CONTENT GENERATION

[UNDER EDUSAT PROGRAMME]

MICROCONTROLLER, EMBEDDED SYSTEM AND PLC

SUBJECT CODE-Theory 3

6TH SEM ETC, DIPLOMA ENGG.

UNDER SCTEVT ODISHA

PREPARED BY:

1. *Er. PRIYANKA DAS*

   [Lecturer (PT), Dept. of ETC, UCP ENGG.SCHOOL, Berhampur]

2. *Er. CHINMOY KUMAR PATNAIK*

   [Lecturer (PT), Dept. of ETC, UCP ENGG.SCHOOL, Berhampur]

INTRODUCTION TO EMBEDDED SYSTEM

EMBEDDED SYSTEM OVERVIEW:-

- Computing systems are everywhere. There is no surprise that millions of computing systems are built every year destined for desktop computers (Personal Computers, or PC's), workstations, mainframes and servers.
- The billions of computing systems are built every year for a very different purpose: they are embedded within larger electronic devices, repeatedly carrying out a particular function, often going completely unrecognized by the device's user.
- An embedded system is nearly any computing system other than a desktop, laptop, or mainframe computer.

Shortlist of embedded systems:-

- Embedded systems are found in a variety of common electronic devices, such as:
    1. **consumer electronics** -- cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games, calculators, and personal digital assistants;
    2. **home appliances** -- microwave ovens, answering machines, thermostat, home security, washing machines, and lighting systems;
    3. **office automation** -- fax machines, copiers, printers, and scanners;
    4. **business equipment** -- cash registers, curbside check-in, alarm systems, card readers, product scanners, and automated teller machines;
    5. **automobiles** -- transmission control, cruise control, fuel injection, anti-lock brakes, and active suspension.

Characteristics of embedded systems:-

- Embedded systems have several common characteristics:

1) **Single-functioned:** An embedded system usually executes only one program, repeatedly. For example, a pager is always a pager. A desktop system executes a variety of programs, like spreadsheets, word processors, and video games, with new programs added frequently.
2) **Tightly constrained:** All computing systems have constraints on design metrics. A design metric is a measure of an implementation's features, such as cost, size, performance, and power. Embedded systems often must cost just a few dollars, must be sized to fit on a single chip, must perform fast enough to process data in real-time, and must consume minimum power to extend battery life or prevent the necessity of a cooling fan.

3) **Reactive and real-time:** Many embedded systems must continually react to changes in the system's environment, and must compute certain results in real time without delay. For example, a car's cruise controller continually monitors and reacts to speed and brake sensors. It must compute acceleration or decelerations amounts repeatedly within a limited time; a delayed computation result could result in a failure to maintain control of the car. A desktop system typically focuses on computations, with relatively infrequent (from the computer's perspective) reactions to input devices. In addition, a delay in those computations, while perhaps inconvenient to the computer user, typically does not result in a system failure.

A DIGITAL CAMERA:-

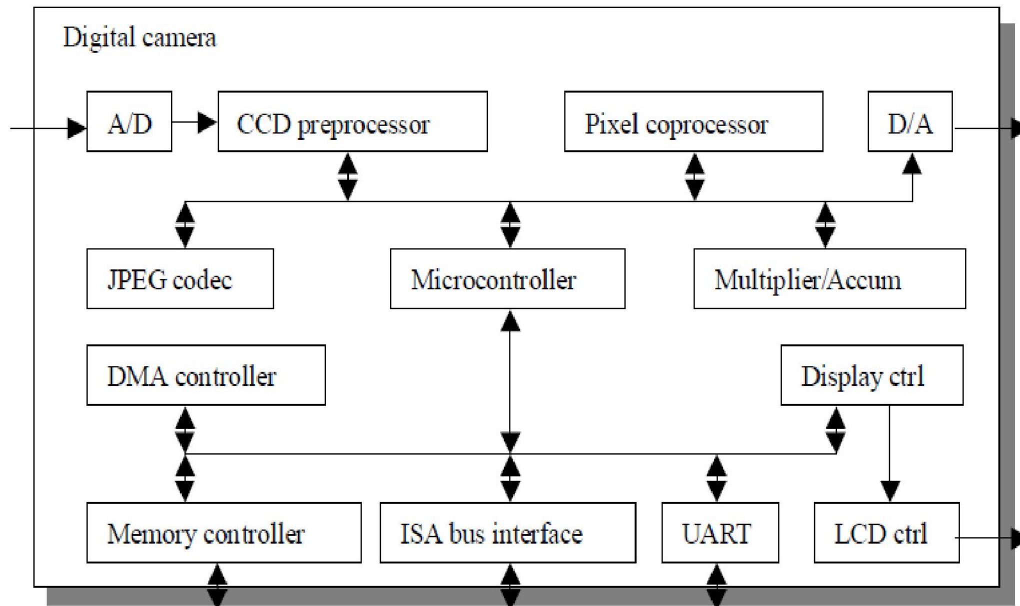
Consider the digital camera system as shown in the below figure .

- The **A2D** and **D2A** circuits convert analog images to digital and digital to analog, respectively.
- The **CCD preprocessor** is a charge-coupled device preprocessor.
- The **JPEG codec** compresses and decompresses an image using the JPEG2 compression standard, enabling compact storage in the limited memory of the camera.
- The **Pixel coprocessor** aids in rapidly displaying images.
- The **Memory controller** controls access to a memory chip also found in the camera, while the **DMA controller** enables direct memory access without requiring the use of the microcontroller.
- The **UART** enables communication with a PC's serial port for uploading video frames, while the **ISA bus interface** enables a faster connection with a PC's ISA bus.
- The **LCD ctrl** and **Display ctrl** circuits control the display of images on the camera's liquid-crystal display device.
- A **Multiplier/Accum** circuit assists with certain digital signal processing.
- The heart of the system is a **microcontroller**, which is a processor that controls the activities of all the other circuits. Each device as a processor designed for a particular task, while the microcontroller is a more general processor designed for general tasks.

This example illustrates some of the embedded system characteristics described above.

- It performs a single function repeatedly. The system always acts as a digital camera, wherein it captures, compresses and stores frames, decompresses and displays frames, and uploads frames.
- It is tightly constrained. The system must be low cost since consumers must be able to afford such a camera. It must be small so that it fits within a standard-sized camera. It must be fast so that it can process numerous images in milliseconds. It must consume little power so that the camera's battery will last a long time.

- This particular system does not possess a high degree of the characteristic of being reactive and real-time, as it only needs to respond to the pressing of buttons by a user, which even for an avid photographer is still quite slow with respect to processor speeds.



An embedded system example -- a digital camera.

EMBEDDED SYSTEMS TECHNOLOGIES:-

- Technology as a manner of accomplishing a task, especially using technical processes, methods, or knowledge.
- There are three technologies in the embedded system design:
    1. Processor technologies
    2. IC technologies
    3. Design technologies

PROCESSOR TECHNOLOGY:-

General Purpose Processors --- Software:-

- The general-purpose processor builds a device suitable for a variety of applications, to maximize the number of devices sold.
- One feature of such a processor is a program memory – the designer does not know what program will run on the processor, so the program cannot be build into the digital circuit.
- Another feature is a general datapath – the datapath must be general enough to handle a variety of computations, so typically has a large register file and one or more general-purpose arithmetic-logic units (ALUs).
- An embedded system simply uses a general-purpose processor, by programming the processor's memory to carry out the required functionality.

- Using a general-purpose processor in an embedded system may result in several design-metric benefits.
    1. **Design time** and **NRE** *cost* are low, because the designer must only write a program, but need not do any digital design.
    2. **Flexibility** is high, because changing functionality requires only changing the program.
    3. **Unit cost** may be relatively low in small quantities, since the processor manufacturer sells large quantities to other customers.
    4. **Performance** may be fast for computation-intensive applications, if using a fast processor, due to advanced architecture features and leading edge IC technology.
- There are also some design-metric drawbacks.
    a. **Unit cost** may be too high for large quantities.
    b. **Performance** may be slow for certain applications.
    c. **Size** and **power** may be large due to unnecessary processor hardware.

Figure 1(b) illustrates that a general-purpose covers the desired functionality, but not necessarily efficiently.

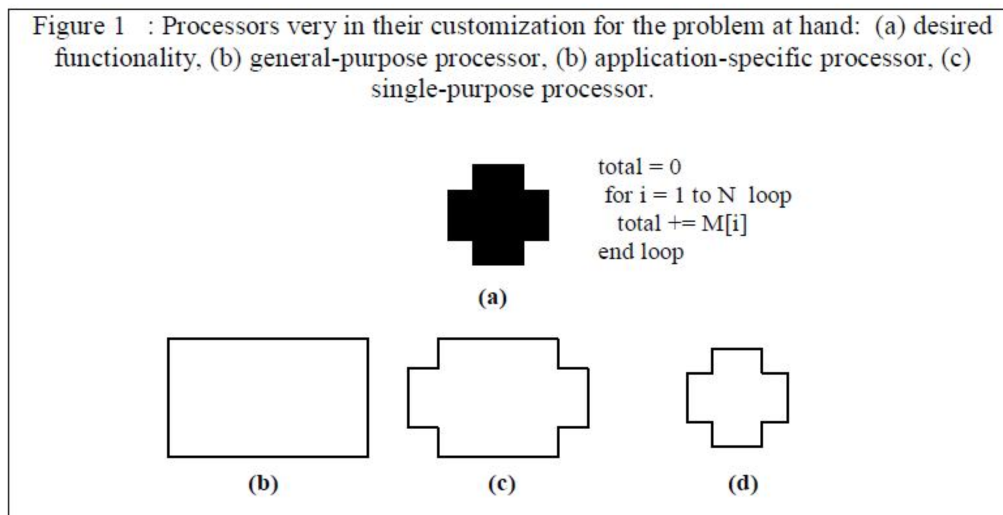
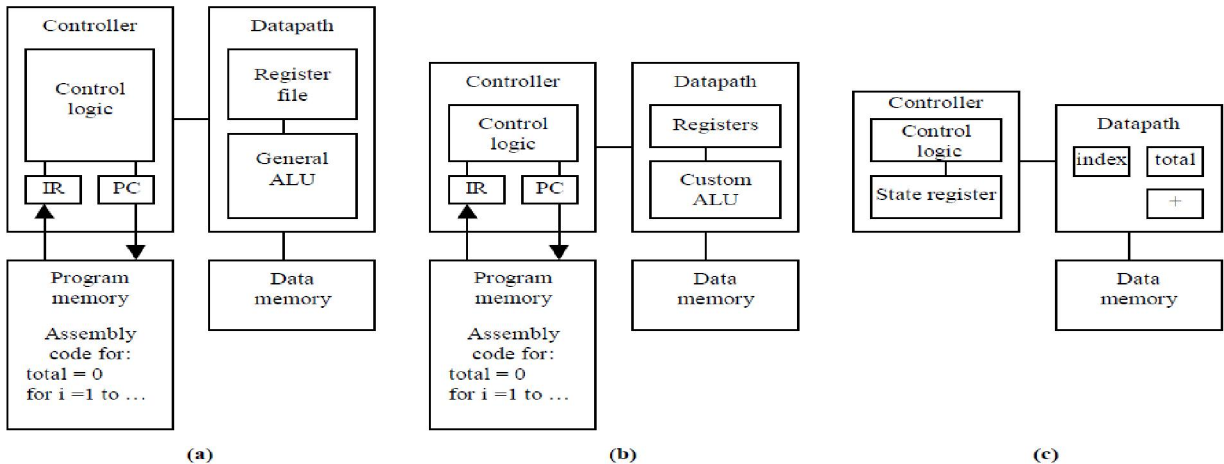


Figure 1 : Processors very in their customization for the problem at hand: (a) desired functionality, (b) general-purpose processor, (b) application-specific processor, (c) single-purpose processor.

Figure 2(a) shows a simple architecture of a general-purpose processor implementing the array summing functionality. The functionality is stored in a program memory. The controller fetches the current instruction, as indicated by the program counter (PC), into the instruction register (IR). It then configures the datapath for this instruction and executes the instruction. Finally, it determines the appropriate next instruction address, sets the PC to this address, and fetches again.

Figure 2:

Implementing desired functionality on different processor types: (a) general-purpose, (b) application-specific, (c) single-purpose.



Single Purpose Processors ---- Hardware:-

- A single-purpose processor is a digital circuit designed to execute exactly one program. For example, consider the digital camera. All of the components other than the microcontroller are single-purpose processors. The JPEG codec, for example, executes a single program that compresses and decompresses video frames.
- An embedded system creates a single-purpose processor by designing a custom digital circuit.
- Using a single-purpose processor in an embedded system results in several design metric benefits and drawbacks, which are essentially the inverse of those for general purpose processors.
- Performance may be fast, size and power may be small, and unit-cost may be low for large quantities, while design time and NRE costs may be high, flexibility is low, unit cost may be high for small quantities, and performance may not match general-purpose processors for some applications.

For example, Figure 1(d) illustrates the use of a single-purpose processor in our embedded system example, representing an exact fit of the desired functionality. Figure 2(c) illustrates the architecture of such a single-purpose processor for the example. Since the example counts from one to N, we add an index register. The index register will be loaded with N, and will then count down to zero, at which time it will assert a status line read by the controller. The example has only one other value, we add only one register labeled total to the datapath. Since the example's only arithmetic operation is addition, we add a single adder to the datapath. Since the processor only executes this one program, we hardware the program directly into the control logic.

APPLICATION ---- SPECIFIC PROCESSORS:-

- An application-specific instruction-set processor (or ASIP) can serve as a compromise between the other processor.
- An ASIP is designed for a particular class of applications with common characteristics, such as digital-signal processing, telecommunications, embedded control, etc.
- An ASIP in an embedded system can provide the benefit of flexibility while still achieving good performance, power and size.
- Such processors can require large NRE cost to build the processor itself.

Microcontrollers:-

- A microcontroller is a microprocessor that has been optimized for embedded control applications.
- Such applications typically monitor and set numerous single bit control signals but do not perform large amount of data computations. Thus microcontrollers tend to have simple datapaths that excel bit-level operations and reading and writing external bits.
- Furthermore, they tend to incorporate on the microprocessor chip several peripheral components common in control applications like serial communication peripherals, times, counters, pulse width modulators and analog to digital converters. Such incorporation of peripherals enables single chip implementations and hence smaller and lower cost product.

Digital Signal Processing:-

- Digital-signal processors (DSPs) are a common class of ASIP.
- A DSP is a processor designed to perform common operations on digital signals, which are the digital encodings of analog signals like video and audio. These operations carry out common signal processing tasks like signal filtering, transformation, or combination.
- Such operations are usually math-intensive, including operations like multiply and add or shift and add.
- To support such operations, a DSP may have special purpose datapath components such a multiply-accumulate unit, which can perform a computation like $T = T + M[i]*k$ using only one instruction.
- Figure 1(c) illustrates the use of an ASIP ; while partially customized to the desired functionality, there is some inefficiency since the processor also contains features to support reprogramming.
- Figure 2(b) shows the general architecture of an ASIP. The datapath may be customized. It may have an auto-incrementing register, a path that allows the add of a register plus a memory location in one instruction, fewer registers, and a simpler controller.

