WELCOME TO CLASS!

let’s introduce ourselves

what is a “Microprocessor”?

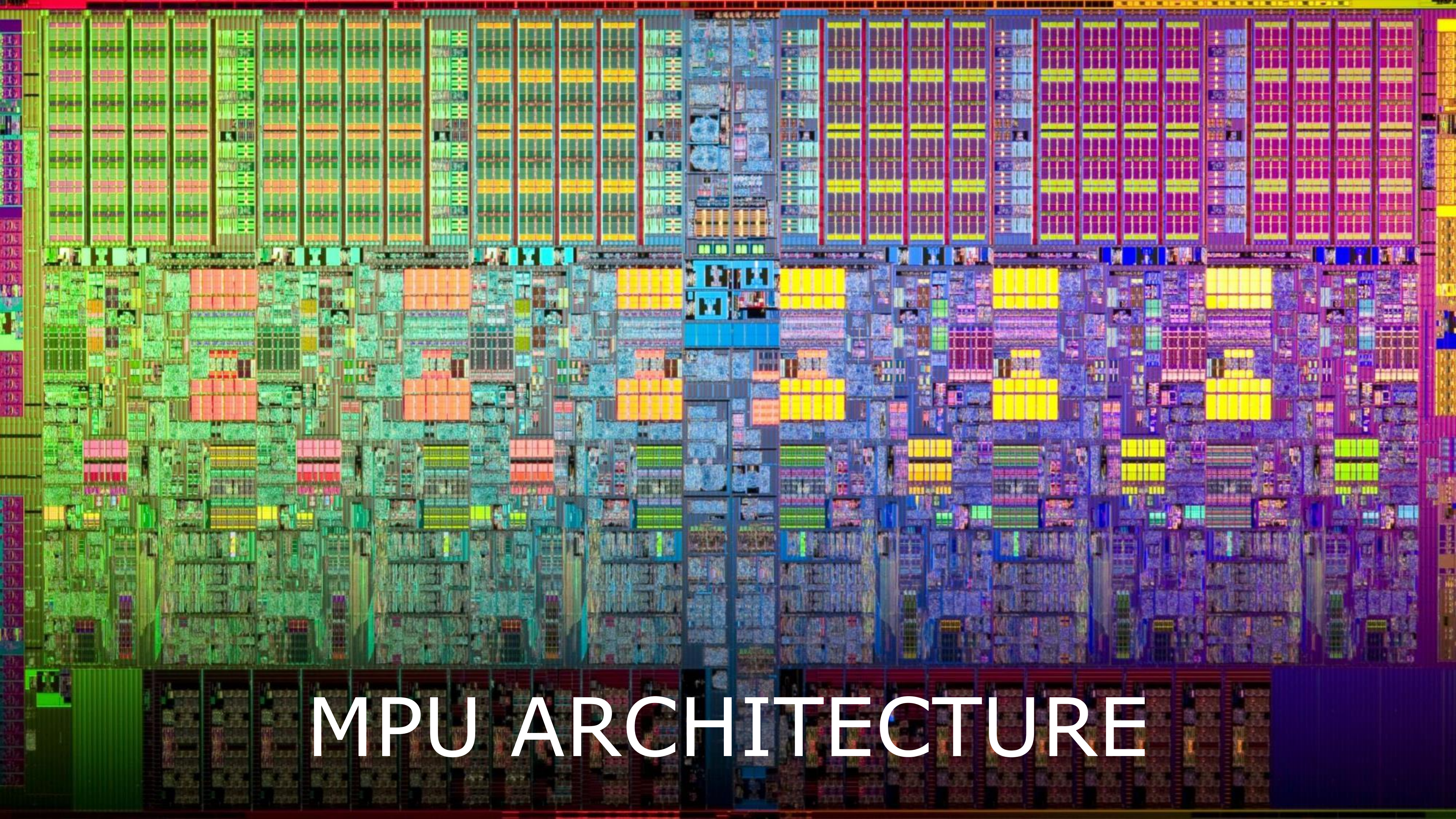
CLASS 1

what is a microprocessor?

The Microprocessor, also known as the Central Processing Unit (CPU), is the brain of all computers and many household and electronic devices

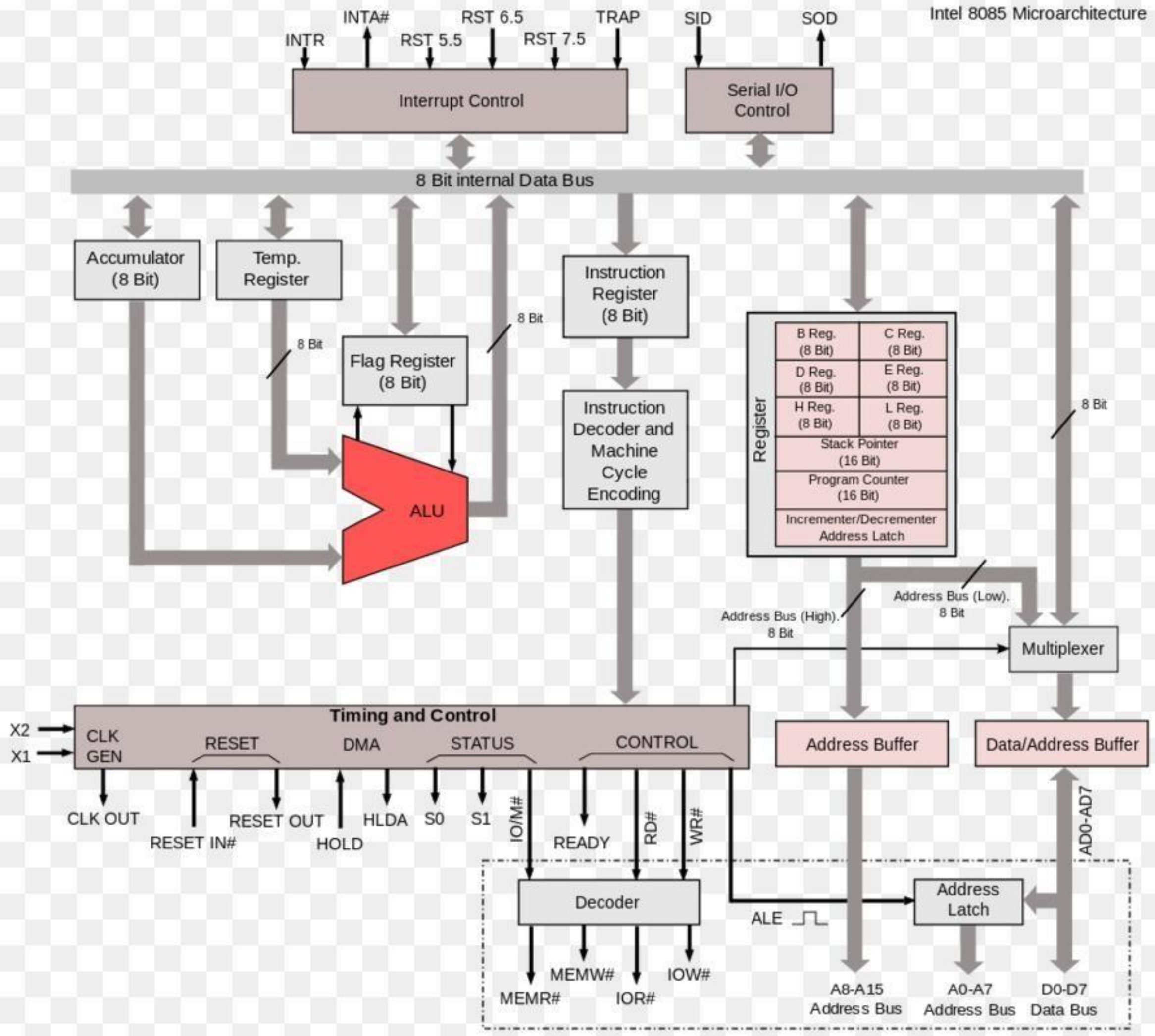
What is a microprocessor?

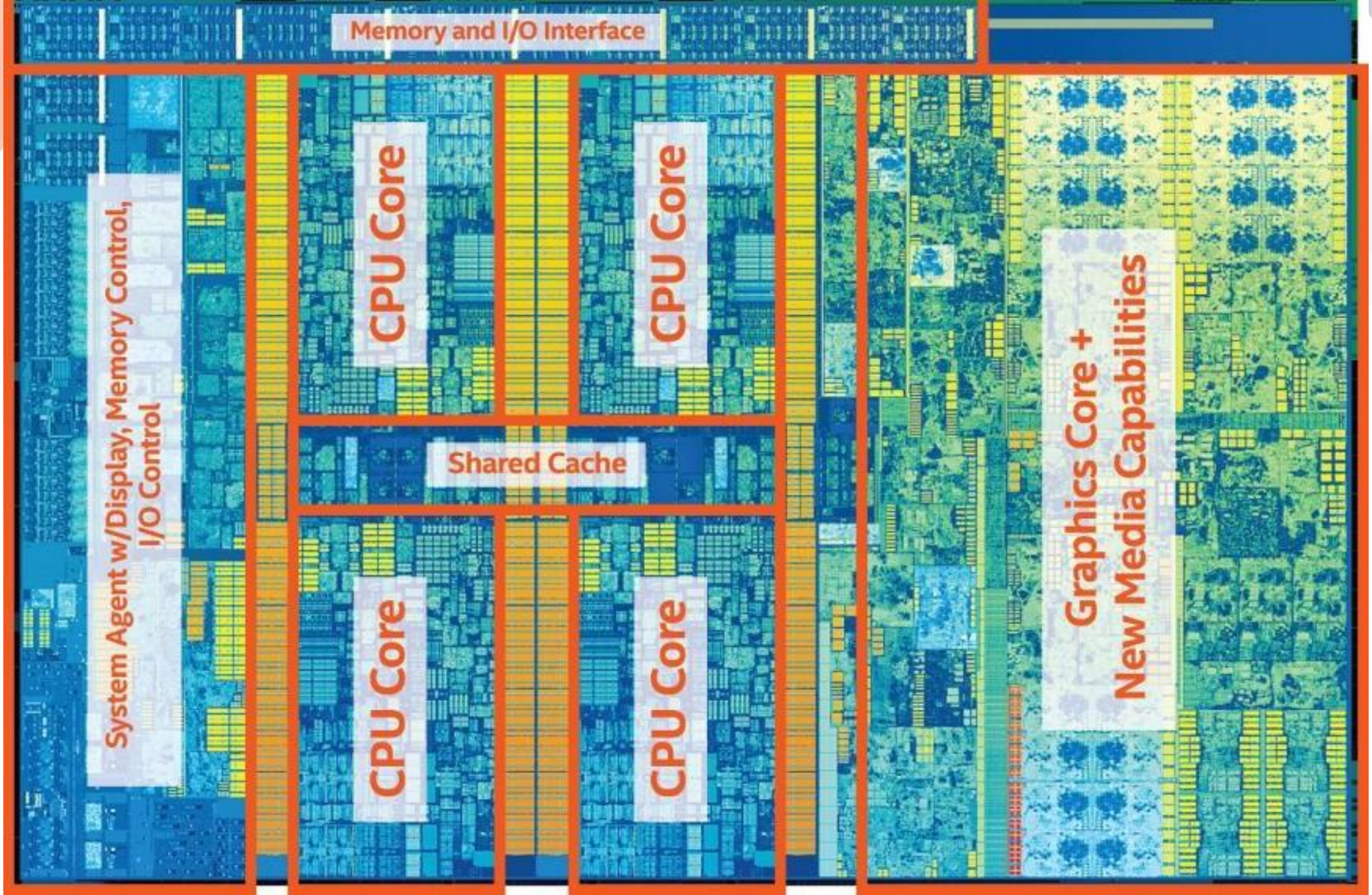
Multiple MPUs, working together, are the "hearts" of datacenters, super-computers, communications products, and other digital devices

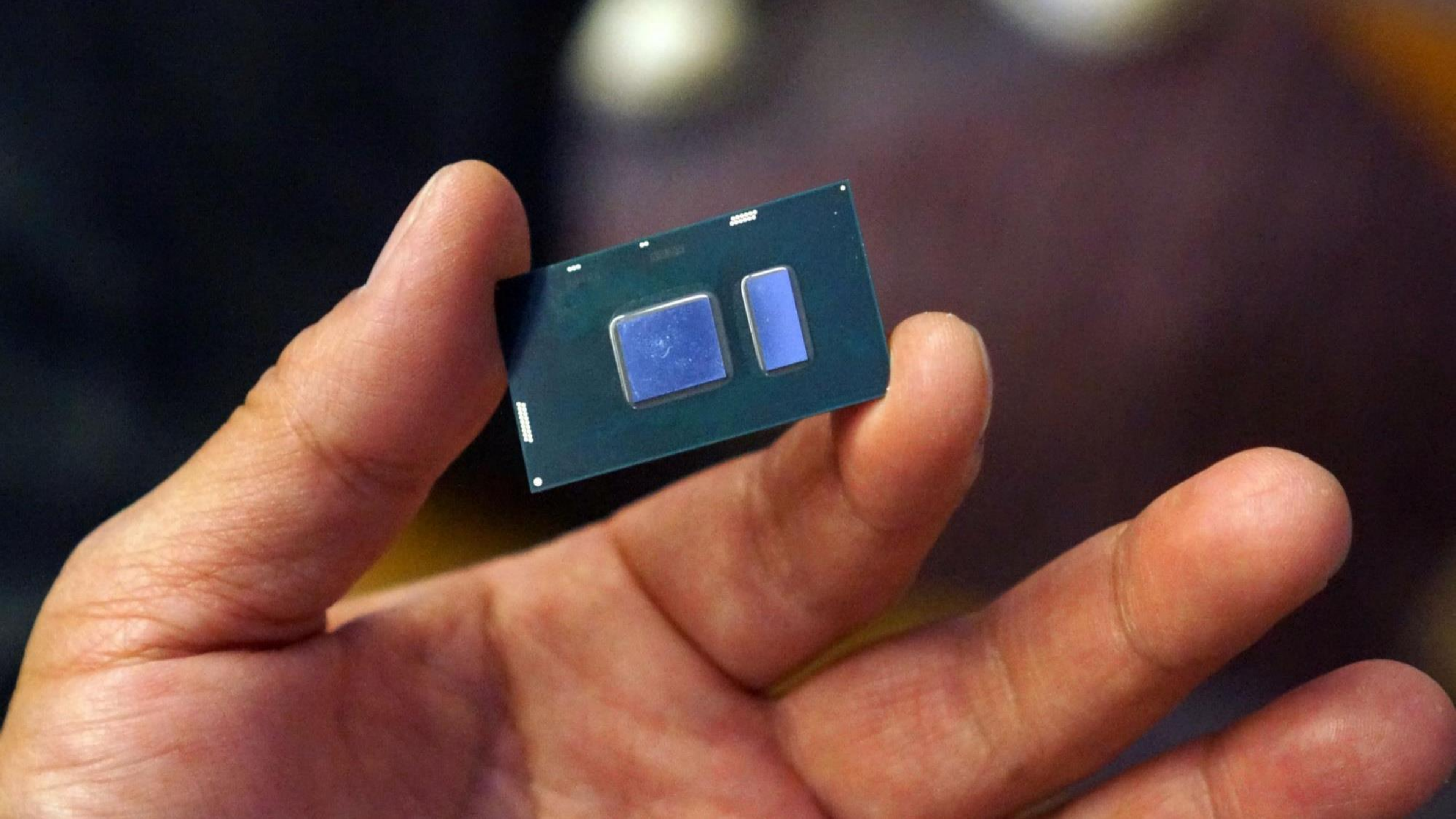


MPU ARCHITECTURE

INTEL 8085 MPU ARCHITECTURE

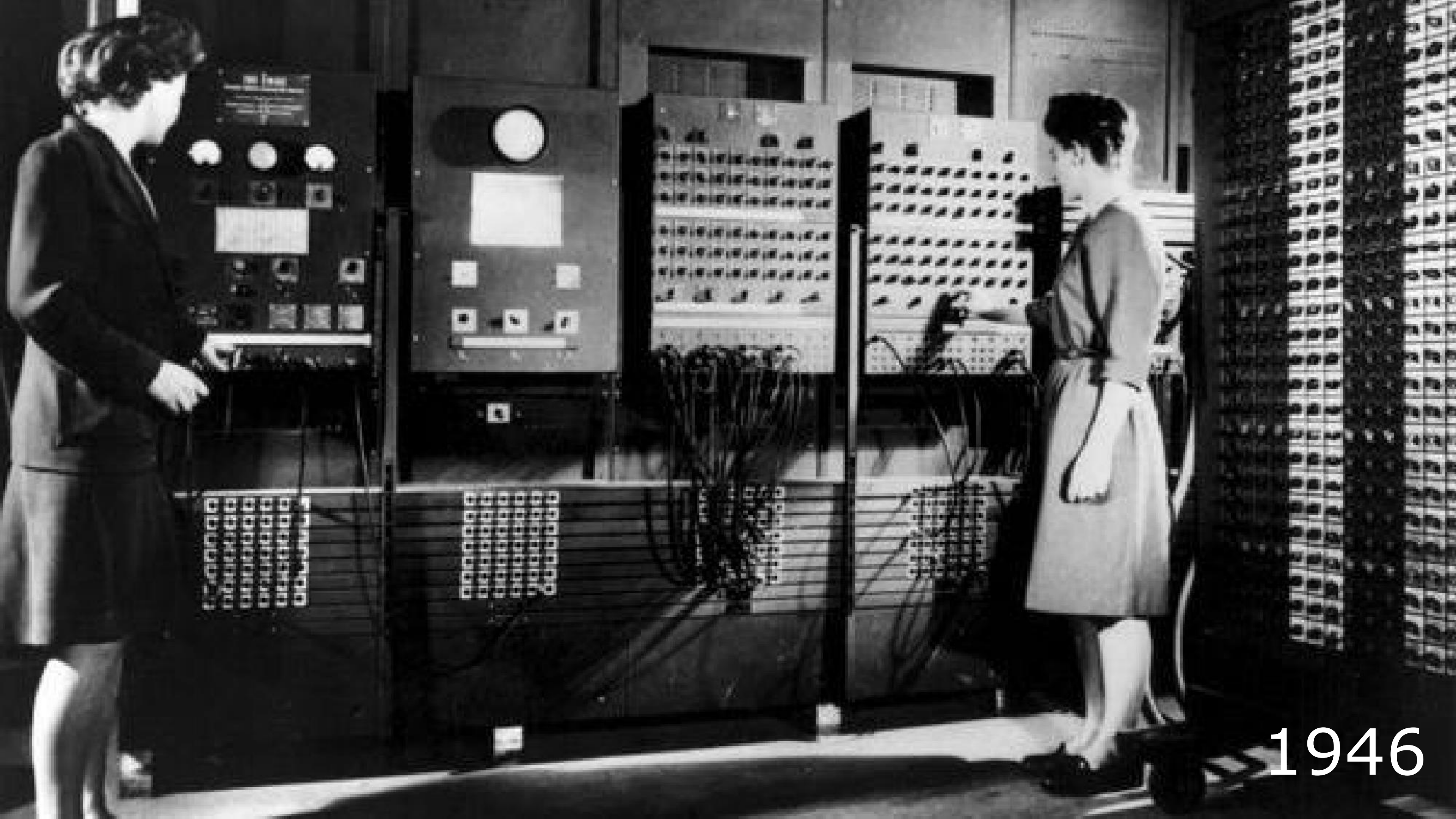



Memory and I/O Interface
System Agent w/Display, Memory Control, I/O Control
CPU Core
CPU Core
Shared Cache
CPU Core
CPU Core
Graphics Core + New Media Capabilities



HISTORY

ENIAC (Electronic Numerical Integrator And Computer) was the world's first general-purpose computer



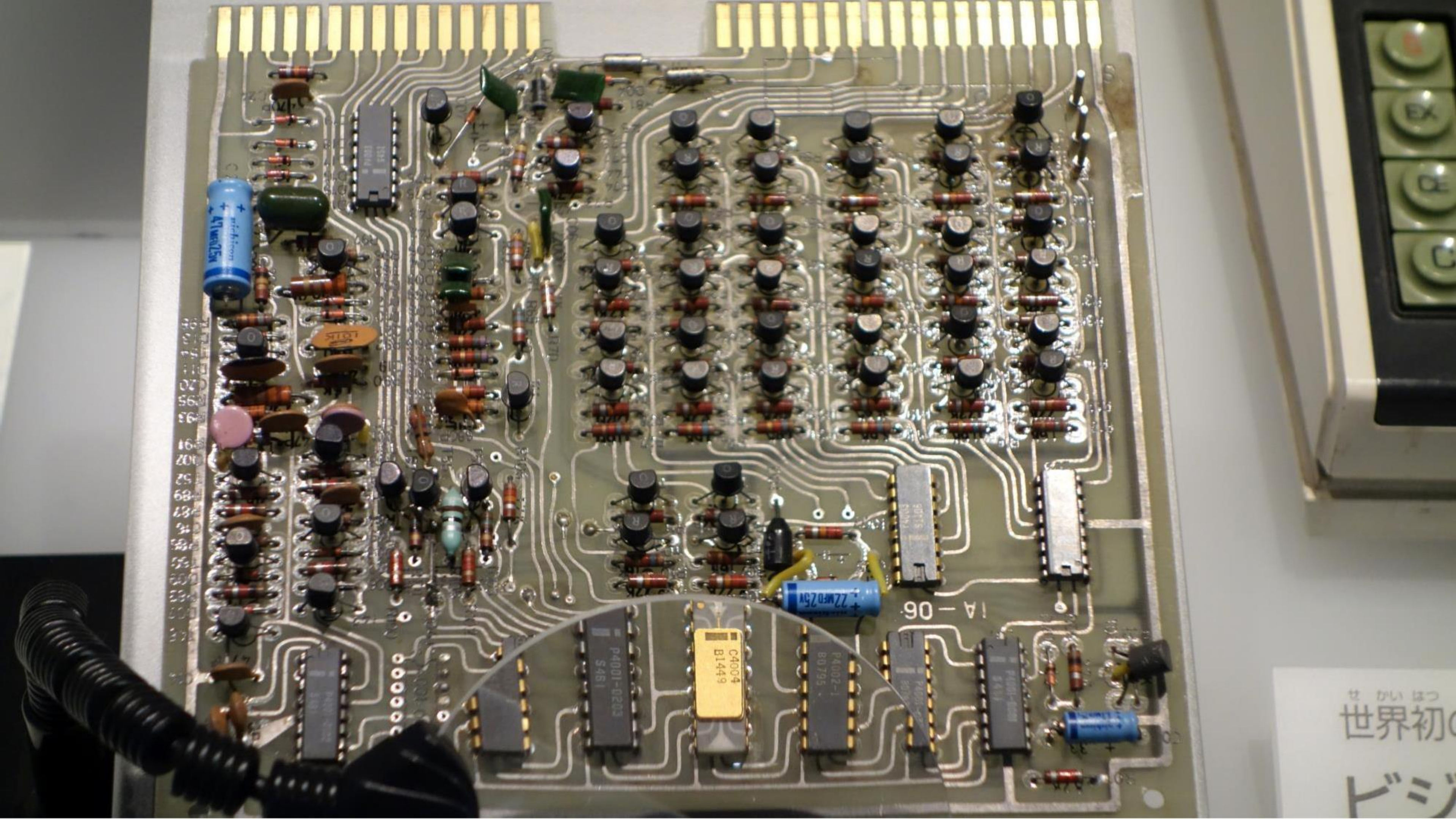
1946

HISTORY

The first microprocessor was the Intel 4004, introduced in 1971

INTEL 4004

- Introduced Nov., 1971 by Intel
- 2250 transistors
- 108 kHz, 60,000 ops/sec
- 16 pins DIP (Dual in-line package)
- 10-micron process
- Targeted use: Calculators
- **Instruction set:** 46 instructions (of which 41 were 8 bits wide and 5 were 16 bits wide)
- Cost: less than $100


C4004
B1449
P4001-0203
S461
P4002-1
B0795
IA-06
47μF 25V
nichicon
22μF 25V
EX
CE
C
世界初の
ビジ

INTRODUCING ICE LAKE: 10NM CPU

2019

NEW SUNNYCOVE CORES

Up to 4 Cores / 8 Threads
Up to 4.1GHz

NEW CONVERGED CHASSIS FABRIC

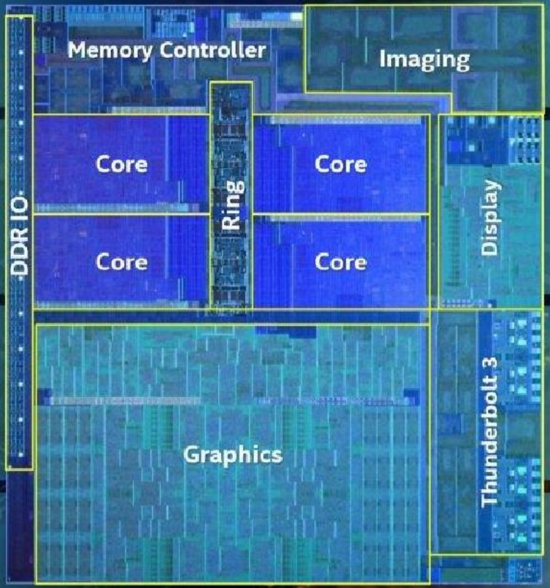
High Bandwidth / Low Latency
IP and Core Scalable

NEW MEMORY CONTROLLER

LP4/x-3733 4x32b up to 32GB
DDR4-3200 2x64b up to 64GB

FIRST INTEGRATED THUNDERBOLT™ 3

Full 4x DP/USB/PCIe mux on-die
Up to 40Gbps bi-directional per port



NEW GEN11 GRAPHICS

Up to 64EU and 1.1GHz
>1TFLOP

NEW 2X MEDIA ENCODERS

Up to 4K60 10b 4:4:4
Up to 8K30 10b 4:2:0

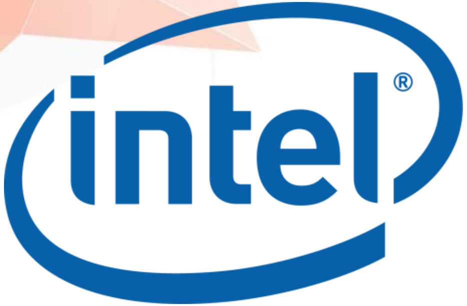
NEW 3X DISPLAY PIPES

Up to 5K60 or 4K120
DP1.4, BT.2020

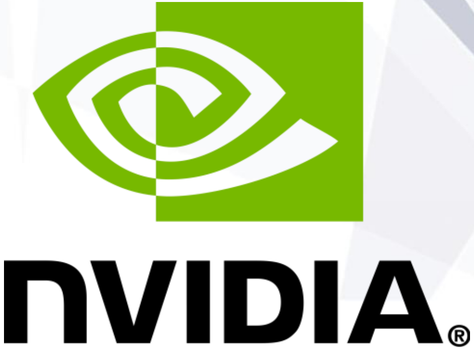
NEW IMAGE PROCESSING UNIT 4

Up to 16MP
Up to 1080p120, 4K30

Intel Core i9-9900KS Processor
18 Billion Transistors


intel®

AMD


NVIDIA®

QUALCOMM®

tsmc
台灣積體電路製造股份有限公司
Taiwan Semiconductor Manufacturing Company, Ltd.

ATMEL

MICROCHIP®

microprocessor vs CPU

A CPU (central processing unit) is the part of a computer that executes instructions. This can be implemented using a single IC, a number of ICs, discrete transistors or a room full of vacuum tubes

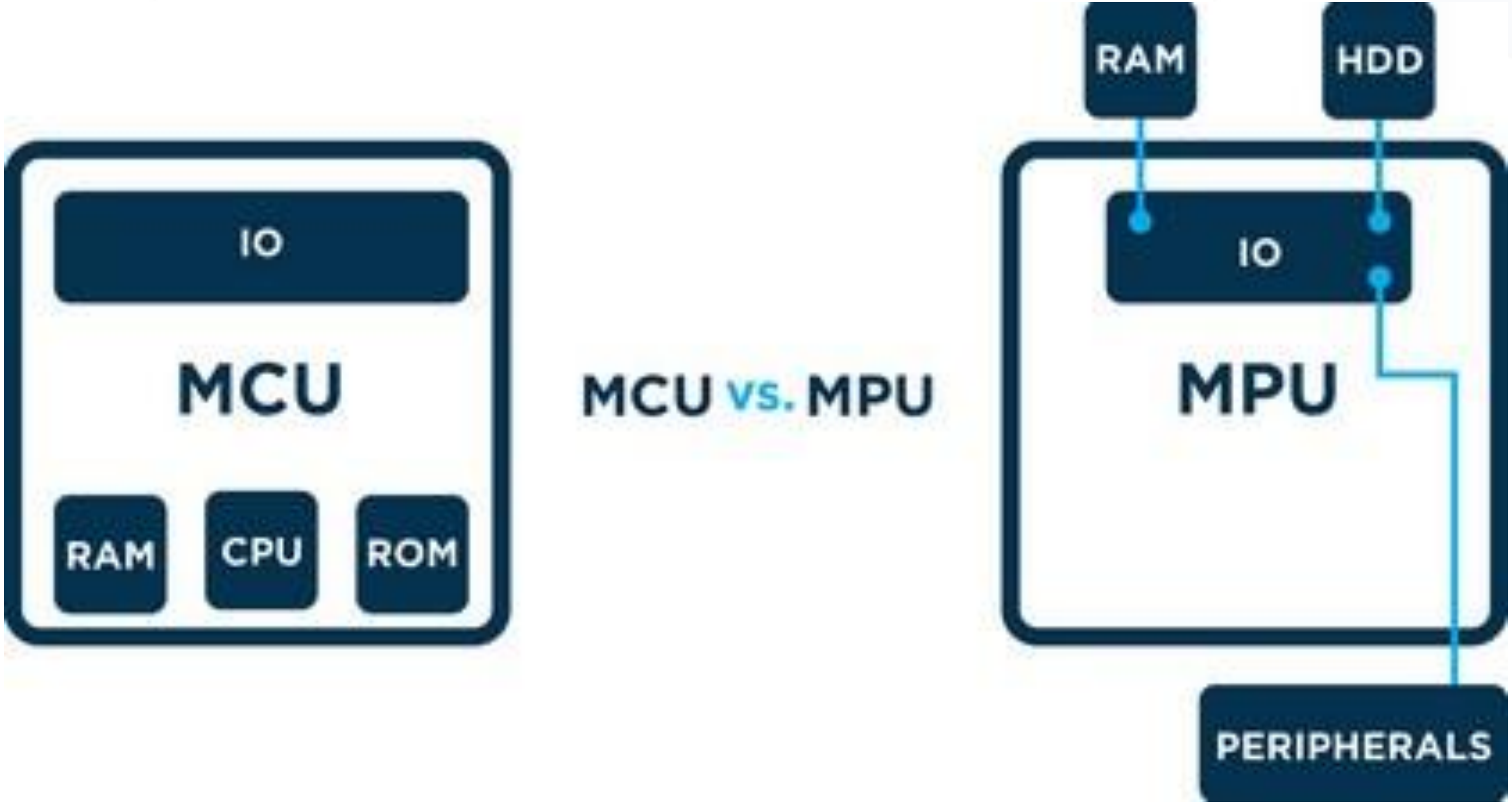
microprocessor vs CPU

A microprocessor is a single-chip implementation of a CPU

microprocessor vs CPU

Nowadays pretty much all CPUs for general use are microprocessors, causing the two terms to be practically synonymous

MPU vs MCU



what is a microcontroller?

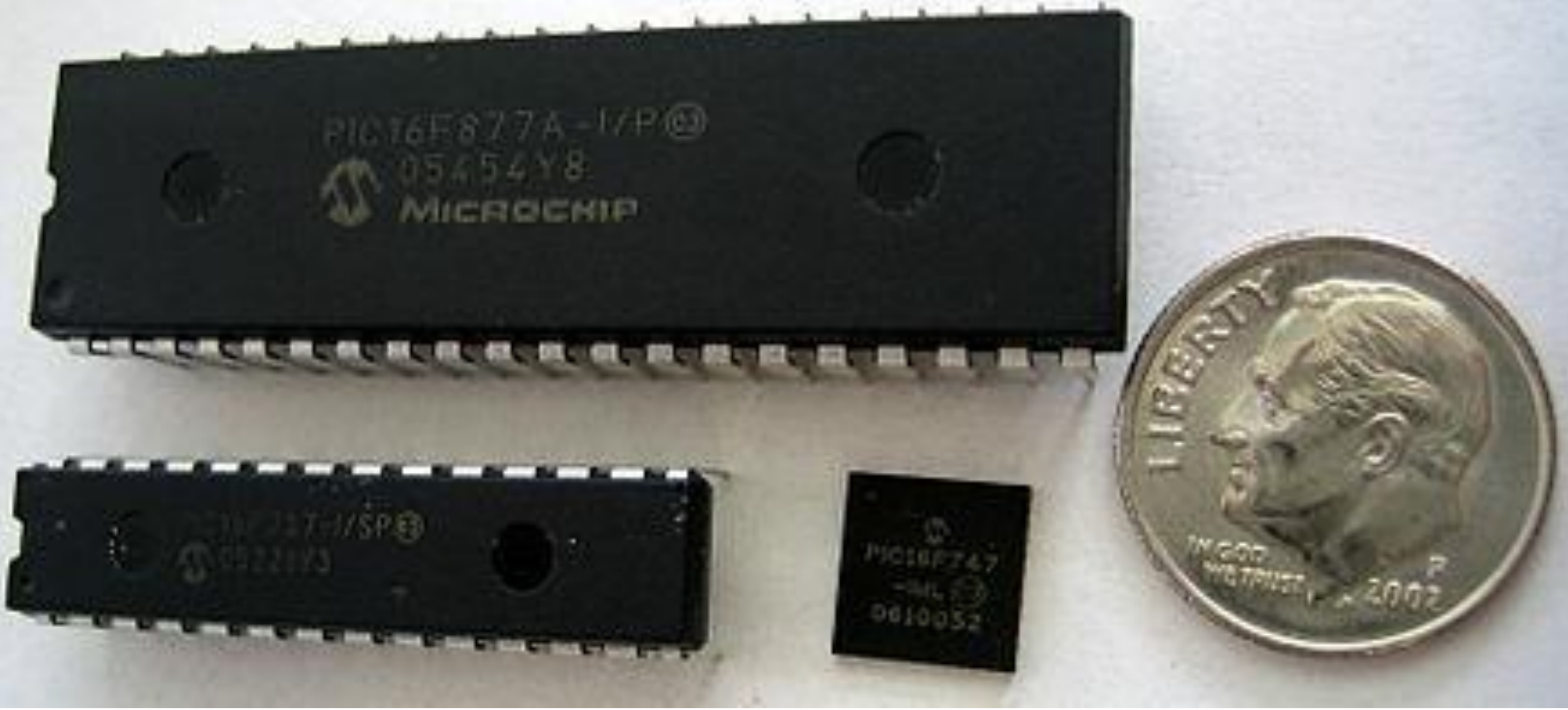
A microcontroller is a computer present in a single integrated circuit which is dedicated to perform one task and execute one specific application

what is a microcontroller?

MCUs contain memory, programmable input/output peripherals as well a processor

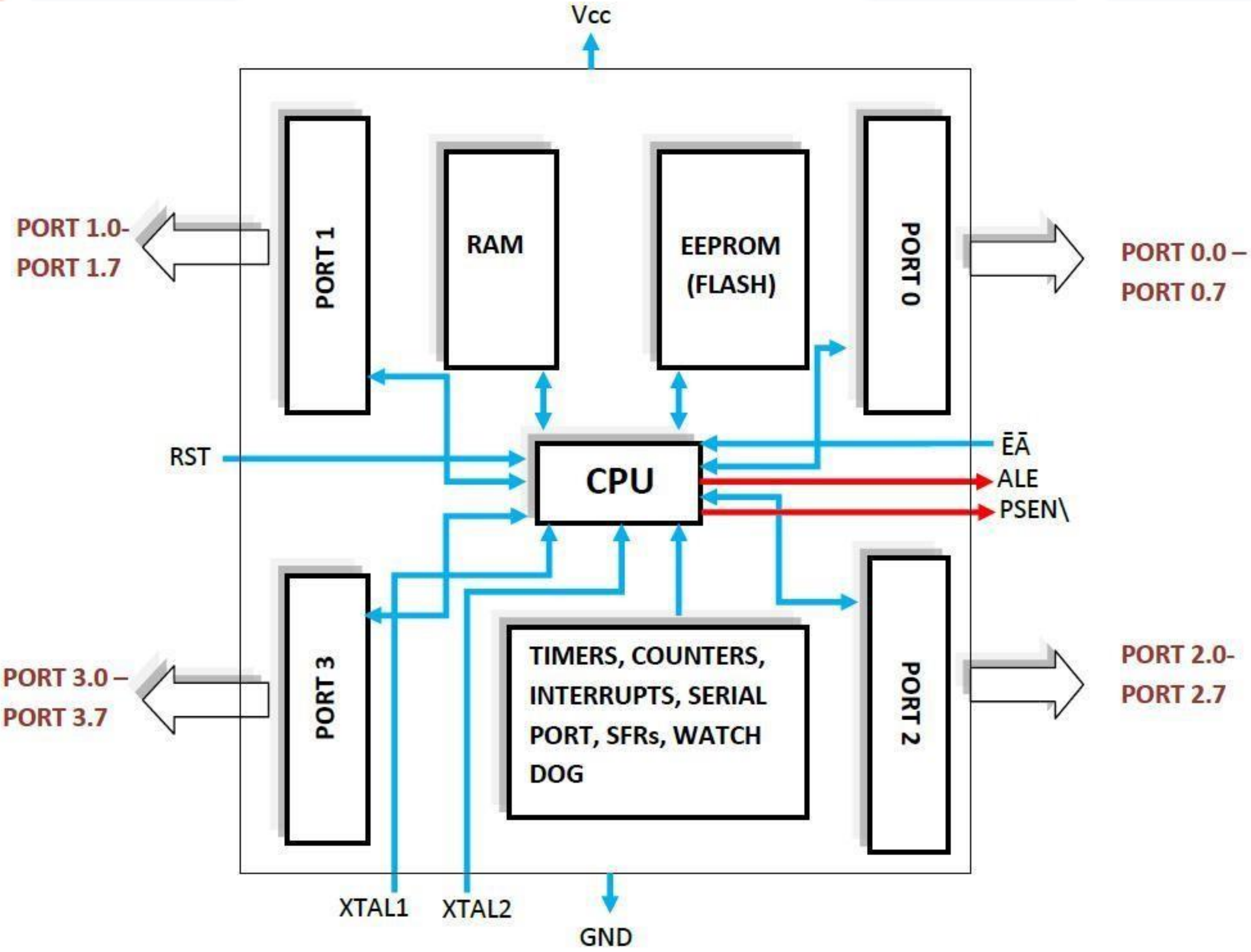
what is a microcontroller?

MCUs are mostly designed for embedded applications and are heavily used in automatically controlled electronic devices such as cellphones, cameras, microwave ovens, washing machines, etc.

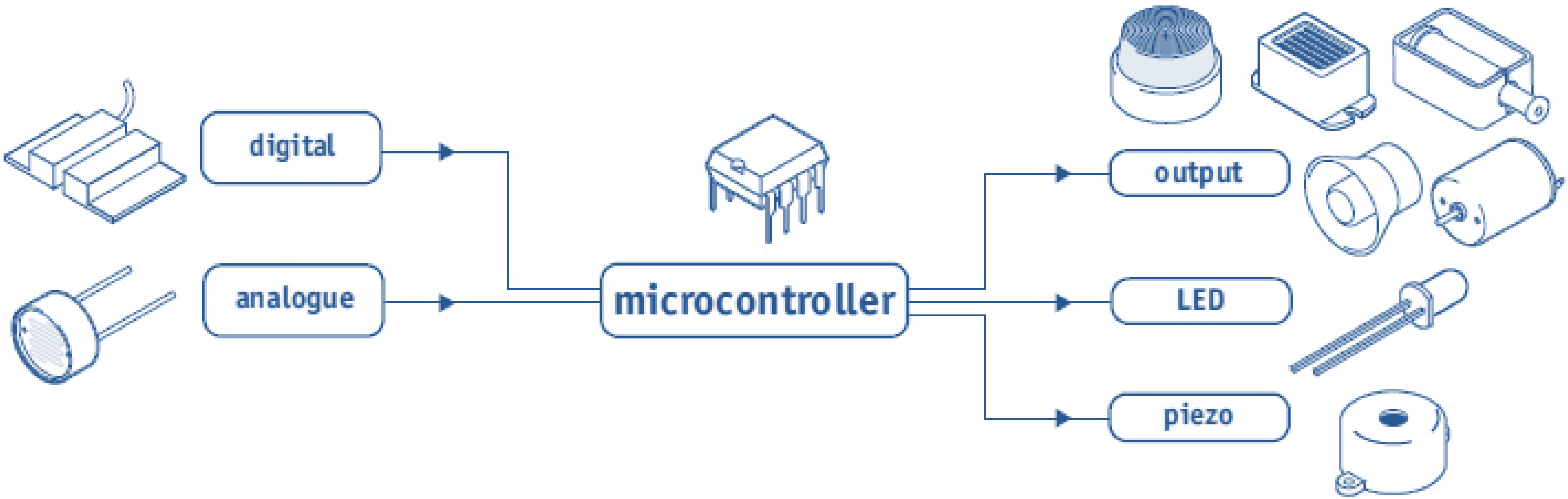


MCUs come in different shapes, sizes and configurations

MICROCONTROLLERS CONTROL CIRCUITS


Vcc
PORT 1.0-
PORT 1.7
PORT 1
RAM
EEPROM
(FLASH)
PORT 0
PORT 0.0 –
PORT 0.7
RST
CPU
ĒĀ
ALE
PSEN\
PORT 3.0 –
PORT 3.7
PORT 3
TIMERS, COUNTERS, INTERRUPTS, SERIAL PORT, SFRs, WATCH DOG
PORT 2
PORT 2.0-
PORT 2.7
XTAL1
XTAL2
GND

INPUT PROCESS OUTPUT



digital

analogue

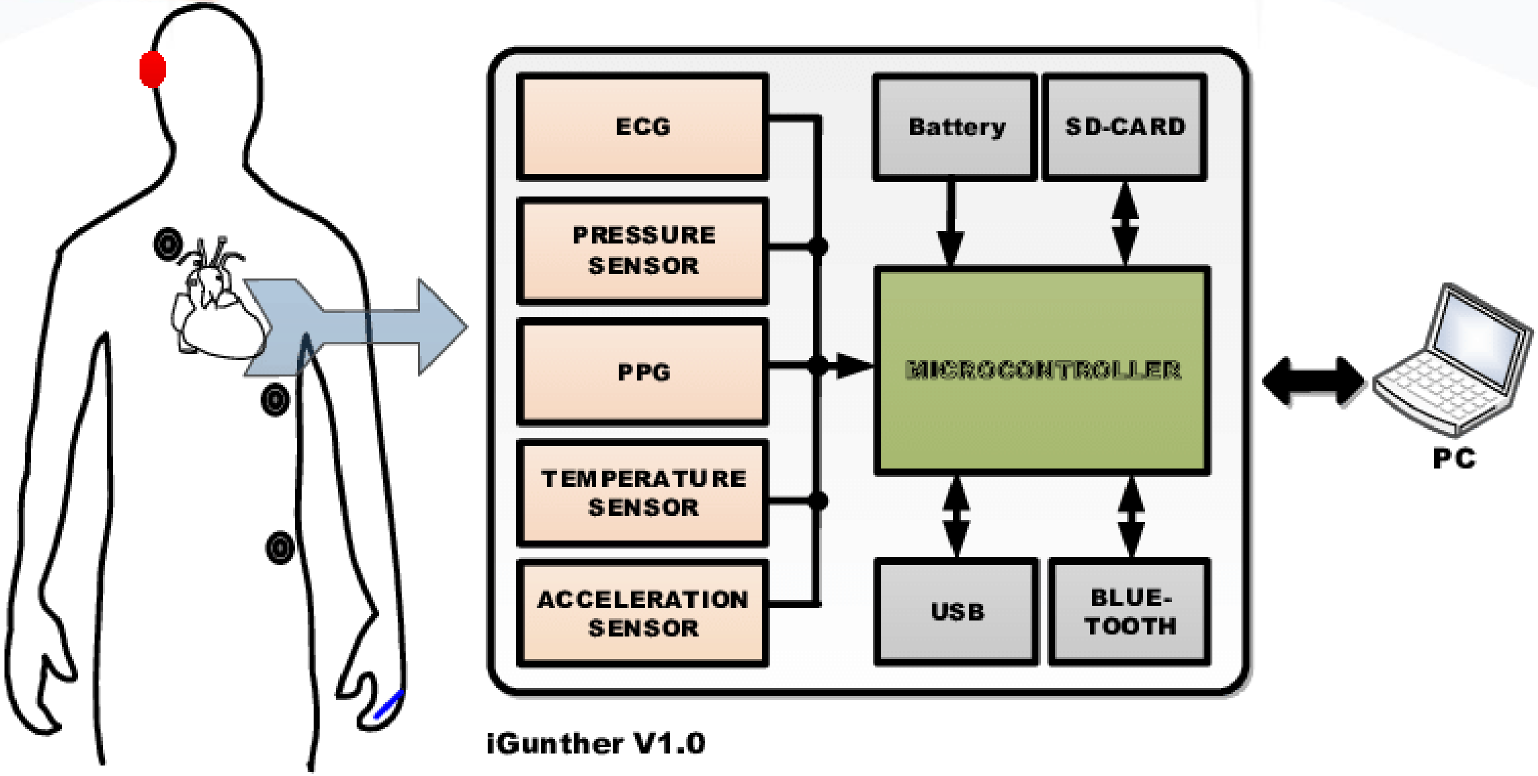
microcontroller

output

LED

piezo

IN-EAR PULSE BIODATA ACQUISITION



ECG

PRESSURE SENSOR

PPG

TEMPERATURE SENSOR

ACCELERATION SENSOR

Battery

SD-CARD

MICROCONTROLLER

USB

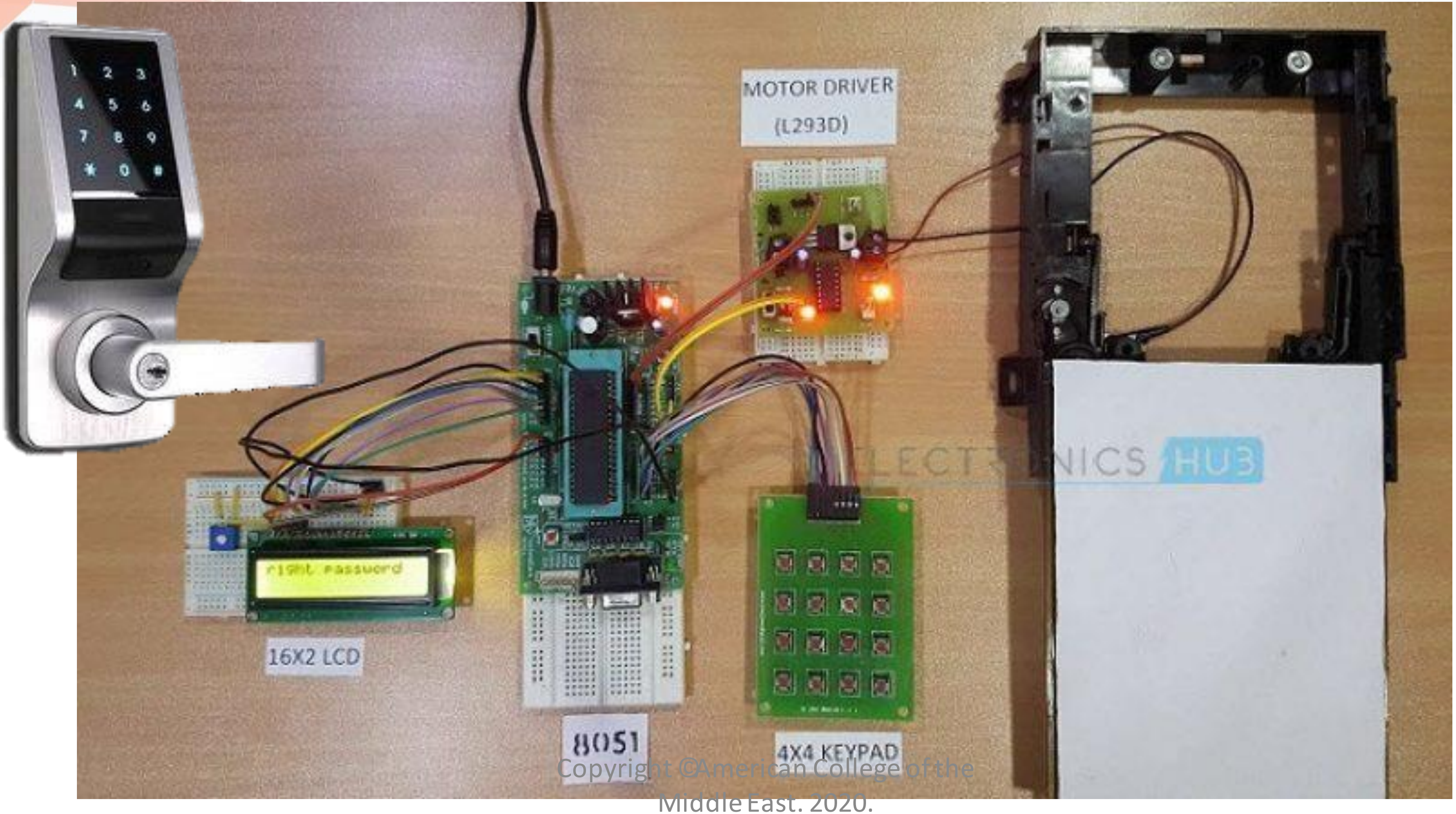
BLUE-TOOTH

PC

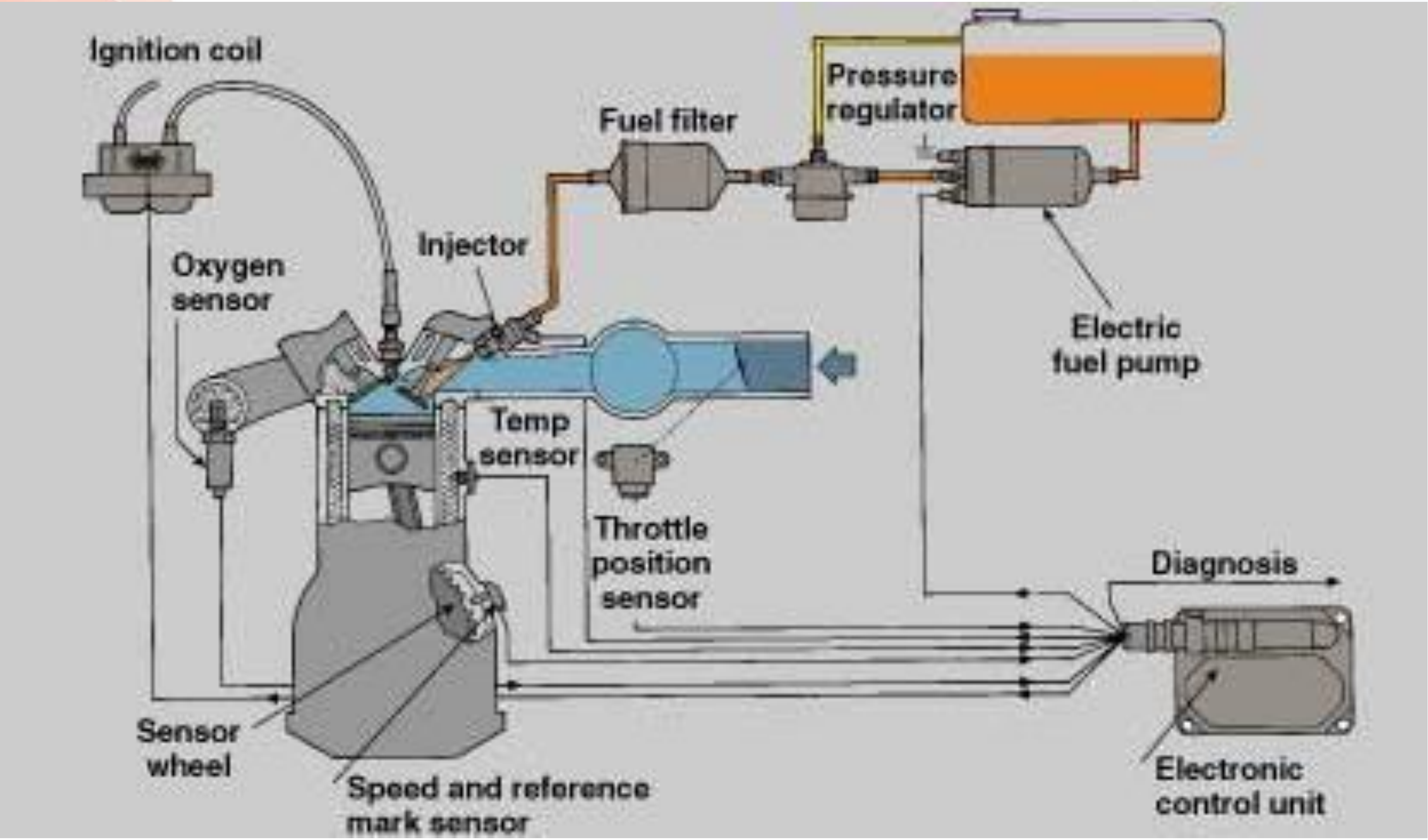
iGunther V1.0

● PRESSURE SENSOR | PPG-SENSOR ◎ ECG ELECTRODE

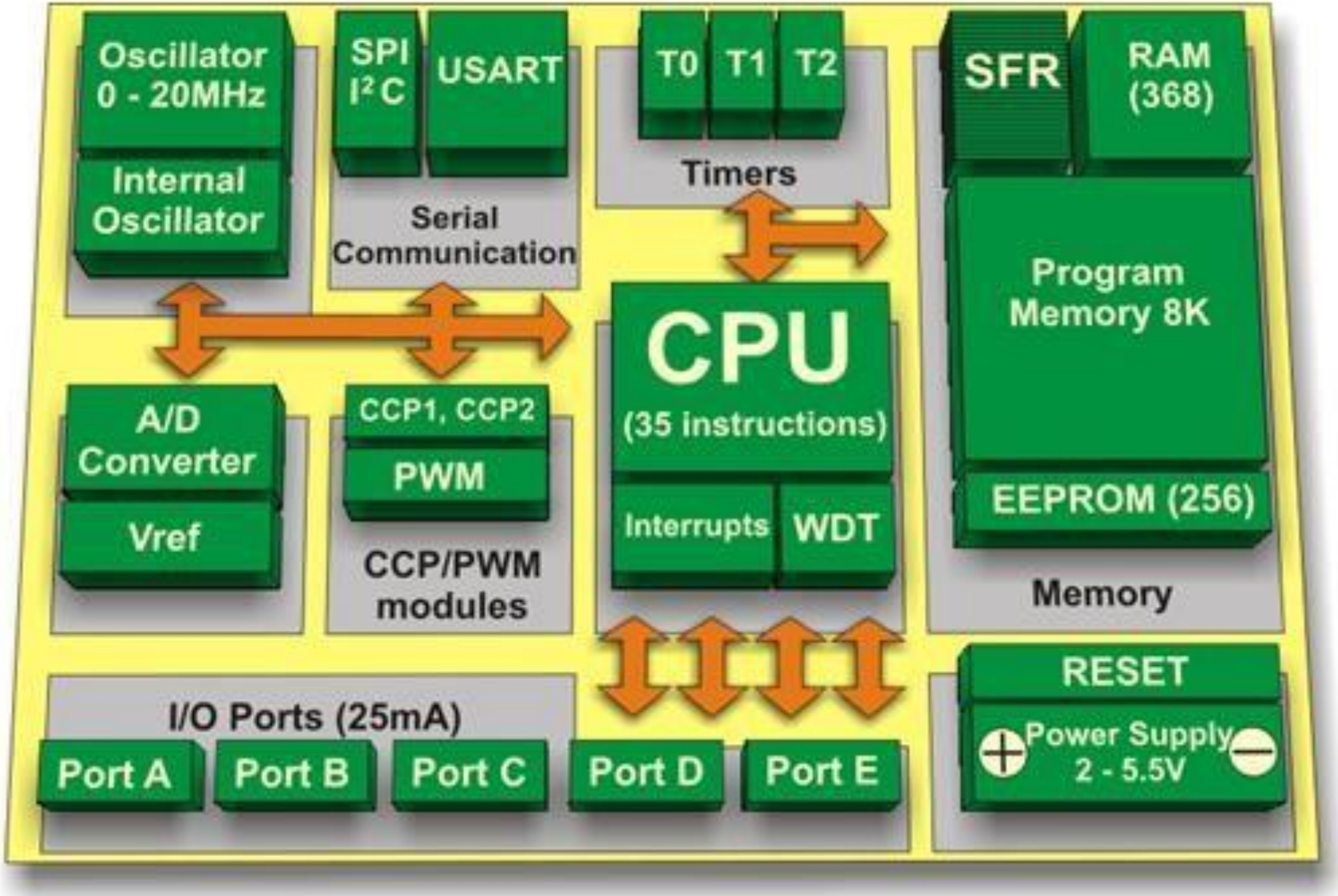
PASSWORD BASED DOOR LOCK SYSTEM



ENGINE CONTROL UNIT



ARCHITECTURE OF A MICROCONTROLLER



in order to work, MCUs need:

1. Power
2. A program (code) to follow
3. Inputs and Outputs (HW & SW)

in order to work, MCUs need:

1. Power
2. A program (code) to follow
3. Inputs and Outputs (HW & SW)

DIGITAL COMPONENTS

in order to work, MCUs need:

1. Power
2. A program (code) to follow
3. Inputs and Outputs (HW & SW)

DATA REPRESENTATION

DIGITAL COMPONENTS

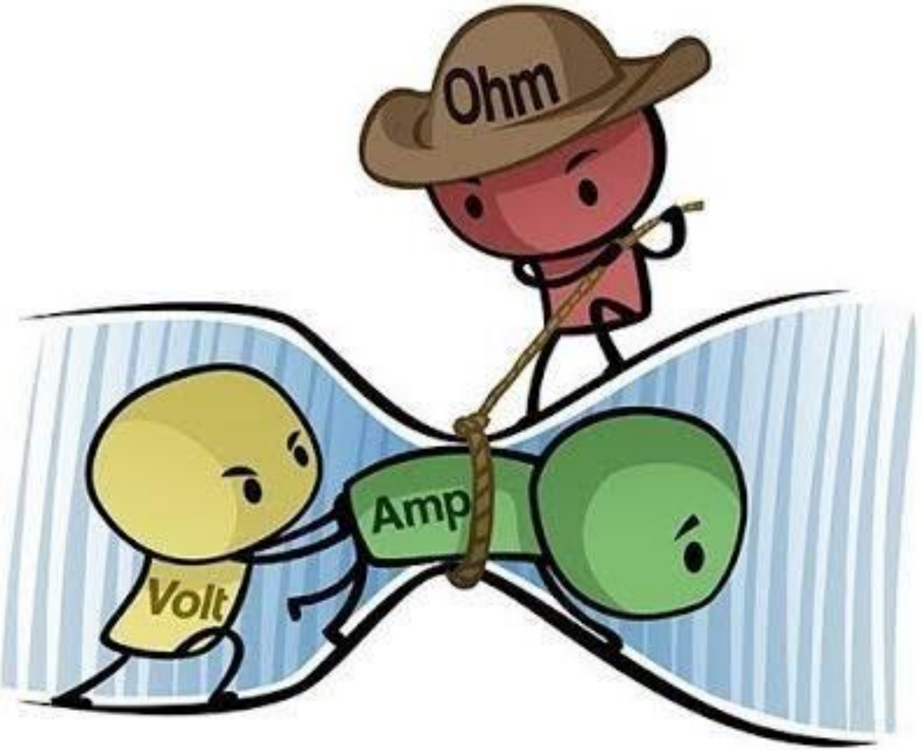
basic electronic components

Ohm’s law

Boolean logic

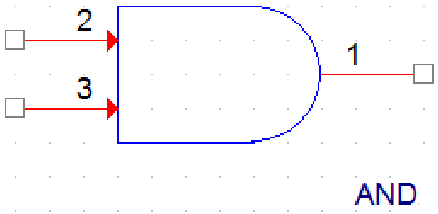
understand variables

OHM'S LAW

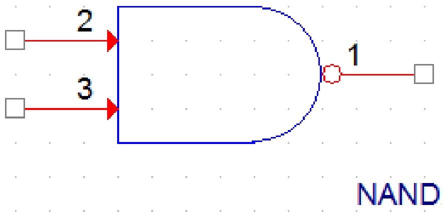


$$V = I \times R$$

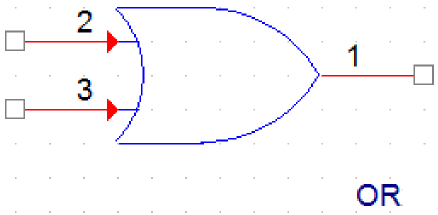
BOOLEAN LOGIC



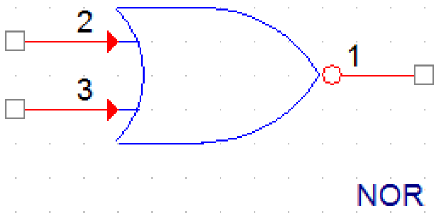
AB	F
00	0
01	0
10	0
11	1



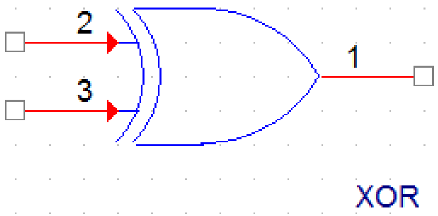
AB	F
00	1
01	1
10	1
11	0



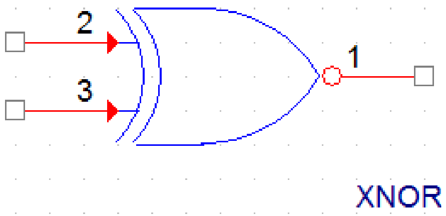
AB	F
00	0
01	1
10	1
11	1



AB	F
00	1
01	0
10	0
11	0

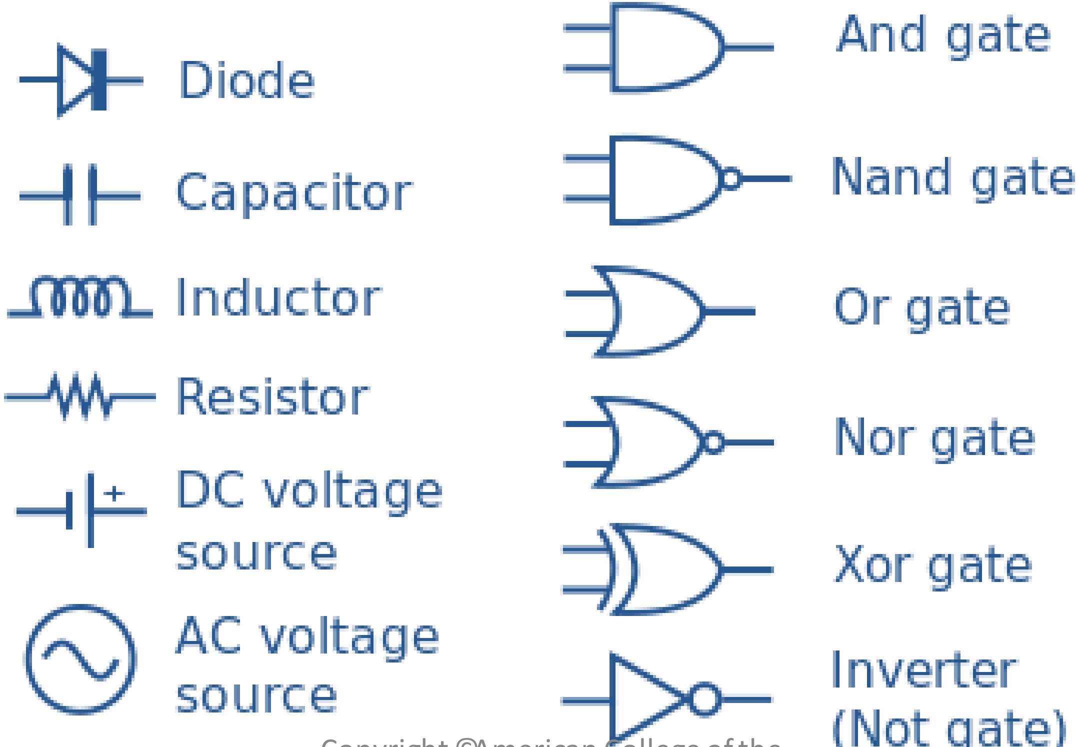


AB	F
00	0
01	1
10	1
11	0



AB	F
00	1
01	0
10	0
11	1

BASIC ELECTRONIC COMPONENTS



Diode

Capacitor

Inductor

Resistor

DC voltage source

AC voltage source

And gate

Nand gate

Or gate

Nor gate

Xor gate

Inverter (Not gate)

DATA REPRESENTATION

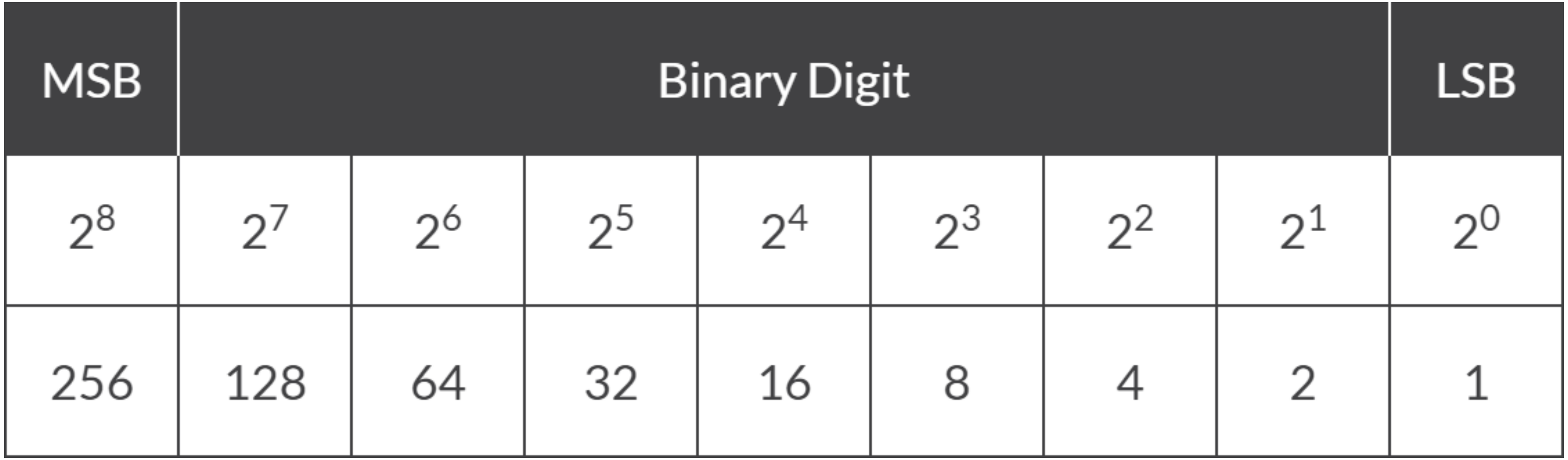
binary system

hexadecimal system

decimal system

how to alternate between them

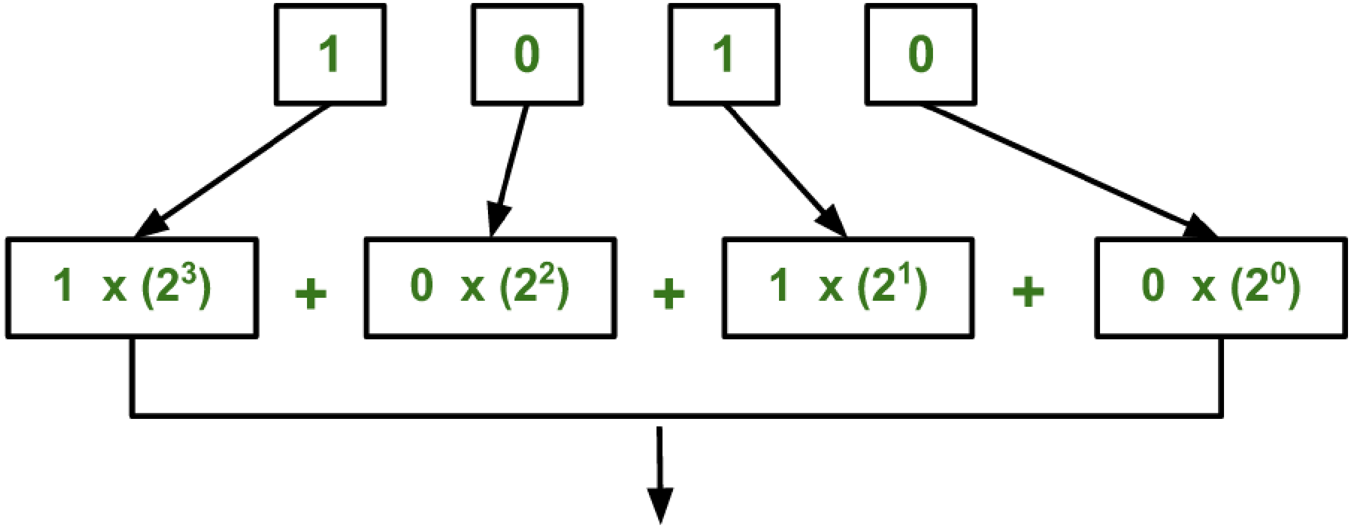
binary to decimal



MSB	Binary Digit							LSB
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

binary to decimal

Binary number - 1010

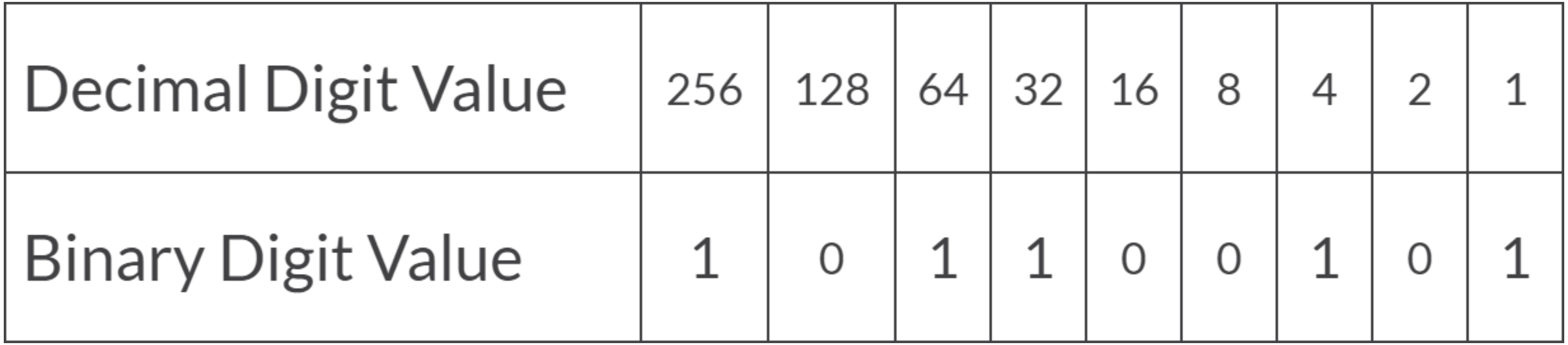


1 0 1 0

$1 \times (2^3)$ + $0 \times (2^2)$ + $1 \times (2^1)$ + $0 \times (2^0)$

10 (Decimal Equivalent)

binary to decimal



Decimal Digit Value	256	128	64	32	16	8	4	2	1
Binary Digit Value	1	0	1	1	0	0	1	0	1

binary to decimal

By adding together ALL the decimal number values from right to left at the positions that are represented by a “1” gives us:

