nyquist rate, we can get the frequency information about the signal. However, to faithfully reconstruct the signal, we have to increase the sampling rate even more.





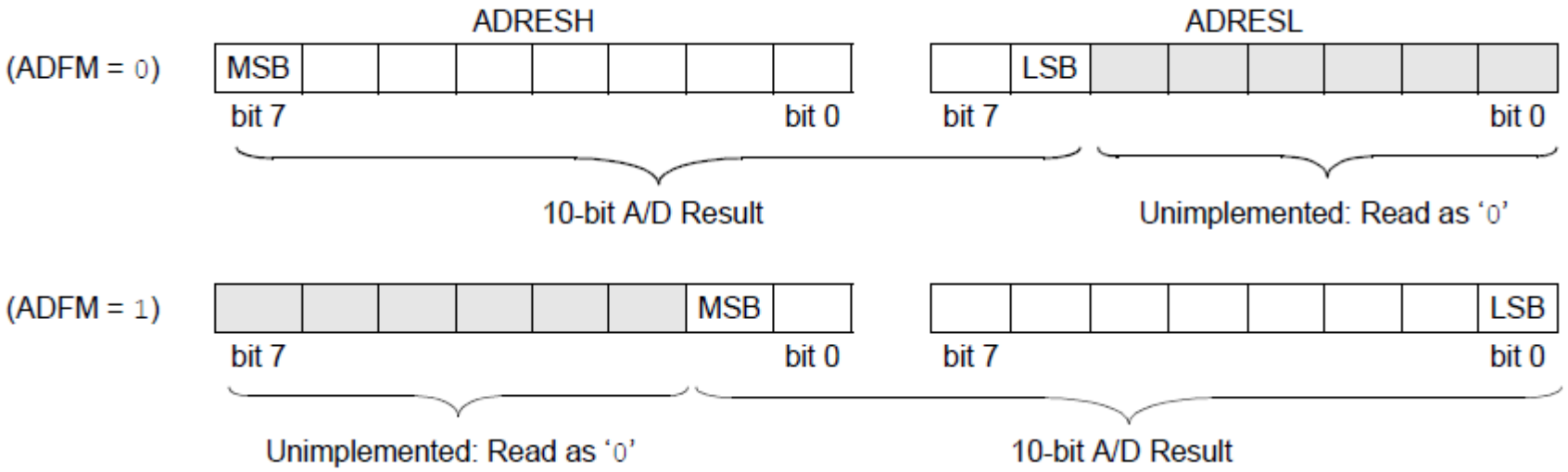
Analog to Digital Conversion

- **AD results**

- After doing the conversion, the PIC fetches the result in 10 bit format. As the RAM of the PIC is only 8bit wide, the A2D result is saved in two registers called ADRESH and ADRESL (Analog to Digital RESult High and Low).
- In fact we have two options.

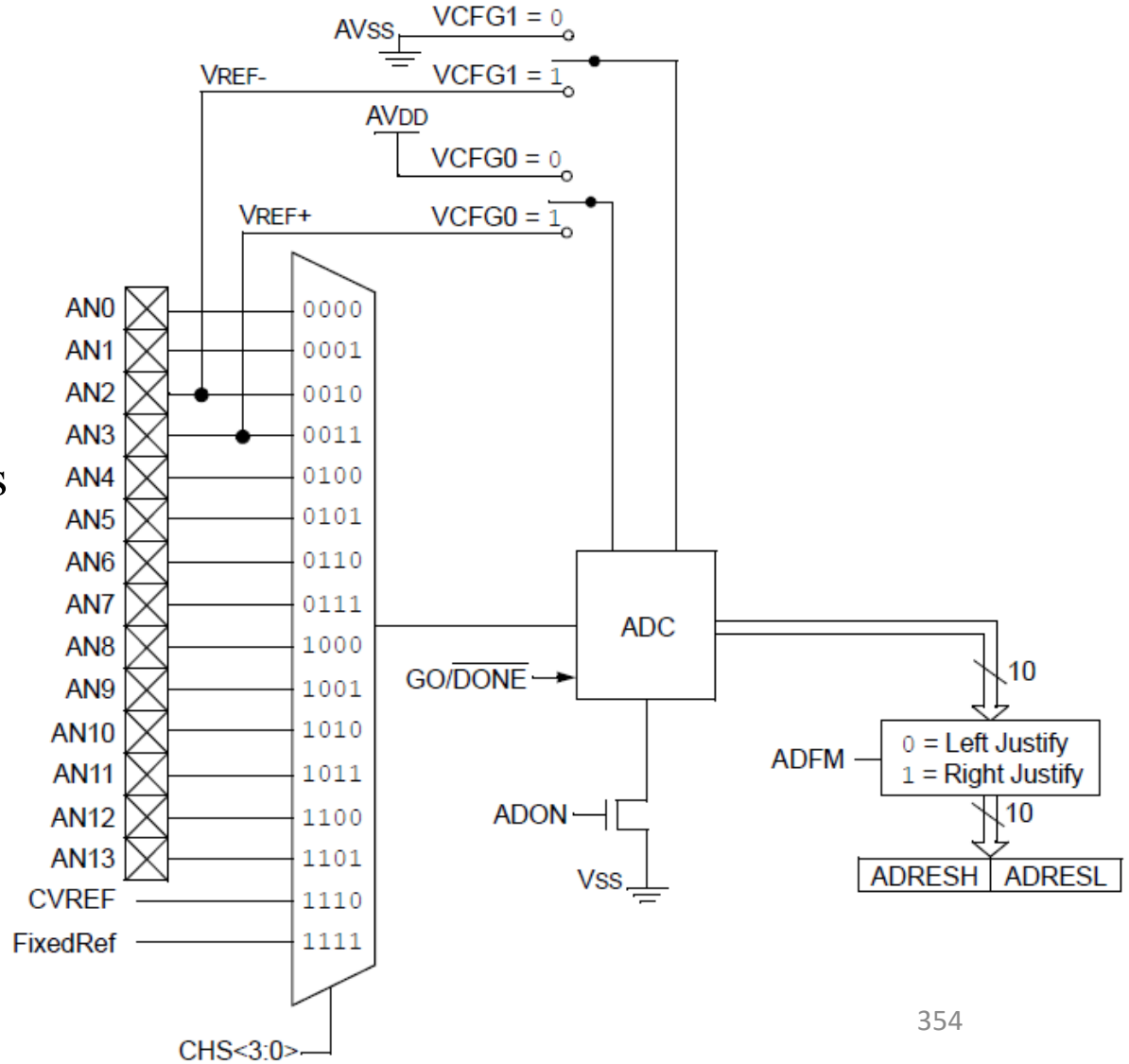
1. Save the 8 higher bits in ADRESH and the 2 remaining in ADRESL (Left justified) or
2. Saves the higher 2 bits in ADRESH and the 8 lower in ADRESL (Right justified)

- Options are selected by ADFM bit



Analog to Digital Conversion

- Although the Pic has 14 Analog channels, the drawback of the module is that it has only one converter. Which means the PIC can process only 1 conversion at a time. The reading of channel is to be done by sequence.
- By default, the Vref+ and Vref- are connected to the VDD and VSS. They can be connected the AN3 and AN2 for external references.
- From the block diagram we can see that the analog input are multiplexed through CHS<3:0>. When the channel is selected, the Chold capacitor is connected. It means that 20µs delay is required before starting the conversion.
- The ADC is controlled by the GO/DONE bit to start the conversion. This bit is cleared automatically when the conversion is done.
- ADON is used to turn on the converter.
- The results are placed in ADRESH and ADRESL based on the ADFM bit.





Analog to Digital Conversion

REGISTER 9-1: ADCON0: A/D CONTROL REGISTER 0

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADCS1	ADCS0	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

bit 7-6 **ADCS<1:0>:** A/D Conversion Clock Select bits
00 = Fosc/2
01 = Fosc/8
10 = Fosc/32
11 = FRC (clock derived from a dedicated internal oscillator = 500 kHz max)

bit 5-2 **CHS<3:0>: Analog Channel Select bits**
0000 = AN0
0001 = AN1
0010 = AN2
0011 = AN3
0100 = AN4
0101 = AN5
0110 = AN6
0111 = AN7
1000 = AN8
1001 = AN9
1010 = AN10
1011 = AN11
1100 = AN12
1101 = AN13
1110 = CVREF
1111 = Fixed Ref (0.6V fixed voltage reference)

bit 1 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit
1 = A/D conversion cycle in progress. Setting this bit starts an A/D conversion cycle.
This bit is automatically cleared by hardware when the A/D conversion has completed.
0 = A/D conversion completed/not in progress

bit 0 **ADON:** ADC Enable bit
1 = ADC is enabled
0 = ADC is disabled and consumes no operating current

REGISTER 9-2: ADCON1: A/D CONTROL REGISTER 1

R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0
ADFM	—	VCFG1	VCFG0	—	—	—	—
bit 7							bit 0

bit 7 **ADFM:** A/D Conversion Result Format Select bit
1 = Right justified
0 = Left justified

bit 6 **Unimplemented:** Read as '0'

bit 5 **VCFG1:** Voltage Reference bit
1 = VREF- pin
0 = VSS

bit 4 **VCFG0:** Voltage Reference bit
1 = VREF+ pin
0 = VDD

bit 3-0 **Unimplemented:** Read as '0'

Lecture

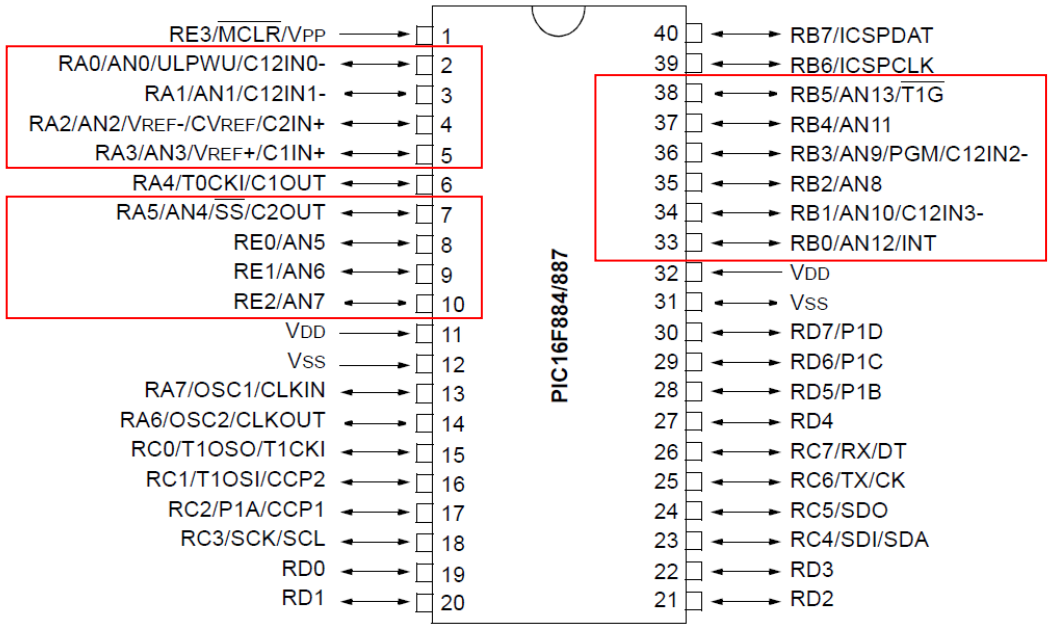
Analog to Digital Conversion

- Not to forget that ANSEL and ANSELH are the two registers dedicated to analog/digital pin configuration.

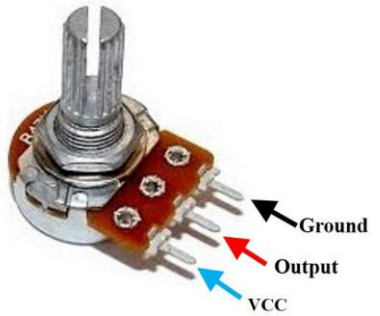
TABLE 9-2: SUMMARY OF ASSOCIATED ADC REGISTERS

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
ADCON0	ADCS1	ADCS0	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON	0000 0000	0000 0000
ADCON1	ADFM	—	VCFG1	VCFG0	—	—	—	—	0-00 ----	-000 ----
ANSEL	ANS7	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0	1111 1111	1111 1111
ANSELH	—	—	ANS13	ANS12	ANS11	ANS10	ANS9	ANS8	--11 1111	--11 1111
ADRESH	A/D Result Register High Byte								xxxx xxxx	uuuu uuuu
ADRESL	A/D Result Register Low Byte								xxxx xxxx	uuuu uuuu
INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000x
PIE1	—	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	-000 0000	-000 0000
PIR1	—	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	-000 0000	-000 0000
PORTA	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0	xxxx xxxx	uuuu uuuu
PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx	uuuu uuuu
PORTE	—	—	—	—	RE3	RE2	RE1	RE0	---- xxxx	---- uuuu
TRISA	TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	1111 1111	1111 1111
TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111	1111 111
TRISE	—	—	—	—	TRISE3	TRISE2	TRISE1	TRISE0	---- 1111	---- 111

Legend: x = unknown, u = unchanged, – = unimplemented read as '0'. Shaded cells are not used for ADC module.



Activity 14 : Reading Potentiometer on AN0

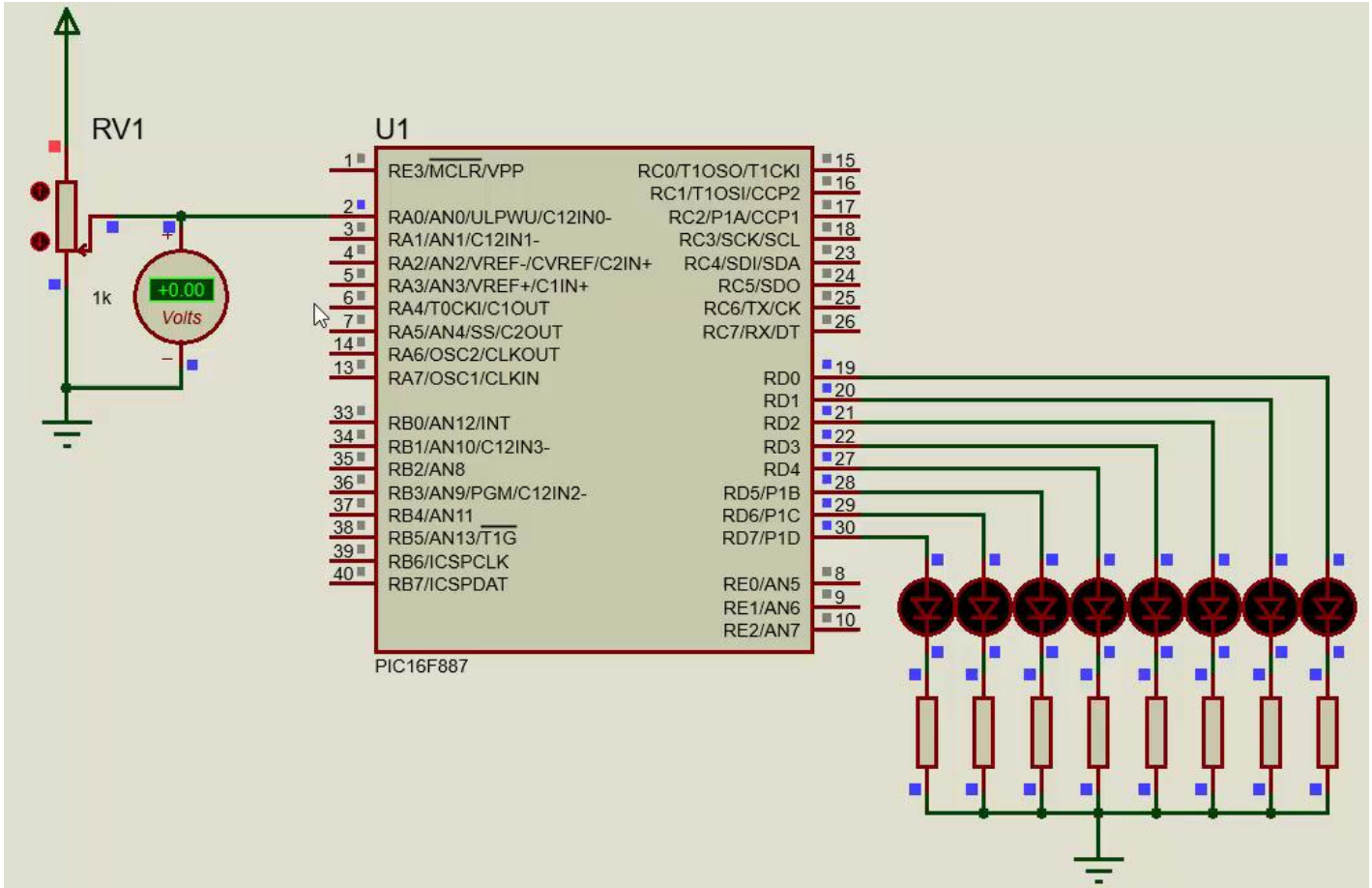


- In this activity, we will apply the concept on A2D conversion. It is required to read the analog channel 0 and display the digital value (8 bit only) on PORTD.
- The analog sensor used is a potentiometer. It is a three-terminal resistor with a sliding or rotating contact that forms an adjustable voltage divider.

- The A2D is configured to work at Fosc/8 to get the minimum Tad time (fastest conversion).

- The left justified configuration is used (ADFM =0)
- Only ADRESH will be read and displayed on PORTD

- Only RA0 is selected to be input (trough TRIS) and analog (through ANSEL)





Activity 14 : Reading Potentiometer on AN0

```
28      A1   EQU     0X70
29
30          ORG     0             ; reset vector
31          GOTO    MAIN
32          ORG     0X4           ; interrupt vector
33          RETFIE
34
35      MAIN:
36          CALL    SETUP
37      REPEAT:
38          CALL    READ_AN0      ; read the analog channel
39          MOVWF   PORTD         ; and place result on PORTD
40          GOTO    REPEAT        ; REPEAT CODE
41
42      READ_AN0:
43          MOVLW   01000001B     ; FOSC/8, CH0, ADON
44          MOVWF   ADCON0
45          CALL    DELAY_20US
46          BSF     GO            ; start conversion
47          BTFSC   GO            ; is it done ?
48          GOTO    $-1           ; not yet, retest
49          MOVF    ADRESH, W     ; done, move ADRESH to W
50          RETURN
```

```
52      DELAY_20US:
53          MOVLW   5
54          MOVWF   A1
55          DECFSZ  A1, F
56          GOTO    $-1
57          RETURN
58
59      SETUP:                    ; by default we are in BANK0
60          CLRF    PORTD         ; clear port at the beginning
61          BSF     RP0           ; RP1 IS BY DEFAUT 0
62          MOVLW   01100000B     ; configuring OSCCON register
63          MOVWF   OSCCON        ; to select 4Mhz oscillator
64          MOVLW   00000000B     ; portD is out
65          MOVWF   TRISD
66          BSF     TRISA0        ; RA0 IS INPUT
67          BSF     RP1
68          BSF     ANS0          ; RA0 IS ANALOG
69          CLRF    ADCON1        ; ADFM = 0, VREF+ = VDD,  VREF- = VSS
70          BCF     RP1
71          BCF     RP0           ; access to BANK 0, RP0 = 0 , RP1 = 0
72          RETURN
```

ADC-reading to voltage conversion and vice-versa

- **Relating ADC reading to Voltage**
- The ADC reports a ratio-metric value. This means that the ADC assumes 5V is 1023 (or 255 in case of 8 bits) and anything less than 5V will be a ratio between 5V and 1023.

$$\frac{2^{\#ofbits}-1}{V^+-V^-}=\frac{ADC_{reading}}{V_{in}-V^-}\rightarrow ADC_{reading}=(2^{\#ofbits}-1)\frac{(V_{in}-V^-)}{V^+-V^-}$$

  - V^+ is the upper reference
  - V^- is the lower reference
  - V_{in} is the input voltage
  - $ADC_{reading}$ is the A2D output result in decimal
- Analog to digital conversions are dependent on the system voltage. Because we predominantly use the 10-bit ADC of the PIC on a 5V system, we can simplify this equation slightly:

$$ADC_{reading}=1023\frac{V_{in}}{5}$$

ADC-reading to voltage conversion and vice-versa

- On the other side, if we have the read value from the PIC and we need to figure out what is the input voltage, the following equation can be used

$$V_{in} = \frac{(V^+ - V^-) \times ADC_{reading}}{(2^{\#ofbits} - 1)} + V^-$$

with

$$0 < ADC_{reading} < 2^{\#ofbits} - 1$$

$$V^- < V_{in} < V^+$$



Test your knowledge

1. Calculate the ADC reading of a 3V input on a 4bit A2D having 4V $Vref^+$ and 0V $Vref^-$.
2. Calculate the ADC reading of a 3V input on a 4bit A2D having 2V $Vref^+$ and 0V $Vref^-$.
3. Calculate the ADC reading of a 3V input on a 4bit A2D having 2V $Vref^+$ and -2V $Vref^-$.
4. Calculate the voltage from a sensor when the AD reading of 50 is captured on the output of a 10bit A2D converter with $Vref^+ = 3$ and $Vref^-=2$
5. Calculate the voltage from a sensor when the AD reading of 4095 is captured on the output of a 12bit A2D converter with $Vref^+ = 3$ and $Vref^-=0$.
6. A 10-bit ADC has Vref+ = 4, and Vref- = 1, the converted decimal value is 230. What would be the input voltage ?



Week 9

Lecture

Lecture 16 / LO4

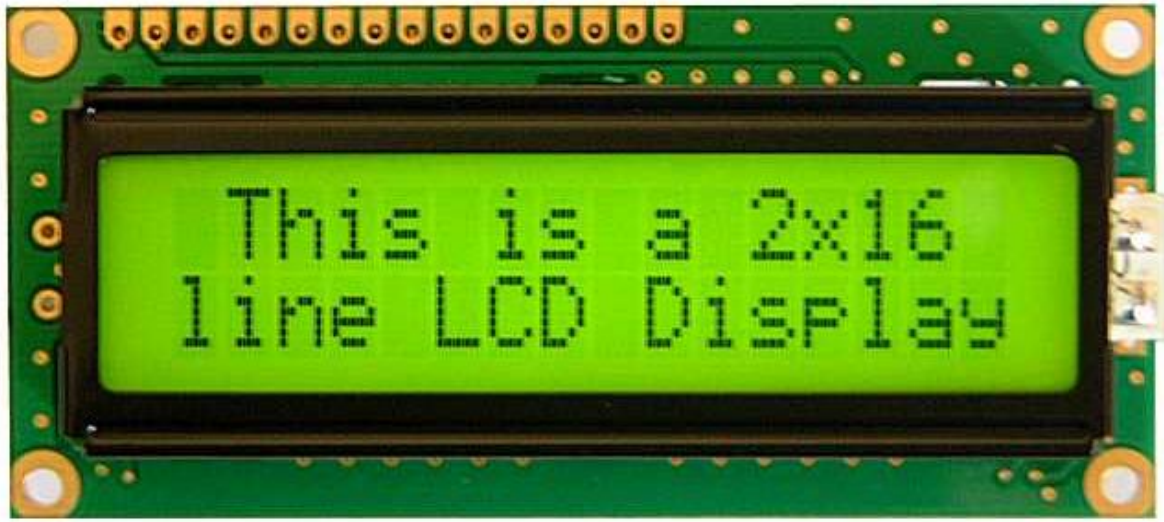
LCD

LCD

- An LCD character display is a unique type of display that can only output individual ASCII characters with fixed size.

- The LCD is controlled by a microcontroller HD44780 from Hitachi.

- Closer look at the display we can notice that there are small rectangular areas composed of 5×8 pixels grid. Each pixel can light up individually, and generates characters within each grid.



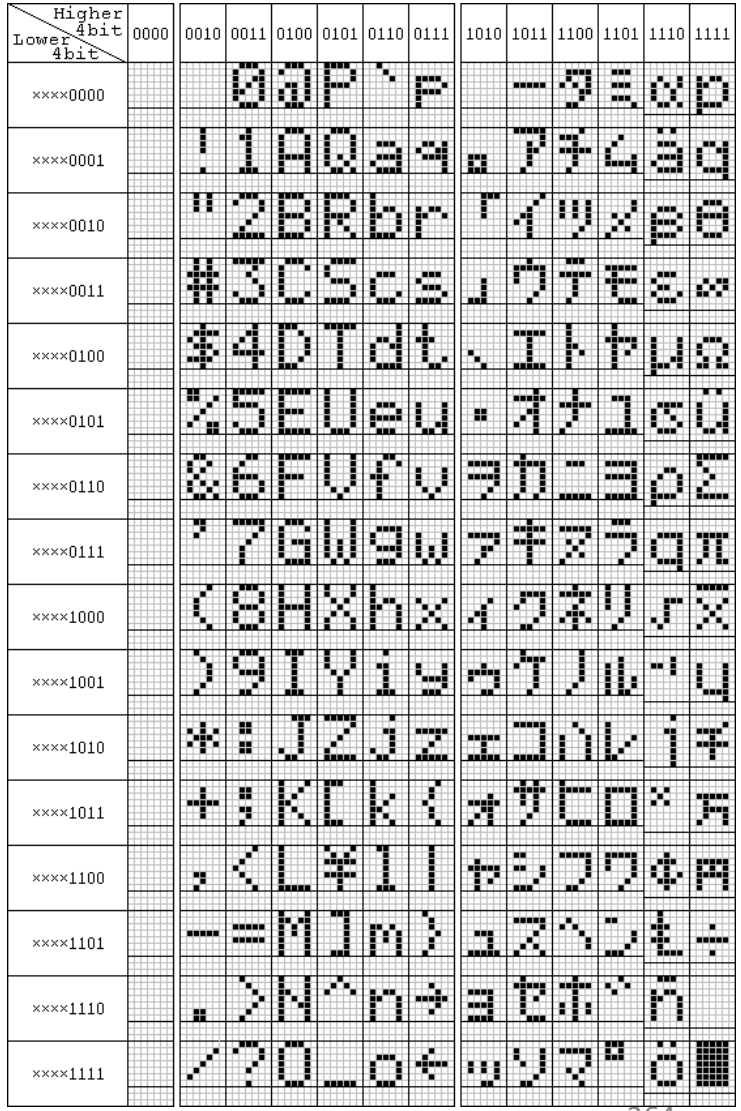




LCD

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]



LCD

- The number of the rectangular areas define the size of the LCD.
- The most popular LCD is the 16×2 LCD, which has two rows with 16 rectangular areas or characters.
- There are other sizes like 8x1, 8x2, 16×1, 16×4, 20×4, 40x4 and so on, but they all work on the same principle. Also, these LCDs can have different background and text color.

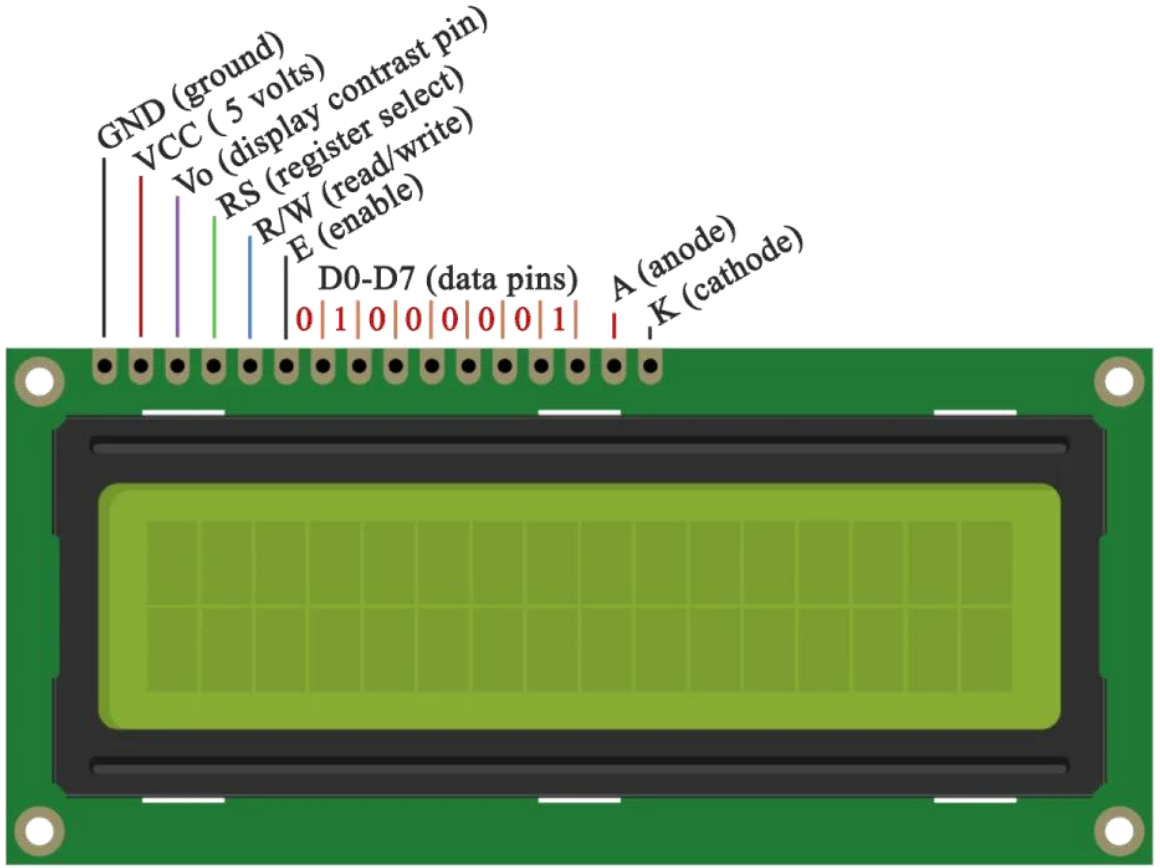




LCD

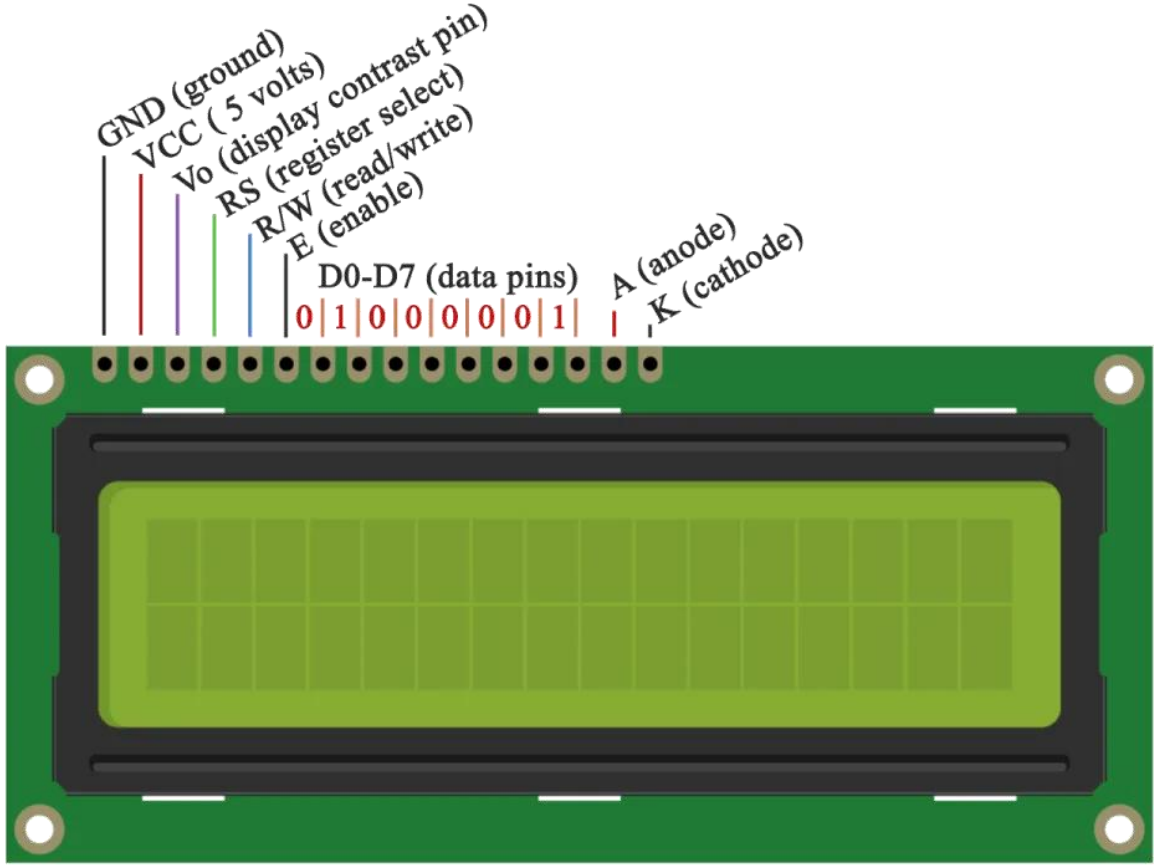
LCD Pinout

- It has 16 pins and the first one from left to right is the ***Ground*** pin.
- The second pin is the ***VCC*** which we connect the 5 volts pin on the Arduino Board.
- Next is the ***Vo*** pin on which we can attach a potentiometer for controlling the contrast of the display.