IC TECHNOLOGY:-

- Every processor must eventually be implemented on an IC.
- An IC (Integrated Circuit), often called a “chip,” is a semiconductor device consisting of a set of connected transistors and other devices.
- A number of different processes exist to build semiconductors, the most popular of which is CMOS (Complementary Metal Oxide Semiconductor).
- Semiconductors consist of numerous layers as shown in the figure given below.

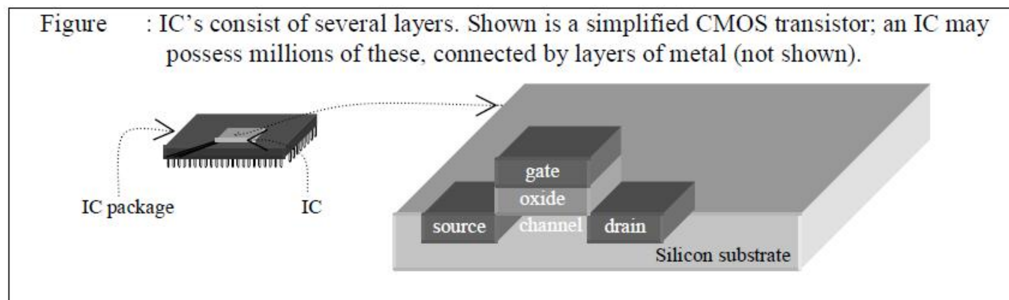


Figure : IC’s consist of several layers. Shown is a simplified CMOS transistor; an IC may possess millions of these, connected by layers of metal (not shown).

- The bottom layers form the transistors. The middle layers form logic gates. The top layers connect these gates with wires. These layers can be created by depositing photo-sensitive chemicals on the chip surface and then shining light through masks to change regions of the chemicals. A set of masks is often called a layout. The narrowest line that we can create on a chip is called the feature size.

Full Custom / VLSI:-

- In a full-custom IC technology, we optimize all layers for our particular embedded system’s digital implementation.
- Such optimization includes placing the transistors to minimize interconnection lengths, sizing the transistors to optimize signal transmissions and routing wires among the transistors.
- Once all the masks are completed, then we send the mask specifications to a fabrication plant that builds the actual ICs.
- Full-custom IC design, often referred to as VLSI (Very Large Scale Integration) design, has very high NRE cost and long turnaround times (typically months) before the IC becomes available, but can yield excellent performance with small size and power.
- It is usually used only in high-volume or extremely performance-critical applications.

Semicustom ASIC (Gate Array and Standard Cell):-

- In an ASIC (Application-Specific IC) technology, the lower layers are fully or partially built, leaving us to finish the upper layers.

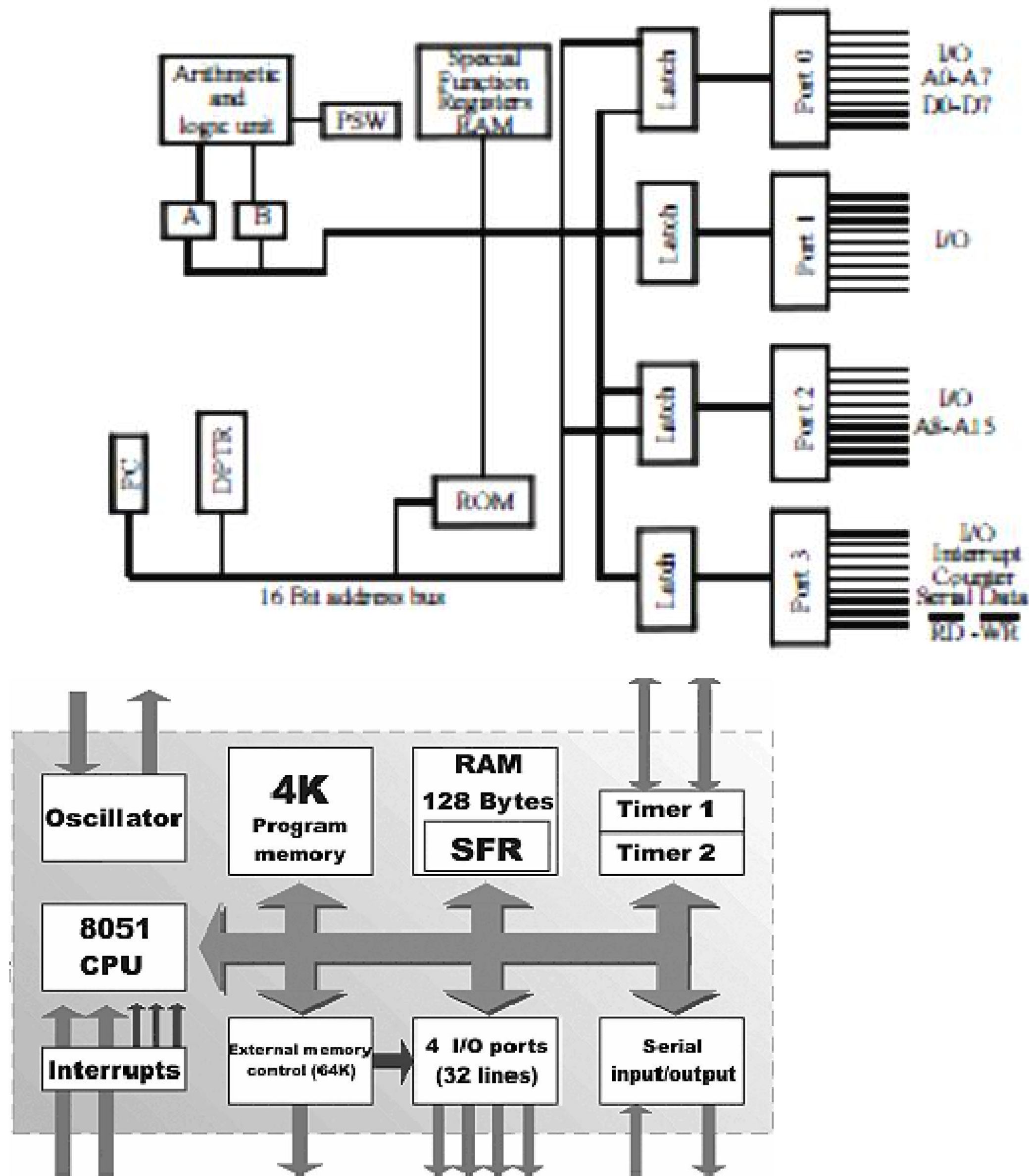
- In a gate array technology, the masks for the transistor and gate levels are already built (i.e., the IC already consists of arrays of gates).
- The remaining task is to connect these gates to achieve our particular implementation.
- In a standard cell technology, logic-level cells (such as an AND gate or an AND-OR-INVERT combination) have their mask portions pre-designed, usually by hand.
- Thus, the remaining task is to arrange these portions into complete masks for the gate level, and then to connect the cells.
- ASICs are by far the most popular IC technology, as they provide for good performance and size, with much less NRE cost than full-custom IC's.

PLD:-

- In a PLD (Programmable Logic Device) technology, layers implement a programmable circuit, where programming has a lower-level meaning than a software program.
- The programming that takes place may consist of creating or destroying connections between wires that connect gates, either by blowing a fuse, or setting a bit in a programmable switch.
- Small devices, called programmers, connected to a desktop computer can typically perform such programming.
- PLD's of two types, simple and complex. One type of simple PLD is a PLA (Programmable Logic Array), which consists of a programmable array of AND gates and a programmable array of OR gates.
- Another type is a PAL (Programmable Array Logic), which uses just one programmable array to reduce the number of expensive programmable components.
- One type of complex PLD, growing very rapidly in popularity over the past decade, is the FPGA (Field Programmable Gate Array), which offers more general connectivity among blocks of logic, rather than just arrays of logic as with PLAs and PALs, and are thus able to implement far more complex designs. PLDs offer very low NRE cost and almost instant IC availability.
- They are typically bigger than ASICs, may have higher unit cost, may consume more power, and may be slower (especially FPGAs). They still provide reasonable performance, though, so are especially well suited to rapid prototyping.

8051 MICROCONTROLLER ARCHITECTURE

The architecture of the 8051 family of microcontrollers is referred to as the MCS-51 architecture (Micro Controller Series-51), or sometimes simply as MCS-51.The microcontrollers have an 8-bit data bus. They are capable of addressing 64K of program memory and a separate 64K of data memory. The block diagram of 8051 microcontroller is shown below.

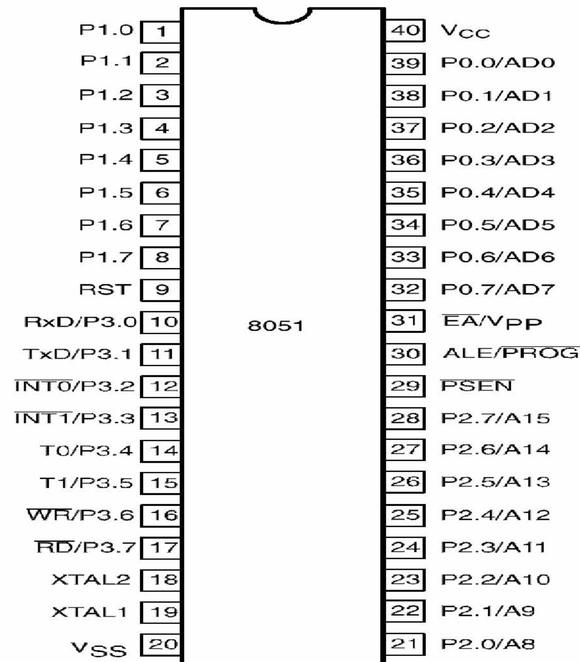


The 8051 have 4K of code memory implemented as on-chip *Read Only Memory* (ROM). The 8051 have 128 bytes of internal *Random Access Memory* (RAM).

The 8051 has two timer/counters, a serial port, 4 general purpose parallel input/output ports, and interrupt control logic with five sources of interrupts.

Besides internal RAM, the 8051 have various Special *Function Registers* (SFR), which are the control and data registers for on-chip facilities.
The SFRs also include the accumulator, the B register, and the *Program StatusWord* (PSW), which contains the CPU flags.

Programming the various internal hardware facilities of the 8051 is achieved by placing the appropriate control words into the corresponding SFRs. The 8031 are similar to the 8051, except it lacks the on-chip ROM.



1–8: Port 1: Each of these pins can be used as either input or output according to user needs. Also, pins 1 and 2 (P1.0 and P1.1) have special functions associated with Timer 2.

9: Reset Signal: High logical state on this input halts the MCU and clears all the registers. Bringing this pin back to logical state zero starts the program a new as if the power had just been turned on. In another words, positive voltage impulse on this pin resets the MCU. Depending on the device's purpose and environs, this pin is usually connected to the push-button, reset-upon-start circuit or a brown out reset circuit.

10-17: Port 3:as with Port 1, each of these pins can be used as universal input or output. However, each pin of Port 3 has an alternative function:

1. Pin 10: **RXD** - serial input for asynchronous communication or serial output for synchronous communication.
2. Pin 11: **TXD** - serial output for asynchronous communication or clock output for synchronous communication

3. Pin 12: **INT0*** - input for interrupt 0
4. Pin 13: **INT1*** - input for interrupt 1
5. Pin 14: **T0** - clock input of counter 0
6. Pin 15: **T1** - clock input of counter 1
7. Pin 16: **WR*** - signal for writing to external (add-on) RAM memory.
8. Pin 17: **RD*** - signal for reading from external RAM memory

18-19: X2 and X1: Input and output of internal oscillator. Quartz crystal controlling the frequency commonly connects to these pins. Capacitances within the oscillator mechanism optimal voltage.

20: GND: Ground

21- 28: Port 2: if external memory is not present, pins of Port 2 act as universal input/output. If external memory is present, this is the location of the higher address byte, i.e. addresses A8 – A15. It is important to note that in cases when not all the 8 bits are used for addressing the memory (i.e. memory is smaller than 64kB), the rest of the unused bits are not available as input/output.

29: PSEN*: MCU activates this bit (brings to low state) upon each reading of byte instruction) from program memory. If external ROM is used for storing the program, PSEN- is directly connected to its control pins.

30: ALE: before each reading of the external memory, MCU sends the lower byte of the address register (addresses A0 – A7) to port P0 and activates the output ALE. External Chip (eg: 74HC373), memorizes the state of port P0 upon receiving a signal from ALE pin, and uses it as part of the address for memory chip. During the second part of the MCU cycle, signal on ALE is off, and port P0 is used as *Data Bus*. In this way, by adding only one integrated circuit, data from port can be multiplexed and the port simultaneously used for transferring both addresses and data.

31: EA*: Bringing this pin to the logical state zero designates the ports P2 and P3 for transferring addresses regardless of the presence of the internal memory. This means that even if there is a program loaded in the MCU it will not be executed, but the one from the external ROM will be used instead. Conversely, bringing the pin to the high logical state causes the controller to use both memories, first the internal, and then the external (if present).

32-39: Port 0: Similar to Port 2, pins of Port 0 can be used as universal input/output, if external memory is not used. If external memory is used, P0 behaves as address output (A0 – A7) when ALE pin is at high logical level, or as data output (Data Bus) when ALE pin is at low logical level.

40: VCC: Power +5V

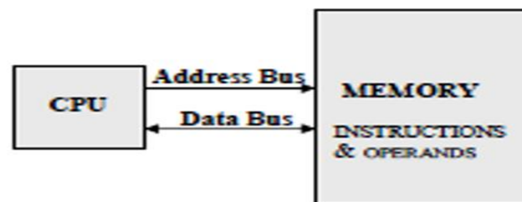
MEMORY ORGANIZATION

During the runtime, microcontroller uses two different types of memory: one for holding the program being executed (ROM memory), and the other for temporary storage of data and auxiliary variables (RAM memory). Depending on the particular model from 8051 family, this is usually few kilobytes of ROM and 128/256 bytes of RAM. This amount is built-in and is sufficient for common tasks performed "independently" by the MCU. However, 8051 can address up to 64KB of external memory.

MEMORY ARCHITECTURE

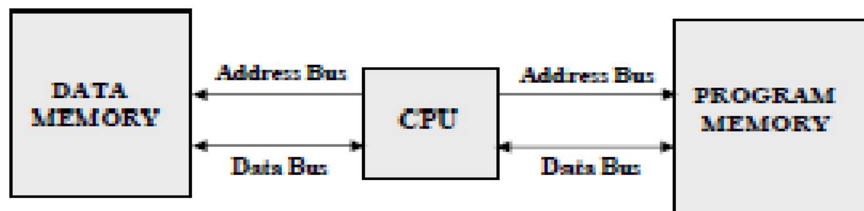
(i) VON-NEUMAN ARCHITECTURE

Von Neumann architectures are computer architectures that use the same storage device for both instructions and data. By treating the instructions in the same way as the data, the machine could easily change the instructions. In other words the machine was reprogram able. Because the machine did not distinguish between instructions and data, it allowed a program to modify or replicate a program.