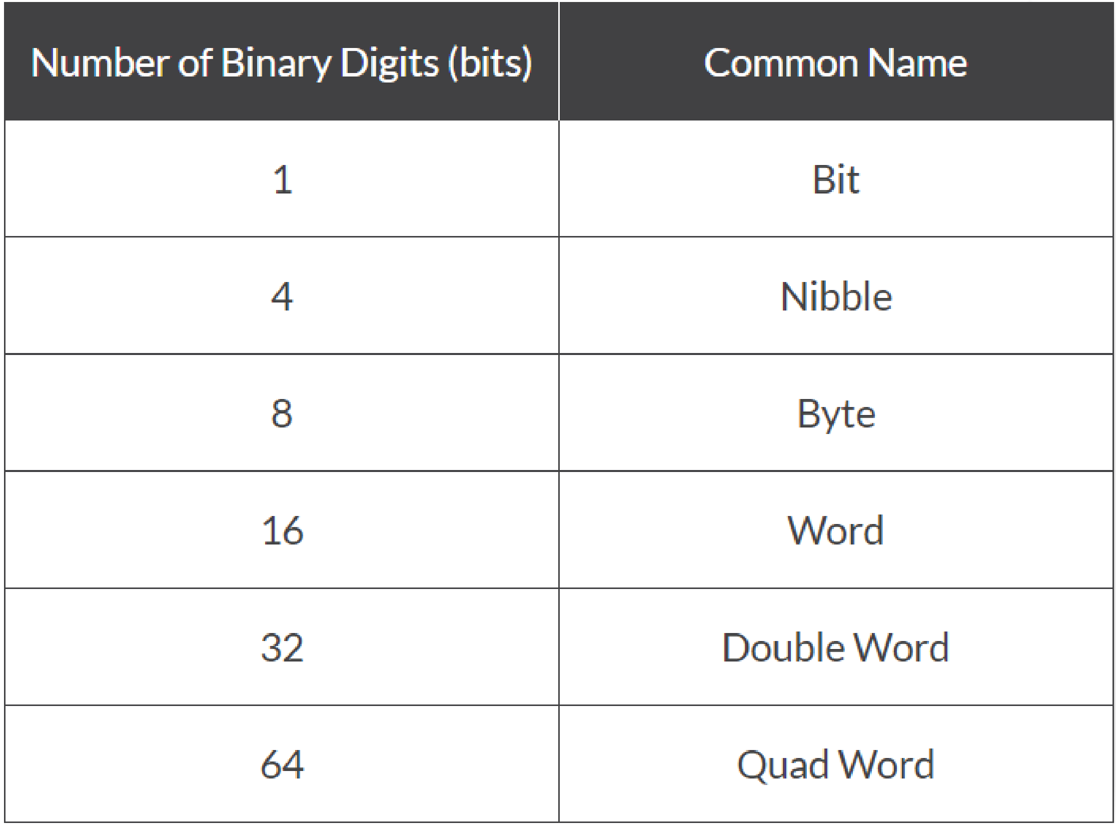
$(256) + (64) + (32) + (4) + (1) = 357_{10}$

or three hundred and fifty seven as a decimal number.

binary to decimal

Decimal Digit Value	256	128	64	32	16	8	4	2	1
Binary Digit Value						0			

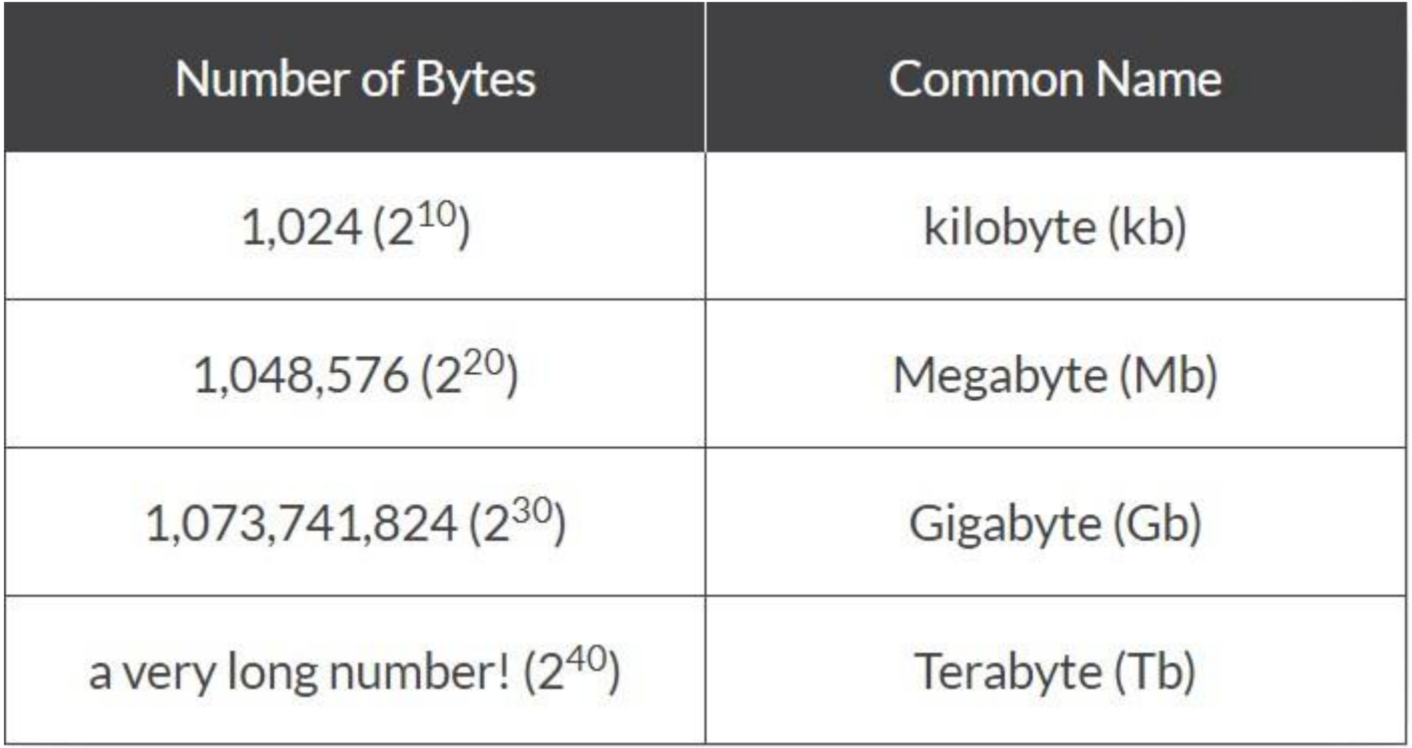
binary system units



Number of Binary Digits (bits)	Common Name
1	Bit
4	Nibble
8	Byte
16	Word
32	Double Word
64	Quad Word

Today, as micro-controller or microprocessor systems become increasingly larger, the individual binary digits (bits) are now grouped together into 8’s to form a single BYTE

file system units

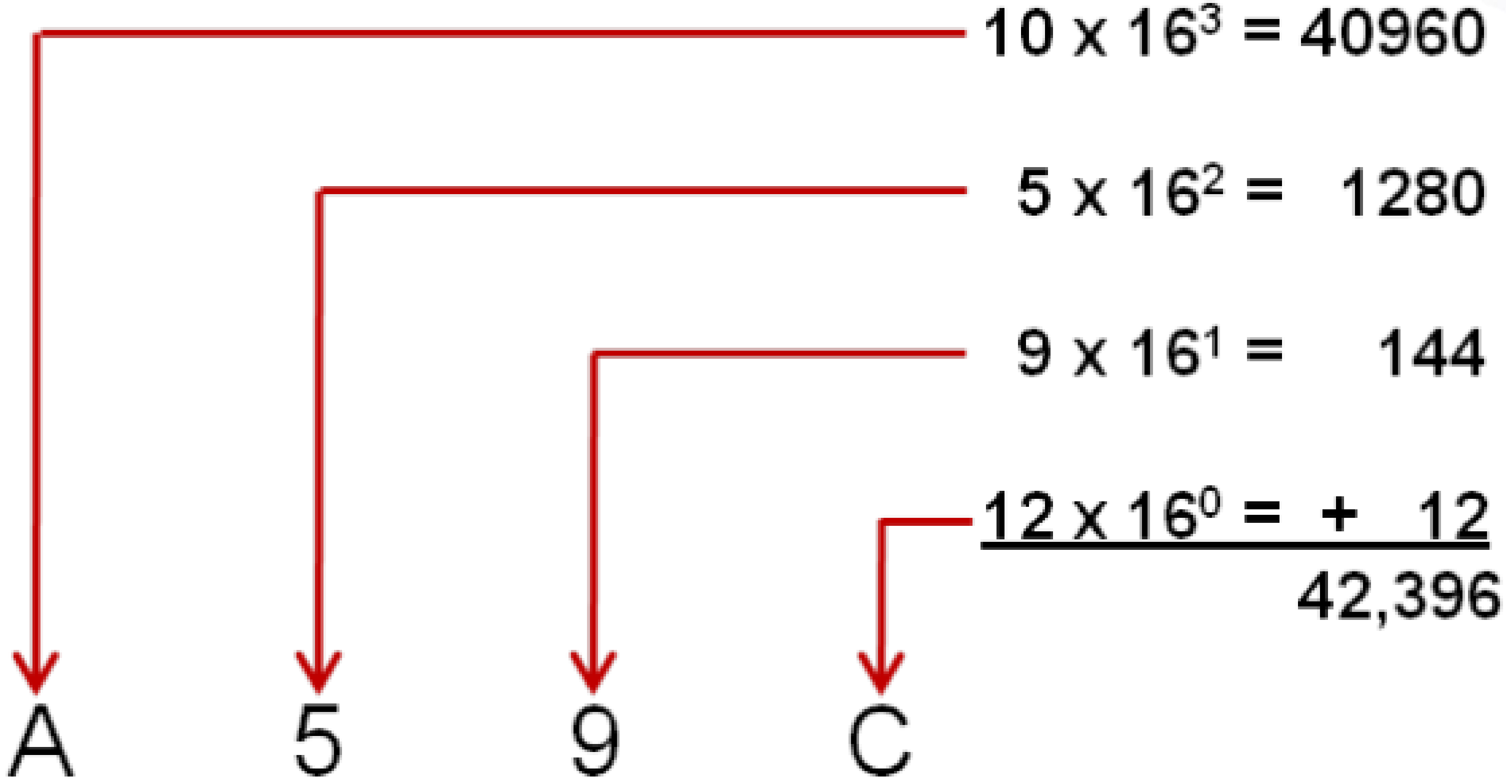


Number of Bytes	Common Name
1,024 (2^{10})	kilobyte (kb)
1,048,576 (2^{20})	Megabyte (Mb)
1,073,741,824 (2^{30})	Gigabyte (Gb)
a very long number! (2^{40})	Terabyte (Tb)

BINARY TO DECIMAL SUMMARY

- A “BIT” is the abbreviated term derived from BInary digiT
- A Binary system has only two states, Logic “0” and Logic “1” giving a base of 2
- A Decimal system uses 10 different digits, 0 to 9 giving it a base of 10
- A Binary number is a weighted number who’s weighted value increases from right to left
- The weight of a binary digit doubles from right to left
- A decimal number can be converted to a binary number by using the sum-of-weights method
    - When we convert numbers from binary to decimal, or decimal to binary, subscripts are used to avoid errors

hexadecimal to decimal



$10 \times 16^3 = 40960$

$5 \times 16^2 = 1280$

$9 \times 16^1 = 144$

$\underline{12 \times 16^0 = +\ 12}$

42,396

A 5 9 C

THANK YOU

CLASS 2

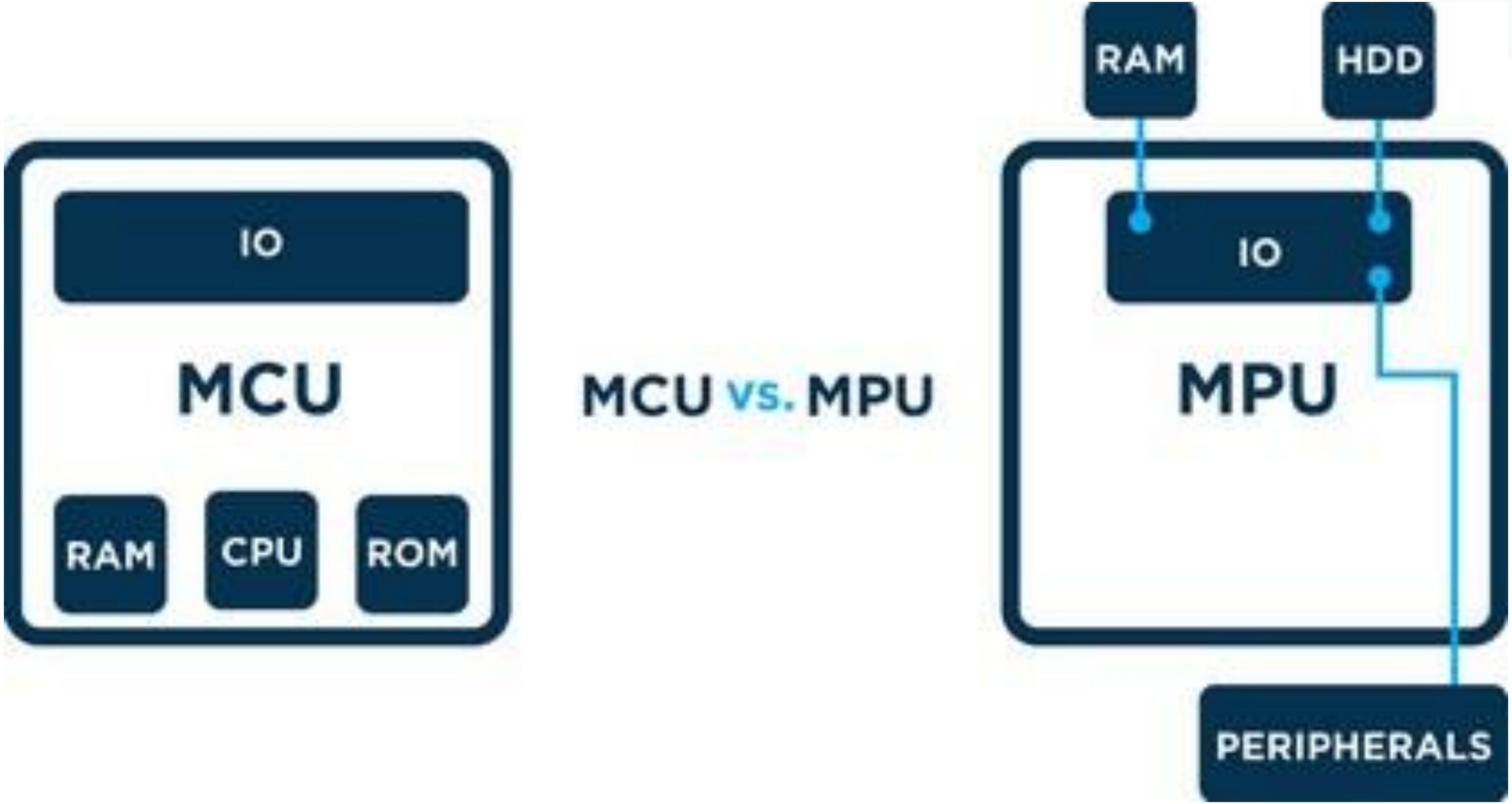
CLASS 1 REVIEW

what is a “Microprocessor”?

what is a microprocessor?

The Microprocessor, also known as the Central Processing Unit (CPU), is the brain of all computers and many household and electronic devices

MPU vs MCU



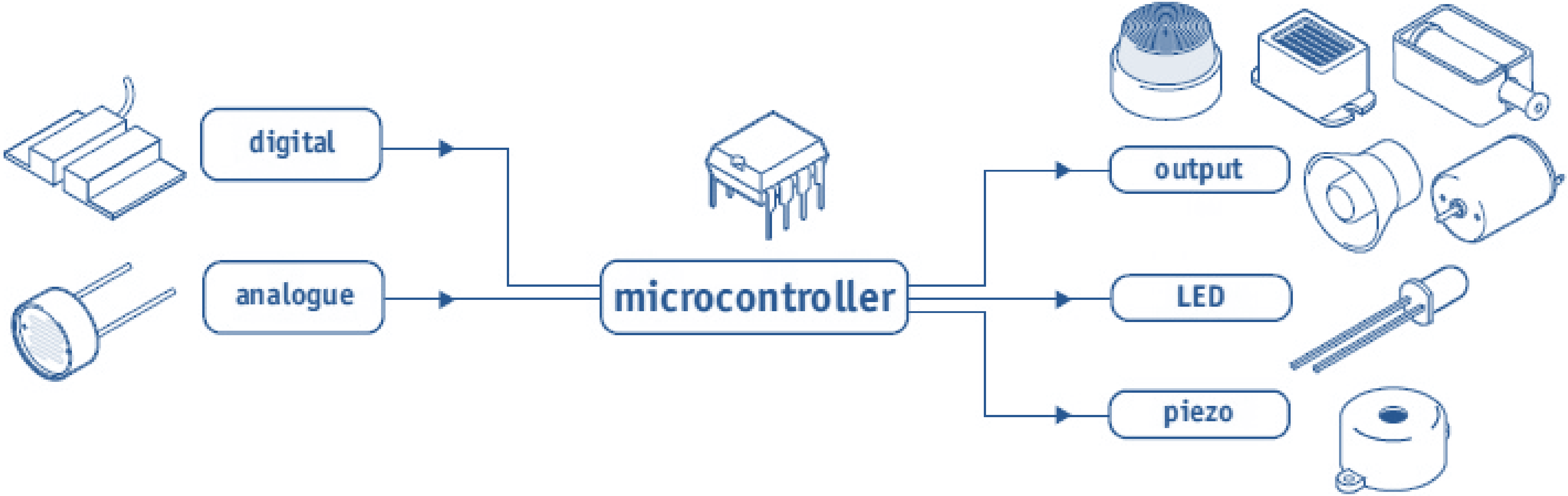
what is a microcontroller?

A microcontroller is a computer present in a single integrated circuit which is dedicated to perform one task and execute one specific application

what is a microcontroller?

MCUs contain memory, programmable input/output peripherals as well a processor

INPUT PROCESS OUTPUT



digital

analogue

microcontroller

output

LED

piezo

in order to work, MCUs need:

1. Power
2. A program (code) to follow
3. Inputs and Outputs (HW & SW)

DIGITAL COMPONENTS

basic electronic components

Ohm’s law

Boolean logic

understand variables

DATA REPRESENTATION

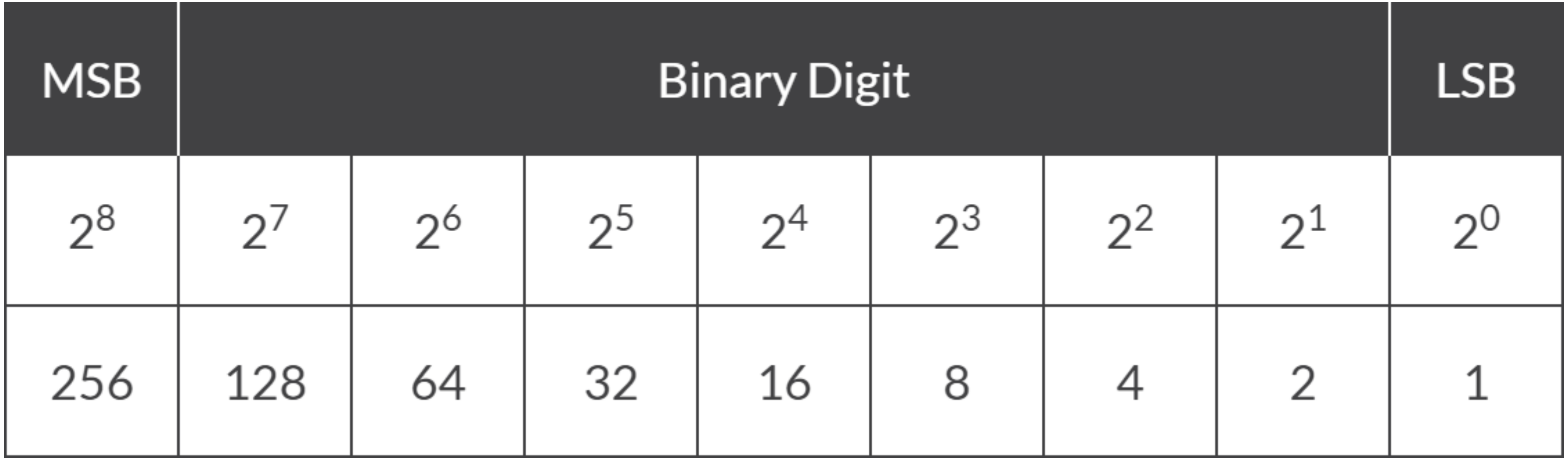
binary system

hexadecimal system

decimal system

how to alternate between them

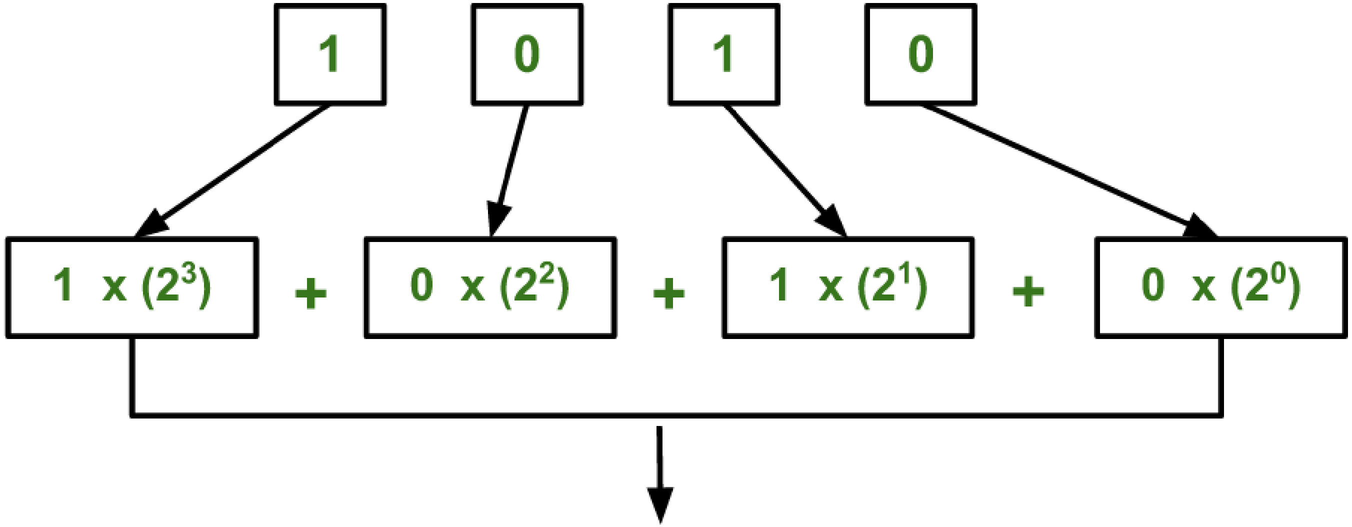
binary to decimal



MSB	Binary Digit							LSB
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

binary to decimal

Binary number - 1010

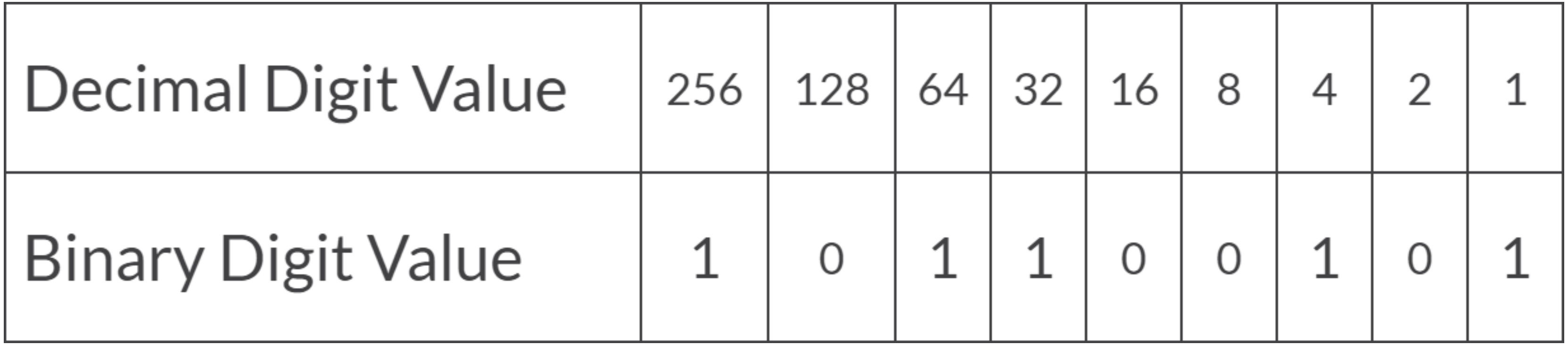


1 0 1 0

$1\ x\ (2^{3})$ + $0\ x\ (2^{2})$ + $1\ x\ (2^{1})$ + $0\ x\ (2^{0})$

10 (Decimal Equivalent)

binary to decimal



Decimal Digit Value	256	128	64	32	16	8	4	2	1
Binary Digit Value	1	0	1	1	0	0	1	0	1

CLASS 2

- INTRODUCTION TO ASSEMBLY LANGUAGE
- ADDING AND SUBSTRACTING VALUES
- REGISTERS AND OPERATIONS
- TRIS AND PORT REGISTERS
- LED BLINK

what is a “Assembly Language”?

ASSEMBLY LANGUAGE IS AN EXTREMELY LOW-LEVEL PROGRAMMING LANGUAGE THAT HAS A 1-TO-1 CORRESPONDENCE TO MACHINE CODE — THE SERIES OF BINARY INSTRUCTIONS WHICH MOVE VALUES IN AND OUT OF REGISTERS IN A CPU

GENERAL OPERATIONS

```
MOVLW 0xFF
MOVWF PORTA
ADDLW b101
SUBLW   25

BCF RP1
BSF RP0
```

ADDING AND SUBSTRACTING

A + B

MOVLW A $\boxed{A}_{W}$

ADDLW B $\boxed{A + B}_{W}$

LET’S PRACTICE

10 + 20

LET’S PRACTICE

0xF0 + 0x01

A - B

MOVLW A

$\boxed{A}_{W}$

SUBLW B

$\boxed{A - B}_{W}$

LET’S PRACTICE

50 - 10

LET’S PRACTICE

0xFF - 0xAA

REGISTERS AND OPERATIONS

STATUS

To change from Bank 0 to Bank 1 we talk to the STATUS register. We do this by setting the RP0 and RP1 bits. In most cases we'll be moving only between Bank 0 and Bank 1, thus we can just modify the value of the bit 5 of the STATUS register.

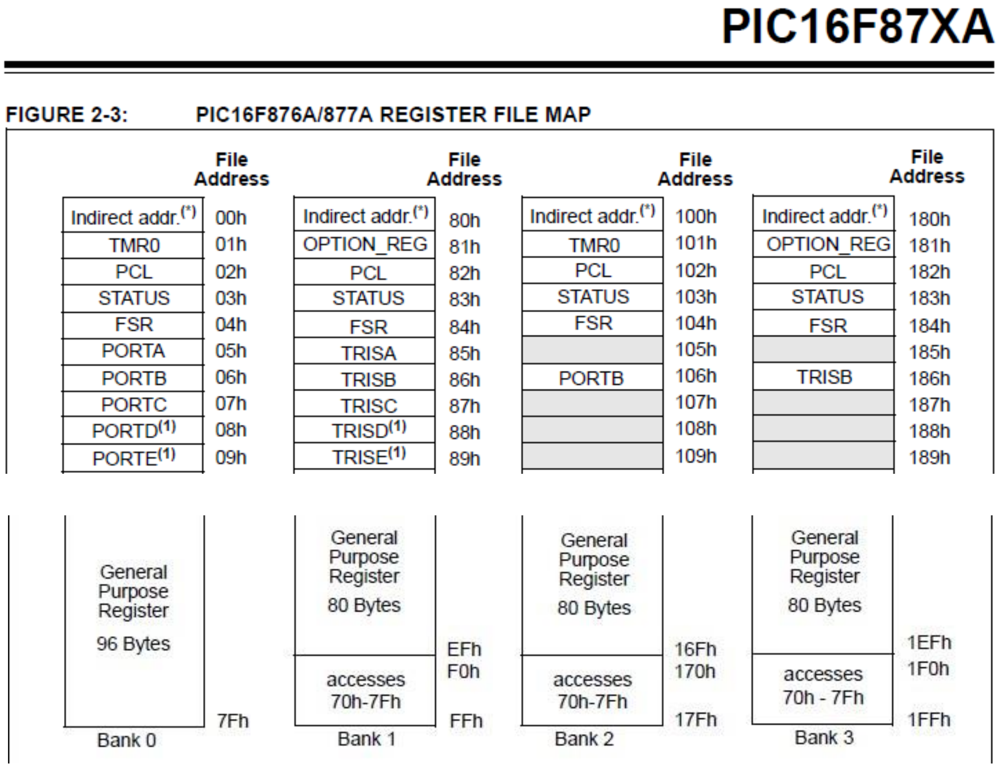
TRISD is in BANK 1

```
BSF     STATUS, 5
```

PORTD is in BANK 0

```
BCF     STATUS, 5
```

BANK SELECTION

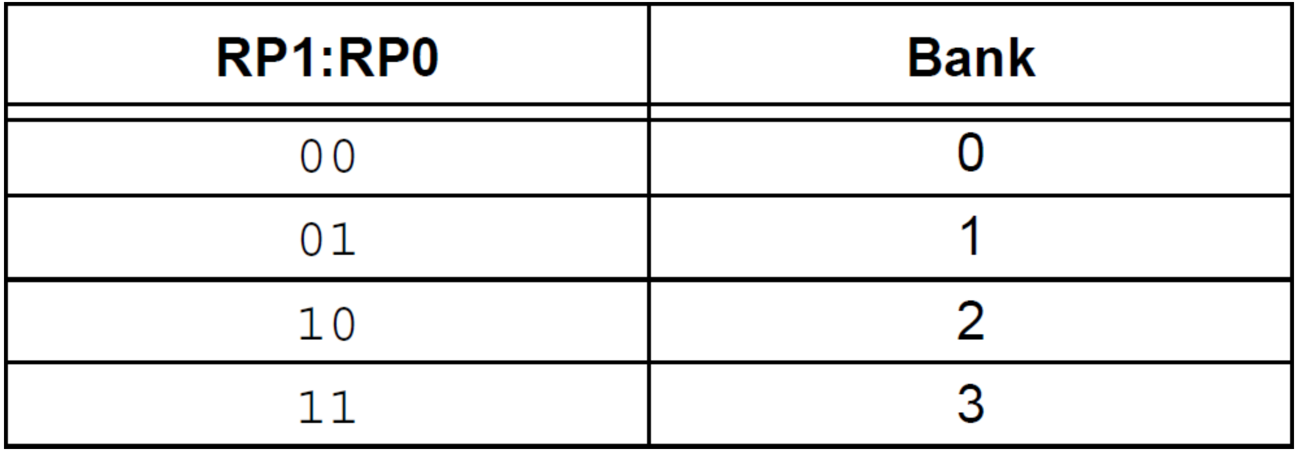


PIC16F87XA

FIGURE 2-3: PIC16F876A/877A REGISTER FILE MAP

	File Address		File Address		File Address		File Address
Indirect addr.[*]	00h	Indirect addr.[*]	80h	Indirect addr.[*]	100h	Indirect addr.[*]	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		105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h		107h		187h
PORTD[1]	08h	TRISD[1]	88h		108h		188h
PORTE[1]	09h	TRISE[1]	89h		109h		189h
General Purpose Register 96 Bytes		General Purpose Register 80 Bytes	EFh	General Purpose Register 80 Bytes	16Fh	General Purpose Register 80 Bytes	1EFh
		accesses 70h-7Fh	F0h	accesses 70h-7Fh	170h	accesses 70h - 7Fh	1F0h
	7Fh		FFh		17Fh		1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

BANK SELECTION



RP1:RP0	Bank
00	0
01	1
10	2
11	3

TRISD is in BANK 1

```
BCF     STATUS, RP1
BSF     STATUS, RP0
```

PORTD is in BANK 0

```
BCF     STATUS, RP1
BCF     STATUS, RP0
```

W REGISTER

The W register is a general register in which you can put any value that you wish. Once you have assigned a value to W, you can add it to another value, or move.

MOVLW

The MOVLW command means ‘Move Literal Value Into W’, which in English means put the value that follows directly into the W register.

```
MOVLW  0xAA
```

MOVWF

This instruction means “Move The Contents Of W Into The Register Address That Follows”.

MOVWF TRISB

TRIS AND PORT REGISTERS

TRIS

We use the TRIS Register to program a pin to be an output or an input by simply sending a 0 (out) or a 1 (in) to the relevant bit in the register.

```
MOVLW 0xFF
MOVWF TRISB
```

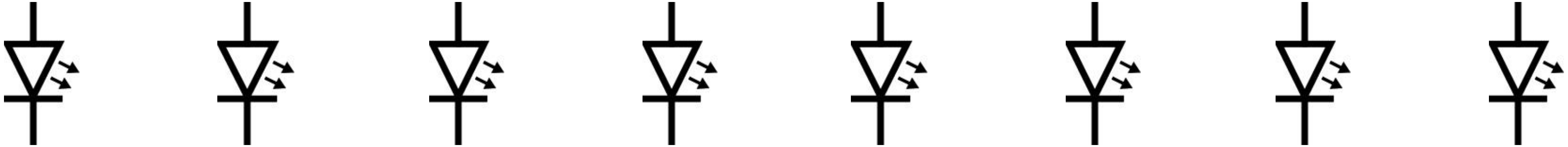
PORT

To send one of our output pins high, we simply send a ‘1’ to the corresponding bit in our PORTx register.

```
MOVLW 0xFF
MOVWF PORTx
```

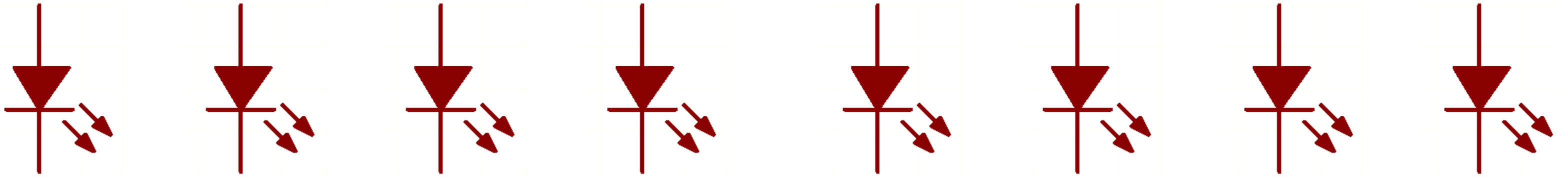
PORTB

PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



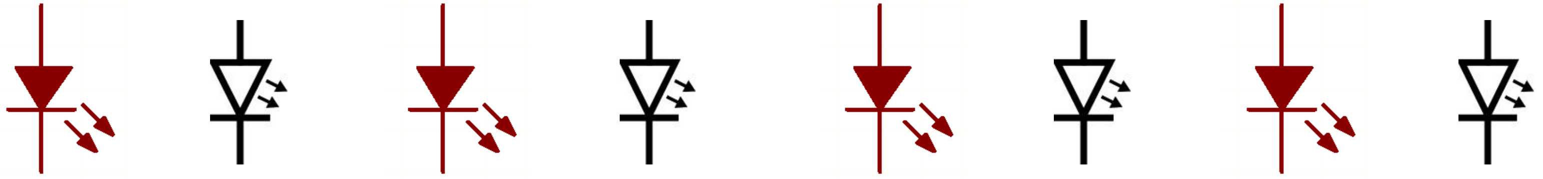
PORTB

PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



PORTB

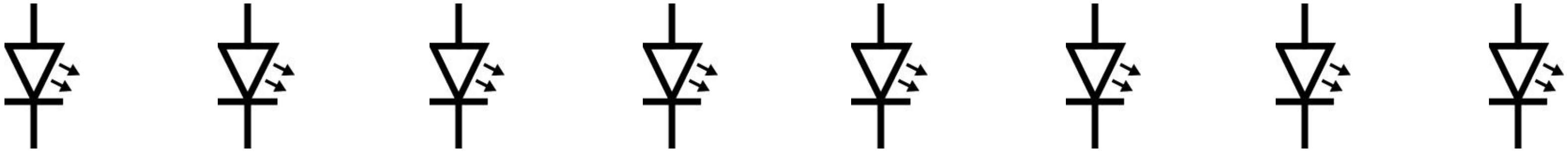
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



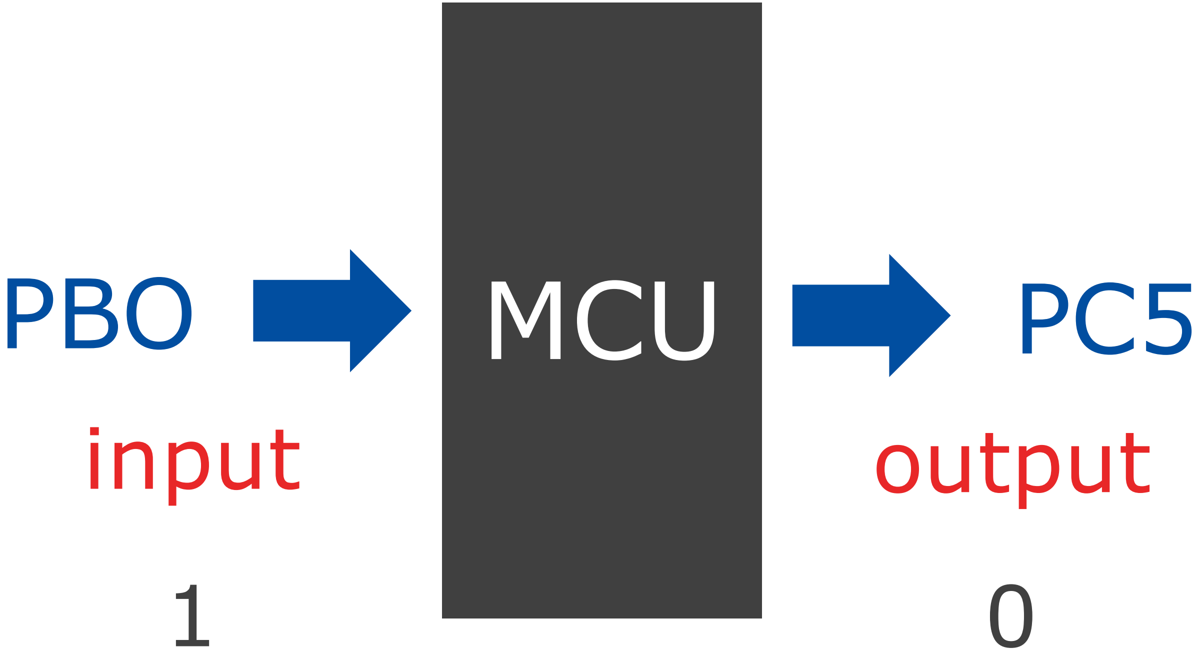
TRISB

PORTB

PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



TRISx



PBO

MCU

PC5

input

output

1

0

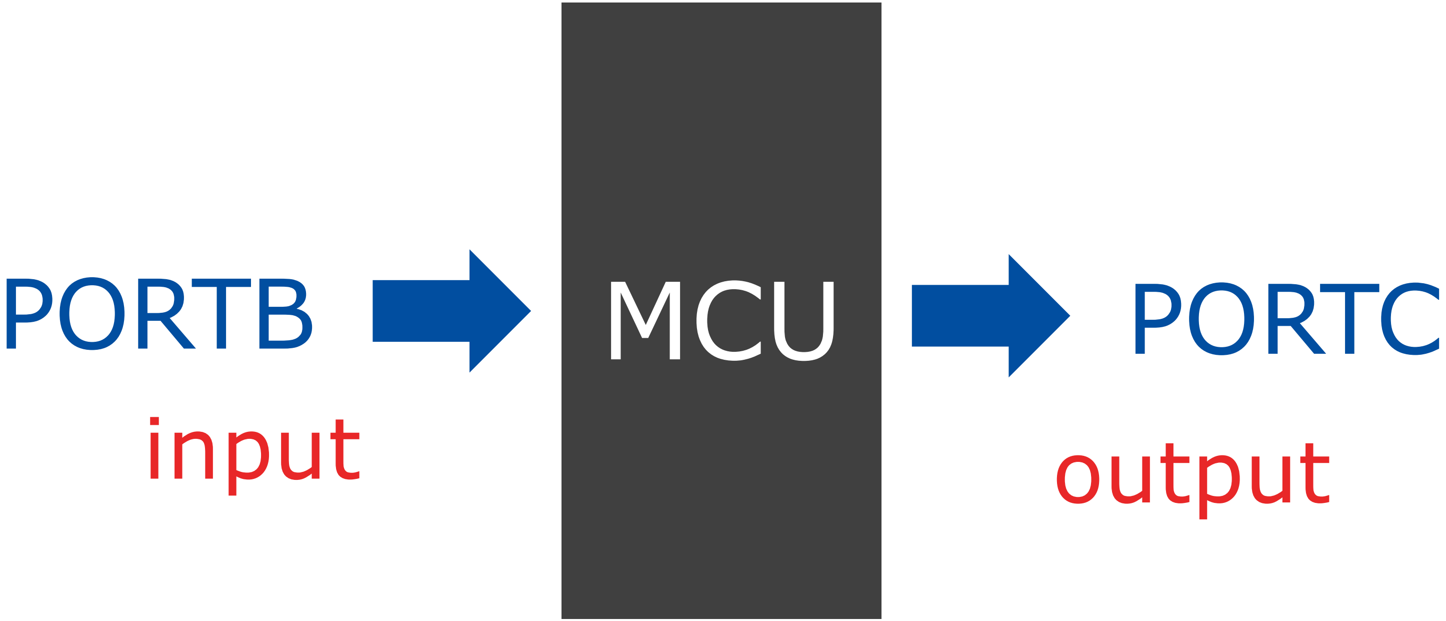
PORTB.0 = 1

```
BSF      TRISB, 0
```

PORTC.5 = 0

```
BCF      TRISC, 5
```

TRISx



PORTB

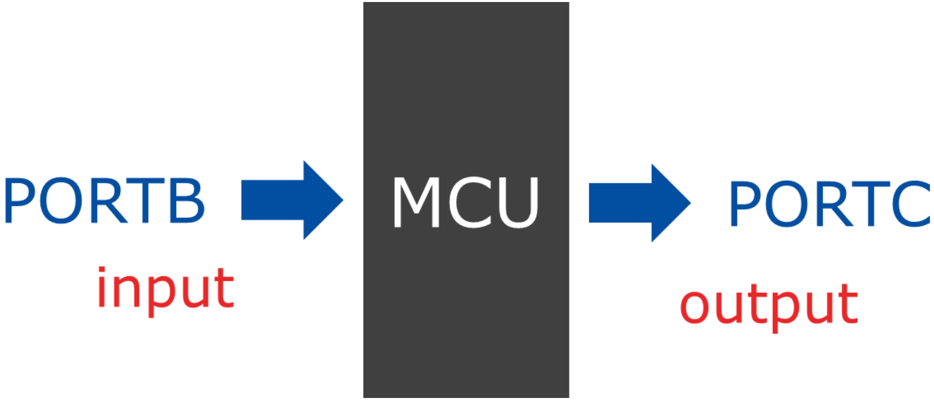
input

MCU

PORTC

output

TRISx



PORTB

input

MCU

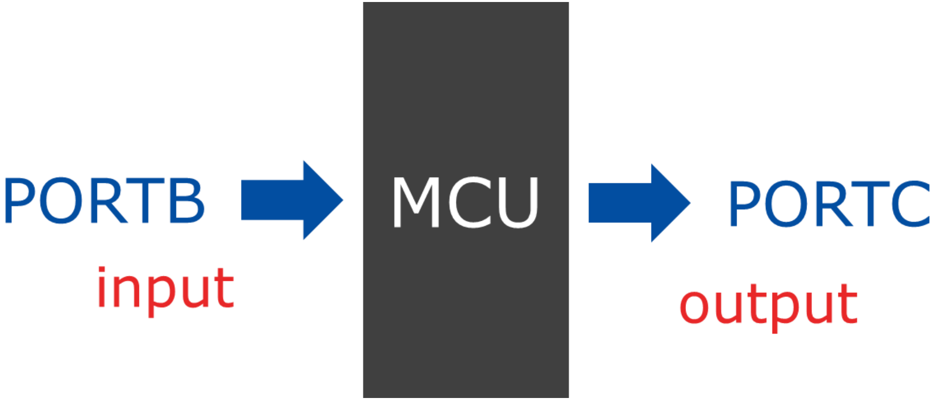
PORTC

output

0xFF -> TRISB

0x00 -> TRISC

TRISx



PORTB

input

MCU

PORTC

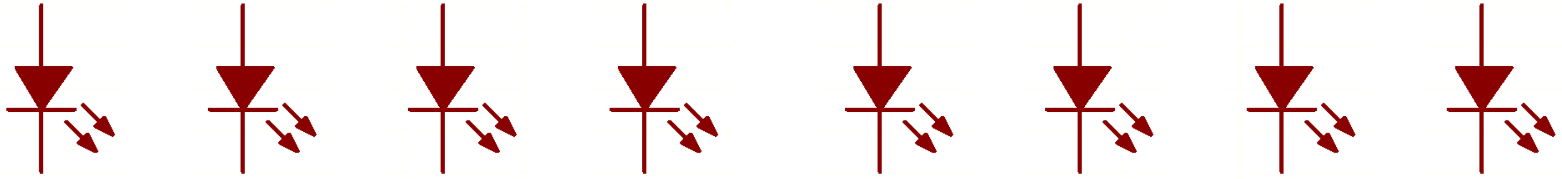
output

```
MOVLW 0xFF
MOVWF TRISB
CLRF TRISC
```

TURN LEDs ON AND OFF

PORTB

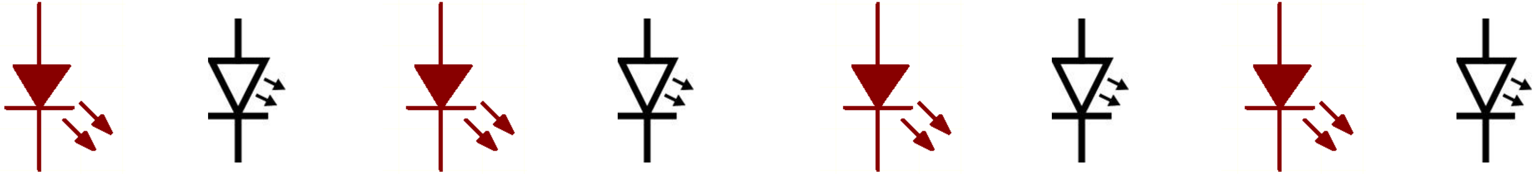
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



```
ORG 0x00
BCF       STATUS, RP1
BSF       STATUS, RP0
CLRF      TRISB
MAIN
BCF       STATUS, RP1
BCF       STATUS, RP0
MOVLW     0xFF
MOVWF     PORTB
END
```

PORTB

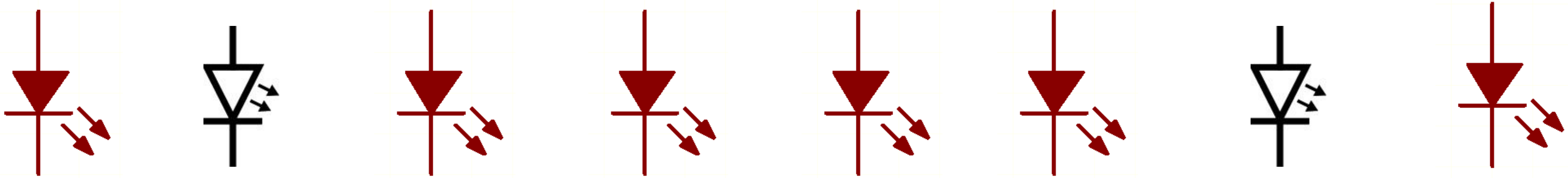
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



```
ORG 0x00
BSF       STATUS, 5
CLRF      TRISB
MAIN
BCF       STATUS, 5
MOVLW     0xAA
MOVWF     PORTB
END
```

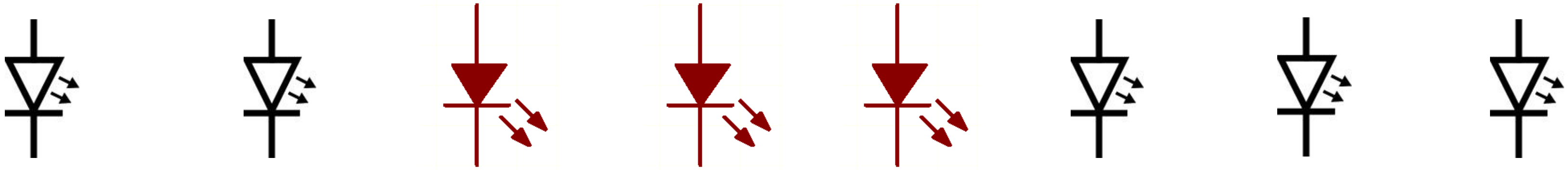
PORTC

PC7 PC6 PC5 PC4 PC3 PC2 PC1 PC0



PORTD

PD7 PD6 PD5 PD4 PD3 PD2 PD1 PD0



THANK YOU

CLASS 3

CLASS 3

- Review INTRODUCTION TO ASSEMBLY LANGUAGE
- ADDING AND SUBSTRACTING VALUES
- REGISTERS AND OPERATIONS
- TRIS AND PORT REGISTERS
- LED BLINK

what is a “Assembly Language”?

ASSEMBLY LANGUAGE IS AN EXTREMELY LOW-LEVEL PROGRAMMING LANGUAGE THAT HAS A 1-TO-1 CORRESPONDENCE TO MACHINE CODE — THE SERIES OF BINARY INSTRUCTIONS WHICH MOVE VALUES IN AND OUT OF REGISTERS IN A CPU

GENERAL OPERATIONS

```
MOVLW 0xFF
MOVWF PORTA
ADDLW b101
SUBLW  25
BCF RP1
BSF RP0
```

ADDING AND SUBSTRACTING

A + B

MOVLW A

$$\boxed{A}_{w}$$

ADDLW B

$$\boxed{A + B}_{w}$$

LET’S PRACTICE

0xF0 + 0x01

A - B

MOVLW B $\boxed{B}_W$

SUBLW A $\boxed{A - B}_W$

SUBLW subtract W from Literal Operation: k-(W)->W

LET'S PRACTICE

0xFF - 0xAA

REGISTERS AND OPERATIONS

STATUS

To change from Bank 0 to Bank 1 we talk to the STATUS register. We do this by setting the RP0 and RP1 bits. In most cases we'll be moving only between Bank 0 and Bank 1, thus we can just modify the value of the bit 5 of the STATUS register.

TRISD is in BANK 1

```
BCF     STATUS, RP1
BSF     STATUS, RP0
```

PORTD is in BANK 0

```
BCF     STATUS, RP1
BCF     STATUS, RP0
```

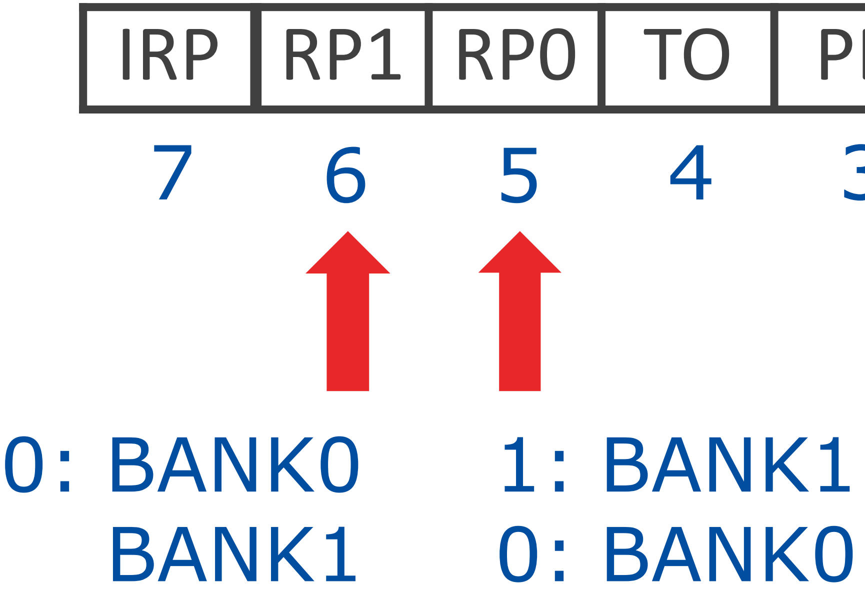
TRISD is in BANK 1

```
BCF     STATUS, RP1
BSF     STATUS, RP0
```

PORTD is in BANK 0

```
BCF     STATUS, RP1
BCF     STATUS, RP0
```

STATUS



IRP	RP1	RP0	TO	PD	Z	DC	C
7	6	5	4	3	2	1	0

0: BANK0
BANK1

1: BANK1
0: BANK0

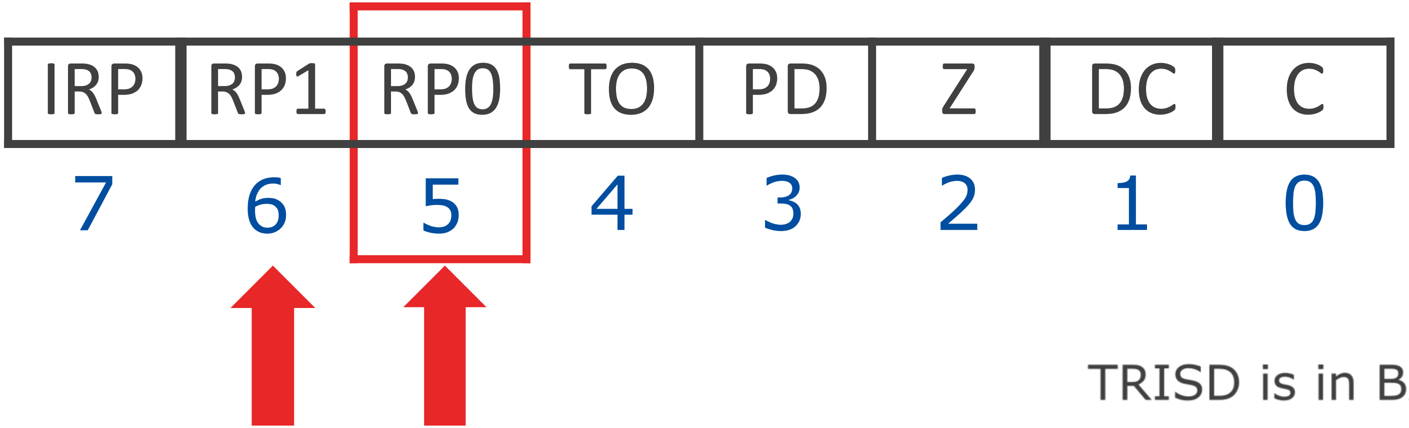
TRISD is in BANK 1

```
BCF     STATUS, RP1
BSF     STATUS, RP0
```

PORTD is in BANK 0

```
BCF     STATUS, RP1
BCF     STATUS, RP0
```

STATUS, 5



IRP	RP1	RP0	TO	PD	Z	DC	C
7	6	5	4	3	2	1	0

0: BANK0 1: BANK1
BANK1 0: BANK0

TRISD is in BANK 1

```
BCF     STATUS, RP1
BSF     STATUS, RP0
```

PORTD is in BANK 0

```
BCF     STATUS, RP1
BCF     STATUS, RP0
```

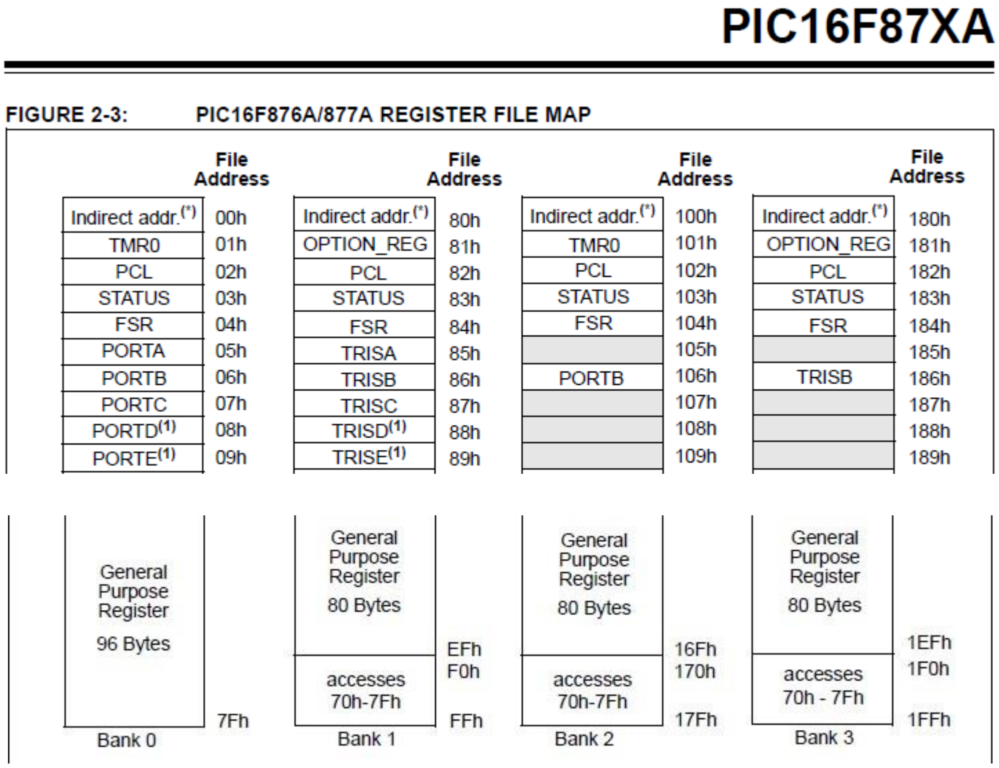
TRISD is in BANK 1

```
BSF     STATUS, 5
```

PORTD is in BANK 0

```
BCF     STATUS, 5
```

BANK SELECTION

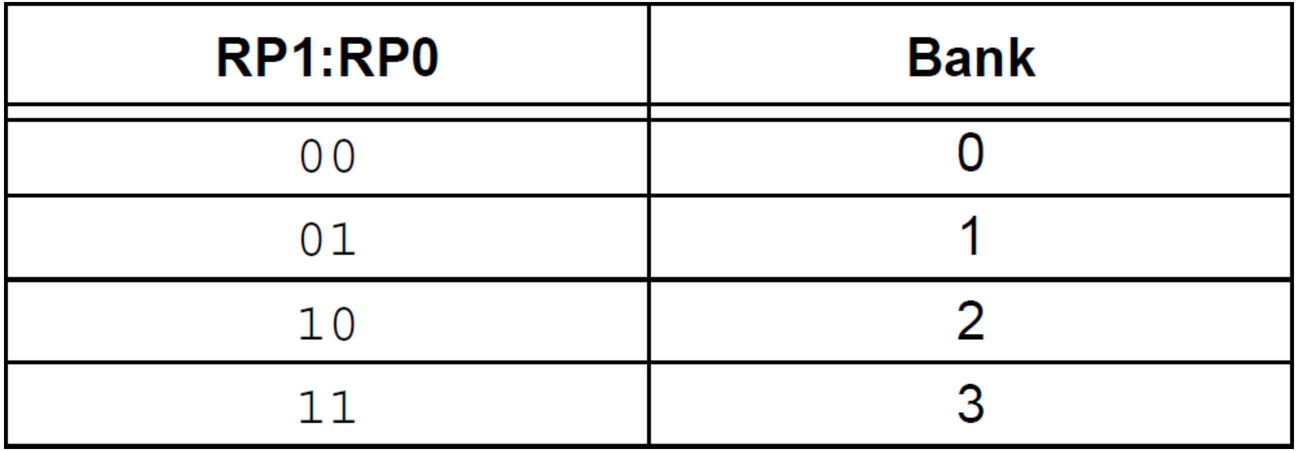


PIC16F87XA

FIGURE 2-3: PIC16F876A/877A REGISTER FILE MAP

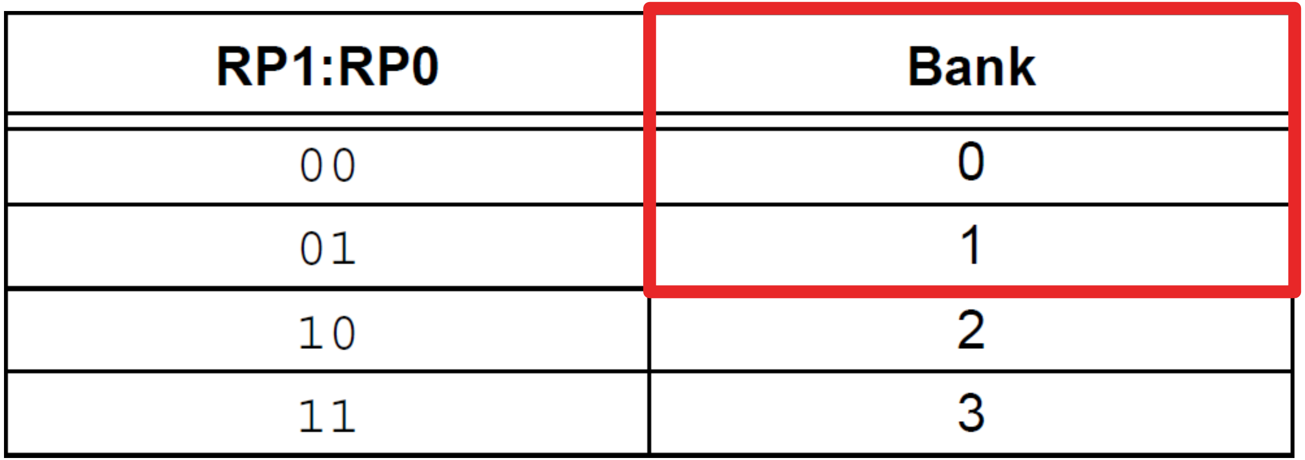
	File Address		File Address		File Address		File Address
Indirect addr.[*]	00h	Indirect addr.[*]	80h	Indirect addr.[*]	100h	Indirect addr.[*]	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h		105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h		107h		187h
PORTD[1]	08h	TRISD[1]	88h		108h		188h
PORTE[1]	09h	TRISE[1]	89h		109h		189h
General Purpose Register 96 Bytes		General Purpose Register 80 Bytes	EFh	General Purpose Register 80 Bytes	16Fh	General Purpose Register 80 Bytes	1EFh
	7Fh	accesses 70h-7Fh	F0h FFh	accesses 70h-7Fh	170h 17Fh	accesses 70h - 7Fh	1F0h 1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

BANK SELECTION



RP1:RP0	Bank
00	0
01	1
10	2
11	3

BANK SELECTION



RP1:RP0	Bank
00	0
01	1
10	2
11	3

W REGISTER

The W register is a general register in which you can put any value that you wish. Once you have assigned a value to W, you can add it to another value, or move.

MOVLW

The MOVLW command means ‘Move Literal Value Into W’, which in English means put the value that follows directly into the W register.

```
MOVLW  0xAA
```

MOVWF

This instruction means “Move The Contents Of W Into The Register Address That Follows”.

MOVWF TRISB

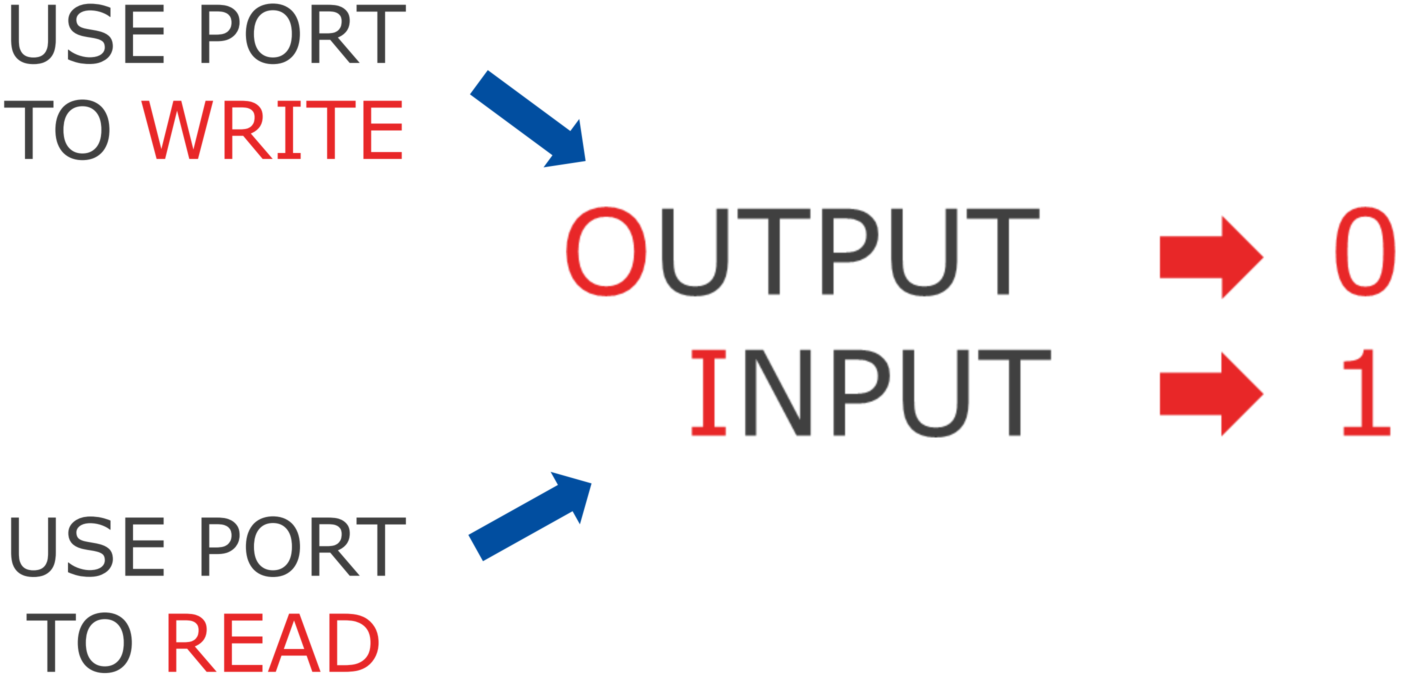
TRIS AND PORT REGISTERS

TRIS

We use the TRIS Register to program a pin to be an output or an input by simply sending a 0 or a 1 to the relevant bit in the register.

```
MOVLW 0xFF
MOVWF TRISB
```

TRIS



USE PORT TO WRITE

USE PORT TO READ

OUTPUT → 0

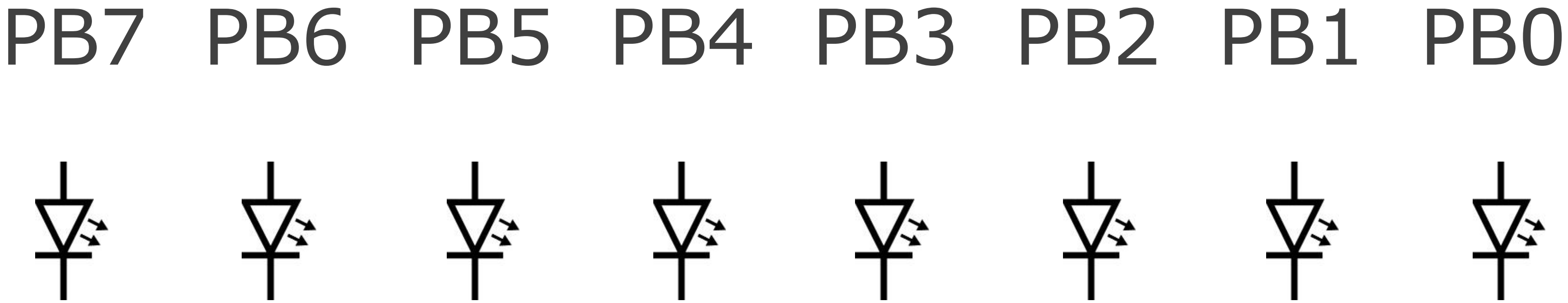
INPUT → 1

PORT

To send one of our output pins high, we simply send a ‘1’ to the corresponding bit in our PORTx register.