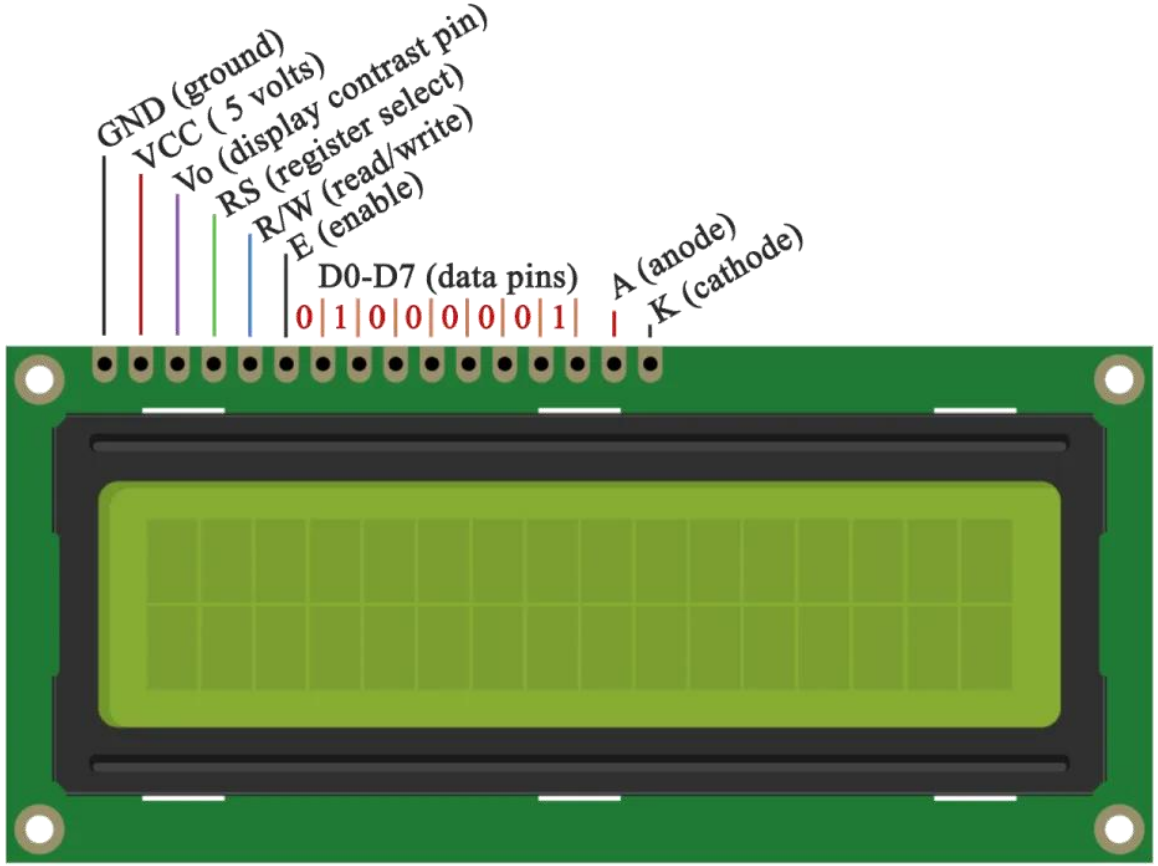
LCD

- ***RS*** pin or register select pin is used for selecting whether we will send commands or data to the LCD.
- For example if the *RS* pin is set on low state or zero volts, then we are sending commands to the LCD like: set the cursor to a specific location, clear the display, turn off the display and so on.
- And when *RS* pin is set on High state or 5 volts we are sending data or characters to the LCD.



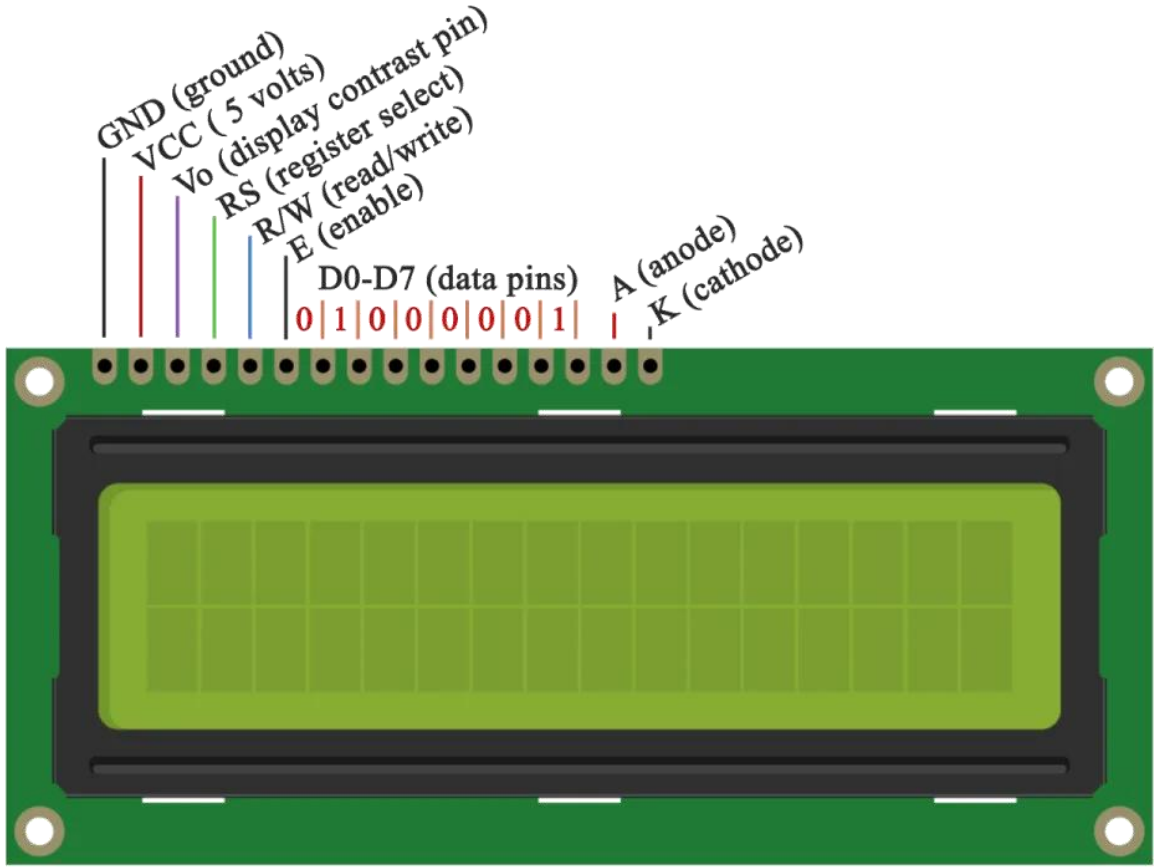
LCD

- ***R/W*** pin which selects the mode whether we will read or write to the LCD.
- Here the write mode is obvious and it is used for writing or sending commands and data to the LCD.
- The read mode is used to read the CGRAM and the busy flag



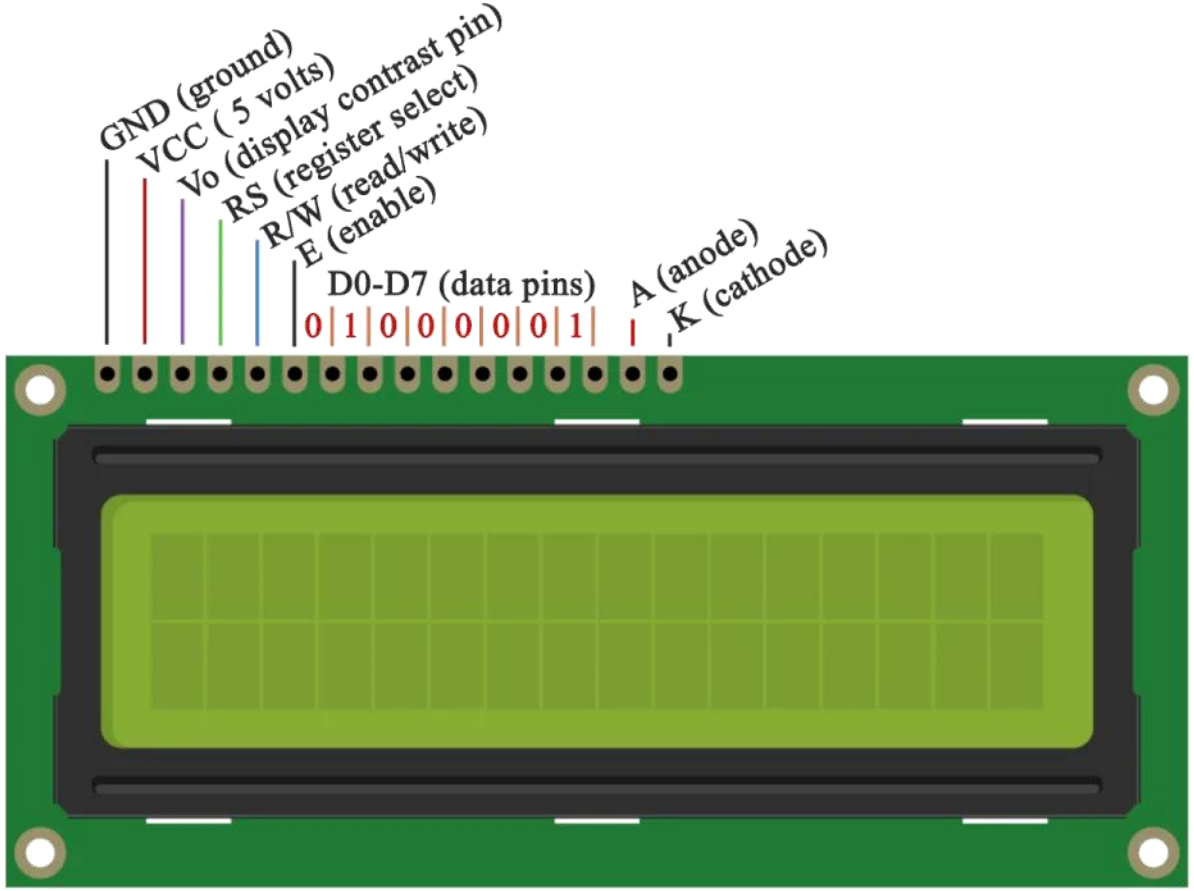
LCD

- ***E*** pin enables the writing to the registers, or the next 8 data pins from D0 to D7.
- So through this pins we are sending the 8 bits data when we are writing to the registers or for example if we want to see the letter uppercase A on the display we will send 0100 0001 to the registers according to the ASCII table.



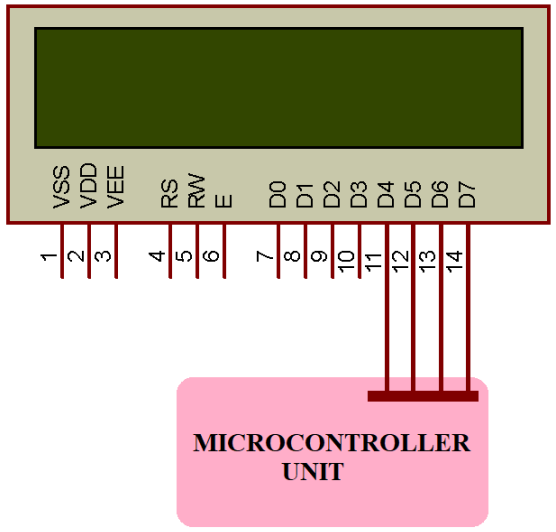
LCD

- **LED Backlight (*A* and *K*)** pins that are used to turn on and off the LED backlight.
- A resistor should be connected in series to protect the LED. The value of the resistor can be in a range of 100 to 470 Ohm (when supplied by a 5V source)

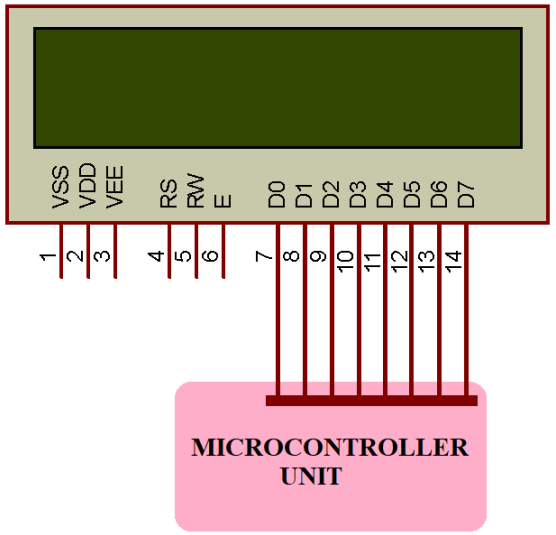


LCD

- The HD44780 can receive data in either two 4-bit operations or one 8-bit operation, thus allowing interfacing with 4- or 8-bit microcontroller.
- For 4-bit interface data, only four bus lines (DB4 to DB7) are used for transfer. Bus lines DB0 to DB3 are disabled. The data transfer between the HD44780U and the MPU is completed after the 4-bit data has been transferred twice.
- As for the order of data transfer, the four high order bits (for 8-bit operation, DB4 to DB7) are transferred before the four low order bits (for 8-bit operation, DB0 to DB3).



LCD : 4 BIT MODE



LCD : 8 BIT MODE



LCD

Writing Instruction

- A set of instructions of the HD44780 are listed in the table on the right.
- For example LCD is cleared by sending B'00000001' while RS and RW are zeros.
- Notice the bold "**1**" in every instruction; this represents the select bit of the HD44780 internal register.

BIT name	Settings	
I/D	0 = Decrement cursor position	1 = Increment cursor position
S	0 = No display shift	1 = Display shift
D	0 = Display off	1 = Display on
C	0 = Cursor off	1 = Cursor on
B	0 = Cursor blink off	1 = Cursor blink on
S/C	0 = Move cursor	1 = Shift display
R/L	0 = Shift left	1 = Shift right
DL	0 = 4-BIT interface	1 = 8-BIT interface
N	0 = 1/8 or 1/11 Duty (1 line)	1 = 1/16 Duty (2 lines)
F	0 = 5x7 dots	1 = 5x10 dots
BF	0 = Can accept instruction	1 = Internal operation in progress

Instruction	Code										Description
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	
Clear display	0	0	0	0	0	0	0	0	0	**1**	Clears display and returns cursor to the home position (address 0).
Cursor home	0	0	0	0	0	0	0	0	**1**	*	Returns cursor to home position (address 0). Also returns display being shifted to the original position. DDRAM contents remains unchanged.
Entry mode set	0	0	0	0	0	0	0	**1**	I/D	S	Sets cursor move direction (I/D), specifies to shift the display (S). These operations are performed during data read/write.
Display On/Off control	0	0	0	0	0	0	**1**	D	C	B	Sets On/Off of all display (D), cursor On/Off (C) and blink of cursor position character (B).
Cursor/display shift	0	0	0	0	0	**1**	S/C	R/L	*	*	Sets cursor-move or display-shift (S/C), shift direction (R/L). DDRAM contents remains unchanged.
Function set	0	0	0	0	**1**	DL	N	F	*	*	Sets interface data length (DL), number of display line (N) and character font(F).
Set CGRAM address	0	0	0	**1**	CGRAM address						Sets the CGRAM address. CGRAM data is sent and received after this setting.
Set DDRAM address	0	0	**1**	DDRAM address							Sets the DDRAM address. DDRAM data is sent and received after this setting.
Read busy-flag and address counter	0	1	BF	CGRAM / DDRAM address							Reads Busy-flag (BF) indicating internal operation is being performed and reads CGRAM or DDRAM address counter contents (depending on previous instruction).
Write to CGRAM or DDRAM	1	0	write data								Writes data to CGRAM or DDRAM.
Read from CGRAM or DDRAM	1	1	read data								Reads data from CGRAM or DDRAM.



LCD

Writing Instruction

- **Clear Display**
- Clear display writes space code 20H (character pattern for character code 20H must be a blank pattern) into all DDRAM addresses. It then sets DDRAM address 0 into the address counter, and returns the display to its original status if it was shifted. In other words, the display disappears and the cursor goes to the left display edge
- **Return Home**
- Return home sets DDRAM address 0 into the address counter, and returns the display to its original status if it was shifted. The DDRAM contents do not change. The cursor goes to the left display edge.
- **Entry Mode Set**
- I/D: Increments (I/D = 1) or decrements (I/D = 0) the DDRAM address by one when a character code is written into or read from DDRAM. The cursor moves to the right when incremented by one and to the left when decremented by one. The same applies to writing and reading of CGRAM.
- S: Shifts the entire display either to the right (I/D = 0) or to the left (I/D = 1) when S is 1. The display does not shift if S is 0. If S is 1, it will seem as if the cursor does not move but the display does. The display does not shift when reading from DDRAM. Also, writing into or reading out from CGRAM does not shift the display.

LCD

Writing Instruction

- **Display On/Off Control**
- D: The display is on when D is 1 and OFF when D is 0. When off, the display data remains in DDRAM, but can be displayed instantly by setting D to 1.
- C: The cursor is displayed when C is 1 and not displayed when C is 0. Even if the cursor disappears, the function of I/D or other specifications will not change during display data write. The cursor is displayed using 5 dots in the 8th line for 5x8-dot character font selection and in the 11th line for the 5x10-dot character font selection.
- B: The character indicated by the cursor blinks when B is 1. The blinking is displayed as switching between all blank dots and displayed characters. The cursor and blinking can be set to display simultaneously.

- **Cursor or Display Shift**
- Cursor or display shift shifts the cursor position or display to the right or left without writing or reading display data. This function is used to correct or search the display. In a 2-line display, the cursor moves to the second line when it passes the 40th digit of the first line. Note that the first and second line displays will shift at the same time.
- When the displayed data is shifted repeatedly each line moves only horizontally. The second line display does not shift into the first line position. The address counter (AC) contents will not change if the only action performed is a display shift.



LCD

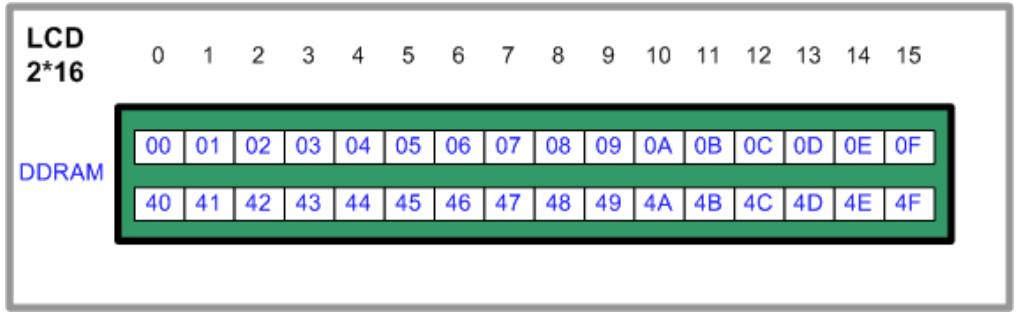
Writing Instruction

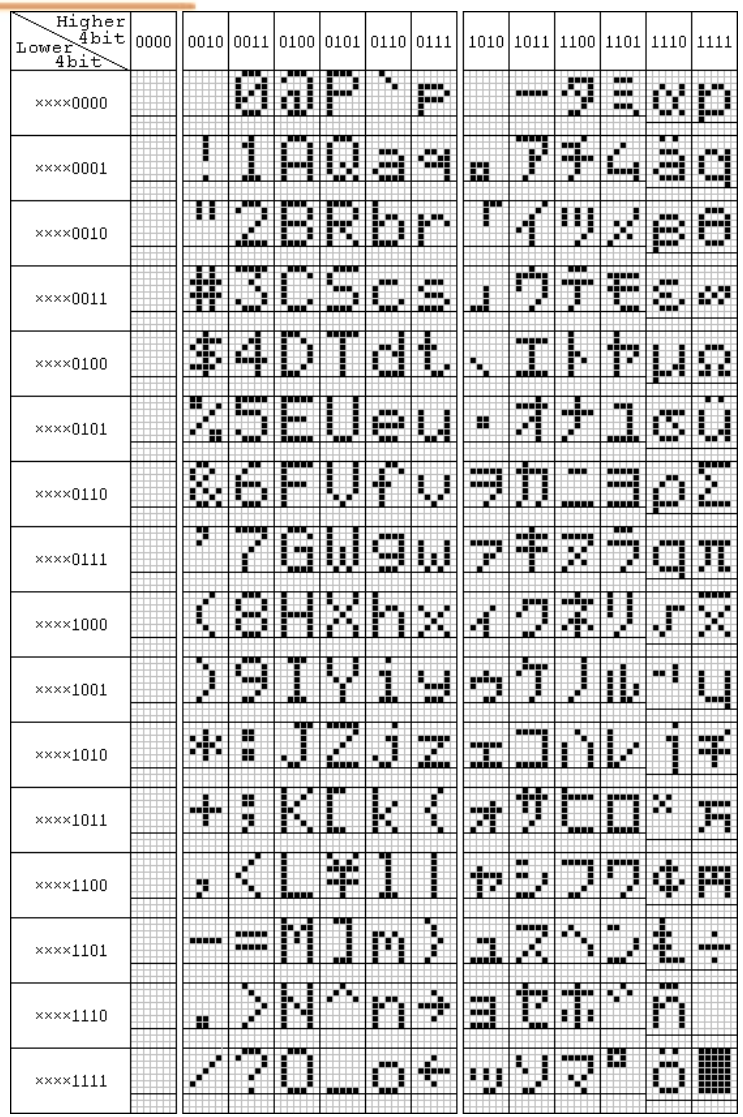
- **Function Set**
- DL: Sets the interface data length. Data is sent or received in 8-bit lengths (DB7 to DB0) when DL is 1, and in 4-bit lengths (DB7 to DB4) when DL is 0. When 4-bit length is selected, data must be sent or received twice.
- N: Sets the number of display lines.
- F: Sets the character font.

LCD

Writing Data

- The HD44780 has an internal Character Generator ROM CGROM that saves all ASCII characters.
- The CGROM generates 5x8 dot or 5x10 dot character patterns from 8-bit character codes as shown in the Table.
- It can generate 208 “5x8” dots character patterns and 32 “5x10” dots character patterns.
- To display the character “A” for example, you have to send B’01000001’ to the LCD port while RS is 1 and R/W is 0. Notice that transforming every character to binary code is very difficult especially if you want to write a phrase on the LCD.
- Display data RAM (DDRAM) stores display data represented in 8-bit character codes. Its extended capacity is 80x8 bits, or 80 characters. The area in display data RAM (DDRAM) that is not used for display can be used as general data RAM.



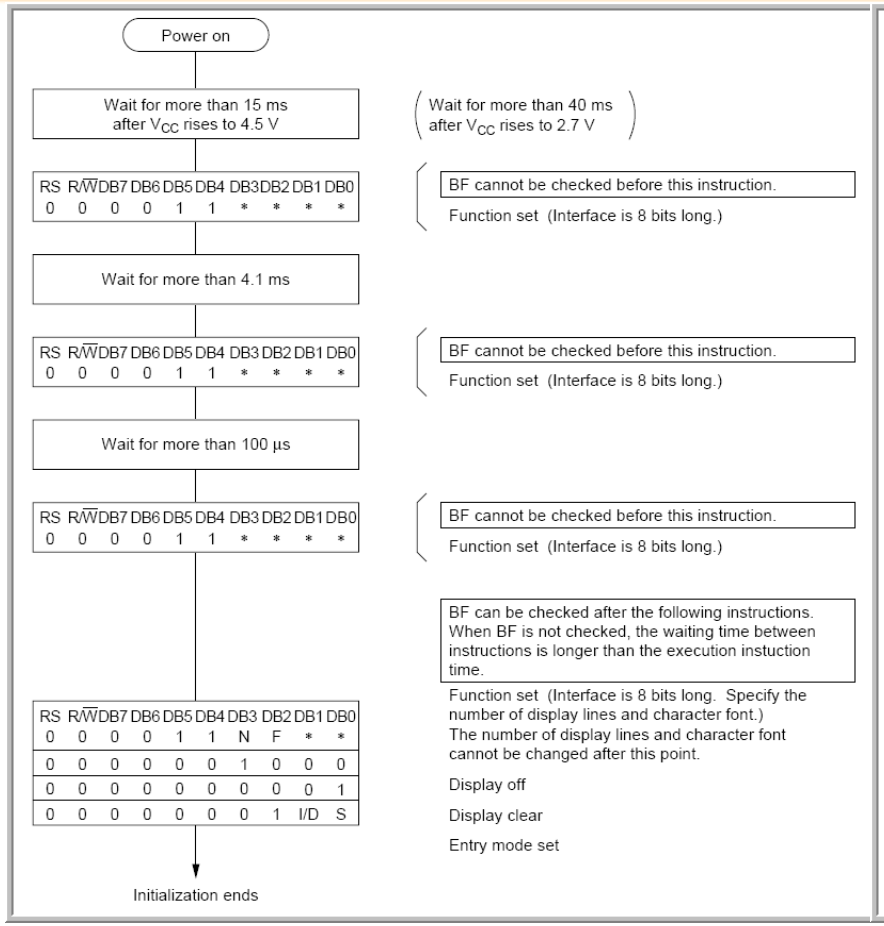


Lecture

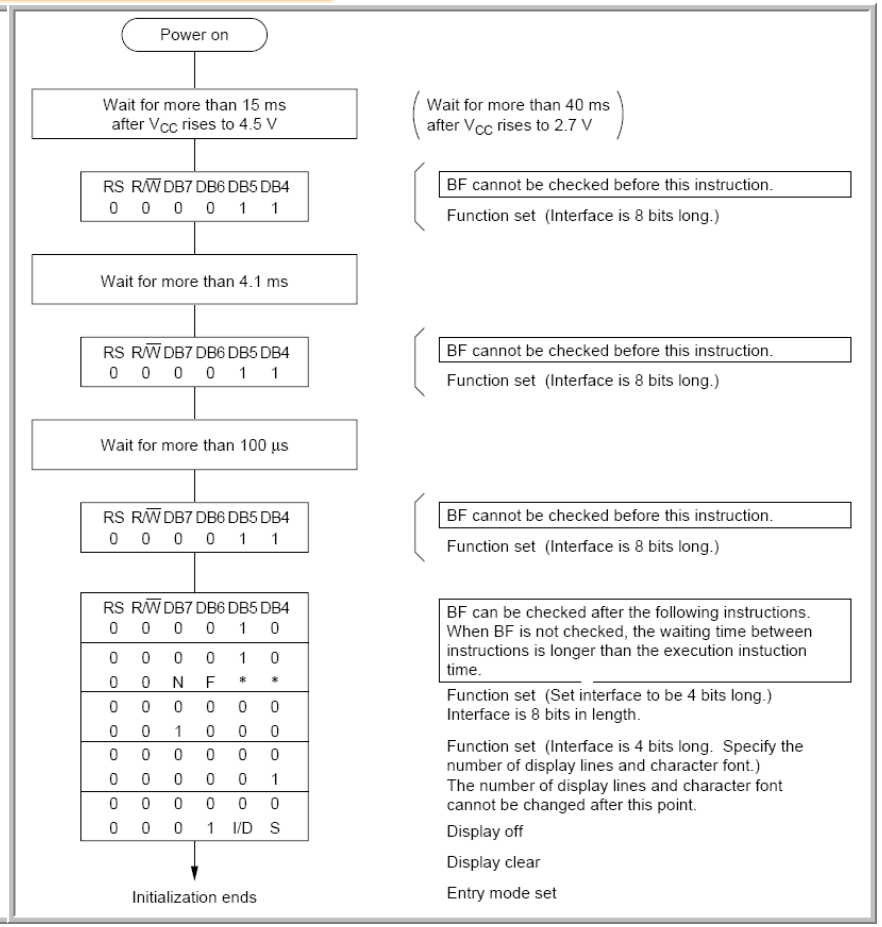
LCD

LCD Initialization

- Before being able to receive instruction and data, the LCD needs to be initialized to the appropriate mode of function. We have mentioned that the HD44780 controller is available for almost all sizes of LCD; therefore, you have to tell the HD44780 that the character type is “5x7” or “5x10”, the number of display lines is 1, 2 or 4, and the control mode is 4-bit or 8-bit.