(ii)HARVARD ARCHITECTURE

The term **Harvard architecture** originally referred to computer architectures that uses physically separate storage devices for their instructions and data. **Harvard architecture** has separate data and instruction busses, allowing transfers to be performed simultaneously on both buses.



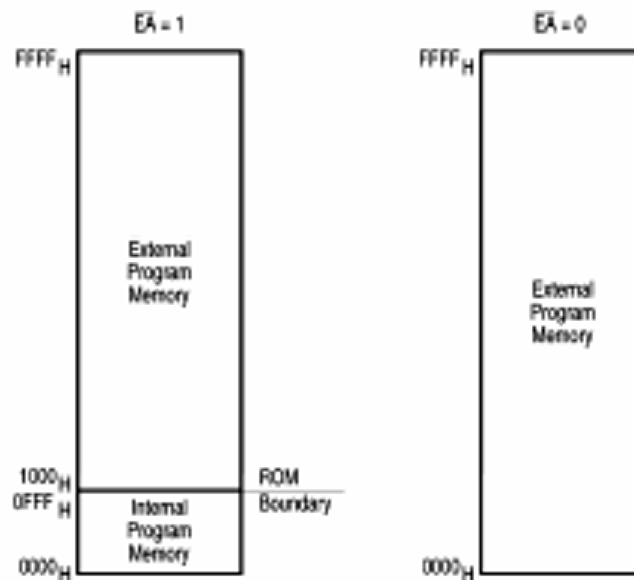
The Harvard architecture executes instructions in fewer instruction cycles than the Von Neumann architecture. This is because a much greater amount of instruction parallelism is possible in the Harvard architecture. Parallelism means that fetches for the next instruction can take place during the execution of the current instruction, without having to either wait for a "dead" cycle of the instruction's execution or stop the processor's operation while the next instruction is being fetched.

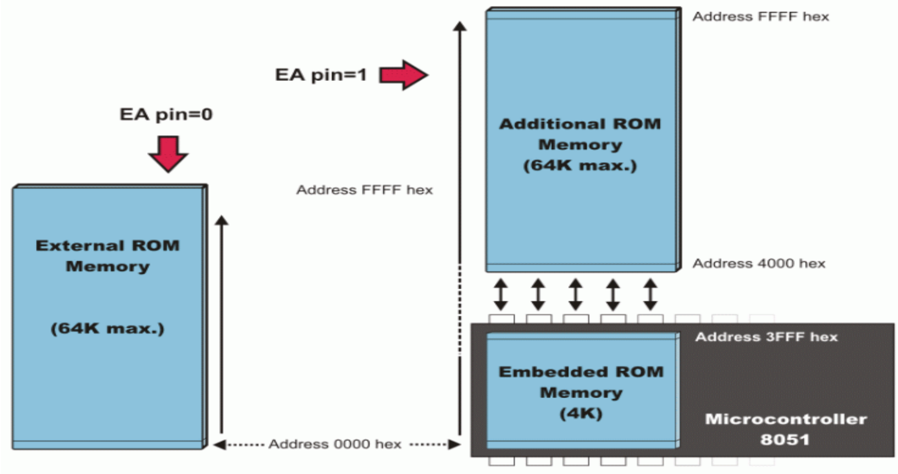
MEMORY ORGANIZATION

The 8051 has two types of memory and these are Program Memory and Data Memory. Program Memory (ROM) is used to permanently save the program being executed, while Data Memory (RAM) is used for temporarily storing data and intermediate results created and used during the operation of the microcontroller. Depending on the model in use (we are still talking about the 8051 microcontroller family in general) at most a few Kb of ROM and 128 or 256 bytes of RAM is used. All 8051 microcontrollers have a 16-bit addressing bus and are capable of addressing 64 kb memory. It is neither a mistake nor a big ambition of engineers who were working on basic core development. It is a matter of smart memory organization which makes these microcontrollers a real “programmers’ goody“.

PROGRAM MEMORY/ROM MEMORY

8051 have built-in ROM, although there are substantial variations. With some models internal memory cannot be programmed directly by the user. Instead, the user needs to precede the program to the manufacturer, so that the MCU can be programmed appropriately in the process of fabrication. Obviously, this option is cost-effective only for large series. Fortunately, there are MCU models ideal for experimentation and small specialized series. Many manufacturers deliver controllers that can be programmed directly by the user. These come in an EPROM version or EEPROM version or OTP or FLASH type.





EA=0 In this case, the microcontroller completely ignores internal program memory and executes only the program stored in external memory.
EA=1 In this case, the microcontroller executes first the program from built-in ROM, then the program stored in external memory.
In both cases, P0 and P2 are not available for use since being used for data and address transmission. Besides, the ALE and PSEN pins are also used.

RAM MEMORY/DATA MEMORY

Data Memory is used for temporarily storing data and intermediate results created and used during the operation of the microcontroller. Besides, RAM memory built in the 8051 family includes many registers such as hardware counters and timers, input/output ports, serial data buffers etc.

The previous models had 256 RAM locations, while for the later models this number was incremented by additional 128 registers.

However, the first 256 memory locations (addresses 0-FFh) are the heart of memory common to all the models belonging to the 8051 family.

Locations available to the user occupy memory space with addresses 0-7Fh, i.e. first 128 registers. This part of RAM is divided in several blocks.

The first block consists of 4 banks each including 8 registers denoted by R0-R7.

Prior to accessing any of these registers, it is necessary to select the bank containing it.

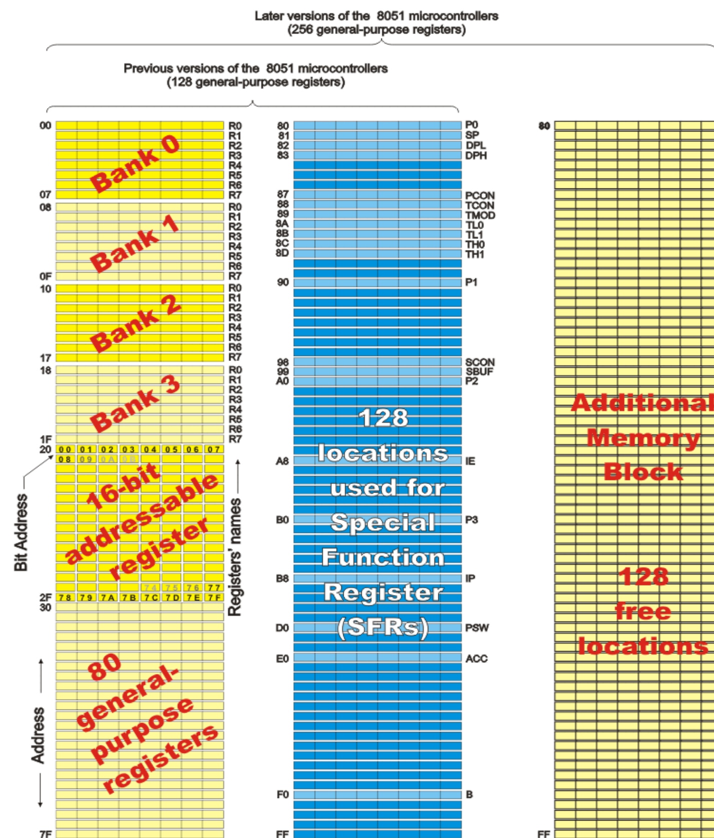
The next memory block (address 20h-2Fh) is bit- addressable, which means that each bit has its own address (0-7Fh). Since there are 16 such registers, this block contains in total of 128 bits with separate addresses (address of bit 0 of the 20h byte is 0, while address of bit 7 of the 2Fh byte is 7Fh).

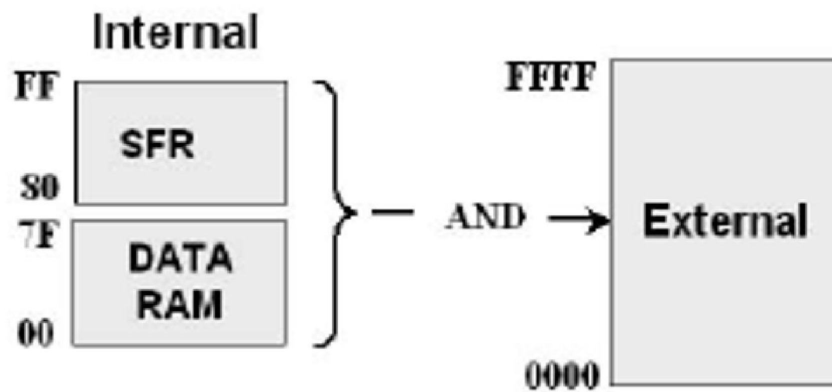
The third group of registers occupy addresses 2Fh-7Fh, i.e. 80 locations, and does not have any special functions or features.

ADDITIONAL RAM

In order to satisfy the programmers' constant hunger for Data Memory, the manufacturers decided to embed an additional memory block of 128 locations into the latest versions of the 8051 microcontrollers. (However, it's not as simple as it seems to be… The problem is that electronics performing addressing has 1 byte (8 bits) on disposal and is capable of reaching only the first 256 locations, therefore. In order to keep already existing 8-bit architecture and compatibility with other existing models a small trick was done. It means that additional memory block shares the same addresses with locations intended for the SFRs (80h- FFh).)

In order to differentiate between these two physically separated memory spaces, different ways of addressing are used. The SFRs memory locations are accessed by direct addressing, while additional RAM memory locations are accessed by indirect addressing.





Address								
FF_H								
$F8_H$	FF	FE	FD	FC	FB	FA	F9	F8
$F0_H$	F7	F6	F5	F4	F3	F2	F1	F0
$E8_H$	EF	EE	ED	EC	EB	EA	E9	E8
$E0_H$	E7	E6	E5	E4	E3	E2	E1	E0
$D8_H$	DF	DE	DD	DC	DB	DA	D9	D8
$D0_H$	D7	D6	D5	D4	D3	D2	D1	D0
$C8_H$	CF	CE	CD	CC	CB	CA	C9	C8
$C0_H$	C7	C6	C5	C4	C3	C2	C1	C0
$B8_H$	BF	BE	BD	BC	BB	BA	B9	B8
$B0_H$	B7	B6	B5	B4	B3	B2	B1	B0
$A8_H$	AF	AE	AD	AB	AC	AA	A8	A9
$A0_H$	A7	A6	A5	A4	A3	A2	A1	A0
98_H	9F	9E	9D	9B	9C	9A	98	99
90_H	97	96	95	94	93	92	91	90
88_H	8F	8E	8D	8B	8C	8A	88	89
80_H	87	86	85	84	83	82	81	80

Internal SFR Area
(direct addressable)
128 Byte

Address	Area
FF_H – 80_H	Upper Internal Data RAM (Optional) (indirect addressable) 128 Byte
$7F_H$ – 00_H	Lower Internal Data RAM (indirect & direct addressable) 128 Byte

Address								
$7F_H$ – 30_H	RAM Area							
$2F_H$	7F	7E	7D	7C	7B	7A	79	78
$2E_H$	77	76	75	74	73	72	71	70
$2D_H$	6F	6E	6D	6C	6B	6A	69	68
$2C_H$	67	66	65	64	63	62	61	60
$2B_H$	5F	5E	5D	5C	5B	5A	59	58
$2A_H$	57	56	55	54	53	52	51	50
29_H	4F	4E	4D	4C	4B	4A	49	48
28_H	47	46	45	44	43	42	41	40
27_H	3F	3E	3D	3C	3B	3A	39	38
26_H	37	36	35	34	33	32	31	30
25_H	2F	2E	2D	2C	2B	2A	29	28
24_H	27	26	25	24	23	22	21	20
23_H	1F	1E	1D	1C	1B	1A	19	18
22_H	17	16	15	14	13	12	11	10
21_H	0F	0E	0D	0C	0B	0A	09	08
20_H	07	06	05	04	03	02	01	00
$1F_H$ – 18_H	Registerbank 3							
17_H – 10_H	Registerbank 2							
$0F_H$ – 08_H	Registerbank 1							
07_H	R7							
06_H	R6							
05_H	R5							
04_H	R4							
03_H	R3							
02_H	R2							
01_H	R1							
00_H	R0							

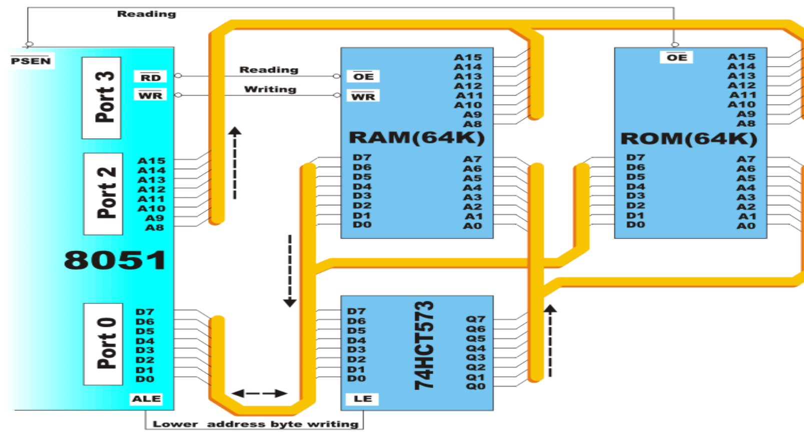
16 Bytes with 128 bit addressable Bits

Registerbank 0

Optional = This area of memory is not contain in 8051, it is available in other MCS-51 series

MEMORY EXPANSION

In case memory (RAM or ROM) built in the microcontroller is not sufficient, it is possible to add two external memory chips with capacity of 64Kb each. P2 and P3 I/O ports are used for their addressing and data transmission.



The 8051 microcontroller has two pins for data read RD#(P3.7) and PSEN#.
The first one is used for reading data from external data memory (RAM), while the other is used for reading data from external program memory (ROM). Both pins are active low.
A typical example of memory expansion by adding RAM and ROM chips (Hardward architecture), is shown in figure above.
Even though additional memory is rarely used with the latest versions of the microcontrollers, we will describe in short what happens when memory chips are connected according to the previous schematic. The whole process described below is performed automatically.

- When the program during execution encounters an instruction which resides in external memory (ROM), the microcontroller will activate its control output ALE and set the first 8 bits of address (A0-A7) on P0. IC circuit 74HCT573 passes the first 8 bits to memory address pins.
- A signal on the ALE pin latches the IC circuit 74HCT573 and immediately afterwards 8 higher bits of address (A8-A15) appear on the port. In this way, a desired location of additional program memory is addressed. It is left over to read its content.
- Port P0 pins are configured as inputs, the PSEN pin is activated and the microcontroller reads from memory chip.

Similar occurs when it is necessary to read location from external RAM. Addressing is performed in the same way, while read and write are performed via signals appearing on the control outputs RD (is short for read) or WR (is short for write).