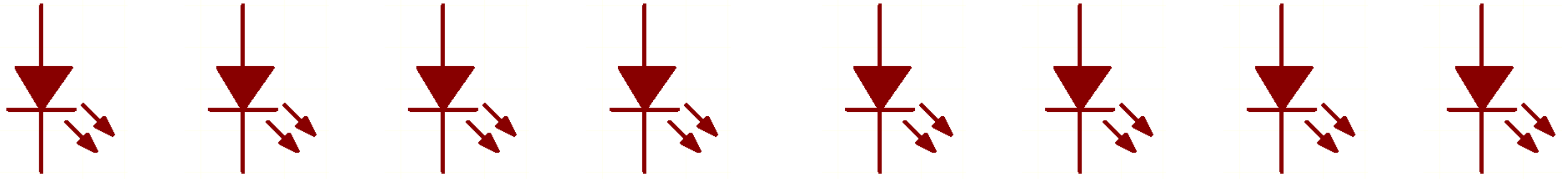
```
MOVLW 0xFF
MOVWF PORTx
```

PORTB



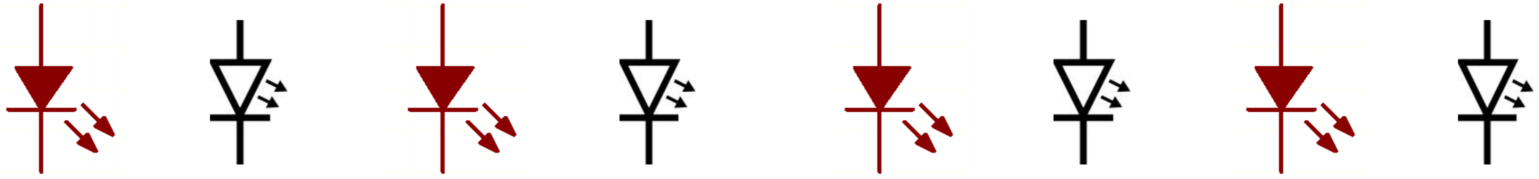
PORTB

PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



PORTB

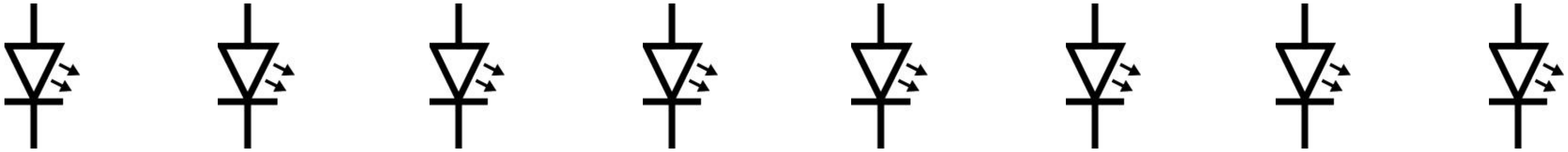
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



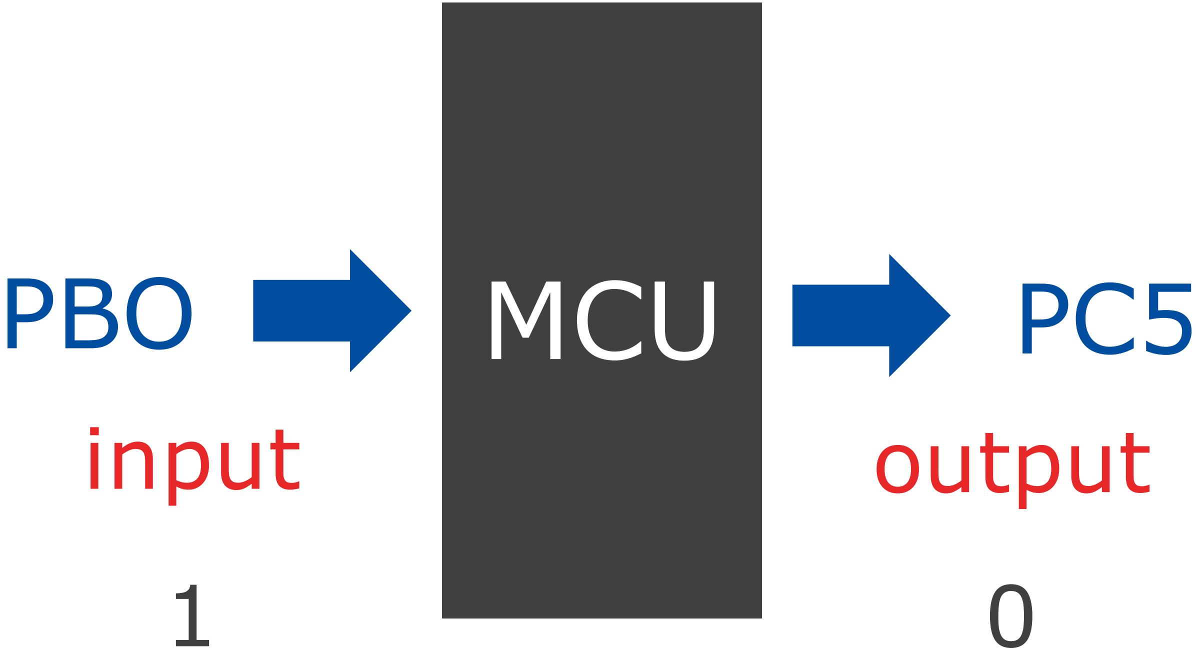
TRISB

PORTB

PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



TRISx



PBO

MCU

PC5

input

output

1

0

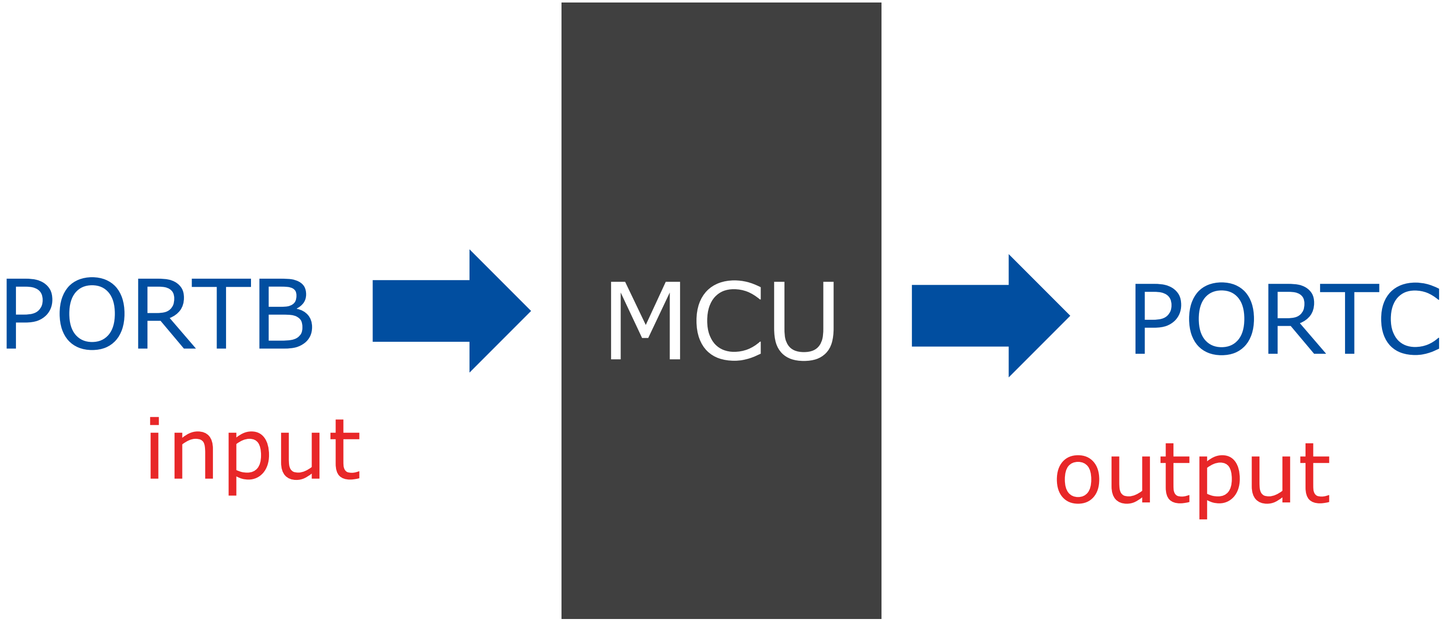
PORTB.0 = 1

```
BSF        TRISB, 0
```

PORTC.5 = 0

```
BCF        TRISC, 5
```

TRISx



PORTB

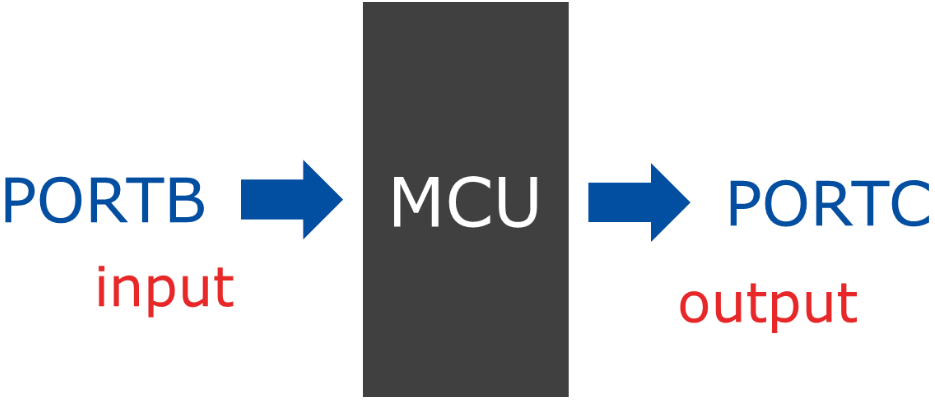
input

MCU

PORTC

output

TRISx



PORTB

input

MCU

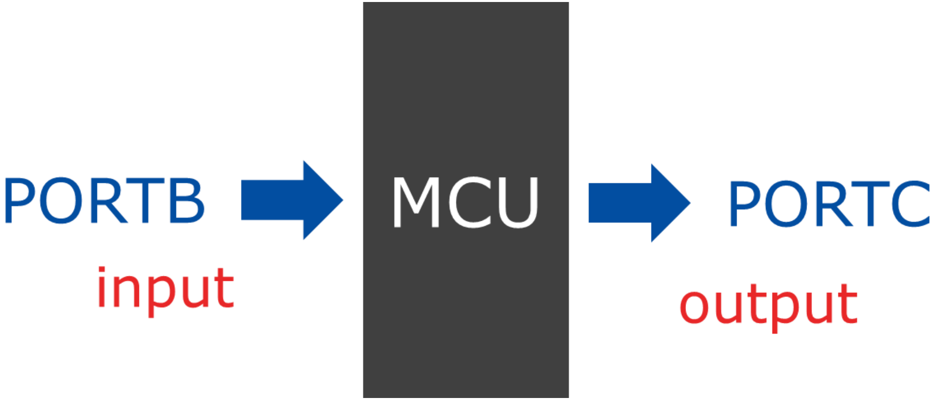
PORTC

output

0xFF -> TRISB

0x00 -> TRISC

TRISx

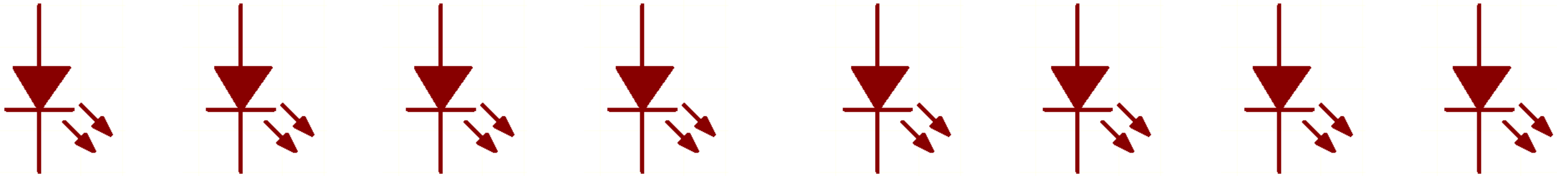


```
MOVLW 0xFF
MOVWF TRISB
CLRF TRISC
```

TURN LEDs ON AND OFF

PORTB

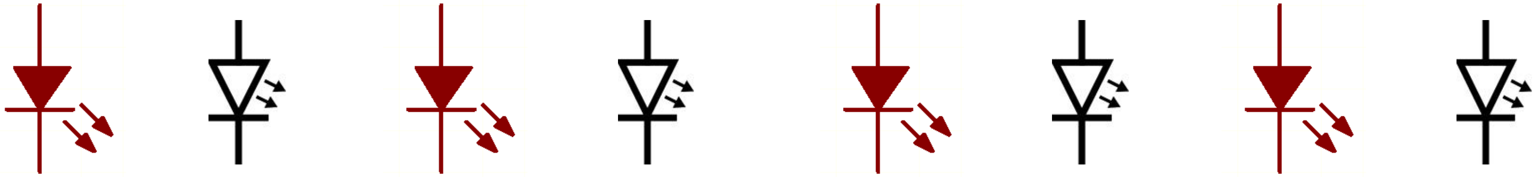
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



```
ORG 0x00
BSF     STATUS, 5
CLRF    TRISB
MAIN
BCF     STATUS, 5
MOVLW   0xFF
MOVWF   PORTB
END
```

PORTB

PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



```
ORG 0x00
BSF       STATUS, 5
CLRF      TRISB
MAIN
BCF       STATUS, 5
MOVLW     0xAA
MOVWF     PORTB
END
```

PORTC

PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
🔴	⚫	⚫	⚫	⚫	⚫	⚫	🔴

1

PORTA

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
🔴	⚫	⚫	🔴	🔴	⚫	⚫	🔴

PORTE

PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0
🔴	⚫	🔴	⚫	⚫	🔴	⚫	🔴

WRITE AN ASSEMBLY PROGRAM IN
ORDER TO TURN ON ONLY
THE FIRST THREE LEDS OF PORTB

WRITE AN ASSEMBLY PROGRAM IN ORDER TO TURN ON
THE EVEN BITS OF PORTA

WRITE AN ASSEMBLY PROGRAM IN ORDER TO TURN ON
THE ODD BITS OF PORTD

WRITE AN ASSEMBLY PROGRAM IN ORDER TO TURN ON ONLY THE LAST TWO LEDS OF PORTC

WRITE AN ASSEMBLY PROGRAM IN ORDER TO TURN ON THE EVEN BITS OF PORTB AND THE ODD BITS OF PORTC

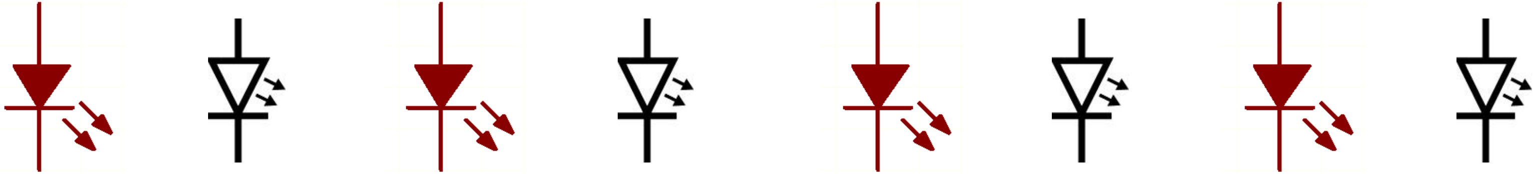
THANK YOU

CLASS 4

CLASS 3 REVIEW

PORTB

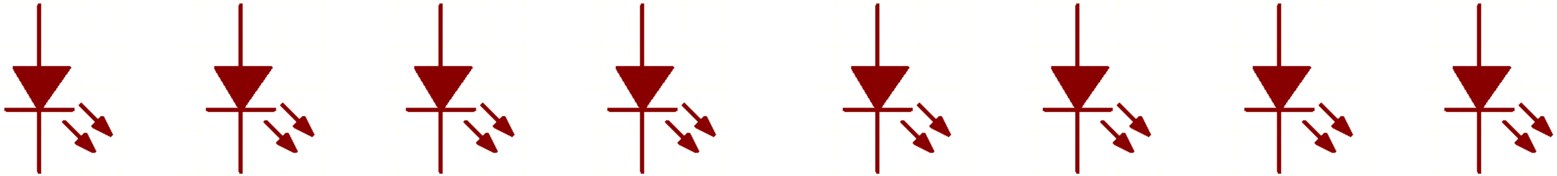
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



TURN LEDs ON AND OFF

PORTB

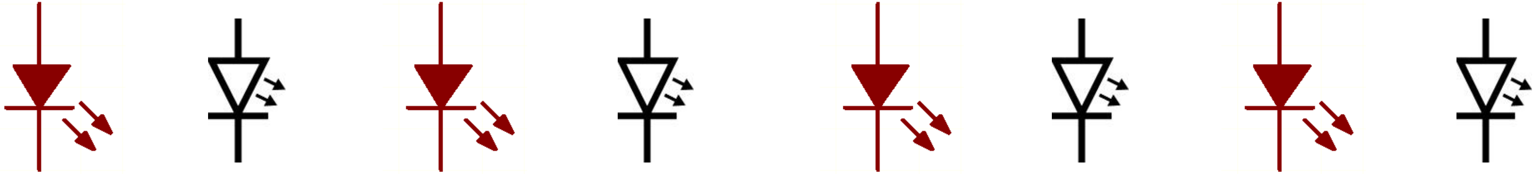
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



```
ORG 0x00
BSF     STATUS, 5
CLRF    TRISB
MAIN
BCF     STATUS, 5
MOVLW   0xFF
MOVWF   PORTB
END
```

PORTB

PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



```
ORG 0x00
BSF       STATUS, 5
CLRF      TRISB
MAIN
BCF       STATUS, 5
MOVLW     0xAA
MOVWF     PORTB
END
```

PORTC

PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
🔴	⚫	⚫	⚫	⚫	⚫	⚫	🔴

1

PORTA

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
🔴	⚫	⚫	🔴	🔴	⚫	⚫	🔴

PORTE

PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0
🔴	⚫	🔴	⚫	⚫	🔴	⚫	🔴

WRITE AN ASSEMBLY PROGRAM IN ORDER TO TURN ON ONLY THE FIRST THREE LEDS OF PORTB

WRITE AN ASSEMBLY PROGRAM IN ORDER TO TURN ON THE EVEN BITS OF PORTA

WRITE AN ASSEMBLY PROGRAM IN ORDER TO TURN ON THE ODD BITS OF PORTD

WRITE AN ASSEMBLY PROGRAM IN ORDER TO TURN ON ONLY THE LAST TWO LEDS OF PORTC

WRITE AN ASSEMBLY PROGRAM IN ORDER TO TURN ON THE EVEN BITS OF PORTB AND THE ODD BITS OF PORTC

CLASS 4

- TURN ON LEDs
- 7-SEGMENT DISPLAY
- BASICS OF CENTRAL PROCESSING UNIT

TURN LEDS ON AND OFF

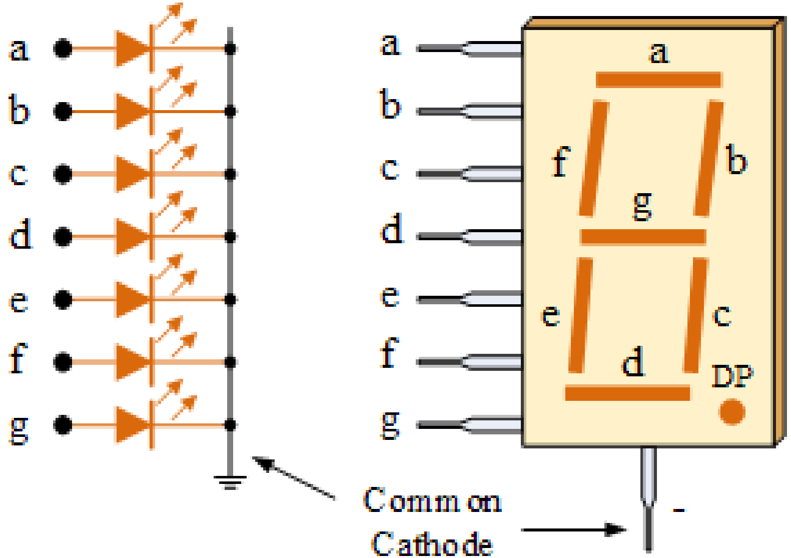
PORTA

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
🔴	⚫	⚫	🔴	🔴	⚫	⚫	🔴

```
ORG 0x00
BSF       STATUS, 5
CLRF      TRISA
MAIN
BCF       STATUS, 5
MOVLW     0x99
MOVWF     PORTA
END
```

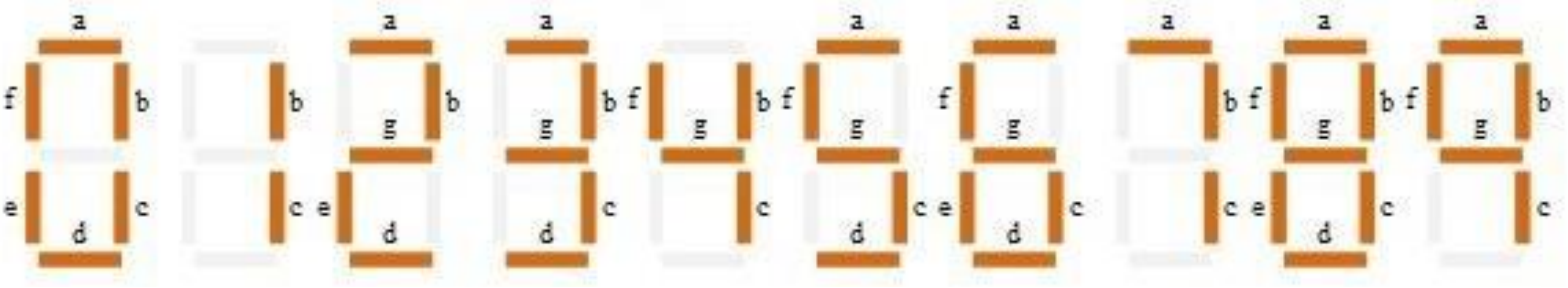
7-SEGMENT DISPLAY

7-SEGMENT DISPLAY



COMMON CATHODE

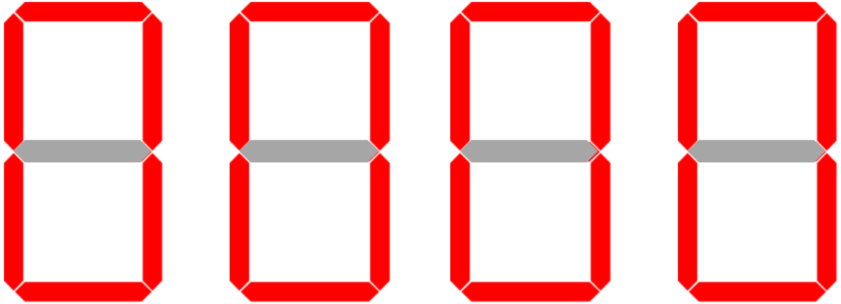
7-SEGMENT DISPLAY



7-SEGMENT DISPLAY

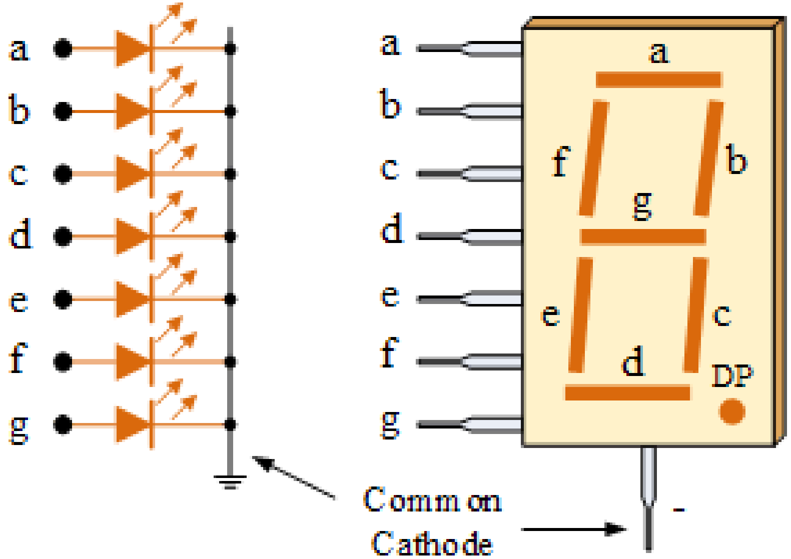
Decimal	DP	g	f	e	d	c	b	a	Hex
0	0	0	1	1	1	1	1	1	3F
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	0	1	1	0	0	1	1	1	67

7-SEGMENT DISPLAY



SHOW NUMBER “0”
IN ALL THE 7-SEG DISPLAYS

7-SEGMENT DISPLAY

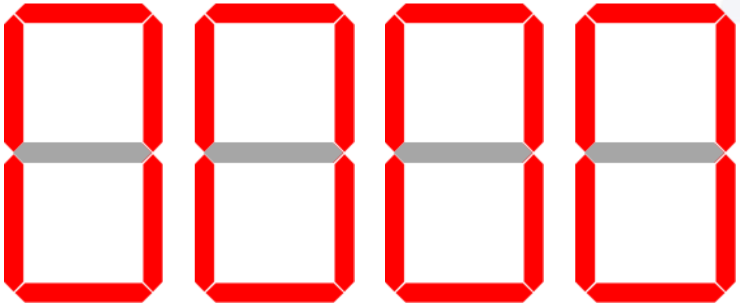


COMMON CATHODE

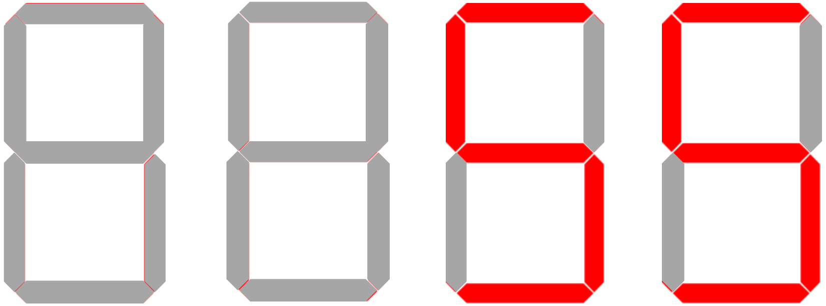
7-SEGMENT DISPLAY

Decimal	DP	g	f	e	d	c	b	a	Hex
0	0	0	1	1	1	1	1	1	3F
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	0	1	1	0	0	1	1	1	67

```
ORG 0x00
BSF STATUS, 5
CLRF TRISA
CLRF TRISD
MAIN
BCF STATUS, 5
MOVLW 0xFF
MOVWF PORTA
MOVLW 0x3F
MOVWF PORTD
END
```

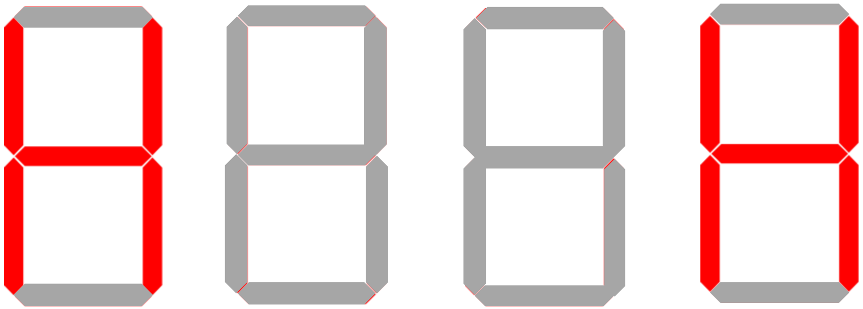


7-SEGMENT DISPLAY



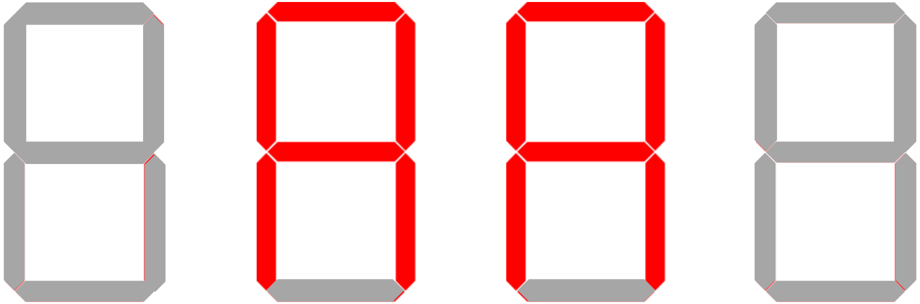
SHOW NUMBER “5”
IN THE FIRST TWO 7-SEG DISPLAYS

7-SEGMENT DISPLAY



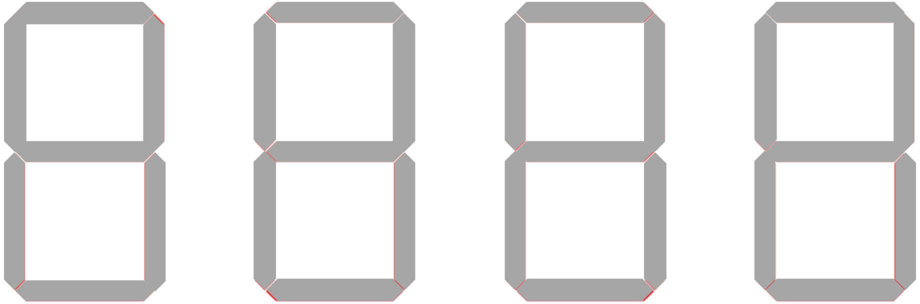
SHOW CHARACTER “H”
IN THE FIRST AND LAST 7-SEG DISPLAY

7-SEGMENT DISPLAY



SHOW CHARACTER “A”
IN THE SECOND AND THIRD 7-SEG DISPLAY

7-SEGMENT DISPLAY



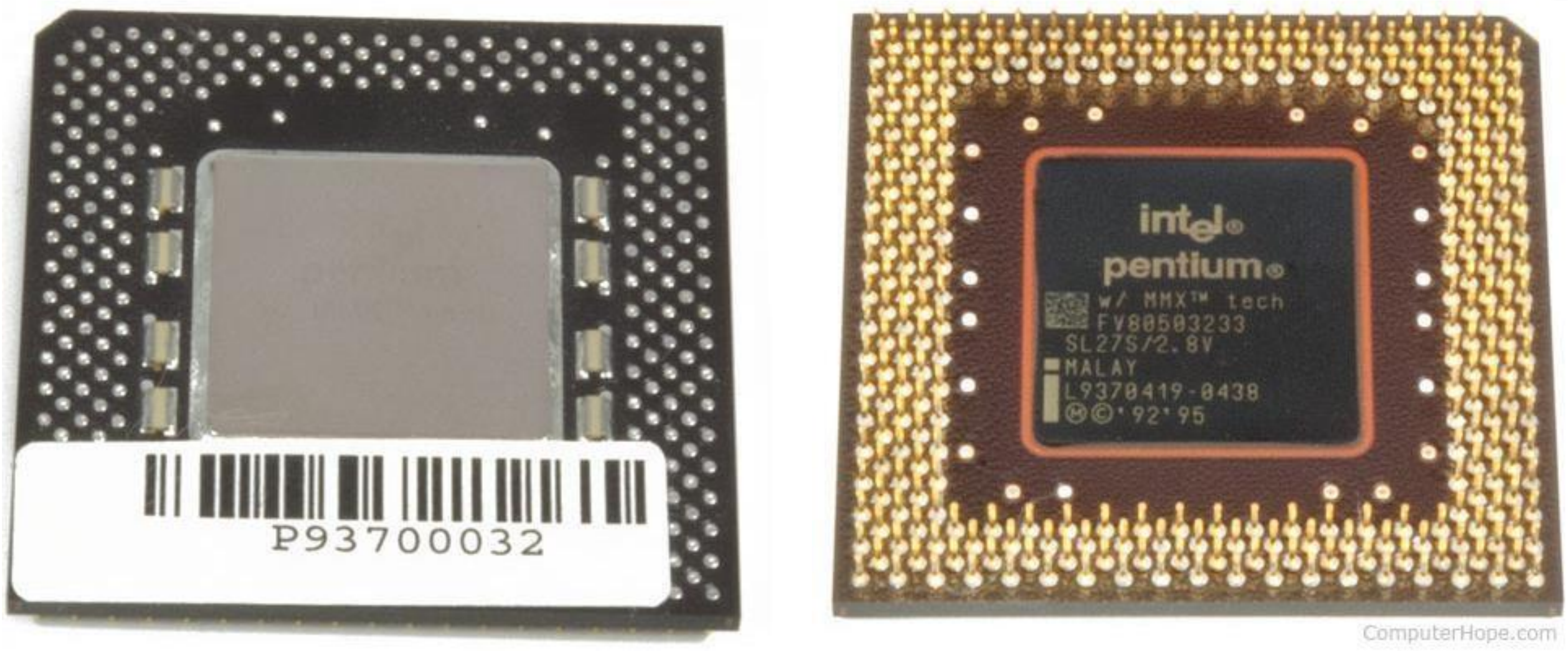
SHOW THE 7-SEG DISPLAY OUTPUT
IF PORTD=0xC9

CENTRAL PROCESSING UNIT

CPU

Also known as Microprocessor, handles all instructions it receives from hardware and software running on the computer

INTEL PENTIUM



ComputerHope.com

CENTRAL PROCESSING UNIT

- The processor is placed and secured into a compatible CPU socket found on the motherboard

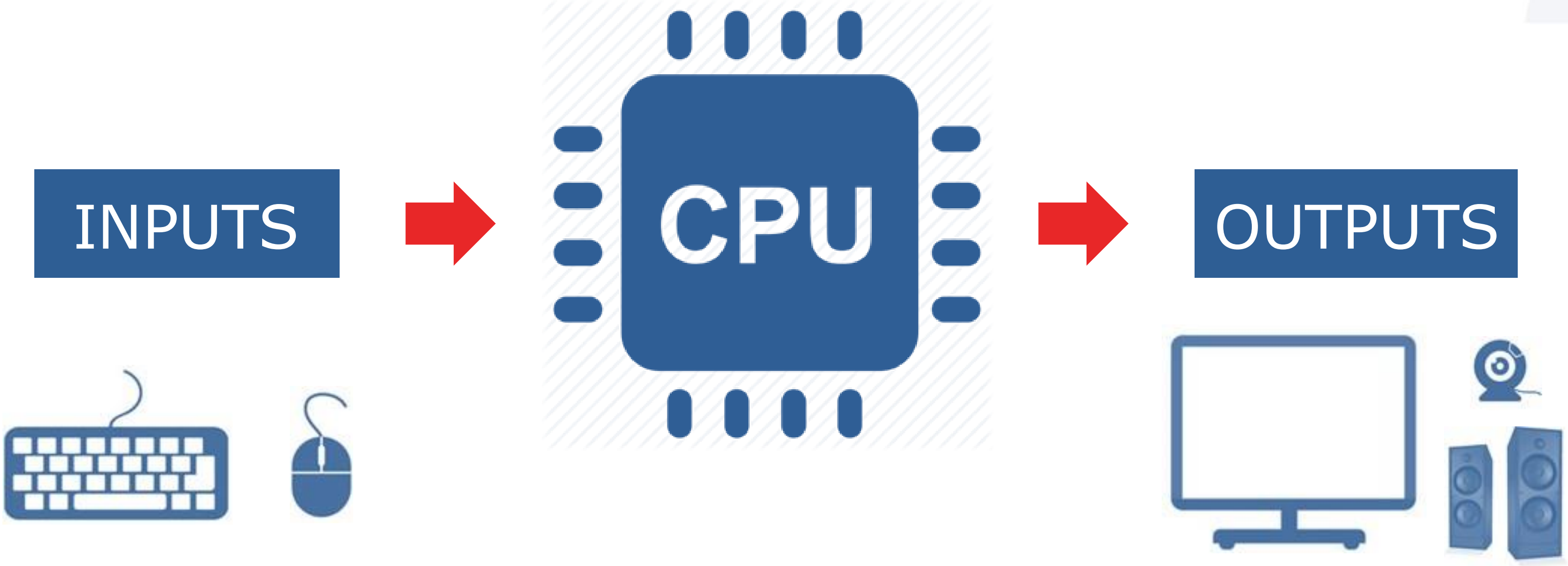
- Processors produce heat, so they are covered with a heat sink to keep them cool and running smoothly

WHAT DOES THE CPU DO?

WHAT DOES THE CPU DO?

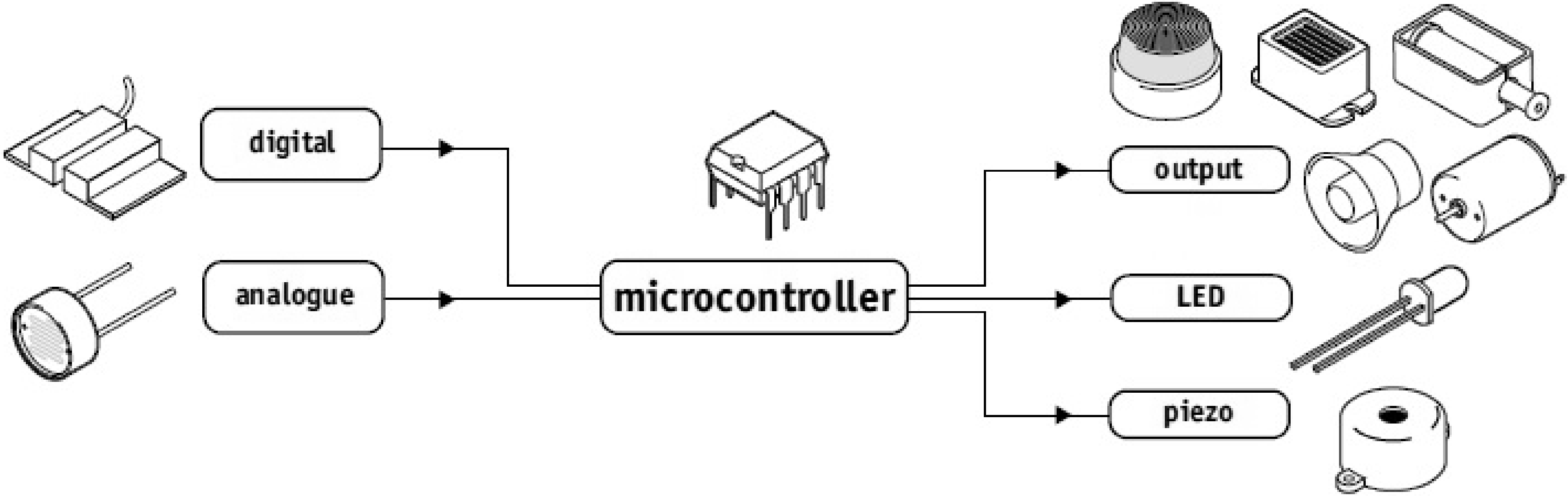
- Takes input from a peripheral (keyboard, mouse, printer, etc) or computer program
- Interprets what it needs
- Outputs information to your monitor, or perform the requested task

WHAT DOES THE CPU DO?



INPUTS → CPU → OUTPUTS

INPUT PROCESS OUTPUT



digital

analogue

microcontroller

output

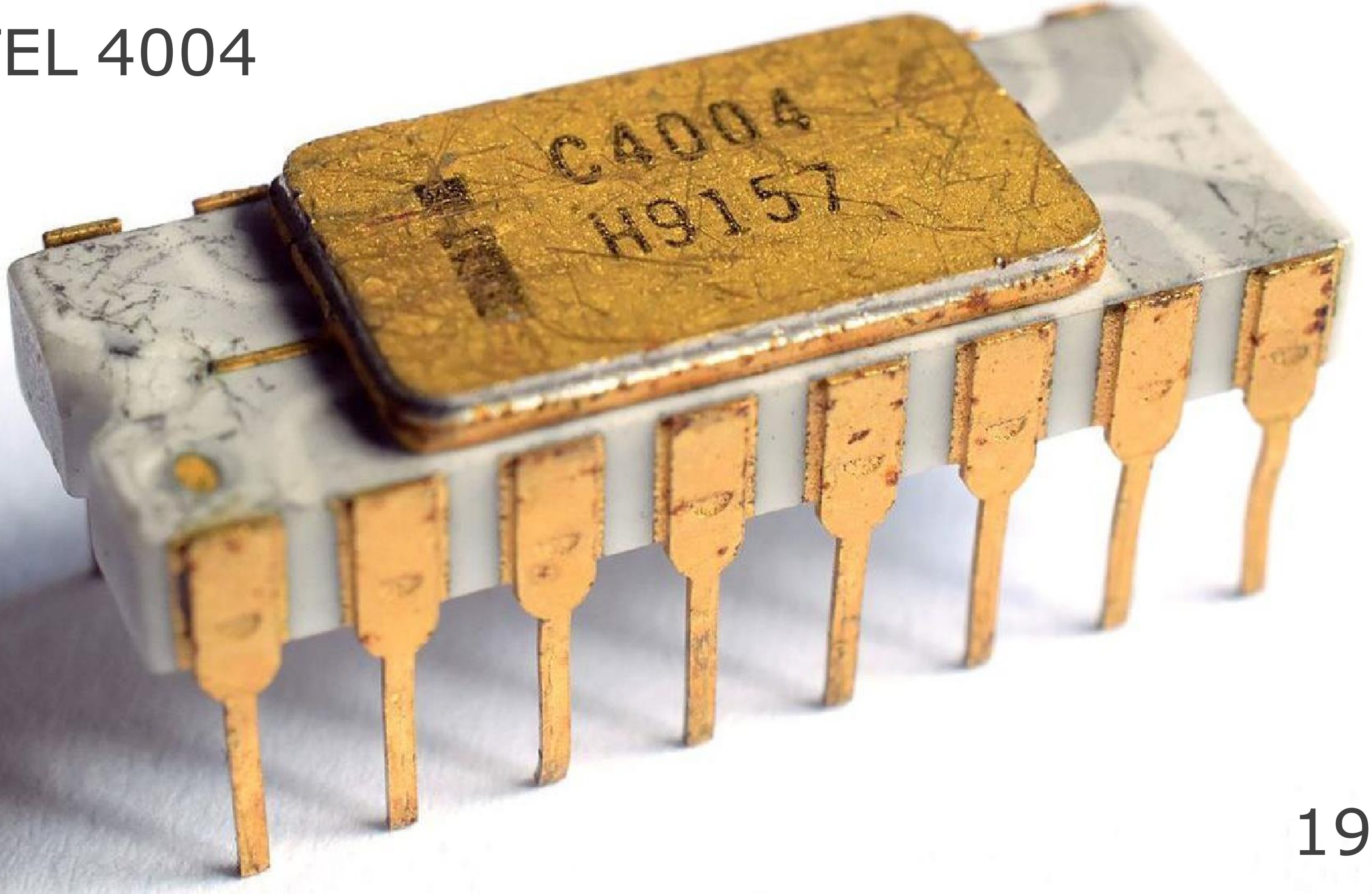
LED

piezo

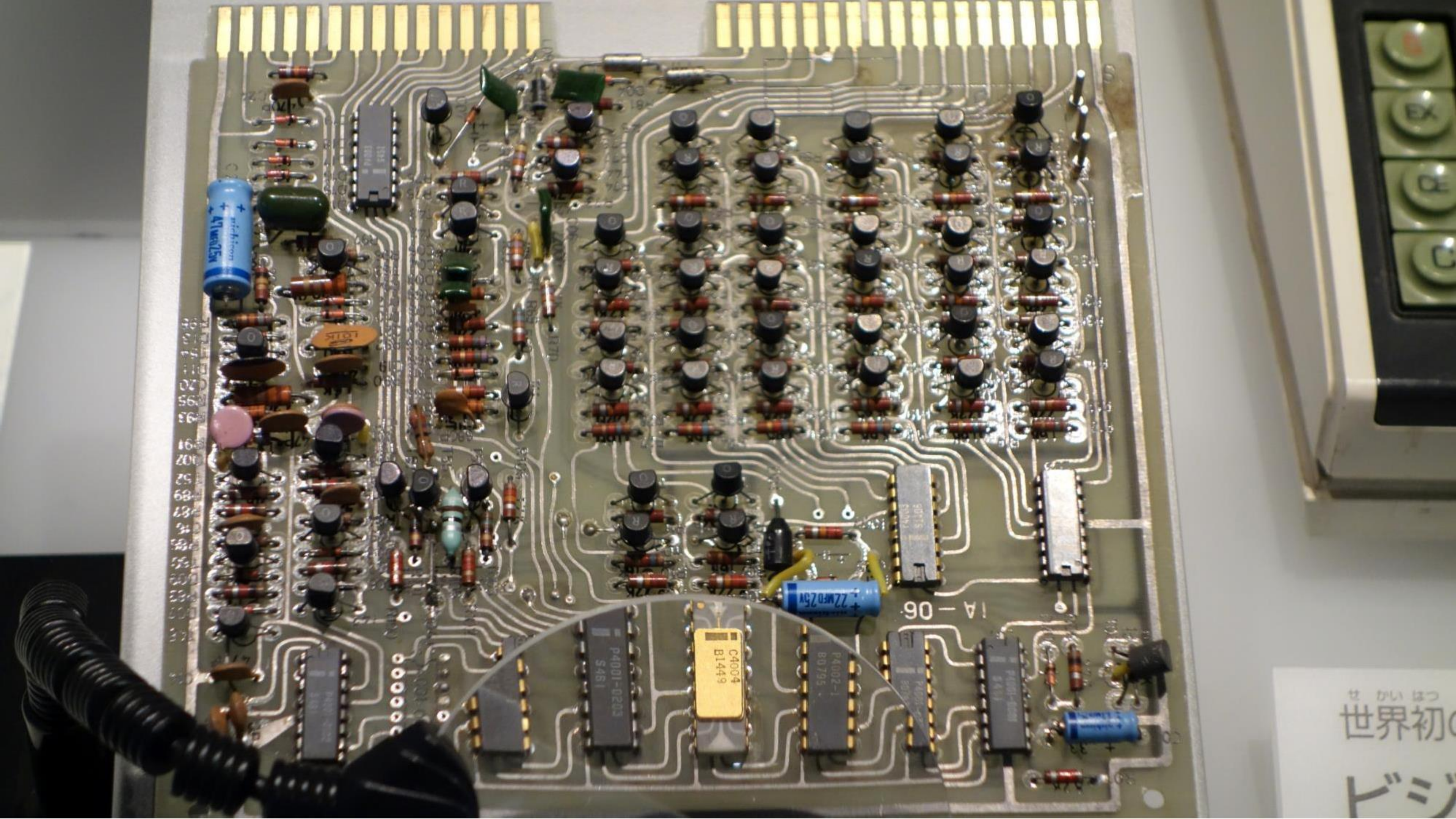
HISTORY

THE FIRST MICROPROCESSOR WAS THE INTEL 4004, INTRODUCED IN 1971

INTEL 4004



1971



C4004
B1449
P4001-0203
S461
P4002-1
80795
IA-06
22MFD25V
47MFD25V
nichicon
EX
CE
C
世界初の
ビジ

INTRODUCING ICE LAKE: 10NM CPU

2019

NEW SUNNYCOVE CORES

Up to 4 Cores / 8 Threads
Up to 4.1GHz

NEW CONVERGED CHASSIS FABRIC

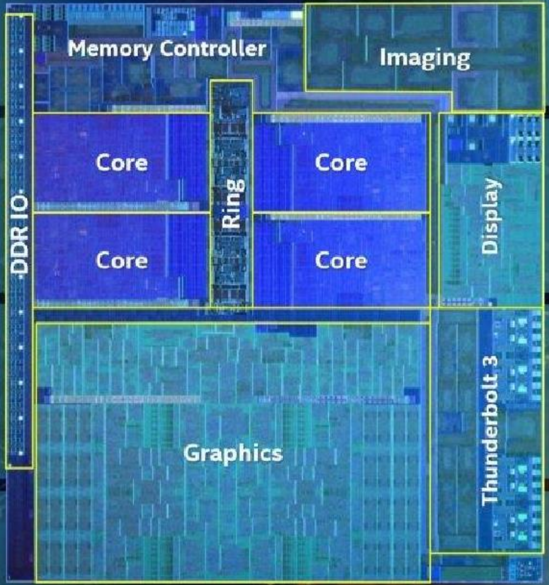
High Bandwidth / Low Latency
IP and Core Scalable

NEW MEMORY CONTROLLER

LP4/x-3733 4x32b up to 32GB
DDR4-3200 2x64b up to 64GB

FIRST INTEGRATED THUNDERBOLT™ 3

Full 4x DP/USB/PCIe mux on-die
Up to 40Gbps bi-directional per port



NEW GEN11 GRAPHICS

Up to 64EU and 1.1GHz
>1TFLOP

NEW 2X MEDIA ENCODERS

Up to 4K60 10b 4:4:4
Up to 8K30 10b 4:2:0

NEW 3X DISPLAY PIPES

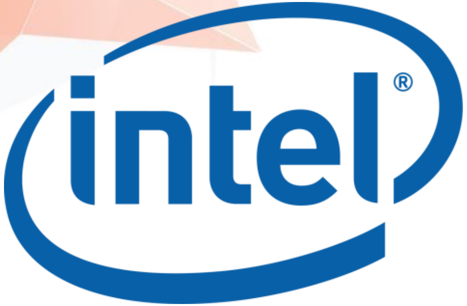
Up to 5K60 or 4K120
DP1.4, BT.2020

NEW IMAGE PROCESSING UNIT 4

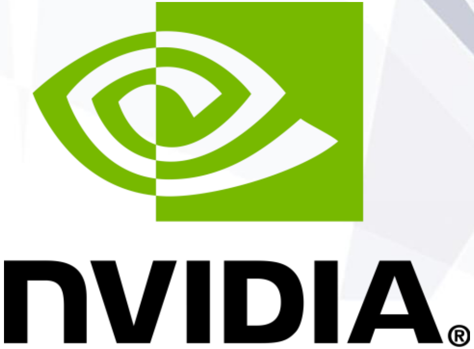
Up to 16MP
Up to 1080p120, 4K30

Intel Confidential. Internal Use Only - Embargoed until 11 p.m. PT on May 27.

Intel Core i9-9900KS Processor
18 Billion Transistors


intel®

AMD


NVIDIA®

QUALCOMM®

tsmc
台灣積體電路製造股份有限公司
Taiwan Semiconductor Manufacturing Company, Ltd.

ATMEL

®
MICROCHIP

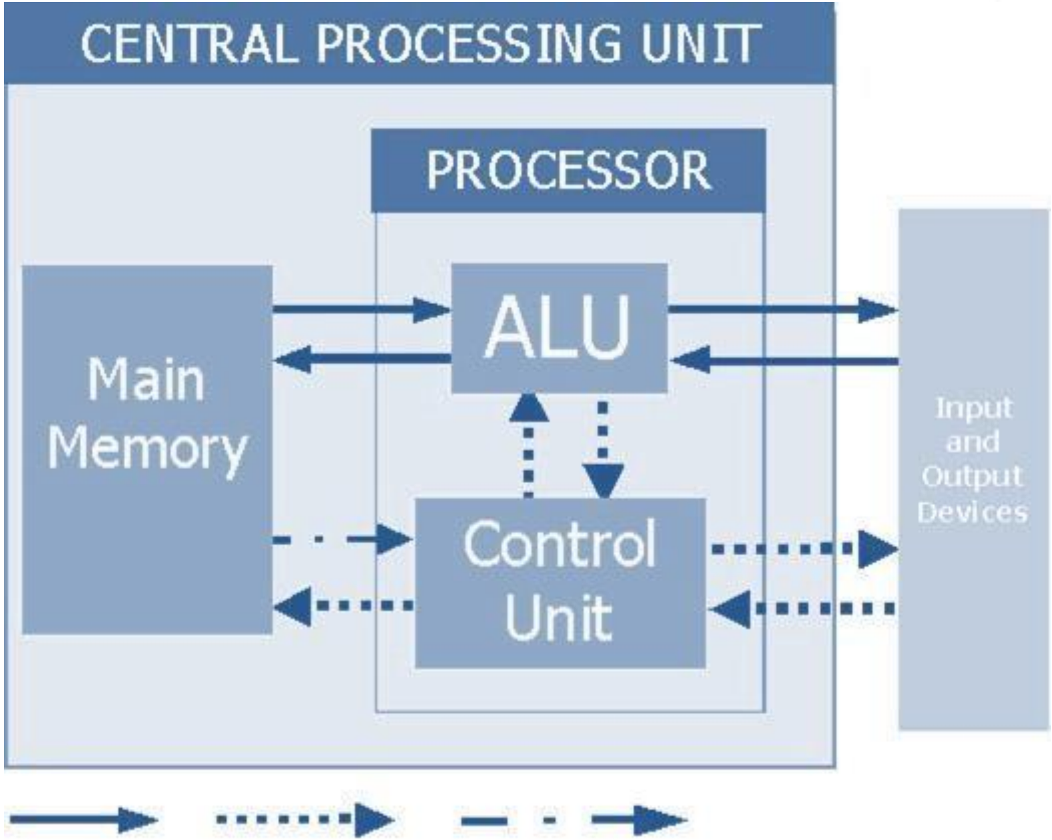
COMPONENTS OF THE CPU

COMPONENTS OF THE CPU

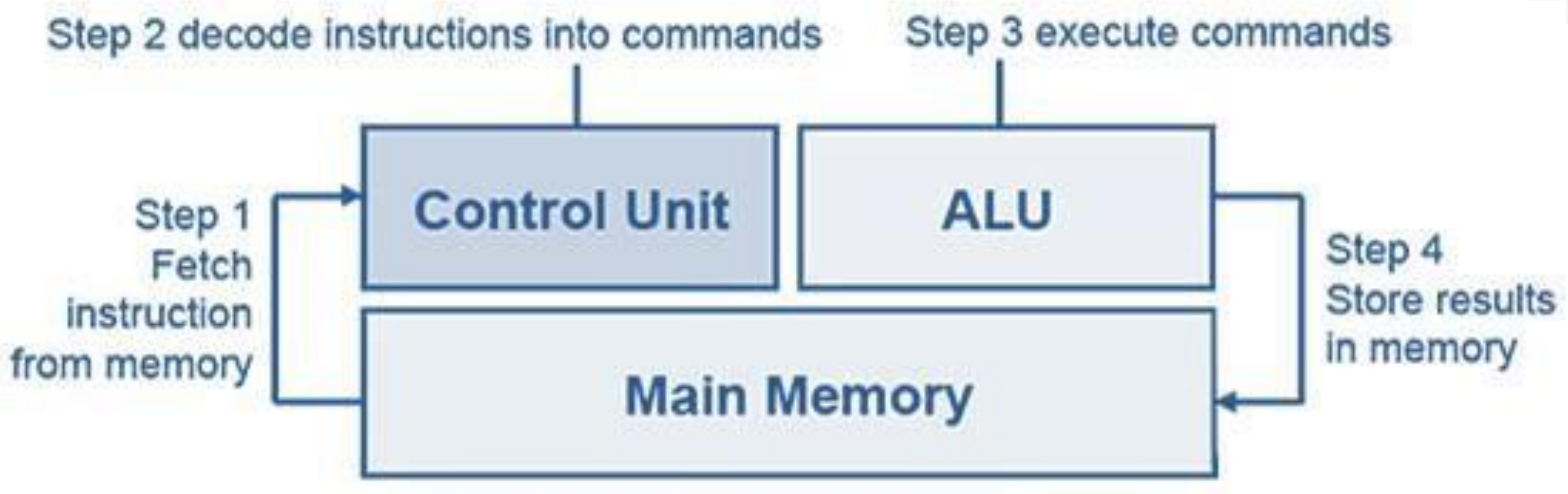
The primary components are:

- The ALU (Arithmetic Logic Unit) that performs mathematical, logical, and decision operations and
- The CU (Control Unit) that directs all of the processors operations.

COMPONENTS OF THE CPU



MACHINE CYCLE


Step 2 decode instructions into commands
Step 3 execute commands
Step 1 Fetch instruction from memory
Control Unit
ALU
Step 4 Store results in memory
Main Memory

CONTROL UNIT

The control unit has the task of decoding the instructions, interpreting them by generating the appropriate signals to be sent to the executing organs at the clock pulse rate

CONTROL UNIT

The control unit's activity is generally divided into three main phases:

- Fetch
- Decode
- Execute

ARITHMETIC LOGIC UNIT

The arithmetic-logic unit is formed by a set of circuits capable of performing elementary arithmetic operations such as addition, subtraction, increment, decay, multiplication, division, data exchange between registers and control operations

HOW FAST DOES A CPU TRANSFER DATA?

CPU TRANSFER DATA SPEED

Like any device that utilizes electrical signals, the data travels very near the speed of light, which is approximately 300,000,000 m/s

CPU TRANSFER DATA SPEED

- This speed depends on the medium (type of metal in the wire) through which the signal is traveling.
- Most electrical signals are traveling at about 75 to 90% the speed of light.

CPU CLOCK SPEED

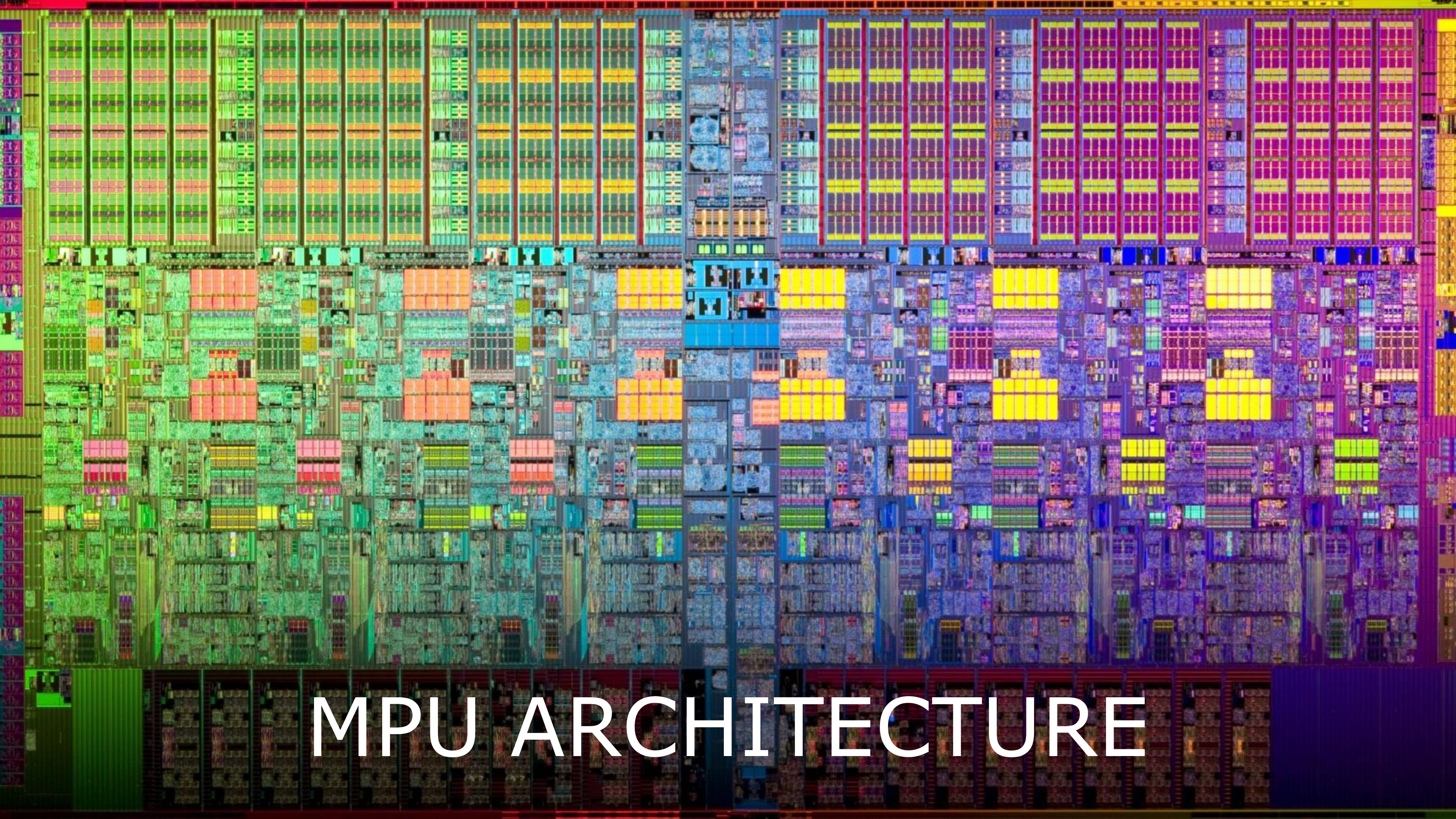
The clock speed of a CPU is the number of instructions it can process in any given second, measured in gigahertz (GHz)

CPU CLOCK SPEED

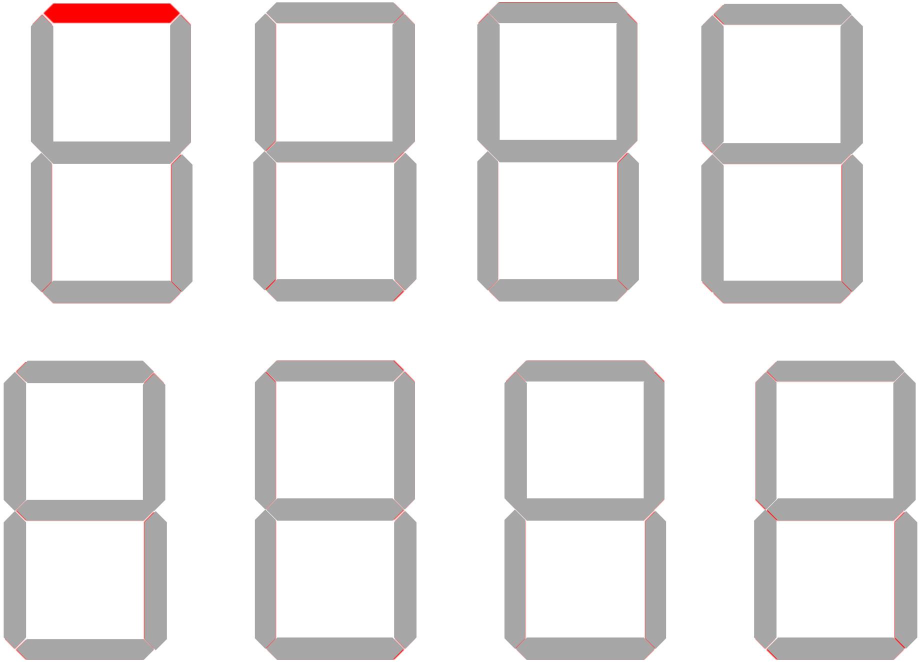
For example, a CPU has a clock speed of 1 Hz if it can process one piece of instruction every second. Extrapolating this to a more real-world example: a CPU with a clock speed of 3.0 GHz can process 3 billion instructions each second

CPU CORES

Some devices have a single-core processor while others may have a dual-core (or quad-core, etc.) processor. As might already be apparent, having two processor units working side by side means that the CPU can simultaneously manage twice the instructions every second, drastically improving performance



MPU ARCHITECTURE



THANK YOU

CLASS 5

Quiz review

CLASS – WEEK 4

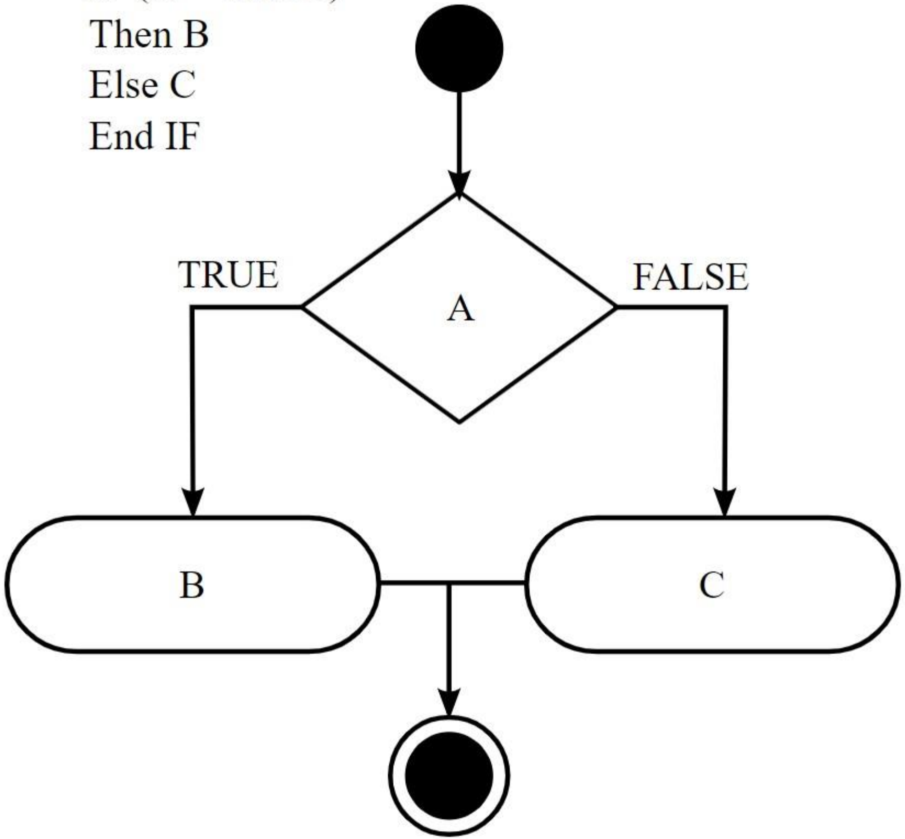
ASSEMBLY LANGUAGE

- IF STATEMENT
- READ INPUTS

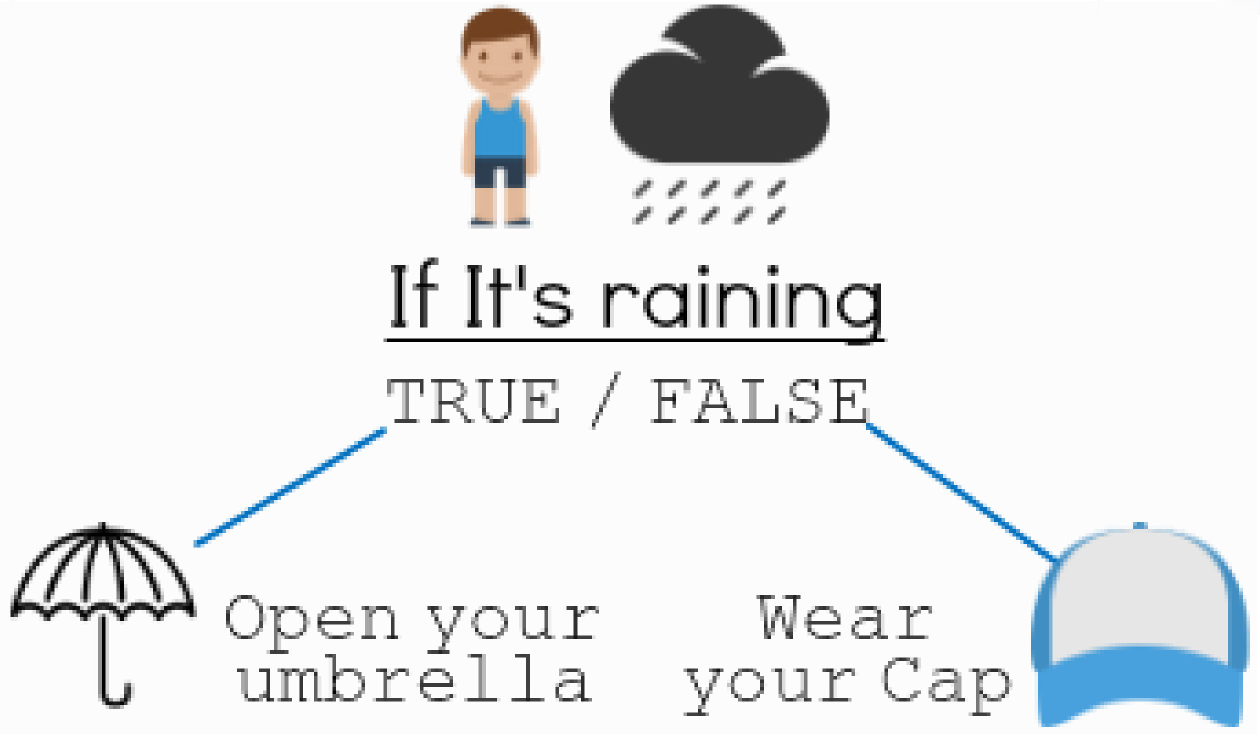
IF STATEMENT

IF STATEMENT

```
IF (A = TRUE)
Then B
Else C
End IF
```



IF STATEMENT



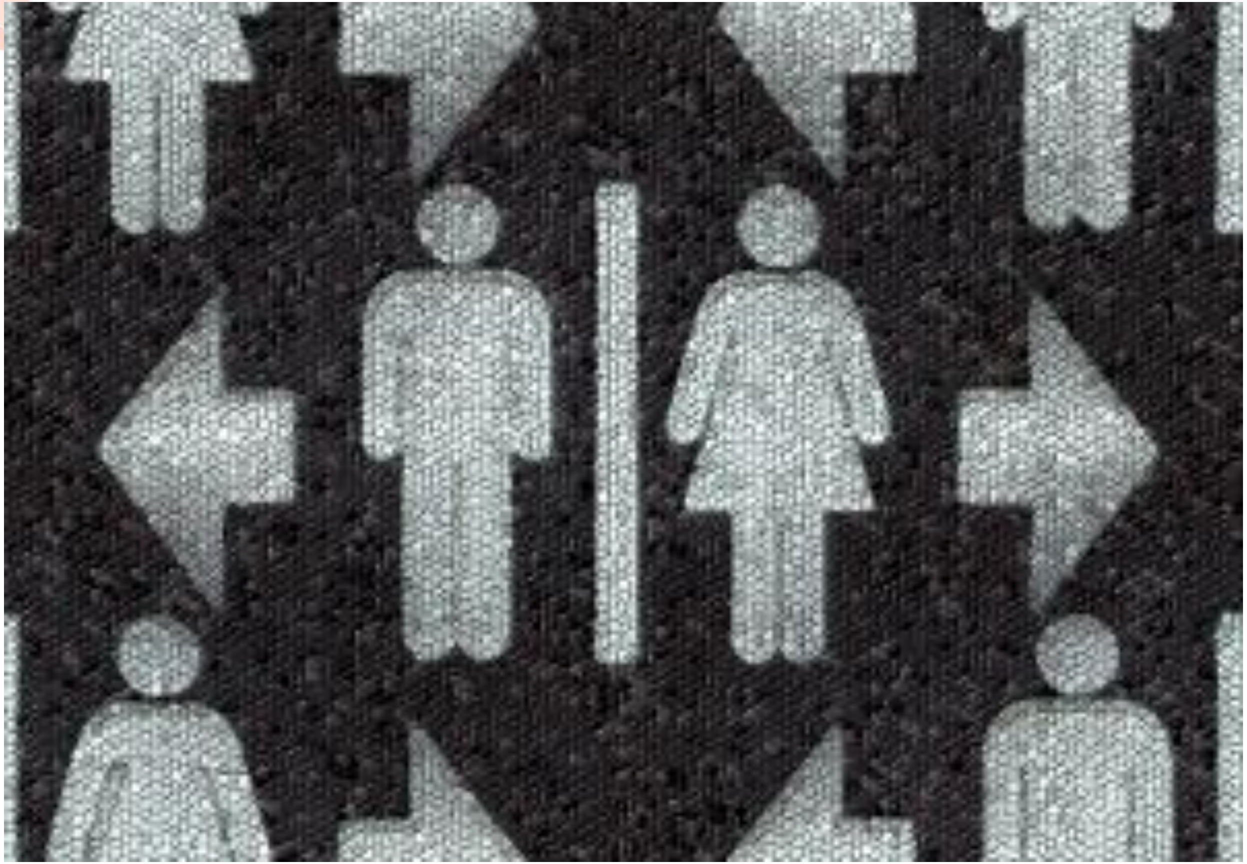
If It's raining

TRUE / FALSE

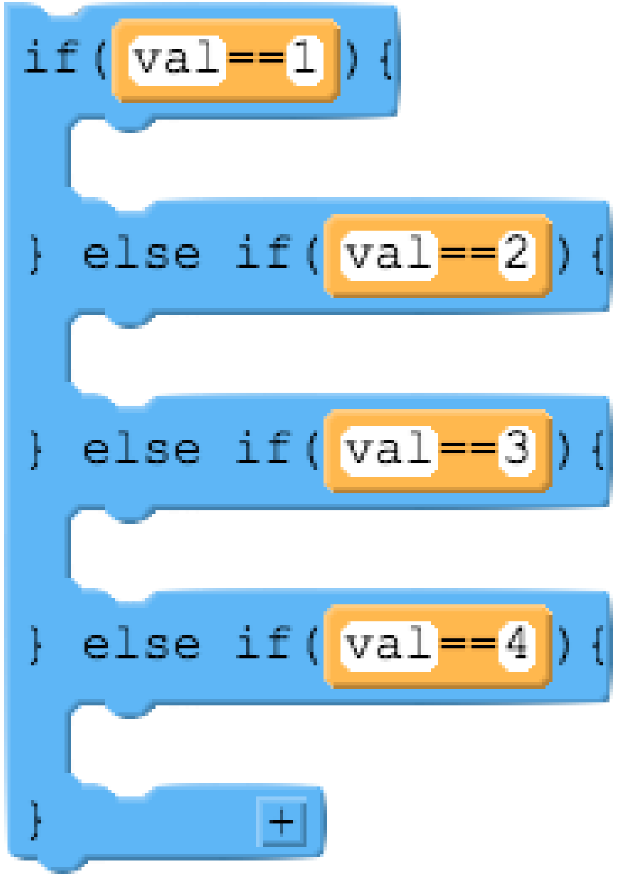
Open your umbrella

Wear your Cap

IF STATEMENT

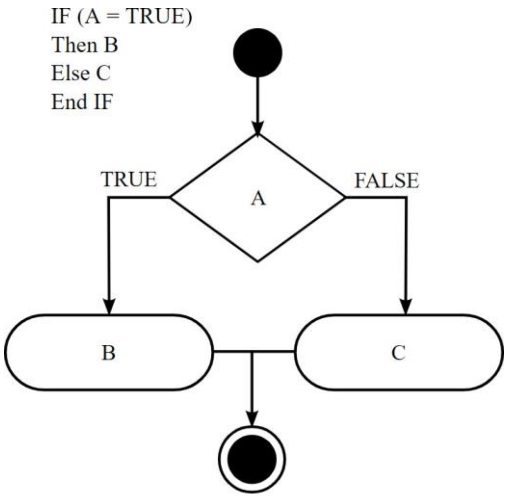


IF STATEMENT



```
if(val==1){
} else if(val==2){
} else if(val==3){
} else if(val==4){
}
```