8-bit mode initialization steps



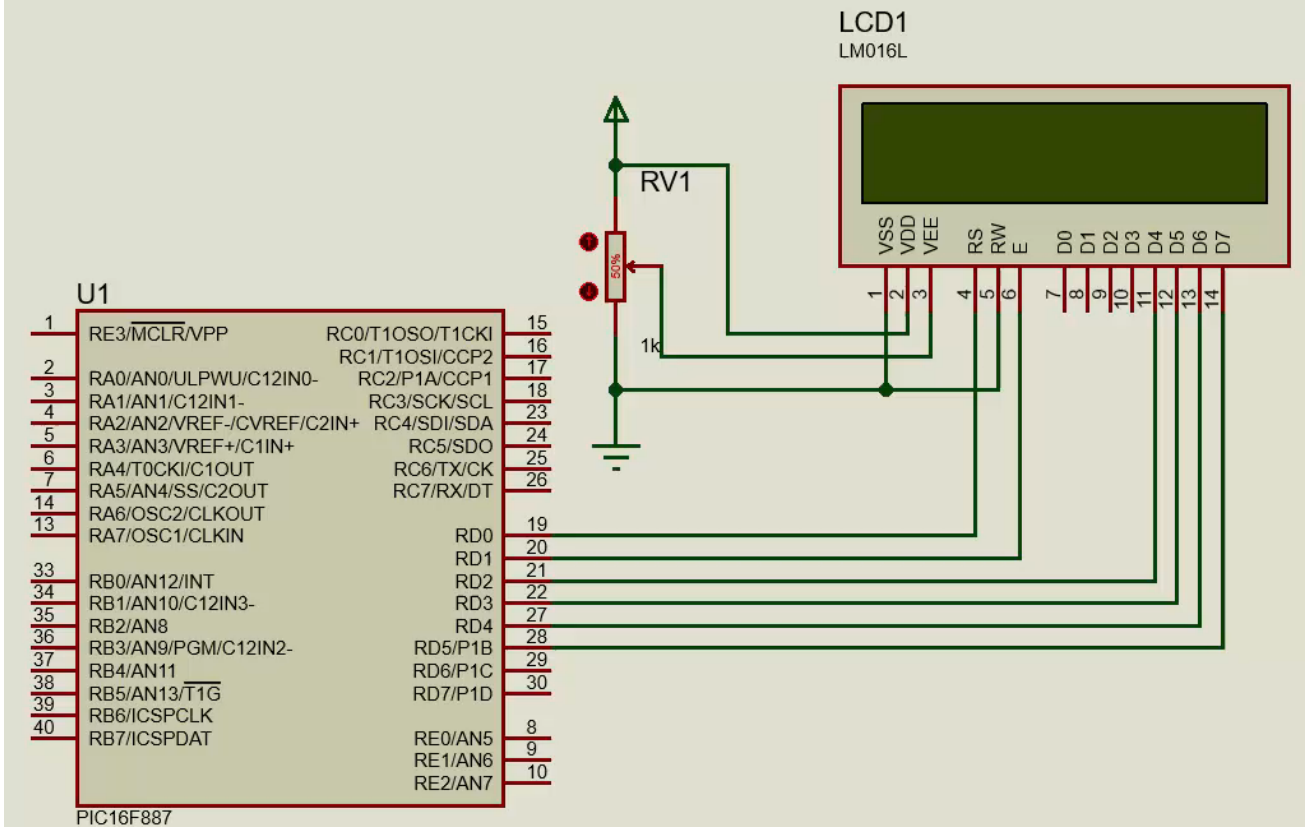
4-bit mode initialization steps

Activity 16 : Writing msg on LCD

In this activity, we are interested in writing a message on the LCD. In order to reduce the connection complexity, the LCD is interfaced in 4 bit mode. i.e., 6 lines are only required from the PIC (RS, E, D4…D7). The R/W line will be grounded as the LCD required to write only.

A potentiometer of 1k to 10k is required to be connected on VEE line to define the contract of the LCD.

In Proteus, the backlight pins (A, K) are not available. In case of hardware implementation, these pins should be connected to VCC though a 200-300 ohm resistor (Same as LED connection)



Activity 15 : Writing msg on LCD

```
28    A1          EQU 0X70
29    A2          EQU 0X71
30    A3          EQU 0X72
31    LCD_TEMP    EQU 0X73
32
33     #define RS RD0
34     #define E  RD1
35     #define D4 RD2
36     #define D5 RD3
37     #define D6 RD4
38     #define D7 RD5
```

- First, the variables are declared and the pins are defines.
- The main code includes the steps to be repeated. Display messages, wait, display another msg and wait…
- The LCD_BEGIN is the initialization function required to setup the LCD in 4 bits mode. These steps are to be respected as recommended by Hitachi

```
45    MAIN:
46        CALL    SETUP
47        CALL    LCD_BEGIN    ; initialize the LCD
48    REPEAT:
49        CALL    LCD_CLEAR    ; clear lcd
50        CALL    LCD_LIN1     ; focus on line 1
51        CALL    MSG1         ; print mgs 1
52
53        CALL    LCD_LIN2     ; focus on line 2
54        CALL    MSG2         ; print msg 2
55        CALL    DELAY_2S     ; and wait 2s
56
57        CALL    LCD_CLEAR    ; clear again
58        CALL    LCD_LIN1     ; focus on the line
59        CALL    MSG3         ; and print another msg
60        CALL    DELAY_2S     ; for 2s
61        GOTO    REPEAT
```

```
74    LCD_BEGIN:
75        BCF     E
76        BCF     RS
77        CALL    DELAY_20MS       ; WAIT 20 MS
78        MOVLW   00110000B        ; SEND 0X3
79        CALL    LCD_4H
80        CALL    FETCH_CMD
81        CALL    DELAY_4500US
82        MOVLW   00110000B        ; SEND 0X3
83        CALL    LCD_4H
84        CALL    FETCH_CMD
85        CALL    DELAY_4500US
86        MOVLW   00110000B        ; SEND 0X3
87        CALL    LCD_4H
88        CALL    FETCH_CMD
89        CALL    DELAY_150US
90        MOVLW   00100000B        ; SEND 0X2
91        CALL    LCD_4H
92        CALL    FETCH_CMD
93        MOVLW   00101000B        ; SEND 0X28
94        CALL    LCD_CMD          ; 2 LINES 4-BIT INTERFACE
95        MOVLW   00001000B        ; DISPLAY OFF
96        CALL    LCD_CMD
97        CALL    LCD_CLEAR        ; CLEAR ALL
98        MOVLW   00001100B        ; DISPLAY ON, NO CURSOR
99        CALL    LCD_CMD
100       MOVLW   00000110B        ; AUTO INCREMENT
101       CALL    LCD_CMD
102       RETURN
```

Activity 15 : Writing msg on LCD

- LCD_CMD is the function that sends commands to the LCD
- LCD_WRITE is used to write data on LCD
- These functions fetch the higher and lower nibbles to the LCD in order

```
136     LCD_CMD:
137         CALL    LCD_4H
138         CALL    FETCH_CMD
139         CALL    LCD_4L
140         CALL    FETCH_CMD
141         CALL    DELAY_2000US
142         RETURN
143
144     LCD_WRITE:
145         CALL    LCD_4H
146         CALL    FETCH_DATA
147         CALL    LCD_4L
148         CALL    FETCH_DATA
149         CALL    DELAY_2000US
150         RETURN
```

```
152     FETCH_CMD:
153         BCF     RS  ; RS = 0 => INSTRUCTION
154         BSF     E
155         NOP
156         NOP
157         BCF     E
158         RETURN
159
160     FETCH_DATA:
161         BSF     RS  ; RS = 1 => DATA
162         BSF     E
163         NOP
164         NOP
165         BCF     E
166         RETURN
```

- LCD_4H and _4L are the functions that reads the data to be sent to the LCD and places them on the selected pins D4…D7. these are created so that user can modify the lcd-pic pin connection easily.

```
104     LCD_4H:
105         MOVWF   LCD_TEMP    ; save to temporary register
106         BCF     D4          ; clear lines first
107         BCF     D5
108         BCF     D6
109         BCF     D7
110         BTFSC   LCD_TEMP, 7 ; check the bit and place
111         BSF     D7          ; its value on the pin
112         BTFSC   LCD_TEMP, 6
113         BSF     D6
114         BTFSC   LCD_TEMP, 5
115         BSF     D5
116         BTFSC   LCD_TEMP, 4
117         BSF     D4
118         RETURN
119
120     LCD_4L:
121         MOVWF   LCD_TEMP
122         BCF     D4
123         BCF     D5
124         BCF     D6
125         BCF     D7
126         BTFSC   LCD_TEMP, 3
127         BSF     D7
128         BTFSC   LCD_TEMP, 2
129         BSF     D6
130         BTFSC   LCD_TEMP, 1
131         BSF     D5
132         BTFSC   LCD_TEMP, 0
133         BSF     D4
134         RETURN
```

Activity 15 : Writing msg on LCD

- 4 important functions are designed to easily clear the lcd, focus on line1, or line 2 and to return home.

```
168     LCD_CLEAR:
169         MOVLW       00000001B
170         CALL        LCD_CMD
171         RETURN
172
173     LCD_HOME:
174         MOVLW       00000010B
175         CALL        LCD_CMD
176         RETURN
177
178     LCD_LIN1:
179             MOVLW   10000000B       ; ADDRESS 0X0 OF DDRAM
180             CALL    LCD_CMD
181             RETURN
182
183     LCD_LIN2:
184             MOVLW   11000000B       ; ADDRESS 0X40 OF DDRAM
185             CALL    LCD_CMD
186             RETURN
```

```
292     MSG3:
293         MOVLW   'E'
294         CALL    LCD_WRITE
295         MOVLW   'L'
296         CALL    LCD_WRITE
297         MOVLW   'E'
298         CALL    LCD_WRITE
299         MOVLW   '3'
300         CALL    LCD_WRITE
301         MOVLW   '6'
302         CALL    LCD_WRITE
303         MOVLW   '1'
304         CALL    LCD_WRITE
305         MOVLW   '4'
306         CALL    LCD_WRITE
307         RETURN
```

- Finally, writing msg on LCD is simply done by calling the LCD_WRITE while placing ASCII value on W



Week 9

Lab 7 / LO4

LCD



Activity 15 : Writing msg on LCD

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. PIC programmer
  4. Power supply 5V
  5. Breadboard wires (male-male)

- Required components:
  1. PIC 16F887
  2. Resistor from 100 to 330 x 1 (for the lcd back light)
  3. LCD 16 characters 2 lines x1
  4. Potentiometer 10k



Week 10

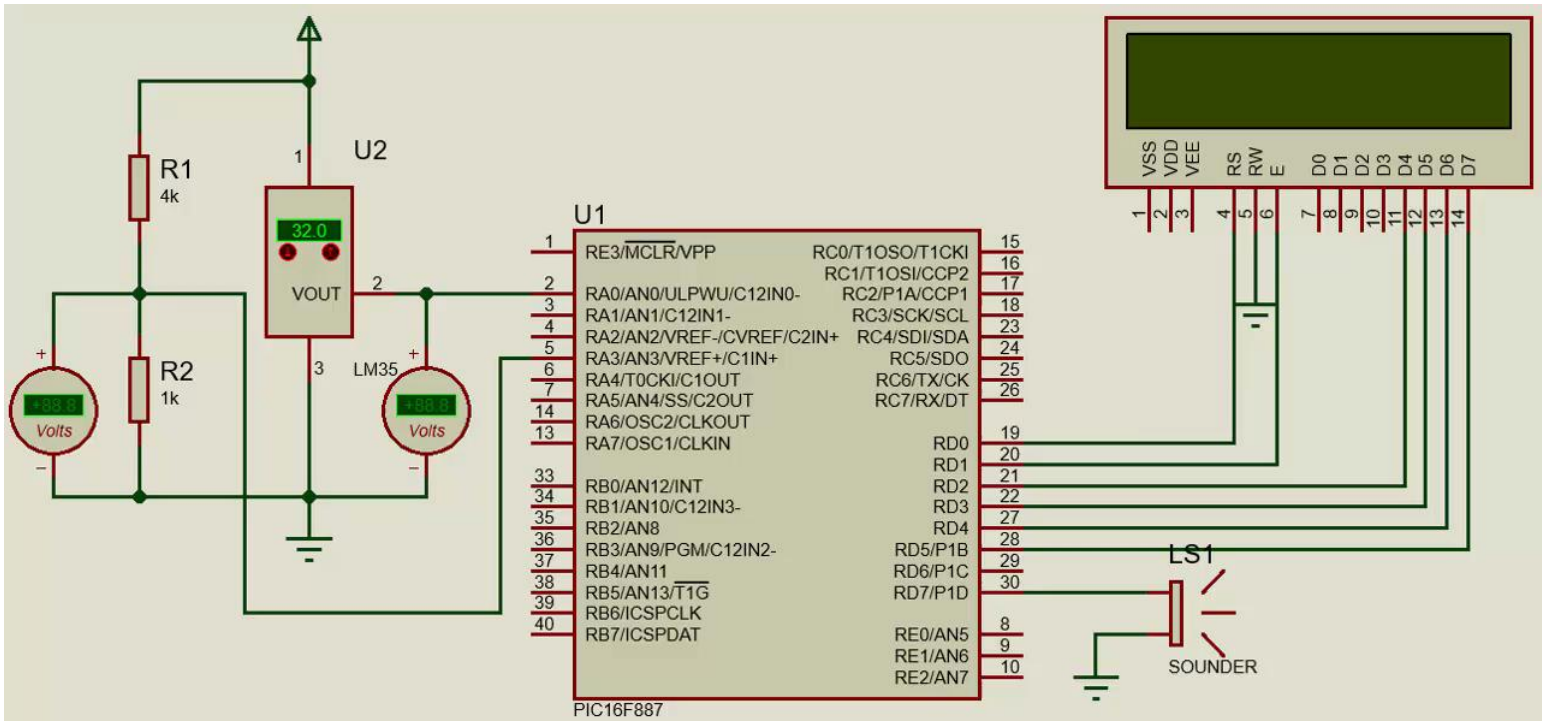
Lecture 17 / LO4

A/D in C with Temperature sensor and LCD

Activity 16C : A2D with LCD and Sensor

- In this activity we will connect a temperature sensor to the PIC, read its analog value, convert it into a digital form and display its value on the LCD. Also a piezo buzzer will be added to generate alarm if temperature is above certain limit.

- The temperature sensor to be used is the LM35, linear sensor with 10mV/℃



Activity 16C : A2D with LCD and Sensor

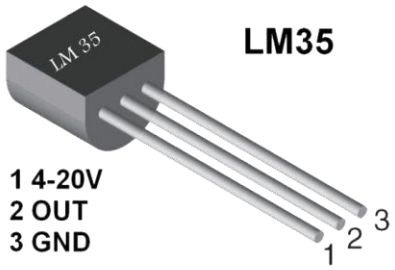
- Let's start first by meeting our devices.

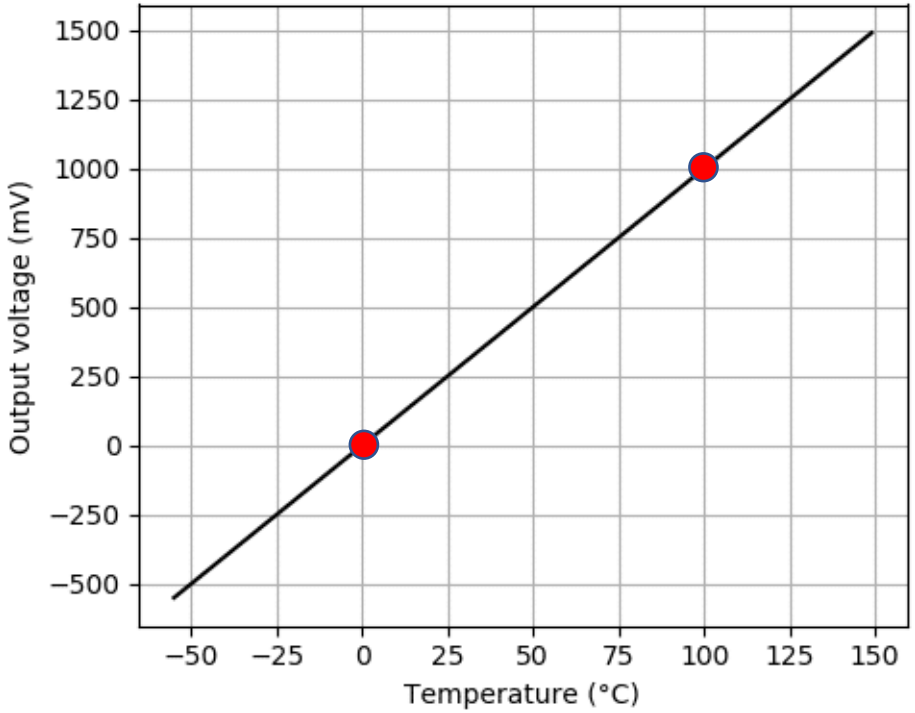
LM35

The LM35 is a cheap temperature sensor, which can be used to measure ambient temperature.
The LM35 looks like a transistor but it is in fact an integrated circuit, with an output voltage linearly proportional to the Centigrade temperature.

Features

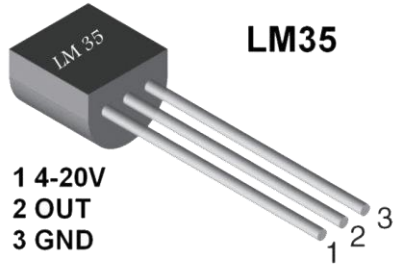
- Calibrated directly in ° Celsius (Centigrade)
- Linear + 10.0 mV/°C scale factor
- 0.5°C accuracy (at +25°C)
- Rated for full −55° to +150°C range
- Operates from 4 to 20 volts
- Output impedance 0.5 ohm
- Can source current up to 10 mA





Activity 16C : A2D with LCD and Sensor

- Let's start first by meeting our devices.



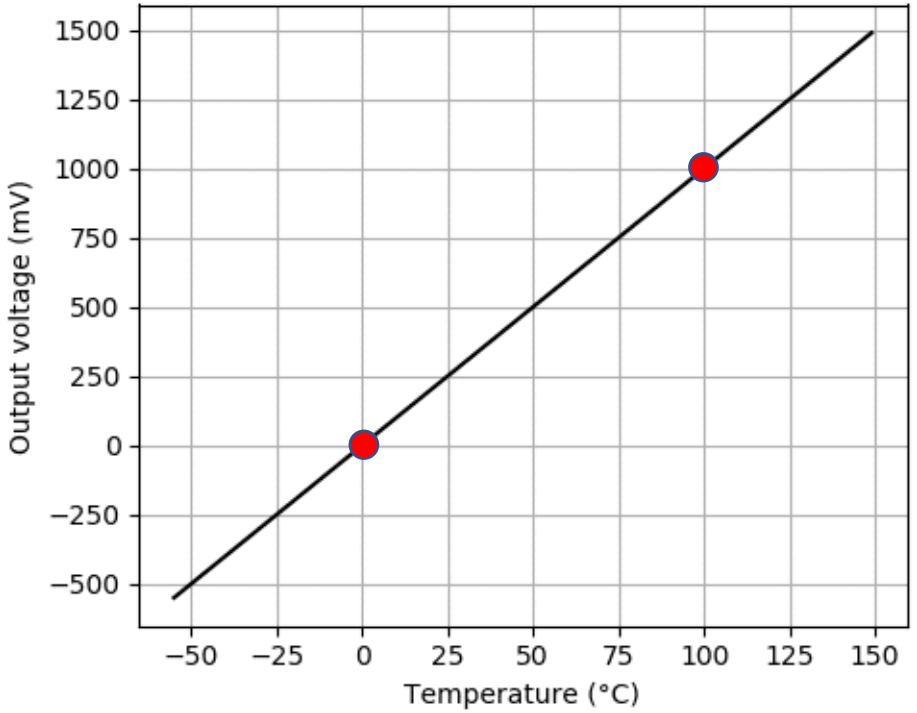
LM35

Converting the LM35 output voltage into temperature
The output scale factor of the LM35 is 10 mV/°C. for example, it provides an output voltage of 250 mV at 25°C… or 1V at 100°C

To convert the output voltage of the sensor into the temperature in degree Celsius, the following formula is used

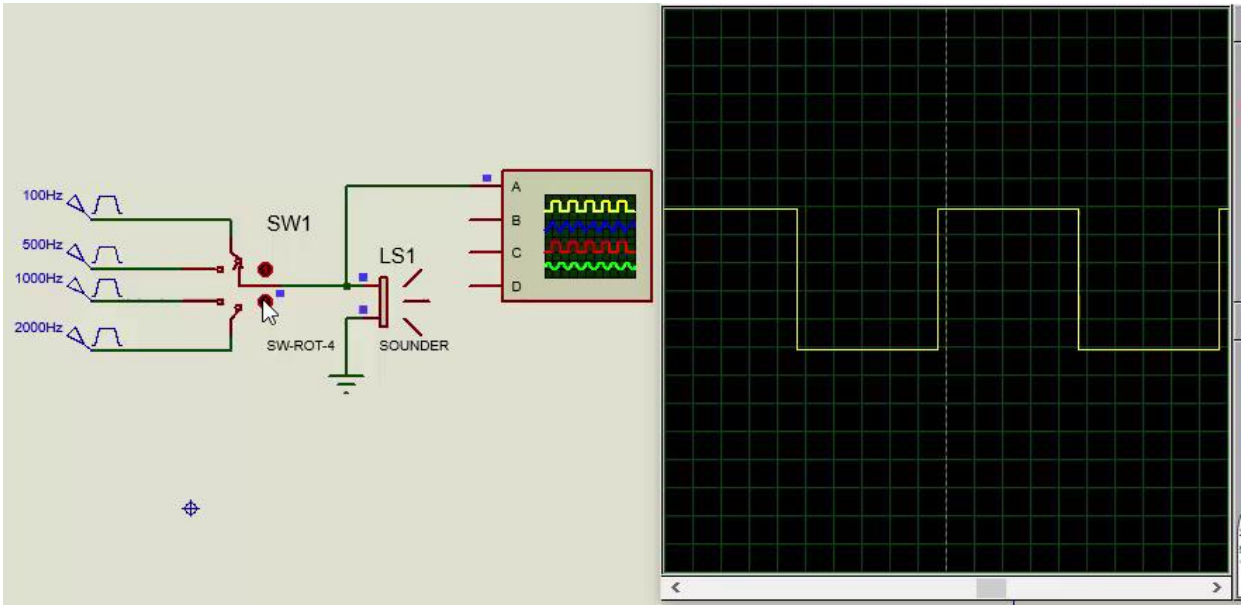
Temperature (°C) = V_{OUT} / 10

with V_{OUT} in millivolt (mV).
So if the output of the sensor is 750 mV, the temperature is 75°C



Activity 16C : A2D with LCD and Sensor

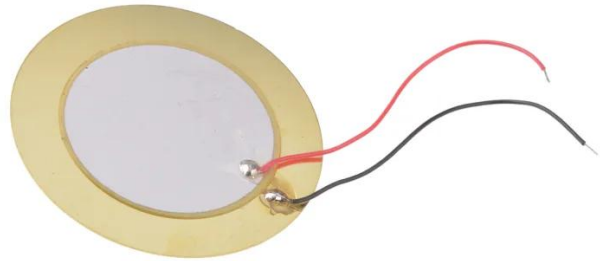
- Let's start first by meeting our devices.



Piezo buzzer

- A piezoelectric buzzer is a loudspeaker that uses the piezoelectric effect for generating sound.
- The initial mechanical motion is created by applying a voltage to a piezoelectric material, and this motion is typically converted into audible sound using diaphragms and resonators.
- Compared to other speaker designs piezoelectric speakers are relatively easy to drive; for example they can be connected directly to TTL outputs, although more complex drivers can give greater sound intensity.
- Typically they operate well in the range of 1 - 5kHz and up to 100kHz in ultrasound applications





Activity 16C : A2D with LCD and Sensor

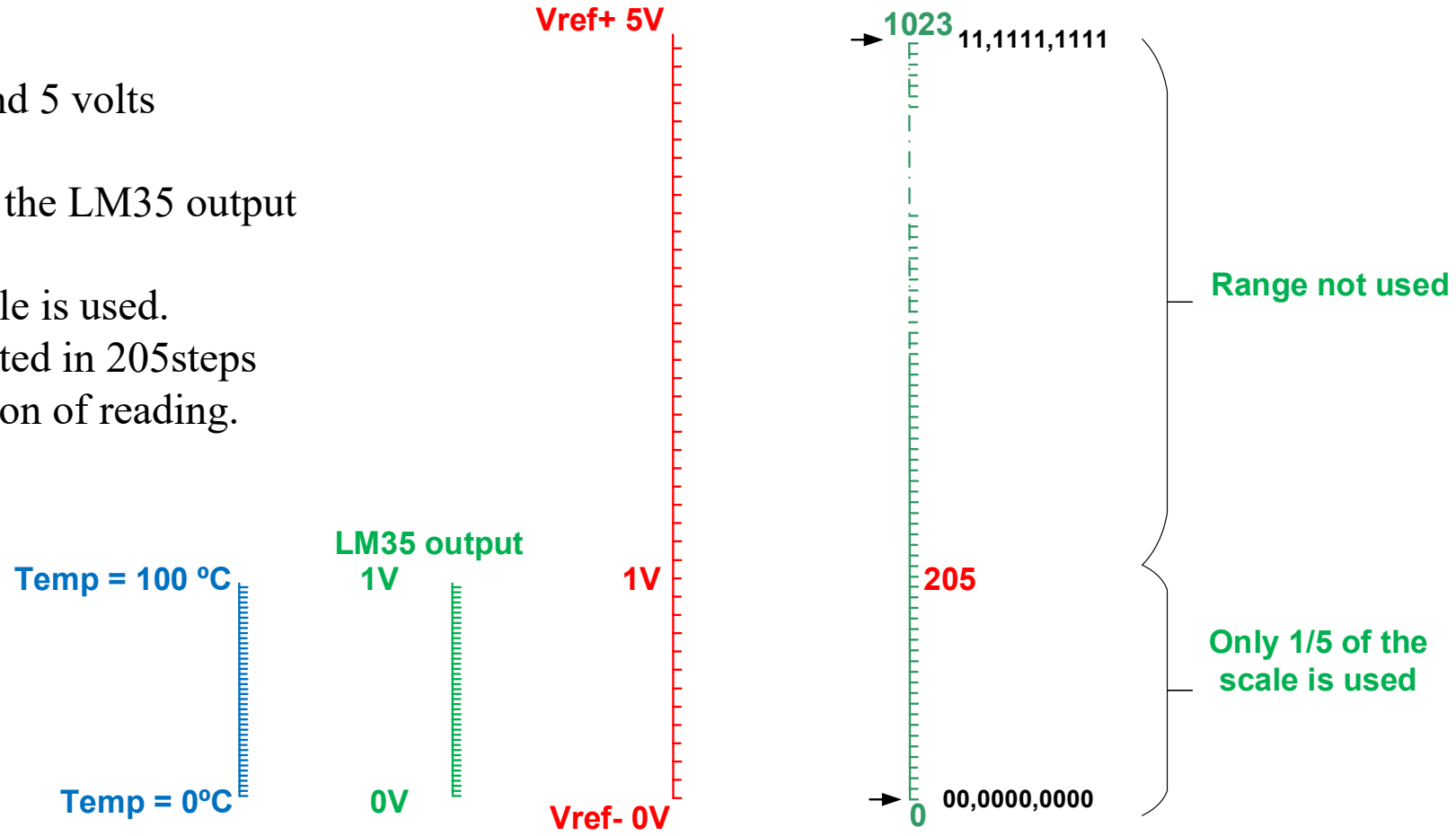
- Circuit / Coding design

LM35

Assuming that the A2D references are 0 and 5 volts

For a temperature range from 0 to 100 °C, the LM35 output voltage ranges from 0 to 1V.
It can be seen that only 1/5 of the A2D scale is used.
i.e., the full range of temperature is translated in 205steps only. Which means we are loosing resolution of reading.

The resolution is 100/205 ≈ 0.5
Means that the PIC can recognize 0.5 degree step changes only.