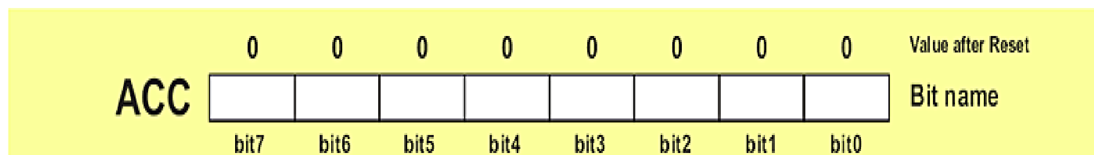
8051 REGISTERS

SFR (SPECIAL FUNCTION REGISTERS)

SFR can be seen as a sort of control panel for managing and monitoring the microcontroller. Every register and each of the belonging bits has its name, specified address in RAM and strictly defined role (e.g. controlling the timer, interrupt, serial connection, etc). Although there are 128 available memory slots for allocating SFR registers.. The rest has been left open intentionally to allow future upgrades while retaining the compatibility with earlier models. This fact makes possible to use programs developed for obsolete models long ago.

NAME	FUNCTION	HEX ADDRES	Bit-Addressable
A	Accumulator	E0	Yes
B	Arithmetic	F0	Yes
DPTR	Data Pointer (2 Bytes)	---	---
DPH	Addressing external memory	83	No
DPL	Addressing external memory	82	No
IE	Interrupt enable control	A8	Yes
IP	Interrupt priority	B8	Yes
P0	Input/output port latch	80	Yes
P1	Input/output port latch	90	Yes
P2	Input/output port latch	A0	Yes
P3	Input/output port latch	B0	Yes
PCON	Power control	87	No
PSW	Program status	D0	Yes
SCON	Serial port control	98	Yes
SBUF	Serial port data buffer	99	No
SP	Stack pointer	81	No
TMOD	Timer/counter mode control	89	Yes
TCON	Timer/counter control	88	Yes
TL0	Timer 0 low byte	8A	No
TH0	Timer 0 high byte	8B	No
TL1	Timer 1 low byte	8C	No
TH1	Timer 1 high byte	8D	No

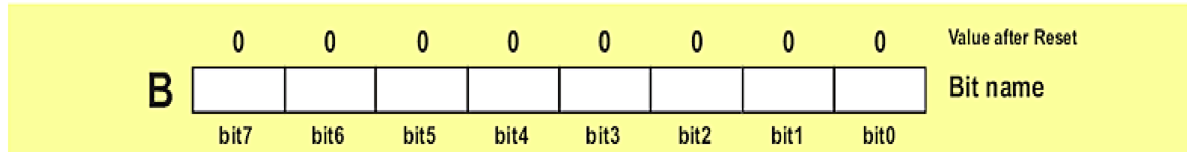
A REGISTER (ACCUMULATOR)



A register is a general-purpose register used for storing intermediate results obtained during operation. Prior to executing an instruction upon any number or operand it is necessary to store it in the accumulator first. All results obtained from arithmetical operations performed by the ALU are stored in the accumulator. Data to be moved from one register to another must go through the accumulator. In other words, the A register is the most commonly used register and it is impossible to imagine a microcontroller without it. More than half instructions used by the 8051 microcontroller use somehow the accumulator.

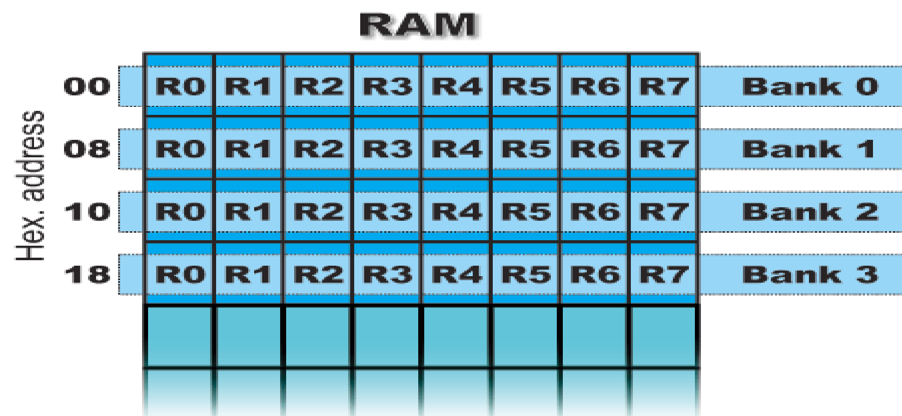
B REGISTER

Multiplication and division can be performed only upon numbers stored in the A and B registers. All other instructions in the program can use this register as a spare accumulator (A).
Instructions of multiplication and division can be applied only to operands located in registers A and B. Other instructions can use this register as a secondary accumulator (A).



During the process of writing a program, each register is called by its name so that their exact addresses are not of importance for the user. During compilation, their names will be automatically replaced by appropriate addresses.

R REGISTERS (R0-R7)



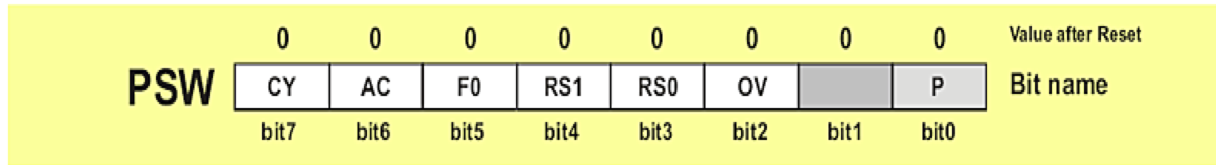
This is a common name for 8 general-purpose registers (R0, R1, R2 ...R7). Even though they are not true SFRs, they deserve to be discussed here because of their purpose. They occupy 4 banks within RAM. Similar to the accumulator, they are used for temporary storing variables and intermediate results during operation. Which one of these banks is to be active depends on two bits of the PSW Register. Active bank is a bank the registers of which are currently used.

The following example best illustrates the purpose of these registers. Suppose it is necessary to perform some arithmetical operations upon numbers previously stored in the R registers: (R1+R2) - (R3+R4). Obviously, a register for temporary storing results of addition is needed. This is how it looks in the program:

```
MOV A,R3; Means: move number from R3 into accumulator

ADD A,R4; Means: add number from R4 to accumulator (result remains in accumulator)

MOV R5,A; Means: temporarily move the result from accumulator into R5

MOV A,R1; Means: move number from R1 to accumulator

ADD A,R2; Means: add number from R2 to accumulator
```

```
SUBB A,R5; Means: subtract number from R5 (there are R3+R4)
```

PROGRAM STATUS WORD (PSW) REGISTER



PSW register is one of the most important SFRs. It contains several status bits that reflect the current state of the CPU. Besides, this register contains Carry bit, Auxiliary Carry, two register bank select bits, Overflow flag, parity bit and user-definable status flag.

P - Parity bit. If a number stored in the accumulator is even then this bit will be automatically set (1), otherwise it will be cleared (0). It is mainly used during data transmit and receive via serial communication.

- Bit 1. This bit is intended to be used in the future versions of microcontrollers.

OV Overflow occurs when the result of an arithmetical operation is larger than 255 and cannot be stored in one register. Overflow condition causes the OV bit to be set (1). Otherwise, it will be cleared (0).

RS0, RS1 - Register bank select bits. These two bits are used to select one of four register banks of RAM. By setting and clearing these bits, registers R0-R7 are stored in one of four banks of RAM.

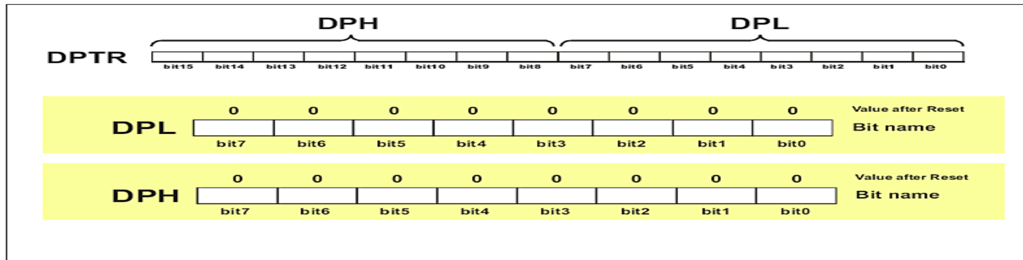
RS1	RS2	SPACE IN RAM
0	0	Bank0 00h-07h
0	1	Bank1 08h-0Fh
1	0	Bank2 10h-17h
1	1	Bank3 18h-1Fh

F0 - Flag 0. This is a general-purpose bit available for use.
AC - Auxiliary Carry Flag is used for BCD operations only.
CY - Carry Flag is the (ninth) auxiliary bit used for all arithmetical operations and shift instructions.

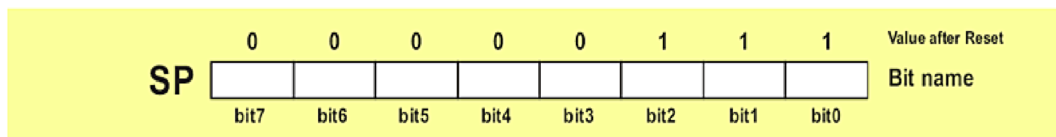
DATA POINTER REGISTER (DPTR)

DPTR register is not a true one because it doesn't physically exist. It consists of two separate registers: DPH (Data Pointer High) and (Data Pointer Low). For this reason it may be treated as a

16-bit register or as two independent 8-bit registers. Their 16 bits are primarly used for external memory addressing. Besides, the DPTR Register is usually used for storing data and intermediate results.

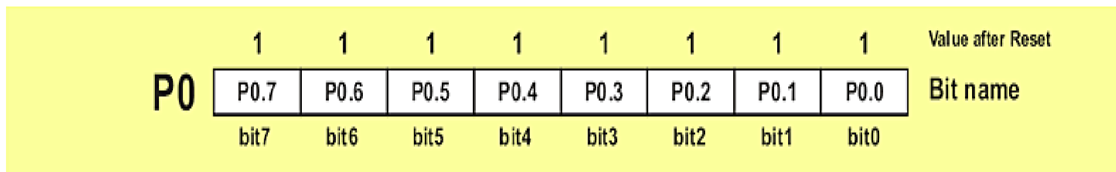


STACK POINTER (SP) REGISTER



A value stored in the Stack Pointer points to the first free stack address and permits stack availability. Stack pushes increment the value in the Stack Pointer by 1. Likewise, stack pops decrement its value by 1. Upon any reset and power-on, the value 7 is stored in the Stack Pointer, which means that the space of RAM reserved for the stack starts at this location. If another value is written to this register, the entire Stack is moved to the new memory location.

P0, P1, P2, P3 - INPUT/OUTPUT REGISTERS



If neither external memory nor serial communication system are used then 4 ports within total of 32 input/output pins are available for connection to peripheral environment. Each bit within these ports affects the state and performance of appropriate pin of the microcontroller. Thus, bit logic state is reflected on appropriate pin as a voltage (0 or 5 V) and vice versa, voltage on a pin reflects the state of appropriate port bit.

As mentioned, port bit state affects performance of port pins, i.e. whether they will be configured as inputs or outputs. If a bit is cleared (0), the appropriate pin will be configured as an output, while if it is set (1), the appropriate pin will be configured as an input. Upon reset and power-on, all port bits are set (1), which means that all appropriate pins will be configured as inputs.

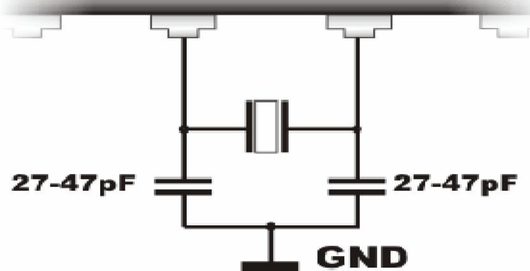
I/O ports are directly connected to the microcontroller pins. Accordingly, logic state of these registers can be checked by voltmeter and vice versa, voltage on the pins can be checked by inspecting their bits.

8051 OSCILLATOR AND CLOCK

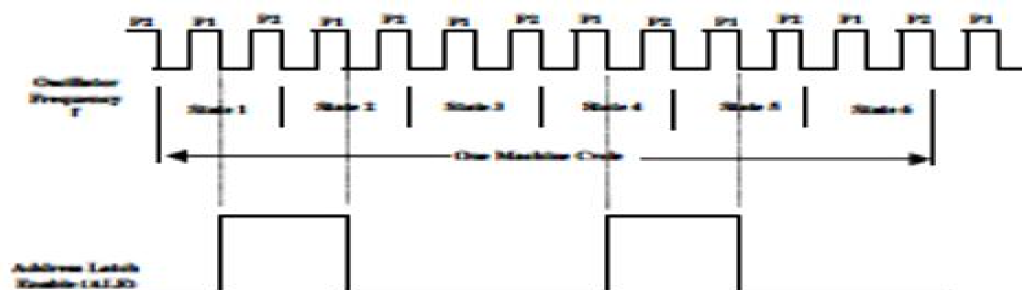
The manufacturers make available 8051 designs that can run at specified maximum and minimum frequencies, typically 1 megahertz to 16 megahertz. Minimum frequencies imply that some internal memories are dynamic and must always operate above a minimum frequency or data will be lost. The oscillator formed by the crystal, capacitors and on – chip inverter generates a pulse train at the frequency of the crystal. The time to execute the instruction is found by using the expression,

T (inst) = (C * 12) / (crystal frequency)

Presently PHILIPS 8051 flash microcontrollers are working with the double speed of normal 8051 chip (40MHz operation) and\ CYGNAL 8051 series working with a speed of 100 MIPS.

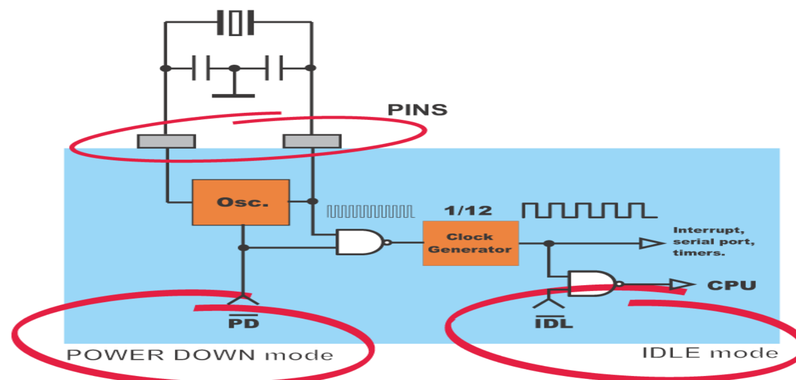


It must be noted that there are various speed of the 8051 families. Speed refers to the maximum oscillator frequency connected to XTAL. For example, a 12 MHz chip must be connected to a crystal with 12MHz frequency or less. Likewise a 20KHz microcontroller requires a crystal frequency of no more than 20MHz. When the 8051 is connected to a crystal oscillator and is powered up, we can observe the frequency on the XTAL2 pins using the oscilloscope.