IF STATEMENT IN ASSEMBLY



```
IF (A = TRUE)
Then B
Else C
End IF
```

IF STATEMENT IN ASSEMBLY

BTFSC (Bit Test File Skip if Clear)

BTFSS (Bit Test File Skip if Set)

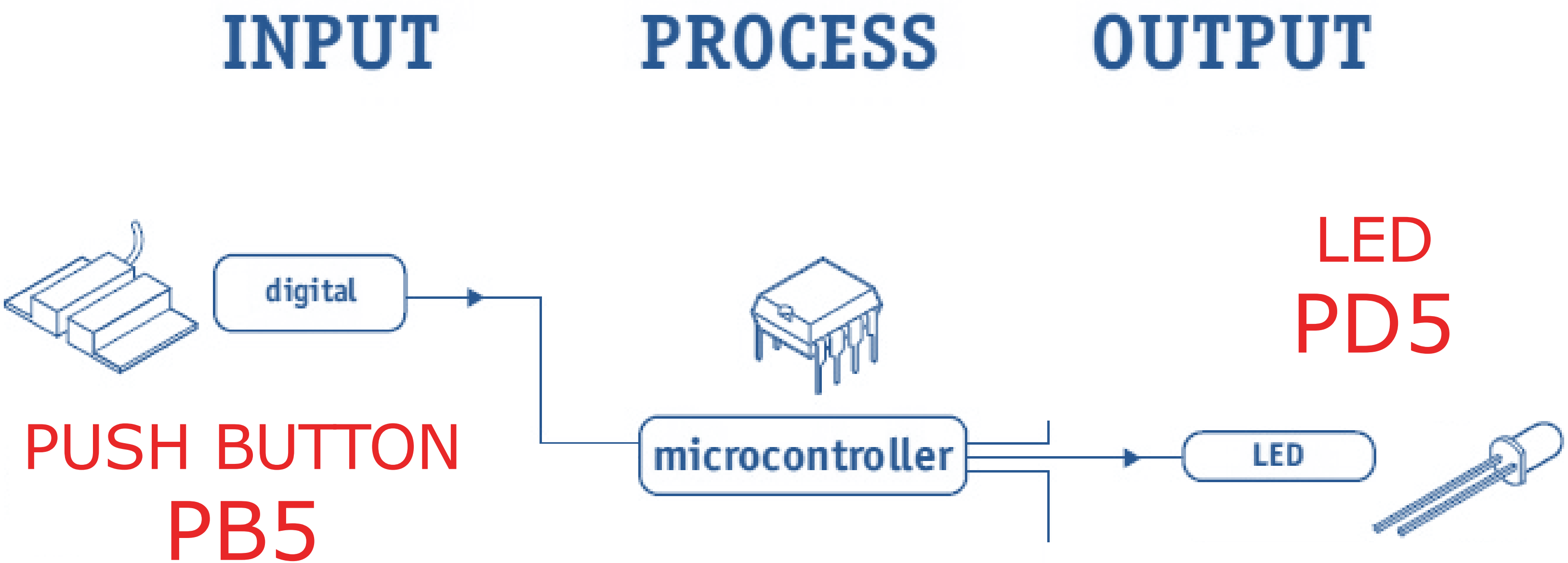
BTFSC (Bit Test File Skip if Clear)

IF THE LOGIC AT LOCATION F IS HIGH (1), THEN THE BTFSC FUNCTION WILL NOT SKIP THE NEXT LINE OF CODING

BTFSS (Bit Test File Skip if Set)

IF THE LOGIC AT LOCATION F IS HIGH (1), THEN THE BTFSS FUNCTION WILL SKIP THE NEXT LINE OF CODING

MCU SYSTEM



PUSH BUTTON
PB5

LED
PD5

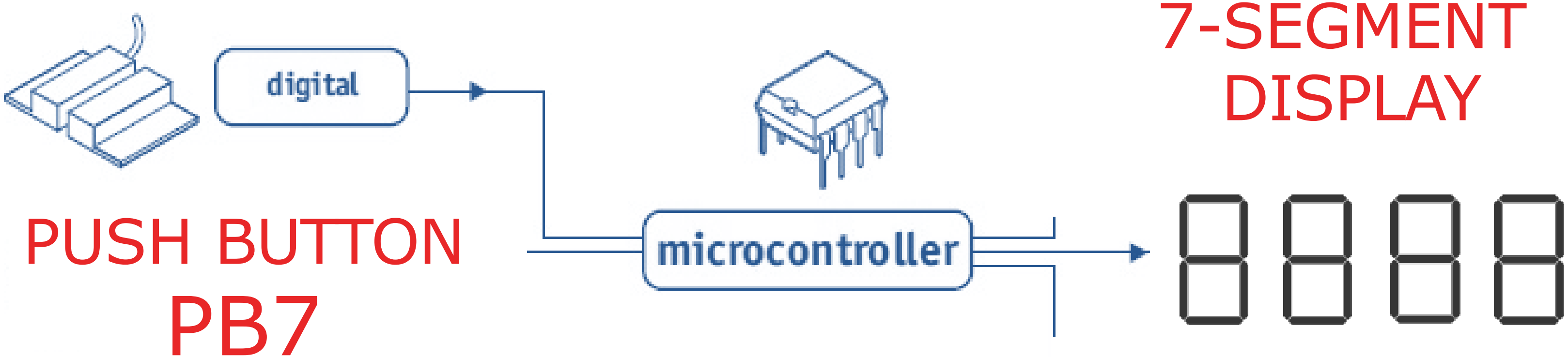
```
ORG 0x00
BSF STATUS, 5
BSF TRISB, 5
CLRF TRISD
MAIN
BCF STATUS, 5
BTFSC PORTB,5
GOTO LEDON
GOTO LEDOFF

LEDON
BSF PORTD,5
GOTO MAIN

LEDOFF
BCF PORTD, 5
GOTO MAIN
END
```

```
ORG 0x00
BSF STATUS, 5
BSF TRISB, 5
CLRF TRISD
MAIN
BCF STATUS, 5
BTFSS PORTB, 5
GOTO LEDOFF
GOTO LEDON
```

```
LEDON
BSF PORTD, 5
GOTO MAIN

LEDOFF
BCF PORTD, 5
GOTO MAIN
END
```

MCU SYSTEM

INPUT PROCESS OUTPUT



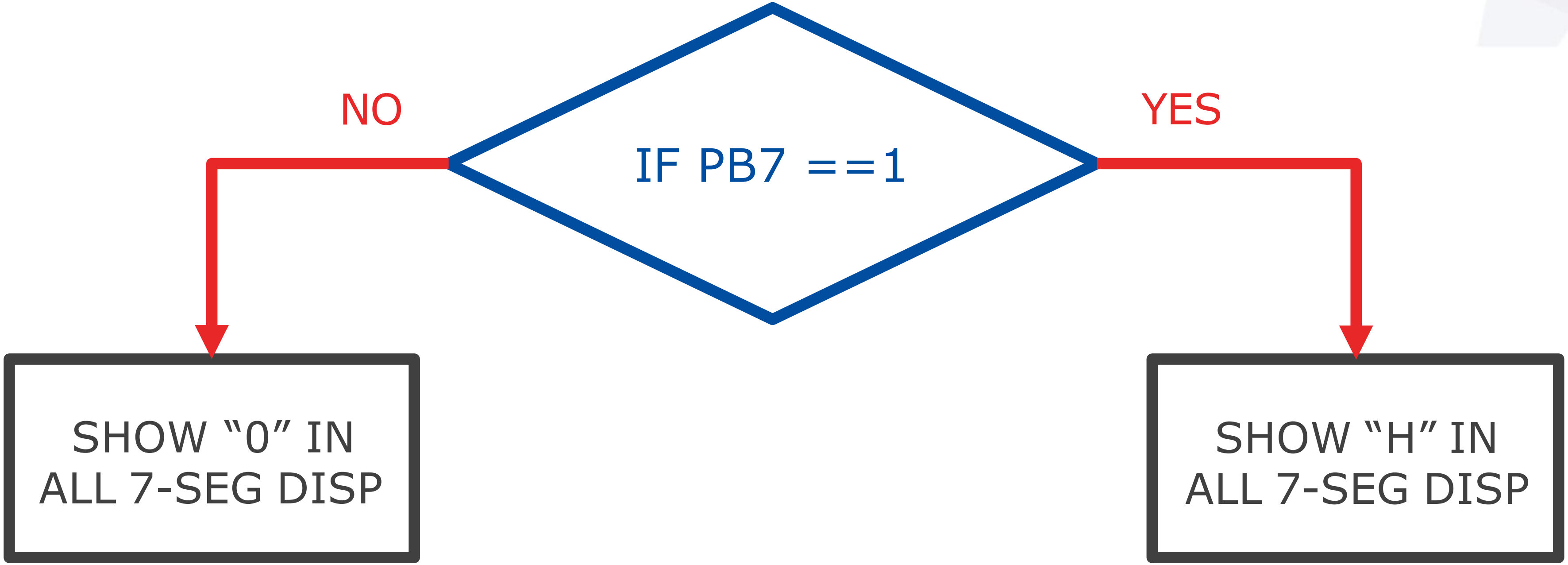
digital

microcontroller

PUSH BUTTON
PB7

7-SEGMENT
DISPLAY

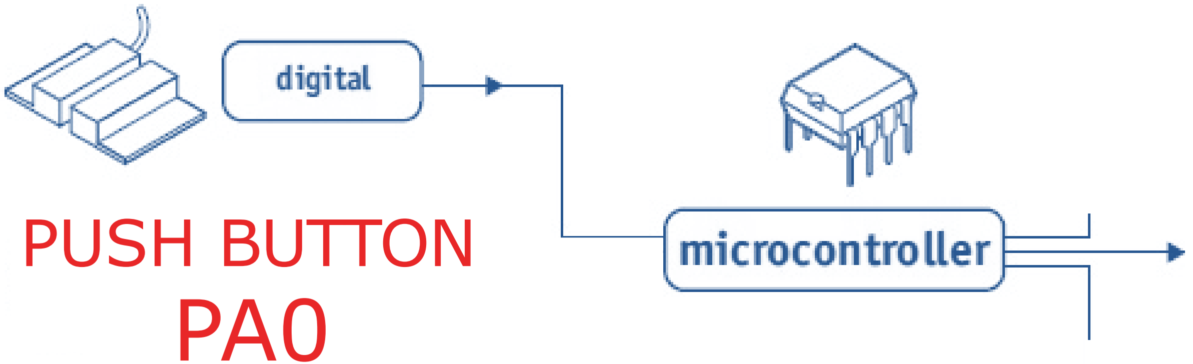
IF STATEMENT IN ASSEMBLY



```
ORG 0x00
BSF STATUS, 5
BSF TRISB, 7
CLRF TRISA
CLRF TRISD
MAIN
BCF STATUS, 5
BTFSC PORTB, 7
  GOTO SHOWH
GOTO SHOWZ
SHOWH
    MOVLW  0x0F
    MOVWF PORTA
    MOVLW  0x76
    MOVWF PORTD
GOTO MAIN
SHOWZ
    MOVLW  0x0F
    MOVWF PORTA
    MOVLW  0x3F
    MOVWF PORTD
GOTO MAIN
END
```

MCU SYSTEM

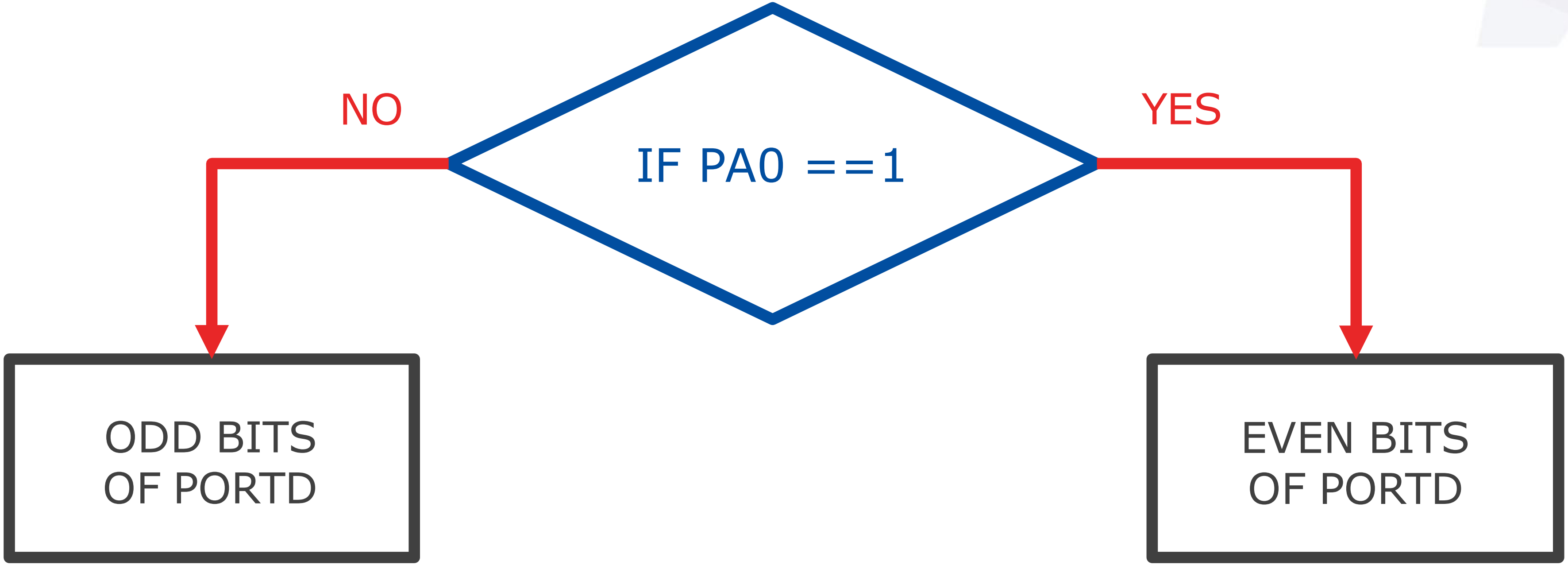
INPUT PROCESS OUTPUT



PUSH BUTTON
PA0

LEDS ON
PORTD

IF STATEMENT IN ASSEMBLY



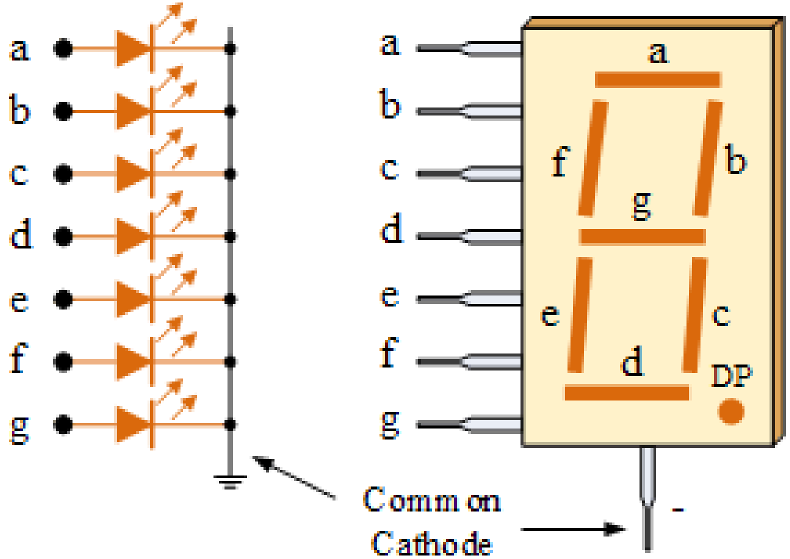
THANK YOU

CLASS 6

CLASS 5 REVIEW

7-SEGMENT DISPLAY

7-SEGMENT DISPLAY



COMMON CATHODE

7-SEGMENT DISPLAY

SHOW CHARACTER “C”
IN THE FOURTH AND SECOND 7-SEG DISPLAYS

7-SEGMENT DISPLAY

SHOW THE 7-SEG DISPLAY OUTPUT
IF PORTA=0x09 AND PORTD=0x6B

CLASS – WEEK 6

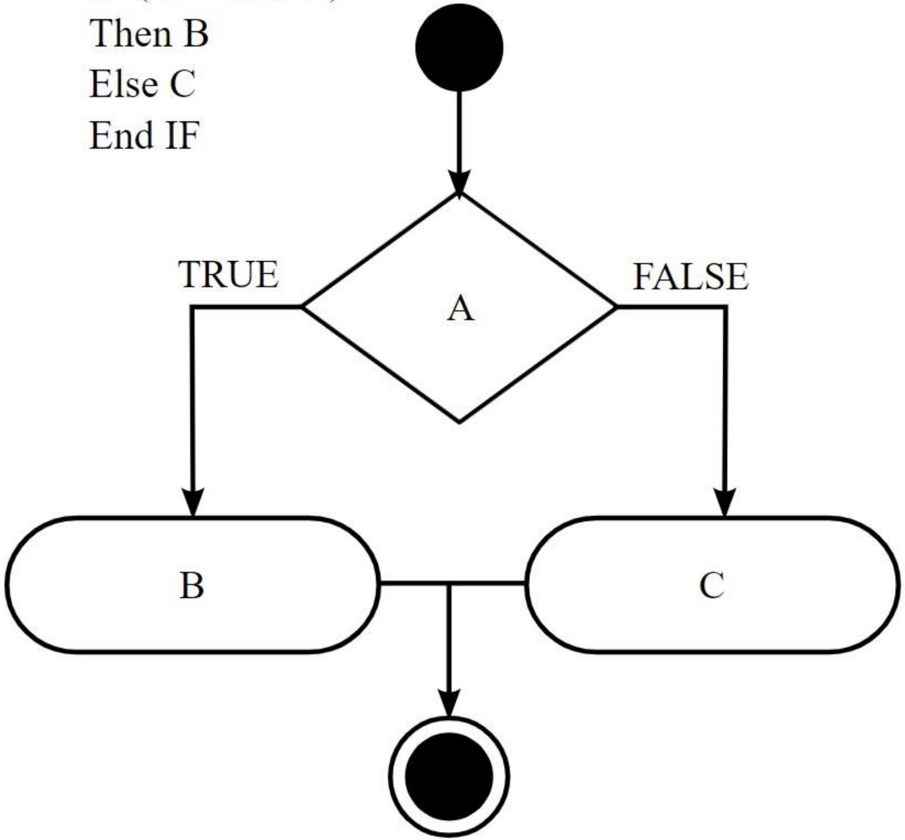
ASSEMBLY LANGUAGE

- IF STATEMENT
- READ INPUTS
- FUNCTIONS

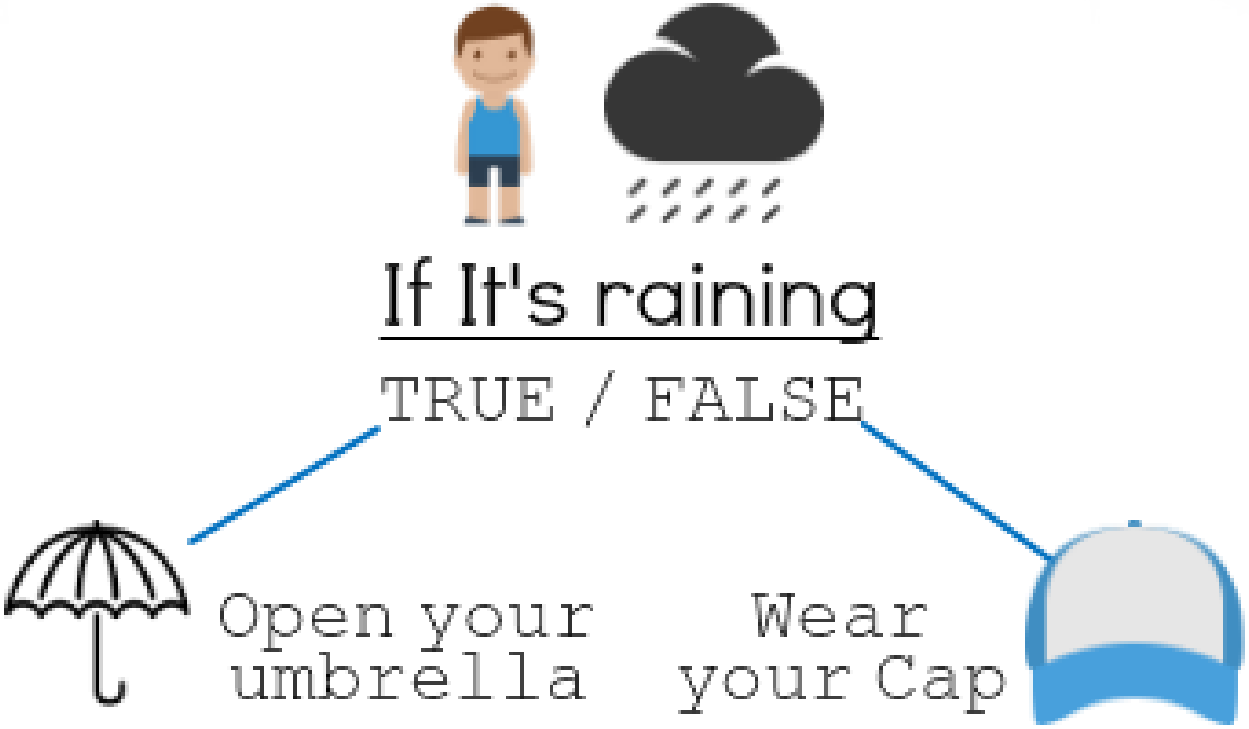
IF STATEMENT

IF STATEMENT

```
IF (A = TRUE)
Then B
Else C
End IF
```



IF STATEMENT



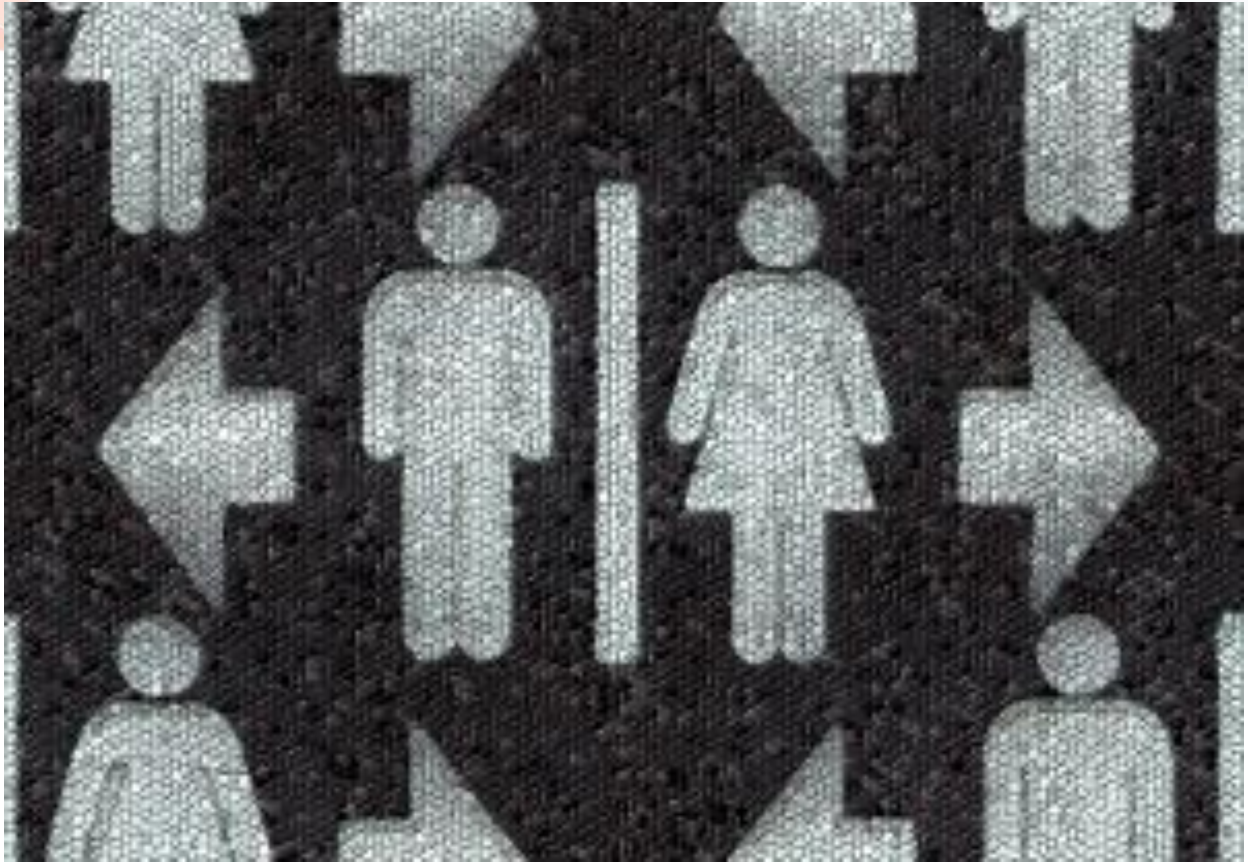
If It's raining

TRUE / FALSE

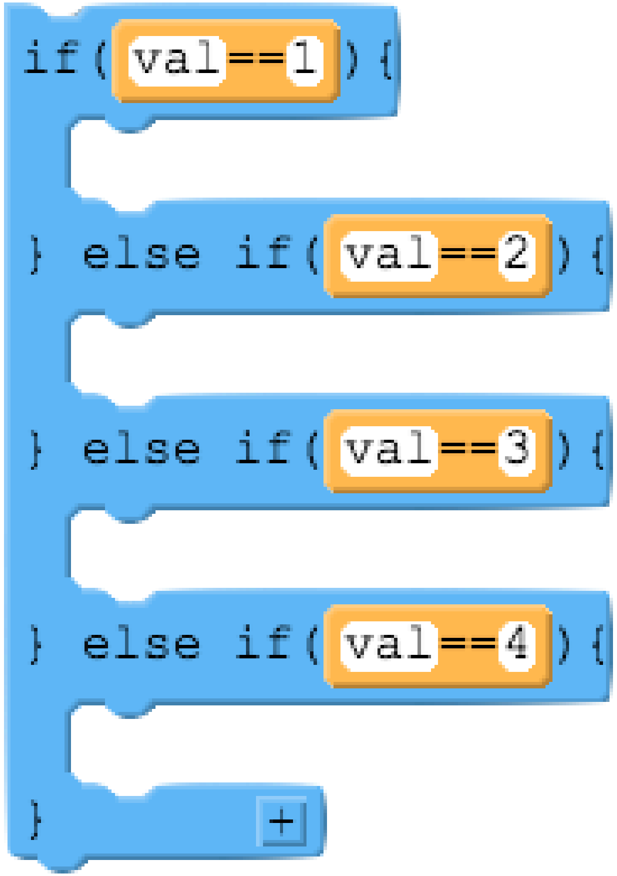
Open your umbrella

Wear your Cap

IF STATEMENT

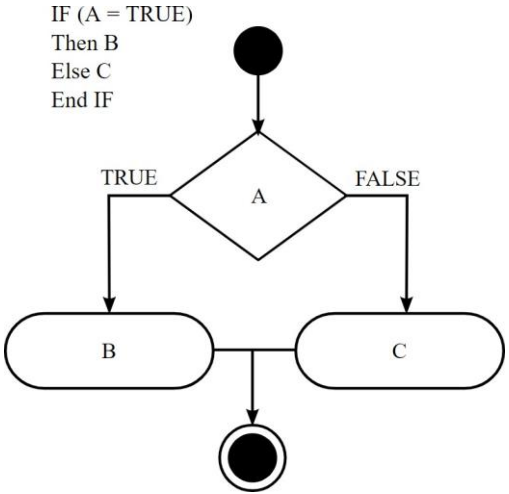


IF STATEMENT



```
if(val==1){

} else if(val==2){

} else if(val==3){

} else if(val==4){

}
```

IF STATEMENT IN ASSEMBLY



```
IF (A = TRUE)
Then B
Else C
End IF
```

IF STATEMENT IN ASSEMBLY

BTFSC (Bit Test File Skip if Clear)

BTFSS (Bit Test File Skip if Set)

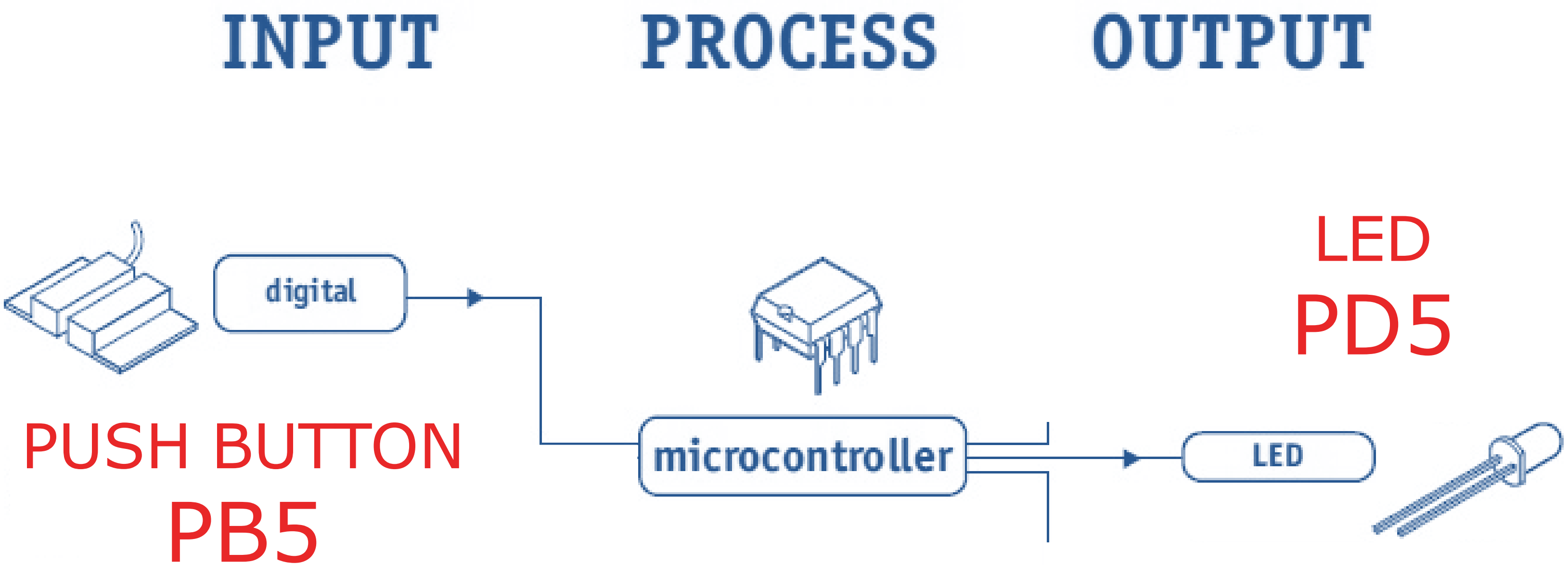
BTFSC (Bit Test File Skip if Clear)

IF THE LOGIC AT LOCATION F IS HIGH (1), THEN THE BTFSC FUNCTION WILL NOT SKIP THE NEXT LINE OF CODING

BTFSS (Bit Test File Skip if Set)

IF THE LOGIC AT LOCATION F IS HIGH (1), THEN THE BTFSS FUNCTION WILL SKIP THE NEXT LINE OF CODING

MCU SYSTEM



PUSH BUTTON
PB5

LED
PD5

```
ORG 0x00
BSF STATUS, 5
BSF TRISB, 5
CLRF TRISD
MAIN
BCF STATUS, 5
BTFSC PORTB, 5
  GOTO LEDON
GOTO LEDOFF
```

```
LEDON
  BSF PORTD, 5
GOTO MAIN

LEDOFF
  BCF PORTD, 5
GOTO MAIN
END
```

```
ORG 0x00
BSF STATUS, 5
BSF TRISB, 5
CLRF TRISD
MAIN
BCF STATUS, 5
BTFSS PORTB, 5
  GOTO LEDOFF
GOTO LEDON
```

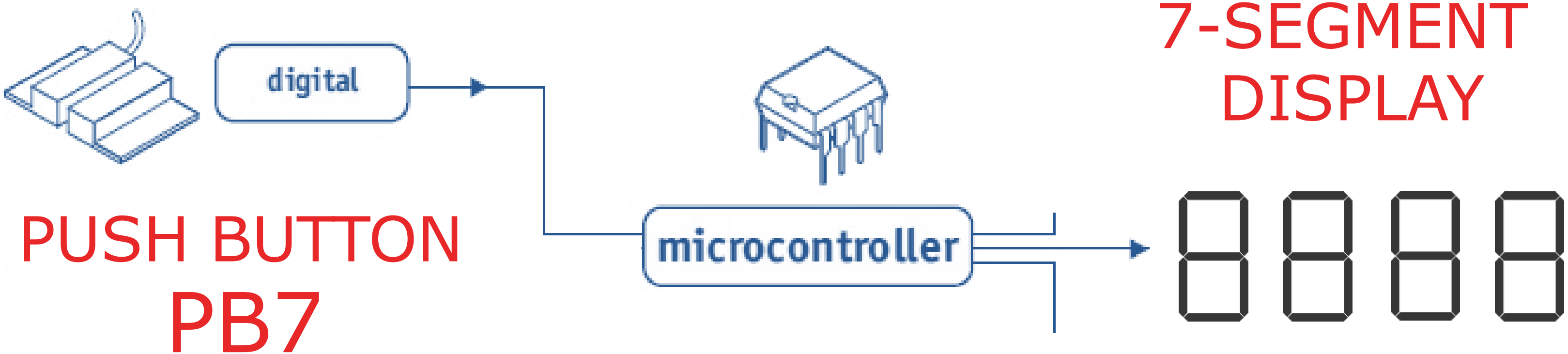
```
LEDON
  BSF PORTD, 5
GOTO MAIN

LEDOFF
  BCF PORTD, 5
GOTO MAIN
END
```

MCU SYSTEM

INPUT

PROCESS

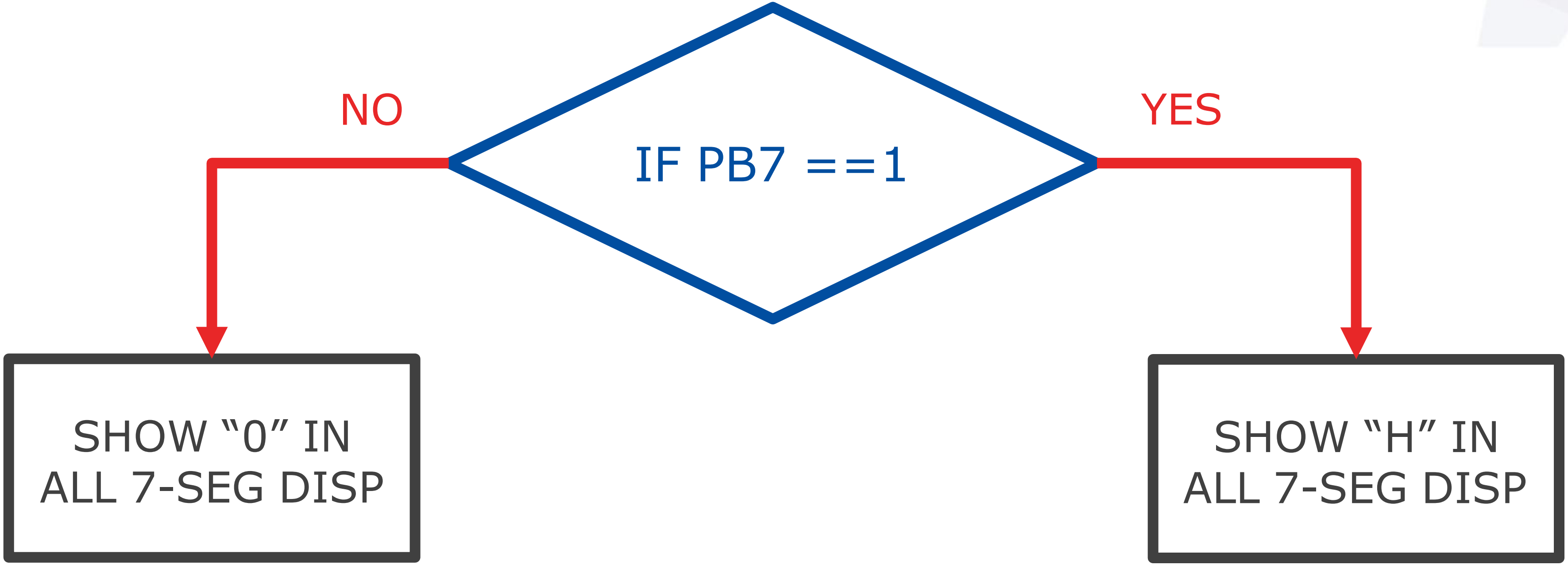
OUTPUT



7-SEGMENT DISPLAY

PUSH BUTTON PB7

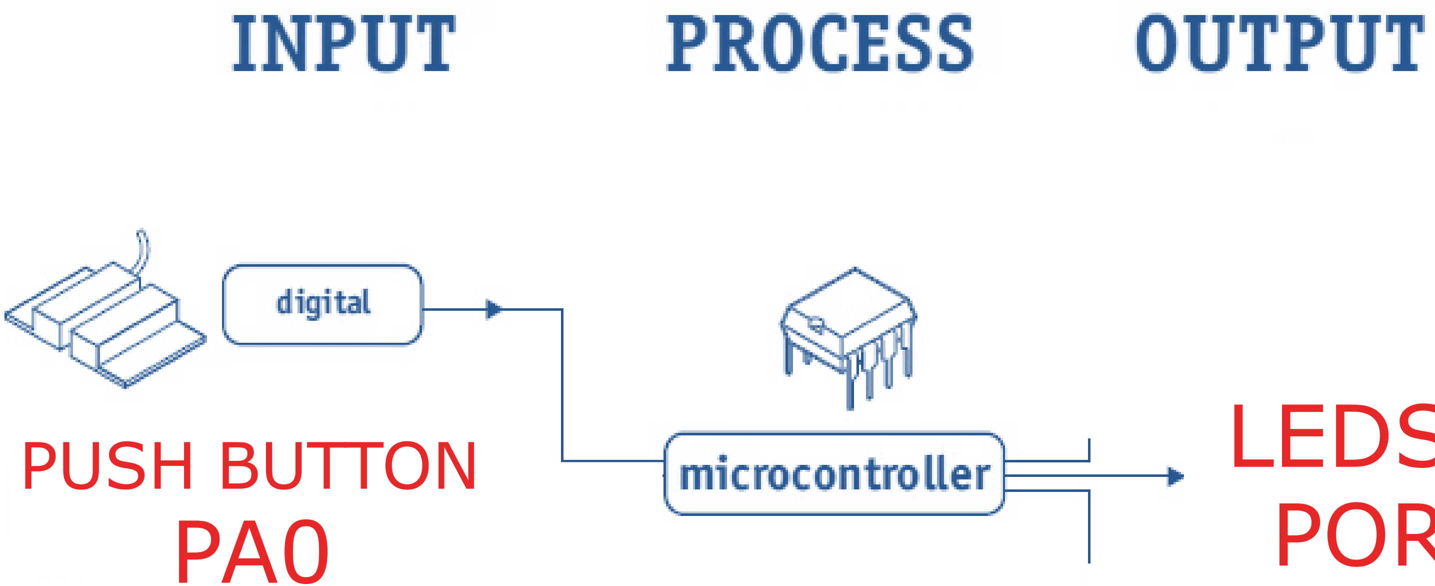
IF STATEMENT IN ASSEMBLY



```
ORG 0x00
BSF STATUS, 5
BSF TRISB, 7
CLRF TRISA
CLRF TRISD
MAIN
BCF STATUS, 5
BTFSC PORTB, 7
  GOTO SHOWH
GOTO SHOWZ
```

```
SHOWH
    MOVLW  0x0F
    MOVWF PORTA
    MOVLW  0x76
    MOVWF PORTD
GOTO MAIN
SHOWZ
    MOVLW  0x0F
    MOVWF PORTA
    MOVLW  0x3F
    MOVWF PORTD
GOTO MAIN
END
```

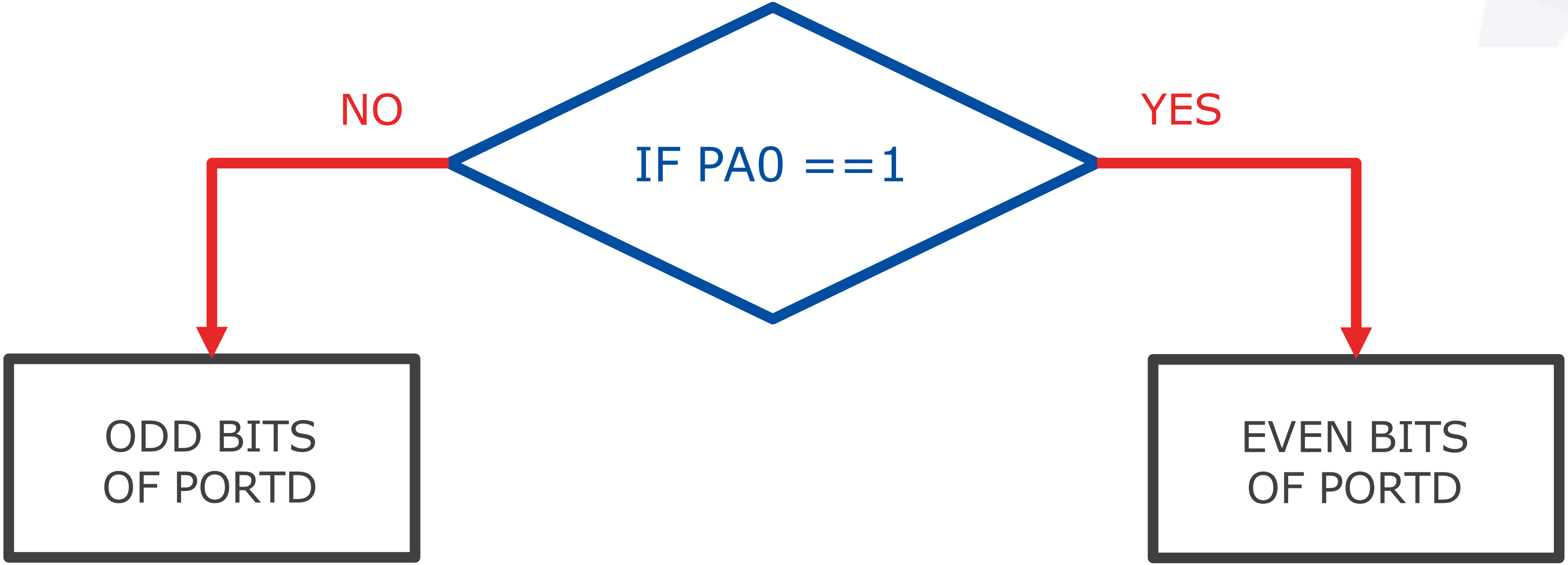
MCU SYSTEM



PUSH BUTTON
PA0

LEDS ON
PORTD

IF STATEMENT IN ASSEMBLY



IF PA0 ==1

NO

YES

ODD BITS OF PORTD

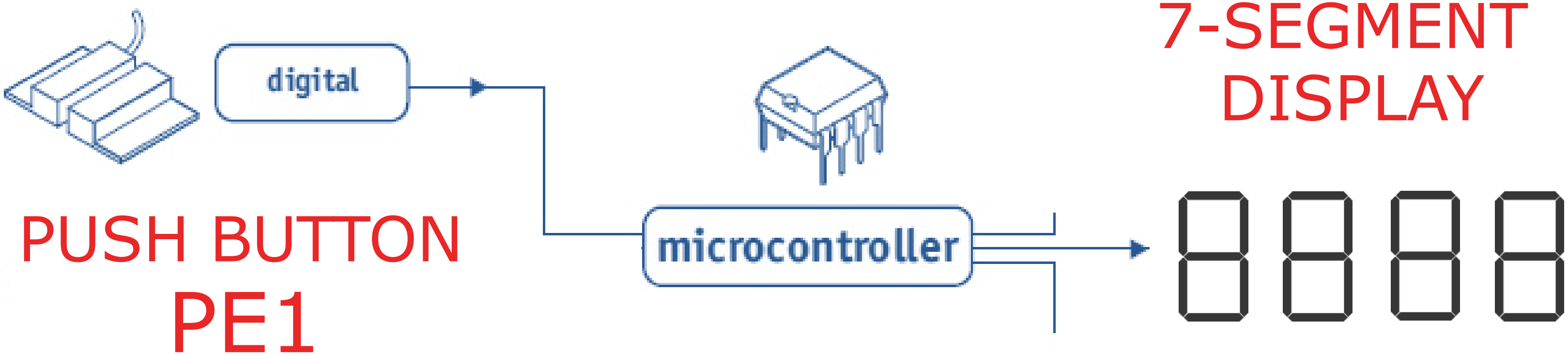
EVEN BITS OF PORTD

MCU SYSTEM

INPUT

PROCESS

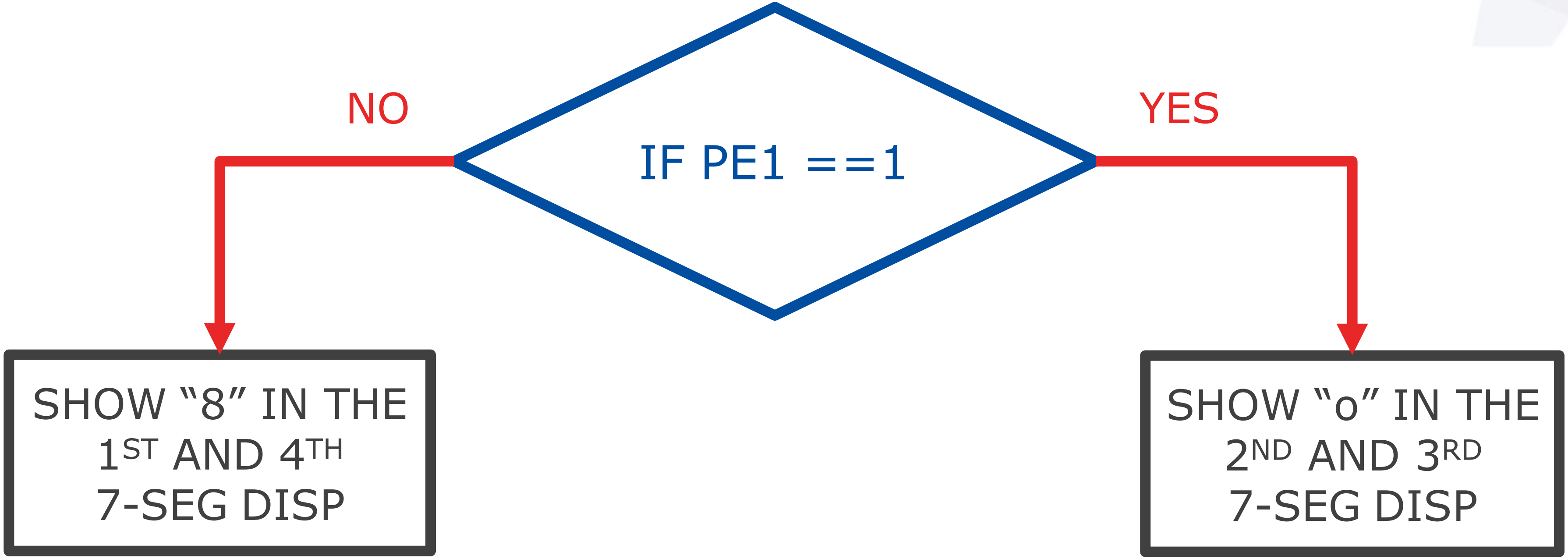
OUTPUT



7-SEGMENT DISPLAY

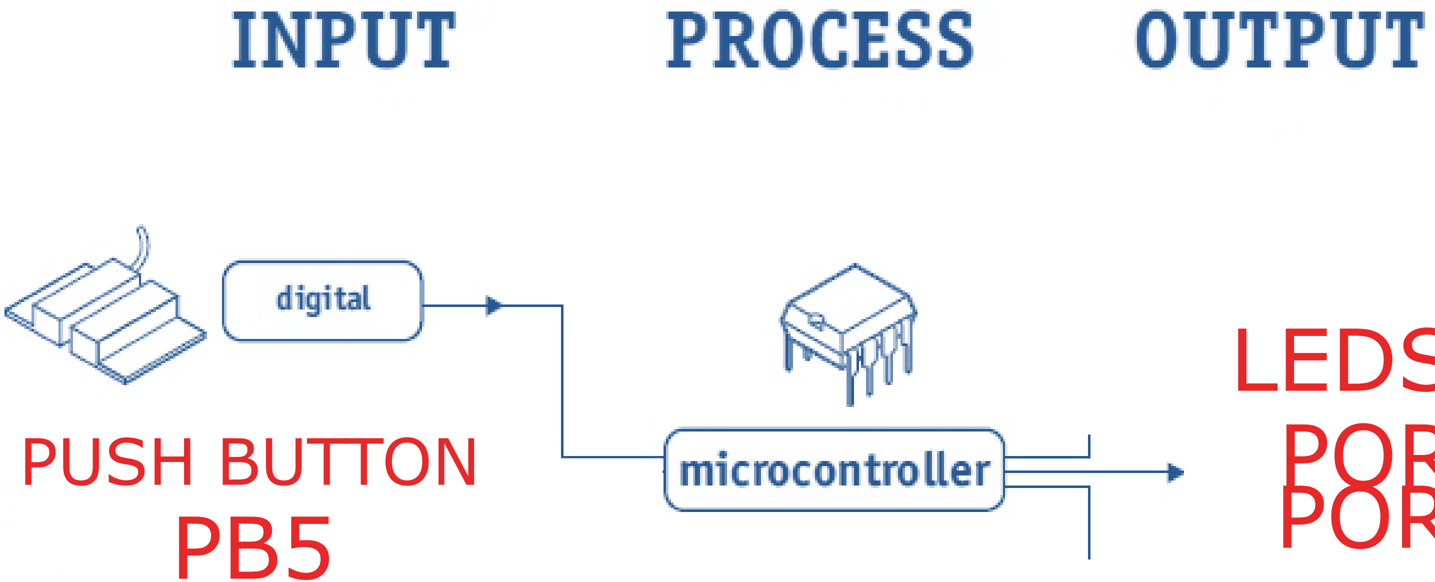
PUSH BUTTON
PE1

IF STATEMENT IN ASSEMBLY



NO
IF PE1 ==1
YES
SHOW “8” IN THE 1ST AND 4TH 7-SEG DISP
SHOW “o” IN THE 2ND AND 3RD 7-SEG DISP

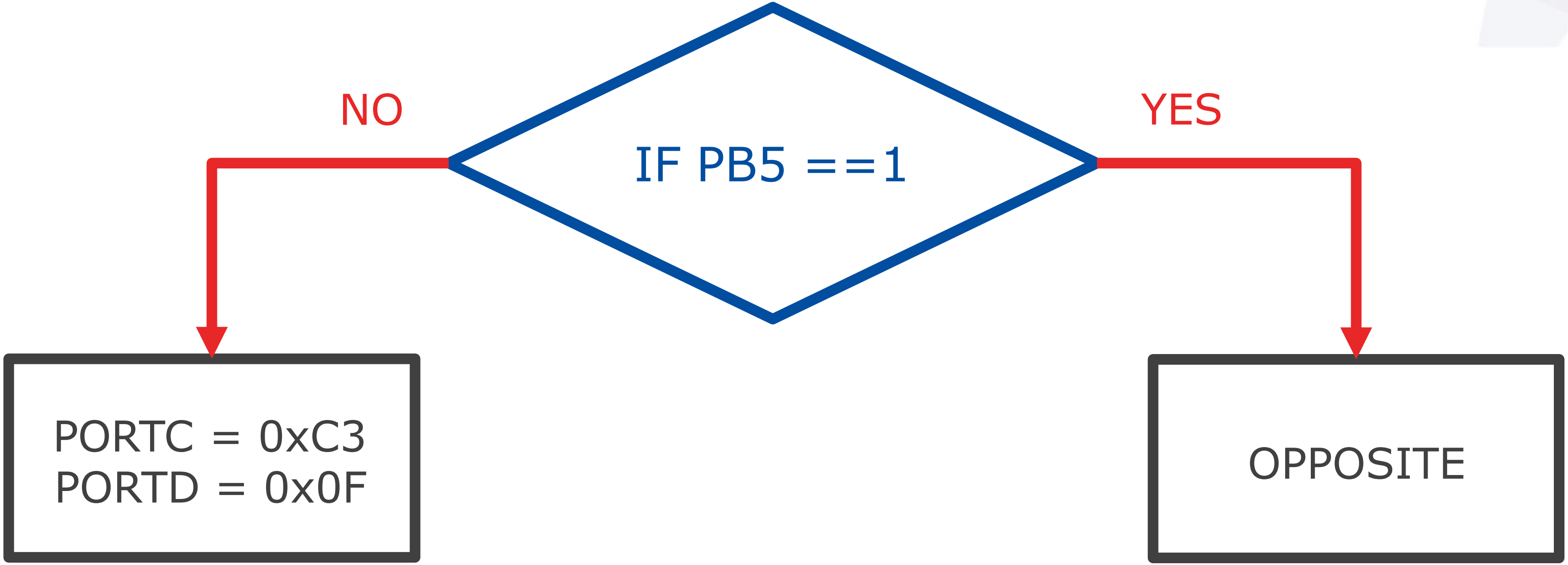
MCU SYSTEM



PUSH BUTTON
PB5

LEDS ON
PORTC
PORTD

IF STATEMENT IN ASSEMBLY

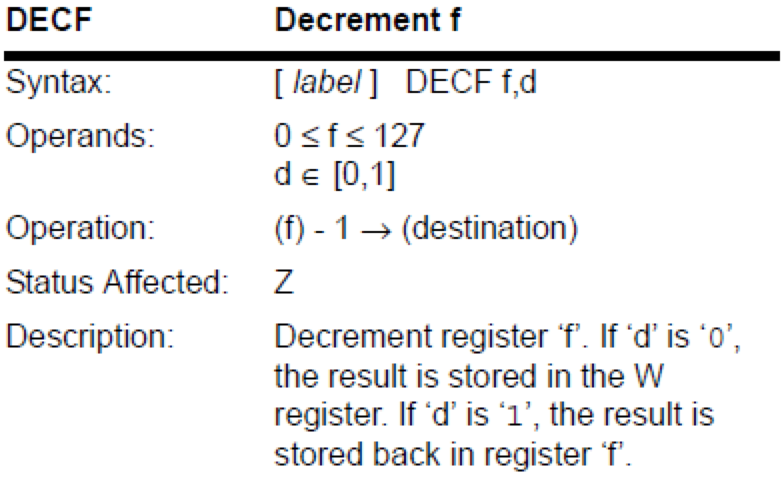


CLASS 6

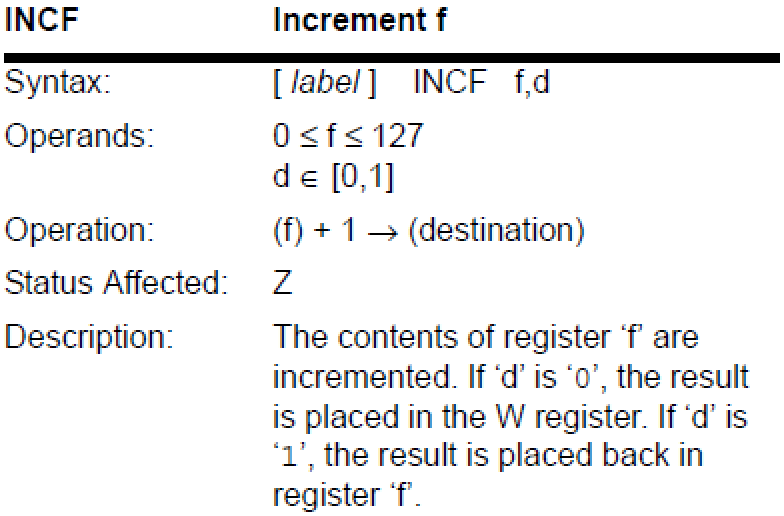
ASSEMBLY LANGUAGE

- Delays

Instruction Descriptions

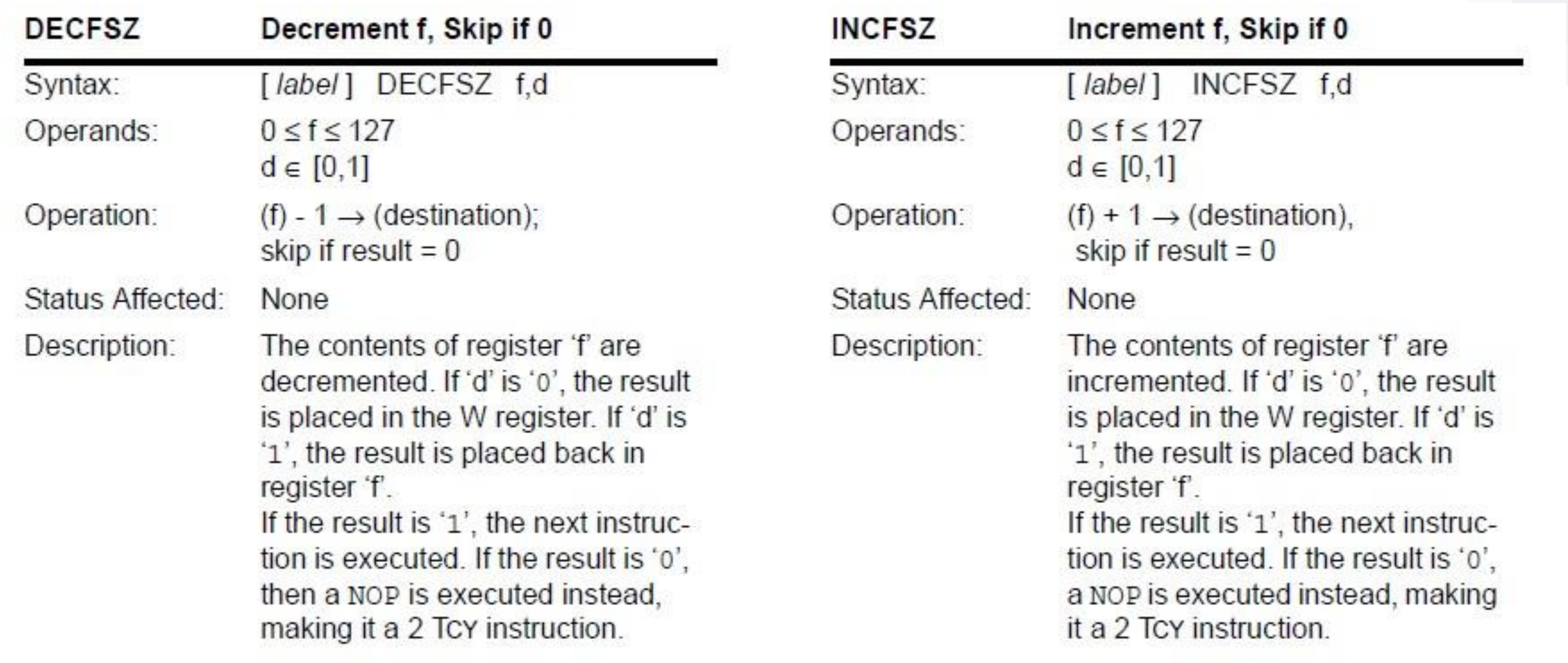


DECF	**Decrement f**
Syntax:	[*label*] DECF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) - 1 → (destination)
Status Affected:	Z
Description:	Decrement register ‘f’. If ‘d’ is ‘0’, the result is stored in the W register. If ‘d’ is ‘1’, the result is stored back in register ‘f’.



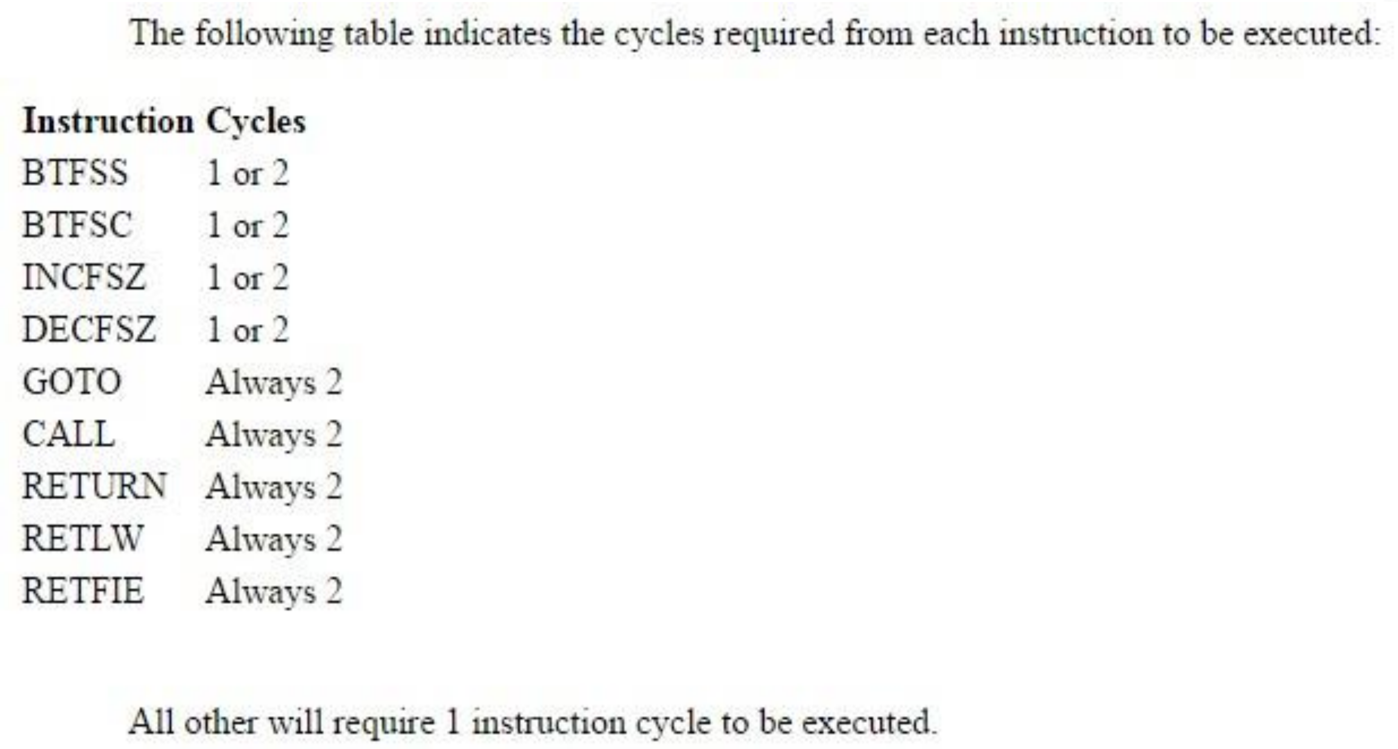
INCF	**Increment f**
Syntax:	[*label*] INCF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) + 1 → (destination)
Status Affected:	Z
Description:	The contents of register ‘f’ are incremented. If ‘d’ is ‘0’, the result is placed in the W register. If ‘d’ is ‘1’, the result is placed back in register ‘f’.

Instruction Descriptions



DECFSZ	Decrement f, Skip if 0
Syntax:	[*label*] DECFSZ f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) - 1 → (destination); skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are decremented. If 'd' is '0', the result is placed in the W register. If 'd' is '1', the result is placed back in register 'f'. If the result is '1', the next instruction is executed. If the result is '0', then a NOP is executed instead, making it a 2 TCY instruction.

INCFSZ	Increment f, Skip if 0
Syntax:	[*label*] INCFSZ f,d
Operands:	$0 \leq f \leq 127$ $d \in [0,1]$
Operation:	(f) + 1 → (destination), skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are incremented. If 'd' is '0', the result is placed in the W register. If 'd' is '1', the result is placed back in register 'f'. If the result is '1', the next instruction is executed. If the result is '0', a NOP is executed instead, making it a 2 TCY instruction.



The following table indicates the cycles required from each instruction to be executed:

Instruction	Cycles
BTFSS	1 or 2
BTFSC	1 or 2
INCFSZ	1 or 2
DECFSZ	1 or 2
GOTO	Always 2
CALL	Always 2
RETURN	Always 2
RETLW	Always 2
RETFIE	Always 2

All other will require 1 instruction cycle to be executed.