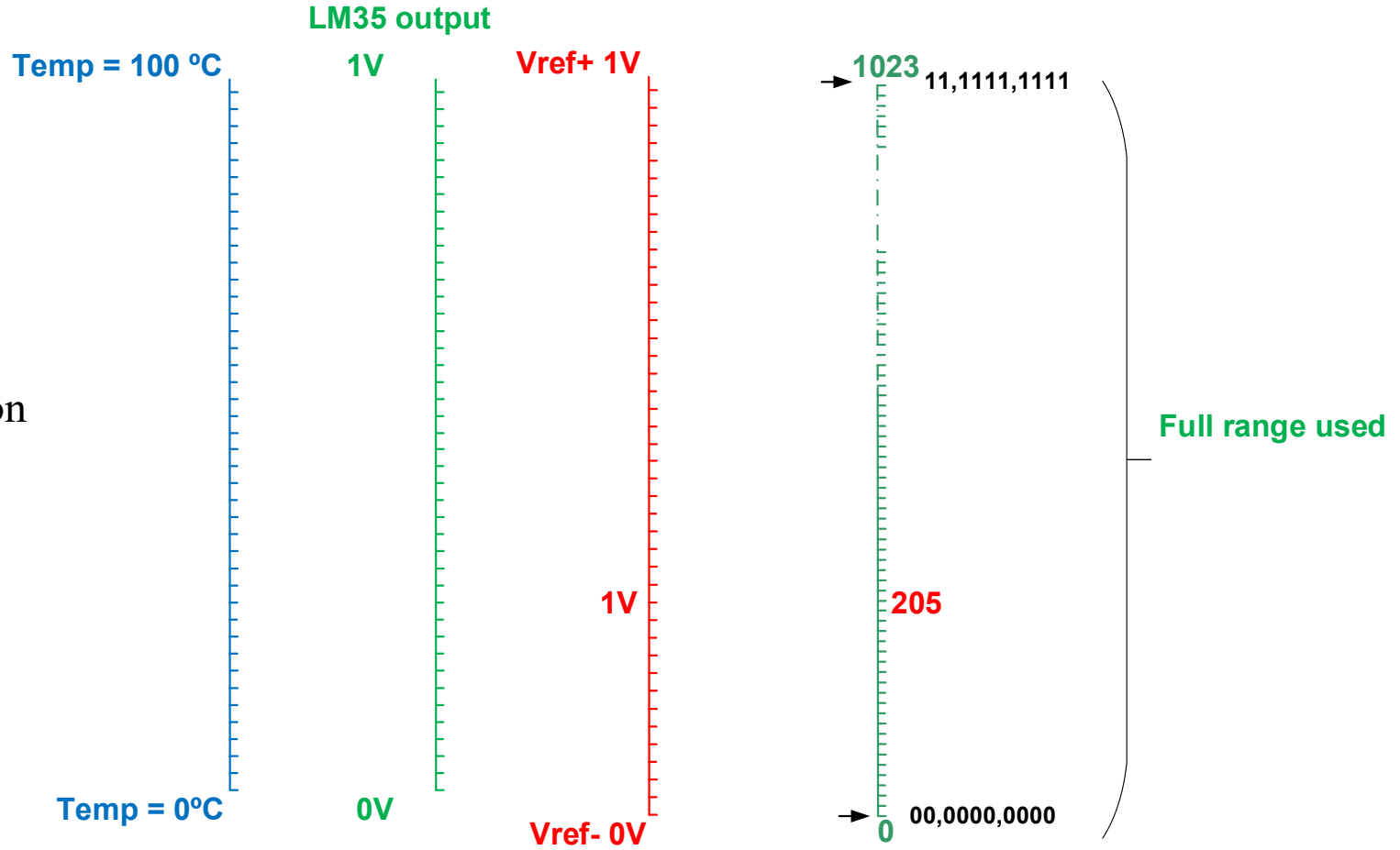
Activity 16C : A2D with LCD and Sensor

- Circuit / Coding design

LM35

Changing the Vref+ to 1V, assures that the full range of temperature is having 1024 different outputs.

The resolution now is 100/1024 ≈ 0.1
Means that the PIC is able to recognize a variation of 0.1 degree which is 5x more better that the previous case

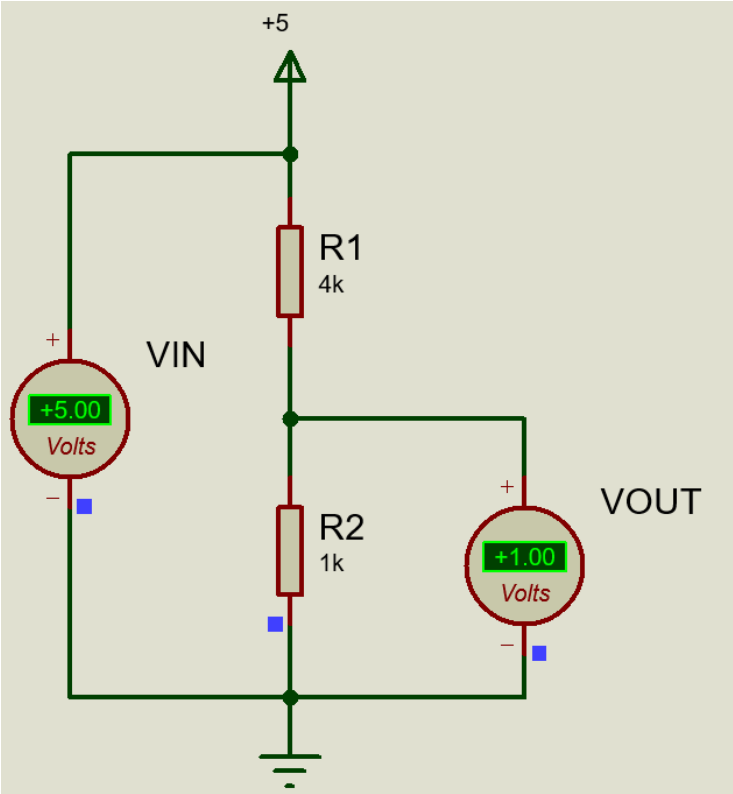


Activity 16C : A2D with LCD and Sensor

- Circuit / Coding design

Designing the 1V ref voltage

The PIC circuit is supplied by a 5v source.
Usually, when designing a circuit, we don't use different power source of different voltages.
We use the main voltage and we design converters to satisfy the circuit needs.



A voltage divider made of two resistors is the simplest solution to provide 1V output.

- Vin = 5
- Vout = 1V (the desired voltage)
- 2 resistors are to be calculated to get Vout =1 (we assume one and calculate the second)

The voltage divider equation is

$$V_{out} = \frac{R_2}{R_1 + R_2} V_{in} \rightarrow R_1 = R_2 \left(\frac{V_{in}}{V_{out}} - 1 \right) = 4R_2$$

For R2 = 1k, R1 = 4k

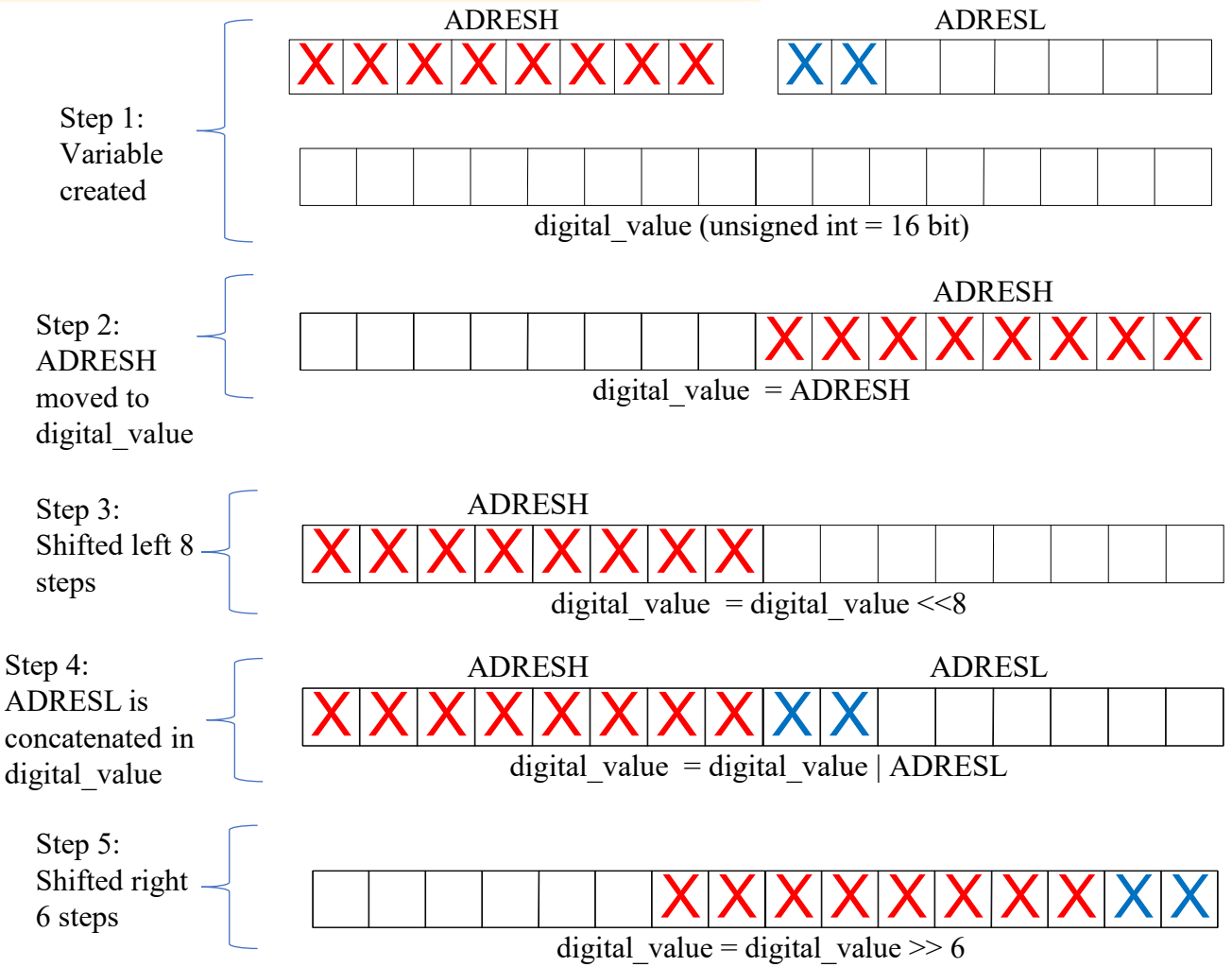
Activity 16C : A2D with LCD and Sensor

- Circuit / Coding design

A2D conversion in C in 10bits

The first part in coding is to read the 10bit AD result.
To do that, we configure ADCON0 and start the conversion using the GO bit. The result will be available in ADRESH and ADRESL.
To merge them in one variable, an unsigned integer "digital_value" is created

```
54  unsigned int READ_AN0() {
55      ADCON0 = 0b01000001;
56      _delay(20);
57      GO = 1;
58      while (GO);
59      unsigned int digital_value = (unsigned int) (((ADRESH) << 8) | ADRESL);
60      digital_value = digital_value >> 6;
61      return digital_value;
62  }
```



Activity 16C : A2D with LCD and Sensor

- Circuit / Coding design

Main code

- The main code includes the setup of I/O pins, and the setup of the LCD.
- A welcome message is first displayed for 2s.
- In the while loop, the temperature is read (AN0_value has the raw value of reading). Then this value is converted into float to be suitable for the mathematical operations.
- Integer to Ascii (itoa) is used to convert the temperature value into a set of characters saved in "phrase" ready to be sent to the LCD.
- An "if condition" is used to compare the temperature to 40

```
64  void main(void) {
65      setup();
66      lcd_begin();
67      lcd_clear();
68      lcd_print(line1, (char*) "I love Micro  :)");
69      lcd_print(line2, (char*) "----------------");
70      __delay_ms(2000);
71      while (1) {
72          AN0_value = READ_AN0();
73          float temperature = (float) AN0_value * 100 / 1023;
74
75          itoa(phrase, (int) temperature, 10);
76          lcd_clear();
77          lcd_print(line1, (char*) "Temp is:");
78          lcd_print(line1 + 9, phrase);
79          lcd_print(line1 + 11, (char*) "c");
80
81          if (temperature > 40) {
82              lcd_print(line2, (char*) "Hot");
83              buzz();
84          } else lcd_print(line2, (char*) "Cold");
85          __delay_ms(100);
86      }
87  }
```

Activity 16C : A2D with LCD and Sensor

- Circuit / Coding design

(itoa) is a function available in stdlib; thus needs to be included.

The buzz function is designed to generate pulses to the piezo buzzer. The signal has a period of 200µs (5000Hz) generated for ≈ 2000 x 100µs = 0.2s

```
28  #include "stdlib.h"
29
30  unsigned int AN0_value;
31  char phrase[7];
32
33  #define buzzer RD7
34
35  void setup() {
36      PORTD = 0; // clear port
37      PORTC = 0;
38      PORTA = 0;
39      OSCCON = 0b01100000; // select 4MHz oscillator
40      TRISC = 0b00000000; // and as out
41      TRISD = 0b00000000;
42      ADCON1 = 0b00010000;
43      ANS0 = 1;
44      TRISA0 = 1;
45  }
46
47  void buzz() {
48      for (int i = 0; i < 2000; i++) {
49          buzzer = !buzzer;
50          __delay_us(100);
51      }
52  }
```

Activity 16C : A2D with LCD and Sensor

- Circuit / Coding design

The LCD functions are already placed in a header file and included to the main file

```
24    #define _XTAL_FREQ 4000000
25    #include <xc.h>
26    #include "LiquidCrystal.h"
27    #include "stdlib.h"
```

First the pins are declared and the functions prototype are defined

The functions are similar to the one discussed in assembly

```
1    #define RS RD0
2    #define EN RD1
3    #define D4 RD2
4    #define D5 RD3
5    #define D6 RD4
6    #define D7 RD5
7    const char line1 = 0b10000000;
8    const char line2 = 0b11000000;
9    const char line3 = 0b10010100;
10   const char line4 = 0b11010100;
11
12   unsigned char LCD_PORT_TEMP;
13   void send_to_port(void);
14   void lcd_char(char);
15   void lcd_cmd(char);
16   void lcd_print(char,char *);
17   void lcd_nib(char);
18   void lcd_clear(void);
19   void lcd_begin(void);
```



Test your knowledge

1. What is the role of itoa in C code
2. Design a voltage divider circuit in order to have an output voltage of 2.3V from a source of 1000V
3. Check on the web the specs of the TMP36 and compare it to the LM35
4. What is the difference between an NTC sensor and the LM35. Why it is better to work with LM35 in A2D conversion !
5. What is the difference between a piezo buzzer and a speaker



Week 10

Lecture 18 / LO4

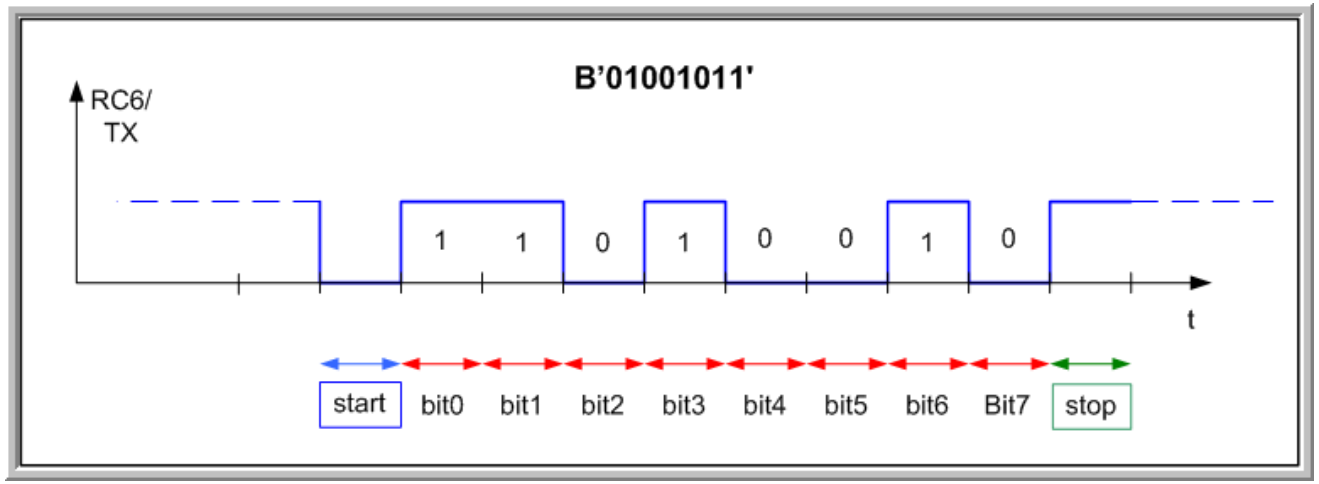
USART

USART in TX mode

USART

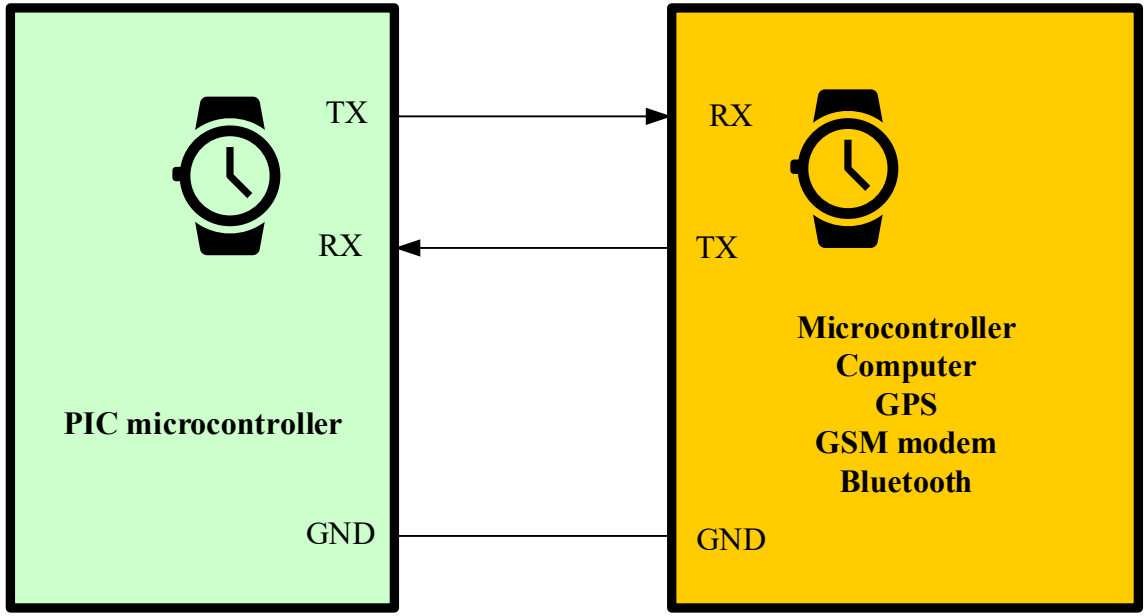
- The Universal Synchronous Asynchronous Receiver/Transmitter (USART) is a built-in hardware serial data transfer module.
- The synchronous mode is done via two communication lines. The first is for data transfer and the second is the clock that determines the speed of communication.
- In the asynchronous mode, data are transferred via one line only; the speed (bit rate) is already defined from the transmitter and the receiver sides.
- The asynchronous communication is mostly used to transfer data between PIC and PC.

- When doing data communications, the condition of "0" and "1" out from the side of the sending must be able to be recognized in the receiving side.
- Asynchronous communication, starts by a START bit, then 8or 9 bits of data, then finished by a STOP bit.



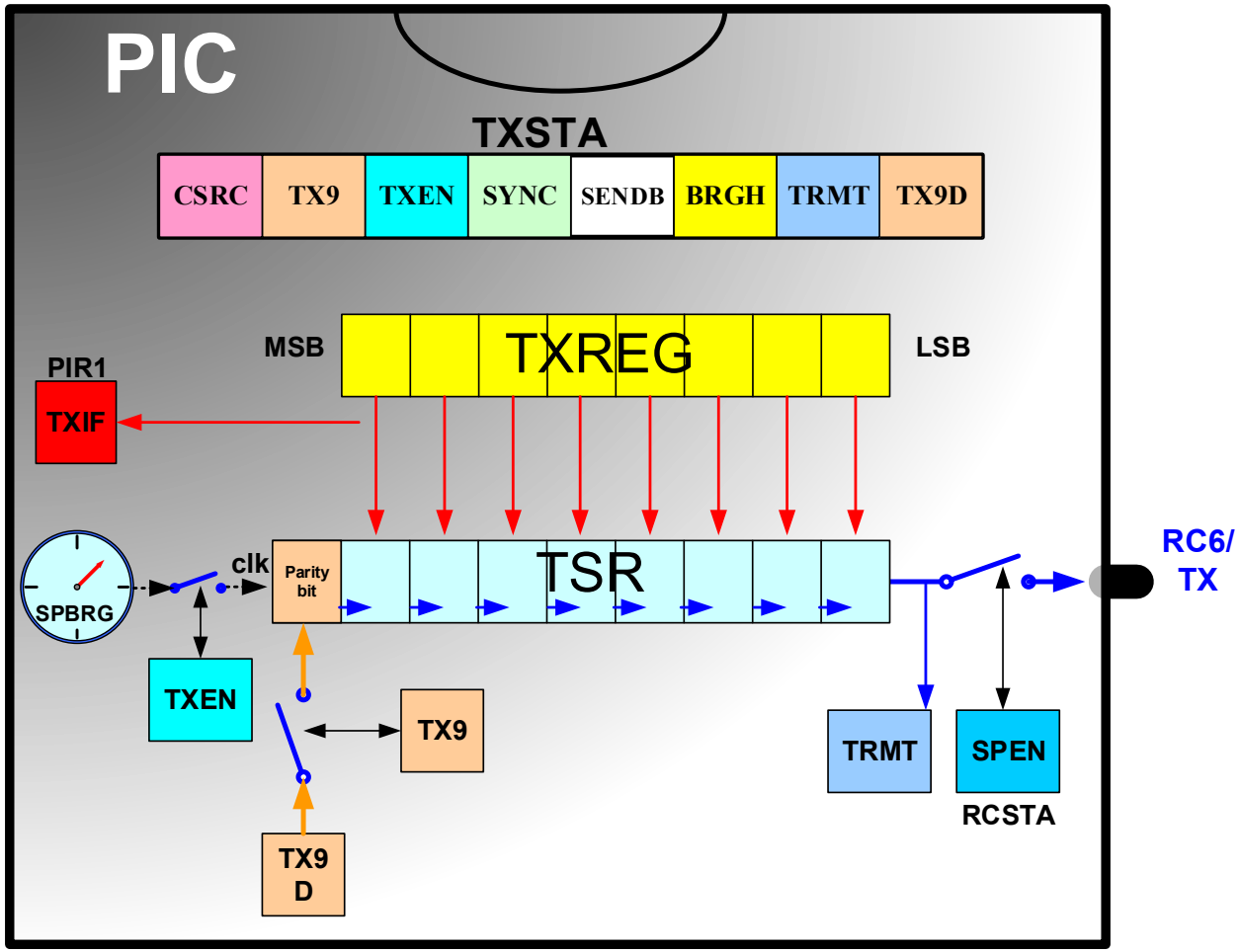
USART

- In the asynchronous mode communication of USART, the RX port is used for receiving and the TX port is used for the transmission, so, it is possible to send and receive at the same time.
- This type of communication is called Full duplex.
- The transmission is achieved based on a stopwatch running at the same devices. This will assure that the devices are communicating at the same speed (same baud rate)
- **The Baud rate refers to the total number of signal units transmitted in one second.**



USART (Asynchronous transmitter)

- The TX pin is multiplexed with RC6 (output pin).
- The transmitter is managed by TXSTA register.
- When TXREG is loaded, the data is shifted to the transmit shift register TSR and shifter serially one by one (the least significant bit first) with a start bit first and a stop bit at the end.
- When data is fully sent, the TRMT flag bit is raised.
- The speed of pushing the data is managed by the SPBRG clock generator.
- User can also add the 9th bit (called parity bit)
- In order to make the serial port active, the SPEN bit should be enabled.



USART (Asynchronous transmitter)

REGISTER 12-1: TXSTA: TRANSMIT STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN(1)	SYNC	SENDB	BRGH	TRMT	TX9D
bit 7							bit 0

bit 7 **CSRC:** Clock Source Select bit
Asynchronous mode:
Don't care
Synchronous mode:
1 = Master mode (clock generated internally from BRG)
0 = Slave mode (clock from external source)

bit 6 **TX9:** 9-bit Transmit Enable bit
1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5 **TXEN:** Transmit Enable bit(1)
1 = Transmit enabled
0 = Transmit disabled

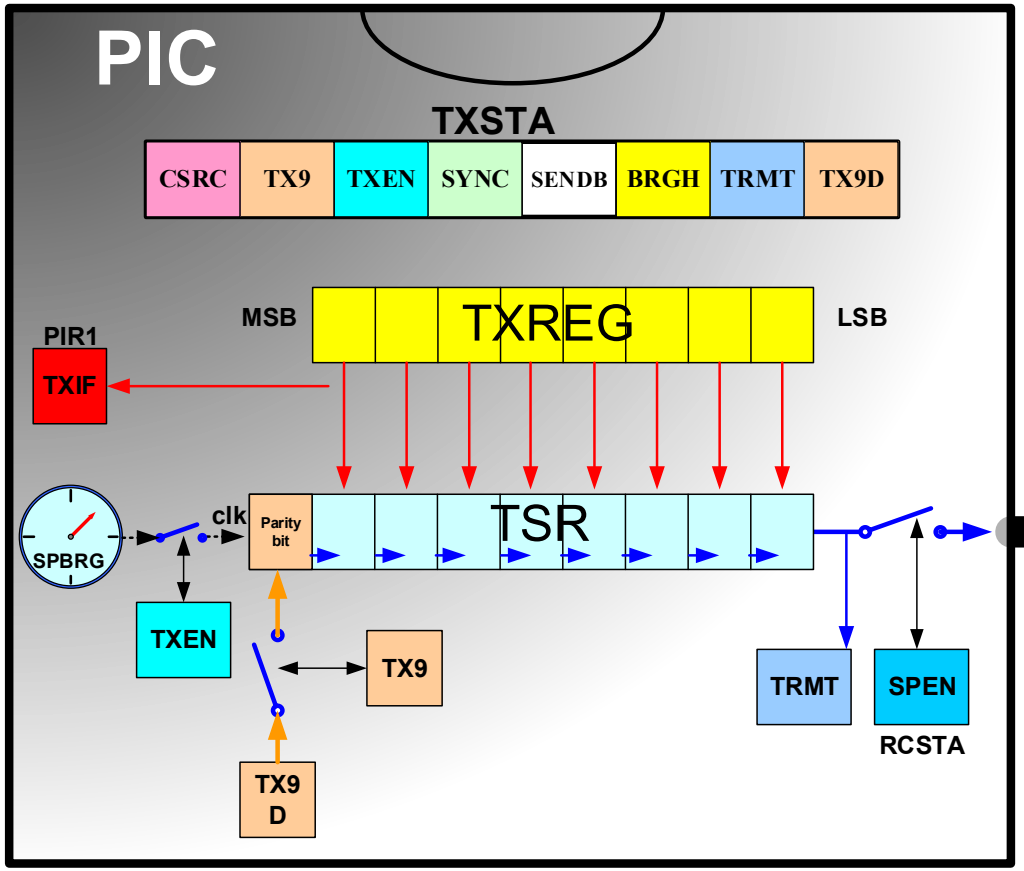
bit 4 **SYNC:** EUSART Mode Select bit
1 = Synchronous mode
0 = Asynchronous mode

bit 3 **SENDB:** Send Break Character bit
Asynchronous mode:
1 = Send Sync Break on next transmission (cleared by hardware upon completion)
0 = Sync Break transmission completed
Synchronous mode:
Don't care

bit 2 **BRGH:** High Baud Rate Select bit
Asynchronous mode:
1 = High speed
0 = Low speed
Synchronous mode:
Unused in this mode

bit 1 **TRMT:** Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

bit 0 **TX9D:** Ninth bit of Transmit Data
Can be address/data bit or a parity bit.



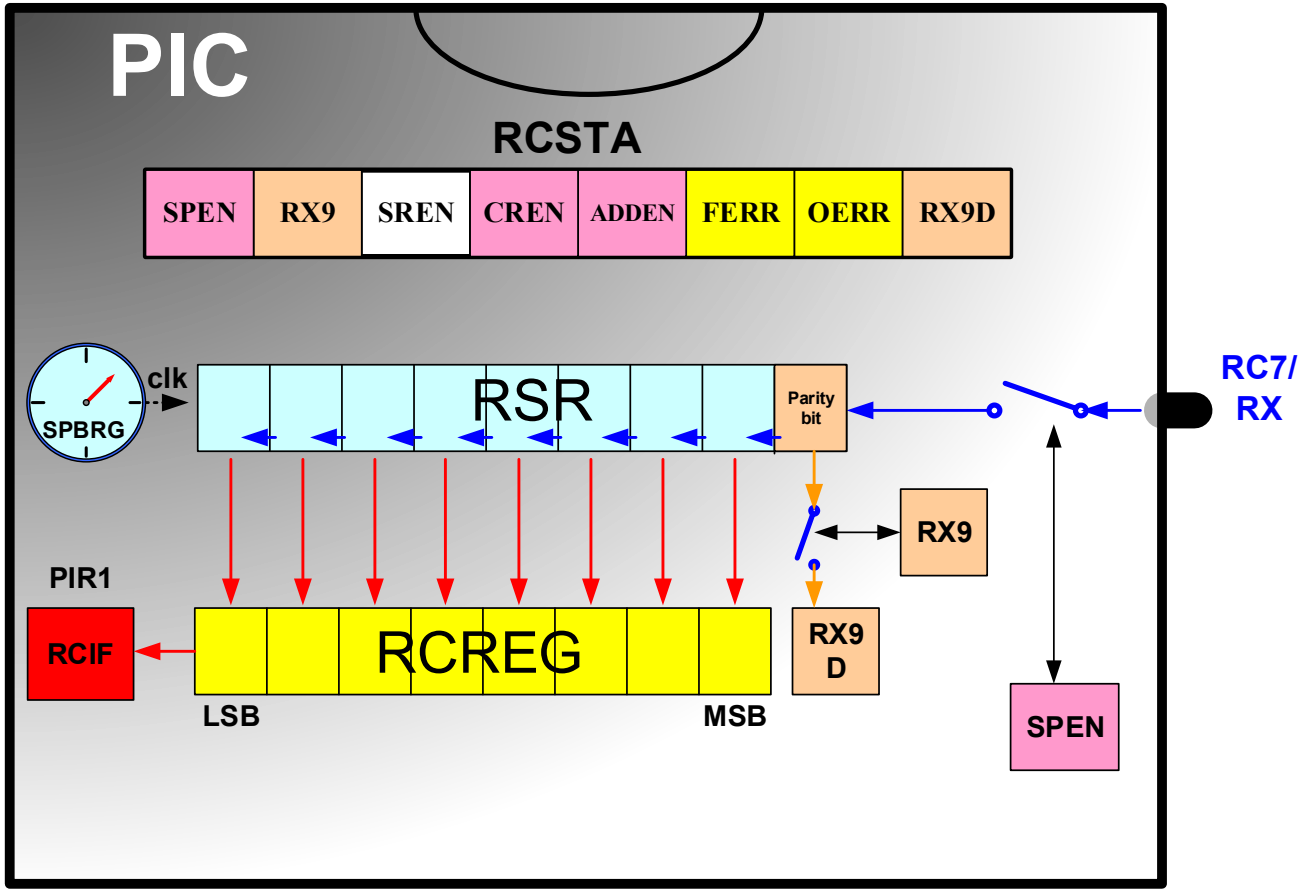
Lecture

USART (Asynchronous receiver)

- The RX pin is multiplexed with RC7 (input pin)
- The receiver is managed by the RCSTA register.
- When data is received serially, it is pushed into the Receive Shift Register (RSR). when it is full, the data in shifted to the RCREG and the RCIF is raised.

- The speed of receive is also managed by the SPBRG clock generator.
- SPEN is used to enable the serial port.