8051 MICROCONTROLLER POWER CONSUMPTION CONTROL

Generally speaking, the microcontroller is inactive for the most part and just waits for some external signal in order to takes its role in a show. (This can cause some problems in case batteries are used for power supply. In extreme cases, the only solution is to set the whole electronics in sleep mode in order to minimize consumption. A typical example is a TV remote controller: it can be out of use for months but when used again it takes less than a second to send a command to TV receiver.) The AT89S53 uses approximately 25mA for regular operation, which doesn't make it a pover-saving microcontroller. Anyway, it doesn’t have to be always like that, it can easily switch the operating mode in order to reduce its total consumption to approximately 40uA. Actually, there are two power-saving modes of operation:Idle and Power Down.



IDLE MODE

Upon the IDL bit of the PCON register is set, the microcontroller turns off the greatest power consumer- CPU unit while peripheral units such as serial port, timers and interrupt system continue operating normally consuming 6.5mA. In Idle mode, the state of all registers and I/O ports remains unchanged.

In order to exit the Idle mode and make the microcontroller operate normally, it is necessary to enable and execute any interrupt or reset. It will cause the IDL bit to be automatically cleared and the program resumes operation from instruction having set the IDL bit. It is recommended that first three instructions to execute now are NOP instructions. They don't perform any operation but provide some time for the microcontroller to stabilize and prevents undesired changes on the I/O ports.

POWER DOWN MODE

By setting the PD bit of the PCON register from within the program, the microcontroller is set to Power down mode, thus turning off its internal oscillator and reduces power consumption enormously. The microcontroller can operate using only 2V power supply in power- down mode,

while a total power consumption is less than 40uA. The only way to get the microcontroller back to normal mode is by reset.

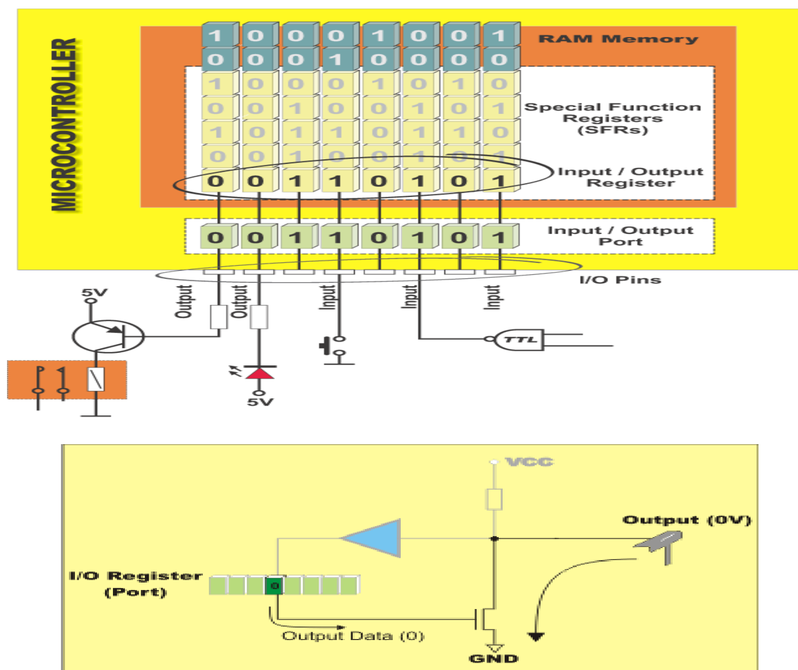
While the microcontroller is in Power Down mode, the state of all SFR registers and I/O ports remains unchanged. By setting it back into the normal mode, the contents of the SFR register is lost, but the content of internal RAM is saved. Reset signal must be long enough, approximately 10mS, to enable stable operation of the quartz oscillator.

INPUT/OUTPUT PORTS (I/O PORTS)

All 8051 microcontrollers have 4 I/O ports each comprising 8 bits which can be configured as inputs or outputs. Accordingly, in total of 32 input/output pins enabling the microcontroller to be connected to peripheral devices are available for use.

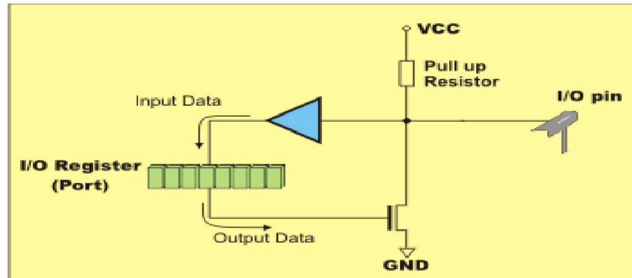
Pin configuration, i.e. whether it is to be configured as an input (1) or an output (0), depends on its logic state. In order to configure a microcontroller pin as an output, it is necessary to apply a logic zero (0) to appropriate I/O port bit. In this case, voltage level on appropriate pin will be 0.

Similarly, in order to configure a microcontroller pin as an input, it is necessary to apply a logic one (1) to appropriate port. In this case, voltage level on appropriate pin will be 5V (as is the case with any TTL input). This may seem confusing but it becomes clear after studying simple electronic circuits connected to an I/O pin.



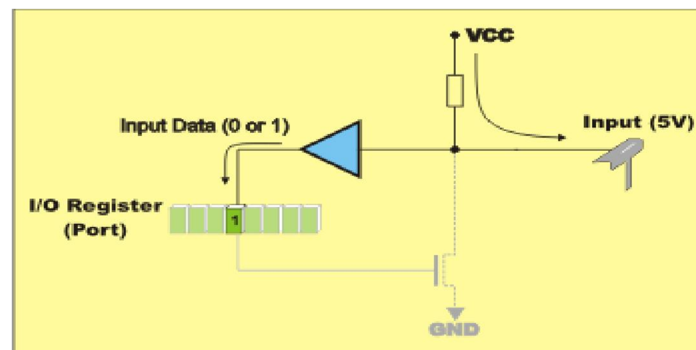
INPUT/OUTPUT(I/O)PIN

Figure above illustrates a simplified schematic of all circuits within the microcontroler connected to one of its pins. It refers to all the pins except those of the P0 port which do not have pull-up resistors built-in.



OUTPUT PIN

A logic zero (0) is applied to a bit of the P register. The output FE transistor is turned on, thus connecting the appropriate pin to ground.



INPUT PIN

A logic one (1) is applied to a bit of the P register. The output FE transistor is turned off and the appropriate pin remains connected to the power supply voltage over a pull-up resistor of high resistance.

Logic state (voltage) of any pin can be changed or read at any moment. A logic zero (0) and logic one (1) are not equal. A logic one (0) represents a short circuit to ground. Such a pin acts as an output.

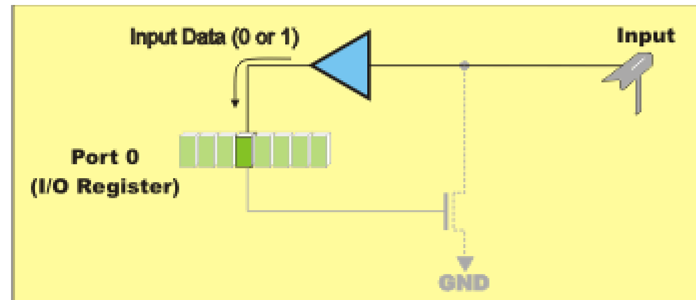
A logic one (1) is "loosely" connected to the power supply voltage over a resistor of high resistance. Since this voltage can be easily "reduced" by an external signal, such a pin acts as an input.

PORT 0

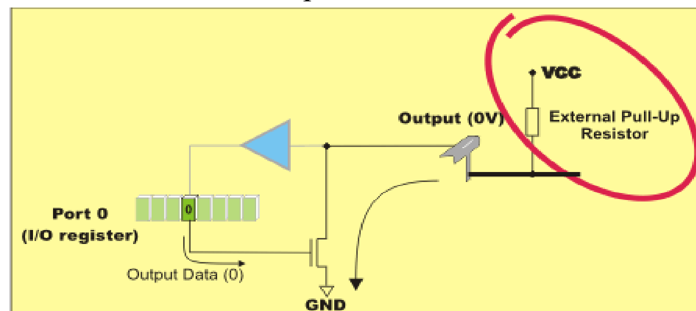
The P0 port is characterized by two functions. If external memory is used then the lower address byte (addresses A0-A7) is applied on it.

Otherwise, all bits of this port are configured as inputs/outputs.

The other function is expressed when it is configured as an output.
Unlike other ports consisting of pins with built-in pull-up resistor connected by its end to 5 V power supply, pins of this port have this resistor left out. This apparently small difference has its consequences:



If any pin of this port is configured as an input then it acts as if it "floats". Such an input has unlimited input resistance and indetermined potential.



When the pin is configured as an output, it acts as an "open drain".
By applying logic 0 to a port bit, the appropriate pin will be connected to ground (0V).
By applying logic 1, the external output will keep on "floating".
In order to apply logic 1 (5V) on this output pin, it is necessary to built in an external pull-up resistor.

Only in case P0 is used for addressing external memory, the microcontroller will provide internal power supply source in order to supply its pins with logic one. There is no need to add external pull-up resistors.

PORT 1

P1 is a true I/O port, because it doesn't have any alternative functions as is the case with P0, but can be cofigured as general I/O only. It has a pull-up resistor built-in and is completely compatible with TTL circuits.

PORT 2

P2 acts similarly to P0 when external memory is used. Pins of this port occupy addresses intended for external memory chip. This time it is about the higher address byte with addresses A8-A15. When no memory is added, this port can be used as a general input/output port showing features similar to P1.

PORT 3

All port pins can be used as general I/O, but they also have an alternative function. In order to use these alternative functions, a logic one (1) must be applied to appropriate bit of the P3 register. In tems of hardware, this port is similar to P0, with the difference that its pins have a pull-up resistor built-in.

PIN'S CURRENT LIMITATIONS

When configured as outputs (logic zero (0)), single port pins can receive a current of 10mA. If all 8 bits of a port are active, a total current must be limited to 15mA (port P0: 26mA). If all ports (32 bits) are active, total maximum current must be limited to 71mA. When these pins are configured as inputs (logic 1), built-in pull-up resistors provide very weak current, but strong enough to activate up to 4 TTL inputs of LS series.

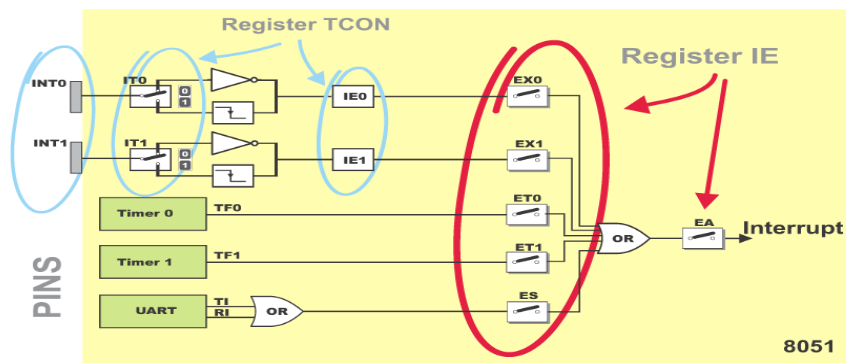
8051 *MICROCONTROLLER INTERRUPTS*

INTERRUPT SERVICE ROUTINE

For every interrupt, there must be an interrupt service routine (ISR). When an interrupt is invoked, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its ISR.

Interrupt	ROM location (HEX)	Pin
Reset	0000	9
External hardware interrupt 0 (INT 0)	0003	
Timer 0 interrupt (TF 0)	000B	P3.2 (12)
External hardware interrupt 1 (INT 1)	0013	
Timer 1 interrupt (TF 1)	001B	
Serial COM interrupt (R1 and T1)	0023	P3.3 (13)

There are five interrupt sources for the 8051, which means that they can recognize 5 different events that can interrupt regular program execution. Each interrupt can be enabled or disabled by setting bits of the IE register. Likewise, the whole interrupt system can be disabled by clearing the EA bit of the same register. Refer to figure below.



Now, it is necessary to explain a few details referring to external interrupts- INT0 and INT1. If the IT0 and IT1 bits of the TCON register are set, an interrupt will be generated on high to low transition, i.e. on the falling pulse edge (only in that moment). If these bits are cleared, an interrupt will be continuously executed as far as the pins are held low.

IE Register (Interrupt Enable)

IE	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
Value after Reset	0	X	0	0	0	0	0	0
Bit name	EA		ET2	ES	ET1	EX1	ET0	EX0

EA - global interrupt enable/disable:

0 - disables all interrupt requests.

1 - enables all individual interrupt requests.

ES - enables or disables serial interrupt:

0 - UART system cannot generate an interrupt.

1 - UART system enables an interrupt.

ET1 - bit enables or disables Timer 1 interrupt:

0 - Timer 1 cannot generate an interrupt.

1 - Timer 1 enables an interrupt.

EX1 - bit enables or disables external 1 interrupt:

0 - change of the pin INT0 logic state cannot generate an interrupt.

1 - enables an external interrupt on the pin INT0 state change.

ET0 - bit enables or disables timer 0 interrupt:

0 - Timer 0 cannot generate an interrupt.

1 - enables timer 0 interrupt.

EX0 - bit enables or disables external 0 interrupt:

0 - change of the INT1 pin logic state cannot generate an interrupt.

1 - enables an external interrupt on the pin INT1 state change.

INTERRUPT PRIORITIES

The IP Register (Interrupt Priority Register) specifies which one of existing interrupt sources have higher and which one has lower priority. Interrupt priority is usually specified at the beginning of the program. According to that, there are several possibilities:

1. If an interrupt of higher priority arrives while an interrupt is in progress, it will be immediately stopped and the higher priority interrupt will be executed first.
2. If two interrupt requests, at different priority levels, arrive at the same time then the higher priority interrupt is serviced first.

3. If the both interrupt requests, at the same priority level, occur one after another, the one which came later has to wait until routine being in progress ends.
4. If two interrupt requests of equal priority arrive at the same time then the interrupt to be serviced is selected according to the following priority list:

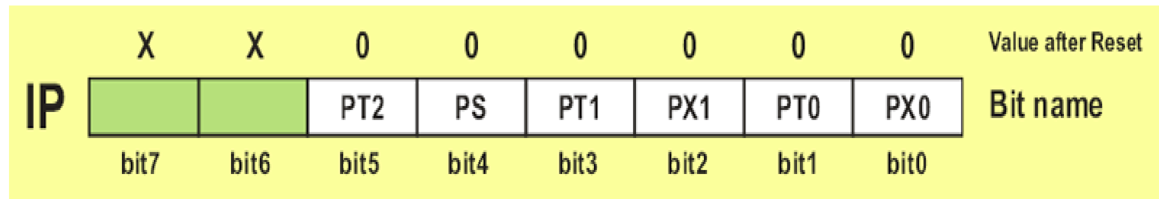
I. External interrupt INT0
II. Timer 0 interrupt
III. External Interrupt INT1
IV. Timer 1 interrupt
V. Serial Communication Interrupt

HIGHEST TO LOWEST PRIORITY

1. External Interrupt 0 (INT 0) Highest
2. Timer Interrupt 0 (TF 0)
3. External Interrupt 1 (INT 1)
4. Timer Interrupt 1 TF 1)
5. Serial Communication RI + TI) Lowest

IP Register (Interrupt Priority)

The IP register bits specify the priority level of each interrupt (high or low priority).