EX1: Calculate the delay created by the below code segment if oscillator of 4 MHz. (Assume the number 5 is loaded into Counter)

```
LOOP
DECFSZ      Counter,F
GOTO        LOOP
```

Solution:

No. of cycles = 5 X(1 +2) = 15
Delay by this code= No. of cycles X Instruction cycle
= 15 X (4 X 0.25uS)
= 15 us

EX2: Calculate the delay created by the below code segment if oscillator of 8 MHz. (Assume the number 5 is loaded into Counter)

```
LOOP
DECFSZ        Counter , F
GOTO          LOOP
```

Solution:

No. of cycles = 5 X(1 +2) = 15
Delay by this code= No. of cycles X Instruction cycle
= 15   X   (0.5uS)
= 7.5 us

EX3: Calculate the delay created by the Loop in the below code segment oscillator of 4 MHz:

```
Counter1        EQU       35H

START

MOVLW       0F2H
MOVWF       Counter1

LOOP
DECFSZ      Counter1,F
GOTO     LOOP

END
```

Solution:

No. of cycles in Loop =   [242 X (1 +2)] = 726
No. of cycles in code =   1 + 1 + 726= 728

Delay by this code= No. of cycles X Instruction cycle
= 728   X   (4 X 0.25uS)
= 728 uS

EX 2: Calculate the delay created by the Loop in the below code segment oscillator of 8 MHz:

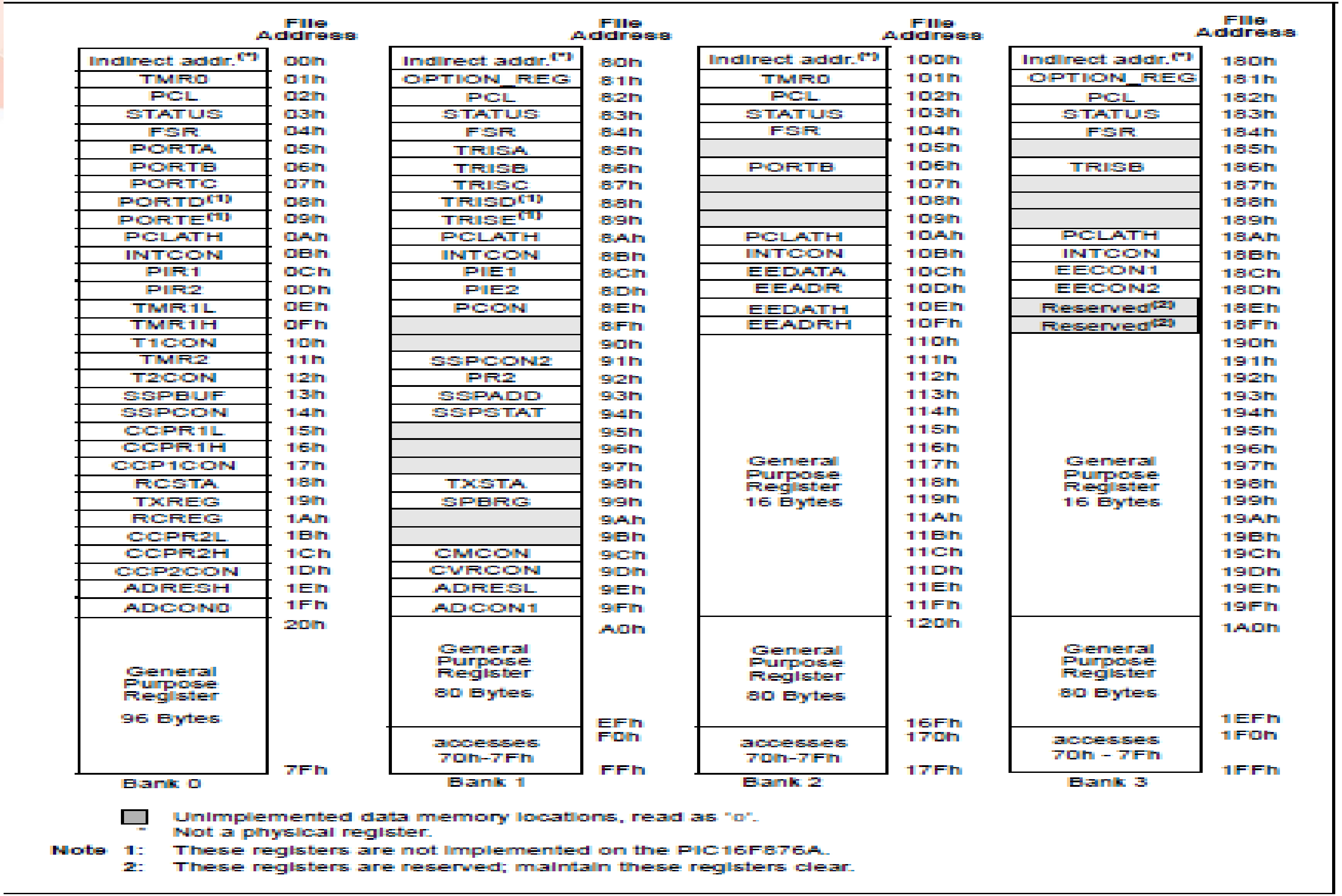
```
Counter1      EQU 70H

START

MOVLW        0F2H
MOVWF        Counter1

LOOP
DECFSZ       Counter1,F
GOTO         LOOP

END
```

Solution:

No. of cycles in Loop =   [242 X (1 +2)] = 726
No. of cycles in code =   1 + 1 + 726= 728

Delay by this code= No. of cycles X Instruction cycle
= 728   X   (0.5 uS)
= 364 uS

FIGURE 2-3: PIC16F876A/877A REGISTER FILE MAP



	File Address		File Address		File Address		File Address
Indirect addr.(*)	00h	Indirect addr.(*)	80h	Indirect addr.(*)	100h	Indirect addr.(*)	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	▨	105h	▨	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	▨	107h	▨	187h
PORTD(1)	08h	TRISD(1)	88h	▨	108h	▨	188h
PORTE(1)	09h	TRISE(1)	89h	▨	109h	▨	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDATA	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved(2)	18Eh
TMR1H	0Fh	▨	8Fh	EEADRH	10Fh	Reserved(2)	18Fh
T1CON	10h	▨	90h	General Purpose Register 16 Bytes	110h	General Purpose Register 16 Bytes	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	▨	95h		115h		195h
CCPR1H	16h	▨	96h		116h		196h
CCP1CON	17h	▨	97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah	▨	9Ah		11Ah		19Ah
CCPR2L	1Bh	▨	9Bh		11Bh		19Bh
CCPR2H	1Ch	CMCON	9Ch		11Ch		19Ch
CCP2CON	1Dh	CVRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Register 96 Bytes	20h	General Purpose Register 80 Bytes	A0h EFh	General Purpose Register 80 Bytes	120h 16Fh	General Purpose Register 80 Bytes	1A0h 1EFh
	7Fh	accesses 70h-7Fh	F0h FFh	accesses 70h-7Fh	170h 17Fh	accesses 70h - 7Fh	1F0h 1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

▨ Unimplemented data memory locations, read as '0'.

* Not a physical register.

Note 1: These registers are not implemented on the PIC16F876A.

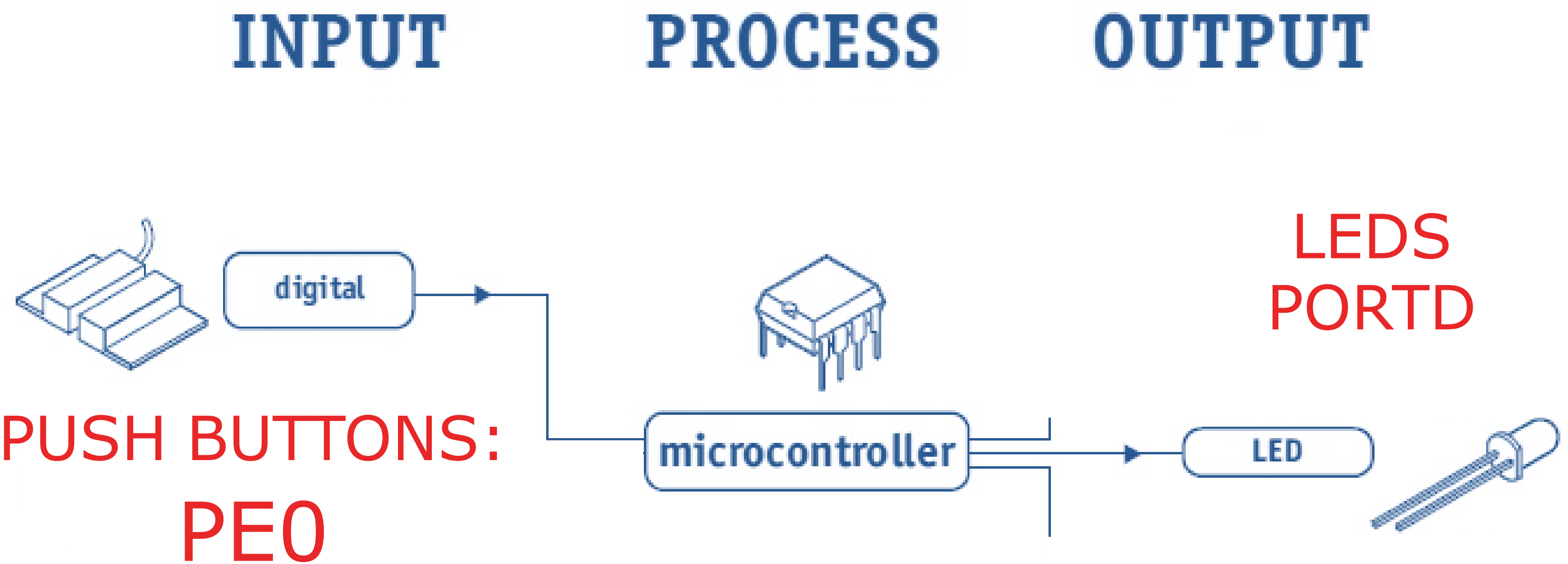
2: These registers are reserved; maintain these registers clear.

CLASS 6

ASSEMBLY LANGUAGE

- CONDITIONAL STATEMENT: BTFSC, BTFSS
- READ MULTIPLE INPUTS
- FUNCTIONS

MCU SYSTEM



LEDS
PORTD

PUSH BUTTONS:

PE0

PE1

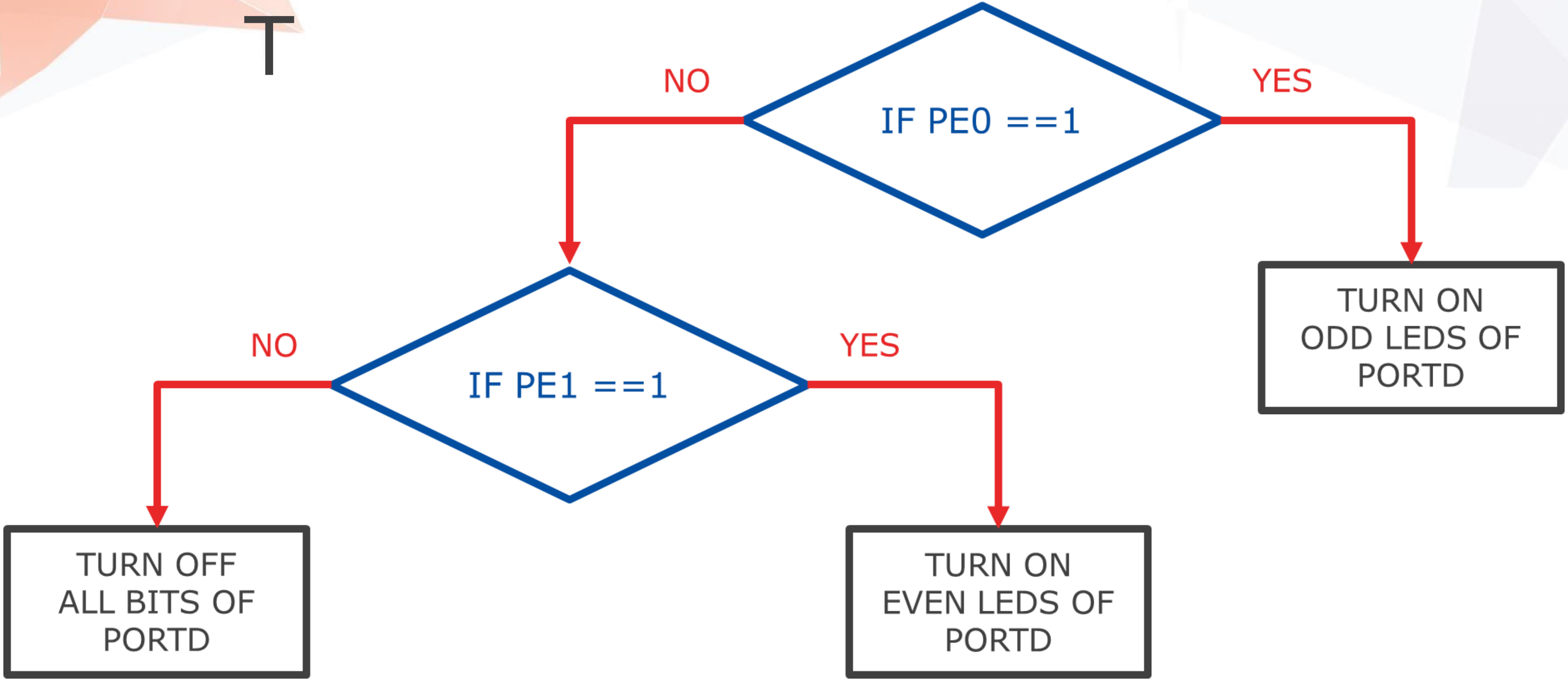
PSEUDOCODE

```
IF PE0 IS PRESSED THEN
    TURN ON ODD BITS OF PORTD
ELSE IF PE1 IS PRESSED THEN
    TURN ON EVEN BITS OF PORTD
ELSE
    TURN OFF ALL BITS OF PORTD
END IF
```

FLOWCHART

?

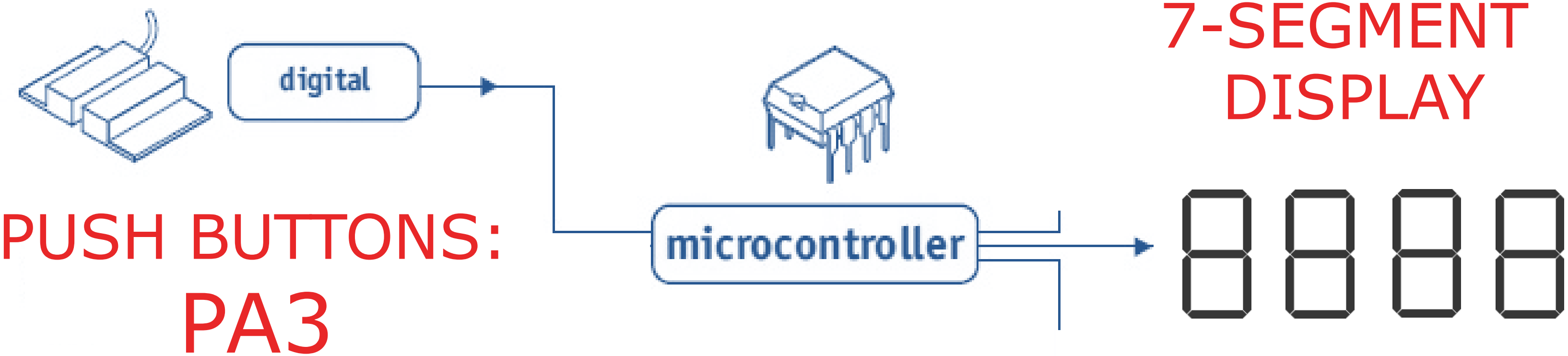
FLOWCHAR T



IF PE0 ==1
NO
YES
TURN ON ODD LEDS OF PORTD
IF PE1 ==1
NO
YES
TURN OFF ALL BITS OF PORTD
TURN ON EVEN LEDS OF PORTD

MCU SYSTEM

INPUT PROCESS OUTPUT



digital

microcontroller

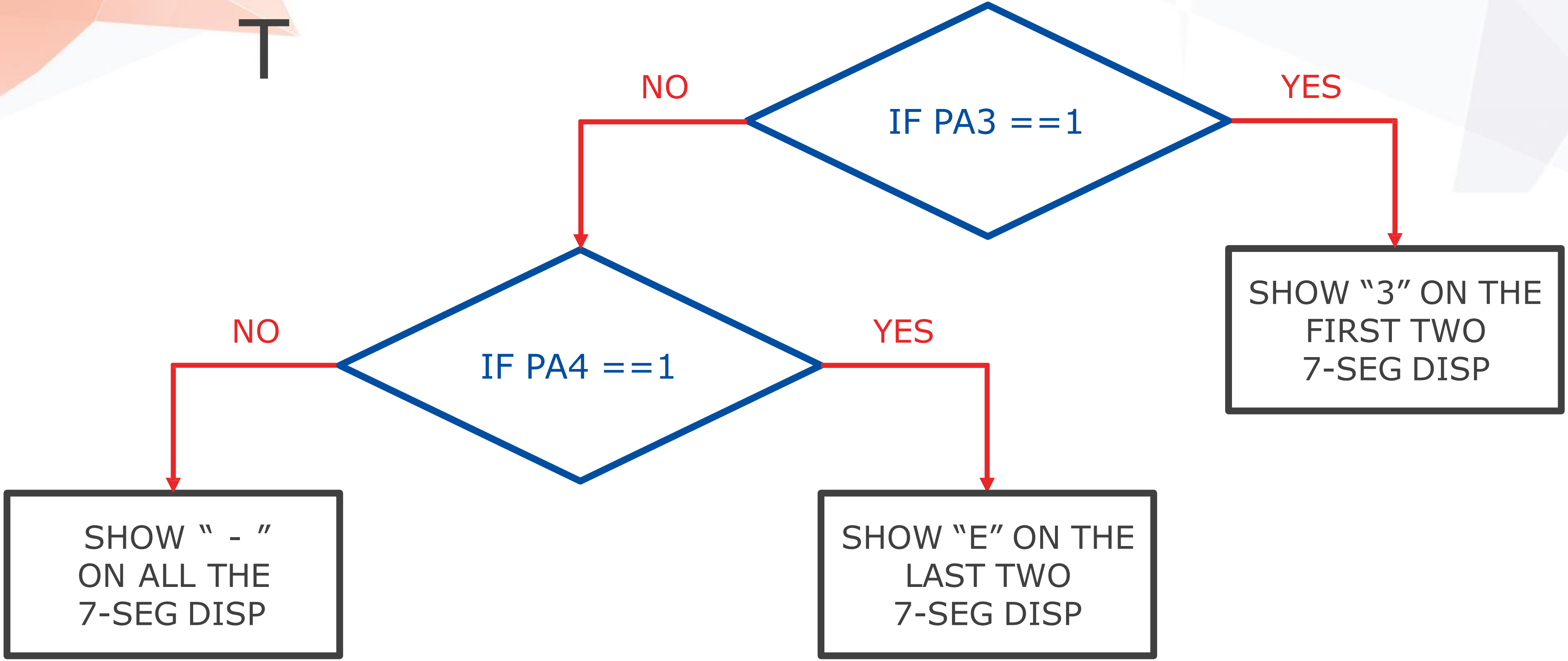
7-SEGMENT DISPLAY

PUSH BUTTONS:

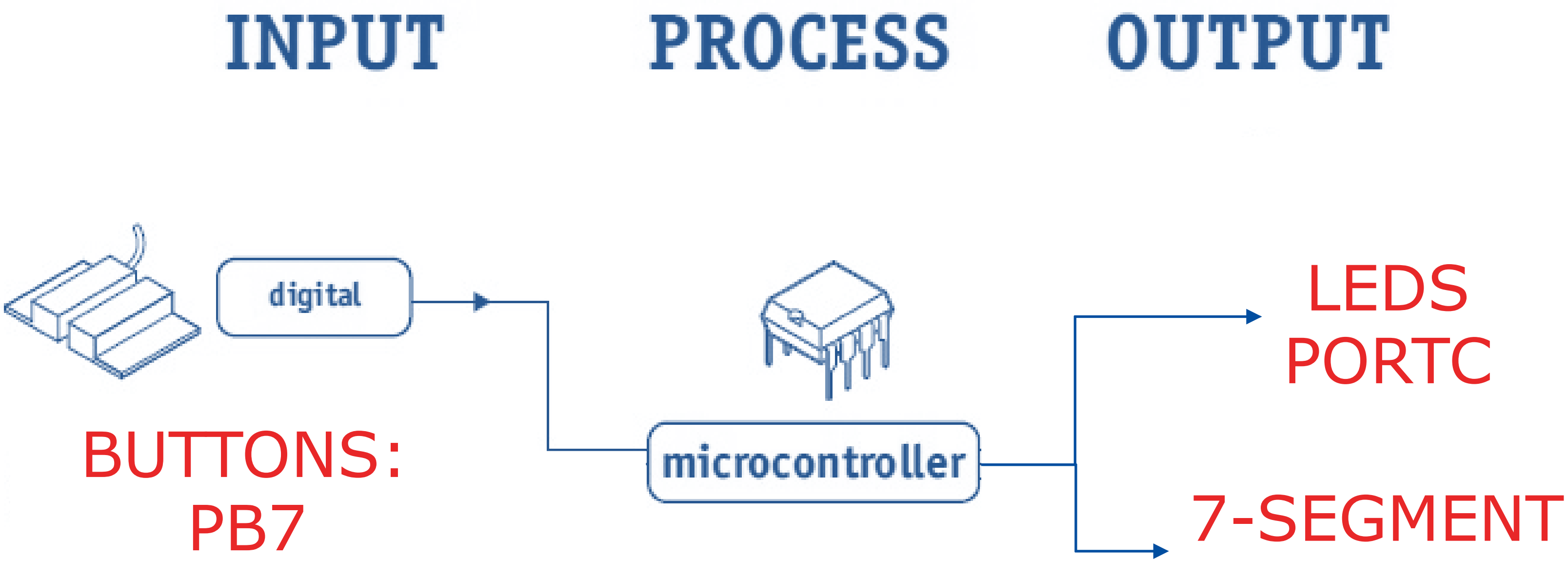
PA3

PA4

FLOWCHAR T


NO
IF PA3 ==1
YES
SHOW "3" ON THE FIRST TWO 7-SEG DISP
NO
IF PA4 ==1
YES
SHOW " - " ON ALL THE 7-SEG DISP
SHOW "E" ON THE LAST TWO 7-SEG DISP

MCU SYSTEM

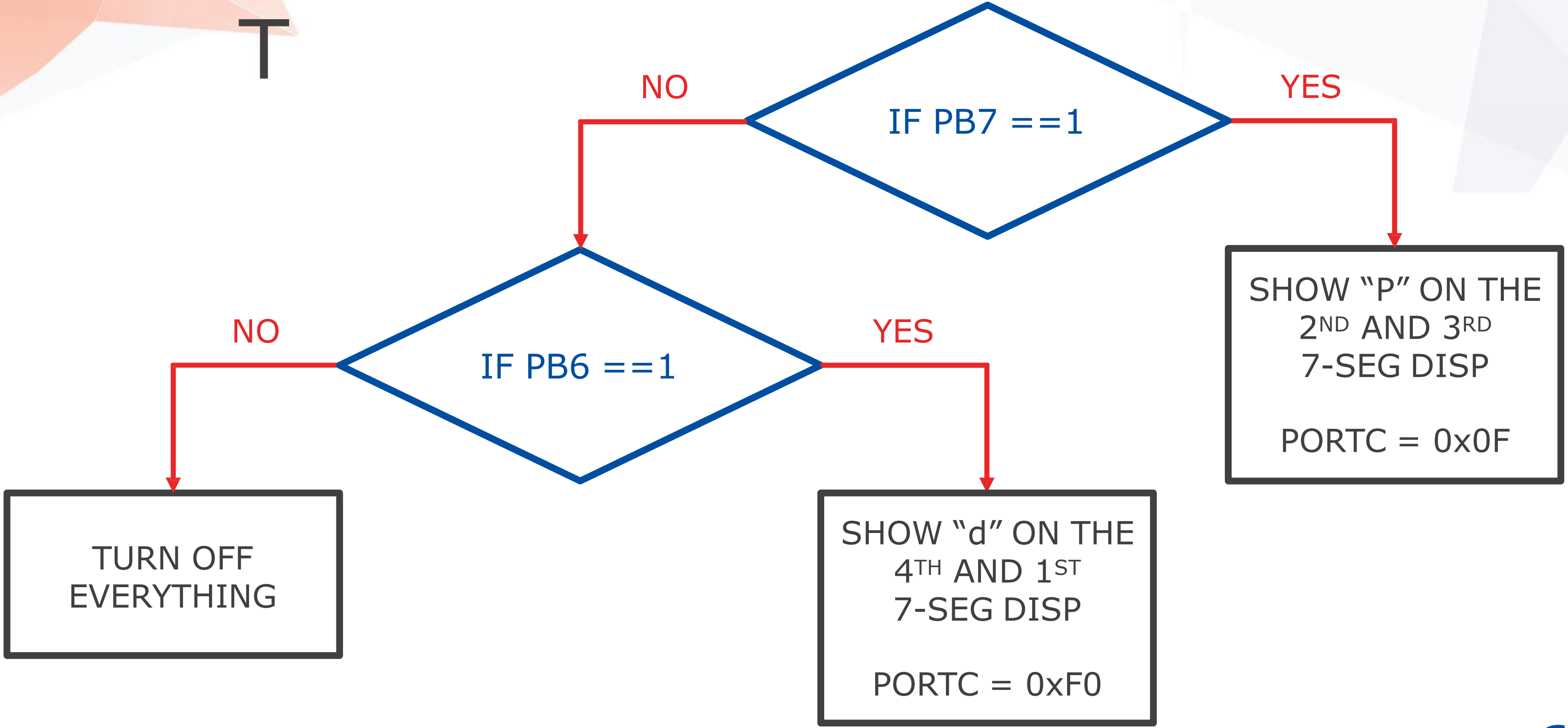


BUTTONS:
PB7
PB6

LEDS
PORTC

7-SEGMENT
DISPLAY

FLOWCHAR
T



NO
IF PB7 ==1
YES
SHOW "P" ON THE 2ND AND 3RD 7-SEG DISP
PORTC = 0x0F
NO
IF PB6 ==1
YES
TURN OFF EVERYTHING
SHOW "d" ON THE 4TH AND 1ST 7-SEG DISP
PORTC = 0xF0

THANK YOU

This is a good video to remember or learn the basics of C Programming Language

www.youtube.com/watch?v=3lQEunpmtRA

You have the direct link on your Moodle page

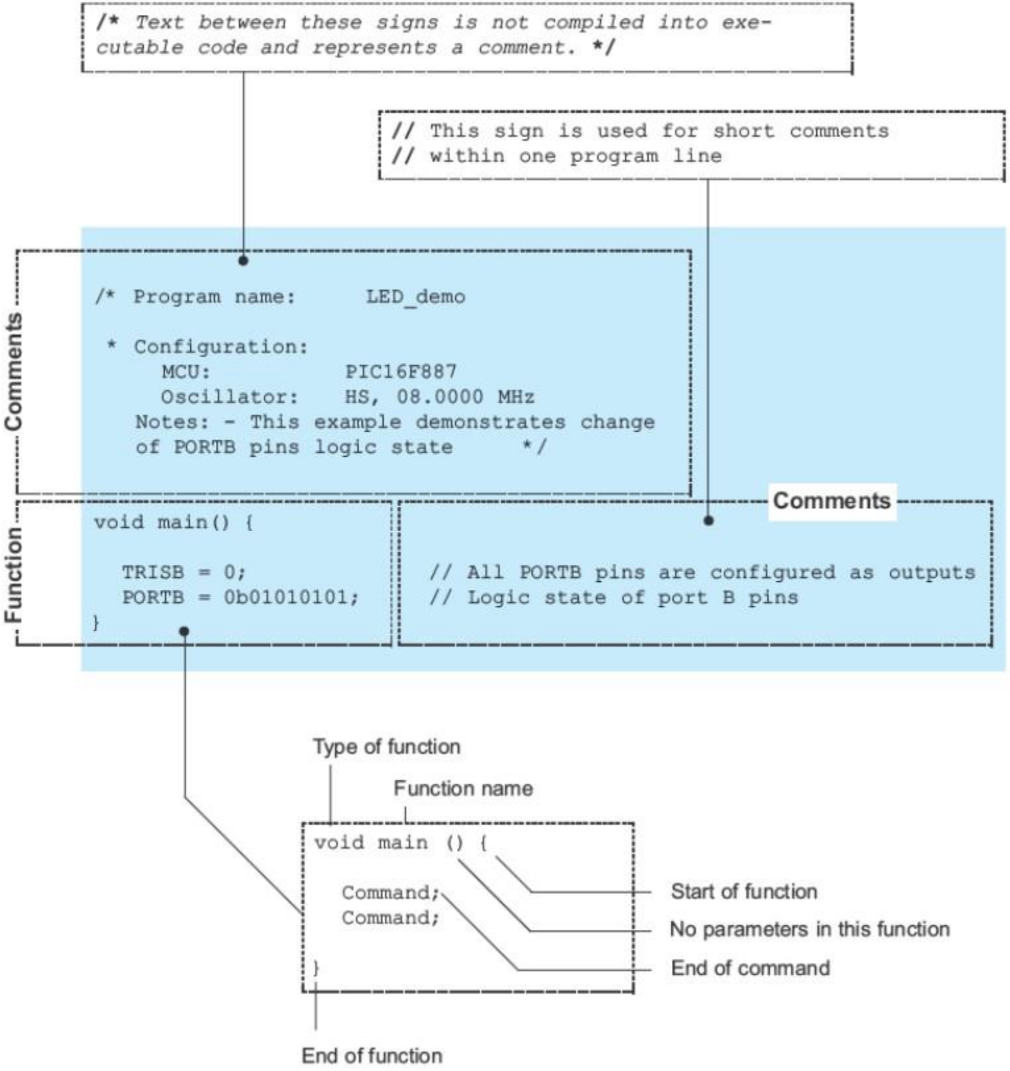
CLASS CONTENT

C LANGUAGE

- C LANGUAGE SYNTAX
- INPUTS AND OUTPUTS
- BLINK LEDS
- 7-SEGMENT DISPLAY

mikroC
PRO for PIC
mikro e

C PROGRAM STRUCTURE



```
/* Text between these signs is not compiled into exe-
cutable code and represents a comment. */
```

```
// This sign is used for short comments
// within one program line
```

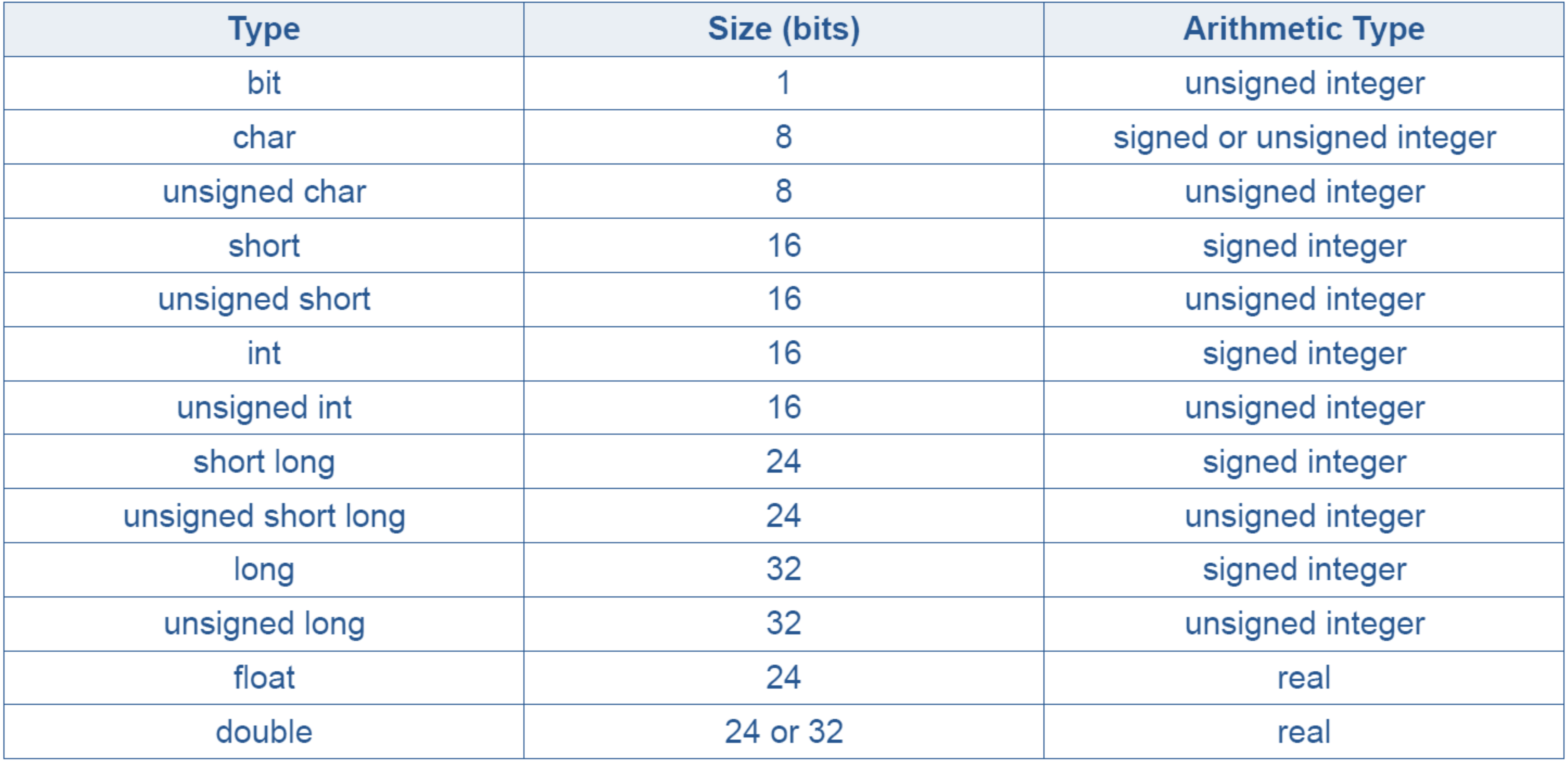
```
/* Program name:      LED_demo

 * Configuration:
     MCU:             PIC16F887
     Oscillator:      HS, 08.0000 MHz
   Notes: - This example demonstrates change
   of PORTB pins logic state      */
void main() {

  TRISB = 0;             // All PORTB pins are configured as outputs
  PORTB = 0b01010101;    // Logic state of port B pins
}
```

```
void main () {

  Command;
  Command;

}
```

COMMENTS

- COMMENTS ARE PARTS OF THE PROGRAM USED TO CLARIFY THE OPERATION
- COMMENTS ARE IGNORED AND NOT COMPILED INTO EXECUTABLE CODE BY THE COMPILER
- **(/* …. */)** DESIGNATES LONG COMMENTS
- **(//)** DESIGNATES SHORT COMMENTS

DATA TYPES



Type	Size (bits)	Arithmetic Type
bit	1	unsigned integer
char	8	signed or unsigned integer
unsigned char	8	unsigned integer
short	16	signed integer
unsigned short	16	unsigned integer
int	16	signed integer
unsigned int	16	unsigned integer
short long	24	signed integer
unsigned short long	24	unsigned integer
long	32	signed integer
unsigned long	32	unsigned integer
float	24	real
double	24 or 32	real

VARIABLES

ANY NUMBER CHANGING ITS VALUE DURING PROGRAM OPERATION IS CALLED A VARIABLE.

E.G. if the program adds two numbers (number1 and number2), it is necessary to have a value to represent what we in everyday life call the sum. in this case number1, number2 and sum are variables.

VARIABLE DECLARATION

- EVERY VARIABLE MUST BE DECLARED PRIOR TO BEING USED FOR THE FIRST TIME IN THE PROGRAM.
- VARIABLES ARE STORED IN THE RAM MEMORY.

E.G. int gate1; *// Declare name and type of variable gate1*

VARIABLE DECLARATION

- VARIABLE NAMES CAN INCLUDE ANY OF THE ALPHABETICAL CHARACTERS (A-Z), THE DIGITS 0-9 AND THE UNDERSCORE CHARACTER ("_").
- THE COMPILER IS CASE SENSITIVE AND DIFFERENTIATES BETWEEN CAPITAL AND SMALL LETTERS.

VARIABLE DECLARATION

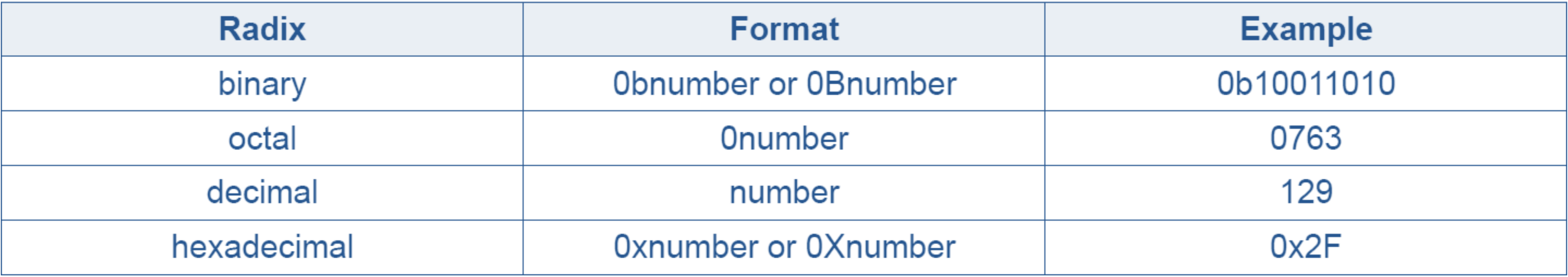
- FUNCTIONS AND VARIABLES NAMES USUALLY CONTAIN LOWER CASE CHARACTERS, WHILE CONSTANT NAMES CONTAIN UPPERCASE CHARACTERS.

VARIABLE DECLARATION

- VARIABLE NAMES MUST NOT START WITH A DIGIT.
- SOME OF THE NAMES CANNOT BE USED AS VARIABLE NAMES AS ALREADY BEING USED BY THE COMPILER ITSELF. SUCH NAMES ARE CALLED THE KEY WORDS.

INTEGER CONSTANTS

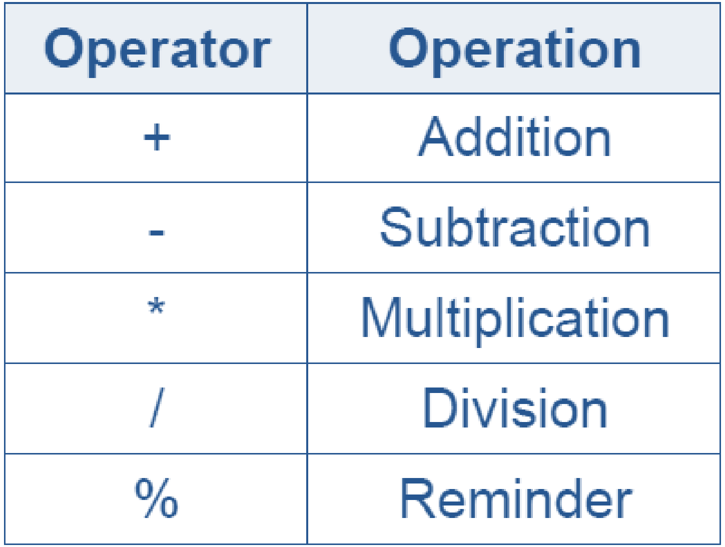
A CONSTANT IS A NUMBER OR A CHARACTER HAVING FIXED VALUE THAT CANNOT BE CHANGED DURING PROGRAM EXECUTION



Radix	Format	Example
binary	0bnumber or 0Bnumber	0b10011010
octal	0number	0763
decimal	number	129
hexadecimal	0xnumber or 0Xnumber	0x2F

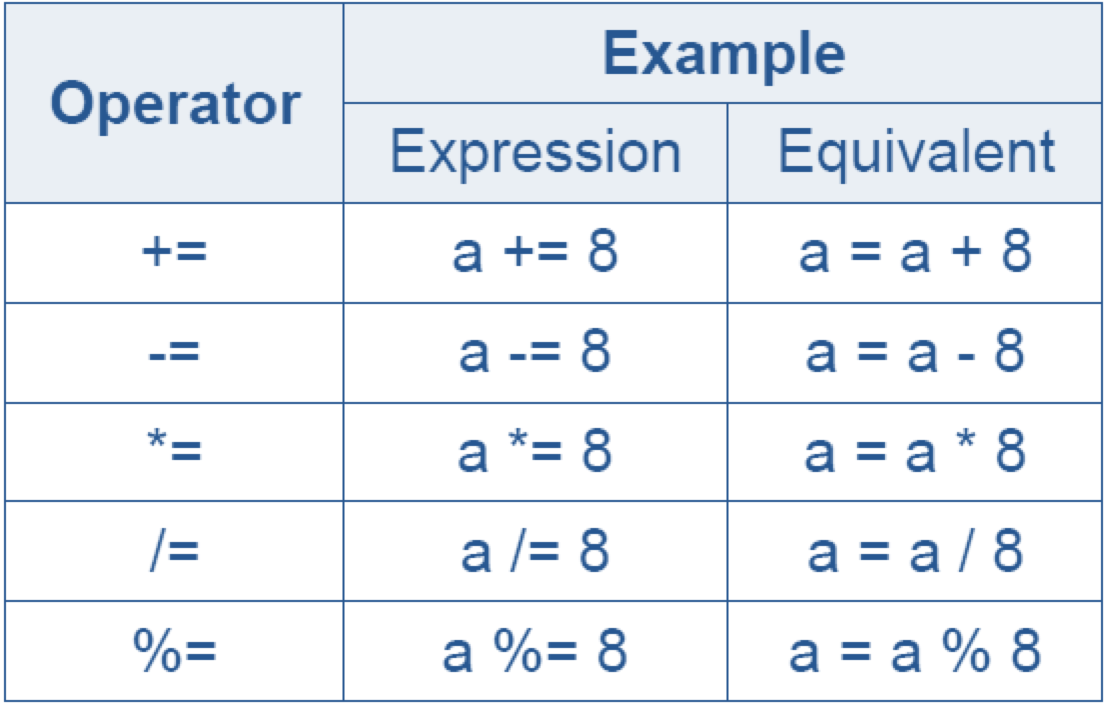
```
const int MINIMUM = -100; // Declare constant MINIMUN
```

ARITHMETIC OPERATORS



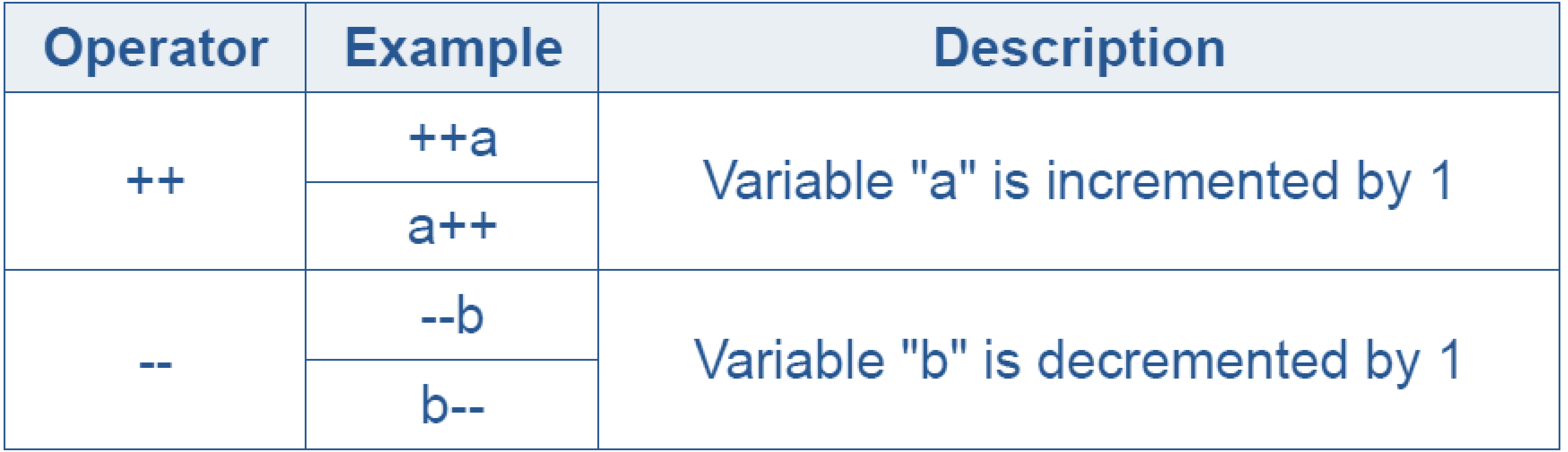
Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Reminder

ASSIGNMENT OPERATORS



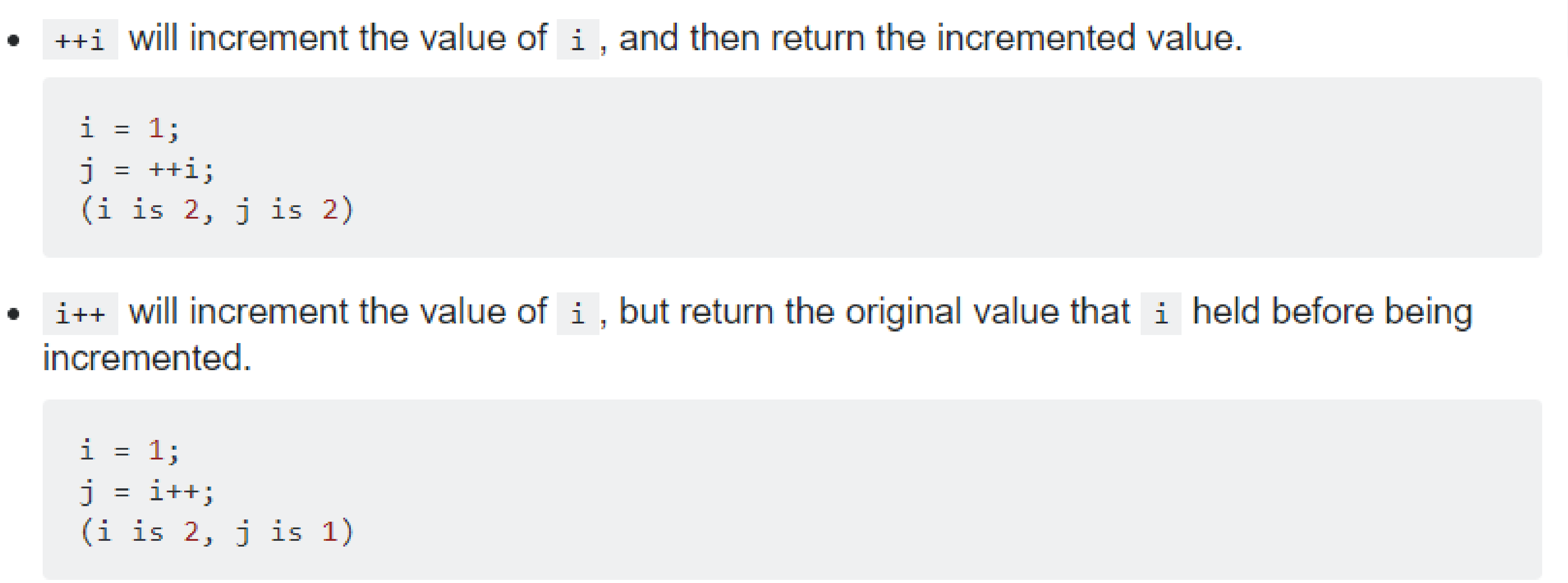
Operator	Example	
	Expression	Equivalent
+=	a += 8	a = a + 8
-=	a -= 8	a = a - 8
*=	a *= 8	a = a * 8
/=	a /= 8	a = a / 8
%=	a %= 8	a = a % 8

INCREMENT AND DECREMENT OPERATORS



Operator	Example	Description
++	++a	Variable "a" is incremented by 1
	a++	
--	--b	Variable "b" is decremented by 1
	b--	

INCREMENT AND DECREMENT OPERATORS



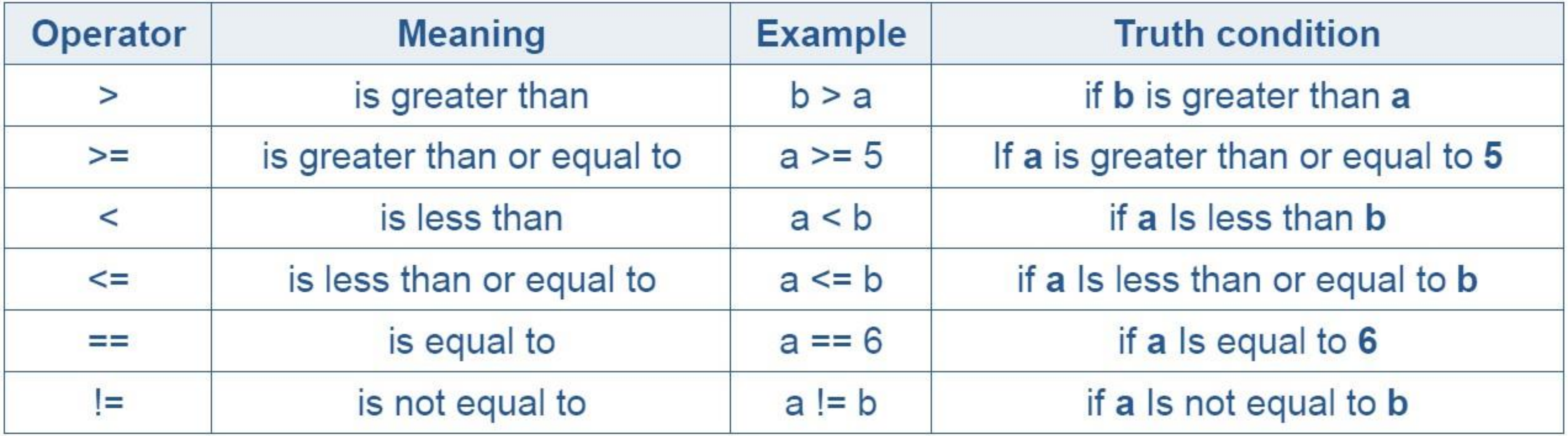
- `++i` will increment the value of `i`, and then return the incremented value.

```
i = 1;
j = ++i;
(i is 2, j is 2)
```

- `i++` will increment the value of `i`, but return the original value that `i` held before being incremented.

```
i = 1;
j = i++;
(i is 2, j is 1)
```

RELATIONAL OPERATORS



Operator	Meaning	Example	Truth condition
>	is greater than	b > a	if **b** is greater than **a**
>=	is greater than or equal to	a >= 5	If **a** is greater than or equal to **5**
<	is less than	a < b	if **a** Is less than **b**
<=	is less than or equal to	a <= b	if **a** Is less than or equal to **b**
==	is equal to	a == 6	if **a** Is equal to **6**
!=	is not equal to	a != b	if **a** Is not equal to **b**

LOGIC OPERATORS

Operator	Logical AND		
& &	Operand1	Operand2	Result
	0	0	0
	0	1	0
	1	0	0
	1	1	1

Operator	Logical OR		
| |	Operand1	Operand2	Result
	0	0	0
	0	1	1
	1	0	1
	1	1	1

Operator	Logical NOT	
!	Operand1	Result
	0	1
	1	0

BITWISE OPERATORS

Operand	Meaning	Example	Result	
~	Bitwise complement	a = ~b	b = 5	a = -5
<<	Shift left	a = b << 2	b = 11110011	a = 11001100
>>	Shift right	a = b >> 3	b = 11110011	a = 00011110

&	Bitwise AND	c = a & b	a = 11100011
			b = 11001100
				c = 11000000

|	Bitwise OR	c = a | b	a = 11100011
			b = 11001100
				c = 11101111

^	Bitwise EXOR	c = a ^ b	a = 11100011
			b = 11001100
				c = 00101111

CONDITIONAL OPERATORS

A CONDITION IS A COMMON INGREDIENT OF THE PROGRAM. WHEN MET, IT IS NECESSARY TO PERFORM ONE OUT OF SEVERAL OPERATIONS. IN OTHER WORDS 'IF THE CONDITION IS MET (...), DO (...). OTHERWISE, IF THE CONDITION IS NOT MET, DO (...)' .

CONDITIONAL OPERANDS IF-ELSE AND SWITCH ARE USED IN CONDITIONAL OPERATIONS.

CONDITIONAL OPERATOR: IF-ELSE

```
if(expression)
    operation1;
else
    operation2;
```

```
if(expression)
{   operation1;
    operation2; }
else
{   operation3;
    operation4; }
```

CONDITIONAL OPERATOR: SWITCH

```
switch (selector) { // Selector is of char or int type
case constant1:
operation1; // Group of operators are executed if
… // selector and constant1 are equal
break;
case constant2:
operation2; // Group of operators are executed if
… // selector and constant2 are equal
break;
…
default:
expected_operation;// Group of operators are executed if no
… // constant is equal to selector
break;
```

PROGRAM LOOP

IT IS OFTEN NECESSARY TO REPEAT A CERTAIN OPERATION FOR A COUPLE OF TIMES IN THE PROGRAM.

A SET OF COMMANDS BEING REPEATED IS CALLED THE PROGRAM LOOP.

HOW MANY TIMES IT WILL BE EXECUTED, **I.E. HOW LONG THE PROGRAM WILL STAY IN THE LOOP**, DEPENDS ON THE CONDITIONS TO LEAVE THE LOOP.

WHILE LOOP

```
while(expression)
{
    commands;
    …
}
```

WHILE LOOP

THE COMMANDS ARE EXECUTED REPEATEDLY (THE PROGRAM REMAINS IN THE LOOP) UNTIL THE EXPRESSION BECOMES FALSE.

IF THE EXPRESSION IS FALSE ON ENTRY TO THE LOOP, THEN THE LOOP WILL NOT BE EXECUTED AND THE PROGRAM WILL PROCEED FROM THE END OF THE WHILE LOOP.

ENDLESS LOOP

```
while(1)
{
    ...  // Expressions enclosed within
         // curly brackets will be
    ... // endlessly executed (endless
        // loop).
}
```

FOR LOOP

```
for(initial_expression; condition_expression; change_expression)
{
    operation;
    ...
}
```

FOR LOOP

```
for(k=1; k<5; k++) // Increase variable k 5 times (from 1 to 5)
{
    operation;             // repeat expression operation every time
    ...
}
```

Operation is to be performed five times. After that, it will be validated by checking that the expression k<5 is false (after 5 iterations k=5) and the program will exit the for loop.

DO-WHILE LOOP

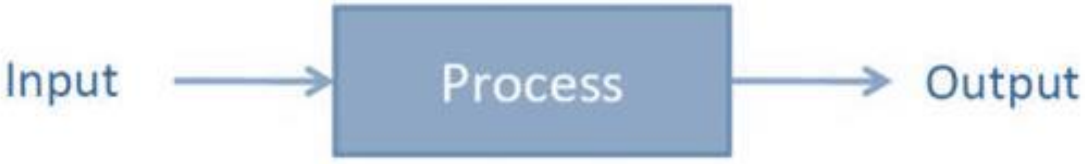
```
do
{
    operation;
    ...
} while (check_condition);
```

DO-WHILE LOOP

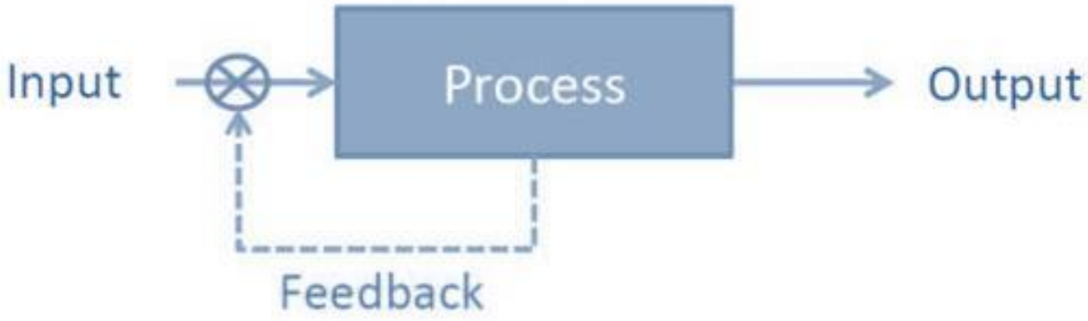
```
a = 0;                    // Set initial value
do
{
   a = a+1;               // Operation in progress
}
while (a <= 10); // Check condition
```

MCU SYSTEMS

OPEN FEEDBACK SYSTEM



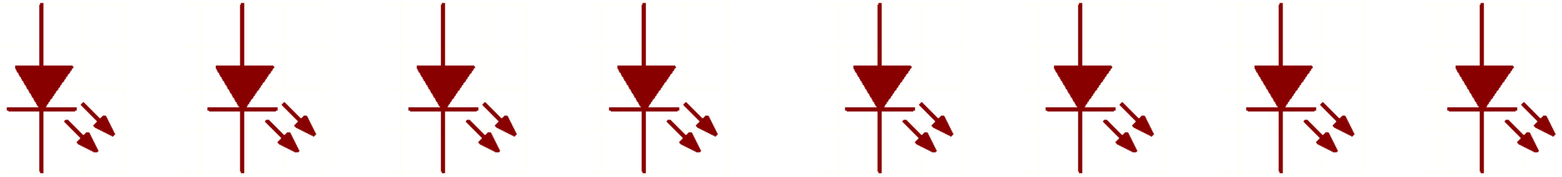
CLOSED-LOOP FEEDBACK SYSTEM



TURN LEDs ON AND OFF

PORTB

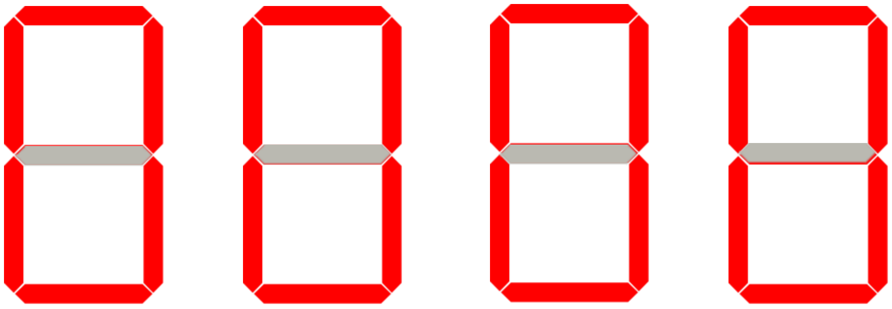
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



PORTB = 0xFF

```
void main()
{
    TRISB = 0x00;
    PORTB=0x00;
    while(1)
    {
        PORTB=0xFF;
    }
}
```

7-SEGMENT DISPLAY



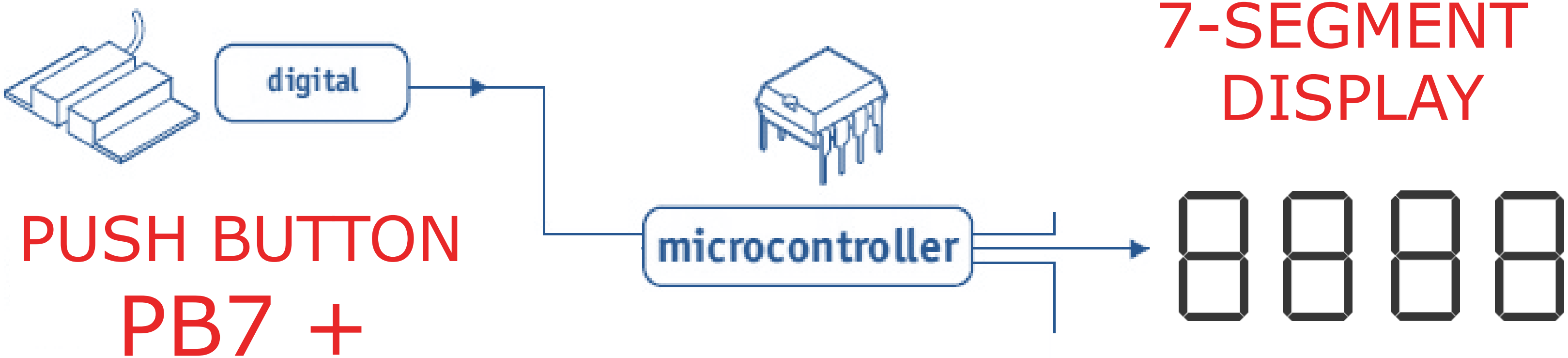
SHOW NUMBER “0”

```
void main()
{
    TRISA = 0x00;
    TRISD = 0x00;
    PORTA=0x00;
    PORTD=0x00;
```

```
while(1)
      {
        PORTA=0x0F;
        PORTD=0x3F;
      }
}
```

MCU SYSTEM

INPUT PROCESS OUTPUT



7-SEGMENT DISPLAY

PUSH BUTTON

PB7 +

PB6 -

0-9

```
int mask(int num) {
 switch (num)
 {
   case 0 : return 0x3F;
   case 1 : return 0x06;
   case 2 : return 0x5B;
   case 3 : return 0x4F;
   case 4 : return 0x66;
   case 5 : return 0x6D;
   case 6 : return 0x7D;
   case 7 : return 0x07;
   case 8 : return 0x7F;
   case 9 : return 0x6F;
 } //case end
}
```

```
void main()
{
int counter1;

TRISA = 0x00;
TRISB = 0xFF;
TRISC = 0x00;
TRISD = 0x00;
PORTA=0x00;
PORTD=0x00;
counter1=0;
while(1)
{
        PORTA=0x0F;
        PORTD=mask(counter1);
        Delay_ms(300);
        if (PORTB.B7  == 1) // button_A: Increase Value
        {
          counter1++;
          Delay_ms(100);
        }
        else if (PORTB.B6  == 1) // button_B: Decrease Value
        {
          counter1--;
          Delay_ms(100);
        }
        if (counter1>9)
        { counter1=0;}
        if (counter1<0)
        { counter1=9;}
   }
}
```

THANK YOU

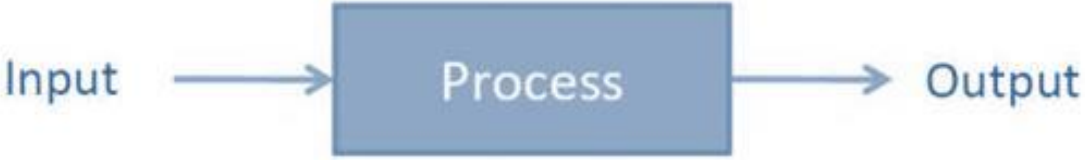
Lecture 8

C LANGUAGE

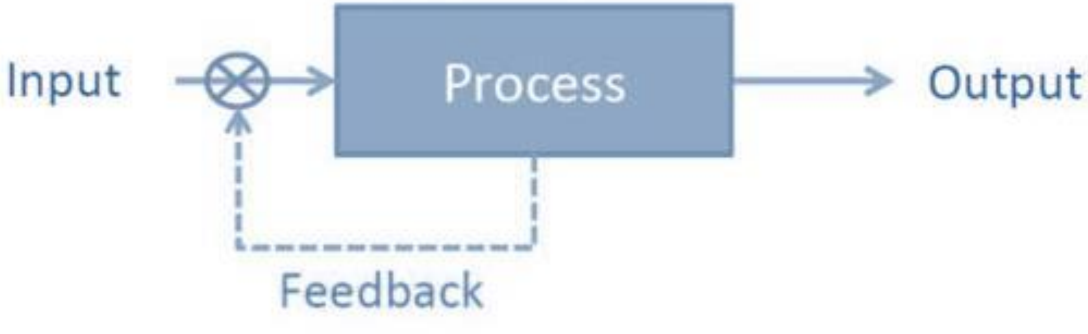
- INPUTS AND OUTPUTS
- BLINK LEDS
- 7-SEGMENT DISPLAY

MCU SYSTEMS

OPEN FEEDBACK SYSTEM



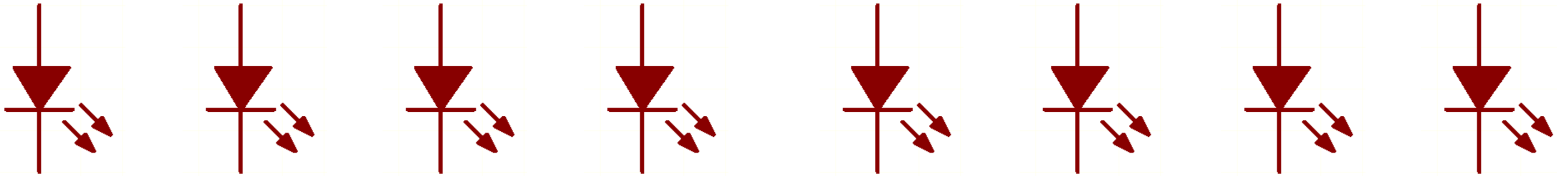
CLOSED-LOOP FEEDBACK SYSTEM



TURN LEDs ON AND OFF

PORTB

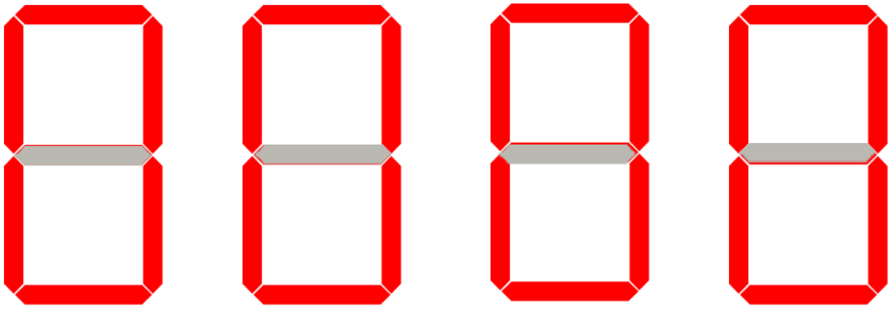
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0



PORTB = 0xFF

```
void main()
{
    TRISB = 0x00;
    PORTB=0x00;
    while(1)
    {
        PORTB=0xFF;
    }
}
```

7-SEGMENT DISPLAY

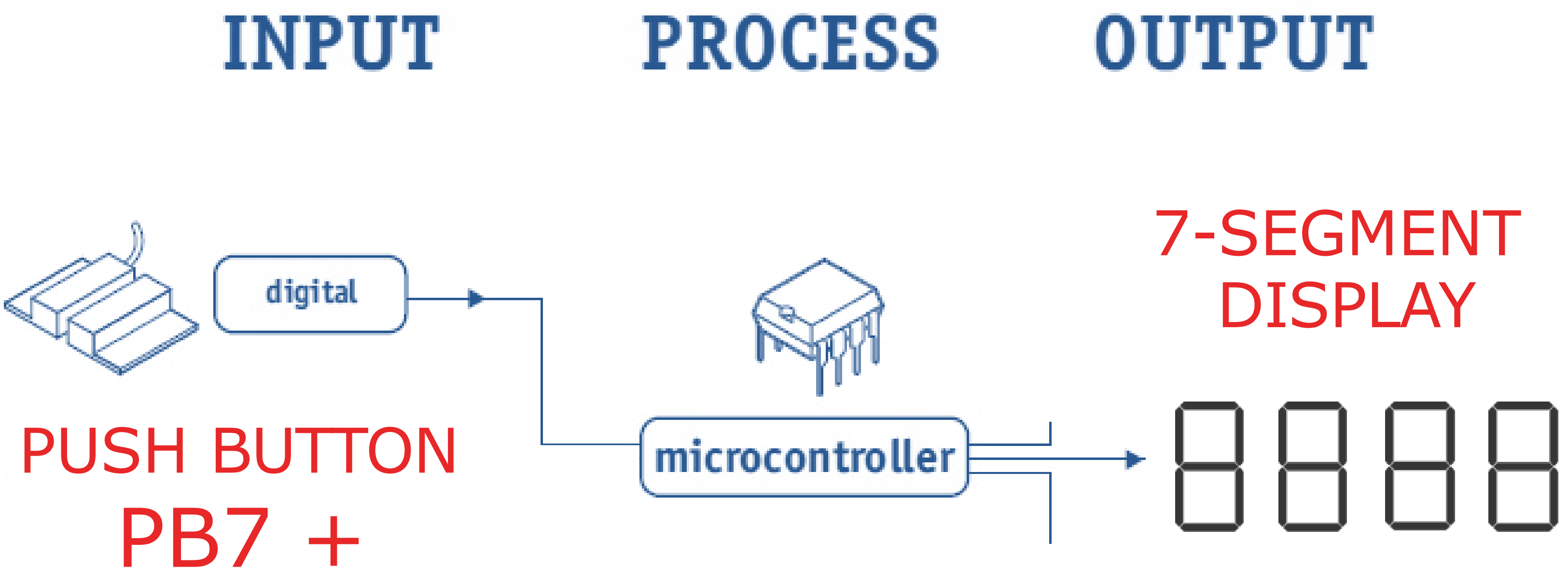


SHOW NUMBER “0”

```
void main()
{
    TRISA = 0x00;
    TRISD = 0x00;
    PORTA=0x00;
    PORTD=0x00;
```

```
while(1)
        {
          PORTA=0xFF;
          PORTD=0x3F;
        }
}
```

MCU SYSTEM



7-SEGMENT DISPLAY

PUSH BUTTON

PB7 +

PB6 -

0-9

```
int mask(int num) {
  switch (num)
  {
    case 0 : return 0x3F;
    case 1 : return 0x06;
    case 2 : return 0x5B;
    case 3 : return 0x4F;
    case 4 : return 0x66;
    case 5 : return 0x6D;
    case 6 : return 0x7D;
    case 7 : return 0x07;
    case 8 : return 0x7F;
    case 9 : return 0x6F;
  } //case end
}
```

```
void main()
{
int counter1;

TRISA = 0x00;
TRISB = 0xFF;
TRISC = 0x00;
TRISD  = 0x00;
PORTA=0;
PORTD=0;
counter1=0;
while(1)
{
        PORTA=0xFF;
        PORTD=mask(counter1);
        Delay_ms(300);
        if (PORTB.B7 == 1) // button_A: Increase Value
        {
          counter1++;
          Delay_ms(100);
        }
        else if (PORTB.B6 == 1) // button_B: Decrease Value
        {
          counter1--;
          Delay_ms(100);
        }
        if (counter1>9)
        { counter1=0;}
        if (counter1<0)
        { counter1=9;}
   }
}
```

THANK YOU

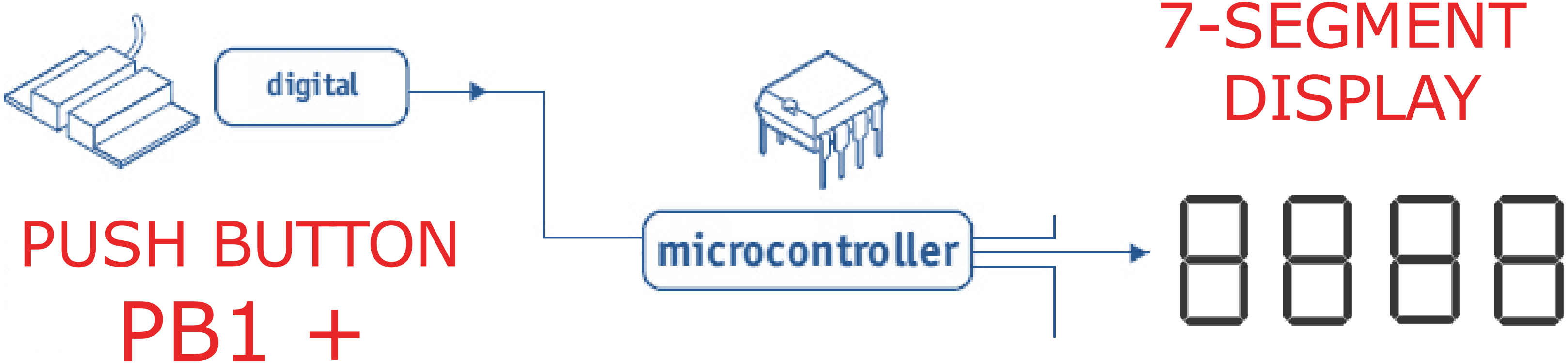
Lecture 9

C LANGUAGE

- Review 7-SEGMENT DISPLAY
- ANALOGUE TO DIGITAL CONVERTER

MCU SYSTEM

INPUT **PROCESS** **OUTPUT**



7-SEGMENT
DISPLAY

PUSH BUTTON
PB1 +
PB0 -

0-9

```
int mask(int num) {
 switch (num)
 {
  case 0 : return 0x3F;
  case 1 : return 0x06;
  case 2 : return 0x5B;
  case 3 : return 0x4F;
  case 4 : return 0x66;
  case 5 : return 0x6D;
  case 6 : return 0x7D;
  case 7 : return 0x07;
  case 8 : return 0x7F;
  case 9 : return 0x6F;
 } //case end
}
```

```
void main()
{
int counter1;

TRISA  =  0x00;
TRISB  =  0xFF;
TRISC  =  0x00;
TRISD  =  0x00;
PORTA=0;
PORTD=0;
counter1=0;
while(1)
{
    PORTA=0xFF;
    PORTD=mask(counter1);
    Delay_ms(300);
    if (PORTB.B1 == 0) // button_A: Increase Value
    {
     counter1++;
     Delay_ms(100);
    }
    else if (PORTB.B0 == 0) // button_B: Decrease Value
    {
     counter1--;
     Delay_ms(100);
    }
    if (counter1>9)
    { counter1=0;}
    if (counter1<0)
    { counter1=9;}
  }
}
```

ANALOGUE TO DIGITAL CONVERTER

ADC

- Analog-to-digital (ADC) converters are among the most widely used devices for data acquisition.
- Digital Computer use binary (discrete) values, but in the physical world is analog (continuous) values.
- Examples of physical quantities: Temperature, Humidity, Pressure, Velocity

ADC

- A physical quantity is converted to electrical (Voltage, Current) signals using a device called transducer (also referred as sensors).
- Sensors for temperature, velocity, pressure, light etc. produce an output that is voltage (or current).