- NB: SPEN is used for both transmitter and receiver. You cannot use the transmitter alone and use RC7 is regular IO pins… same for the receiver.



USART (Asynchronous receiver)

REGISTER 12-2: RCSTA: RECEIVE STATUS AND CONTROL REGISTER(1)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

bit 7 **SPEN:** Serial Port Enable bit
1 = Serial port enabled (configures RX/DT and TX/CK pins as serial port pins)
0 = Serial port disabled (held in Reset)

bit 6 **RX9:** 9-bit Receive Enable bit
1 = Selects 9-bit reception
0 = Selects 8-bit reception

bit 5 **SREN:** Single Receive Enable bit
Asynchronous mode:
Don't care
Synchronous mode – Master:
1 = Enables single receive
0 = Disables single receive
This bit is cleared after reception is complete.
Synchronous mode – Slave
Don't care

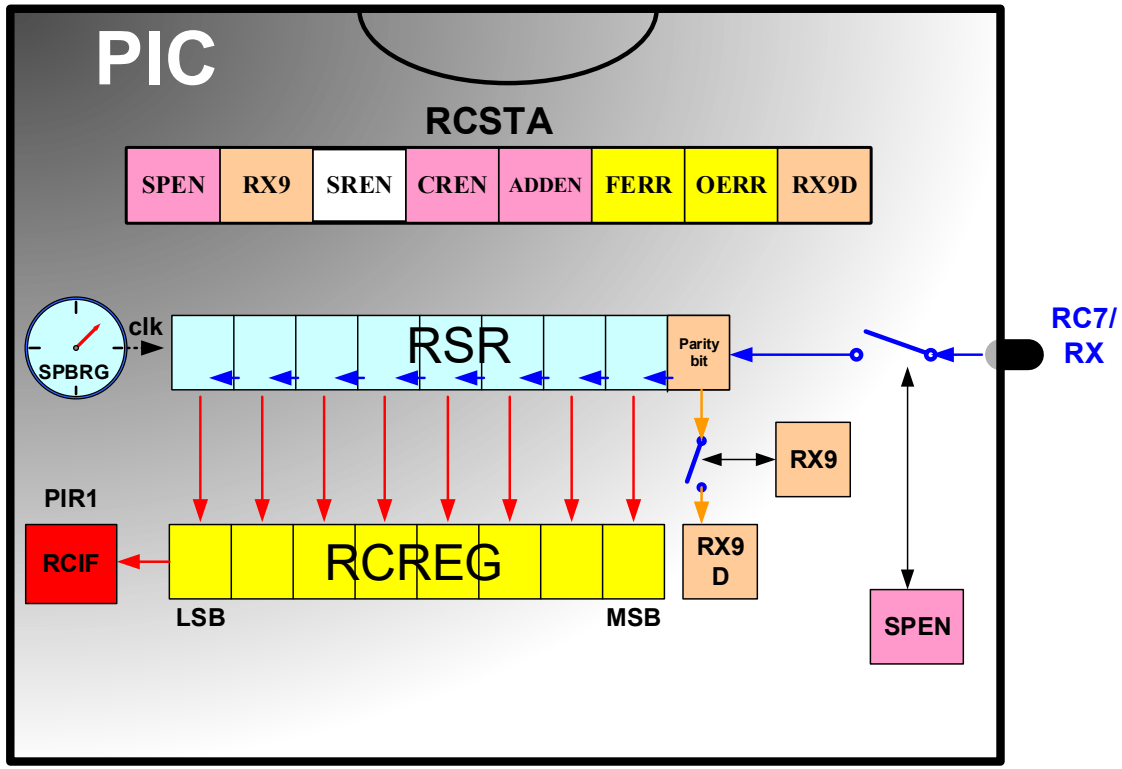
bit 4 **CREN:** Continuous Receive Enable bit
Asynchronous mode:
1 = Enables receiver
0 = Disables receiver
Synchronous mode:
1 = Enables continuous receive until enable bit CREN is cleared (CREN overrides SREN)
0 = Disables continuous receive

bit 3 **ADDEN:** Address Detect Enable bit
Asynchronous mode 9-bit (RX9 = 1):
1 = Enables address detection, enable interrupt and load the receive buffer when RSR<8> is set
0 = Disables address detection, all bytes are received and ninth bit can be used as parity bit
Asynchronous mode 8-bit (RX9 = 0):
Don't care

bit 2 **FERR:** Framing Error bit
1 = Framing error (can be updated by reading RCREG register and receive next valid byte)
0 = No framing error

bit 1 **OERR:** Overrun Error bit
1 = Overrun error (can be cleared by clearing bit CREN)
0 = No overrun error

bit 0 **RX9D:** Ninth bit of Received Data
This can be address/data bit or a parity bit and must be calculated by user firmware.



USART (bit rate configuration)

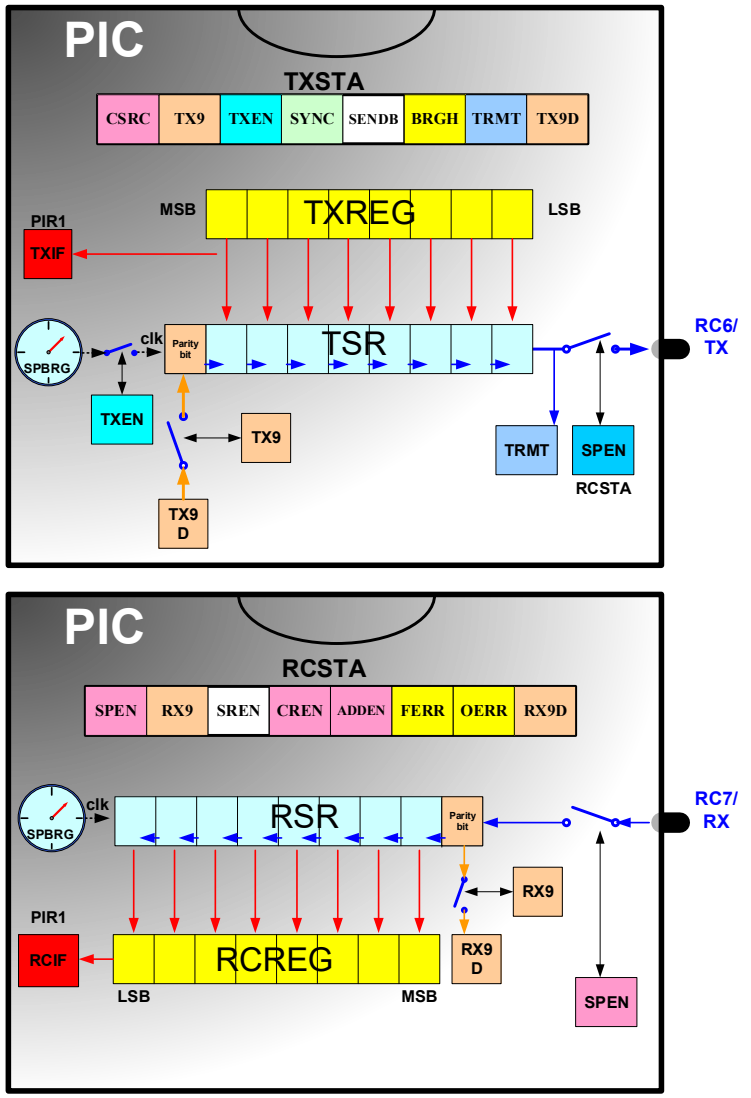
- The USART of the PIC16F887 is an enhanced USART, means that it has the ability to detect any Asynchronous data speed (baud rate).
- In this course, we are interested in having a fixed baud rate configuration between the PIC and the connected device.

- This is managed by the BAUDCTL baud control register.

REGISTER 12-3: BAUDCTL: BAUD RATE CONTROL REGISTER

R-0	R-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN
bit 7							bit 0

- The Baud Rate Generator (BRG) is an 8-bit or 16-bit timer that is dedicated to the support of both the asynchronous and synchronous EUSART operation.
- By default, the BRG operates in 8-bit mode.
- Setting theBRG16 bit of the BAUDCTL register selects 16-bit mode.



USART (bit rate configuration)

- The SPBRGH, SPBRG register pair determines the period of the free running baud rate timer.
- In Asynchronous mode the multiplier of the baud rate period is determined by both the BRGH bit of the TXSTA register and the BRG16 bit of the BAUDCTL register.

TABLE 12-3: BAUD RATE FORMULAS

Configuration Bits			BRG/EUSART Mode	Baud Rate Formula
SYNC	BRG16	BRGH		
0	0	0	8-bit/Asynchronous	Fosc/[64 (n+1)]
0	0	1	8-bit/Asynchronous	Fosc/[16 (n+1)]
0	1	0	16-bit/Asynchronous	
0	1	1	16-bit/Asynchronous	Fosc/[4 (n+1)]

- We can also use the table below to find out the value of SPBRG based on our settings
- For example, using a 4MHz oscillator, at high speed bard rate, in 8 bit BRG, the speed of 9600bit/sec is achieved by loading SPBRG by 25

TABLE 12-5: BAUD RATES FOR ASYNCHRONOUS MODES (CONTINUED)

BAUD RATE	SYNC = 0, BRGH = 1, BRG16 = 0											
	Fosc = 4.000 MHz			Fosc = 3.6864 MHz			Fosc = 2.000 MHz			Fosc = 1.000 MHz		
	Actual Rate	% Error	SPBRG value (decimal)	Actual Rate	% Error	SPBRG value (decimal)	Actual Rate	% Error	SPBRG value (decimal)	Actual Rate	% Error	SPBRG value (decimal)
300	—	—	—	—	—	—	—	—	—	300	0.16	207
1200	1202	0.16	207	1200	0.00	191	1202	0.16	103	1202	0.16	51
2400	2404	0.16	103	2400	0.00	95	2404	0.16	51	2404	0.16	25
9600	9615	0.16	25	9600	0.00	23	9615	0.16	12	—	—	—
10417	10417	0.00	23	10473	0.53	21	10417	0.00	11	10417	0.00	5
19.2k	19.23k	0.16	12	19.2k	0.00	11	—	—	—	—	—	—
57.6k	—	—	—	57.60k	0.00	3	—	—	—	—	—	—
115.2k	—	—	—	115.2k	0.00	1	—	—	—	—	—	—

USART (bit rate configuration)

- $Baud\ rate = \frac{F_{osc}}{16(SPBRG+1)}$
- $SPBRG = \frac{F_{osc}}{16 \times baud\ rate} - 1$
- For example at 4Mhz and 9600 Baud rate
- $SPBRG = \frac{4000000}{16 \times 9600} - 1 = 25.04 \approx 25$

TABLE 12-3: BAUD RATE FORMULAS

Configuration Bits			BRG/EUSART Mode	Baud Rate Formula
SYNC	BRG16	BRGH		
0	0	0	8-bit/Asynchronous	Fosc/[64 (n+1)]
0	0	1	8-bit/Asynchronous	Fosc/[16 (n+1)]
0	1	0	16-bit/Asynchronous	
0	1	1	16-bit/Asynchronous	Fosc/[4 (n+1)]

TABLE 12-5: BAUD RATES FOR ASYNCHRONOUS MODES (CONTINUED)

BAUD RATE	SYNC = 0, BRGH = 1, BRG16 = 0											
	Fosc = 4.000 MHz			Fosc = 3.6864 MHz			Fosc = 2.000 MHz			Fosc = 1.000 MHz		
	Actual Rate	% Error	SPBRG value (decimal)	Actual Rate	% Error	SPBRG value (decimal)	Actual Rate	% Error	SPBRG value (decimal)	Actual Rate	% Error	SPBRG value (decimal)
300	—	—	—	—	—	—	—	—	—	300	0.16	207
1200	1202	0.16	207	1200	0.00	191	1202	0.16	103	1202	0.16	51
2400	2404	0.16	103	2400	0.00	95	2404	0.16	51	2404	0.16	25
9600	9615	0.16	25	9600	0.00	23	9615	0.16	12	—	—	—
10417	10417	0.00	23	10473	0.53	21	10417	0.00	11	10417	0.00	5
19.2k	19.23k	0.16	12	19.2k	0.00	11	—	—	—	—	—	—
57.6k	—	—	—	57.60k	0.00	3	—	—	—	—	—	—
115.2k	—	—	—	115.2k	0.00	1	—	—	—	—	—	—



USART (bit rate configuration)

TABLE 12-1: REGISTERS ASSOCIATED WITH ASYNCHRONOUS TRANSMISSION

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
BAUDCTL	ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN	01-0 0-00	01-0 0-00
INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000x
PIE1	—	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	-000 0000	-000 0000
PIR1	—	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	-000 0000	-000 0000
RCREG	EUSART Receive Data Register								0000 0000	0000 0000
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	0000 000x
SPBRG	BRG7	BRG6	BRG5	BRG4	BRG3	BRG2	BRG1	BRG0	0000 0000	0000 0000
SPBRGH	BRG15	BRG14	BRG13	BRG12	BRG11	BRG10	BRG9	BRG8	0000 0000	0000 0000
TRISC	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0	1111 1111	1111 1111
TXREG	EUSART Transmit Data Register								0000 0000	0000 0000
TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D	0000 0010	0000 0010

Legend: x = unknown, – = unimplemented read as '0'. Shaded cells are not used for Asynchronous Transmission.

TABLE 12-2: REGISTERS ASSOCIATED WITH ASYNCHRONOUS RECEPTION

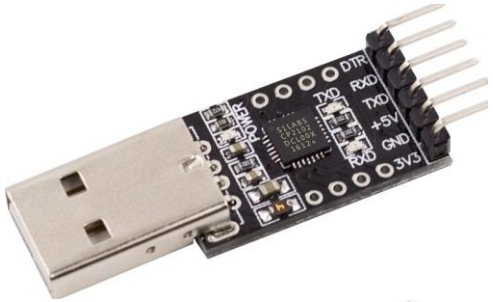
Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
BAUDCTL	ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN	01-0 0-00	01-0 0-00
INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000x
PIE1	—	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	-000 0000	-000 0000
PIR1	—	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	-000 0000	-000 0000
RCREG	EUSART Receive Data Register								0000 0000	0000 0000
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	0000 000x
SPBRG	BRG7	BRG6	BRG5	BRG4	BRG3	BRG2	BRG1	BRG0	0000 0000	0000 0000
SPBRGH	BRG15	BRG14	BRG13	BRG12	BRG11	BRG10	BRG9	BRG8	0000 0000	0000 0000
TRISC	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0	1111 1111	1111 1111
TXREG	EUSART Transmit Data Register								0000 0000	0000 0000
TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D	0000 0010	0000 0010

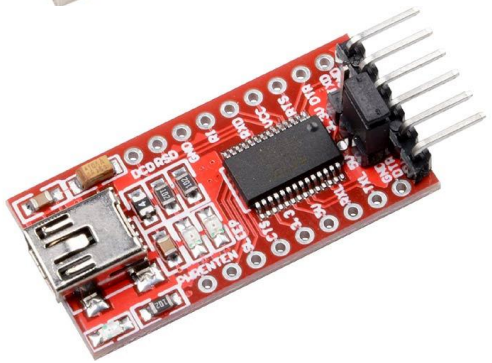
Legend: x = unknown, – = unimplemented read as '0'. Shaded cells are not used for Asynchronous Reception.



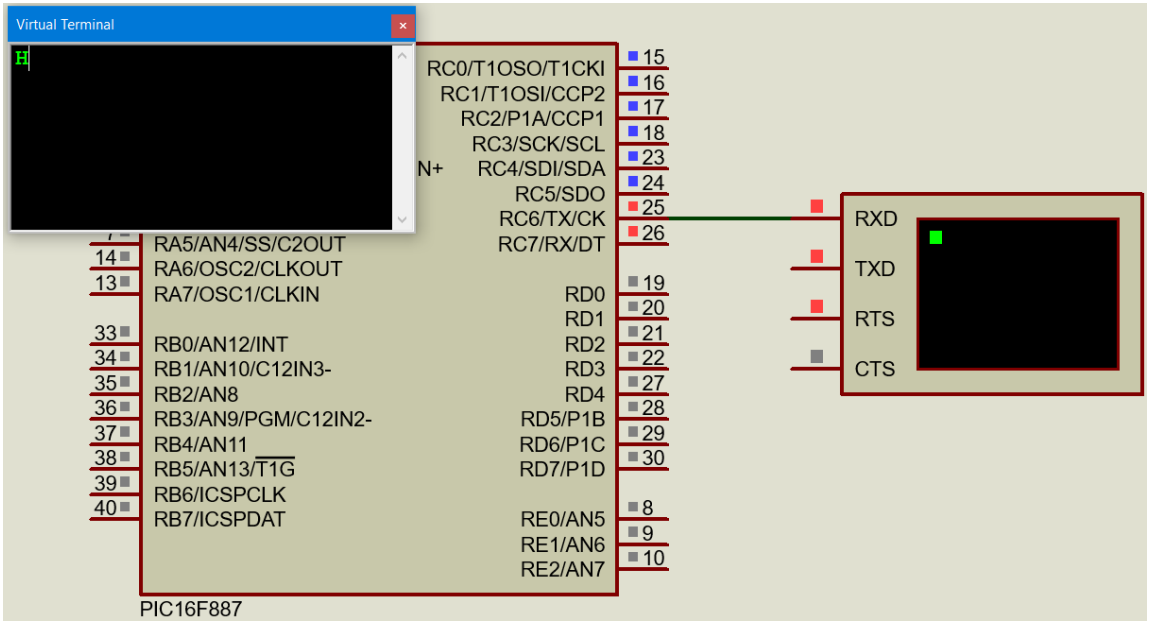
Activity 17 : USART (TX mode)

- In this activity, we will apply the rules of sending a message through USART.
- First we will start by sending a character serially.
- In Proteus, the “Virtual Terminal” is used to mimic the behavior of any USART receiver. This will show the sent characters on the screen.
- In reality, if we want to interface the PIC with the PC, we have to use the USB to Serial converter as the new PCs do not have a real serial port (the old DB9 or DB25 serial port with RS232 protocol). We can use the FT232 module, the CH340 module or the CP2102 module









Activity 17 : USART (TX mode)

- In the Setup, the baud rate is set to 9600 through SPBRG. The TX pin is configured as output. The USART is enabled in TX asynchronous mode.

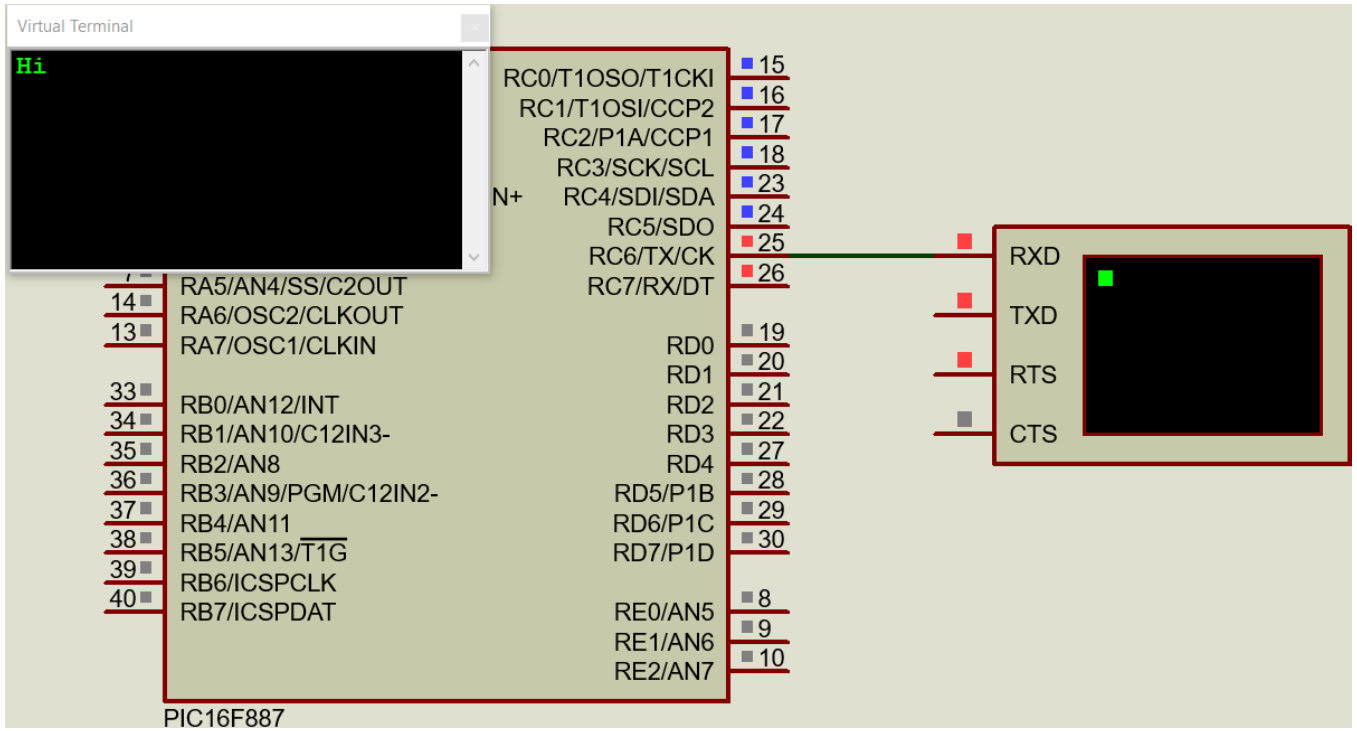
- In the main code, the character H is loaded into the TXREG. Nothing else is required…all the transmission is processed automatically.

```
33    MAIN:
34        CALL    SETUP
35        MOVLW   'H'
36        MOVWF   TXREG          ; SEND 'H'
37        GOTO    $
```

```
63    SETUP:                     ; by default we are in BANK0
64        CLRF    PORTC
65        BSF     RP0            ; RP1 IS BY DEFAUT 0
66        BCF     TRISC6         ; TX IS OUTPUT
67        MOVLW   25             ; 9600 BIT/S
68        MOVWF   SPBRG
69        CLRF    SPBRGH
70        MOVLW   00100100B      ; ENABLE TX, BRGH=1
71        MOVWF   TXSTA
72        MOVLW   01100000B      ; configuring OSCCON register
73        MOVWF   OSCCON         ; to select 4Mhz oscillator
74        BCF     RP0            ; access to BANK 0, RP0 = 0 , RP1 = 0
75        BSF     SPEN           ; ENABLE SERIAL PORT (BIT IN RCSTA)
76        RETURN
```

Activity 17 : USART (TX mode)

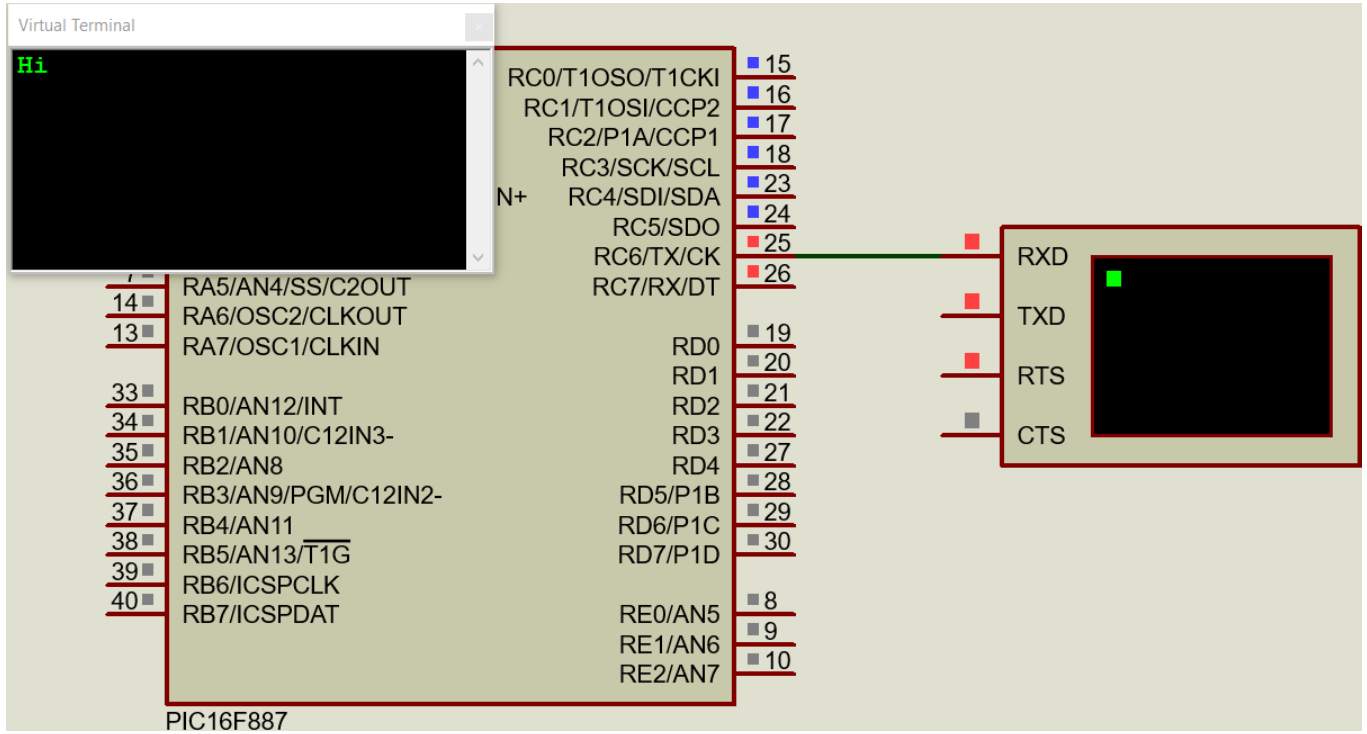
- The code is modified to send a complete phrase "HCT Dubai" serially. The letters are sent one by one. The simulation shows that the serial terminal is receiving only "Hi".




```
33  MAIN:
34      CALL    SETUP
35      MOVLW   'H'
36      MOVWF   TXREG           ; SEND 'H'
37      MOVLW   'C'
38      MOVWF   TXREG           ; SEND 'C'
39      MOVLW   'T'
40      MOVWF   TXREG           ; SEND 'T'
41      MOVLW   ' '
42      MOVWF   TXREG           ; SEND SPACE
43      MOVLW   'D'
44      MOVWF   TXREG           ; SEND 'D'
45      MOVLW   'u'
46      MOVWF   TXREG           ; SEND 'u'
47      MOVLW   'b'
48      MOVWF   TXREG           ; SEND 'b'
49      MOVLW   'a'
50      MOVWF   TXREG           ; SEND 'a'
51      MOVLW   'i'
52      MOVWF   TXREG           ; SEND 'i'
53      GOTO    $
```

Activity 17 : USART (TX mode)

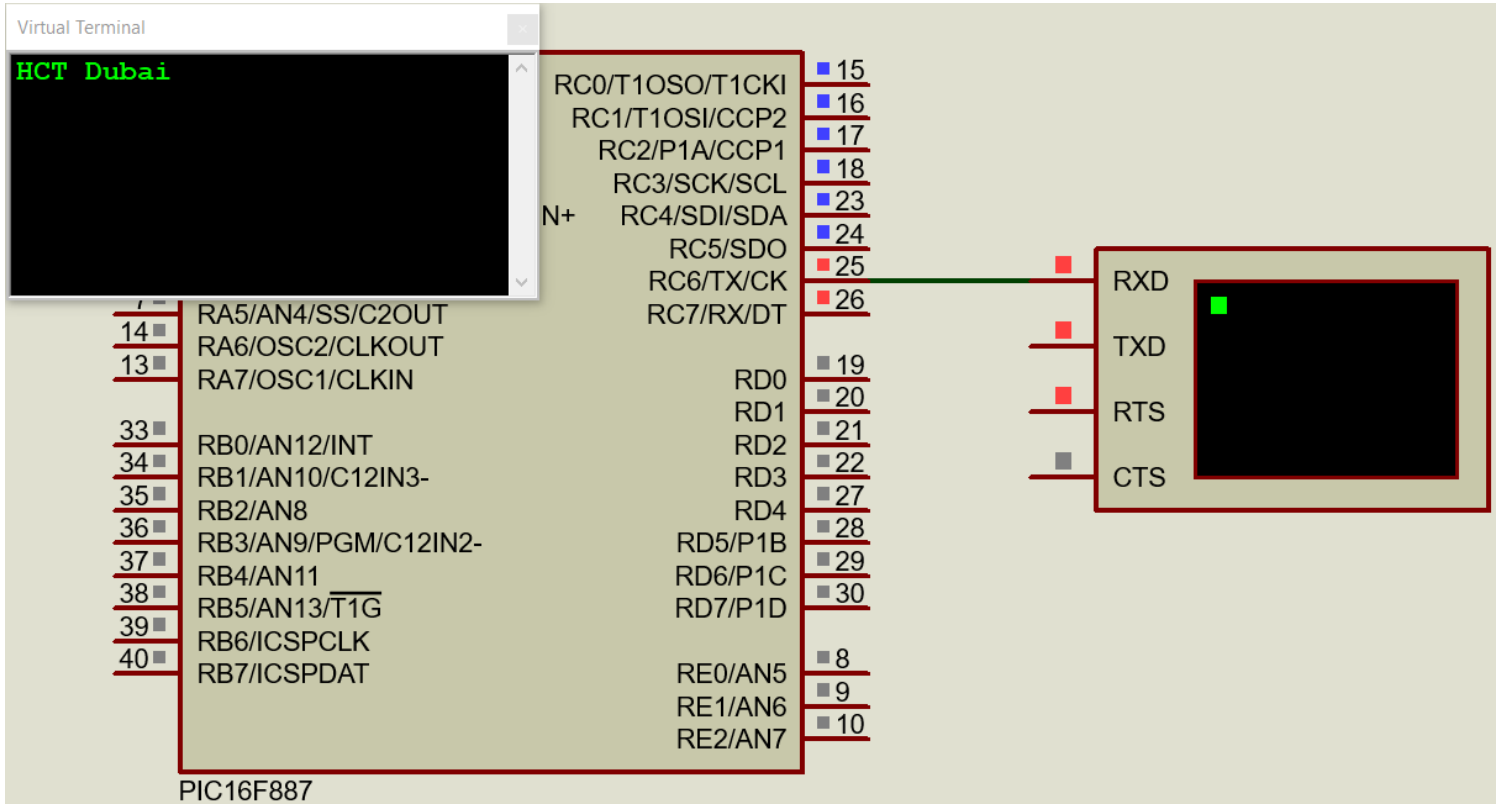
- Actually, only the first and last letter is received. The second letter is sent to the TXREG while the TSR is still sending the previous letter. It needs only 2µs to send the letter to TXREG but needs about 10/9600 seconds to send the data serially (10 because we have 8 bits, 1 start and 1 stop) = 1.04ms




```
33  MAIN:
34      CALL    SETUP
35      MOVLW   'H'
36      MOVWF   TXREG           ; SEND 'H'
37      MOVLW   'C'
38      MOVWF   TXREG           ; SEND 'C'
39      MOVLW   'T'
40      MOVWF   TXREG           ; SEND 'T'
41      MOVLW   ' '
42      MOVWF   TXREG           ; SEND SPACE
43      MOVLW   'D'
44      MOVWF   TXREG           ; SEND 'D'
45      MOVLW   'u'
46      MOVWF   TXREG           ; SEND 'u'
47      MOVLW   'b'
48      MOVWF   TXREG           ; SEND 'b'
49      MOVLW   'a'
50      MOVWF   TXREG           ; SEND 'a'
51      MOVLW   'i'
52      MOVWF   TXREG           ; SEND 'i'
53      GOTO    $
```