PS - Serial Port Interrupt priority bit

1. Priority 0
2. Priority 1

PT1 - Timer 1 interrupt priority

1. Priority 0
2. Priority 1

PX1 - External Interrupt INT1 priority

1. Priority 0
2. Priority 1

PT0 - Timer 0 Interrupt Priority

1. Priority 0
2. Priority 1

PX0 - External Interrupt INT0 Priority

1. Priority 0
2. Priority 1

Upon reset, the IP register contains all 0s, making the priority sequence based on the priority

table. To give a higher priority to any of the interrupts, we make the corresponding bit in the IP register high. When two or more interrupt bits in the IP register are set to high, those interrupts having higher priority than others are serviced according to the sequence of the priority table.Upon receiving an interrupt request, following scenario takes place:

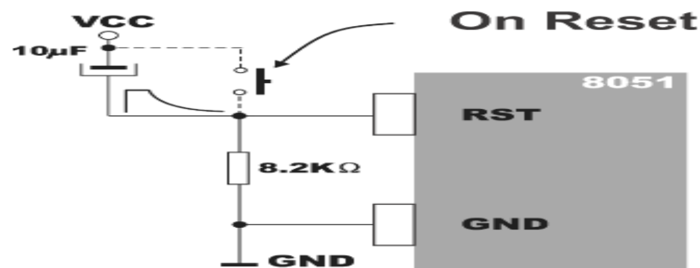
1. Current instruction is executed first.
2. Address of the instruction that would be executed next if there was no interrupt request is put away to stack.
3. Depending on the interrupt in question, program counter will take value of one of possible 6 vectors (addresses) according to the table below

Interrupt source	Vector (address in hex)
IE0	3H
TF0	0BH
IE1	13H
TF1	1BH
RI, TI, SPIF	23H
TF2, EXF2	2BH

These addresses store appropriate subroutines processing interrupts. Instead of them, there are usually jump instructions specifying locations on which these subroutines reside.
When an interrupt routine is executed, the address of the next instruction to execute is poped from the stack to the program counter and interrupted program resumes operation from where it left off. From the moment an interrupt is enabled, the microcontroller is on alert all the time. When an interrupt request arrives, the program execution is stopped, electronics recognizes the source and the program "jumps" to the appropriate address (see the table above). This address usually stores a jump instruction specifying the start of appropriate subroutine. Upon its execution, the program resumes operation from where it left off.

RESET

Reset occurs when the RS pin is supplied with a positive pulse in duration of at least 2 machine cycles (24 clock cycles of crystal oscillator). After that, the microcontroller generates an internal reset signal which clears all SFRs, except SBUF registers, Stack Pointer and ports (the state of the first two ports is not defined, while FF value is written to the ports configuring all their pins as inputs). Depending on surrounding and purpose of device, the RS pin is usually connected to a power-on reset push button or circuit or to both of them. Figure below illustrates one of the simplest circuit providing safe power-on reset.



Basically, everything is very simple: after turning the power on, electrical capacitor is being charged for several milliseconds through a resistor connected to the ground. The pin is driven high during this process. When the capacitor is charged, power supply voltage is already stable and the pin remains connected to the ground, thus providing normal operation of the microcontroller. Pressing the reset button causes the capacitor to be temporarily discharged and the microcontroller is reset. When released, the whole process is repeated.

Microcontrollers normally operate at very high speed. The use of 12 Mhz quartz crystal enables 1.000.000 instructions to be executed per second. Basically, there is no need for higher operating rate. In case it is needed, it is easy to built in a crystal for high frequency. The problem arises when it is necessary to slow down the operation of the microcontroller. For example during testing in real environment when it is necessary to execute several instructions step by step in order to check I/O pins' logic state.

Interrupt system of the 8051 microcontroller practically stops operation of the microcontroller and enables instructions to be executed one after another by pressing the button. Two interrupt features enable that: Interrupt request is ignored if an interrupt of the same priority level is in progress. Upon interrupt routine execution, a new interrupt is not executed until at least one instruction from the main program is executed. In order to use this in practice, the following steps should be done: External interrupt sensitive to the signal level should be enabled (for example INT0).
Three following instructions should be inserted into the program (at the 03hex. address):

```
JNB   P3.2$     ←  Means: wait here until the pin P3.2 (INT0) is set to "1".
JB    P3.2$     ←  Means: wait here until the pin P3.2 (INT0) is set to "0".
RETI            ←  Means: go back to the main program
```

As soon as the P3.2 pin is cleared (for example, by pressing the button), the microcontroller will stop program execution and jump to the 03hex address will be executed. This address stores a short interrupt routine consisting of 3 instructions.

The first instruction is executed until the push button is realised (logic one (1) on the P3.2 pin). The second instruction is executed until the push button is pressed again. Immediately after that, the RETI instruction is executed and the processor resumes operation of the main program. Upon execution of any program instruction, the interrupt INT0 is generated and the whole procedure is repeated (push button is still pressed). In other words, one button press - one instruction.

8051ADDRESSING MODES & INSTRUCTION SET

ADDRESSING MODES

The CPU can access data in various ways. The data could be in a register, or in memory, or to be provided as an immediate value. These various ways of accessing data are called addressing modes. The various addressing modes are determined when it is designed and therefore cannot be changed by the programmer.

8051 *ADDRESSING MODES*

The 8051 provide a total of five distinct addressing modes. They are:
1. Immediate
2. Direct
3. Register
4. Register indirect
5. Indexed
We can use direct or register indirect addressing modes to access data stored either in RAM or registers of the 8051.

i. IMMEDIATE ADDRESSING MODE

Immediate addressing is so-named because the value to be stored in memory immediately follows the operation code in memory. That is to say, the instruction itself dictates what value will be stored in memory. Immediate data has to be preceded by "#" sign.
For example, the instruction: **MOV A, #20H**
This instruction uses Immediate Addressing because the Accumulator will be loaded with the value that immediately follows; in this case 20 (hexadecimal). Immediate addressing is very fast since the value to be loaded is included in the instruction. However, since the value to be loaded is fixed at compile-time it is not very flexible.

ii. DIRECT ADDRESSING

In Direct addressing the value to be stored in memory is obtained by directly retrieving it from another memory location.
For example: **MOV A, 30H**
This instruction will read the data from the Internal RAM address 30 (hexadecimal) and store it in the Accumulator.
Also, it is important to note that when using direct addressing any instruction, which refers to an address between 00h, and 7Fh is referring to Internal Memory. Any instruction, which refers to an address between 80h and FFh, is referring to the SFR control registers. Direct addressing is the only method of accessing the special function registers. The lower 128 bytes

of internal RAM are also directly addressable.

iii. REGISTER ADDRESSING

Register addressing accesses the eight working registers (R0 - R7) of the selected register bank. This instruction put the operand in a register and manipulates it by referring to the register (by name) in the instruction & in this type of instruction the source & destination registers must match in size.

Example: **Mov A, R0 ;A ? contents (R0)**
Mov R2, A ;R2 ? contents (A)

ADD A, R1 ;A ? contents (A) + contents (R1)

The least significant bit of the instruction op-code indicates which register is to be used. ACC, B, DPTR and CY, can also be addressed as registers.

iv. REGISTER INDIRECT ADDRESSING

Register indirect addressing is a very powerful addressing mode, which in many cases provides an exceptional level of flexibility. Indirect addressing is also the only way to access the extra 128 bytes of Internal RAM found on an 8052.
Indirect addressing appears as follows:
Example: **MOV A, @R0**
This instruction causes the 8051 to analyze the value of the R0 register. The 8051 will then load the accumulator with the value from Internal RAM, which is found at the address indicated by R0.
For example, let's say R0 holds the value 40H and Internal RAM address 40H holds the value 67H. When the above instruction is executed the 8051 will check the value of R0. Since R0 holds 40H the 8051 will get the value out of Internal RAM address 40H (which holds 67H) and store it in the Accumulator. Thus, the Accumulator ends up holding 67H.
Indirect addressing always refers to Internal RAM; it never refers to an SFR. Indirect addressing only can access the upper half of the internal RAM. Access to the full 64 Kbytes of external data memory address space is accomplished by using the 16-bit data pointer. Execution of PUSH and POP instructions also uses register indirect addressing. The stack may reside anywhere in the internal RAM.

v. INDEX ADDRESSING

Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM space of the 8051. Indexed addressing use a register for storing the pointer and another register for an offset. The Effective address is the sum of base & offset.

Example:
MOVC A, @A+DPTR ; A ? ext_code_mem [(A + DPTR)]
MOVC A, @A+PC ; A ? ext_code_mem [(A + PC)]

8051 *INSTRUCTION SET*
The 8051, 8-bit microcontroller family instruction set includes 111 instructions, 49 of which are single-byte, 45 two-byte and 17 three-byte instructions. The instruction op-code format consists of a function mnemonic followed by a destination & source operand field. The instruction set is divided into four functional groups:

1. **Data transfer**
2. **Arithmetic**
3. **Logic**
4. **Control transfer**

i. DATA TRANSFER INSTRUCTIONS
Data transfer operations are divided into three classes:

I. General - purpose
II. Accumulator-specific
III. Address-object

None of these operations affects the PSW flag settings except a POP or MOV directly to the PSW.
Examples
MOV A, #45 - Immediate Addressing Mode
MOV A, R1 - Register Addressing Mode
MOV 45h,A - Direct Addressing Mode
MOV @R1, 32h - Indirect Addressing Mode

ii. *ARITHMETIC INSTRUCTIONS*
The MCS-51 family microcontrollers have four basic mathematical operations. Only 8-bit operations using unsigned arithmetic are supported directly. The overflow flag, however, permits the addition and subtraction operation to serve for both unsigned and signed binary integers. Arithmetic can also be performed directly on packed BCD representations.
Examples
ADD A, #84 - Immediate Addressing Mode
SUBB A, R2 - Register Addressing Mode
ADD 73h,A - Direct Addressing Mode
ADDC@R1, 25h - Indirect Addressing Mode

iii. LOGIC INSTRUCTIONS
The MCS-51 family microcontrollers perform basic logic operations on both bit and byte

Operands

BIT LEVEL (SINGLE OPERAND) OPERATIONS

In 8051 internal RAM and SFRs can be addressed by the address of each bit within a byte. This bit addressing is very convenient when we wish to alter a single bit of a byte. The ability to operate on individual bits creates the need for an area of RAM that contains data addresses that hold a single bit. The bit addresses are numbered from 00H to 7FH to represent the 128d bit addresses that exist from byte addresses 20H to 2FH.

I. **CLR** sets A or any directly addressable bit to zero (0).
II. **SETB** sets any directly bit-addressable bit to one (1).
III. **CPL** is used to complement the contents of the A register without affecting any flag,
IV. or any directly addressable bit location.
V. RL, RLC, RR, RRC, SWAP are the five operations that can be performed on A. RL,

rotate left, RR, rotate right, RLC, rotate left through carry, **RRC**, rotate right through carry, and SWAP, rotate left four. For **RLC** and **RRC** the **CY** flag become equal to the last bit rotated out. **SWAP** rotates A left four places to exchange bits 3 through 0 with bits 7 through 4.

BYTE LEVEL (TWO-OPERAND) OPERATIONS

I. **ANL** performs bit wise logical AND of two operands (for both bit and byte operands) and returns the result to the location of the first operand.
II. **ORL** performs bit wise logical OR of two source operands (for both bit and byte operands) and returns the result to the location of the first operand.
III. **XRL** performs logical Exclusive OR of two source operands (byte operands) and returns the result to the location of the first operand.

Example

ANL A, #45h - Immediate Addressing Mode
ORL A, R2 - Register Addressing Mode
XRL 52h, A - Direct Addressing Mode
ANL @R3, 65h - Indirect Addressing Mode

iv. CONTROL TRANSFER INSTRUCTIONS

There are three classes of control transfer operations: unconditional calls, returns, jumps, conditional jumps, and interrupts. All control transfer operations, some upon a specific condition, cause the program execution to continue a non-sequential location in program memory.

Example
CJNE A,#22H,loop - Immediate Addressing Mode
DJNZ R1,loop - Register Addressing Mode
DJNZ 30H,loop - Direct Addressing Mode
JMP @A+DPTR - Indirect Addressing Mode

NOTES ON DATA ADDRESSING MODES

Rn - Working register R0-R7
direct - 128 internal RAM locations, any l/O port,control or status register
@Ri - Indirect internal or external RAM location addressed by register R0 or R1
#data - 8-bit constant included in instruction
#data 16 - 16-bit constant included as bytes 2 and 3 of instruction
Bit - 128 software flags, any bit-addressable l/O pin, control or status bit

A - Accumulator
NOTES ON PROGRAM ADDRESSING MODES:
addr16 - Destination address for LCALL and LJMP may be anywhere within the 64-Kbyte program memory address space.
addr11 - Destination address for ACALL and AJMP will be within the same 2-Kbyte page of program memory as the first byte of the following instruction.
rel - SJMP and all conditional jumps include an 8-bit offset byte. Range is + **127/– 128** bytes relative to the first byte of the following instruction.

Instruction Set of 8051 µc

The process of writing program for the microcontroller mainly consists of giving instructions (commands) in the specific order in which they should be executed in order to carry out a specific task.

Types of instructions

Depending on operation they perform, all instructions are divided in several groups:

- Arithmetic Instructions
- Branch Instructions
- Data Transfer Instructions
- Logic Instructions
- Bit-oriented Instructions

The first part of each instruction, called MNEMONIC refers to the operation an instruction performs (copy, addition, logic operation etc.). Mnemonics are abbreviations of the name of operation being executed. The other part of instruction, called OPERAND is separated from

mnemonic by at least one whitespace and defines data being processed by instructions. Some of the instructions have no operand, while some of them have one, two or three. If there is more than one operand in an instruction, they are separated by a comma.

Arithmetic instructions

Arithmetic instructions perform several basic operations such as addition, subtraction, division, multiplication etc. After execution, the result is stored in the first operand.