ADC

- Microcontroller → read digital values only.
- Therefore, ADC converter is needed to translate (convert) the analog signals to digital numbers, so that the microcontroller can read and process them

ADC RESOLUTION

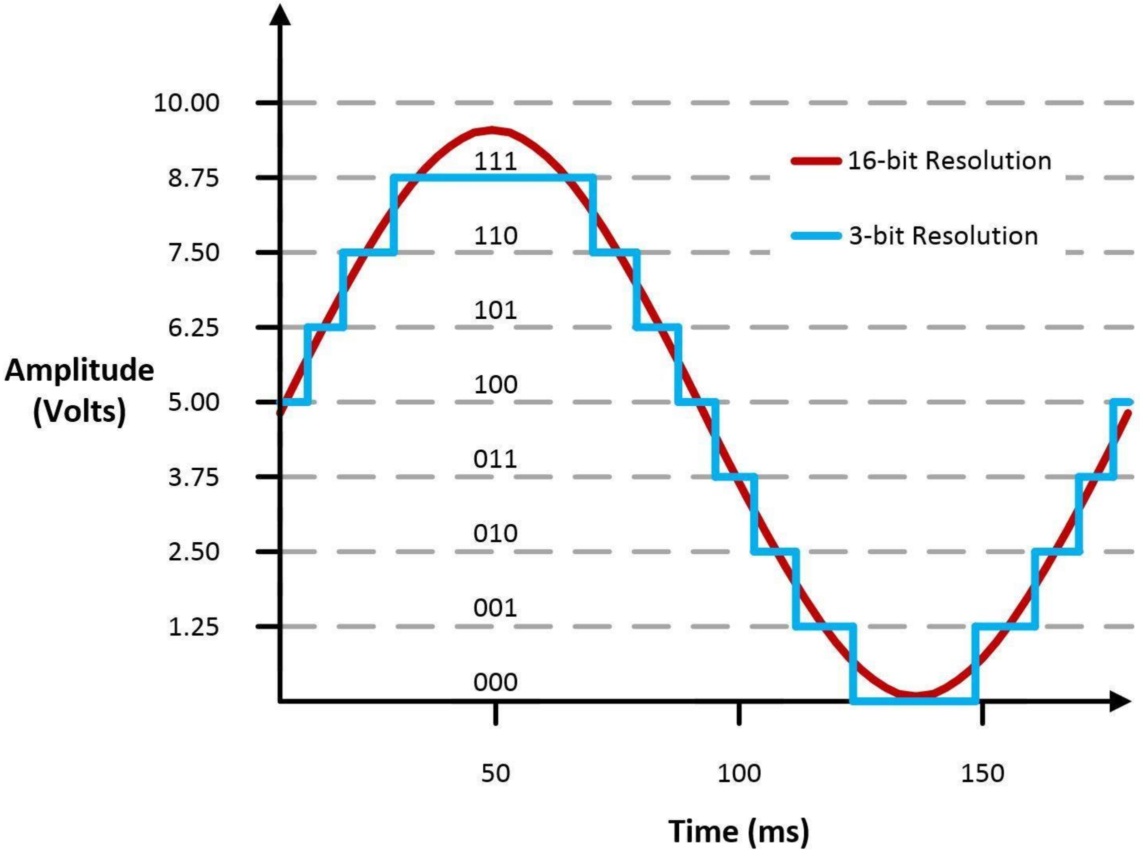
- ADC has *n*-bit resolution, where *n* = 8, 10, 12, 16 or even 24 bits.
- The higher-resolution ADC provides a smaller step size, where *step size* is the smallest change that can be discerned by an ADC.
- Can control the step size with the help of Vref.

n-bit	No. of steps	Step size (mV)
8	$2^8 = 256$	5/256 = 19.53
10	$2^{10} = 1024$	5/1024 = 4.88
12	$2^{12} = 4096$	5/4096 = 1.2
16	$2^{16} = 65,536$	5/65,536 = 0.076

Assuming V_{REF} = 5 V

* **Step Size (Resolution)**: is the smallest change that can be discerned by an ADC

ADC RESOLUTION



ADC REFERENCE VOLTAGE (V_{REF})

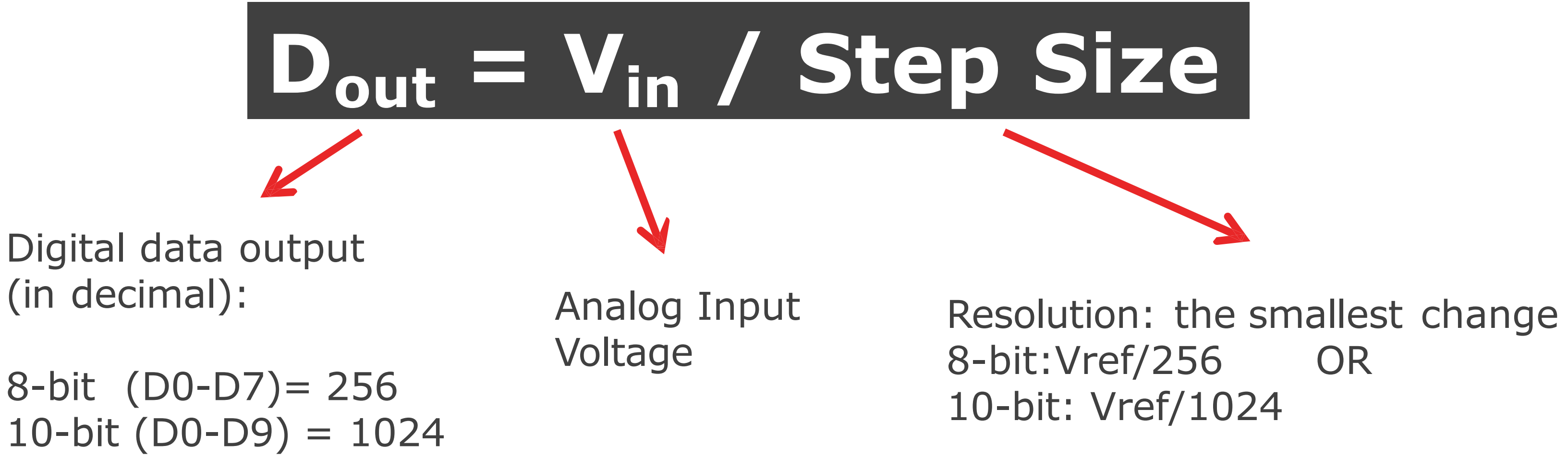
- Vref is an input voltage used for the reference voltage.
- The voltage connected to this pin, along with the resolution of the ADC chip, dictate the step size.
- In some applications, we need the differential reference voltage where Vref = Vref(+) – Vref(-).
- Vref(-) pin is connected to ground, Vref(+) pin is used as the Vref.
- Example: If we need the analog input to be 0 to 5 V, V_{ref} is connected to 5 V

ADC REFERENCE VOLTAGE (V_{REF})

- For an 8-bit ADC, the step size is Vref/256.
  - If Vref = 4 V, the step size is 4 V/256 = 15.62 mV.
  - If need a step size of 10 mV, then Vref = 256 x 10 mV = 2.56 V.
- For the 10-bit ADC, the step size is Vref/1024.
  - If Vref = 5 V, the step size is 5 V/1024 = 4.88 mV.

DIGITAL DATA OUTPUT

- Digital data output:
  - 8-bit ADC: D0-D7
  - 10-bit ADC: D0-D9
- To calculate output voltage:



$$D_{out} = V_{in} \ / \ \text{Step Size}$$

Digital data output
(in decimal):

8-bit (D0-D7)= 256
10-bit (D0-D9) = 1024

Analog Input
Voltage

Resolution: the smallest change
8-bit:Vref/256 OR
10-bit: Vref/1024

DIGITAL DATA OUTPUT

Example:

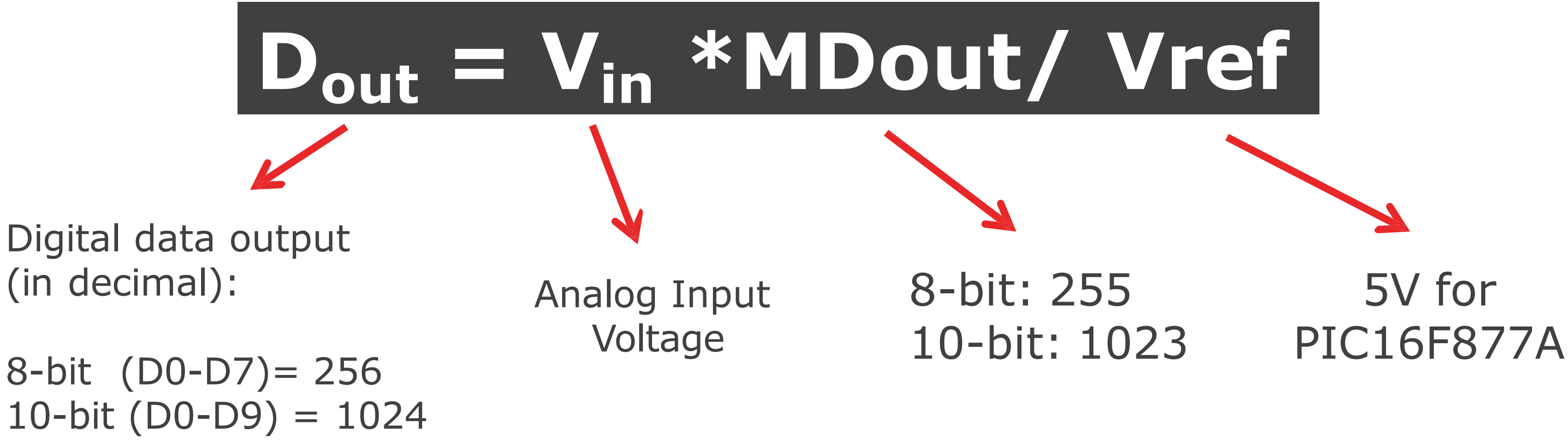
V_{ref} = 2.56, V_{in} = 1.7 V.
Calculate the D0 - D9 output?

Solution:

Step Size = 2.56/1024 = 2.5 mV
Dout = 1.7/2.5 mV = 680 (Decimal)
D0 - D9 = 1010101000

DIGITAL DATA OUTPUT

- Digital data output:
  - 8-bit ADC: D0-D7
  - 10-bit ADC: D0-D9
- To calculate output voltage:



$$D_{out} = V_{in} * MDout / Vref$$

Digital data output (in decimal):

8-bit (D0-D7)= 256
10-bit (D0-D9) = 1024

Analog Input Voltage

8-bit: 255
10-bit: 1023

5V for PIC16F877A

DIGITAL DATA OUTPUT

Example:

V_{ref} = 2.56, V_{in} = 1.7 V.

Calculate the D0 - D9 output?

Solution:

Dout = 1.7*1023/2.56 = 679.36 ≈ 680 (Decimal)

D0 - D9 = 1010101000

ADC USING PIC16F877A

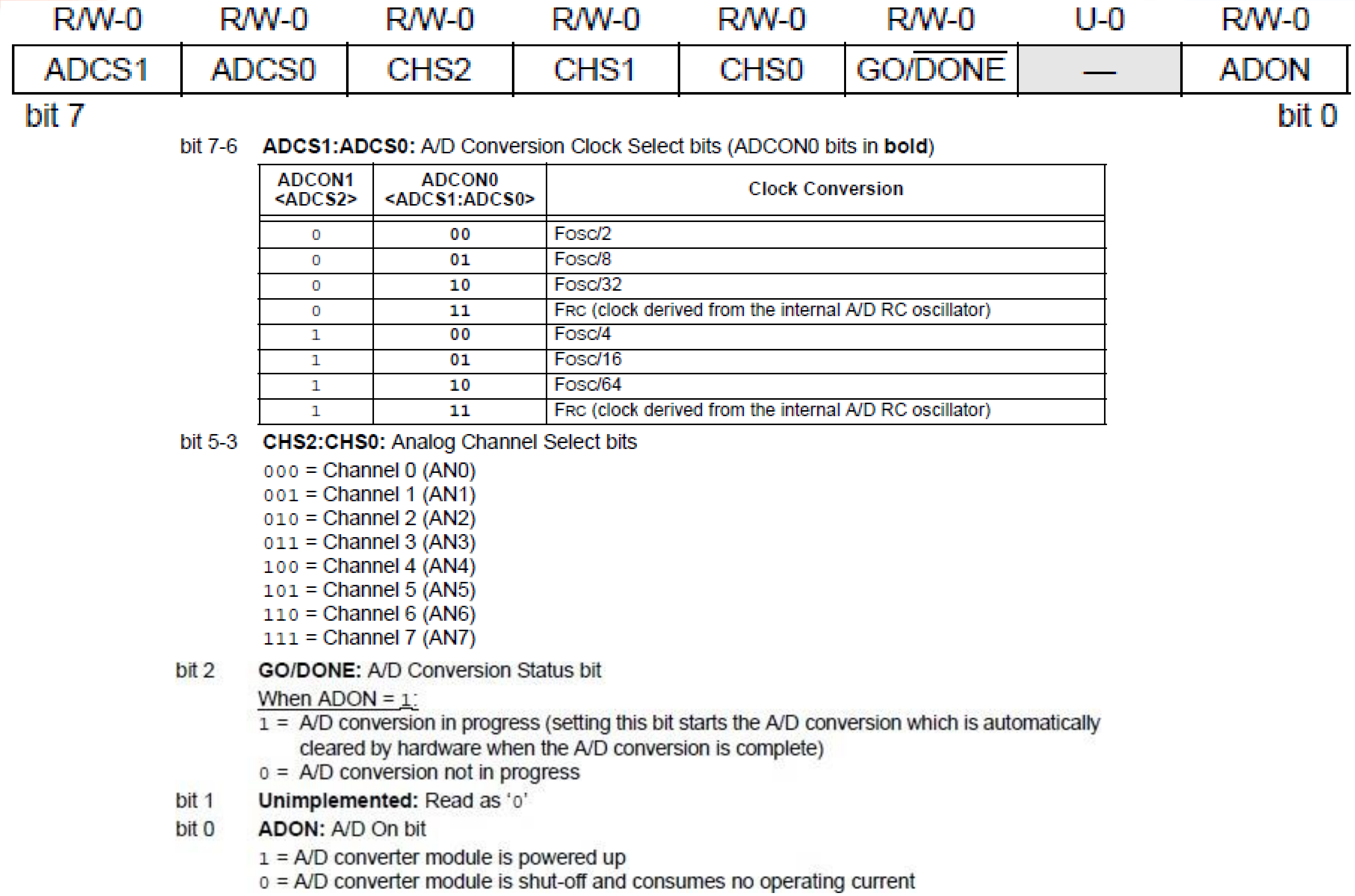
ADC USING PIC16F877A

There are only FOUR registers that you need to understand to configure the ADC. They are ADCON0, ADCON1, ADRESH and ADRESL.

ADC USING PIC16F877A

- The two most important ones are ADCON0 and ADCON1.
- ADRESH and ADRESL are just the registers where the ADC stores the result of the conversion.

ADCON0



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

bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits (ADCON0 bits in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	**00**	FOSC/2
0	**01**	FOSC/8
0	**10**	FOSC/32
0	**11**	FRC (clock derived from the internal A/D RC oscillator)
1	**00**	FOSC/4
1	**01**	FOSC/16
1	**10**	FOSC/64
1	**11**	FRC (clock derived from the internal A/D RC oscillator)

bit 5-3 **CHS2:CHS0:** Analog Channel Select bits

000 = Channel 0 (AN0)
001 = Channel 1 (AN1)
010 = Channel 2 (AN2)
011 = Channel 3 (AN3)
100 = Channel 4 (AN4)
101 = Channel 5 (AN5)
110 = Channel 6 (AN6)
111 = Channel 7 (AN7)

bit 2 **GO/DONE:** A/D Conversion Status bit

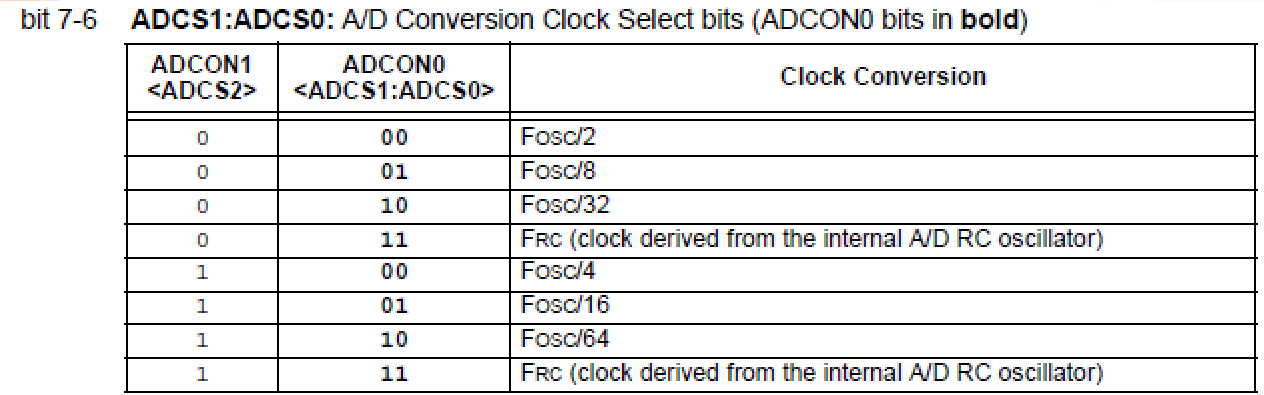
When ADON = 1:
1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)
0 = A/D conversion not in progress

bit 1 **Unimplemented:** Read as '0'

bit 0 **ADON:** A/D On bit

1 = A/D converter module is powered up
0 = A/D converter module is shut-off and consumes no operating current

ADCON0: Conversion Clock Select



bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits (ADCON0 bits in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	**00**	FOSC/2
0	**01**	FOSC/8
0	**10**	FOSC/32
0	**11**	FRC (clock derived from the internal A/D RC oscillator)
1	**00**	FOSC/4
1	**01**	FOSC/16
1	**10**	FOSC/64
1	**11**	FRC (clock derived from the internal A/D RC oscillator)

The user has to select the correct clock conversion. The period must be at least more than 1.6us to obtain an accurate conversion

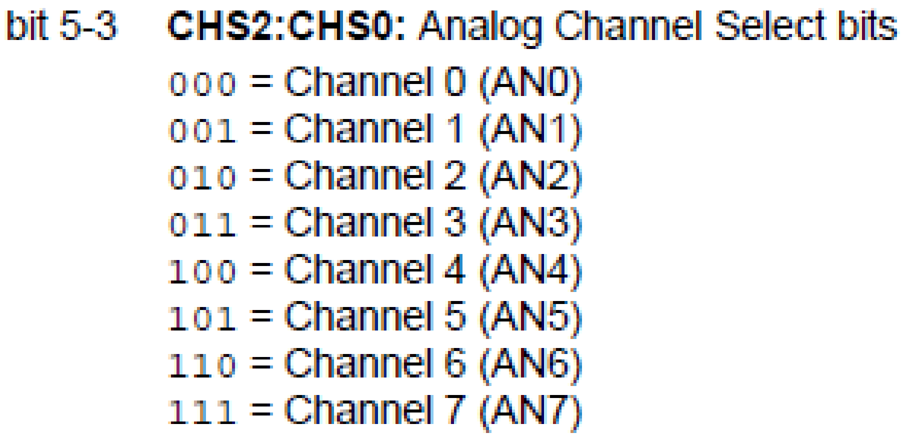
ADCON0: Conversion Clock Select

For example, we use a 8MHz oscillator on the PIC16F877A. So if we select Fosc/4, that’s 2MHz and the period is just 500ns and it’s far less than the 1.6us required.

What if we select Fosc/16? That will give us 0.5MHz and the period is 2us. That is more than 1.6us so it can be selected

Thus, ADCON0 is now 01xx xxxx

ADCON0: Analogue Channel Select



bit 5-3 **CHS2:CHS0:** Analog Channel Select bits

000 = Channel 0 (AN0)
001 = Channel 1 (AN1)
010 = Channel 2 (AN2)
011 = Channel 3 (AN3)
100 = Channel 4 (AN4)
101 = Channel 5 (AN5)
110 = Channel 6 (AN6)
111 = Channel 7 (AN7)

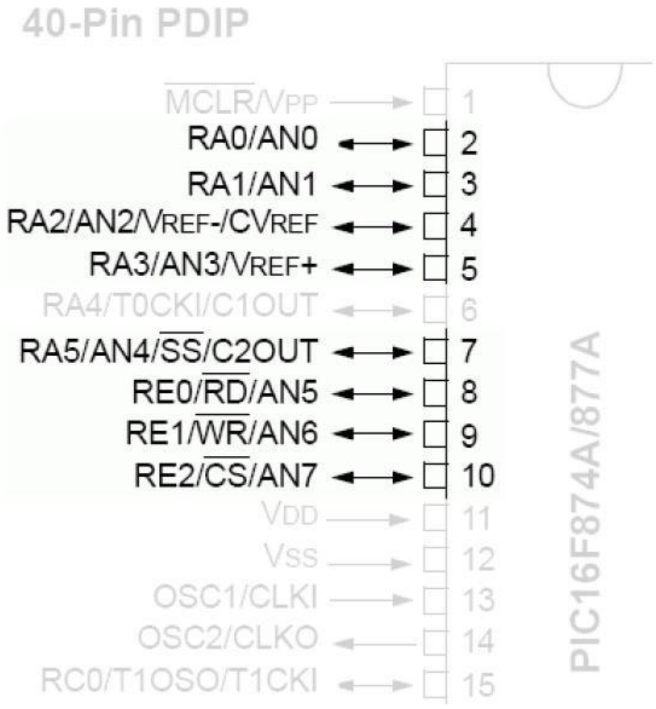
The ADC can only have one input at a time so the user must select which pin to use

ADCON0: Analogue Channel Select

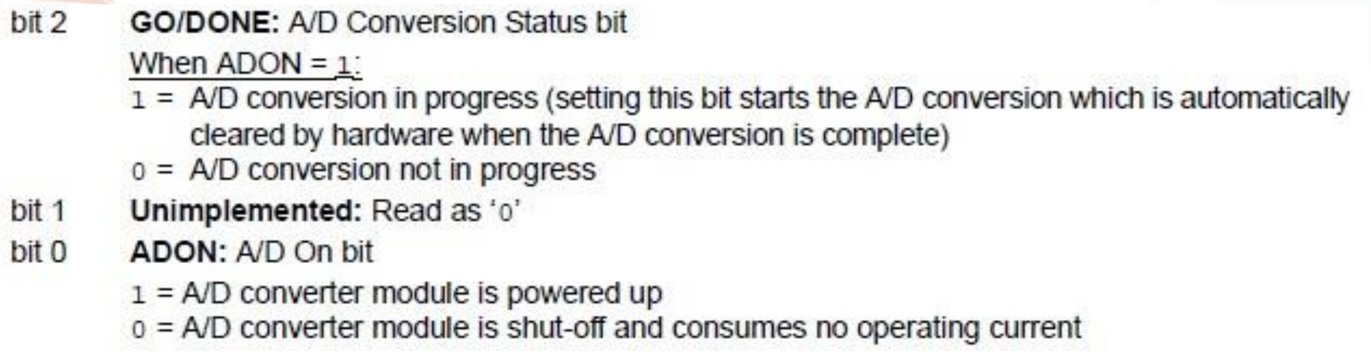
Referring to the PIC16F877A pinout diagram:

These are the available Analog Channels.

If we use Analog Channel 0 (which is PA0), ADCON0 will be set to 0100 0xxx



ADCON0: ADC Initialization



bit 2 **GO/DONE:** A/D Conversion Status bit

When ADON = 1:

1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)

0 = A/D conversion not in progress

bit 1 **Unimplemented:** Read as '0'

bit 0 **ADON:** A/D On bit

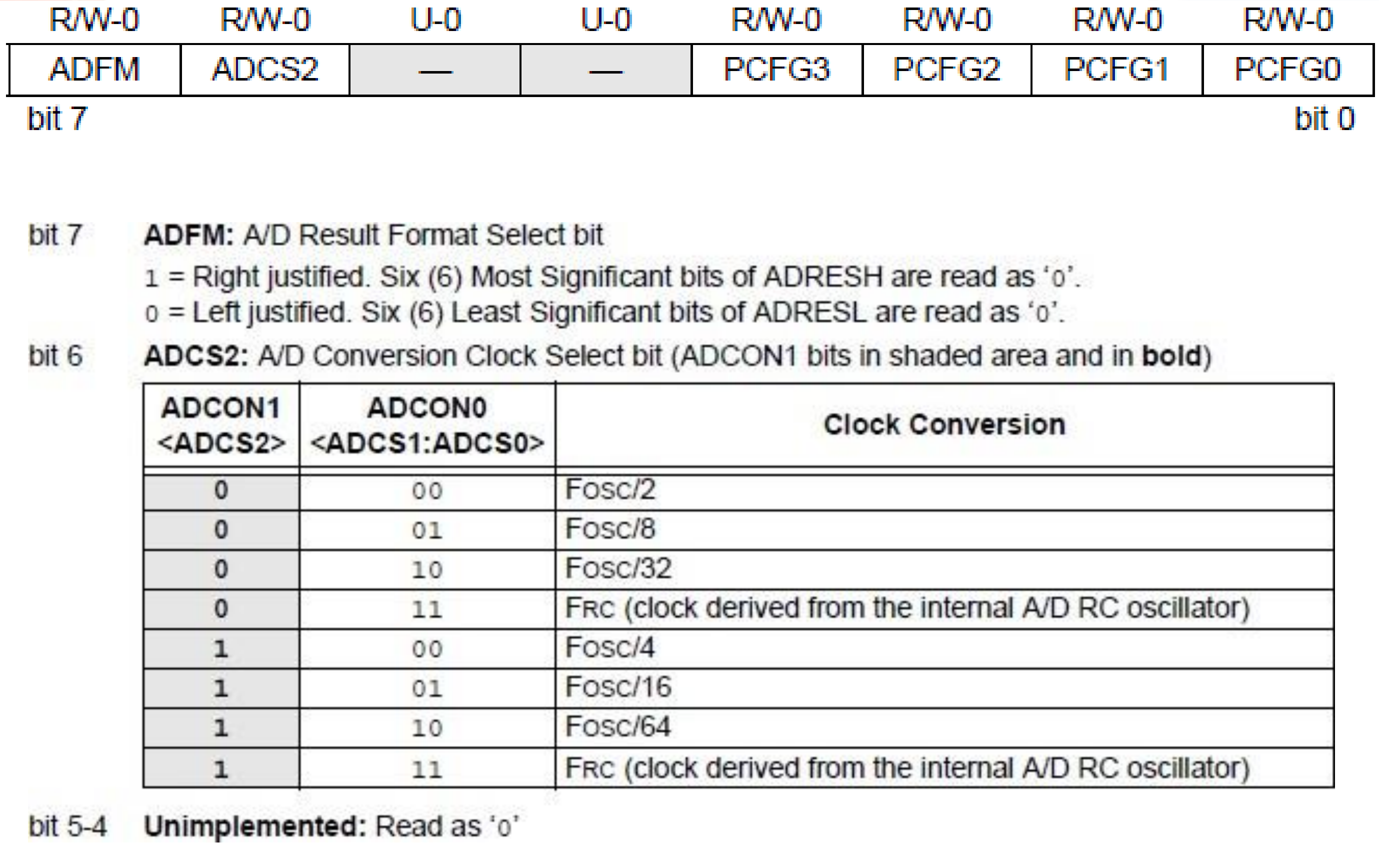
1 = A/D converter module is powered up

0 = A/D converter module is shut-off and consumes no operating current

We set all these bits to 0 because this is just the initialization, the actual program has yet to start (Later in the code we will individually set these bits to enable ADC)

ADCON0 is set to be 0100 0000

ADCON1



R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0

bit 7 bit 0

bit 7 **ADFM:** A/D Result Format Select bit

1 = Right justified. Six (6) Most Significant bits of ADRESH are read as ‘0’.

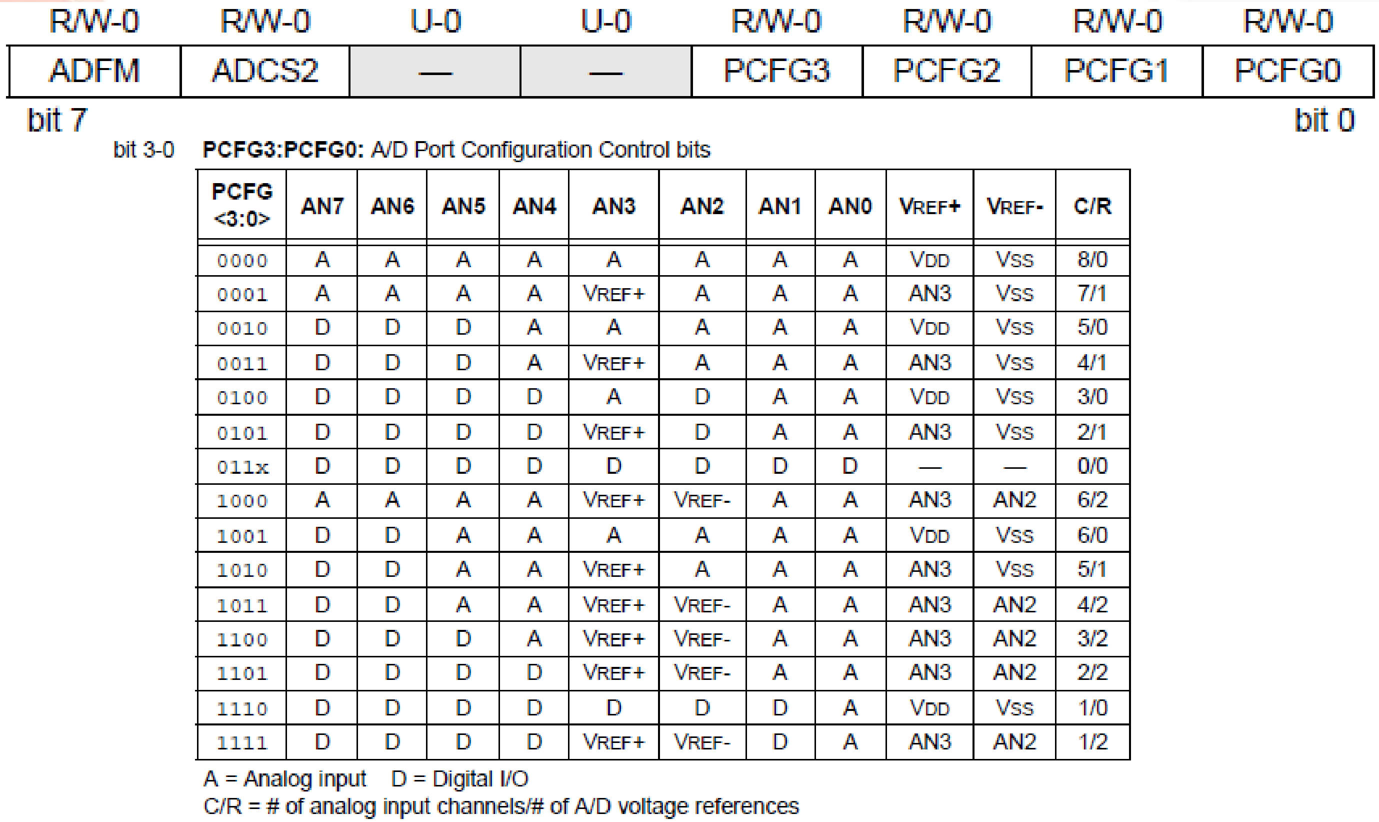
0 = Left justified. Six (6) Least Significant bits of ADRESL are read as ‘0’.

bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

ADCON1 <ADCS2>	**ADCON0 <ADCS1:ADCS0>**	**Clock Conversion**
0	00	Fosc/2
0	01	Fosc/8
0	10	Fosc/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	Fosc/4
1	01	Fosc/16
1	10	Fosc/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

bit 5-4 **Unimplemented:** Read as ‘0’

ADCON1



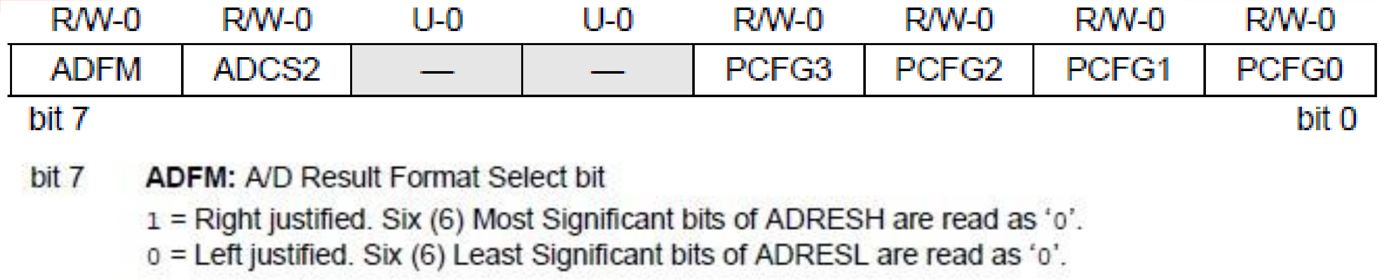
R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O
C/R = # of analog input channels/# of A/D voltage references

ADCON1: A/D Result Format Select



R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

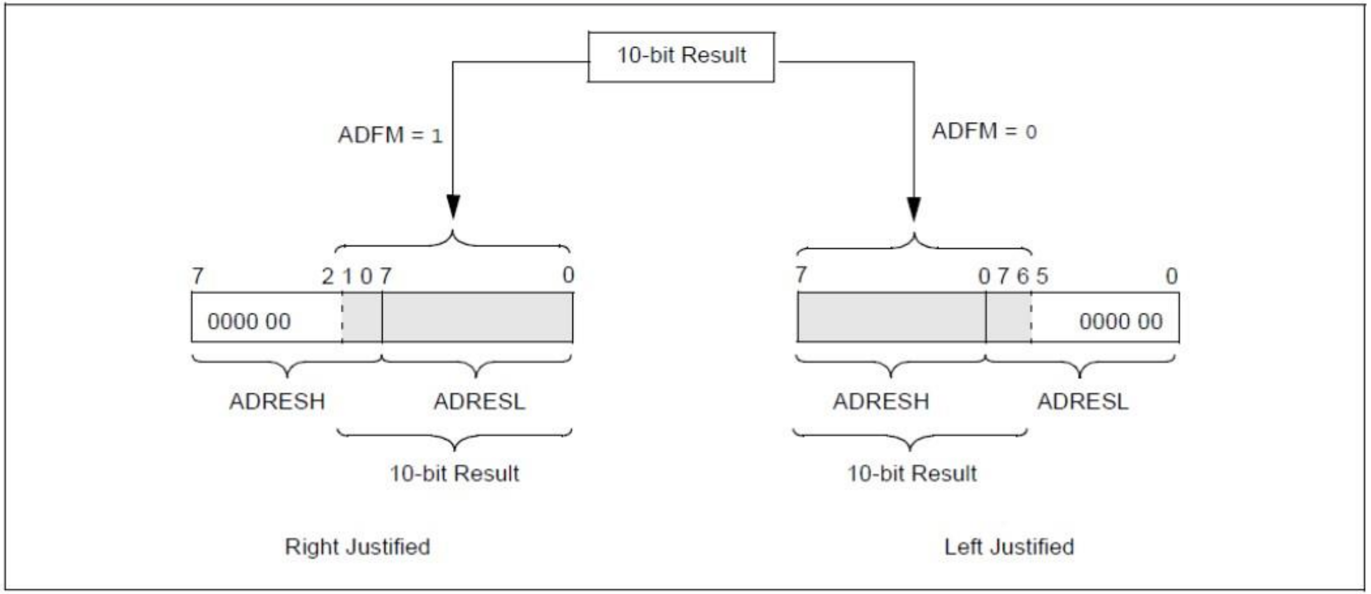
bit 7 **ADFM:** A/D Result Format Select bit

1 = Right justified. Six (6) Most Significant bits of ADRESH are read as ‘0’.
0 = Left justified. Six (6) Least Significant bits of ADRESL are read as ‘0’.

The ADFM bit determines how the result of the ADC is justified. Since the ADC on the PIC16F877A has 10-bits of resolution, logically a single register (that has 8 bits) is not enough to contain the 10-bits result. Therefore, two registers are required to store the results. ADRESH and ADRESL (H is the high byte while L is the low byte).

ADCON1: A/D Result Format Select

FIGURE 11-4: A/D RESULT JUSTIFICATION

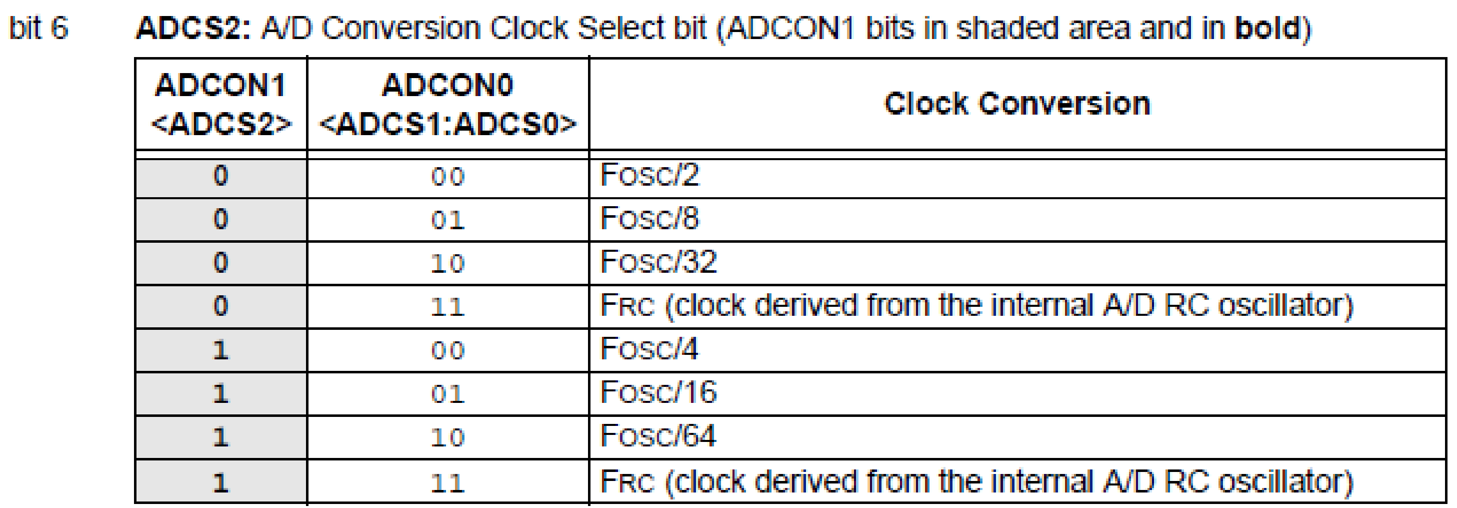


Two registers will allow us to store up to 16 bits, but since there are only 10 bits, we have the flexibility to align it right justified or left justified

ADCON1: A/D Result Format Select

If the application doesn’t need the 10-bit accuracy, 8 bits is more than enough. So we can just take the result in ADRESH and ignore the remaining two least significant bits in ADRESL (we cannot ignore the two highest significant bit because that will cause the result to be inaccurate). That makes it easier to move values to other registers. Yes, the accuracy of the result will be slightly affected but it’s not critical in applications where we don’t need accuracy.
The value of ADCON1 is 0xxx xxxx

ADCON1: Conversion Clock Select

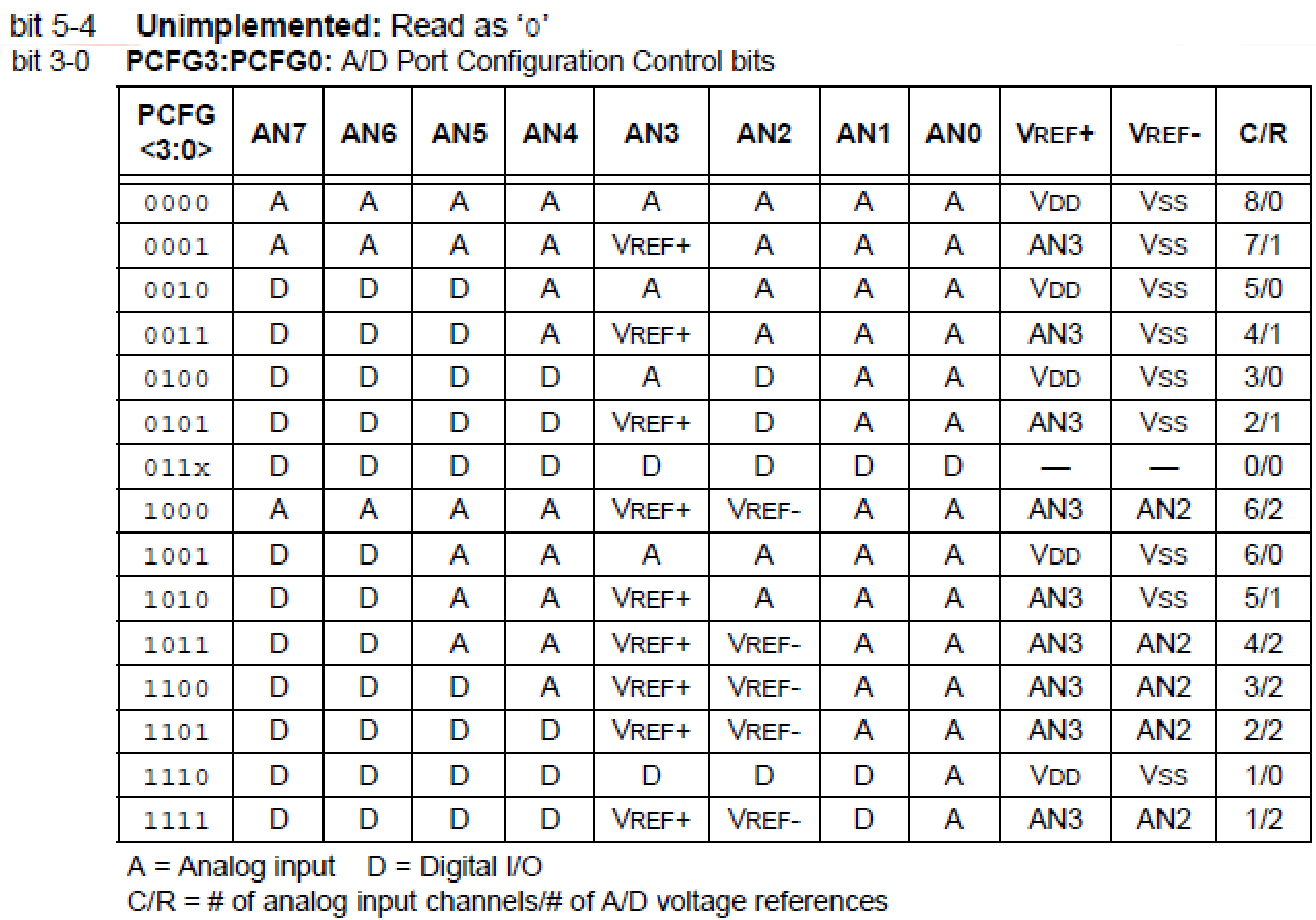


bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

ADCON1 <ADCS2>	**ADCON0 <ADCS1:ADCS0>**	**Clock Conversion**
0	00	FOSC/2
0	01	FOSC/8
0	10	FOSC/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	FOSC/4
1	01	FOSC/16
1	10	FOSC/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

Next is the ADCS2 bit. We agreed that Fosc/16 is adequate, thus we selected it in ADCON0. But for Fosc/16, we need to set the ADCS2 bit in ADCON1 as well. The value of ADCON1 will be 01xx xxxx.

ADCON1: Port Configuration Control



bit 5-4 **Unimplemented:** Read as '0'

bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O

C/R = # of analog input channels/# of A/D voltage references

ADCON1: Port Configuration Control

The most important part of the ADC configuration is to select the mode for each Analog channel. As shown before, we have Analog Channels 0 to 7. All these inputs can either be set to analog or digital. Referring to the table above, if we don’t need any analog inputs and require more digital pins (let’s say for a few LCDs), we can set the PCFG3:0 bits to be 011x. But in the case we do need the Analog inputs, we will set all of them to be in analog mode.

Therefore, the final value for ADCON1 is 0100 0000

ADCON1: Port Configuration Control

One important thing to note is that we’ve selected Vdd as the Vref+ and Vss as the Vref-, that means that our conversion range is from 0V to 5V. If we need it to be other than that, we can set a custom Vref value by choosing other configurations of PCFG3:0.

```
void ADC_initVal()
{
 ADCON0=01000000;
 ADCON1=01000000;
}

void main()
{
unsigned int adc_Dout;
char txt[7];
TRISA = 0x01;
ADC_initVal();
UART1_Init(9600);
Delay_ms(100);
```

```
while(1)
{
      ADCON0=01000001;
      adc_Dout = ADC_Read(0);
      Delay_ms(200);
      IntToStr(adc_Dout, txt);
      UART1_Write_Text("ADC:");
      UART1_Write_Text(txt);
      UART1_Write(13);
      UART1_Write(10);
}
}
```

```
void ADC_initVal()
{
 ADCON0=01000000;
 ADCON1=01000000;
}

void main()
{
unsigned int adc_Dout;
char txt[7];
TRISA = 0x01;
ADC_initVal();
UART1_Init(9600);
Delay_ms(100);
```

```
while(1)
{
        ADCON0=01000001;
        adc_Dout = ADC_Read(0);
        Delay_ms(200);
        IntToStr(adc_Dout, txt);
        UART1_Write_Text("ADC:");
        UART1_Write_Text(txt);
        UART1_Write(13);
        UART1_Write(10);
}
}
```

```
void main()
{
    unsigned int adc_Dout;
    char txt[7];
    TRISA = 0x01;
    UART1_Init(9600);
    Delay_ms(100);
    while(1)
    {     adc_Dout = ADC_Read(0);
          Delay_ms(200);
          IntToStr(adc_Dout, txt);
          UART1_Write_Text("ADC:");
          UART1_Write_Text(txt);
          UART1_Write(13);
          UART1_Write(10);
    }
}
```

THANK YOU

Lecture 10

C LANGUAGE

- Review and continue ANALOGUE TO DIGITAL CONVERTER

ANALOGUE TO DIGITAL CONVERTER

ADC

- Analog-to-digital (ADC) converters are among the most widely used devices for data acquisition.
- Digital Computer use binary (discrete) values, but in the physical world is analog (continuous) values.
- Examples of physical quantities: Temperature, Humidity, Pressure, Velocity

ADC

- A physical quantity is converted to electrical (Voltage, Current) signals using a device called transducer (also referred as sensors).
- Sensors for temperature, velocity, pressure, light etc. produce an output that is voltage (or current).

ADC

- Microcontroller → read digital values only.
- Therefore, ADC converter is needed to translate (convert) the analog signals to digital numbers, so that the microcontroller can read and process them

ADC RESOLUTION

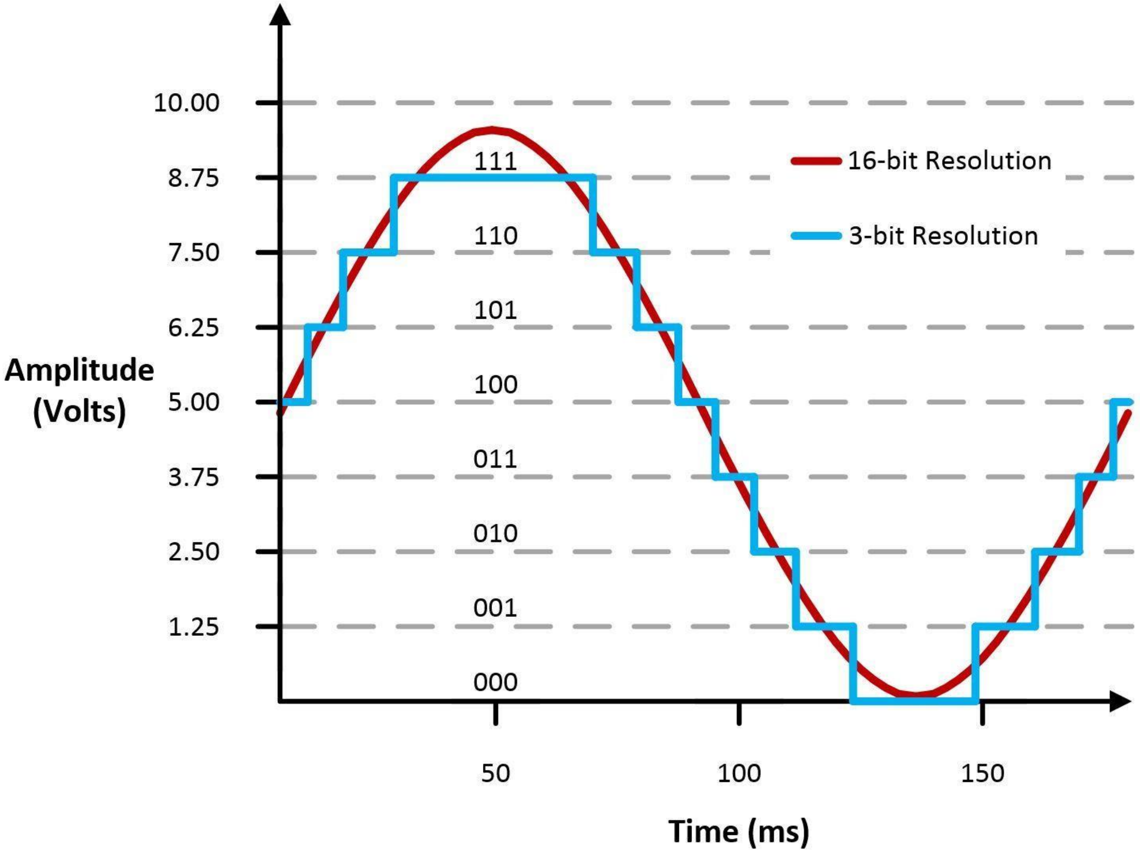
- ADC has *n*-bit resolution, where *n* = 8, 10, 12, 16 or even 24 bits.
- The higher-resolution ADC provides a smaller step size, where *step size* is the smallest change that can be discerned by an ADC.
- Can control the step size with the help of Vref.

n-bit	No. of steps	Step size (mV)
8	$2^8 = 256$	5/256 = 19.53
10	$2^{10} = 1024$	5/1024 = 4.88
12	$2^{12} = 4096$	5/4096 = 1.2
16	$2^{16} = 65,536$	5/65,536 = 0.076

Assuming V_{REF} = 5 V

* **Step Size (Resolution)**: is the smallest change that can be discerned by an ADC

ADC RESOLUTION



ADC REFERENCE VOLTAGE (V_{REF})

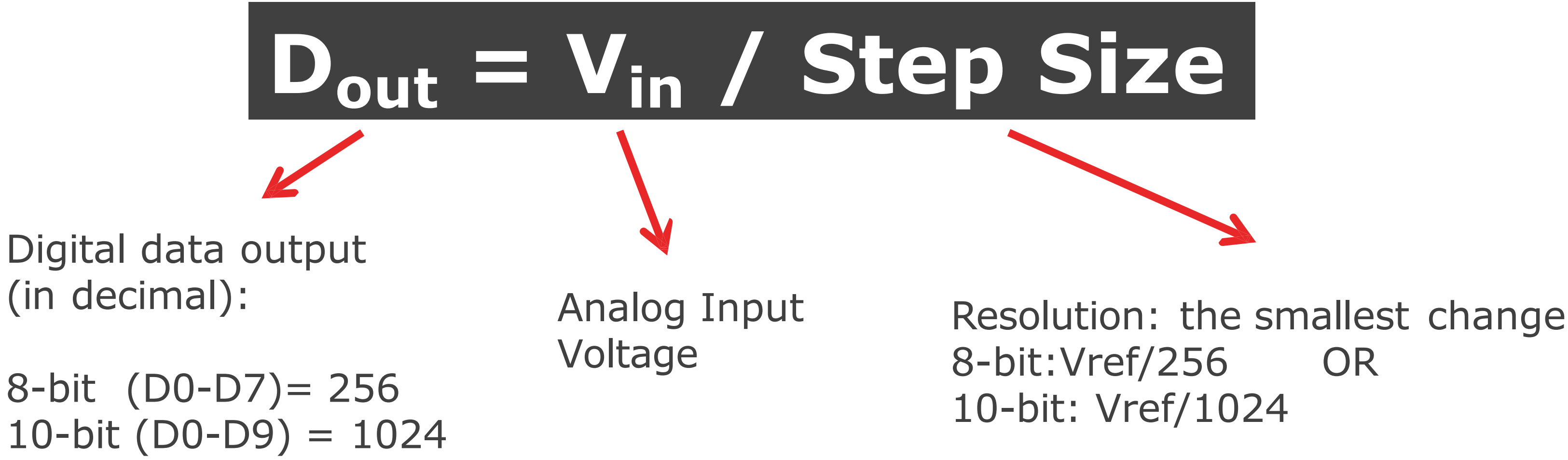
- Vref is an input voltage used for the reference voltage.
- The voltage connected to this pin, along with the resolution of the ADC chip, dictate the step size.
- In some applications, we need the differential reference voltage where Vref = Vref(+) – Vref(-).
- Vref(-) pin is connected to ground, Vref(+) pin is used as the Vref.
- Example: If we need the analog input to be 0 to 5 V, V_{ref} is connected to 5 V

ADC REFERENCE VOLTAGE (V_{REF})

- For an 8-bit ADC, the step size is Vref/256.
  - If Vref = 4 V, the step size is 4 V/256 = 15.62 mV.
  - If need a step size of 10 mV, then Vref = 256 x 10 mV = 2.56 V.
- For the 10-bit ADC, the step size is Vref/1024.
  - If Vref = 5 V, the step size is 5 V/1024 = 4.88 mV.

DIGITAL DATA OUTPUT

- Digital data output:
  - 8-bit ADC: D0-D7
  - 10-bit ADC: D0-D9
- To calculate output voltage:



$$D_{out} = V_{in} \ / \ \text{Step Size}$$

Digital data output
(in decimal):

8-bit (D0-D7)= 256
10-bit (D0-D9) = 1024

Analog Input
Voltage

Resolution: the smallest change
8-bit:Vref/256 OR
10-bit: Vref/1024

DIGITAL DATA OUTPUT

Example:

V_{ref} = 2.56, V_{in} = 1.7 V.

Calculate the D0 - D9 output?

Solution:

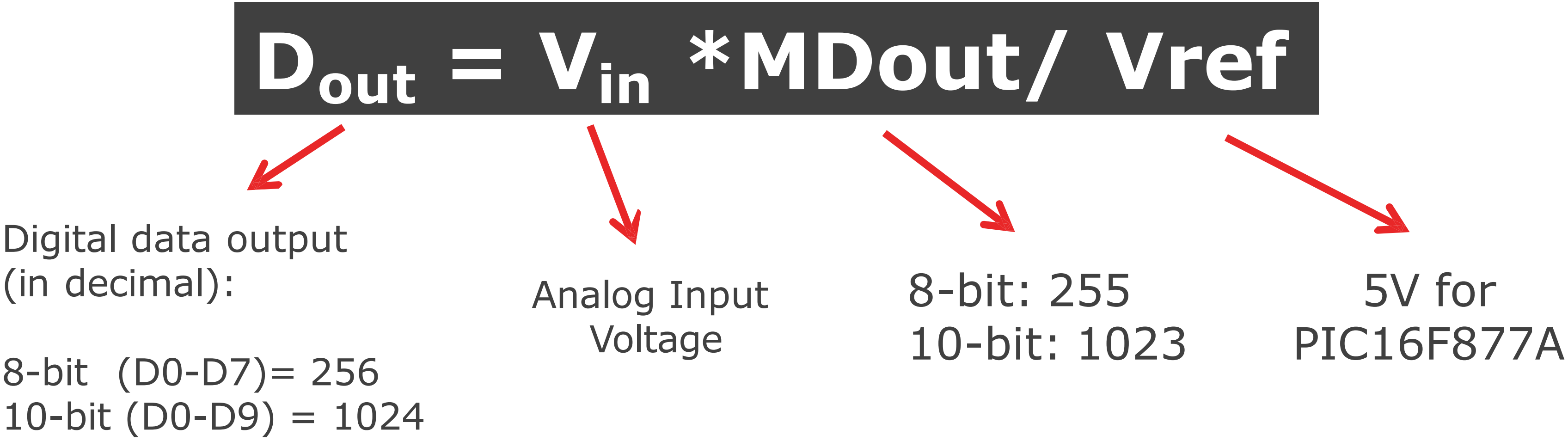
Step Size = 2.56/1024 = 2.5 mV

Dout = 1.7/2.5 mV = 680 (Decimal)

D0 - D9 = 1010101000

DIGITAL DATA OUTPUT

- Digital data output:
  - 8-bit ADC: D0-D7
  - 10-bit ADC: D0-D9
- To calculate output voltage:



$$D_{out} = V_{in} *MDout/ Vref$$

Digital data output (in decimal):

8-bit (D0-D7)= 256
10-bit (D0-D9) = 1024

Analog Input Voltage

8-bit: 255
10-bit: 1023

5V for PIC16F877A

DIGITAL DATA OUTPUT

Example:

V_{ref} = 2.56, V_{in} = 1.7 V.

Calculate the D0 - D9 output?

Solution:

Dout = 1.7*1023/2.56 = 679.36 ≈ 680 (Decimal)

D0 - D9 = 1010101000

ADC USING PIC16F877A

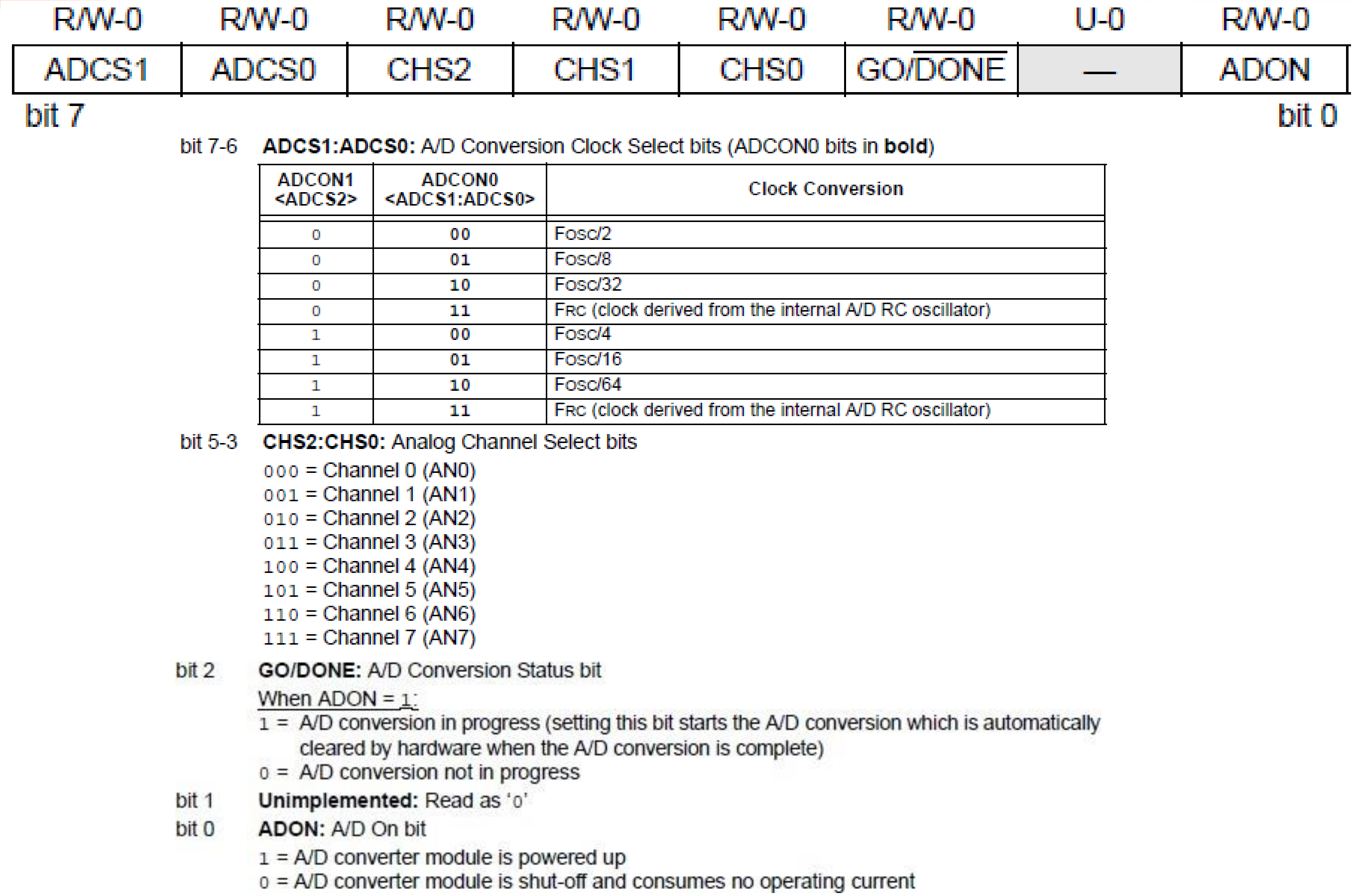
ADC USING PIC16F877A

There are only FOUR registers that you need to understand to configure the ADC. They are ADCON0, ADCON1, ADRESH and ADRESL.

ADC USING PIC16F877A

- The two most important ones are ADCON0 and ADCON1.
- ADRESH and ADRESL are just the registers where the ADC stores the result of the conversion.

ADCON0



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

bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits (ADCON0 bits in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	**00**	Fosc/2
0	**01**	Fosc/8
0	**10**	Fosc/32
0	**11**	FRC (clock derived from the internal A/D RC oscillator)
1	**00**	Fosc/4
1	**01**	Fosc/16
1	**10**	Fosc/64
1	**11**	FRC (clock derived from the internal A/D RC oscillator)

bit 5-3 **CHS2:CHS0:** Analog Channel Select bits

000 = Channel 0 (AN0)
001 = Channel 1 (AN1)
010 = Channel 2 (AN2)
011 = Channel 3 (AN3)
100 = Channel 4 (AN4)
101 = Channel 5 (AN5)
110 = Channel 6 (AN6)
111 = Channel 7 (AN7)

bit 2 **GO/DONE:** A/D Conversion Status bit

When ADON = 1:

1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)

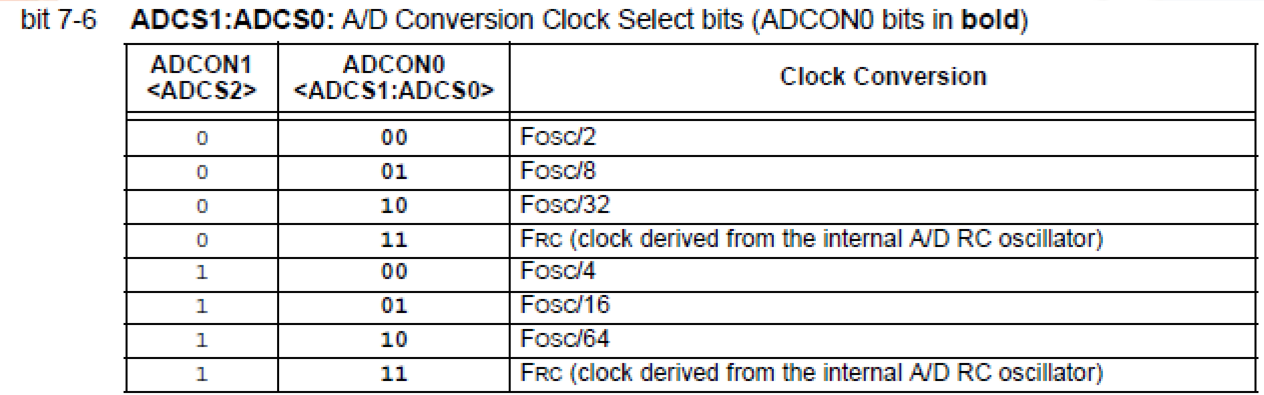
0 = A/D conversion not in progress

bit 1 **Unimplemented:** Read as '0'

bit 0 **ADON:** A/D On bit

1 = A/D converter module is powered up
0 = A/D converter module is shut-off and consumes no operating current

ADCON0: Conversion Clock Select



bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits (ADCON0 bits in **bold**)

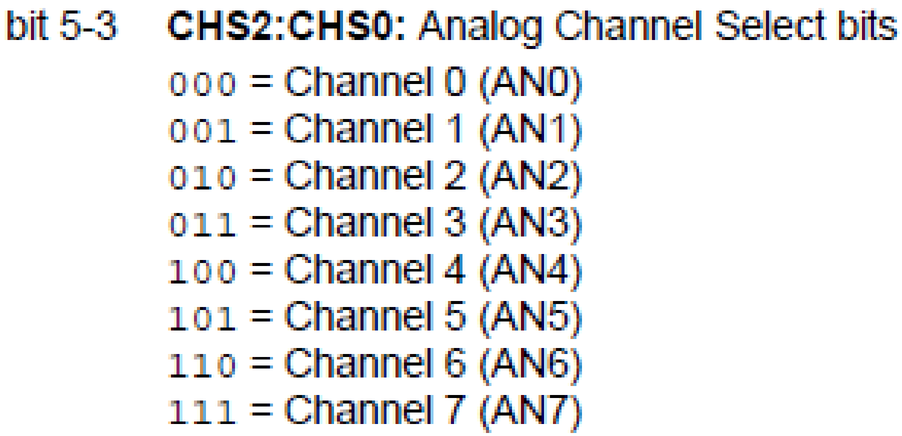
ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	Fosc/2
0	01	Fosc/8
0	10	Fosc/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	Fosc/4
1	01	Fosc/16
1	10	Fosc/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

The user has to select the correct clock conversion. The period must be at least more than 1.6us to obtain an accurate conversion

ADCON0: Conversion Clock Select

For example, we use a 8MHz oscillator on the PIC16F877A. So if we select Fosc/4, that's 2MHz and the period is just 500ns and it's far less than the 1.6us required.
What if we select Fosc/16? That will give us 0.5MHz and the period is 2us. That is more than 1.6us so it can be selected
Thus, ADCON0 is now 01xx xxxx

ADCON0: Analogue Channel Select



bit 5-3 **CHS2:CHS0:** Analog Channel Select bits

`000` = Channel 0 (AN0)
`001` = Channel 1 (AN1)
`010` = Channel 2 (AN2)
`011` = Channel 3 (AN3)
`100` = Channel 4 (AN4)
`101` = Channel 5 (AN5)
`110` = Channel 6 (AN6)
`111` = Channel 7 (AN7)

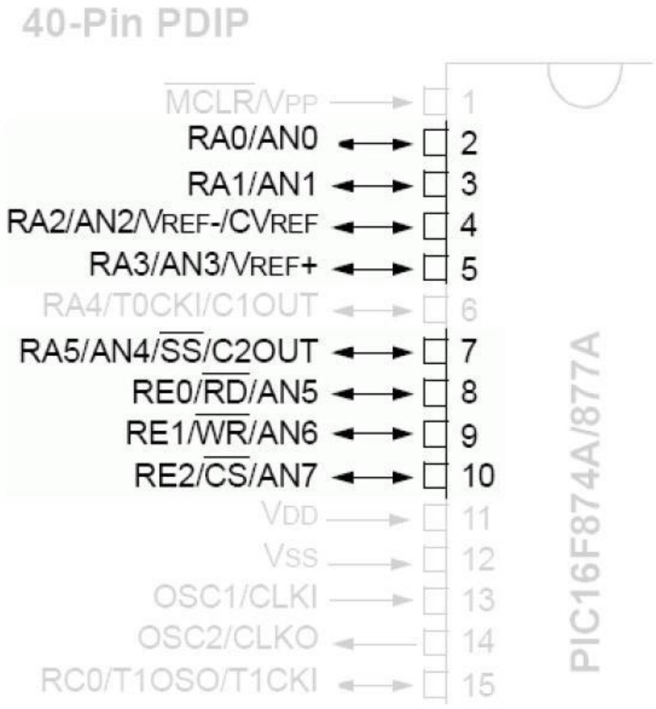
The ADC can only have one input at a time so the user must select which pin to use

ADCON0: Analogue Channel Select

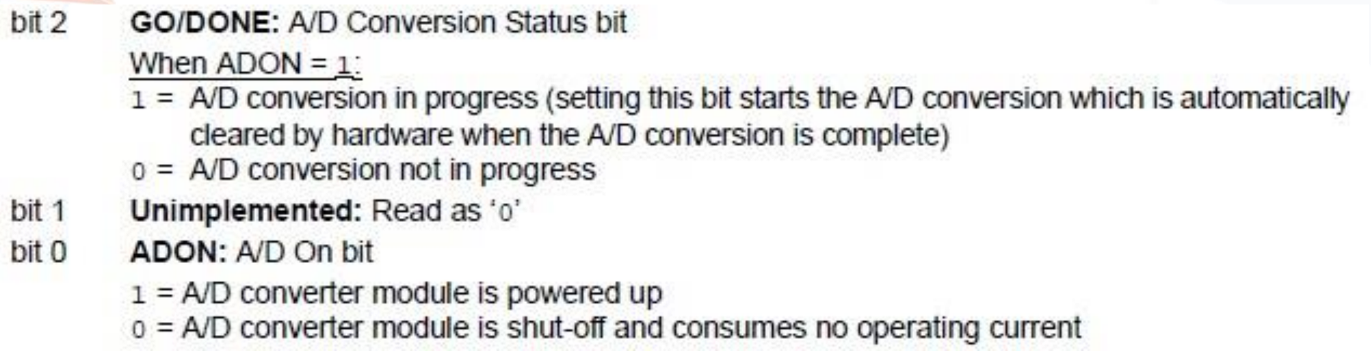
Referring to the PIC16F877A pinout diagram:

These are the available Analog Channels.