Activity 17 : USART (TX mode)

- In order to send all letters correctly, we can wait for at least 1.04ms per character, or we can check the TRMT flag bit. The code is adjusted by testing the TRMT located in bank 1.




```
33      MAIN:
34          CALL    SETUP
35          MOVLW   'H'
36          CALL    USART_SEND  ; SEND 'H'
37          MOVLW   'C'
38          CALL    USART_SEND  ; SEND 'C'
39          MOVLW   'T'
40          CALL    USART_SEND  ; SEND 'T'
41          MOVLW   ' '
42          CALL    USART_SEND  ; SEND SPACE
43          MOVLW   'D'
44          CALL    USART_SEND  ; SEND 'D'
45          MOVLW   'u'
46          CALL    USART_SEND  ; SEND 'u'
47          MOVLW   'b'
48          CALL    USART_SEND  ; SEND 'b'
49          MOVLW   'a'
50          CALL    USART_SEND  ; SEND 'a'
51          MOVLW   'i'
52          CALL    USART_SEND  ; SEND 'i'
53          GOTO    $
54
55      USART_SEND:
56          MOVWF   TXREG
57          BSF     RP0         ; JUMP TO BANK 1
58          BTFSS   TRMT        ; CHECK TRMT
59          GOTO    $-1         ; WAIT UNTIL DATA IS SENT
60          BCF     RP0         ; BACK TO BANK 0
61          RETURN
```



Test your knowledge

1. Calculate the SPBRG in order to have a baud rate of 115200. Do the calculation for BRGH =0 and BRGH=1 and compare the results.
2. Why it is required to wait after loading the TXREG by a value. What is the delay required to send a byte (with start and stop bits) at 115200 bit/sec.
3. What is the time required to send a single bit at 1200 baud rate.
4. What is the time required to send a byte at 1200 baud rate.
5. Check on the web what is the protocol used to communicate with a GSM modem sim900, the GPS NEO 6M and the Bluetooth HC-5. Also check the protocol used to communicate with the RFID RC522 and the nrf24L01 transceiver.
6. What should be in SPBRG to have a baud rate of 19200 with a PIC running at 4Mhz. Make sure to select the one with less error



Week 10

Lab 8 / LO4

LCD with sensor

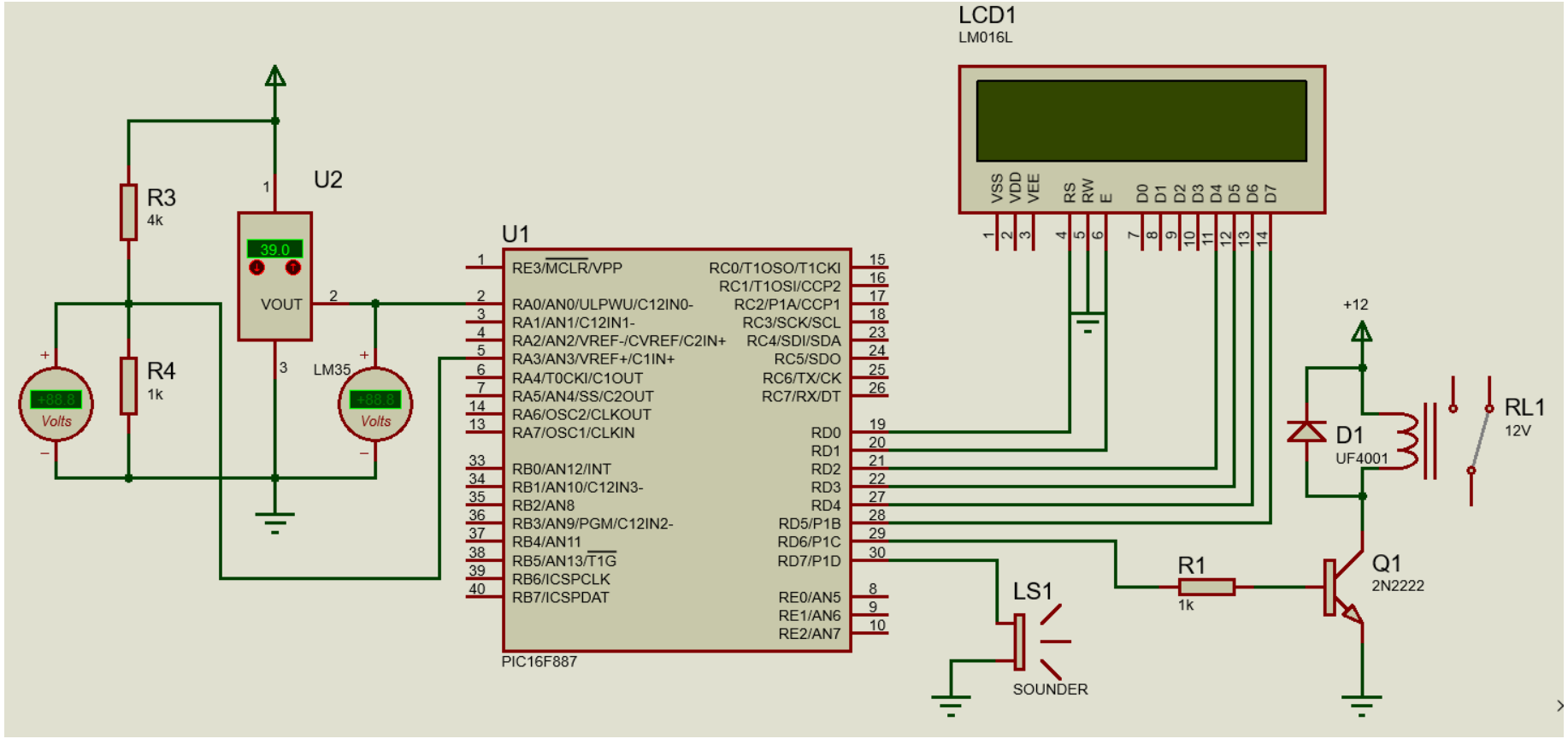
Activity 16C : A2D with LCD and Sensor

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. PIC programmer
  4. Power supply 5V and 12V
  5. Breadboard wires (male-male)

- Required components:
  1. PIC 16F887
  2. Resistor from 100 to 330 x 1 (for the lcd back light)
  3. Resistor 1k x1
  4. LCD 16 characters 2 lines x1
  5. Potentiometer 10k x2 (one for the contract and one for the Vref+)
  6. LM35
  7. Piezo buzzer
  8. Transistor 2n2222 or any NPN equivalent x1
  9. Any relay (5 to 12V) x1
  10. Fast diode UF4001 x1

Activity 16C : A2D with LCD and Sensor

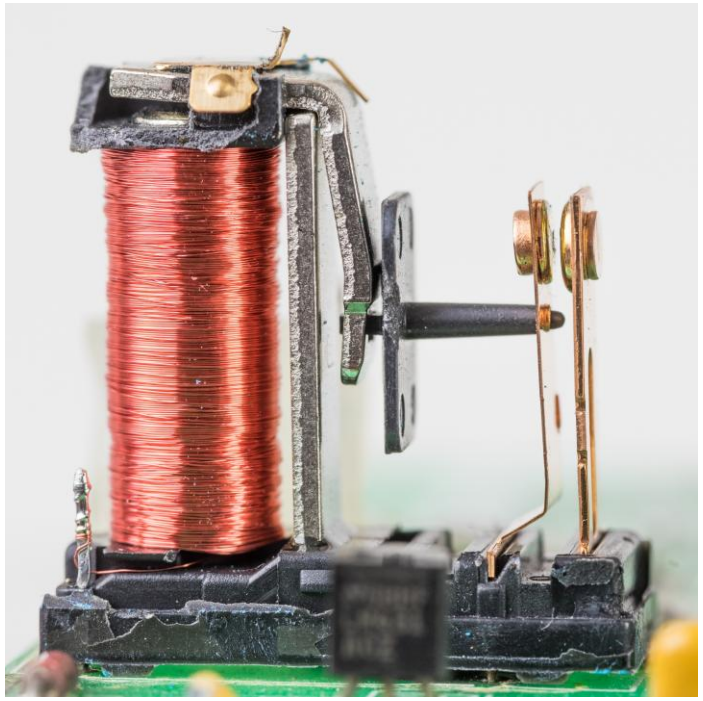
- The lab activity is same as the one done in 16C but with extra relay connected on RD6.
- Activate the relay when the temperature exceed 40. (2 lines of code are required)



Relay

- A relay is an electromechanical switch that opens and closes under the control of another electrical circuit.
- In the original form, the switch is operated by an electromagnet to open or close one or many sets of contacts.

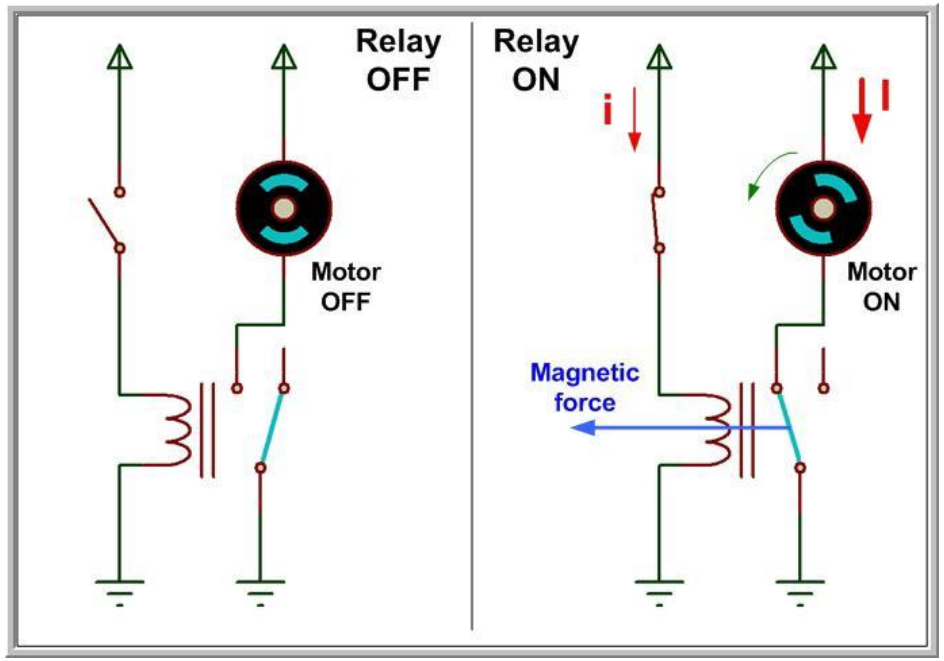
- When a current flows through the coil, the resulting magnetic field attracts an armature that is mechanically linked to a moving contact. The movement either makes or breaks a connection with a fixed contact. When the coil current is switched off, the armature is returned by a force approximately half as strong as the magnetic force to its relaxed position.

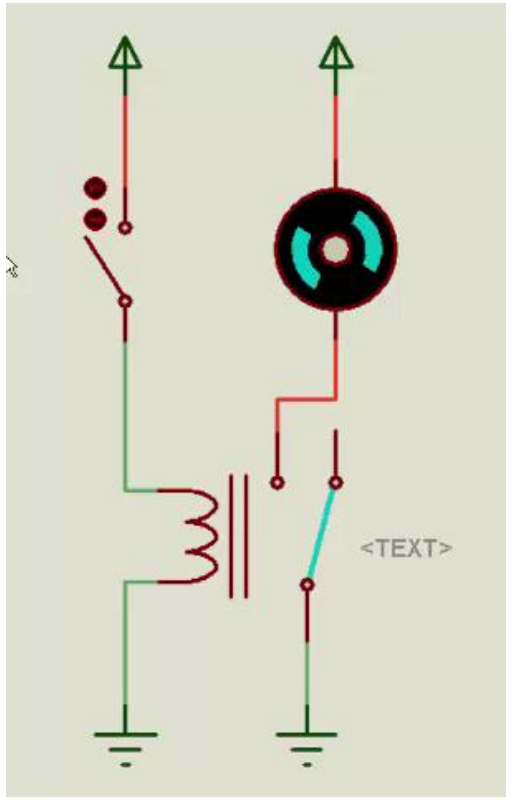




Relay

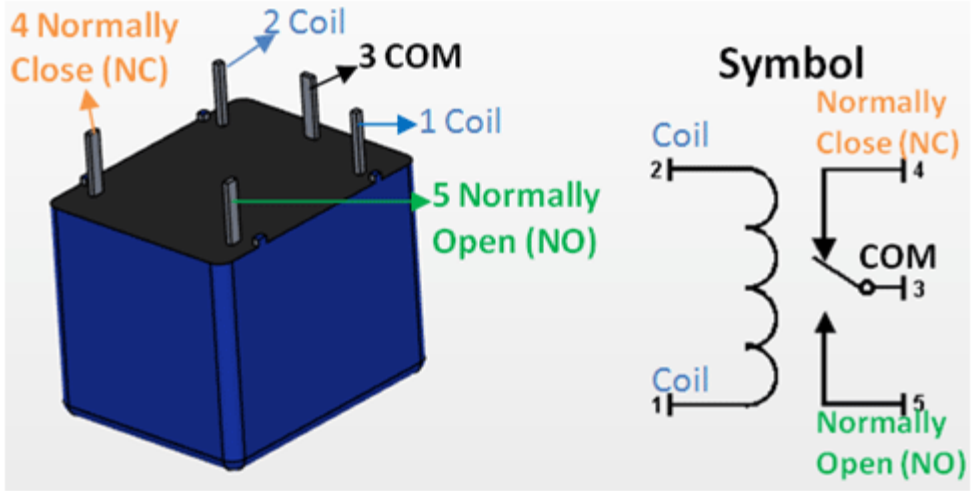
- The relay is used if it is wanted to drive a high power motor without using a power transistor circuits.
- The relay is driven using a signal transistor since its coil consumes around 30 mA and 100 mA depending on its size and manufacturer

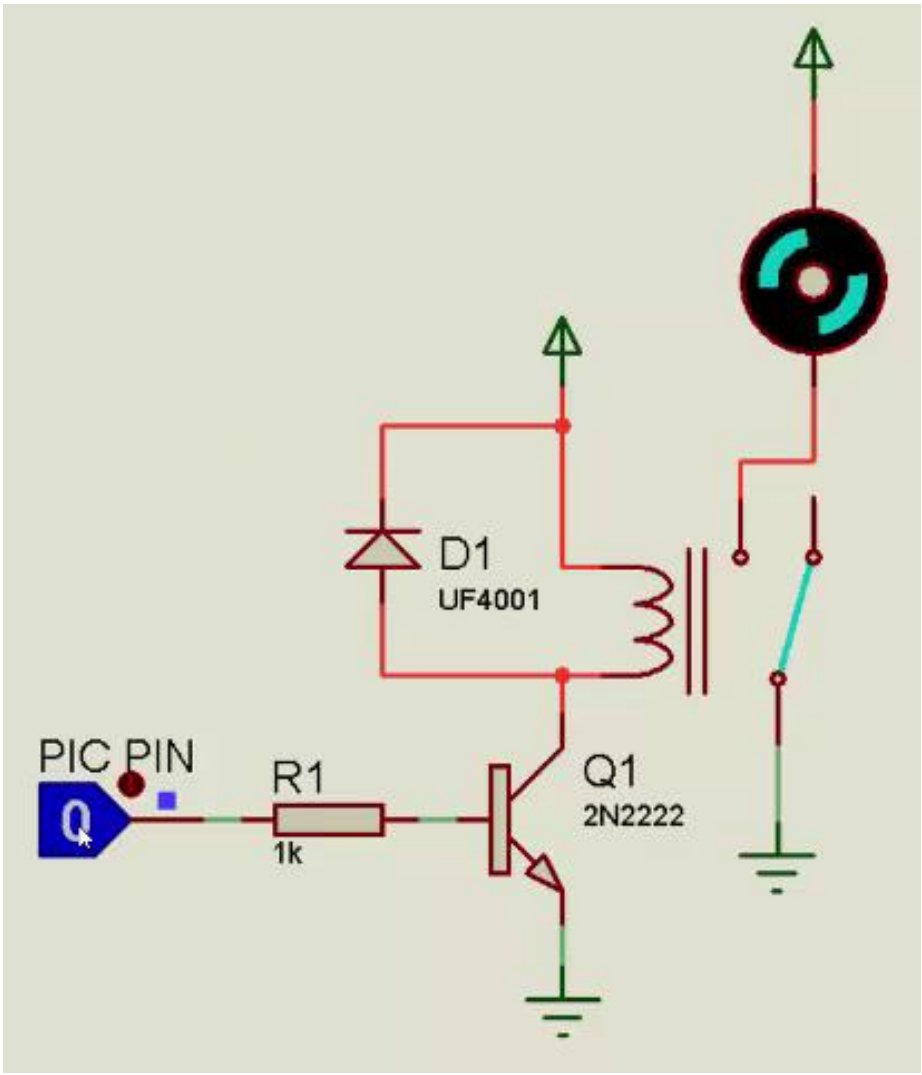




Relay

- Due to the current of the relay that exceeds the limit of the PIC pin, a current amplifier transistor is used to drive it.
- Thus, the PIC sends 1 logic to the transistor that amplify the relay current to switch ON… to drive the motor ON.
- A freewheeling diode is required due to the inductive effect of the relay coil







Week 11

Lecture 19 / LO4

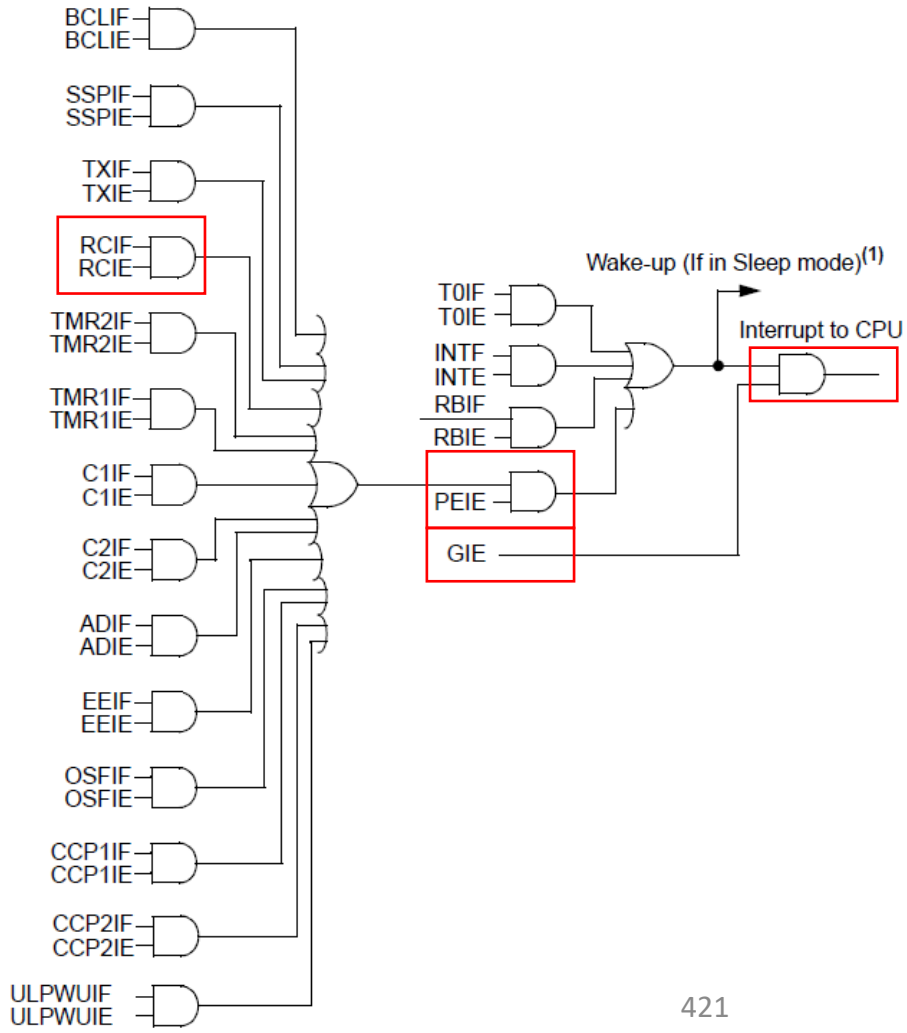
USART

USART in RX mode

Activity 18 : USART (RX mode)

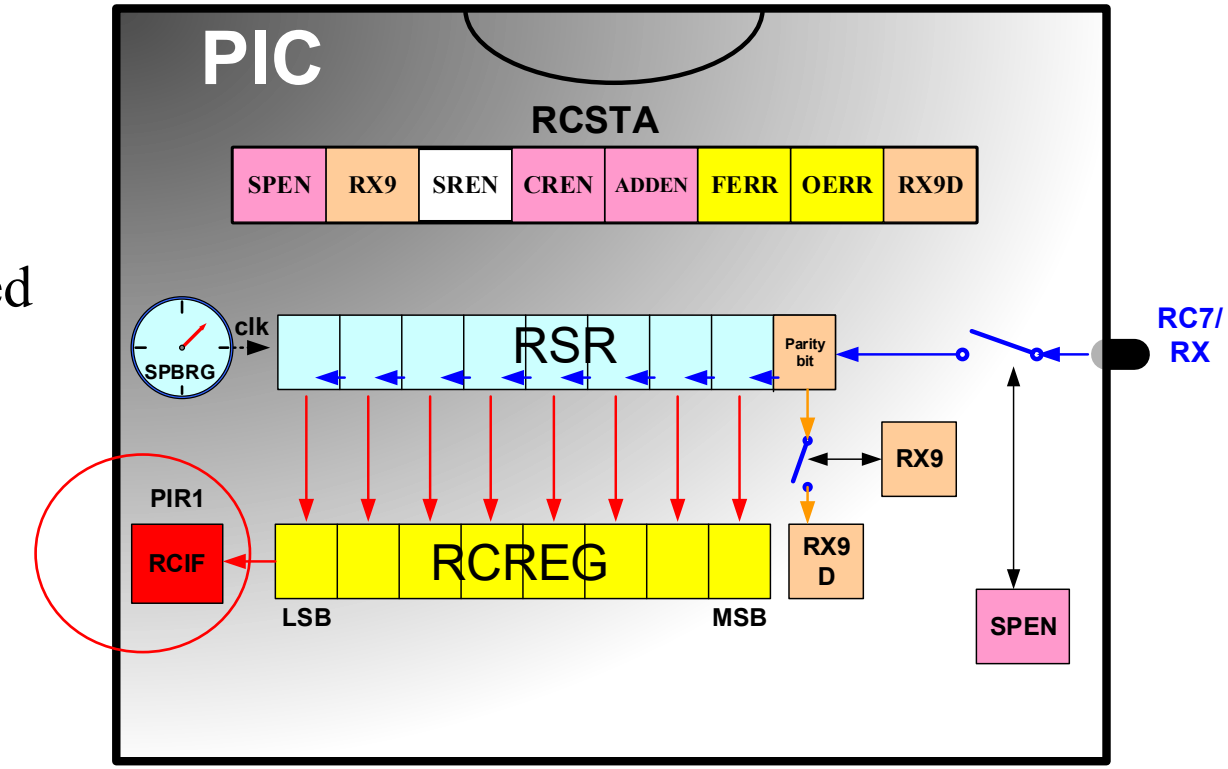
- This activity is a continuation of the previous activity. The USART receiver is now added to the code.
- One of the main issue to consider in the receiver mode is that the PIC should be interrupted to read the received data as it is unknown when it will be available.
- To enable the receiver interrupt, the RCIE bit in PIE1 should be enabled. Also the peripheral bit and the GIE bits should be 1.

- In this activity, the received data is compared to ‘H’. If it is equal, the PIC will send ‘HCT Dubai’. If not, nothing will happed.



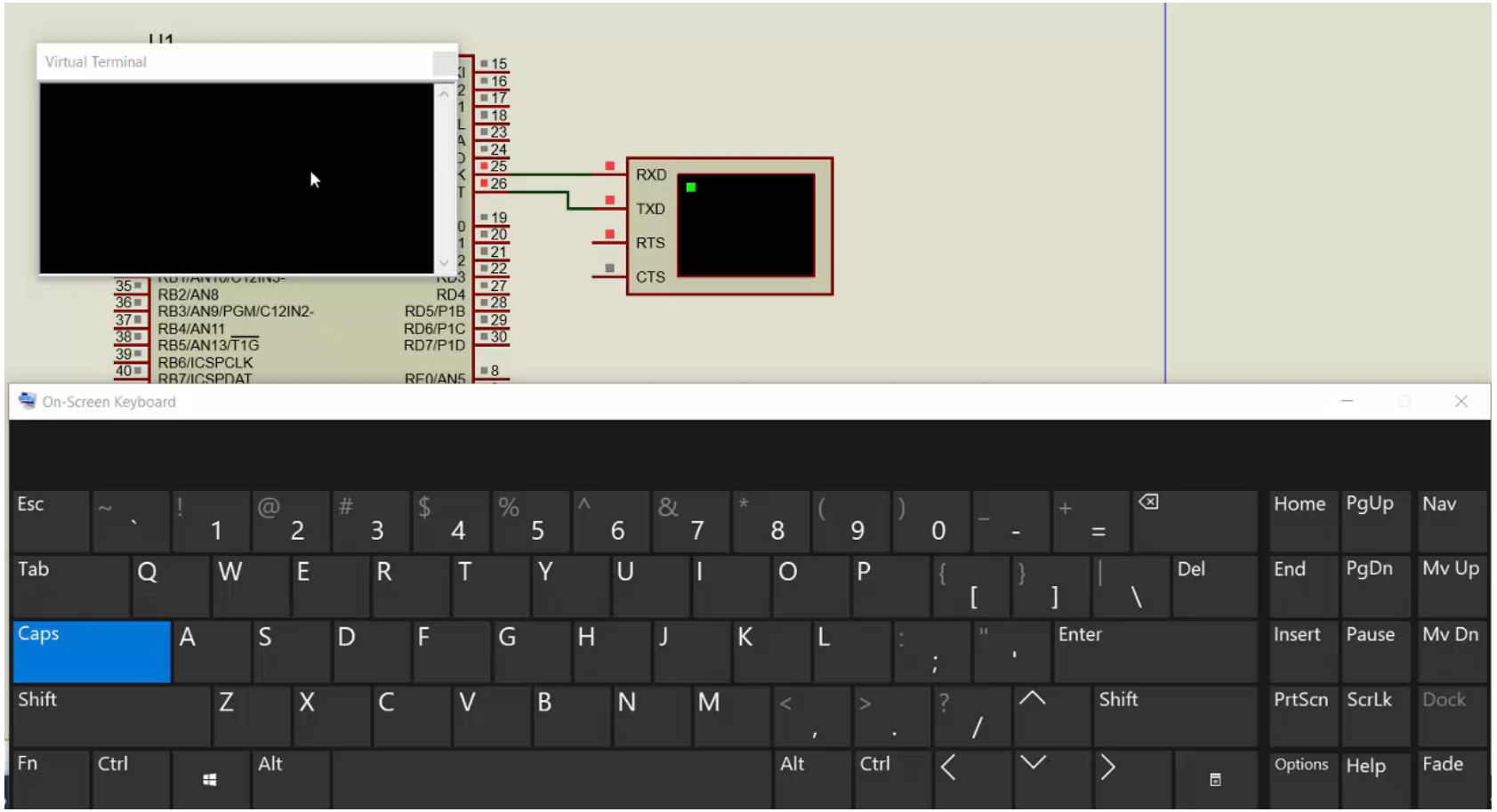
Activity 18 : USART (RX mode)

- When the data is ready in the RSR, the information in shifted to the RCREG and the flag RCIF is raised.
- If all condition are meet, the PC will jump to the address 0x04.
- The RCIF cannot be cleared (as all other flags in PIC) by the BCF RCIF. To clear the flag, it is sufficient to read the RCREG, even if it is not used for any task.