ARITHMETIC INSTRUCTIONS			
Mnemonic	**Description**	**Byte**	**Cycle**
ADD A,Rn	Adds the register to the accumulator	1	1
ADD A,direct	Adds the direct byte to the accumulator	2	2
ADD A,@Ri	Adds the indirect RAM to the accumulator	1	2
ADD A,#data	Adds the immediate data to the accumulator	2	2
ADDC A,Rn	Adds the register to the accumulator with a carry flag	1	1
ADDC A,direct	Adds the direct byte to the accumulator with a carry flag	2	2
ADDC A,@Ri	Adds the indirect RAM to the accumulator with a carry flag	1	2
ADDC A,#data	Adds the immediate data to the accumulator with a carry flag	2	2
SUBB A,Rn	Subtracts the register from the accumulator with a borrow	1	1
SUBB A,direct	Subtracts the direct byte from the accumulator with a borrow	2	2
SUBB A,@Ri	Subtracts the indirect RAM from the accumulator with a borrow	1	2
SUBB A,#data	Subtracts the immediate data from the accumulator with a borrow	2	2
INC A	Increments the accumulator by 1	1	1
INC Rn	Increments the register by 1	1	2

INC Rx	Increments the direct byte by 1	2	3
INC @Ri	Increments the indirect RAM by 1	1	3
DEC A	Decrements the accumulator by 1	1	1
DEC Rn	Decrements the register by 1	1	1
DEC Rx	Decrements the direct byte by 1	1	2
DEC @Ri	Decrements the indirect RAM by 1	2	3
INC DPTR	Increments the Data Pointer by 1	1	3
MUL AB	Multiplies A and B	1	5
DIV AB	Divides A by B	1	5
DA A	Decimal adjustment of the accumulator according to BCD code	1	1

Branch Instructions

There are two kinds of branch instructions:
Unconditional jump instructions: upon their execution a jump to a new location from where the program continues execution is executed.
Conditional jump instructions: a jump to a new program location is executed only if a specified condition is met. Otherwise, the program normally proceeds with the next instruction.

BRANCH INSTRUCTIONS			
Mnemonic	**Description**	**Byte**	**Cycle**
ACALL addr11	Absolute subroutine call	2	6
LCALL addr16	Long subroutine call	3	6
RET	Returns from subroutine	1	4
RETI	Returns from interrupt subroutine	1	4
AJMP addr11	Absolute jump	2	3

LJMP addr16	Long jump	3	4
SJMP rel	Short jump (from –128 to +127 locations relative to the following instruction)	2	3
JC rel	Jump if carry flag is set. Short jump.	2	3
JNC rel	Jump if carry flag is not set. Short jump.	2	3
JB bit,rel	Jump if direct bit is set. Short jump.	3	4
JBC bit,rel	Jump if direct bit is set and clears bit. Short jump.	3	4
JMP @A+DPTR	Jump indirect relative to the DPTR	1	2
JZ rel	Jump if the accumulator is zero. Short jump.	2	3
JNZ rel	Jump if the accumulator is not zero. Short jump.	2	3
CJNE A,direct,rel	Compares direct byte to the accumulator and jumps if not equal. Short jump.	3	4
CJNE A,#data,rel	Compares immediate data to the accumulator and jumps if not equal. Short jump.	3	4
CJNE Rn,#data,rel	Compares immediate data to the register and jumps if not equal. Short jump.	3	4
CJNE @Ri,#data,rel	Compares immediate data to indirect register and jumps if not equal. Short jump.	3	4
DJNZ Rn,rel	Decrements register and jumps if not 0. Short jump.	2	3
DJNZ Rx,rel	Decrements direct byte and jump if not 0. Short jump.	3	4
NOP	No operation	1	1

Data Transfer Instructions

Data transfer instructions move the content of one register to another. The register the content of which is moved remains unchanged. If they have the suffix “X” (MOVX), the data is exchanged with external memory.

DATA TRANSFER INSTRUCTIONS			
Mnemonic	**Description**	**Byte**	**Cycle**

MOV A,Rn	Moves the register to the accumulator	1	1
MOV A,direct	Moves the direct byte to the accumulator	2	2
MOV A,@Ri	Moves the indirect RAM to the accumulator	1	2
MOV A,#data	Moves the immediate data to the accumulator	2	2
MOV Rn,A	Moves the accumulator to the register	1	2
MOV Rn,direct	Moves the direct byte to the register	2	4
MOV Rn,#data	Moves the immediate data to the register	2	2
MOV direct,A	Moves the accumulator to the direct byte	2	3
MOV direct,Rn	Moves the register to the direct byte	2	3
MOV direct,direct	Moves the direct byte to the direct byte	3	4
MOV direct,@Ri	Moves the indirect RAM to the direct byte	2	4
MOV direct,#data	Moves the immediate data to the direct byte	3	3
MOV @Ri,A	Moves the accumulator to the indirect RAM	1	3
MOV @Ri,direct	Moves the direct byte to the indirect RAM	2	5
MOV @Ri,#data	Moves the immediate data to the indirect RAM	2	3
MOV DPTR,#data	Moves a 16-bit data to the data pointer	3	3
MOVC A,@A+DPTR	Moves the code byte relative to the DPTR to the accumulator (address=A+DPTR)	1	3
MOVC A,@A+PC	Moves the code byte relative to the PC to the accumulator (address=A+PC)	1	3
MOVX A,@Ri	Moves the external RAM (8-bit address) to the accumulator	1	3-10
MOVX A,@DPTR	Moves the external RAM (16-bit address) to the accumulator	1	3-10
MOVX @Ri,A	Moves the accumulator to the external RAM (8-bit address)	1	4-11

MOVX @DPTR,A	Moves the accumulator to the external RAM (16-bit address)	1	4-11
PUSH direct	Pushes the direct byte onto the stack	2	4
POP direct	Pops the direct byte from the stack/td>	2	3
XCH A,Rn	Exchanges the register with the accumulator	1	2
XCH A,direct	Exchanges the direct byte with the accumulator	2	3
XCH A,@Ri	Exchanges the indirect RAM with the accumulator	1	3
XCHD A,@Ri	Exchanges the low-order nibble indirect RAM with the accumulator	1	3

Logic Instructions: Logic instructions perform logic operations upon corresponding bits of two registers. After execution, the result is stored in the first operand.

LOGIC INSTRUCTIONS			
Mnemonic	**Description**	**Byte**	**Cycle**
ANL A,Rn	AND register to accumulator	1	1
ANL A,direct	AND direct byte to accumulator	2	2
ANL A,@Ri	AND indirect RAM to accumulator	1	2
ANL A,#data	AND immediate data to accumulator	2	2
ANL direct,A	AND accumulator to direct byte	2	3
ANL direct,#data	AND immediae data to direct register	3	4
ORL A,Rn	OR register to accumulator	1	1
ORL A,direct	OR direct byte to accumulator	2	2
ORL A,@Ri	OR indirect RAM to accumulator	1	2
ORL direct,A	OR accumulator to direct byte	2	3
ORL direct,#data	OR immediate data to direct byte	3	4

XRL A,Rn	Exclusive OR register to accumulator	1	1
XRL A,direct	Exclusive OR direct byte to accumulator	2	2
XRL A,@Ri	Exclusive OR indirect RAM to accumulator	1	2
XRL A,#data	Exclusive OR immediate data to accumulator	2	2
XRL direct,A	Exclusive OR accumulator to direct byte	2	3
XORL direct,#data	Exclusive OR immediate data to direct byte	3	4
CLR A	Clears the accumulator	1	1
CPL A	Complements the accumulator (1=0, 0=1)	1	1
SWAP A	Swaps nibbles within the accumulator	1	1
RL A	Rotates bits in the accumulator left	1	1
RLC A	Rotates bits in the accumulator left through carry	1	1
RR A	Rotates bits in the accumulator right	1	1
RRC A	Rotates bits in the accumulator right through carry	1	1

Bit-oriented Instructions

Similar to logic instructions, bit-oriented instructions perform logic operations. The difference is that these are performed upon single bits.

BIT-ORIENTED INSTRUCTIONS			
Mnemonic	**Description**	**Byte**	**Cycle**
CLR C	Clears the carry flag	1	1
CLR bit	Clears the direct bit	2	3
SETB C	Sets the carry flag	1	1

SETB bit	Sets the direct bit	2	3
CPL C	Complements the carry flag	1	1
CPL bit	Complements the direct bit	2	3
ANL C,bit	AND direct bit to the carry flag	2	2
ANL C,/bit	AND complements of direct bit to the carry flag	2	2
ORL C,bit	OR direct bit to the carry flag	2	2
ORL C,/bit	OR complements of direct bit to the carry flag	2	2
MOV C,bit	Moves the direct bit to the carry flag	2	2
MOV bit,C	Moves the carry flag to the direct bit	2	3

Description of all 8051 instructions

Here is a list of the operands and their meanings:

- **A** -accumulator;

  Rn - is one of working registers (R0-R7) in the currently active RAM memory bank;
- **Direct** - is any 8-bit address register of RAM. It can be any general-purpose register or a SFR (I/O port, control register etc.);
- **@Ri** - is indirect internal or external RAM location addressed by register R0 or R1;
- **#data** - is an 8-bit constant included in instruction (0-255);
- **#data16** - is a 16-bit constant included as bytes 2 and 3 in instruction (0-65535);
- **addr16** - is a 16-bit address. May be anywhere within 64KB of program memory;
- **addr11** - is an 11-bit address. May be within the same 2KB page of program memory as the first byte of the following instruction;
- **rel** - is the address of a close memory location (from -128 to +127 relative to the first byte of the following instruction). On the basis of it, assembler computes the value to add or subtract from the number currently stored in the program counter;
- **bit** - is any bit-addressable I/O pin, control or status bit; and
- **C** - is carry flag of the status register (register PSW).

8051 ASSEMBLY LANGUAGE PROGRAMMING TOOLS

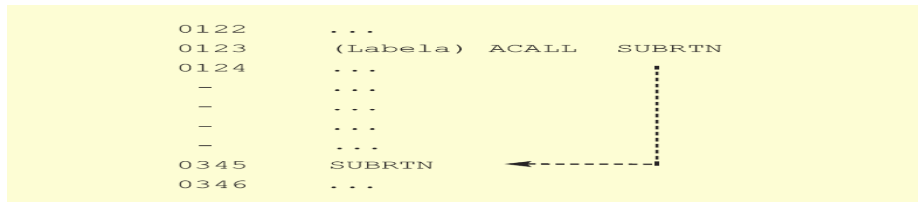
ACALL addr11 - Absolute subroutine call

addr11: Subroutine address

Description: Instruction unconditionally calls a subroutine located at the specified code address. Therefore, the current address and the address of called subroutine must be within the same 2K byte block of the program memory, starting from the first byte of the instruction following ACALL.

Syntax:ACALL[subroutinename]
STATUS register flags: No flags are affected.

EXAMPLE:



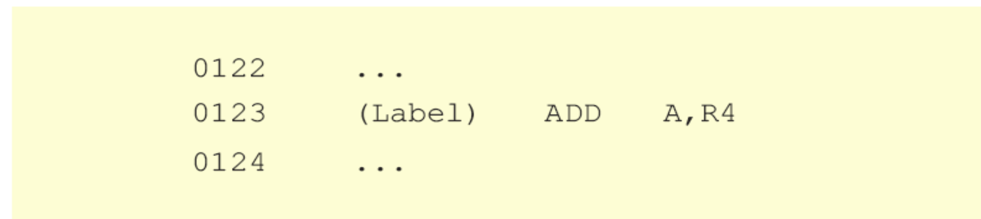
Before execution: PC=0123h
After execution: PC=0345h

ADD A,Rn - Adds the register Rn to the accumulator

A: accumulator, Rn: any R register (R0-R7)

Description: Instruction adds the register Rn (R0-R7) to the accumulator. After addition, the result is stored in the accumulator.

Syntax: ADD A,Rn;
STATUS register flags: C, OV and AC
EXAMPLE:



Before execution: A=2Eh (46 dec.) R4=12h (18 dec.)
After execution: A=40h (64 dec.) R4=12h

ADD A,@Ri - Adds the indirect RAM to the accumulator
A: accumulator, Ri: Register R0 or R1

Description: Instruction adds the indirect RAM to the accumulator. Address of indirect RAM is stored in the Ri register (R0 or R1). After addition, the result is stored in the accumulator.
Syntax: ADD A,@Ri
STATUS register flags: C, OV and AC
EXAMPLE:

```
0122     ...
0123     (Label)      ADD    A,@R0
0124     ...
```

Register address: SUM = 4Fh R0=4Fh
Before execution: A= 16h (22 dec.) SUM= 33h (51 dec.)
After execution : A= 49h (73 dec.)

ADD A,direct - Adds the direct byte to the accumulator
A: accumulator, Direct: Arbitrary register with address 0 - 255 (0 - FFh)
Description: Instruction adds the direct byte to the accumulator. As it is direct addressing, the direct can be any SFR or general-purpose register with address 0-7 Fh. The result is stored in the accumulator.
Syntax: ADD A, register name
STATUS register flags: C, OV and AC
EXAMPLE:

```
0122     ...
0123     (Label)      ADD    A,SUM
0123     ...
```

Before execution: SUM= 33h (51 dec.) A= 16h (22 dec.)
After execution: SUM= 33h (73 dec.) A= 49h (73 dec.)

ADDC A,Rn - Adds the register to the accumulator with a carry flag

A: accumulator, Rn: any R register (R0-R7)
Description: Instruction adds the accumulator with a carry flag and Rn register (R0-R7). After addition, the result is stored in the accumulator.
Syntax: ADDC A,Rn
STATUS register flags: C, OV and AC

EXAMPLE:

```
0122  ...
0123  (Label)     ADDC  A,R0
0124  ...
```

Before execution: A= C3h (195 dec.) R0= AAh (170 dec.) C=1
After execution: A= 6Eh (110 dec.) AC=0, C=1, OV=1

ADD A,#data - Adds the immediate data to the accumulator
A: accumulator, Data: constant within 0-255 (0-FFh)
Description: Instruction adds data (0-255) to the accumulator. After addition, the result is stored in the accumulator.
Syntax: ADD A,#data;
STATUS register flags: C, OV and AC
EXAMPLE:

```
0122  ...
0123  (Label)     ADD   A,#33
0124  ...
```

Before execution: A= 16h (22 dec.)
After execution: A= 49h (73 dec.)