If we use Analog Channel 0 (which is PA0), ADCON0 will be set to 0100 0xxx



ADCON0: ADC Initialization



bit 2 **GO/DONE:** A/D Conversion Status bit

When ADON = 1:

1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)

0 = A/D conversion not in progress

bit 1 **Unimplemented:** Read as '0'

bit 0 **ADON:** A/D On bit

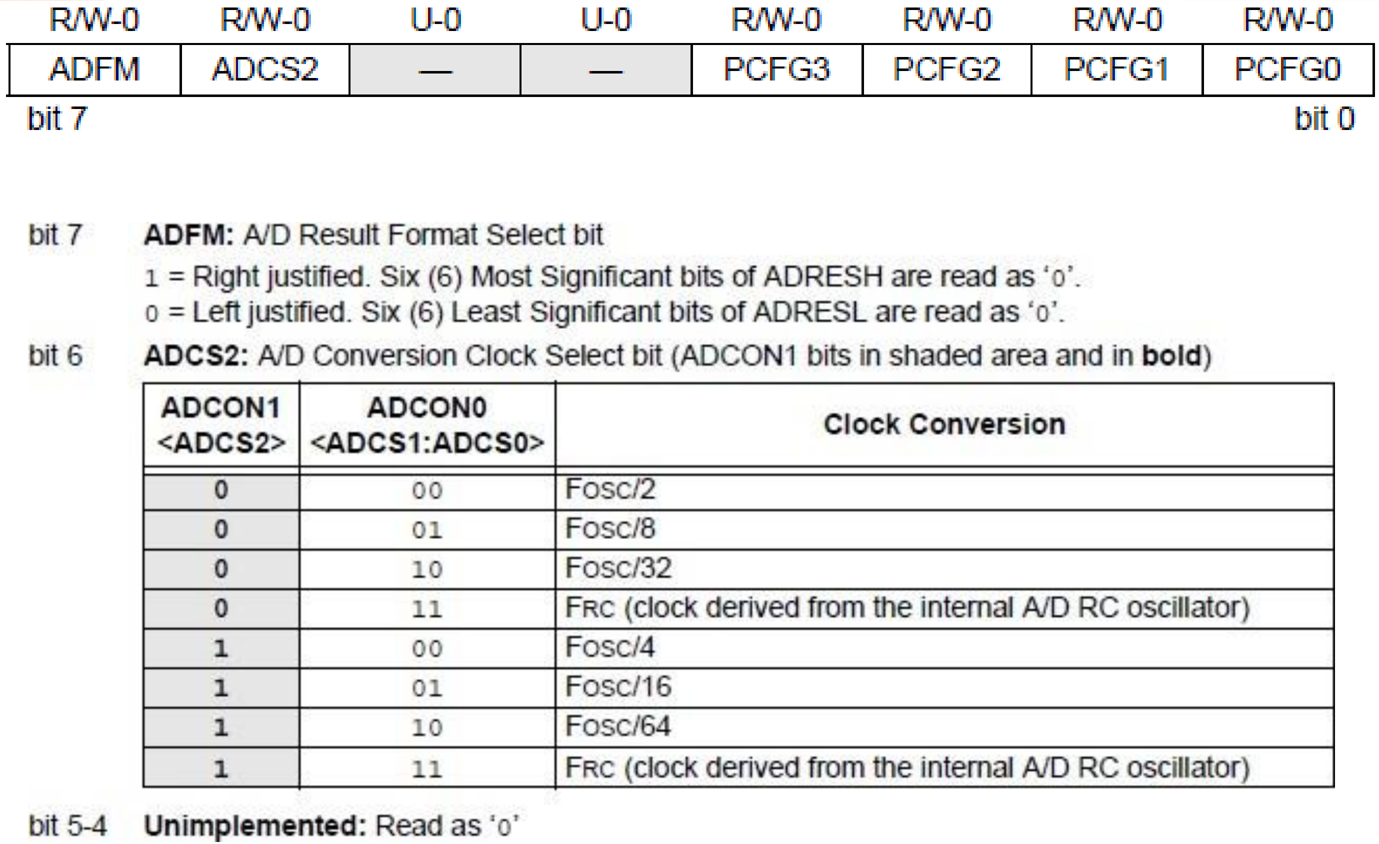
1 = A/D converter module is powered up

0 = A/D converter module is shut-off and consumes no operating current

We set all these bits to 0 because this is just the initialization, the actual program has yet to start (Later in the code we will individually set these bits to enable ADC)

ADCON0 is set to be 0100 0000

ADCON1



R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0

bit 7 bit 0

bit 7 **ADFM:** A/D Result Format Select bit

1 = Right justified. Six (6) Most Significant bits of ADRESH are read as '0'.

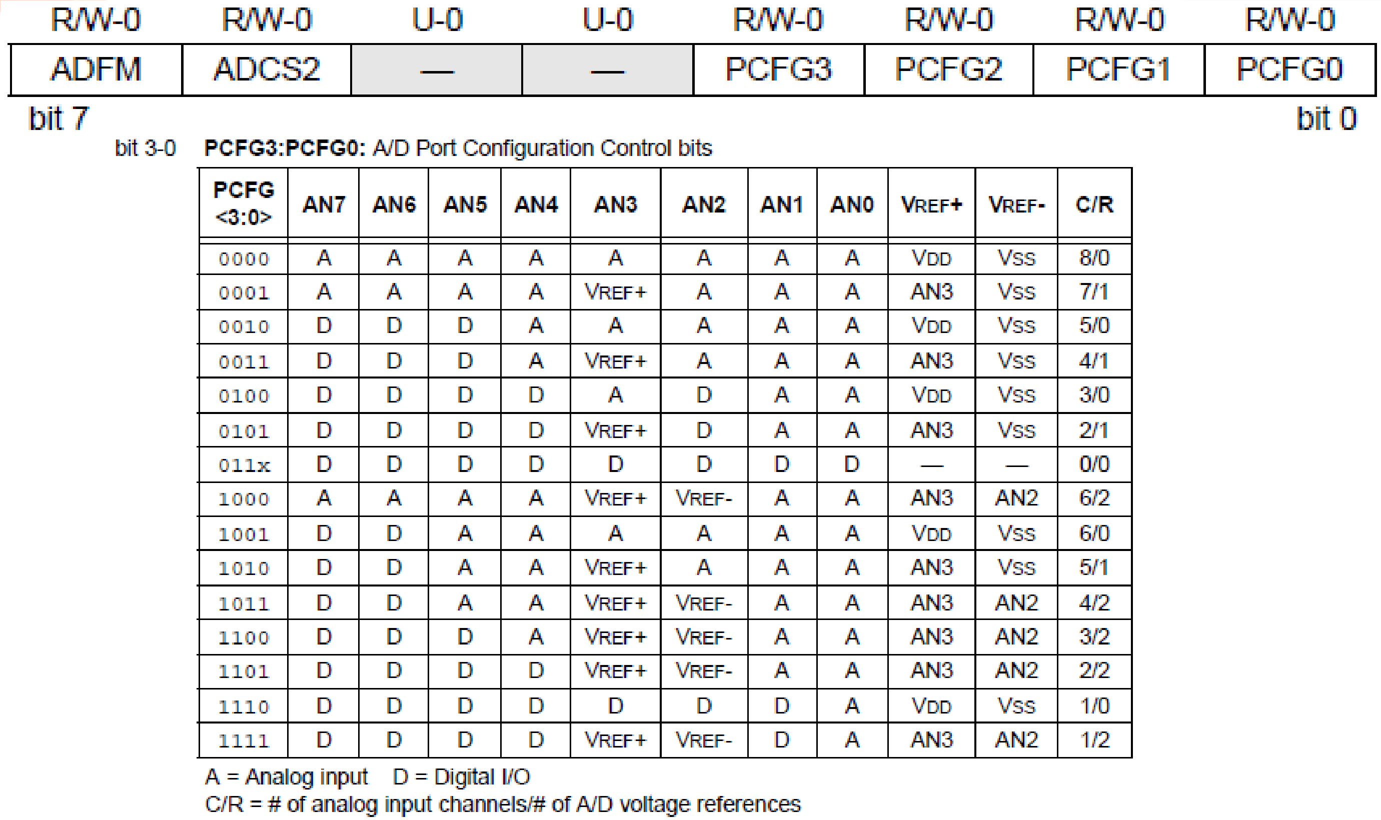
0 = Left justified. Six (6) Least Significant bits of ADRESL are read as '0'.

bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	Fosc/2
0	01	Fosc/8
0	10	Fosc/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	Fosc/4
1	01	Fosc/16
1	10	Fosc/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

bit 5-4 **Unimplemented:** Read as '0'

ADCON1



R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

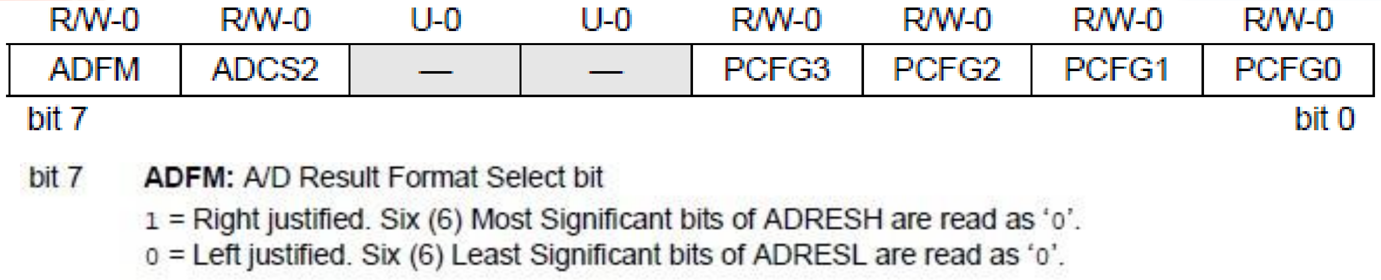
bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O

C/R = # of analog input channels/# of A/D voltage references

ADCON1: A/D Result Format Select



R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

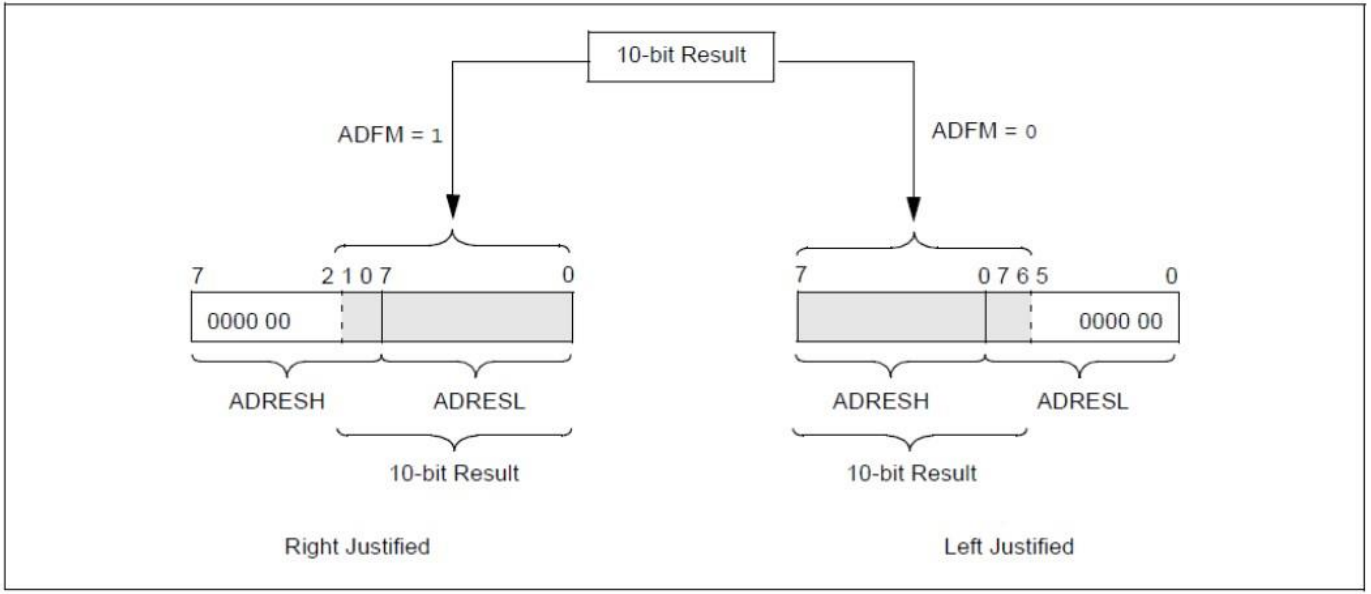
bit 7 **ADFM:** A/D Result Format Select bit

1 = Right justified. Six (6) Most Significant bits of ADRESH are read as '0'.
0 = Left justified. Six (6) Least Significant bits of ADRESL are read as '0'.

The ADFM bit determines how the result of the ADC is justified. Since the ADC on the PIC16F877A has 10-bits of resolution, logically a single register (that has 8 bits) is not enough to contain the 10-bits result. Therefore, two registers are required to store the results. ADRESH and ADRESL (H is the high byte while L is the low byte).

ADCON1: A/D Result Format Select

FIGURE 11-4: A/D RESULT JUSTIFICATION



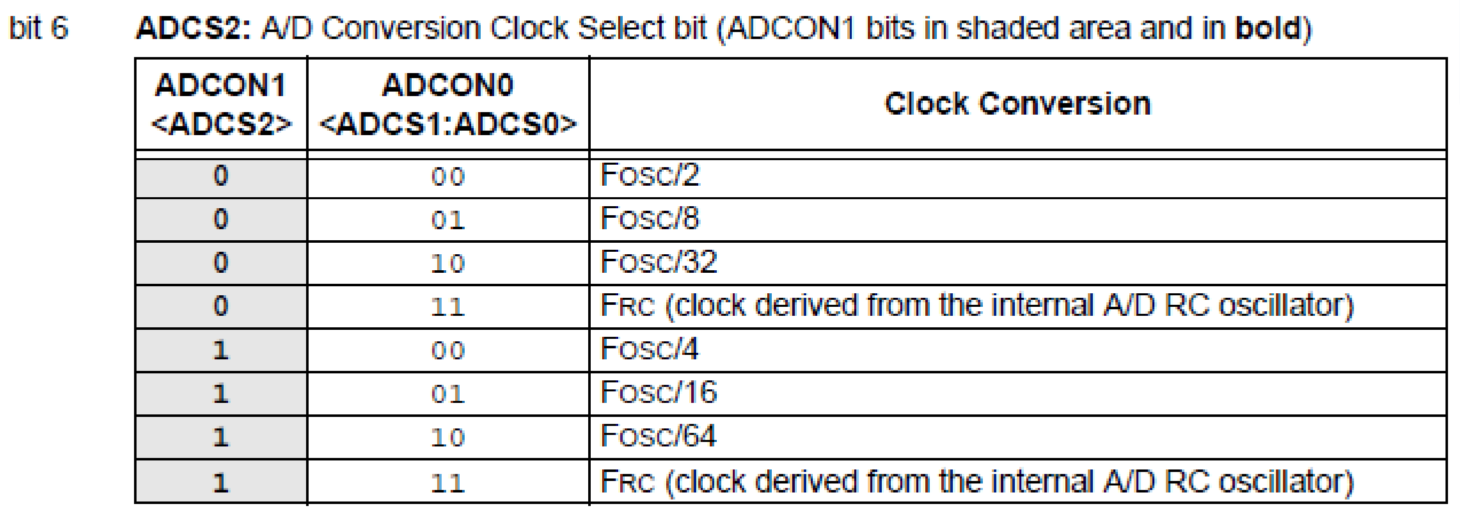
Two registers will allow us to store up to 16 bits, but since there are only 10 bits, we have the flexibility to align it right justified or left justified

ADCON1: A/D Result Format Select

If the application doesn’t need the 10-bit accuracy, 8 bits is more than enough. So we can just take the result in ADRESH and ignore the remaining two least significant bits in ADRESL (we cannot ignore the two highest significant bit because that will cause the result to be inaccurate). That makes it easier to move values to other registers. Yes, the accuracy of the result will be slightly affected but it’s not critical in applications where we don’t need accuracy.

The value of ADCON1 is 0xxx xxxx

ADCON1: Conversion Clock Select

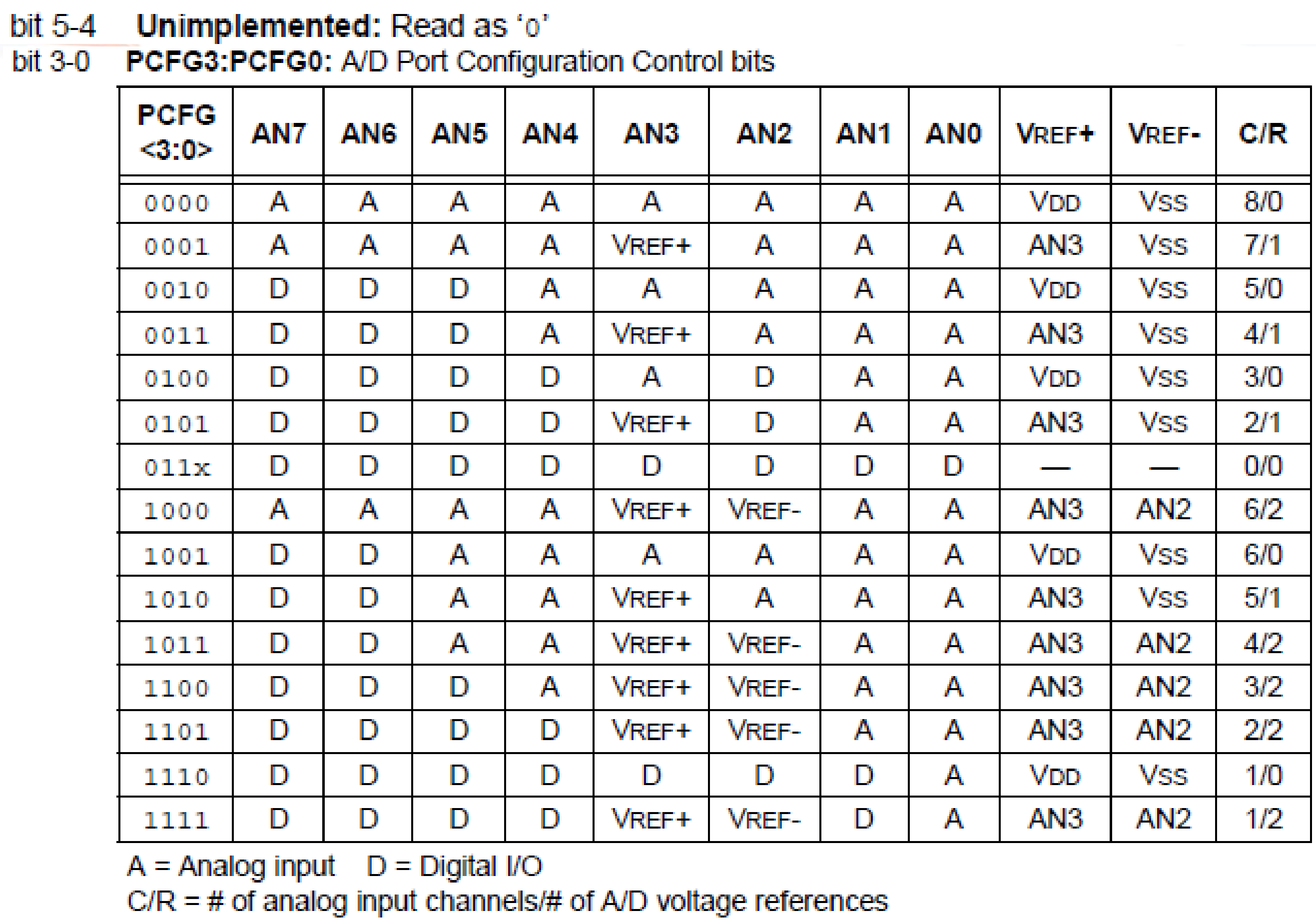


bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

ADCON1 <ADCS2>	**ADCON0 <ADCS1:ADCS0>**	**Clock Conversion**
0	00	FOSC/2
0	01	FOSC/8
0	10	FOSC/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	FOSC/4
1	01	FOSC/16
1	10	FOSC/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

Next is the ADCS2 bit. We agreed that Fosc/16 is adequate, thus we selected it in ADCON0. But for Fosc/16, we need to set the ADCS2 bit in ADCON1 as well. The value of ADCON1 will be 01xx xxxx.

ADCON1: Port Configuration Control



bit 5-4 **Unimplemented:** Read as '0'

bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input   D = Digital I/O

C/R = # of analog input channels/# of A/D voltage references

ADCON1: Port Configuration Control

The most important part of the ADC configuration is to select the mode for each Analog channel. As shown before, we have Analog Channels 0 to 7. All these inputs can either be set to analog or digital. Referring to the table above, if we don’t need any analog inputs and require more digital pins (let’s say for a few LCDs), we can set the PCFG3:0 bits to be 011x. But in the case we do need the Analog inputs, we will set all of them to be in analog mode.

Therefore, the final value for ADCON1 is 0100 0000

ADCON1: Port Configuration Control

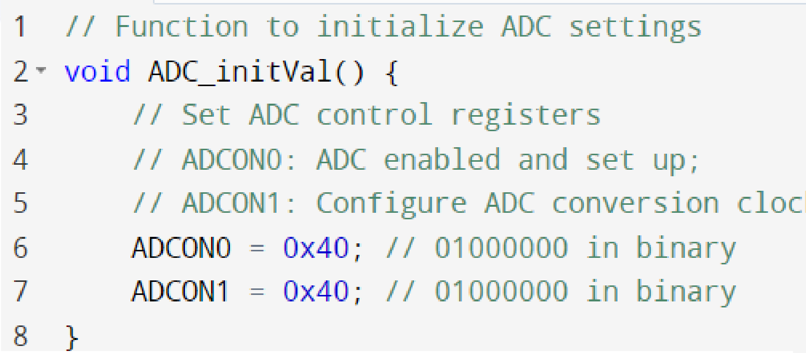
One important thing to note is that we've selected Vdd as the Vref+ and Vss as the Vref-, that means that our conversion range is from 0V to 5V. If we need it to be other than that, we can set a custom Vref value by choosing other configurations of PCFG3:0.

```
void ADC_initVal()
{
 ADCON0=01000000;
 ADCON1=01000000;
}

void main()
{
unsigned int adc_Dout;
char txt[7];
TRISA = 0x01;
ADC_initVal();
UART1_Init(9600);
Delay_ms(100);
```

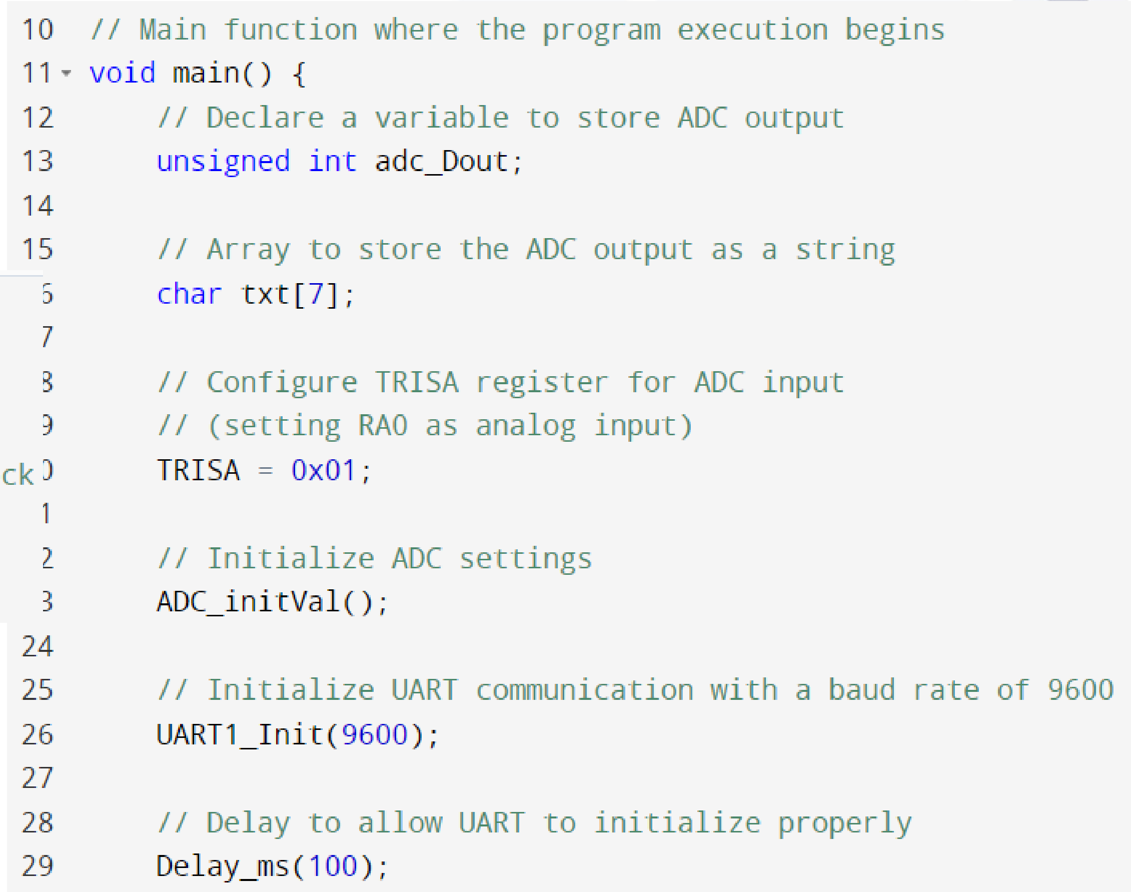
```
while(1)
{
        ADCON0=01000001;
        adc_Dout = ADC_Read(0);
        Delay_ms(200);
        IntToStr(adc_Dout, txt);
        UART1_Write_Text("ADC:");
        UART1_Write_Text(txt);
        UART1_Write(13);
        UART1_Write(10);
}
}
```

```
void ADC_initVal()
{
 ADCON0=01000000;
 ADCON1=01000000;
}

void main()
{
unsigned int adc_Dout;
char txt[7];
TRISA = 0x01;
ADC_initVal();
UART1_Init(9600);
Delay_ms(100);
```

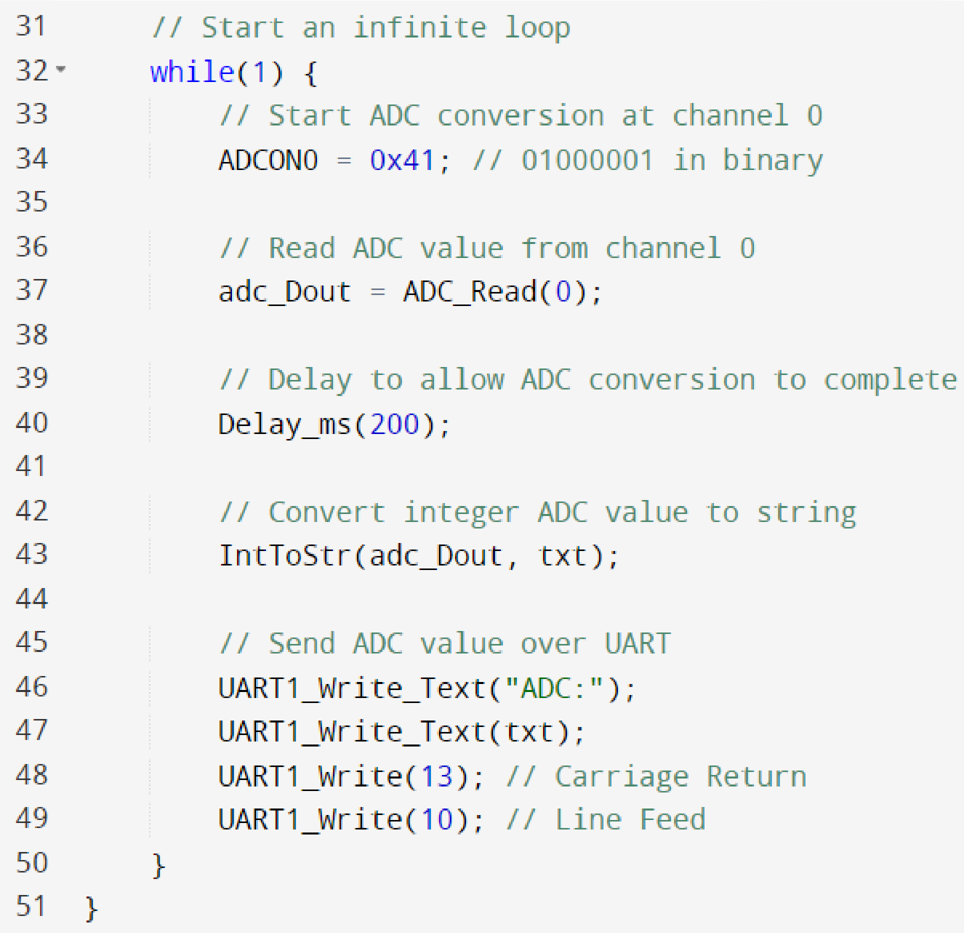
```
while(1)
{
        ADCON0=01000001;
        adc_Dout = ADC_Read(0);
        Delay_ms(200);
        IntToStr(adc_Dout, txt);
        UART1_Write_Text("ADC:");
        UART1_Write_Text(txt);
        UART1_Write(13);
        UART1_Write(10);
}
}
```



```c
// Function to initialize ADC settings
void ADC_initVal() {
    // Set ADC control registers
    // ADCON0: ADC enabled and set up;
    // ADCON1: Configure ADC conversion clock
    ADCON0 = 0x40; // 01000000 in binary
    ADCON1 = 0x40; // 01000000 in binary
}
```



```c
// Main function where the program execution begins
void main() {
    // Declare a variable to store ADC output
    unsigned int adc_Dout;

    // Array to store the ADC output as a string
    char txt[7];

    // Configure TRISA register for ADC input
    // (setting RA0 as analog input)
    TRISA = 0x01;

    // Initialize ADC settings
    ADC_initVal();

    // Initialize UART communication with a baud rate of 9600
    UART1_Init(9600);

    // Delay to allow UART to initialize properly
    Delay_ms(100);
```



```
    // Start an infinite loop
    while(1) {
        // Start ADC conversion at channel 0
        ADCON0 = 0x41; // 01000001 in binary

        // Read ADC value from channel 0
        adc_Dout = ADC_Read(0);

        // Delay to allow ADC conversion to complete
        Delay_ms(200);

        // Convert integer ADC value to string
        IntToStr(adc_Dout, txt);

        // Send ADC value over UART
        UART1_Write_Text("ADC:");
        UART1_Write_Text(txt);
        UART1_Write(13); // Carriage Return
        UART1_Write(10); // Line Feed
    }
}
```

```
void main()
{
    unsigned int adc_Dout;
    char txt[7];
    TRISA = 0x01;
    UART1_Init(9600);
    Delay_ms(100);
    while(1)
    {       adc_Dout = ADC_Read(0);
            Delay_ms(200);
            IntToStr(adc_Dout, txt);
            UART1_Write_Text("ADC:");
            UART1_Write_Text(txt);
            UART1_Write(13);
            UART1_Write(10);
    }
}
```

THANK YOU

Lecture 11

C LANGUAGE

- SERIAL Communication

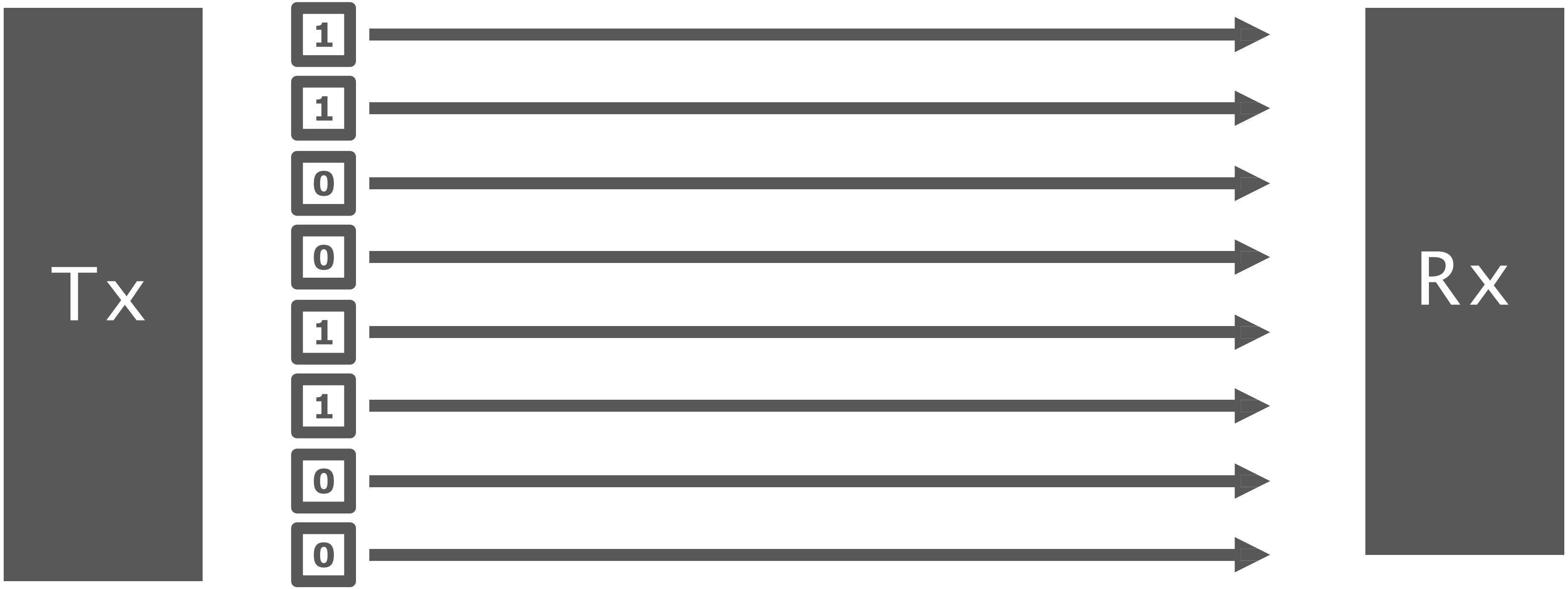
SERIAL COMMUNICATION

SERIAL COMMUNICATION

The UART (Universal Asynchronous Receiver Transmitter) is the universal communication component located within the PIC and can be used as transmitter or as receiver

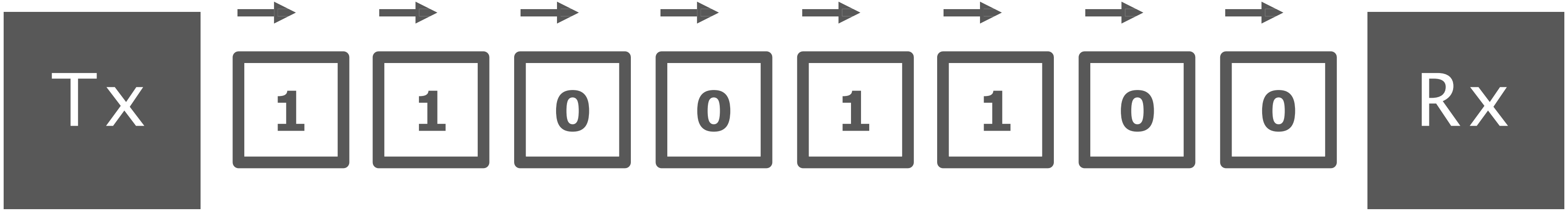
PARALLEL COMMUNICATION

We want to transmit 8 bits: 11001100



SERIAL COMMUNICATION

We want to transmit 8 bits: 11001100

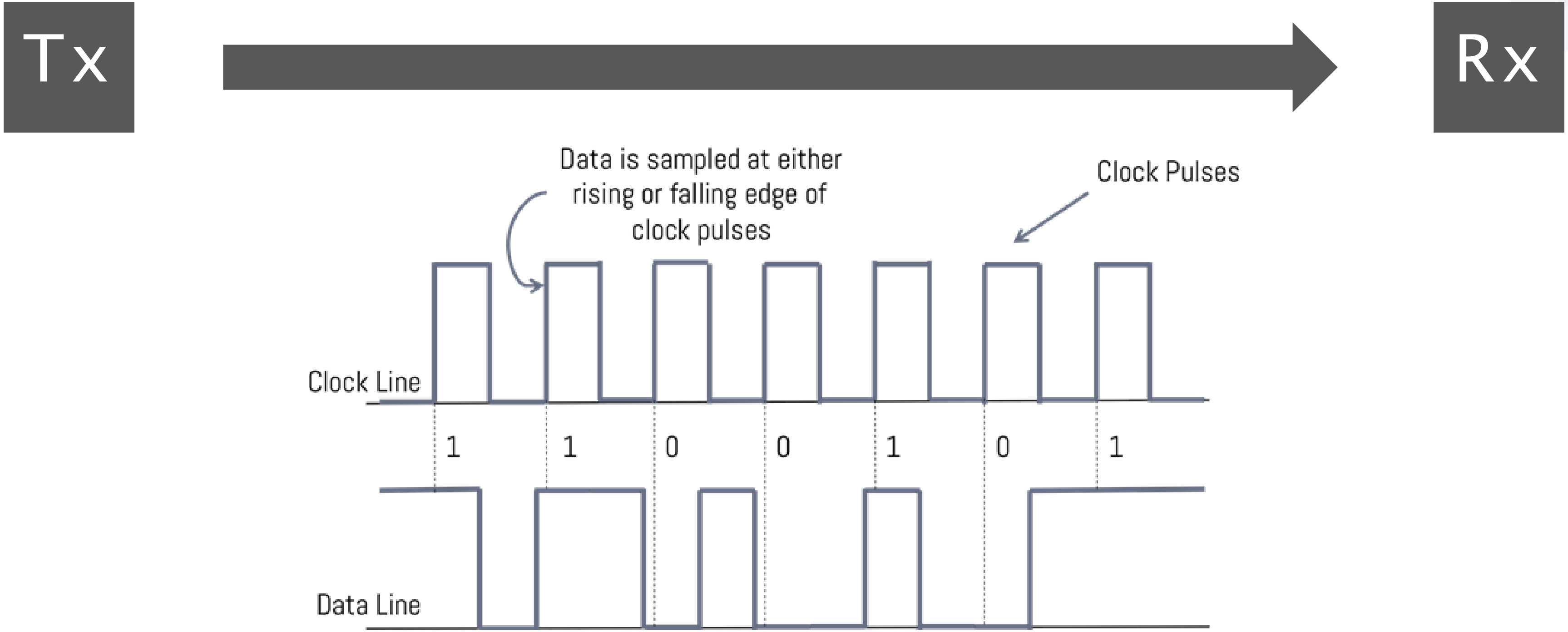


Tx 1 1 0 0 1 1 0 0 Rx

SYNCHRONOUS DATA TRANSFER

The information is sent from the transmitter in sequence, bit after bit, with fixed baud rate, carried by a common clock frequency

SYNCHRONOUS DATA TRANSFER



Tx

Rx

Data is sampled at either rising or falling edge of clock pulses

Clock Pulses

Clock Line

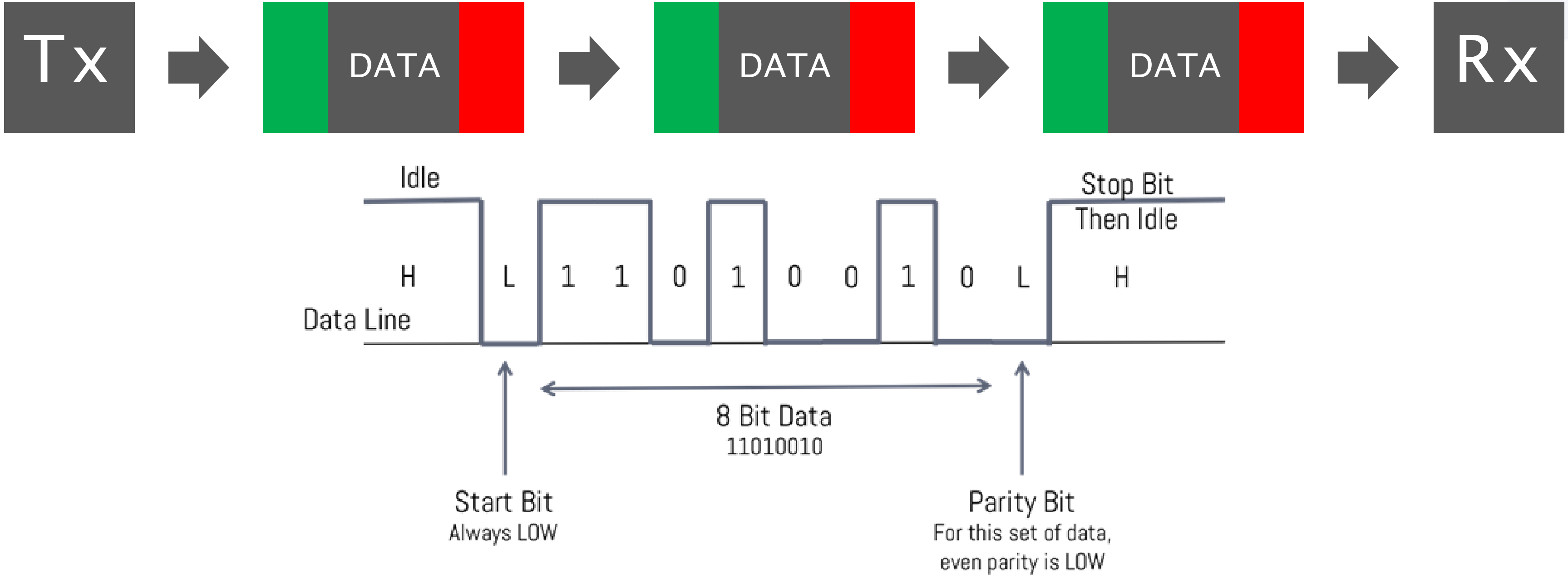
1 1 0 0 1 0 1

Data Line

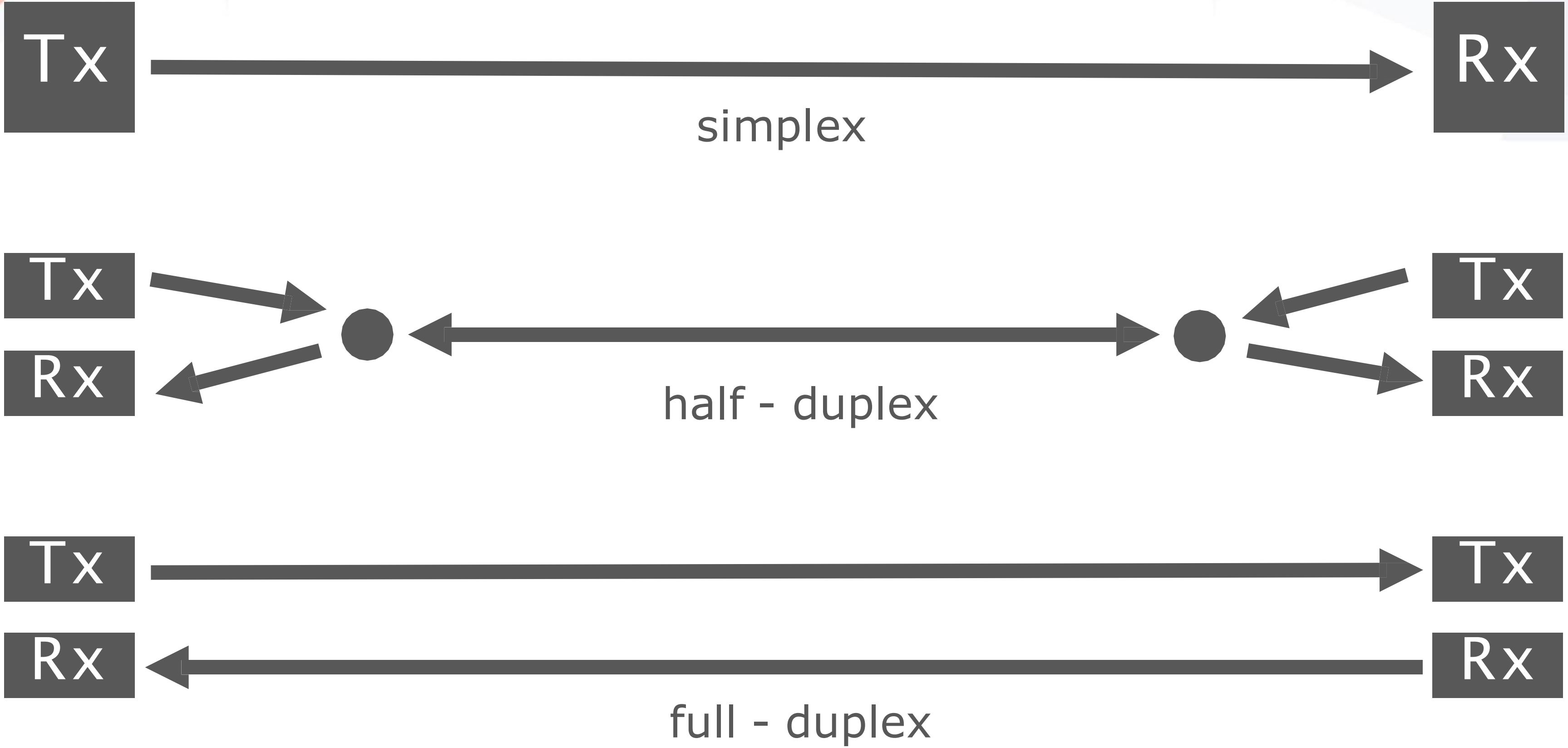
ASYNCHRONOUS DATA TRANSFER

The information is divided into frames, and each frame has a “Start” bit and a “Stop” bit. The “Start” bit marks the beginning of a new frame, the “Stop” bit marks the end. Frames of information are not necessarily transmitted at equal time space, since they are independent of the clock.

ASYNCHRONOUS DATA TRANSFER



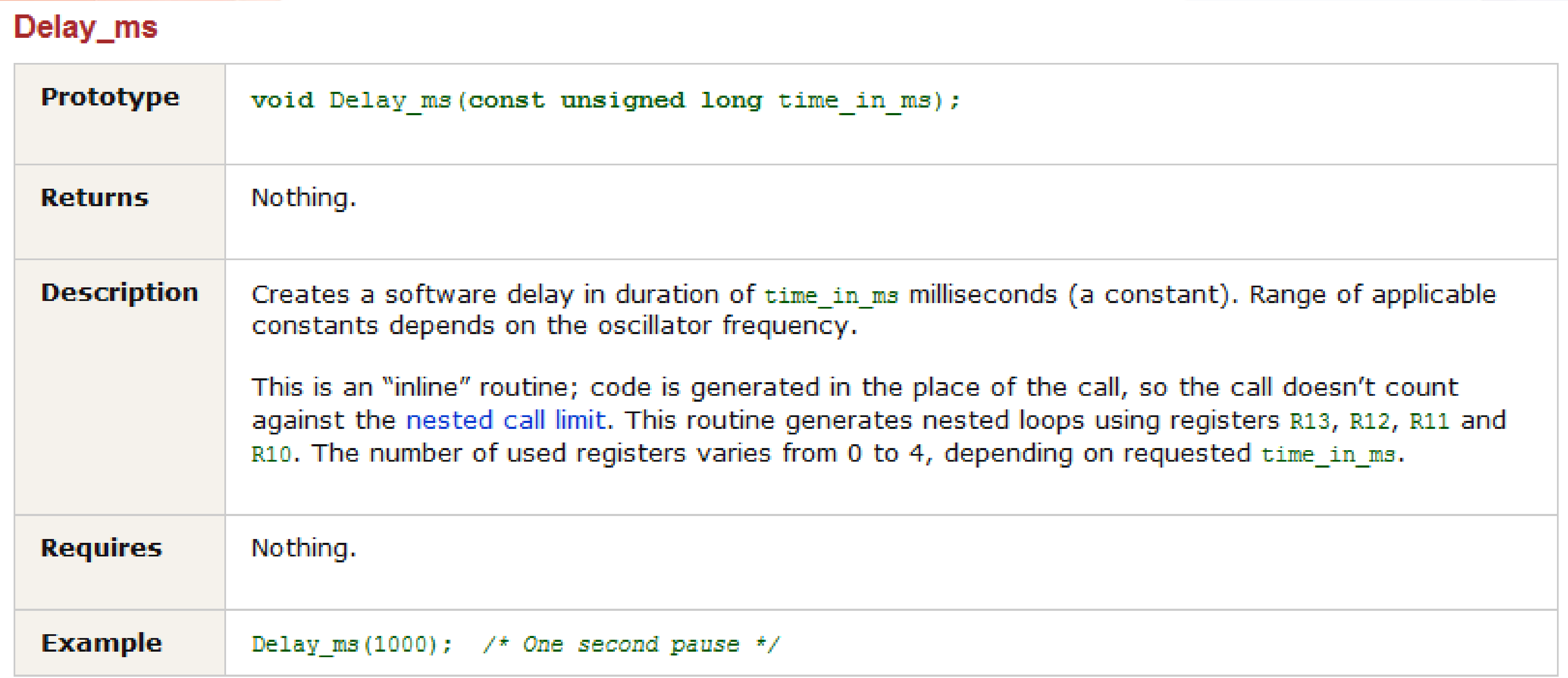
DATA TRANSFER OPERATION MODES



FUNCTIONS

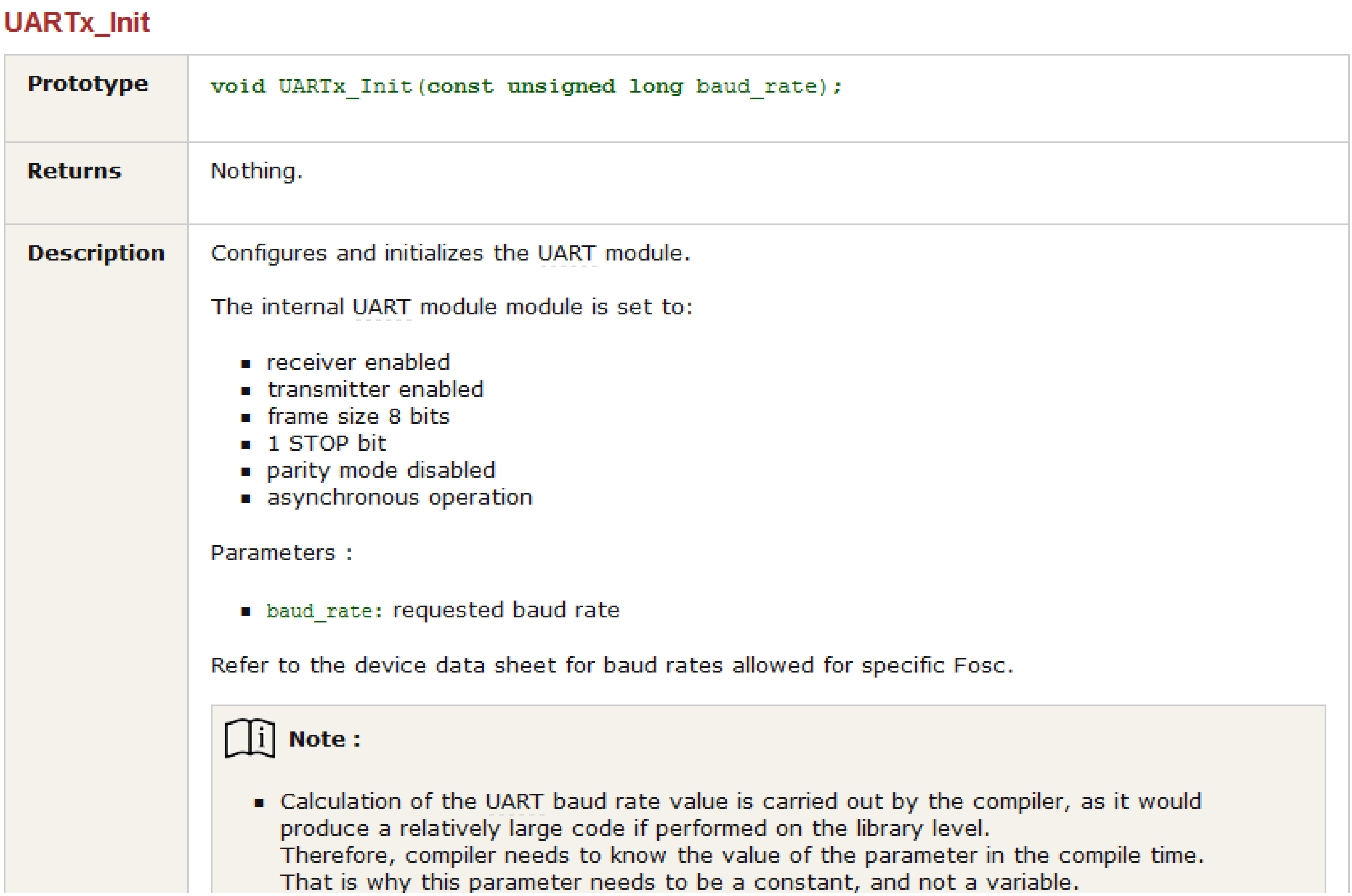
```
type FuncName([arg1, arg2,...])
{
      // function code – some logic
}
```

1. When you see a function first read the description – do you need it?
2. Then you need to check what the function returns. (void doesn‘t return anything)
3. What is the name of the function?
4. Does it have any arguments? If it does make sure to provide literals/constants/variables of the same type.



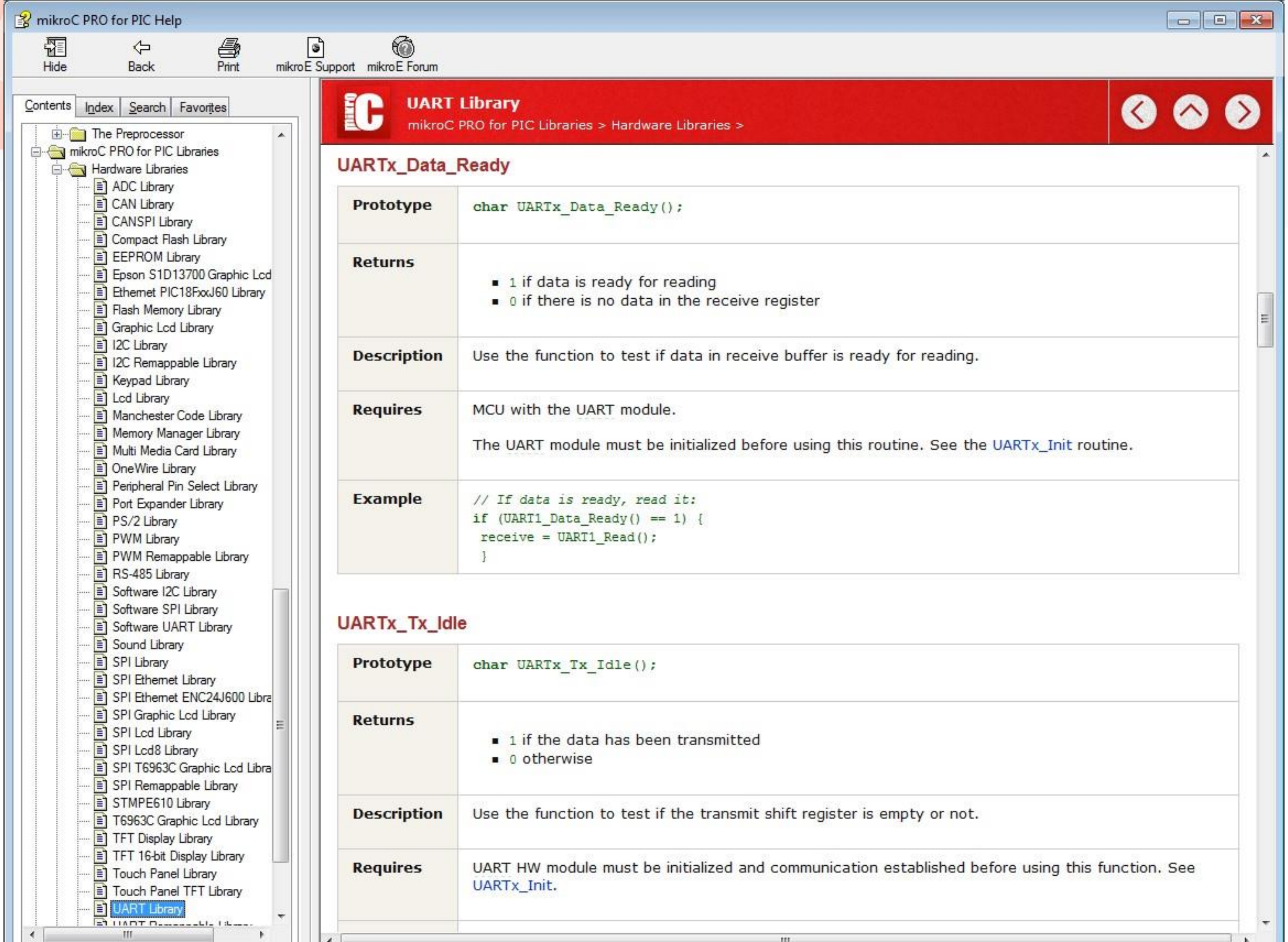
Delay_ms

Prototype	`void Delay_ms(const unsigned long time_in_ms);`
Returns	Nothing.
Description	Creates a software delay in duration of `time_in_ms` milliseconds (a constant). Range of applicable constants depends on the oscillator frequency. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit. This routine generates nested loops using registers `R13`, `R12`, `R11` and `R10`. The number of used registers varies from 0 to 4, depending on requested `time_in_ms`.
Requires	Nothing.
Example	`Delay_ms(1000);  /* One second pause */`



UARTx_Init

Prototype	`void UARTx_Init(const unsigned long baud_rate);`
Returns	Nothing.
Description	Configures and initializes the UART module. The internal UART module module is set to: ▪ receiver enabled ▪ transmitter enabled ▪ frame size 8 bits ▪ 1 STOP bit ▪ parity mode disabled ▪ asynchronous operation Parameters : ▪ `baud_rate:` requested baud rate Refer to the device data sheet for baud rates allowed for specific Fosc. **Note :** ▪ Calculation of the UART baud rate value is carried out by the compiler, as it would produce a relatively large code if performed on the library level. Therefore, compiler needs to know the value of the parameter in the compile time. That is why this parameter needs to be a constant, and not a variable.



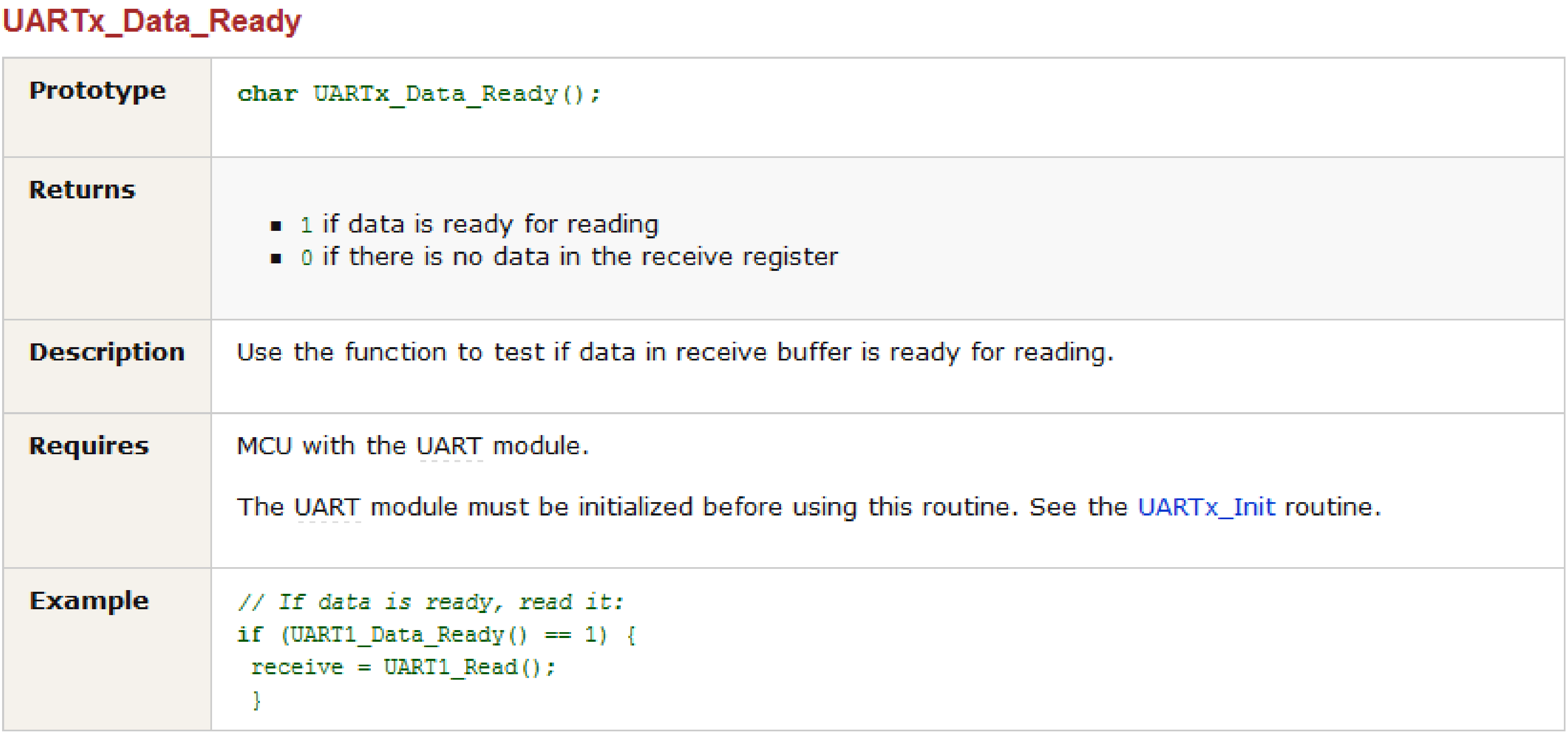
UARTx_Data_Ready

Prototype	`char UARTx_Data_Ready();`
Returns	▪ 1 if data is ready for reading ▪ 0 if there is no data in the receive register
Description	Use the function to test if data in receive buffer is ready for reading.
Requires	MCU with the UART module. The UART module must be initialized before using this routine. See the UARTx_Init routine.

Example	`// If data is ready, read it:`
	`if (UART1_Data_Ready() == 1) {`
	` receive = UART1_Read();`
	` }`

UARTx_Tx_Idle

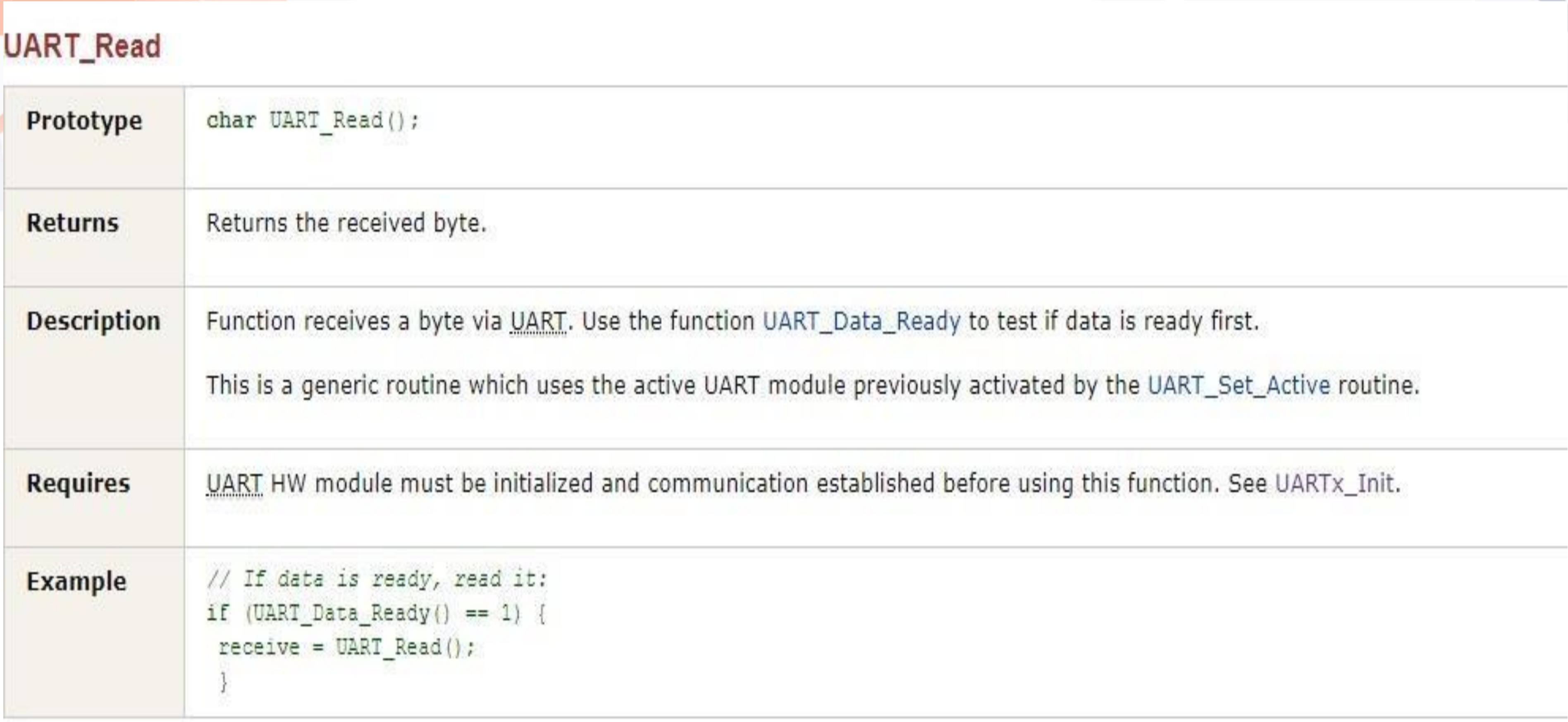
Prototype	`char UARTx_Tx_Idle();`
Returns	▪ 1 if the data has been transmitted ▪ 0 otherwise
Description	Use the function to test if the transmit shift register is empty or not.
Requires	UART HW module must be initialized and communication established before using this function. See UARTx_Init.



UARTx_Data_Ready

Prototype	`char UARTx_Data_Ready();`
Returns	▪ `1` if data is ready for reading ▪ `0` if there is no data in the receive register
Description	Use the function to test if data in receive buffer is ready for reading.
Requires	MCU with the UART module. The UART module must be initialized before using this routine. See the UARTx_Init routine.

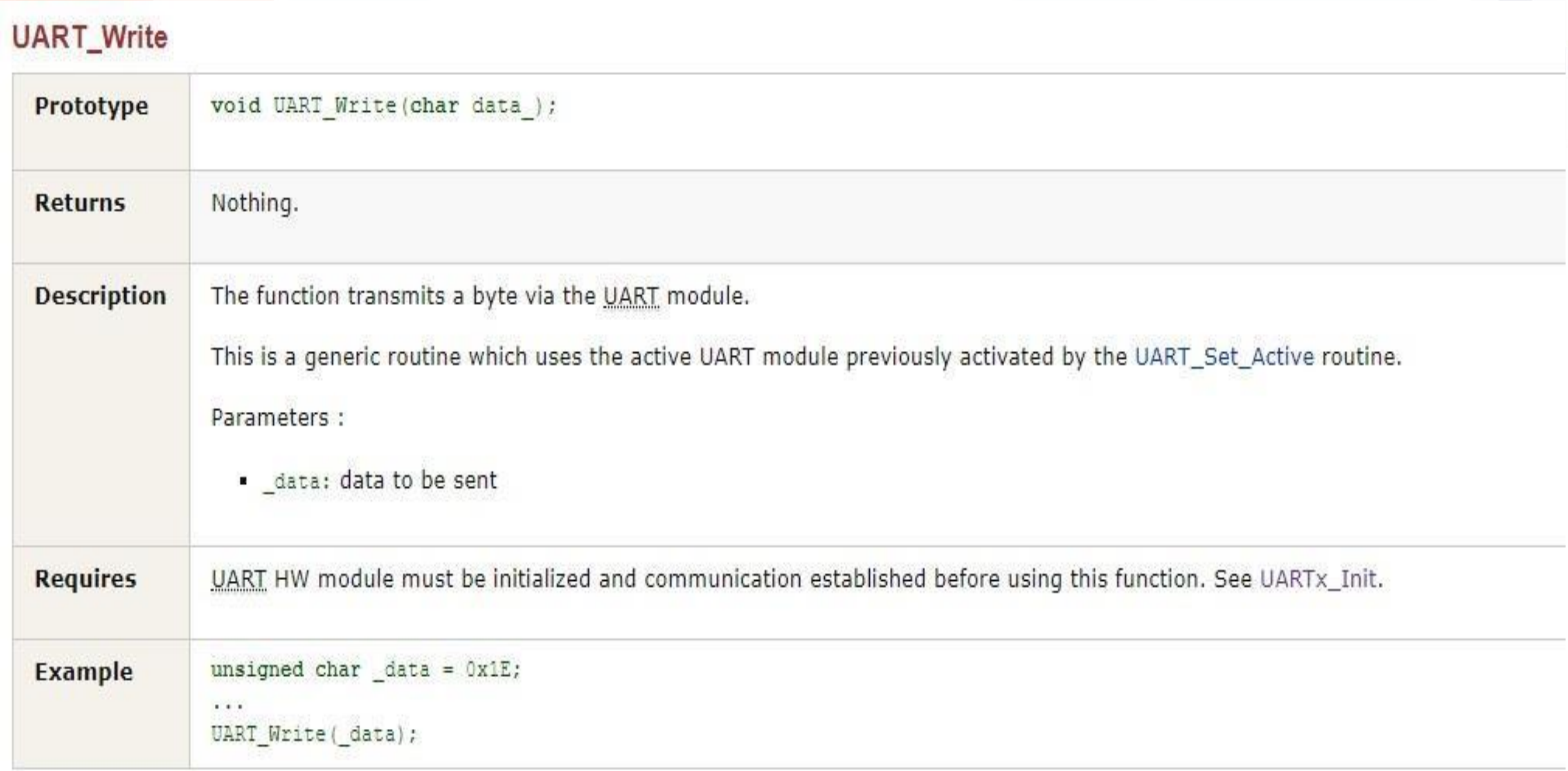
Example	`// If data is ready, read it:`
	`if (UART1_Data_Ready() == 1) {`
	` receive = UART1_Read();`
	` }`



UART_Read

Prototype	`char UART_Read();`
Returns	Returns the received byte.
Description	Function receives a byte via UART. Use the function UART_Data_Ready to test if data is ready first. This is a generic routine which uses the active UART module previously activated by the UART_Set_Active routine.
Requires	UART HW module must be initialized and communication established before using this function. See UARTx_Init.