Activity 18 : USART (RX mode)

- When



Activity 18 : USART (RX mode)

- When

```
58      MAIN:
59          CALL    SETUP
60          GOTO    $
61
62      SETUP:                  ; by default we are in BANK0
63          CLRF    PORTC
64          BSF     RP0         ; RP1 IS BY DEFAUT 0
65          MOVLW   01100000B   ; configuring OSCCON register
66          MOVWF   OSCCON      ; to select 4Mhz oscillator
67          BCF     TRISC6      ; TX IS OUTPUT
68          BSF     TRISC7      ; RX IS INPUT
69          BSF     RCIE        ; ENABLE RECEIVE INT
70          MOVLW   25          ; 9600 BIT/S
71          MOVWF   SPBRG
72          CLRF    SPBRGH
73          MOVLW   00100100B   ; ENABLE TX, BRGH=1
74          MOVWF   TXSTA
75          BCF     RP0         ; access to BANK 0, RP0 = 0 , RP1 = 0
76          MOVLW   10010000B
77          MOVWF   RCSTA
78          BSF     PEIE        ; ENABLE PERIPHERALS
79          BSF     GIE         ; ENABLE INT
80          RETURN
```

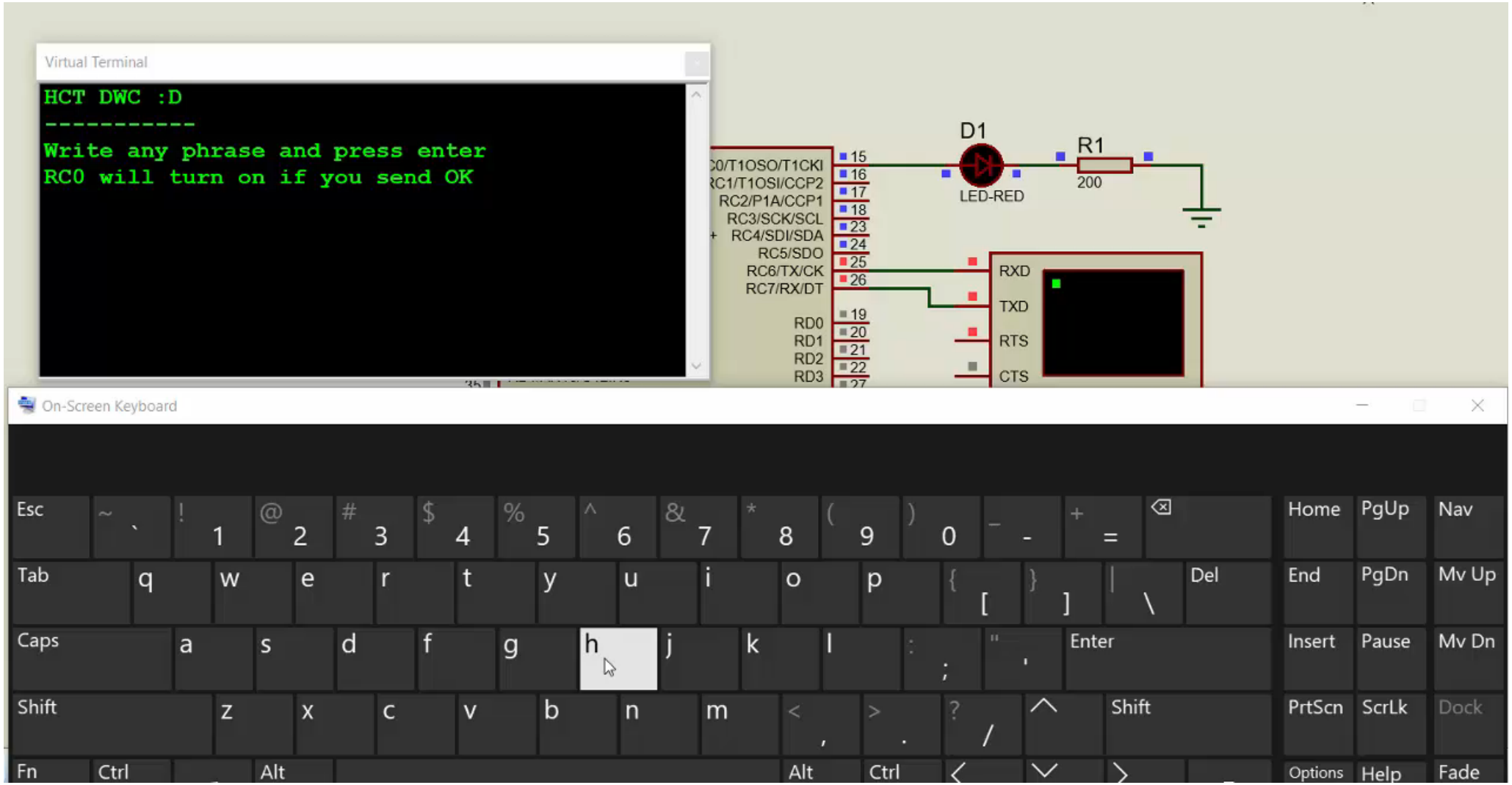
```
28          ORG     0           ; reset vector
29          GOTO    MAIN
30          ORG     0X4         ; interrupt vector
31          MOVF    RCREG, W
32          XORLW   'H'
33          BTFSC   ZERO
34          CALL    SEND_MSG
35          RETFIE
36
37      SEND_MSG:
38          MOVLW   'H'
39          CALL    USART_SEND  ; SEND 'H'
40          MOVLW   'C'
41          CALL    USART_SEND  ; SEND 'C'
42          MOVLW   'T'
43          CALL    USART_SEND  ; SEND 'T'
44          MOVLW   ' '
45          CALL    USART_SEND  ; SEND SPACE
46          MOVLW   'D'
47          CALL    USART_SEND  ; SEND 'D'
48          MOVLW   'u'
49          CALL    USART_SEND  ; SEND 'u'
50          MOVLW   'b'
51          CALL    USART_SEND  ; SEND 'b'
52          MOVLW   'a'
53          CALL    USART_SEND  ; SEND 'a'
54          MOVLW   'i'
55          CALL    USART_SEND  ; SEND 'i'
56          RETURN
```



Activity 19C : USART in TX and RX mode

- In this activity, the USART will be used in C in two modes.
- It is required to send back the received message. The C code is an advance copy of the assembly code done in the previous activity.
- The code is designed to receive and send messages, also compare some received character in the message.
- For example, if the message is “OK”, the led on RC0 will turn on.

Activity 19C : USART IN TX AND RX MODE



Activity 19C : USART IN TX AND RX MODE

- The setup function is similar to the one in assembly. The selected baud rate is 9600.

```
47  void setup() {
48      PORTC = 0;
49      OSCCON = 0b01100000;     // select 4MHz oscillator
50      TRISC = 0b10000000;      // and as out
51      TXSTA = 0b00100100;      // asyn mode
52      RCSTA = 0b10010000;      // serial port enables with continious receive
53      SPBRG = 25; // 9600 bit /s
54      GIE = 1;     // enable int
55      PEIE = 1;
56      RCIE = 1;
57  }
58
59  void main(void) {
60      setup();
61      USART_SEND((char*) "HCT DWC :D");
62      USART_SEND_NEW_LINE();
63      USART_SEND((char*) "----------");
64      USART_SEND_NEW_LINE();
65      USART_SEND((char*) "Write any phrase and press enter ");
66      USART_SEND_NEW_LINE();
67      USART_SEND((char*) "RC0 will turn on if you send OK");
68      USART_SEND_NEW_LINE();
69      while (1);
70  }
```

Activity 19C : USART IN TX AND RX MODE

- A “phrase” buffer of 30 spaces is created with char type.
- In the ISR, every received character is saved in the buffer in the pointer location. The pointer is incremented every time a new character is received until the characters \n or \r are received.
- This means that the user typed “enter” on the keyboard.

- If so, the phrase is sent back to the user.

- If the user typed “OK”, the led turns ON, else it turns OFF

```
29    unsigned int pointer = 0;
30    char phrase[30];
31
32    void __interrupt() my_isr_routine() {
33        phrase[pointer] = RCREG; // once a character is received store it in ch
34        pointer++; // with index pointer}
35        if (RCREG == '\n' || RCREG == '\r') {
36            USART_SEND((char*) "Sent msg was: ");
37            USART_SEND((char*) phrase);
38            pointer = 0;
39            if (phrase[0] == 'O' && phrase[1] == 'K') {
40                RC0 = 1;
41            } else {
42                RC0 = 0;
43            }
44        }
45    }
```

Activity 19C : USART IN TX AND RX MODE

- The functions used to send the phrase are included in usart.h file
- The USART_SEND function sends a phrase starting from its pointer zero until \r
- The USART_SEND_NEW_LINE function sends the characters \n and \r as enter key.
- NB: testing the TXIF or the TRMT flags have almost the same functionality in testing if the TX buffer if full of empty to proceed with the next transmission

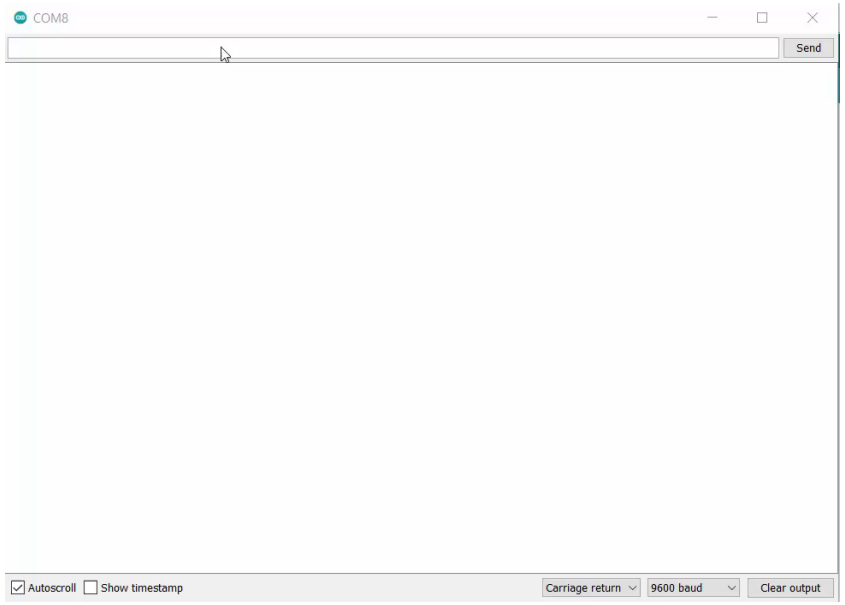
```
1   void USART_SEND(char *usart_dat);
2   void USART_SEND_NEW_LINE();
3
4   void USART_SEND(char *usart_dat) {
5       while (*usart_dat) {
6           if (*usart_dat == '\r') {
7               USART_SEND_NEW_LINE();
8               break;
9           }
10          TXREG = *usart_dat++;
11          while (!TXIF);
12
13      }
14  }
15
16  void USART_SEND_NEW_LINE() {
17      TXREG = '\n'; // once TXREG has been cleared load it with new line
18      while (!TXIF); //  continue to loop while the TXREG still has data
19      TXREG = '\r'; // once TXREG has been cleared load it with carriage return
20      while (!TXIF); //  continue to loop while the TXREG still has data
21  }
22
```

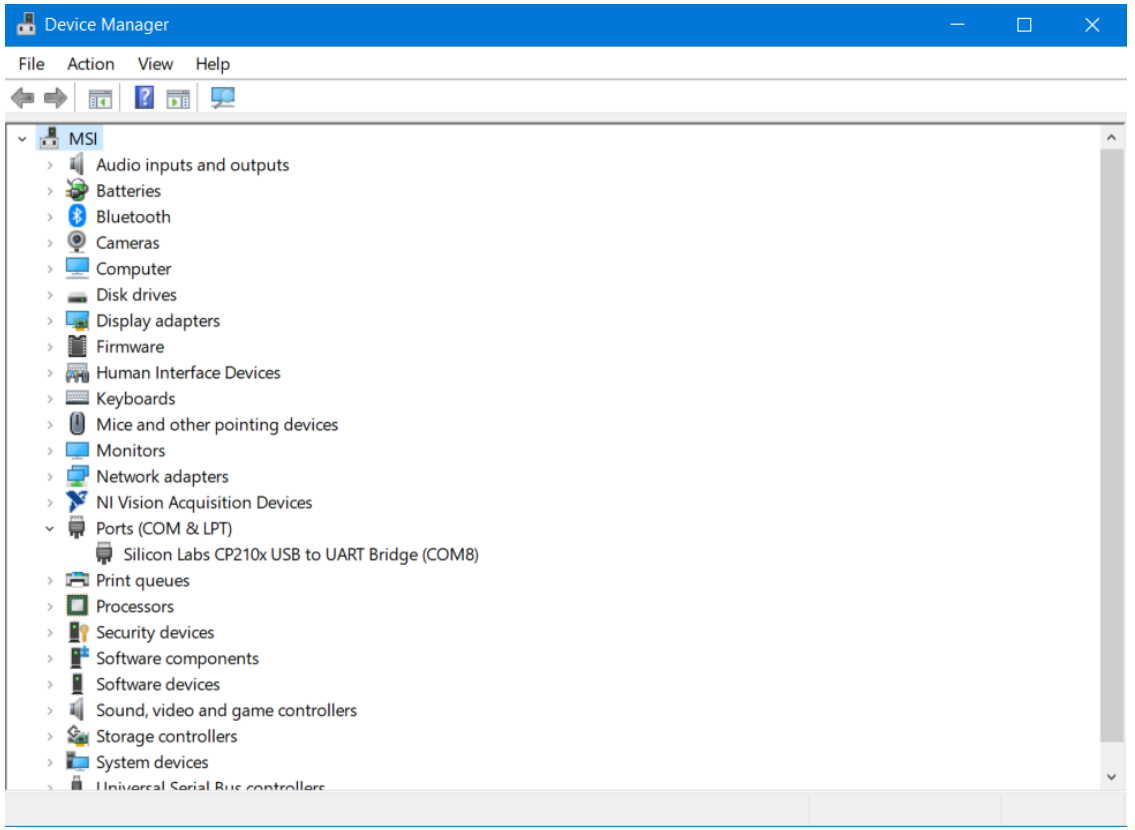
bit 4 **TXIF:** EUSART Transmit Interrupt Flag bit
1 = The EUSART transmit buffer is empty (cleared by writing to TXREG)
0 = The EUSART transmit buffer is full

bit 1 **TRMT:** Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

Activity 19C : USART IN TX AND RX MODE

- The functions used to send the phrase are included in usart.h file







Week 11

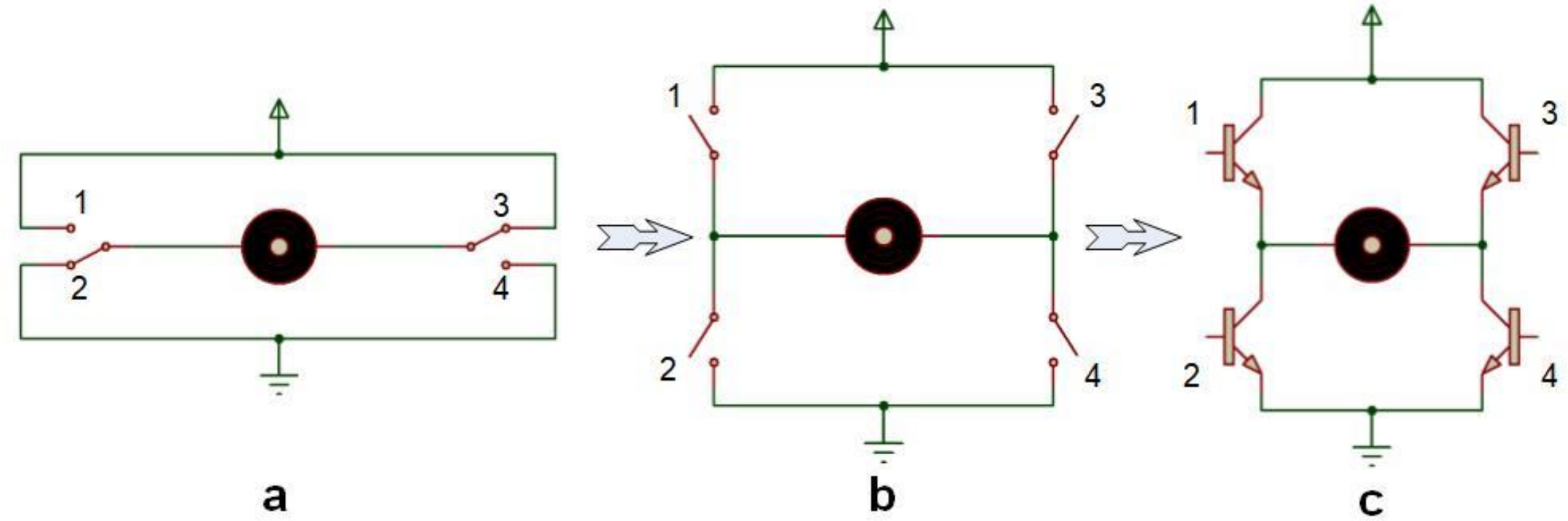
Lecture

Lecture 20 / LO4

H-Bridge / Motor drive through USART

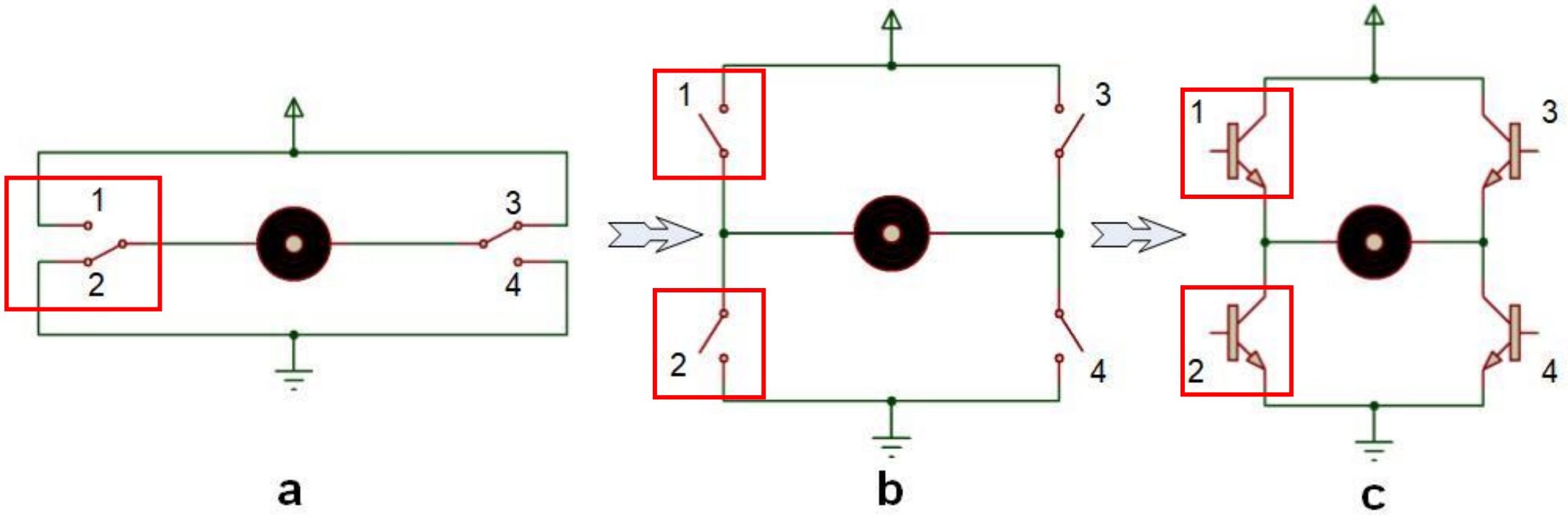
H-Bridge design

- The DC motor inverts its direction when the polarity of the supply voltage is inverted.
- This will invert the flux direction of the coil that consist the motor.
- The switches in circuit a are used to switch the polarity across the motor.



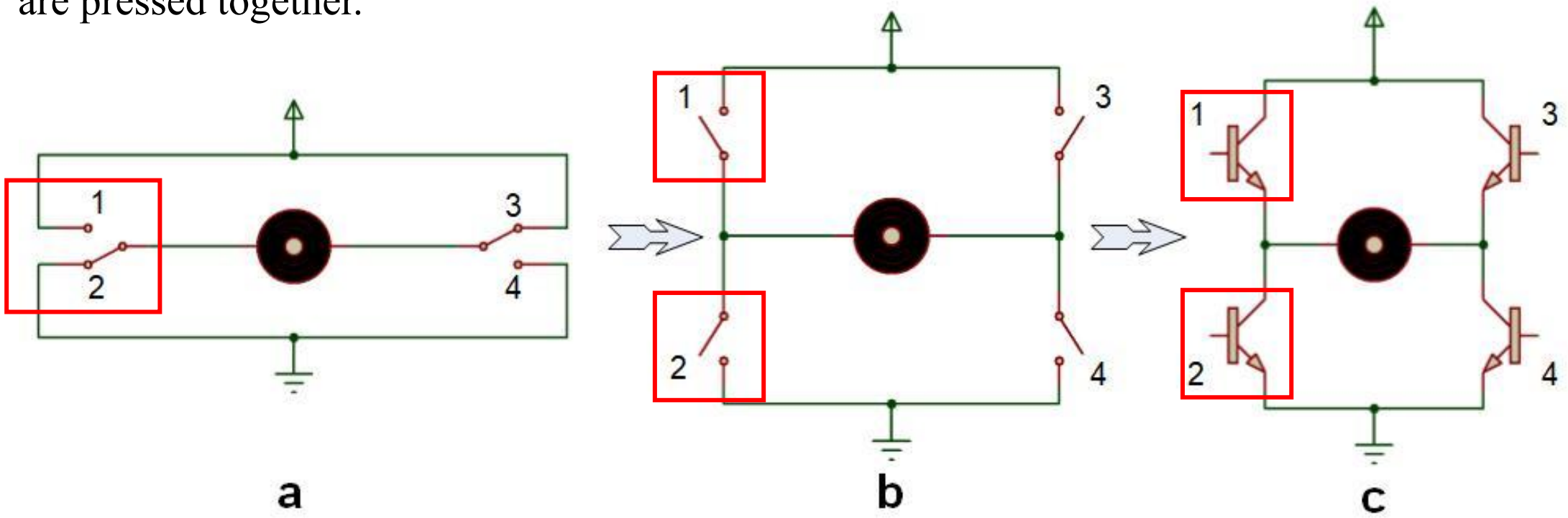
Brushed DC motor driver design (in different directions and speed)

- In this application, the DC motor is driven in double directions with different speed.
- Different speed means a PWM signal and different direction means switches that switch polarity on the motor armature.
- In fact, when merging these two techniques together we can reach what is called the H-bridge.
- We will be using 4 transistor to design an H-Bridge so we can flip the motor polarity and switch it ON and OFF at high speed through a PWM signal



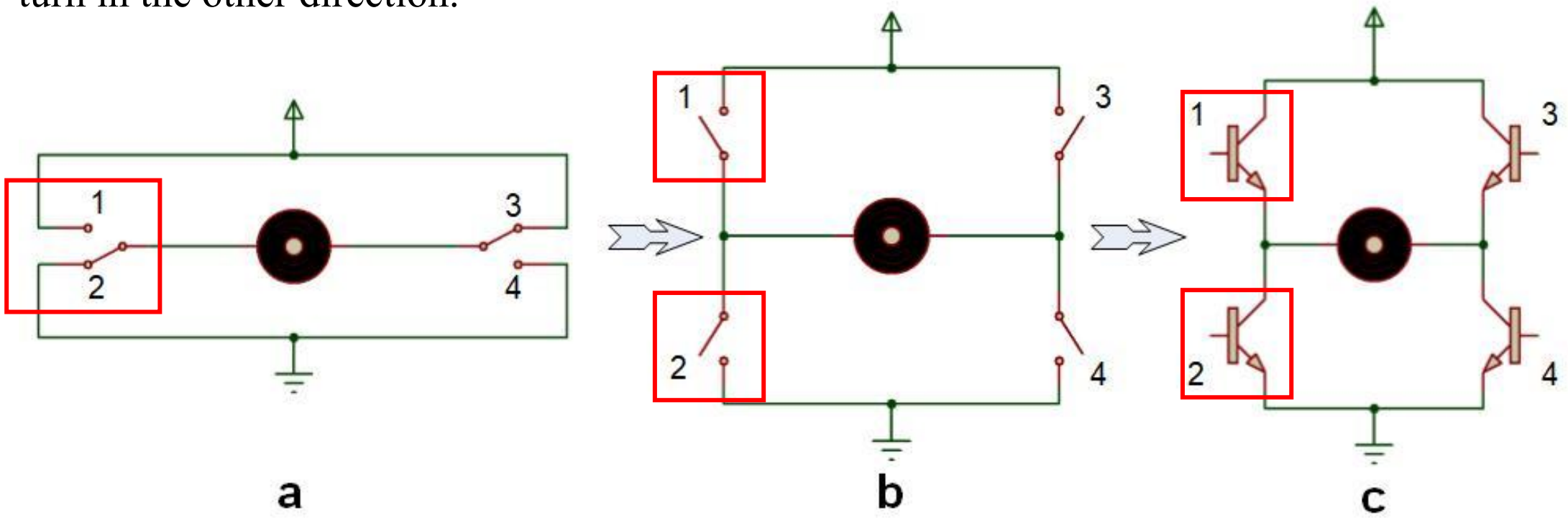
Brushed DC motor driver design (in different directions and speed)

- The Figure “a” is what we have started in switching motor polarity. We have divided each double position switch into two simple switches in Figure “b”.
- If the switches 1and 4 are pressed, the motor will turn, and if 2 and 3 are pressed the motor will turn in the other direction.
- Note that if 1 and 2 are pressed at the same time, a short circuit happened as well if 3 and 4 are pressed together.



Brushed DC motor driver design (in different directions and speed)

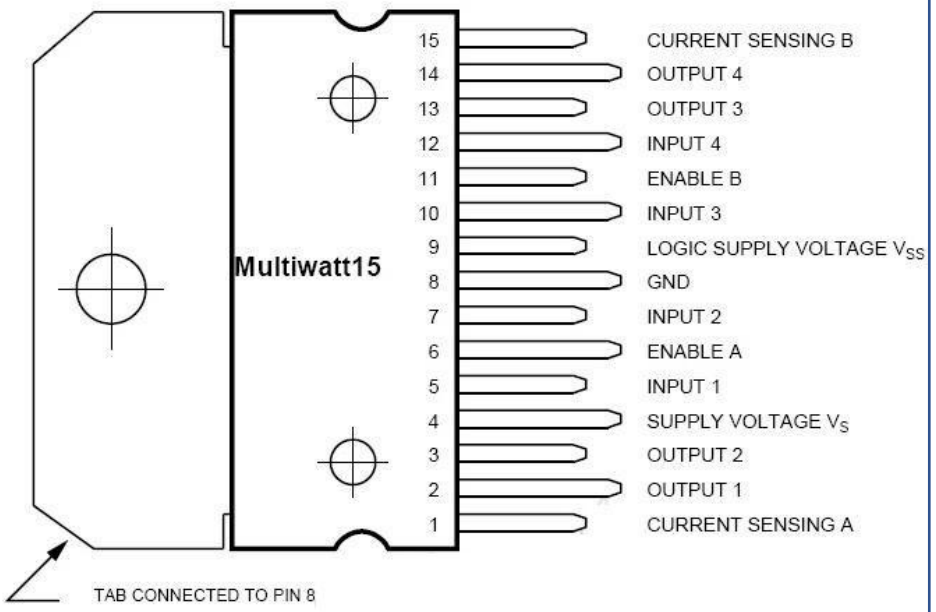
- In the Figure “c”, transistors replace the switches to get the H-bridge that we need to drive our motor in this application.
- By connecting the transistor bases of 1 and 4 and apply a PWM signal, the motor will turn at low speed.
- Also by connecting the transistor base of 2 and 3 and apply the PWM signal, the motor will turn in the other direction.



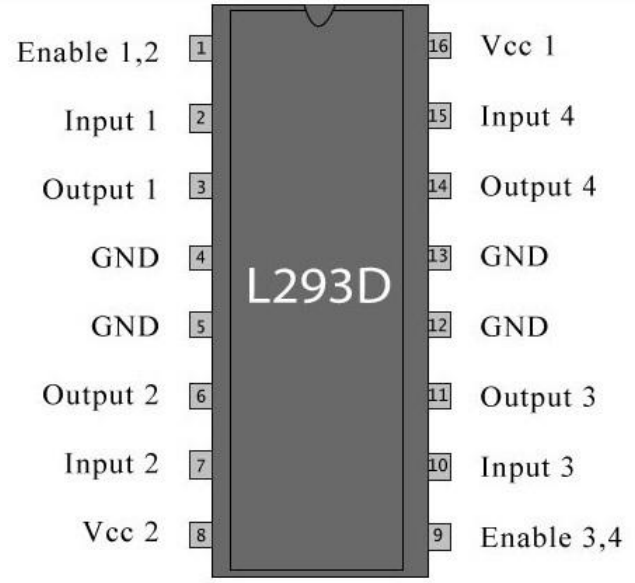
Brushed DC motor driver design (in different directions and speed)

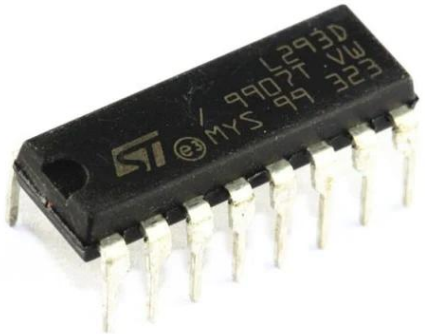
The L298 and L293 are two ships that contain two H-bridges that support 4A and 1A, respectively.
The L298 main feature is the internal protection against short circuit and overheating.