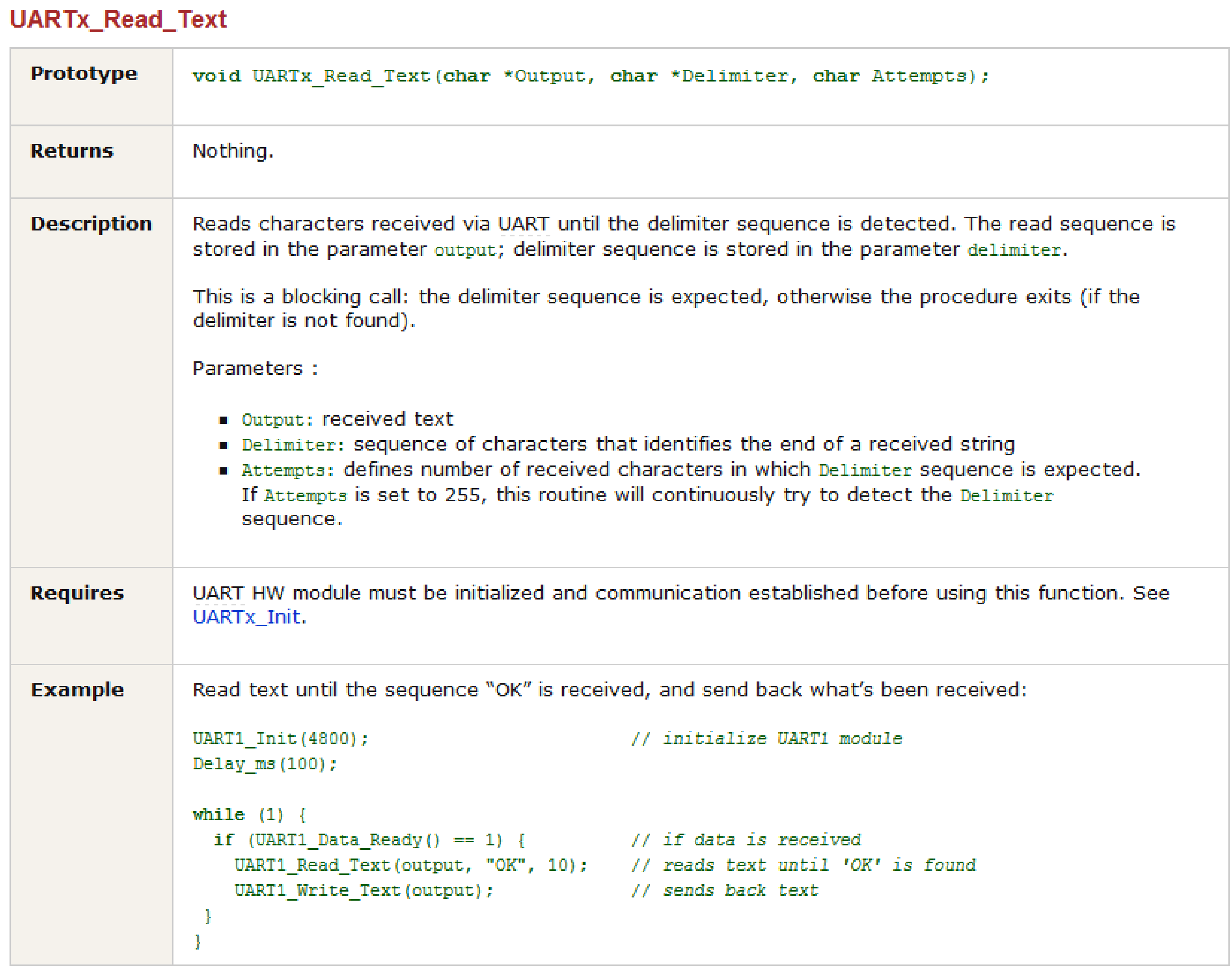
Example	`// If data is ready, read it:`
	`if (UART_Data_Ready() == 1) {`
	` receive = UART_Read();`
	` }`



UART_Write

Prototype	`void UART_Write(char data_);`
Returns	Nothing.
Description	The function transmits a byte via the UART module. This is a generic routine which uses the active UART module previously activated by the UART_Set_Active routine. Parameters : ▪ `_data`: data to be sent
Requires	UART HW module must be initialized and communication established before using this function. See UARTx_Init.

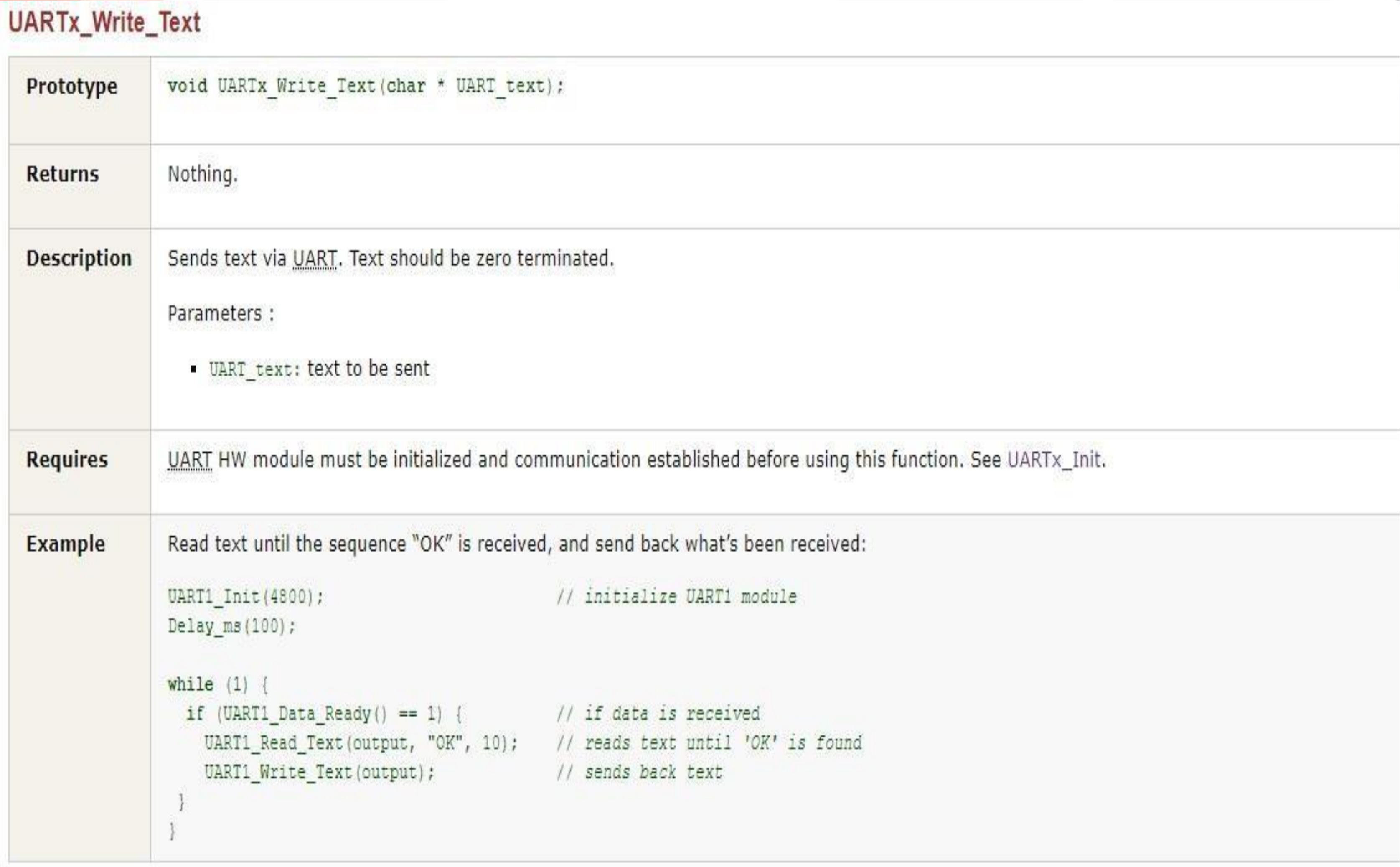
Example	`unsigned char _data = 0x1E;`
	`...`
	`UART_Write(_data);`



UARTx_Read_Text

<table>
<tr><td><b>Prototype</b></td><td><code>void UARTx_Read_Text(char *Output, char *Delimiter, char Attempts);</code></td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>Reads characters received via UART until the delimiter sequence is detected. The read sequence is stored in the parameter <code>output</code>; delimiter sequence is stored in the parameter <code>delimiter</code>.
This is a blocking call: the delimiter sequence is expected, otherwise the procedure exits (if the delimiter is not found).
Parameters :<ul><li><code>Output:</code> received text</li><li><code>Delimiter:</code> sequence of characters that identifies the end of a received string</li><li><code>Attempts:</code> defines number of received characters in which <code>Delimiter</code> sequence is expected. If <code>Attempts</code> is set to 255, this routine will continuously try to detect the <code>Delimiter</code> sequence.</li></ul></td></tr>
<tr><td><b>Requires</b></td><td>UART HW module must be initialized and communication established before using this function. See UARTx_Init.</td></tr>
<tr><td><b>Example</b></td><td>Read text until the sequence “OK” is received, and send back what’s been received:<pre>
UART1_Init(4800);   // initialize UART1 module
Delay_ms(100);

while (1) {
  if (UART1_Data_Ready() == 1) {   // if data is received
    UART1_Read_Text(output, "OK", 10);   // reads text until 'OK' is found
    UART1_Write_Text(output);   // sends back text
 }
}
</pre></td></tr>
</table>



UARTx_Write_Text

Prototype

```
void UARTx_Write_Text(char * UART_text);
```

Returns

Nothing.

Description

Sends text via UART. Text should be zero terminated.

Parameters :

- `UART_text`: text to be sent

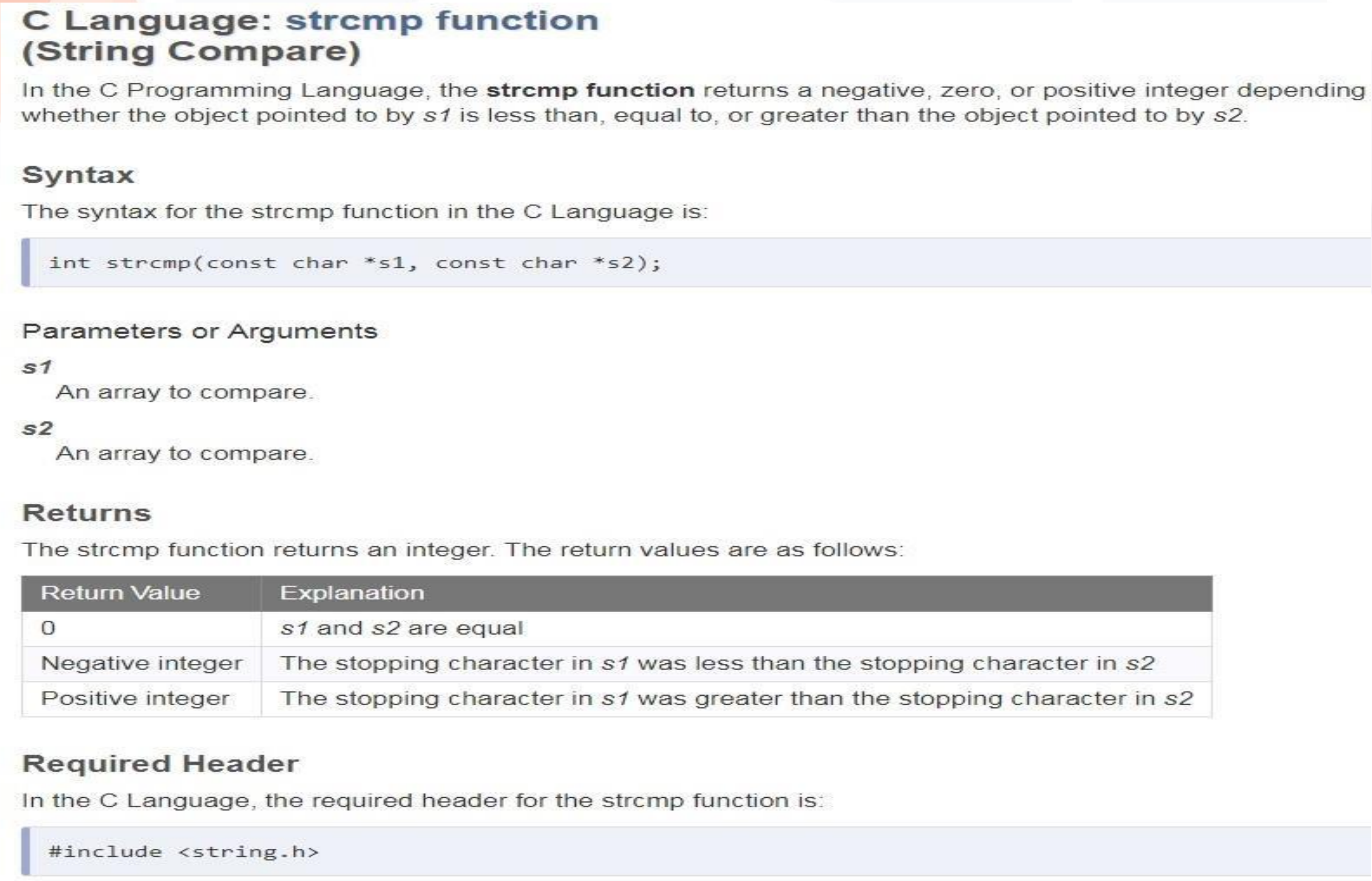
Requires

UART HW module must be initialized and communication established before using this function. See UARTx_Init.

Example

Read text until the sequence "OK" is received, and send back what's been received:

```
UART1_Init(4800);                         // initialize UART1 module
Delay_ms(100);

while (1) {
  if (UART1_Data_Ready() == 1) {          // if data is received
    UART1_Read_Text(output, "OK", 10);    // reads text until 'OK' is found
    UART1_Write_Text(output);             // sends back text
 }
}
```



C Language: strcmp function (String Compare)

In the C Programming Language, the **strcmp function** returns a negative, zero, or positive integer depending whether the object pointed to by *s1* is less than, equal to, or greater than the object pointed to by *s2*.

Syntax

The syntax for the strcmp function in the C Language is:

```
int strcmp(const char *s1, const char *s2);
```

Parameters or Arguments

s1

An array to compare.

s2

An array to compare.

Returns

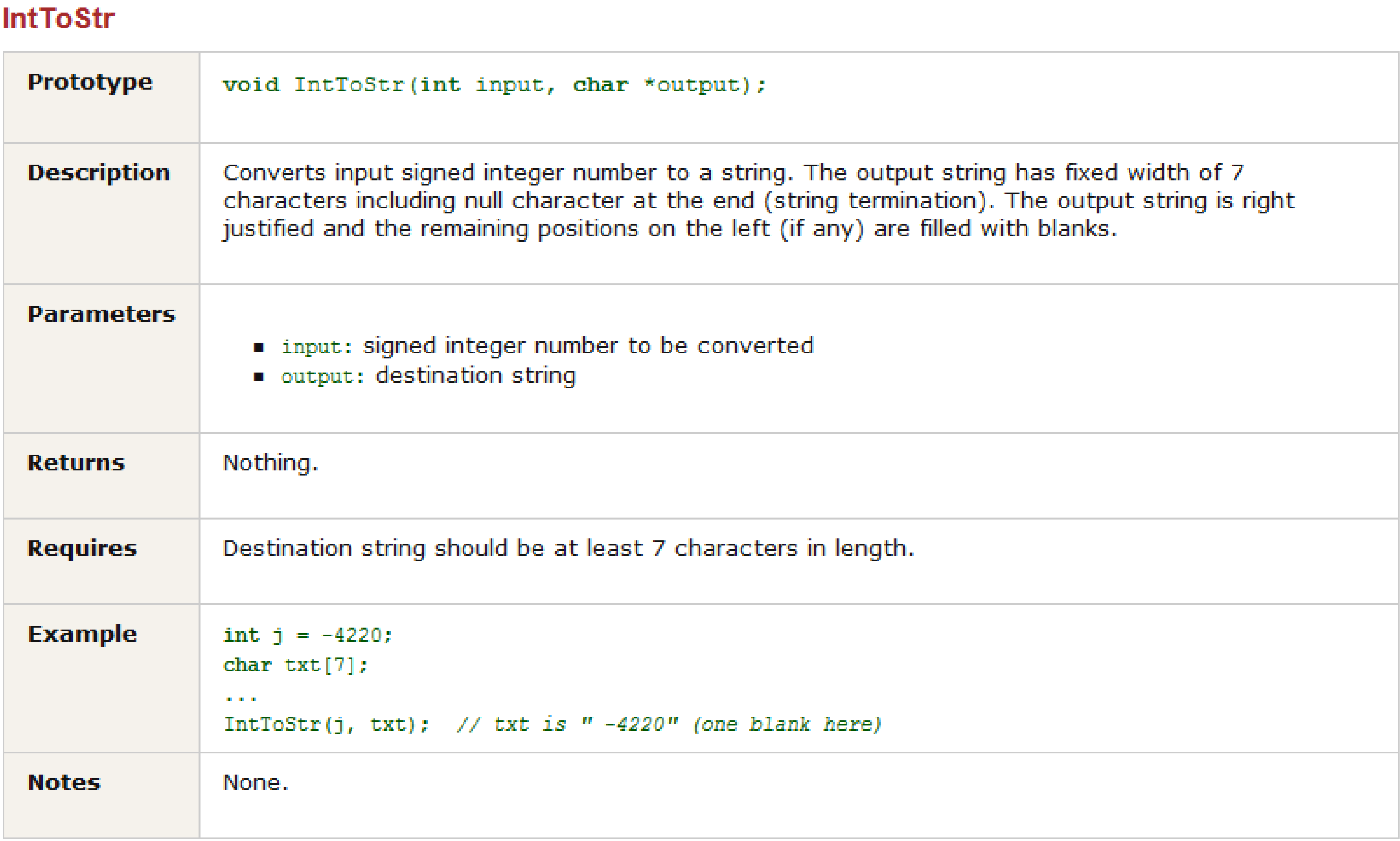
The strcmp function returns an integer. The return values are as follows:

Return Value	Explanation
0	*s1* and *s2* are equal
Negative integer	The stopping character in *s1* was less than the stopping character in *s2*
Positive integer	The stopping character in *s1* was greater than the stopping character in *s2*

Required Header

In the C Language, the required header for the strcmp function is:

```
#include <string.h>
```

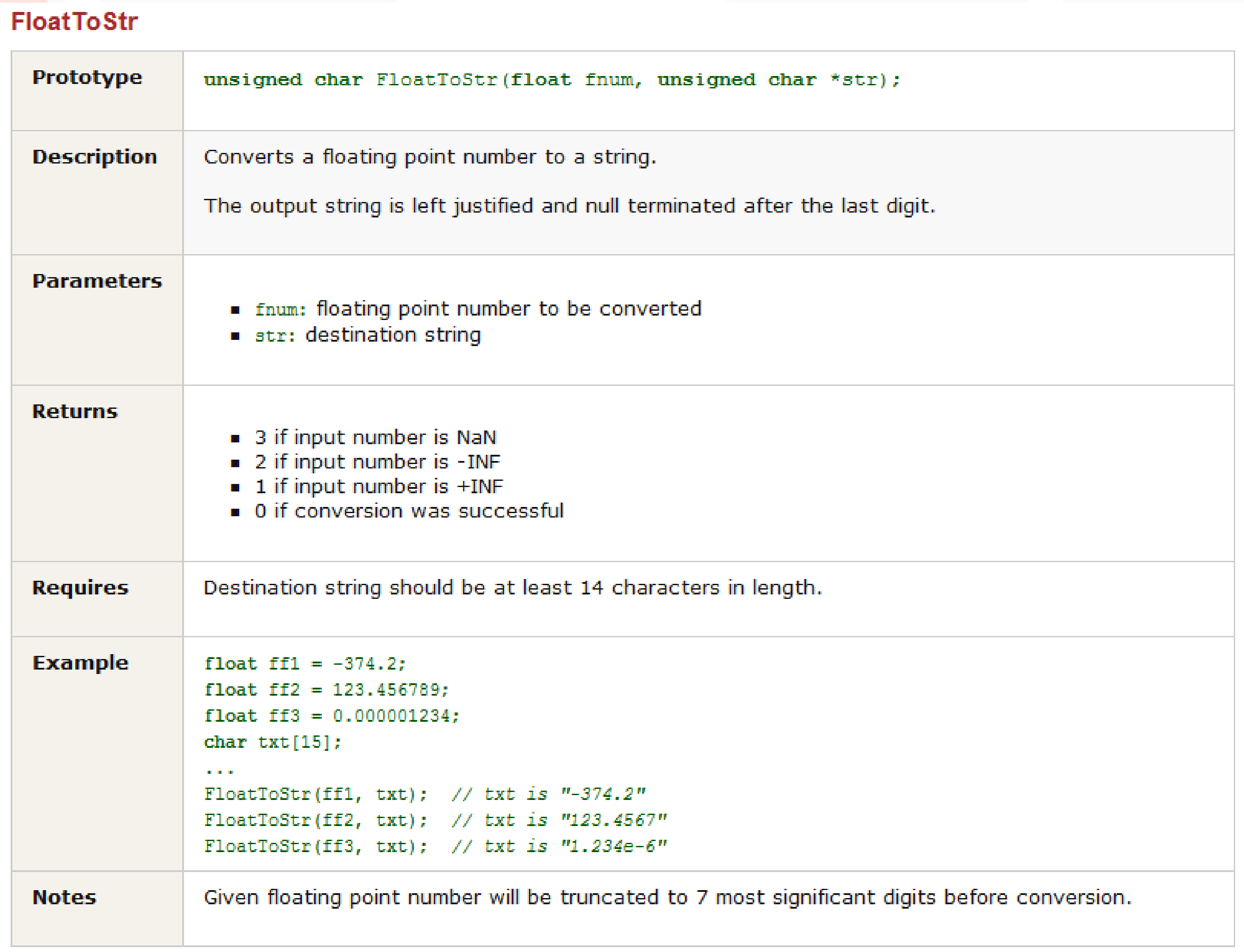


IntToStr

Prototype	`void IntToStr(int input, char *output);`
Description	Converts input signed integer number to a string. The output string has fixed width of 7 characters including null character at the end (string termination). The output string is right justified and the remaining positions on the left (if any) are filled with blanks.
Parameters	- `input`: signed integer number to be converted - `output`: destination string
Returns	Nothing.
Requires	Destination string should be at least 7 characters in length.

Example	`int j = -4220;`
	`char txt[7];`
	`...`
	`IntToStr(j, txt);  // txt is " -4220" (one blank here)`

Notes	None.



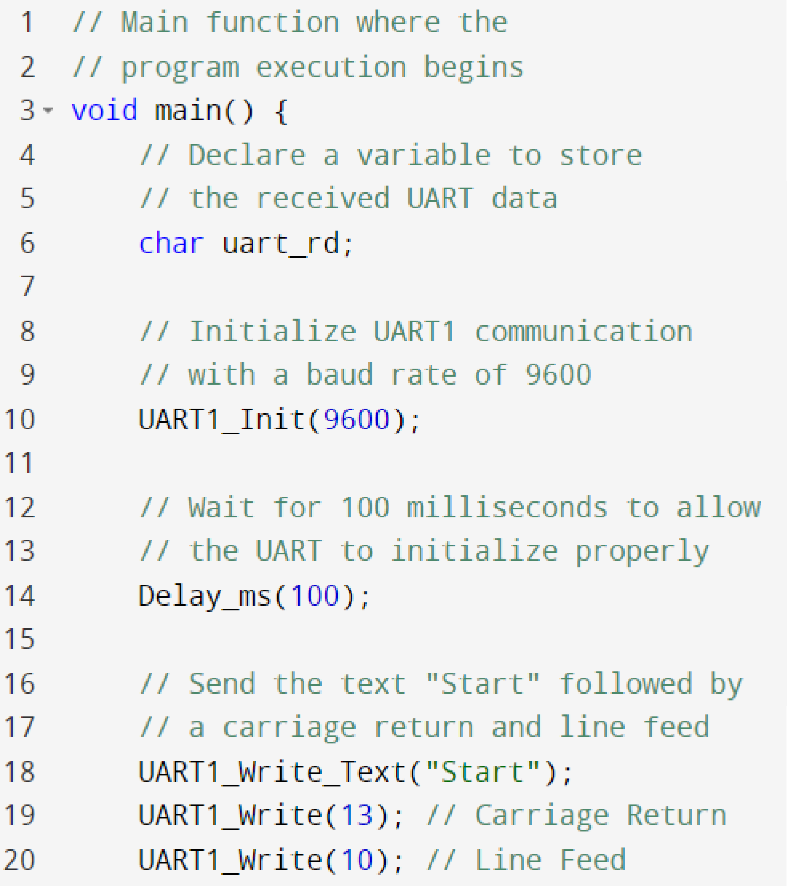
FloatToStr

Prototype	`unsigned char FloatToStr(float fnum, unsigned char *str);`
Description	Converts a floating point number to a string. The output string is left justified and null terminated after the last digit.
Parameters	▪ `fnum:` floating point number to be converted ▪ `str:` destination string
Returns	▪ 3 if input number is NaN ▪ 2 if input number is -INF ▪ 1 if input number is +INF ▪ 0 if conversion was successful
Requires	Destination string should be at least 14 characters in length.

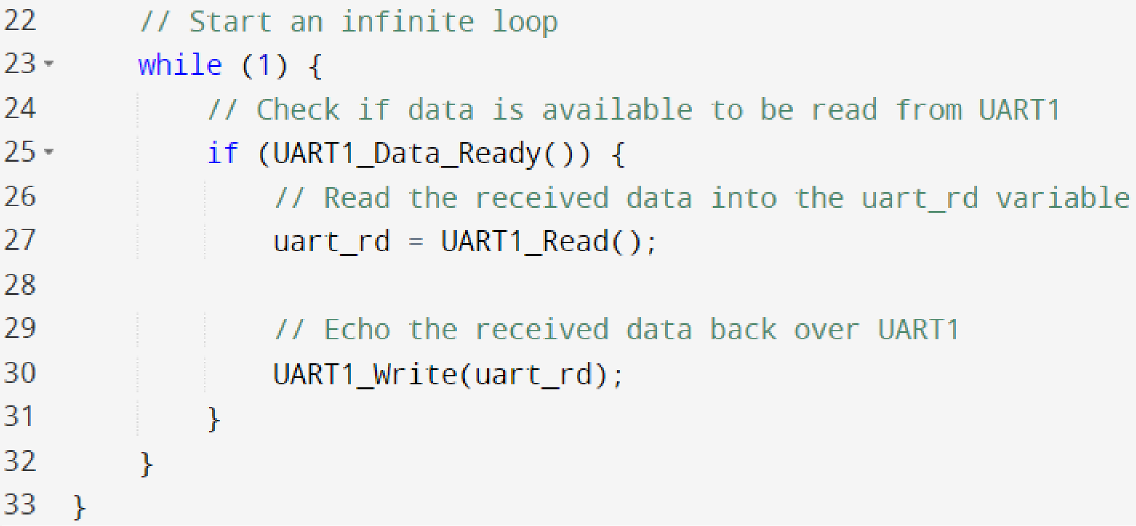
Example	`float ff1 = -374.2;`
	`float ff2 = 123.456789;`
	`float ff3 = 0.000001234;`
	`char txt[15];`
	`...`
	`FloatToStr(ff1, txt);  // txt is "-374.2"`
	`FloatToStr(ff2, txt);  // txt is "123.4567"`
	`FloatToStr(ff3, txt);  // txt is "1.234e-6"`

Notes	Given floating point number will be truncated to 7 most significant digits before conversion.

```
void main()
{
  char uart_rd;
  UART1_Init(9600);                  // Initialize UART module at 9600 bps
  Delay_ms(100);                     // Wait for UART module to stabilize
  UART1_Write_Text("Start");
  UART1_Write(13);
  UART1_Write(10);
  while (1)                          // Endless loop
  {    If (UART1_Data_Ready())       // If data is received,
       {
          uart_rd = UART1_Read();        // read the received data,
          UART1_Write(uart_rd);          // and send data via UART
       }
   }
}
```



```
// Main function where the
// program execution begins
void main() {
    // Declare a variable to store
    // the received UART data
    char uart_rd;

    // Initialize UART1 communication
    // with a baud rate of 9600
    UART1_Init(9600);

    // Wait for 100 milliseconds to allow
    // the UART to initialize properly
    Delay_ms(100);

    // Send the text "Start" followed by
    // a carriage return and line feed
    UART1_Write_Text("Start");
    UART1_Write(13); // Carriage Return
    UART1_Write(10); // Line Feed
```



```
    // Start an infinite loop
    while (1) {
        // Check if data is available to be read from UART1
        if (UART1_Data_Ready()) {
            // Read the received data into the uart_rd variable
            uart_rd = UART1_Read();

            // Echo the received data back over UART1
            UART1_Write(uart_rd);
        }
    }
}
```

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]

THANK YOU

CLASS 10

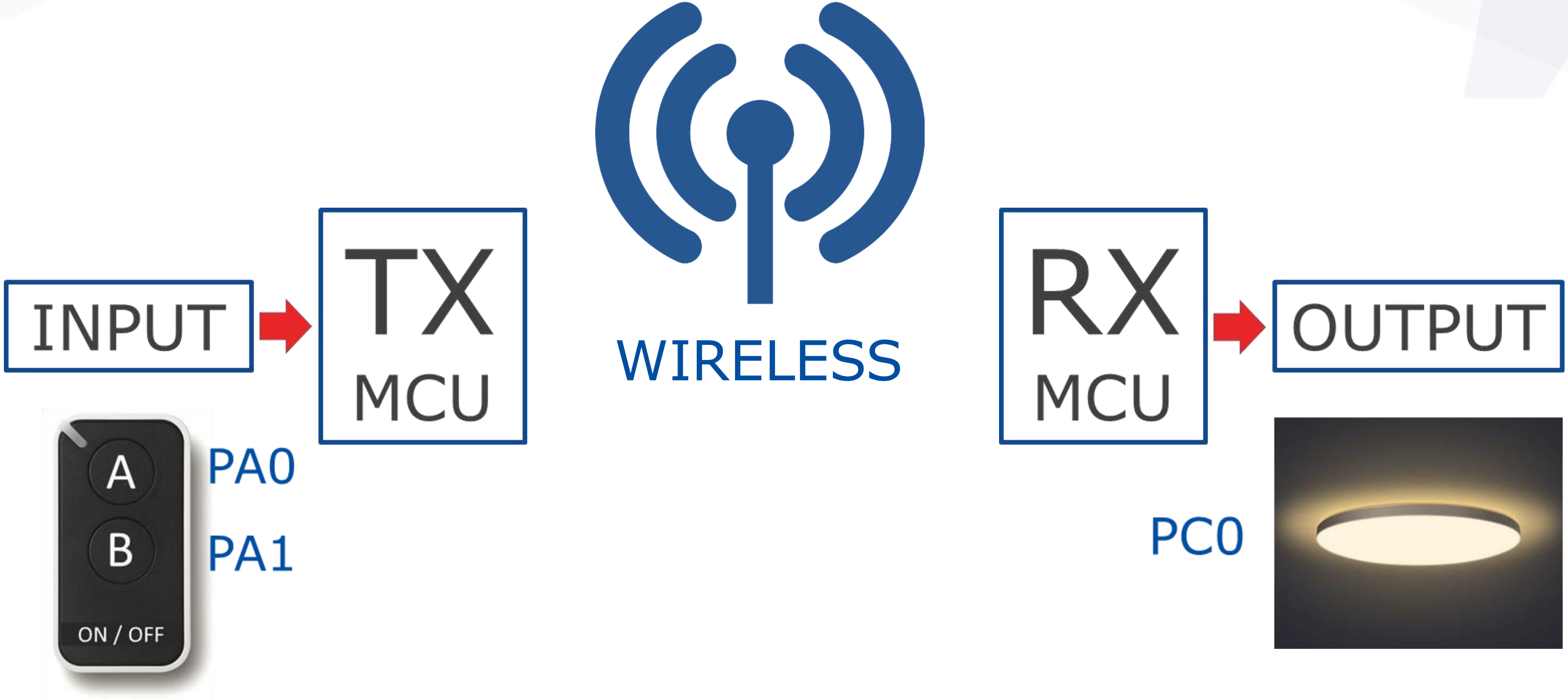
C LANGUAGE

- REVIEW
- INTERRUPTS
- TIMERS AND COUNTERS

REVIEW

SERIAL COMMUNICATION

REMOTE CONTROL: ON/OFF LED



INPUT

TX MCU

WIRELESS

RX MCU

OUTPUT

PA0

PA1

PC0

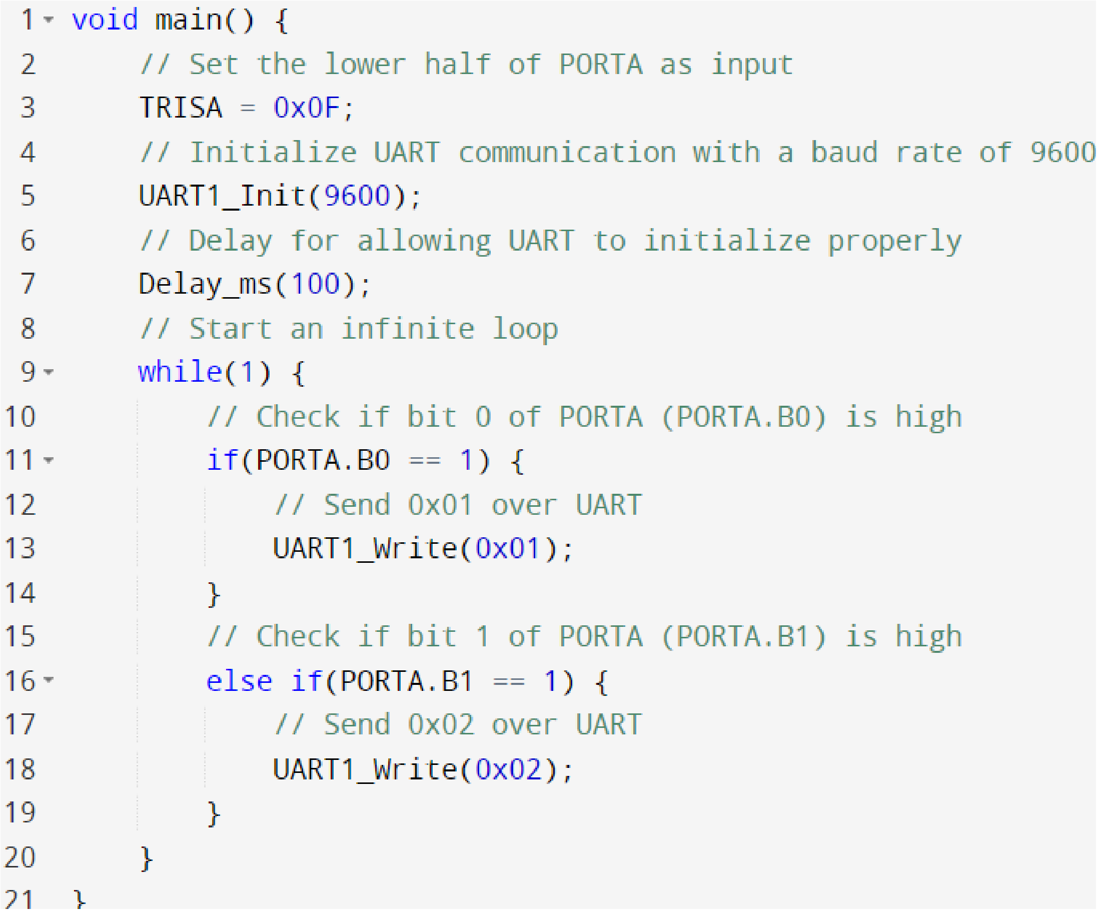
```
void main() {
  TRISA=0x0F;
  UART1_Init(9600);
  Delay_ms(100);
  while(1) {
    if(PORTA.B0==1) {
      UART1_Write(0x01); }
    else if(PORTA.B1==1) {
      UART1_Write(0x02); }
  }
}
```

TX

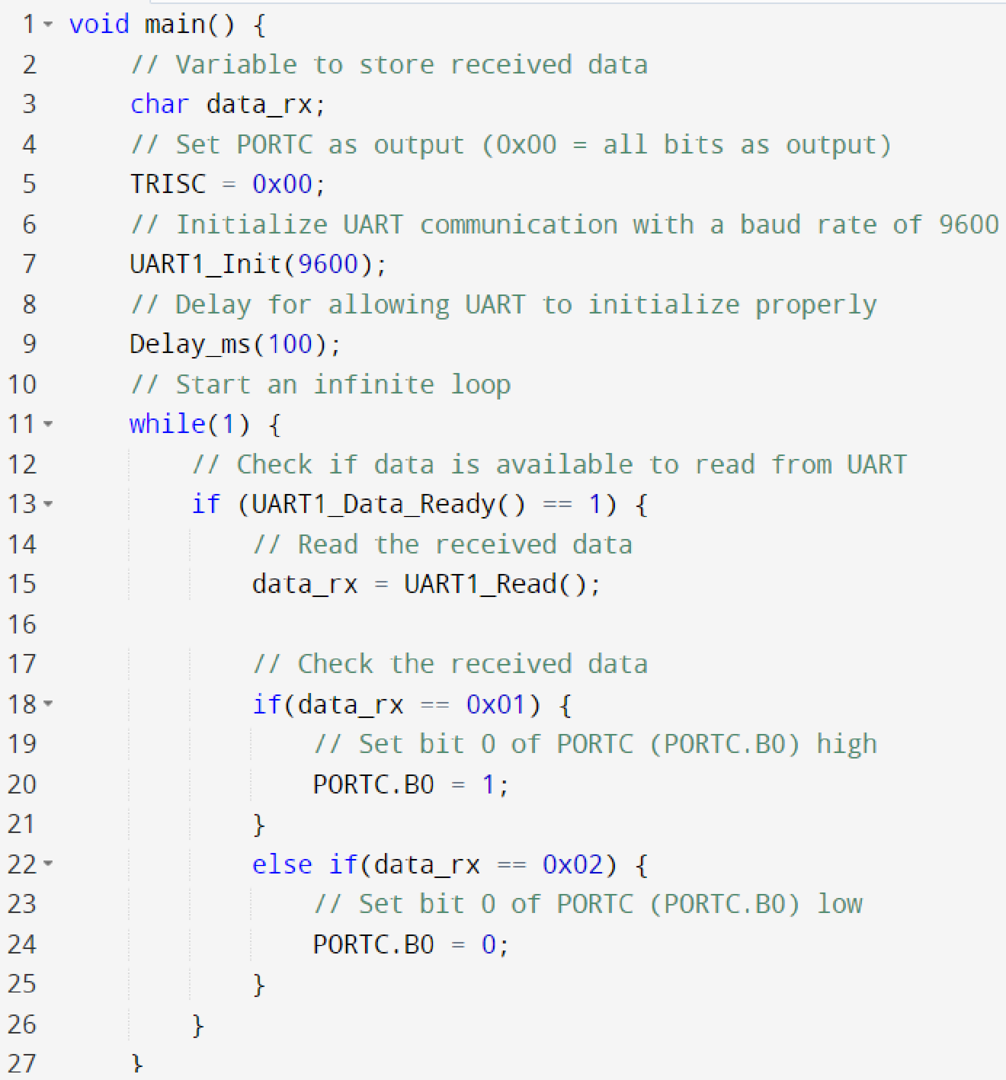
MCU

```
void main() {
  char data_rx;
  TRISC=0x00;
  UART1_Init(9600); Delay_ms(100);
  while(1) {
    if (UART1_Data_Ready()==1) {
      data_rx=UART1_Read();
      if(data_rx==0x01) {
        PORTC.B0=1; }
      else if(data_rx==0x02) {
        PORTC.B0=0; } }
} }
```

RX
MCU



```
void main() {
    // Set the lower half of PORTA as input
    TRISA = 0x0F;
    // Initialize UART communication with a baud rate of 9600
    UART1_Init(9600);
    // Delay for allowing UART to initialize properly
    Delay_ms(100);
    // Start an infinite loop
    while(1) {
        // Check if bit 0 of PORTA (PORTA.B0) is high
        if(PORTA.B0 == 1) {
            // Send 0x01 over UART
            UART1_Write(0x01);
        }
        // Check if bit 1 of PORTA (PORTA.B1) is high
        else if(PORTA.B1 == 1) {
            // Send 0x02 over UART
            UART1_Write(0x02);
        }
    }
}
```



```
void main() {
    // Variable to store received data
    char data_rx;
    // Set PORTC as output (0x00 = all bits as output)
    TRISC = 0x00;
    // Initialize UART communication with a baud rate of 9600
    UART1_Init(9600);
    // Delay for allowing UART to initialize properly
    Delay_ms(100);
    // Start an infinite loop
    while(1) {
        // Check if data is available to read from UART
        if (UART1_Data_Ready() == 1) {
            // Read the received data
            data_rx = UART1_Read();

            // Check the received data
            if(data_rx == 0x01) {
                // Set bit 0 of PORTC (PORTC.B0) high
                PORTC.B0 = 1;
            }
            else if(data_rx == 0x02) {
                // Set bit 0 of PORTC (PORTC.B0) low
                PORTC.B0 = 0;
            }
        }
    }
```

ANALOGUE TO DIGITAL CONVERTER

Microcontrollers are capable of detecting binary signals: is the button pressed or not?

THESE ARE DIGITAL SIGNALS

ADC

- Analog-to-digital (ADC) converters are among the most widely used devices for data acquisition.
- Digital Computer use binary (discrete) values, but in the physical world is analog (continuous) values.
- Examples of physical quantities: Temperature, Humidity, Pressure, Velocity

ADC

- A physical quantity is converted to electrical (Voltage, Current) signals using a device called transducer (also referred as sensors).
- Sensors for temperature, velocity, pressure, light etc. produce an output that is voltage (or current).

ADC

- Microcontroller → read digital values only.
- Therefore, an ADC converter is needed to translate (convert) the analog signals to digital numbers, so that the microcontroller can read and process them

CALCULATE THE DOUT

VREF=5 V.
VIN=3 V.
RESOLUTION=10 Bits.
DOUT=?

CALCULATE THE DOUT

VREF=5 V → MDOUT=1023

VIN=3 V → DOUT=?

DOUT= (VIN* MDOUT)/REF

DOUT= (3*1023)/5

DOUT= 613.8 → 614

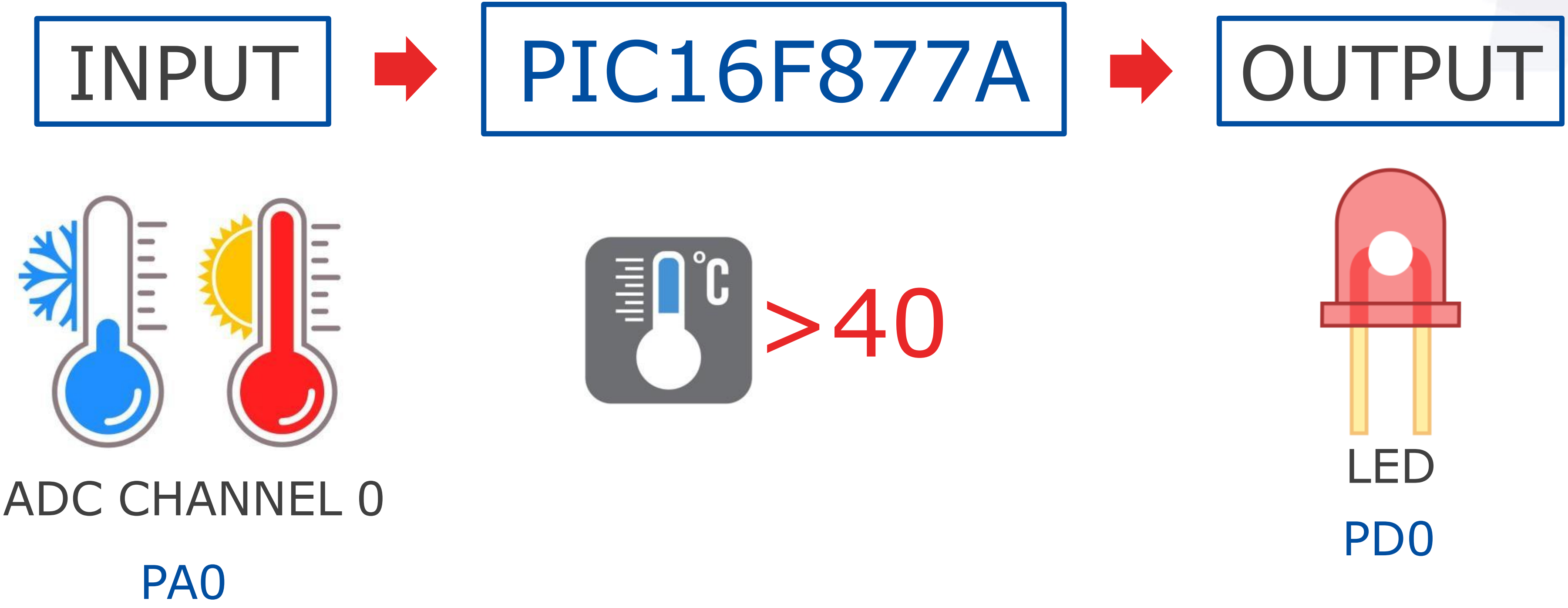
VREF=5 V.
VIN=3 V.
RES=10 Bits.
DOUT=?

TEMPERATURE MONITOR

A Temperature Sensor is connected to the PIC16F877A ADC Channel 0. The ADC has a Reference Voltage of 5V and a 10-Bit resolution. When the Temperature Sensor measures 100 ºC the DOUT is equal to the MAX DOUT for 10-Bit resolution.

When the temperature is higher than 40 ºC an LED should be turned ON to indicate that is too hot to go outside. Otherwise it should remain OFF. The LED is connected to PD0.

TEMPERATURE MONITOR



INPUT → PIC16F877A → OUTPUT

ADC CHANNEL 0

PA0

>40

LED

PD0

CALCULATE THE DOUT FOR 40 ºC

VREF=5 V → MDOUT=1023

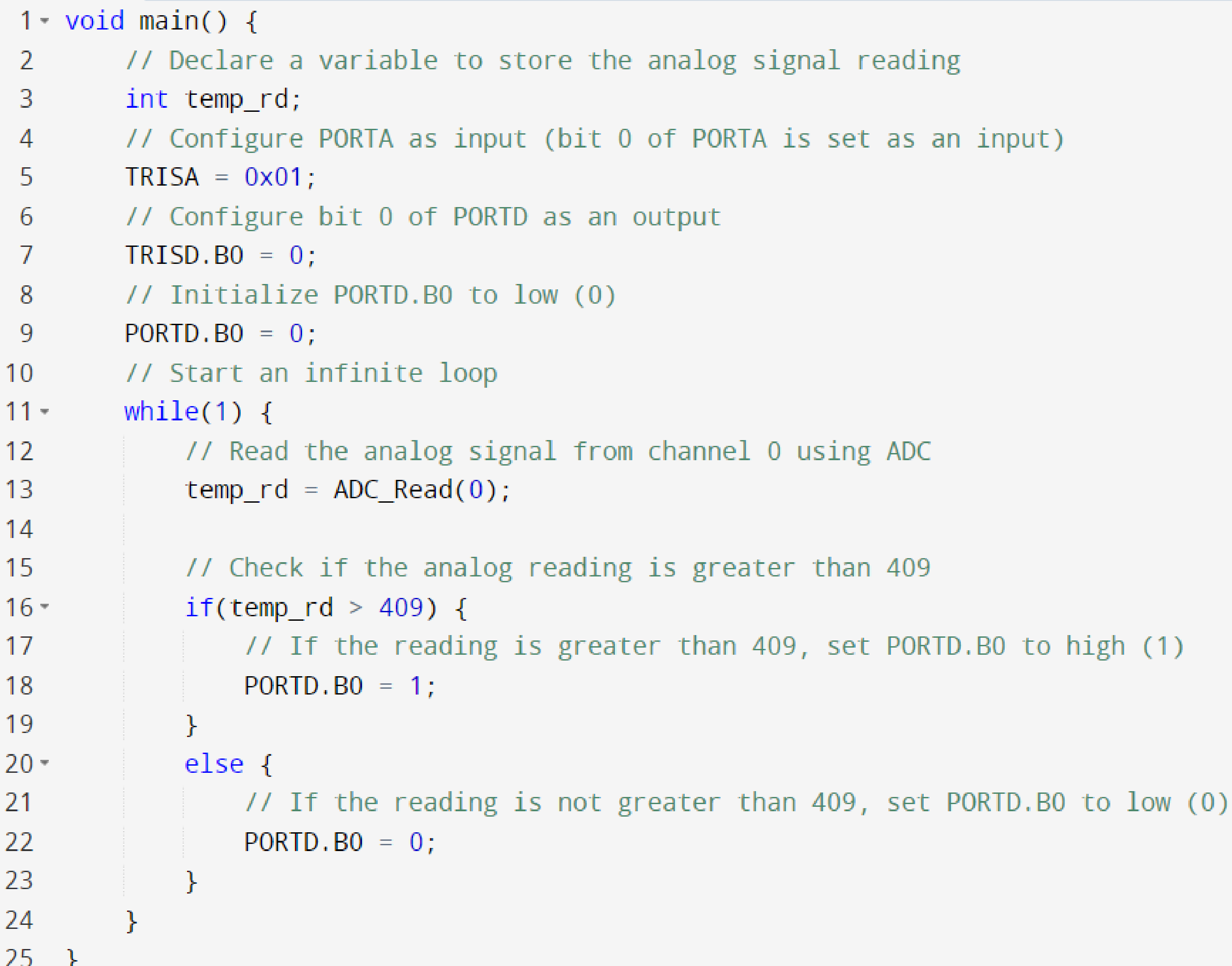
TEMP_IN=100 ºC → DOUT=1023

TEMP_IN=40 ºC → DOUT=?

DOUT= (40*1023)/100

DOUT= 409.2 → 409

```
void main() {
  int temp_rd;
  TRISA=0X01;
  TRISD.B0=0; PORTD.B0=0;
  while(1) {
    temp_rd = ADC_Read(0);
    if(temp_rd>409)
    {PORTD.B0=1;}
    else
    {PORTD.B0=0;}
  }
}
```



```c
void main() {
    // Declare a variable to store the analog signal reading
    int temp_rd;
    // Configure PORTA as input (bit 0 of PORTA is set as an input)
    TRISA = 0x01;
    // Configure bit 0 of PORTD as an output
    TRISD.B0 = 0;
    // Initialize PORTD.B0 to low (0)
    PORTD.B0 = 0;
    // Start an infinite loop
    while(1) {
        // Read the analog signal from channel 0 using ADC
        temp_rd = ADC_Read(0);

        // Check if the analog reading is greater than 409
        if(temp_rd > 409) {
            // If the reading is greater than 409, set PORTD.B0 to high (1)
            PORTD.B0 = 1;
        }
        else {
            // If the reading is not greater than 409, set PORTD.B0 to low (0)
            PORTD.B0 = 0;
        }
    }
}
```

CLASS CONTENT

C LANGUAGE

- INTERRUPTS
- TIMERS AND COUNTERS

INTERRUPTS

INTERRUPTS

MICROCONTROLLERS ARE USED TO PERFORM A SET OF PROGRAMMED TASKS WHICH GENERATE THE NECESSARY OUTPUTS BASED ON THE INPUTS

INTERRUPTS

BUT, WHILE THE MCU IS BUSY WITH EXECUTING ONE SEGMENT OF CODE THERE MIGHT BE AN EMERGENCY SITUATION WHERE ANOTHER SEGMENT OF CODE NEEDS IMMEDIATE ATTENTION

INTERRUPTS

THIS OTHER SEGMENT OF CODE THAT NEEDS IMMEDIATE ATTENTION SHOULD BE TREATED AS AN INTERRUPT, AND IT SERVES A SPECIAL TASK KNOWN AS INTERRUPT SERVICE ROUTINE (ISR) OR INTERRUPT HANDLER

INTERRUPTS

E.G. LET’S IMAGINE THAT YOU ARE PLAYING YOUR FAVORITE GAME ON YOUR PHONE AND THE MCU INSIDE YOUR PHONE IS BUSY THROWING ALL THE GRAPHICS THAT ARE NEEDED FOR YOU TO ENJOY THE GAME.

INTERRUPTS

SUDDENLY YOUR MOTHER CALLS TO YOUR NUMBER. THE WORST THING THAT COULD HAPPEN IS THAT YOUR MOBILE’S MCU NEGLECTS YOUR MOM’S CALL SINCE YOU ARE BUSY PLAYING A GAME. TO PREVENT THIS FROM HAPPENING WE USE SOMETHING CALLED INTERRUPTS.

INTERRUPTS

THESE INTERRUPTS WILL ALWAYS BE ACTIVE LISTENING FOR SOME PARTICULAR ACTIONS TO HAPPEN AND WHEN THEY OCCUR, A SEGMENT OF CODE WILL BE EXECUTED AND THEN THE PROGRAM WILL RETURN TO THE MAIN ROUTINE

INTERRUPTS

- EXTERNAL INTERRUPTS (HARDWARE INTERRUPTS)

- INTERNAL INTERRUPTS (SOFTWARE INTERRUPTS)

EXTERNAL INTERRUPTS

- GENERATED BY EXTERNAL HARDWARE AT CERTAIN PINS OF THE MCU

- THESE INTERRUPTIONS CAN GE TRIGGERED BY THE USER

INTERNAL INTERRUPTS

- GENERATED BY A SEGMENT OF CODE

INTERRUPTS IN PIC16F877A

- EXTERNAL
- TIMER 0
- TIMER 1
- RB PORT CHANGE
- PARALLEL SLAVE PORT READ/WRITE
- A/D CONVERTER
- USART RECEIVE

INTERRUPTS IN PIC16F877A

- USART TRANSMIT
- SYNCHRONOUS SERIAL PORT
- CCP1 (CAPTURE, COMPARE, PWM)
- CCP2 (CAPTURE, COMPARE, PWM)
- TMR2 TO PR2 MATCH
- COMPARATOR
- EEPROM WRITE OPERATION
- BUS COLLISION

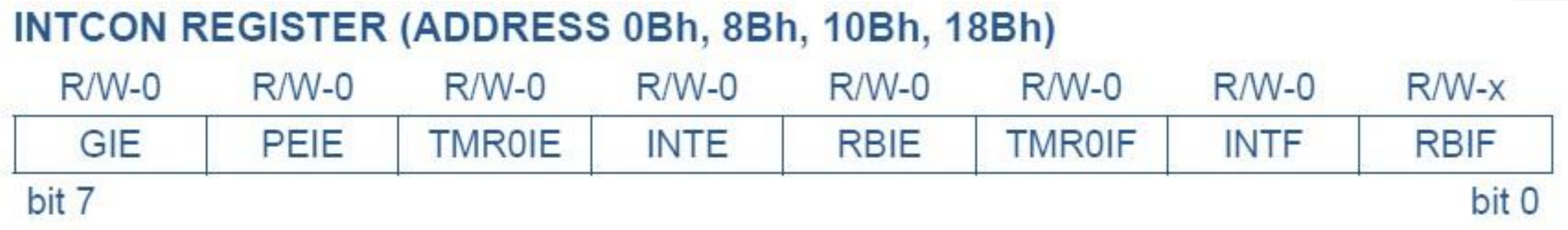
INTERRUPTS IN PIC16F877A

THE 5 REGISTERS THAT USED TO CONTROL THE OPERATION OF INTERRUPTS IN PIC 16F877A MICROCONTROLLER :

- INTCON
- PIE1
- PIR1
- PIE2
- PIR2

EXTERNAL INTERRUPT EXAMPLE

INTCON REGISTER



INTCON REGISTER (ADDRESS 0Bh, 8Bh, 10Bh, 18Bh)

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	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
bit 7							bit 0

INTCON Register is a readable and writeable register which contains various enable and flag bits for External and Internal Interrupts.

INTCON REGISTER

- GIE – Global Interrupt Enable
  1 – Enables all unmasked interrupts
  0 – Disables all interrupts

- PEIE – Peripheral Interrupt Enable
  1 – Enables all unmasked peripheral interrupts
  0 – Disables all peripheral interrupts

INTCON REGISTER

- TMR0IE – Timer 0 Overflow Interrupt Enable
  1 – Enables the TMR0 interrupt
  0 – Disables the TMR0 interrupt

- INTE – RB0/INT External Interrupt Enable
  1 – Enables the RB0/INT external interrupt
  0 – Disables the RB0/INT external interrupt

INTCON REGISTER

- RBIE – RB Port Change Interrupt Enable
  1 – Enables the RB port change interrupt
  0 – Disables the RB port change interrupt

- TMR0IF – Timer 0 Overflow Interrupt Flag
  1 – TMR0 register has overflowed. It must be cleared in software.
  0 – TMR0 register did not overflow

INTCON REGISTER

- INTF – RB0/INT External Interrupt Flag
  1 – The RB0/INT external interrupt occurred. It must be cleared in software.
  0 – The RB0/INT external interrupt did not occur

INTCON REGISTER

- RBIF – RB Port Change Interrupt Flag
  1 – At least one of the RB7 – RB4 pins changed state, a mismatch condition will continue to set the bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared. It must be cleared in software.
  0 – None of the RB7 – RB4 pins have changed state

INTCON REGISTER

- INTEDG bit of OPTION_REG Register is the Interrupt Edge Select bit. When it is 1 interrupt is on rising edge of RB0/INT pin and when it is 0 interrupt is on falling edge of RB0/INT pin.

EXAMPLE

A PUSH BUTTON SWITCH IS CONNECTED TO THE EXTERNAL INTERRUPT PIN INT OF THE PIC MICROCONTROLLER.

WHEN THIS BUTTON IS PRESSED, THE MICROCONTROLLER IS INTERRUPTED AND THE ISR IS EXECUTED. THE ISR TOGGLES THE STATUS OF PORTC FOR 1 SECOND.

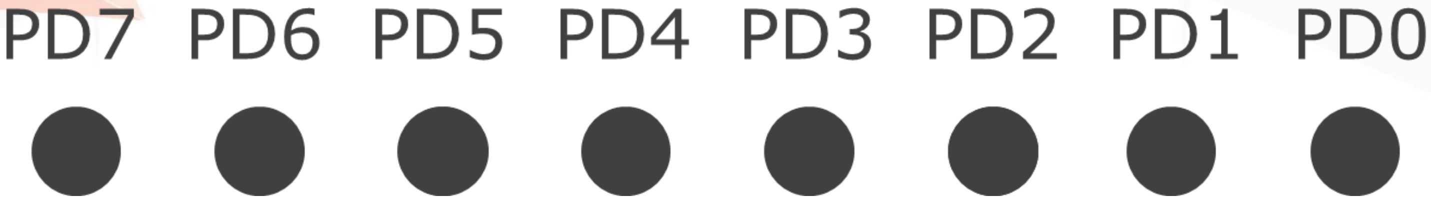
EXAMPLE – MIKROC CODE

INTERRUPTS CAN BE EASILY HANDLED BY USING RESERVED WORD “INTERRUPT”. MIKROC PRO FOR PIC MICROCONTROLLERS IMPLICITLY DECLARES A FUNCTION “INTERRUPT” TO HANDLE INTERRUPTS WHICH CANNOT BE REDECLARED

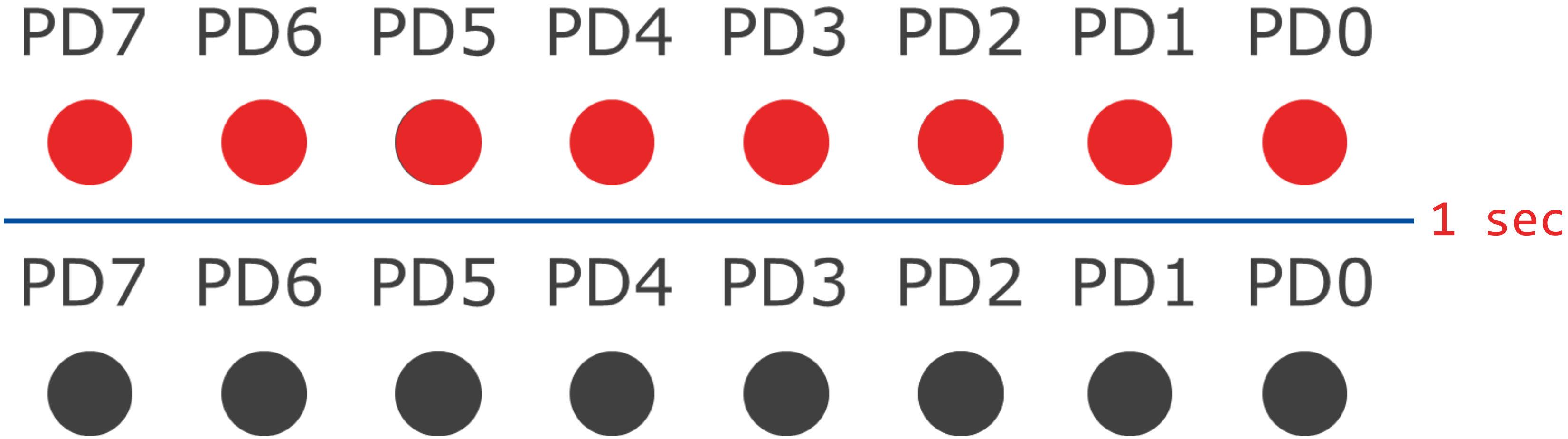
```
void main()
{
   TRISD = 0x00; // To configure PORTD as output port
   OPTION_REG.INTEDG = 1; //Set Rising Edge Trigger for INT
   INTCON.GIE = 1; // Enable The Global Interrupt
   INTCON.INTE = 1; // Enable INT
   while(1)
   {
      PORTD = 0x00; //Set some value at PORTD
   }
}
```

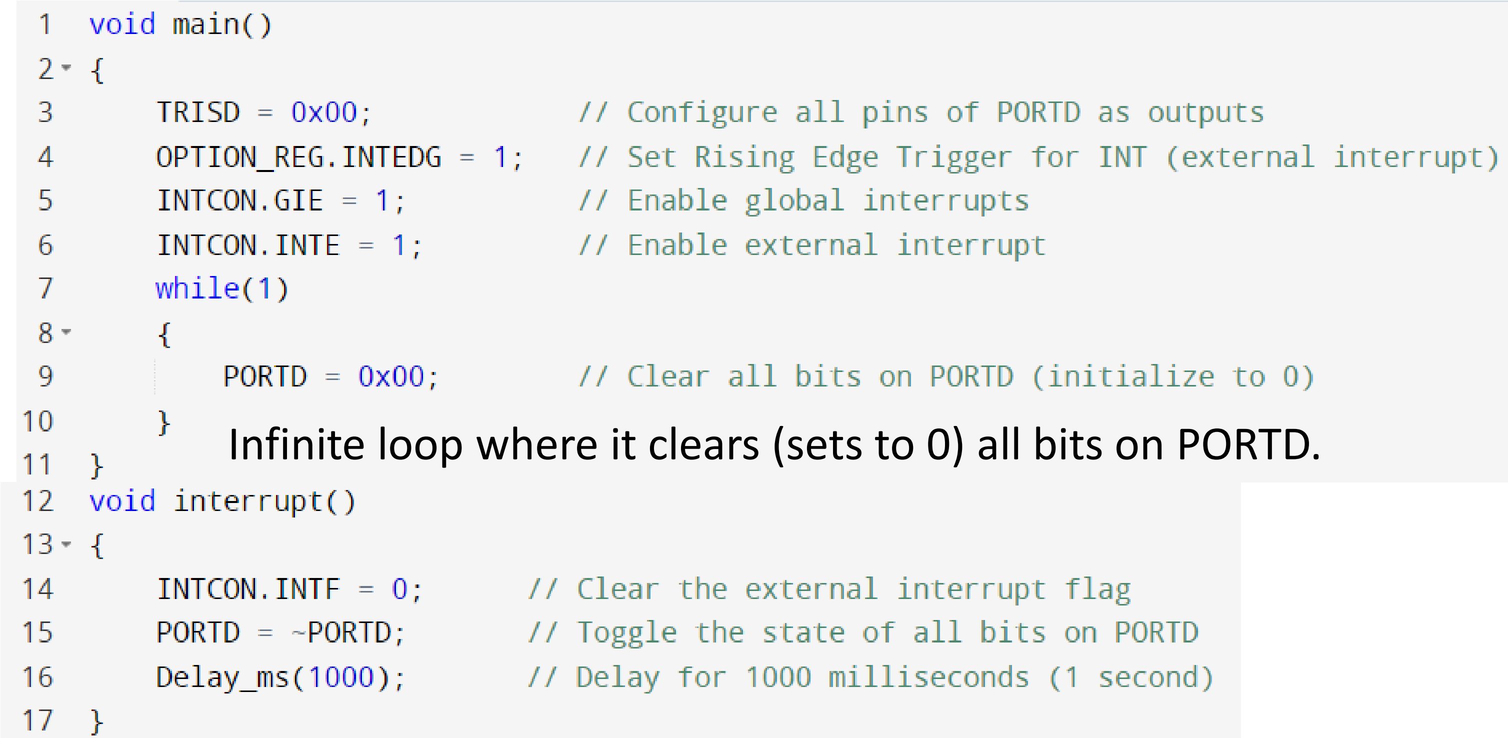
```
void interrupt() //  ISR
{
 INTCON.INTF=0; // Clear the interrupt 0 flag
 PORTD=~PORTD; // Invert (Toggle) the value at PORTD
 Delay_ms(1000); // Delay for 1 sec
}
```

```
while(1)
```



```
void interrupt()
```





```c
void main()
{
    TRISD = 0x00;                 // Configure all pins of PORTD as outputs
    OPTION_REG.INTEDG = 1;        // Set Rising Edge Trigger for INT (external interrupt)
    INTCON.GIE = 1;               // Enable global interrupts
    INTCON.INTE = 1;              // Enable external interrupt
    while(1)
    {
        PORTD = 0x00;             // Clear all bits on PORTD (initialize to 0)
    }
}
void interrupt()
{
    INTCON.INTF = 0;           // Clear the external interrupt flag
    PORTD = ~PORTD;            // Toggle the state of all bits on PORTD
    Delay_ms(1000);            // Delay for 1000 milliseconds (1 second)
}
```

Infinite loop where it clears (sets to 0) all bits on PORTD.

TIMERS AND COUNTERS

TIMERS AND COUNTERS

MANY TIMES, WE PLAN AND BUILD SYSTEMS THAT PERFORM VARIOUS PROCESSES THAT DEPEND ON TIME

TIMERS AND COUNTERS

SIMPLE EXAMPLE OF THIS PROCESS IS THE DIGITAL WRISTWATCH. THE ROLE OF THIS ELECTRONIC SYSTEM IS TO DISPLAY TIME IN A VERY PRECISE MANNER AND CHANGE THE DISPLAY EVERY SECOND (FOR SECONDS), EVERY MINUTE (FOR MINUTES) AND SO ON.

TIMERS AND COUNTERS

TO PERFORM THE STEPS WE'VE LISTED, THE SYSTEM MUST USE A TIMER, WHICH NEEDS TO BE VERY ACCURATE IN ORDER TO TAKE NECESSARY ACTIONS.THE CLOCK IS ACTUALLY A CORE OF ANY ELECTRONIC SYSTEM.

TIMERS AND COUNTERS

PIC MICROCONTROLLERS ARE EQUIPPED WITH ONE OR MORE PRECISION TIMING SYSTEMS KNOWN AS TIMERS.

TIMERS AND COUNTERS

TIMERS CAN BE USED TO PERFORM A VARIETY OF TIME PRECISION FUNCTIONS, SUCH AS GENERATING EVENTS AT SPECIFIC TIMES, MEASURING THE DURATION OF AN EVENT, KEEPING DATE AND TIME RECORD, COUNTING EVENTS, ETC.

TIMERS AND COUNTERS

THE MICROCONTROLLER PIC16F877 HAS 3 DIFFERENT TIMERS:

PIC TIMER0
PIC TIMER1
PIC TIMER2

TIMERS AND COUNTERS

THE TIMER0 MODULE TIMER/COUNTER HAS THE FOLLOWING FEATURES:

- 8-BIT TIMER/COUNTER
- READABLE AND WRITABLE
- 8-BIT SOFTWARE PROGRAMMABLE PRESCALER
- INTERNAL (4 MHZ) OR EXTERNAL CLOCK SELECT
- INTERRUPT ON OVERFLOW FROM 0xFF TO 0x00
- EDGE SELECT (RISING OR FALLING) FOR EXTERNAL CLOCK

TIMERS AND COUNTERS

TIMER0 HAS A REGISTER CALLED TMR0 REGISTER, WHICH IS 8 BITS OF SIZE.

WE CAN WRITE THE DESIRED VALUE INTO THE REGISTER WHICH WILL BE INCREMENT AS THE PROGRAM PROGRESSES. FREQUENCY VARIES DEPENDING ON THE PRESCALER. MAXIMUM VALUE THAT CAN BE ASSIGNED TO THIS REGISTER IS 255.

TIMERS AND COUNTERS

TMR0IF - TMR0 Overflow Interrupt Flag bit.

The TMR0 interrupt is generated when the TMR0 register overflows from 0xFF to 0x00. This overflow sets bit TMR0IF (INTCON<2>). You can initialize the value of this register to what ever you want (not necessarily "0").

We can read the value of the register TMR0 and write into. We can reset its value at any given moment (write) or we can check if there is a certain numeric value that we need (read).

TIMERS AND COUNTERS

WE CAN USE THESE TIMERS FOR VARIOUS IMPORTANT PURPOSES. WE MAINLY USED "DELAY PROCEDURES" TO IMPLEMENT SOME DELAY IN THE PROGRAM, THAT WAS COUNTING UP TO A SPECIFIC VALUE, BEFORE THE PROGRAM COULD BE CONTINUED

THANK YOU