L298





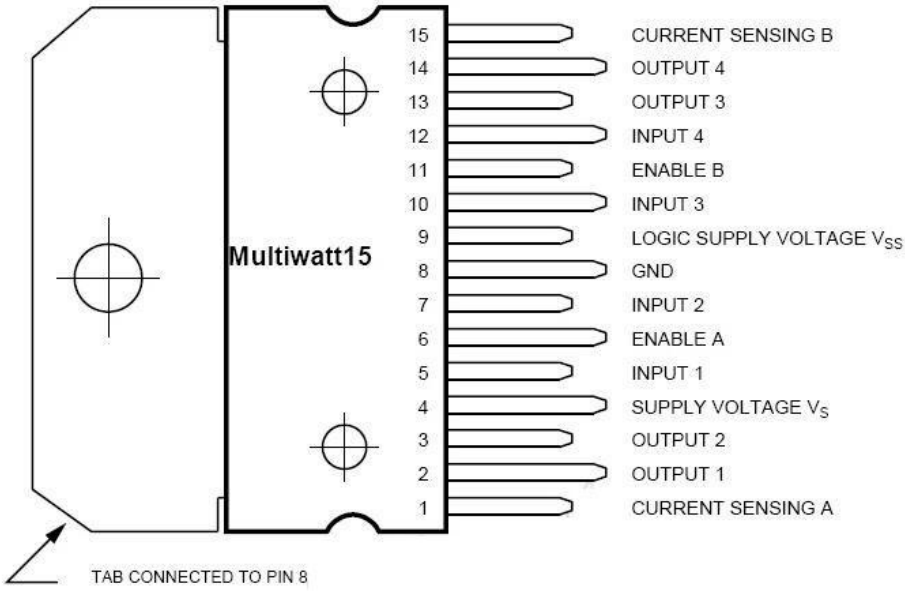
L293

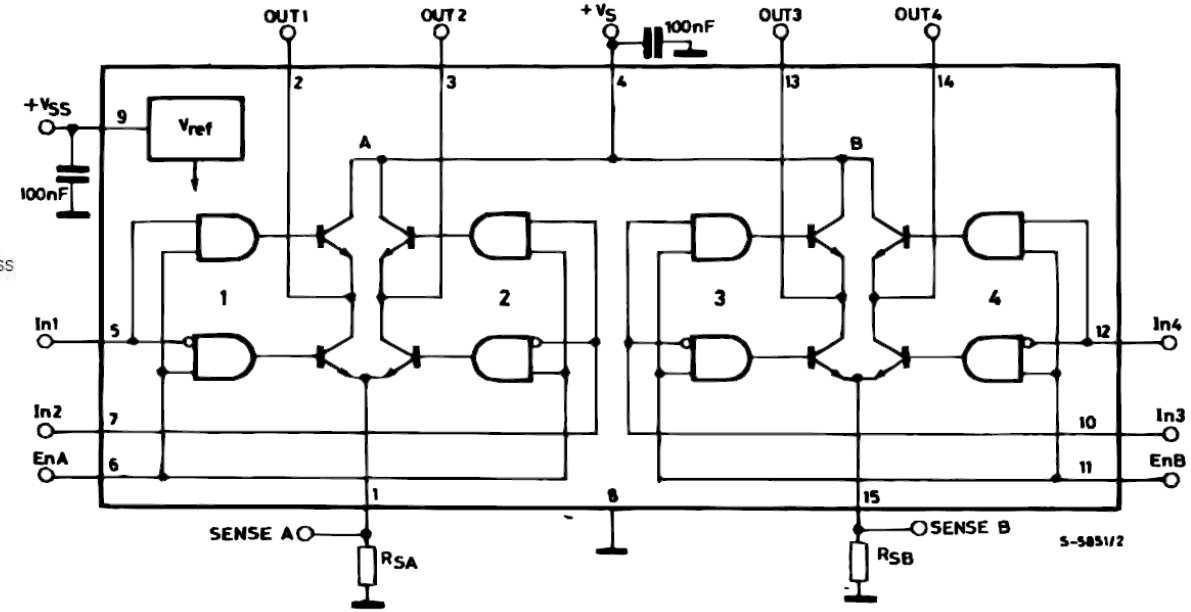
Brushed DC motor driver design (in different directions and speed)

The L298 can be used to drive two DC motors at the same time (It has two H-bridge circuits)



L298

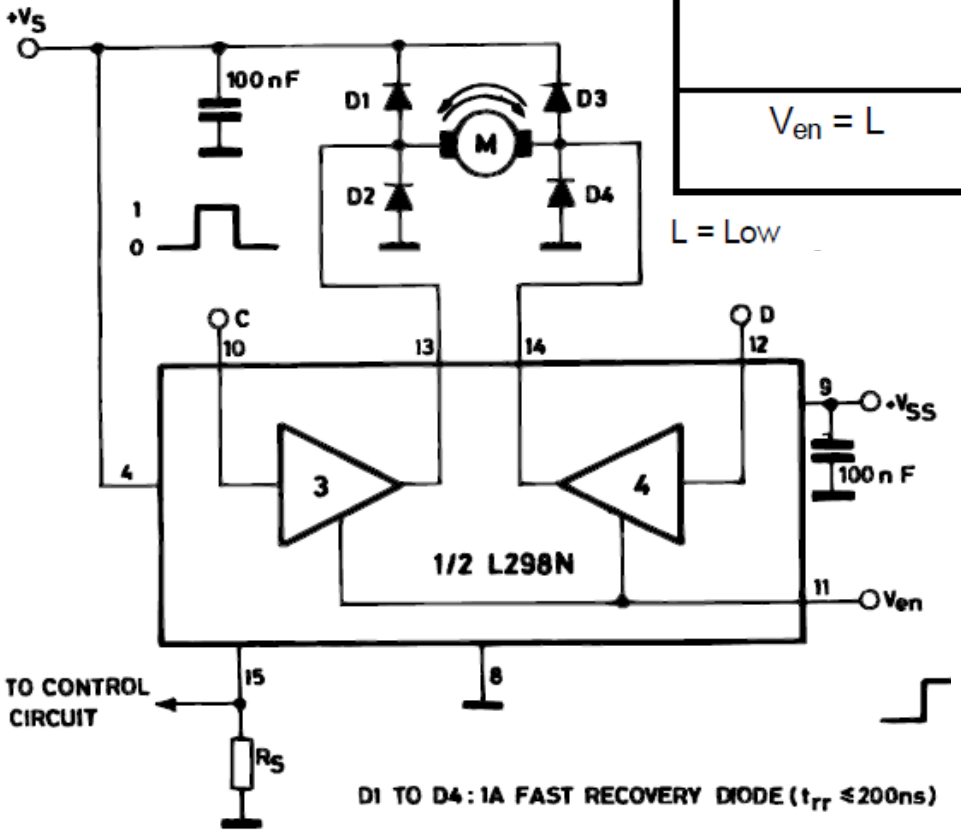


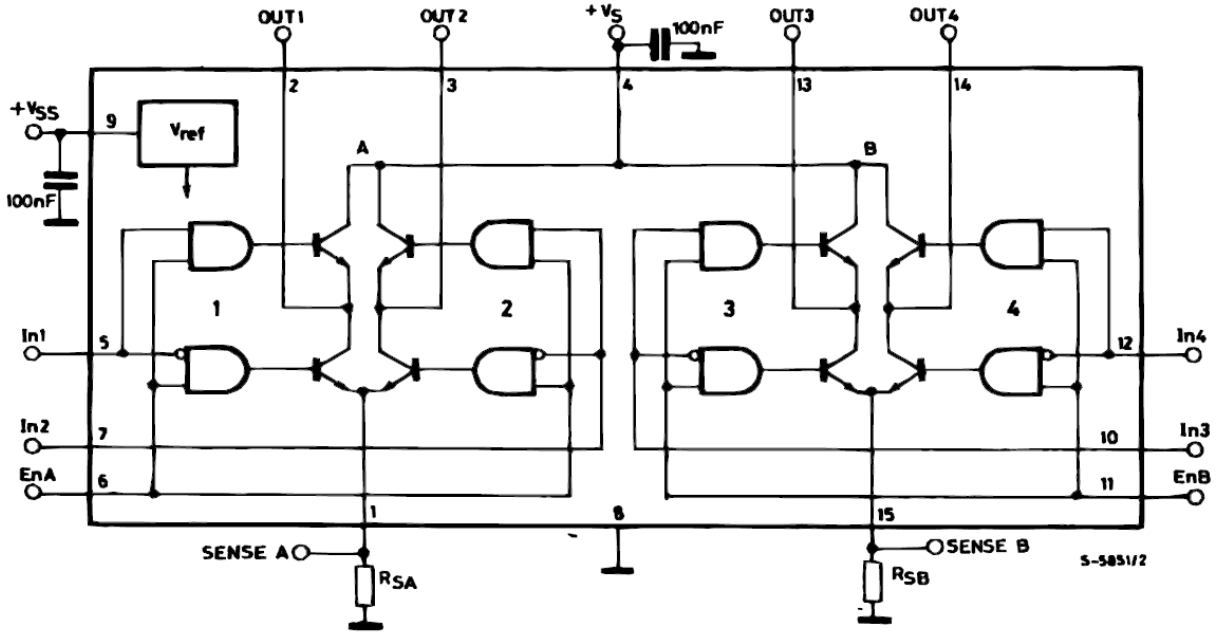


Brushed DC motor driver design (in different directions and speed)

Inputs		Function
V_{en} = H	C = H ; D = L	Forward
	C = L ; D = H	Reverse
	C = D	Fast Motor Stop
V_{en} = L	C = X ; D = X	Free Running Motor Stop

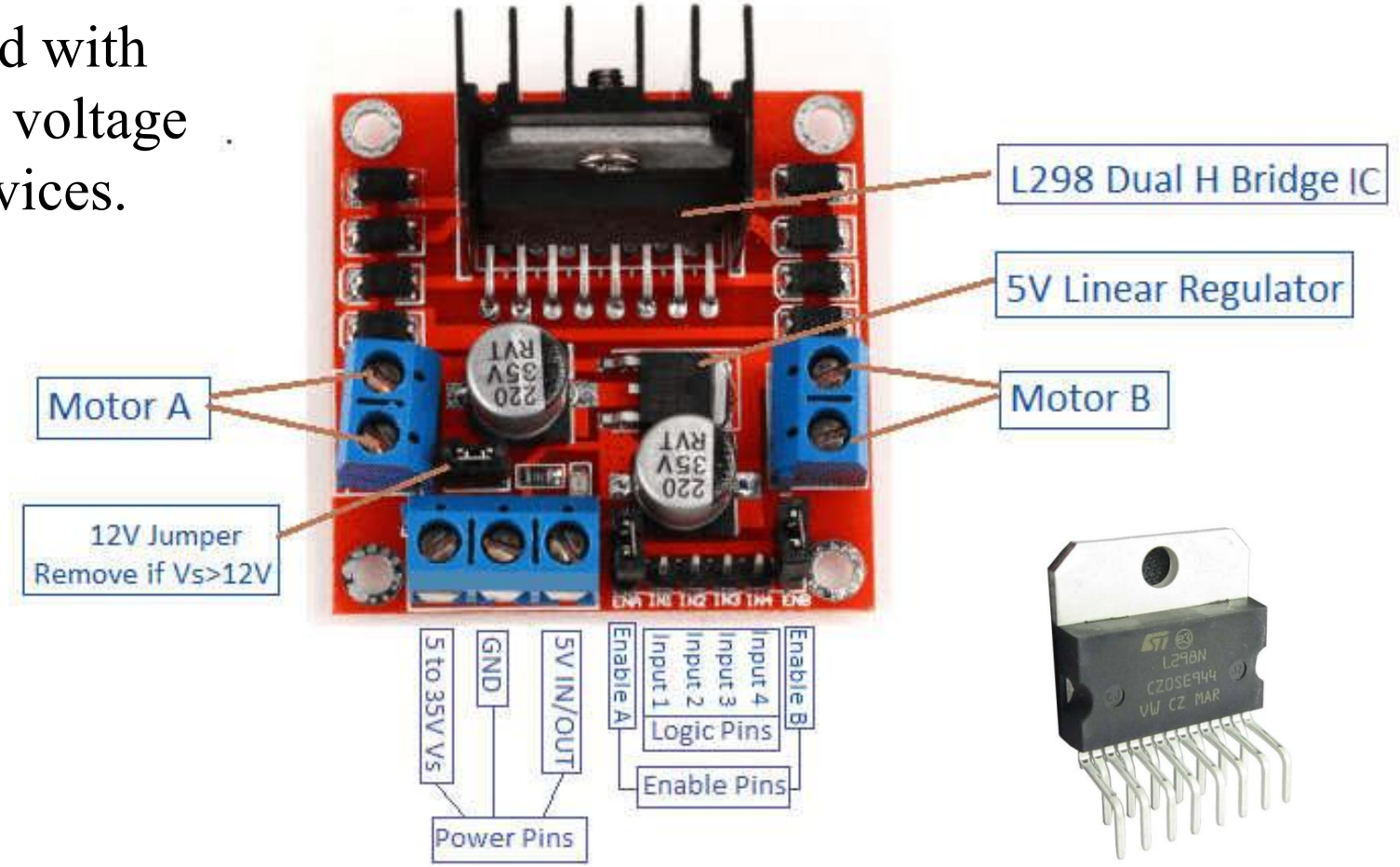
L = Low H = High X = Don't care





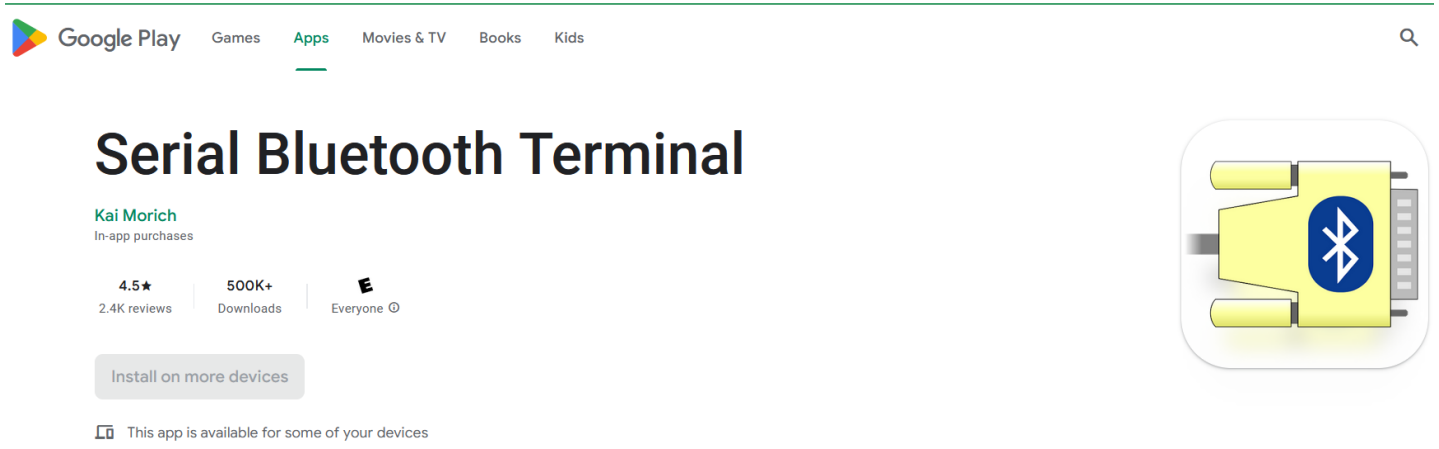
Brushed DC motor driver design (in different directions and speed)

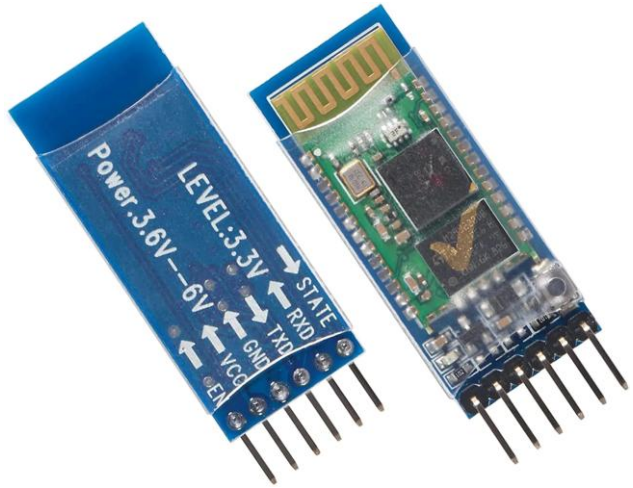
- The L298 module is used to reduce the implementation time and difficulties.
- The module is already equipped with freewheeling diodes and 5 volt voltage regulator to supply external devices.



Activity 20C: Motor drive through Bluetooth

- In this activity, we will drive a motor using the L298 chip through Bluetooth command received from cell phone.
- The Bluetooth transceiver to be used is the HC05 (Master and slave device) or the HC06 (slave device). Both work well with PIC and other microcontrollers.
- Serial Bluetooth terminal is a good Android app to send/receive raw data

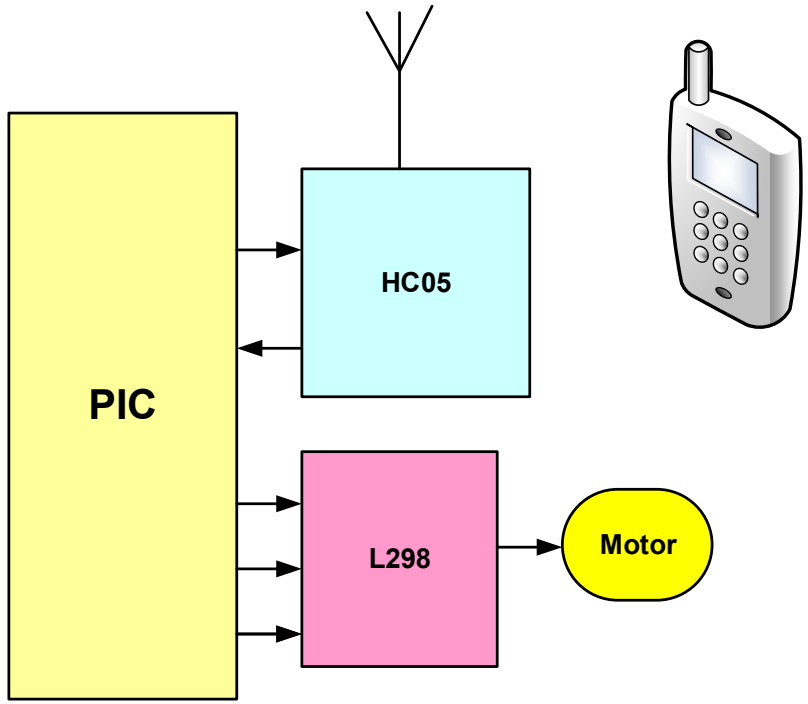


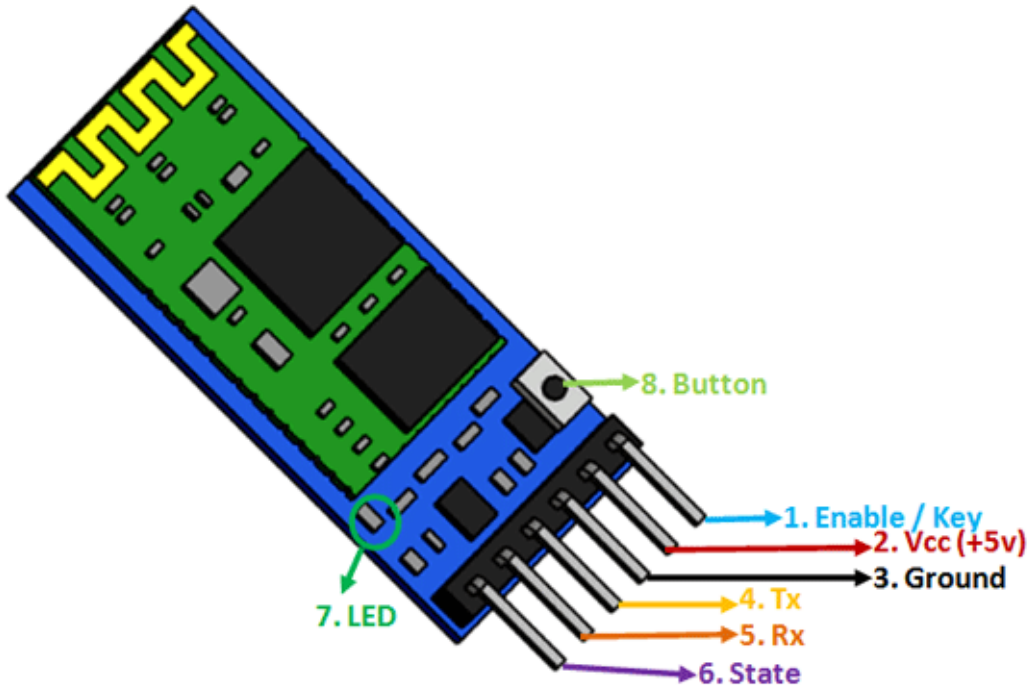


Activity 20C: Motor drive through Bluetooth

- The block diagram of the design is shown. The PIC is connected to the TX and RX of the HC05 (pins 4 and 5). Enable and state pins are not used.

- At the same time, the PIC is connected to the L298 (chip or module) through 3 lines, IN1, IN2 (for direction) and Enable (for speed).



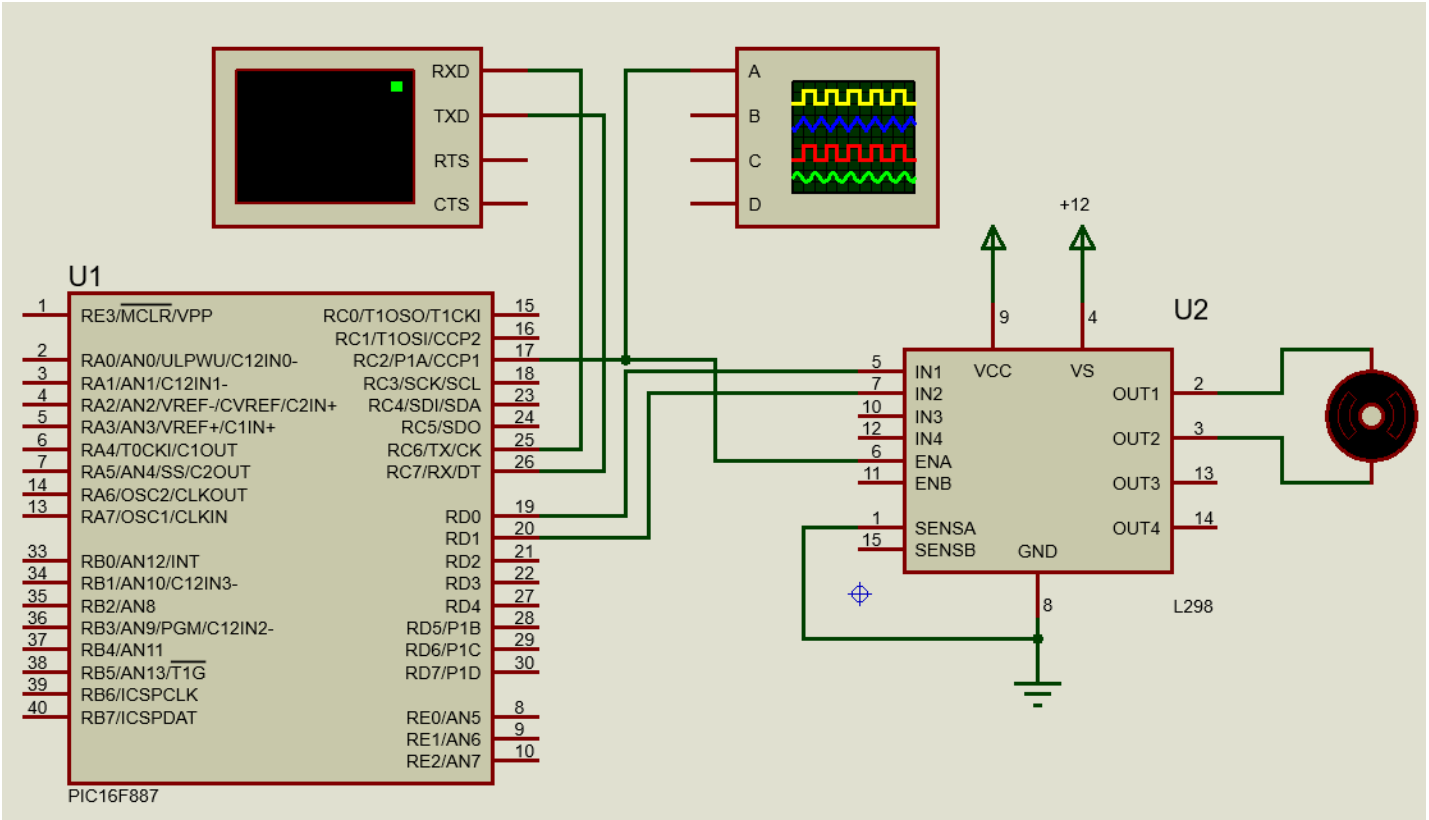


Activity 20C: Motor drive through Bluetooth

- The Bluetooth is replaced by the serial monitor in the simulation.
- The format of data to be transferred to the PIC is XYYY where X is the direction and YYY is the speed
- X can be:
  - L for left
  - R for right
  - S for stop

- YYY can be any number between 0 and 255

- e.g. L255 (Left full speed)
- R127 (Right half speed)
- SYYY(Stop)



Activity 20C: Motor drive through Bluetooth

- The main function simply calls the setup function and send instruction to the user.
- The setup function configures the oscillator, the port direction, the USART and the PWM.
- The interrupt on USART receive is enabled

```
64  void setup() {
65      PORTC = 0;
66      PORTD = 0; // clear port
67      OSCCON = 0b01100000; // select 4MHz oscillator
68      TRISC = 0b10000000; // and as out
69      TRISD = 0b00000000;
70      TXSTA = 0b00100100; // asyn mode
71      RCSTA = 0b10010000; // serial port enables with continious receive
72      SPBRG = 25; // 9600 bit /s
73
74      PR2 = 255; // define the period
75      T2CON = 0b00000111; // prescaler 16, timer 2 on
76      CCP1CON = 0b00001100; // CCP1 in PWM mode
77
78      GIE = 1; // enable int
79      PEIE = 1;
80      RCIE = 1;
81  }
82
83  void main(void) {
84      setup();
85      USART_SEND((char*) "Enter direction and speed");
86      USART_SEND_NEW_LINE();
87      while (1);
88  }
```

Activity 20C: Motor drive through Bluetooth

- The motor function is created to take 2 parameters; the direction and speed.
- The direction can be L for left, R for right and S for stop.
- The speed can be ay value between 0 and 255 as the parameter is loaded onto the CCPR1L

```
36  void motor(char direction, char speed) {
37      CCPR1L = speed;
38      if (direction == 'L') {
39          IN1 = 1;
40          IN2 = 0;
41      }
42      if (direction == 'R') {
43          IN1 = 0;
44          IN2 = 1;
45      }
46      if (direction == 'S') {
47          IN1 = 0;
48          IN2 = 0;
49      }
50  }
```

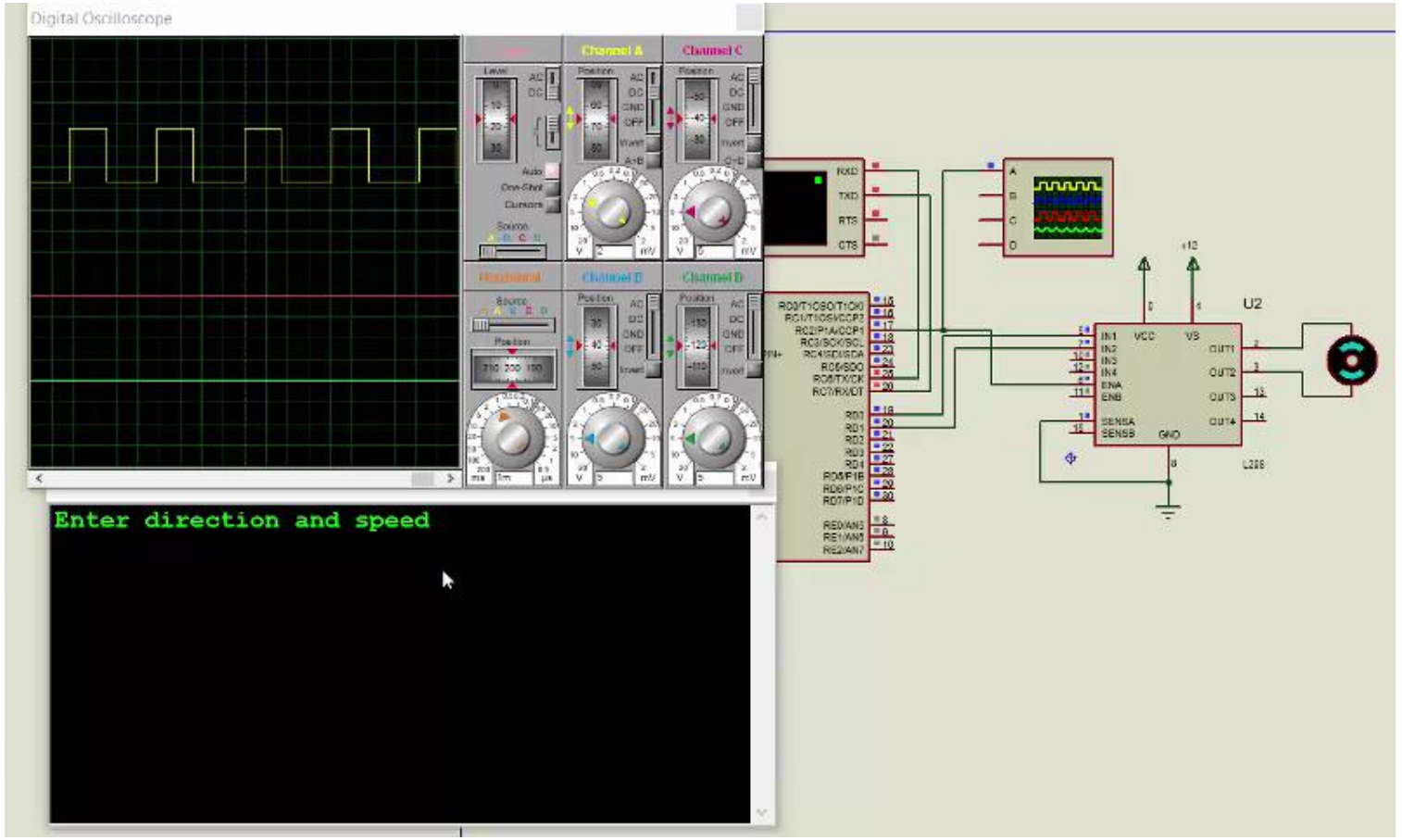
Activity 20C: Motor drive through Bluetooth

- When a value is received through Bluetooth, the interrupt fct is called. The received value is saved in “phrase”
- The phrase[0] represent the direction.
- Phrase[1], [2] and [3] contains the speed. These values are considered as characters and should be converted into one value, i.e., ‘1’ ‘2’ ‘5’ are 3 characters that represents 125.
- Thus to make the conversion, 48 is subtracted from the char and multiplied by its weight.

```
52  void __interrupt() my_isr_routine() {
53      phrase[pointer] = RCREG; // once a character is received store it in phrase
54      pointer++; // with index pointer}
55      if (RCREG == '\n' || RCREG == '\r') {
56          USART_SEND((char*) "Sent msg was: ");
57          USART_SEND((char*) phrase);
58          pointer = 0;
59          int speed = (phrase[1]-48)*100 + (phrase[2]-48)*10 + (phrase[3]-48);
60          motor(phrase[0], speed);
61      }
62  }
```

Activity 20C: Motor drive through Bluetooth

- The table shows the equivalent of character numbers in decimal, hex, octal and binary.



Dec	Hex	Oct	Binary	Char
48	30	60	00110000	0
49	31	61	00110001	1
50	32	62	00110010	2
51	33	63	00110011	3
52	34	64	00110100	4
53	35	65	00110101	5
54	36	66	00110110	6
55	37	67	00110111	7
56	38	70	00111000	8
57	39	71	00111001	9



Week 11

Lab 9 / LO4

Motor drive with Bluetooth



Activity 20C : Motor drive with Bluetooth

- Tools Required
  1. Breadboard x1
  2. Multimeter x 1
  3. PIC programmer
  4. Power supply 5V, 12V
  5. Breadboard wires (male-male)
  6. Oscilloscope

- Required components:
  1. PIC 16F887 x1
  2. L298 module x1
  3. DC motor x1
  4. HC05 or HC06 module x1

End of LO4
End of the course