ADDC A,direct - Adds the direct byte to the accumulator with a carry flag
A: accumulator, Direct: arbitrary register with address 0-255 (0-FFh)
Description: Instruction adds the direct byte to the accumulator with a carry flag. As it is direct addressing, the register can be any SFRs or general purpose register with address 0-7Fh (0-127dec.). The result is stored in the accumulator.
Syntax: ADDC A, register address;
STATUS register flags: C, OV and AC
EXAMPLE:

```
0122  ...
0123  (Label)     ADDC  A,TEMP
0124  ...
```

Before execution: A= C3h (195 dec.) TEMP = AAh (170 dec.) C=1
After execution: A= 6Eh (110 dec.) AC=0, C=1, OV=1

ADDC A,@Ri - Adds the indirect RAM to the accumulator with a carry flag
A: accumulator, Ri: Register R0 or R1
Description: Instruction adds the indirect RAM to the accumulator with a carry flag. RAM address is stored in the Ri register (R0 or R1). After addition, the result is stored in the accumulator.
Syntax: ADDC A,@Ri
STATUS register flags: C, OV and AC
EXAMPLE:

```
0122  ...
0123  (Label)     ADDC  A,@R0
0124  ...
```

Register address: SUM = 4Fh R0=4Fh
Before execution: A= C3h (195 dec.) SUM = AAh (170 dec.) C=1
After execution: A= 6Eh (110 dec.) AC=0, C=1, OV=1

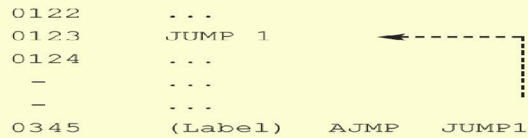
ADDC A,#data - Adds the immediate data to the accumulator with a carry flag
A: accumulator, Data: constant with address 0-255 (0-FFh)
Description: Instruction adds data (0-255) to the accumulator with a carry flag. After addition, the result is stored in the accumulator.
Syntax: ADDC A,#data;
STATUS register flags: C, OV and AC
EXAMPLE:

```
0122  ...
0123  (Label)     ADDC  A,A9
0124  ...
```

Before execution: A= C3h (195 dec.) C=1
After execution: A= 6Dh (109 dec.) AC=0, C=1, OV=1

AJMP addr11 - Absoulte jump
addr11: Jump address
Description: Program continues execution after executing a jump to the specified address. Similar to the ACALL instruction, the jump must be executed within the same 2K byte block of program memory starting from the first byte of the instruction following AJMP.
Syntax: AJMP address (label)
STATUS register flags: No flags are affected

EXAMPLE:



```
0122    ...
0123    JUMP 1
0124    ...
 -      ...
 -      ...
0345    (Label)  AJMP  JUMP1
```

Before execution: PC=0345h SP=07h
After execution: PC=0123h SP=09h

ANL A,Rn - AND register to the accumulator

A: accumulator, Rn: any R register (R0-R7)

Description: Instruction performs logic AND operation between the accumulator and Rn register. The result is stored in the accumulator.

Syntax: ANL A,Rn

STATUS register flags: No flags are affected

EXAMPLE:

```
0222    ...
0223    (Label)   ANL   A,R5
0224    ...
```

Before execution: A= C3h (11000011 Bin.)
R5= 55h (01010101 Bin.)
After execution: A= 41h (01000001 Bin.)

ANL A,direct - AND direct byte to the accumulator

A: accumulator, Direct: arbitrary register with address 0 - 255 (0 - FFh)

Description: Instruction performs logic AND operation between the accumulator and drect register. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh (o-127 dec.). The result is stored in the accumulator.

Syntax: ANL A,direct

STATUS register flags: No flags are affected

EXAMPLE:

```
0222    ...
0223    (Label) ANL   A,MASK
0224    ...
```

Before execution: A= C3h (11000011 Bin.)
MASK= 55h (01010101 Bin.)
After execution: A= 41h (01000001 Bin.)

ANL A,@Ri - AND indirect RAM to the accumulator
A: accumulator, Ri: Register R0 or R1
Description: Instruction performs logic AND operation between the accumulator and register. As it is indirect addressing, the register address is stored in the Ri register (R0 or R1). The result is stored in the accumulator.
Syntax: ANL A,@Ri
STATUS register flags: No flags are affected
EXAMPLE:

```
0222     ...
0223     (Label)   ANL    A,@R0
0224     ...
```

Register address SUM = 4Fh R0=4Fh
Before execution: A= C3h (11000011 Bin.)
R0= 55h (01010101 Bin.)
After execution: A= 41h (01000001 Bin.)

ANL A,#data - AND immediate data to the accumulator
A: accumulator, Data: constant in the range of 0-255 (0-FFh)
Description: Instruction performs logic AND operation between the accumulator and data. The result is stored in the accumulator.
Syntax: ANL A,#data
STATUS register flags: No flags are affected
EXAMPLE:

```
0322     ...
0323     (Labela)    ANL    A,#55
0324     ...
```

Beforeexecution:A=C3h(11000011Bin.)
After execution: A= 41h (01000001 Bin.)

ANL direct,A - AND accumulator to direct byte
Direct: arbitrary register with address 0-255 (0-FFh), A: accumulator
Description: Instruction performs logic AND operation between direct byte and accumulator. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh (0-127 dec.). The result is stored in the direct byte.
Syntax: ANL register address,A
STATUS register flags: No flags are affected.

EXAMPLE:

```
0122     ...
0123     (Label)    ANL    MASK,A
0124     ...
```

Before execution: A= C3h (11000011 Bin.)
MASK= 55h (01010101 Bin.)
After execution: MASK= 41h (01000001 Bin.)

ANL direct,#data - AND immediate data to direct byte
Direct: Arbitrary register with address 0 - 255 (0 - FFh), Data: constant in the range between 0-255 (0-FFh)
Description: Instruction performs logic AND operation between direct byte and data. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh (0-127 dec.). The result is stored in the direct byte.
Syntax: ANL register address ,#data
STATUS register flags: No flags are affected
EXAMPLE:

```
0A22     ...
0A23     (Label)    ANL    MASK,#C3h
0A24     ...
```

Before execution: X= C3h (11000011 Bin.) MASK= 55h (01010101 Bin.) After execution: MASK= 41h (01000001 Bin.)

ANL C,bit - AND direct bit to the carry flag
C: Carry flag, Bit: any bit of RAM
Description: Instruction performs logic AND operation between the direct bit and the carry flag.

BIT	C	C AND BIT
0	0	0
0	1	0
1	0	0
1	1	1

Syntax: ANL C, bit address
STATUS register flags: C

EXAMPLE:

```
0222     ...
0223     (Label)    ANL    C,ACC.7
0224     ...
```

Before execution: ACC= 43h (01000011 Bin.) C=1
After execution: ACC= 43h (01000011 Bin.) C=0

ANL C,/bit - AND complements of direct bit to the carry flag
C: carry flag, Bit: any bit of RAM
Description: Instruction performs logic AND operation between inverted addressed bit and the carry flag. The result is stored in the carry flag.

BIT	BIT	C	C AND BIT
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0

Syntax: ANL C,/[bit address]
STATUS register flags: No flags are affected

EXAMPLE:

```
0222     ...
0223     (Label)    ANL    C,/ACC.7
0224     ...
```

Before execution: ACC= 43h (01000011 Bin.) C=1
After execution: ACC= 43h (01000011 Bin.) C=1

CJNE A,direct,rel - Compares direct byte to the accumulator and jumps if not equal
A: accumulator, Direct: arbitrary register with address 0-255 (0-FFh), addr: jump address
Description: Instruction first compares the number in the accumulator with the directly addressed byte. If they are equal, the program proceeds with execution. Otherwise, a jump to the specified address will be executed. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-128 to +127 locations relative to the first following instruction).
Syntax: CJNE A,direct,[jump address]
STATUS register flags: C

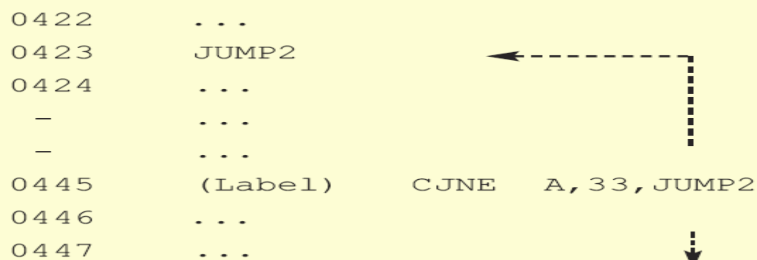
EXAMPLE:

```
0122    ...
0123    JUMP1
0124    ...
 –
0145    (Label)    CJNE    A,MAX,JUMP1
0146    ...
0147    ...
```

Before execution: PC=0145h A=27h
After execution: if MAX≠27: PC=0123h
If MAX=27: PC=0146h

CJNE A,#data,rel - Compares immediate data to the accumulator and jumps if not equal
A: accumulator, Data: constant in the range of 0-255 (0-FFh)
Description: Instruction first compares the number in the accumulator with the immediate data. If they are equal, the program proceeds with execution. Otherwise, a jump to the specified address will be executed. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-128 to +127 locations relative to the first following instruction).
Syntax: CJNE A,X,[jump address]
STATUS register flags: C

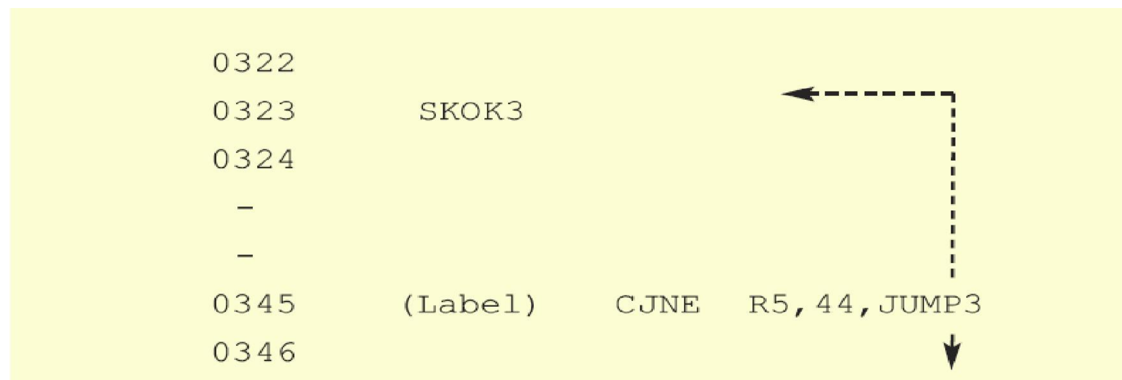
EXAMPLE:



```
0422    ...
0423    JUMP2
0424    ...
 –      ...
 –      ...
0445    (Label)    CJNE   A,33,JUMP2
0446    ...
0447    ...
```

Before execution: PC=0445h
After execution: If A≠33: PC=0423h
If A=33: PC=0446h

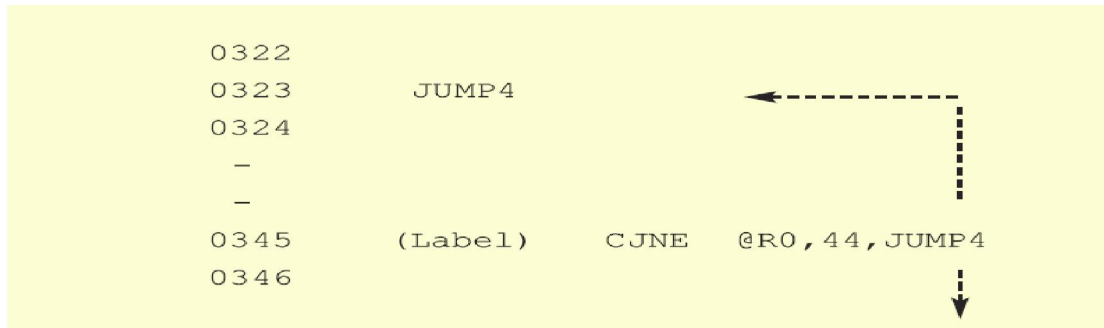
CJNE Rn,#data,rel - Compares immediate data to the register Rn and jumps if not equal
Rn: Any R register (R0-R7), Data: Constant in the range of 0 - 255 (0-FFh), addr: Jump address
Description: Instruction first compares immediate data to the register Rn. If they are equal, the program proceeds with execution. Otherwise, a jump to the specified address will be executed. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-128 to + 127 locations relative to the first following instruction).
Syntax: CJNE Rn,data,[jump address]
STATUS register flags: C
EXAMPLE:



Before execution: PC=0345h
After execution: If R5≠44h: PC=0323h
If R5=44h: PC=0346h

CJNE @Ri,#data,rel - Compares immediate data to indirectly addressed register and jumps if not equal
Ri: Register R0 or R1, Data: Constant in the range of 0 - 255 (0-FFh)
Description: This instruction first compares immediate data to indirectly addressed register. If they are equal, the program proceeds with execution. Otherwise, a jump to the specified address in the program will be executed. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-128 to +127 locations relative to the next instruction).
Syntax: CJNE @Ri,data,[jump address]
STATUS register flags: C

EXAMPLE:



Before execution: Register Address SUM=F3h
PC=0345h R0=F3h
After execution: If SUM≠44h: PC=0323h
If SUM=44h: PC=0346h

CLR A - Clears the accumulator
A: accumulator
Description: Instruction clears the accumulator.
Syntax: CLR A
STATUS register flags: No flags are affected.
EXAMPLE:

```
0322    ...
0323    CLR   A
0324    ...
```

After execution: A=0

CLR C - clears the carry flag
C: Carry flag
Description: Instruction clears the carry flag.
Syntax: CLR C
STATUS register flags: C
EXAMPLE:

```
0322
0323    CLR   C
0324
```

After execution: C=0

CLR bit - clears the direct bit
Bit: any bit of RAM
Description: Instruction clears the specified bit.

Syntax: CLR [bit address];
Bytes: 2 (instruction code, bit address);
STATUS register flags: No flags are affected.
EXAMPLE:

```
0322    ...
0323    CLR    P0.3
0324    ...
```

Before execution: P0.3=1 (input pin)
After execution: P0.3=0 (output pin)

CPL A - Complements the accumulator
A: accumulator
Description: Instruction complements all the bits in the accumulator (1==>0, 0==>1).
Syntax: CPL A
STATUS register flags: No flags are affected.
EXAMPLE:

```
0322    ...
0323    CPL    A
0324    ...
```

Before execution: A= (00110110)
After execution: A= (11001001)

CPL bit - Complements the direct bit
Bit: any bit of RAM
Description: Instruction complements the specified bit of RAM (0=>1, 1=>0).
Syntax: CPL [bit address]
STATUS register flags: No flags are affected

EXAMPLE:

```
0322    ...
0323    CPL    P0.3
0324    ...
```

Before execution: P0.3=1 (input pin)
After execution: P0.3=0 (output pin)

CPL C - Complements the carry flag
C: Carry flag
Description: Instruction complements the carry flag (0==>1, 1==>0).
Syntax: CPL C
STATUS register flags: C
EXAMPLE:

```
0322     ...
0323     CPL   C
0324     ...
```

Before execution: C=1
After execution: C=0

DA A - Decimal adjust accumulator
A: accumulator
Description: Instruction adjusts the contents of the accumulator to correspond to a BCD number after two BCD numbers have been added by the ADD and ADDC instructions. The result in form of two 4-digit BCD numbers is stored in the accumulator.
Syntax: DA A
STATUS register flags: C
EXAMPLE:

```
0322    ...
0323    ADDC      A,B
0324    DA         A
```

Before execution: A=56h (01010110) 56 BCD
B=67h (01100111) 67BCD
After execution: A=BDh (10111101)
After BCD conversion: A=23h (00100011), C=1 (Overflow) (C+23=123) = 56+67

DEC A - Decrements the accumulator by 1
Description: Instruction decrements the value in the accumulator by 1. If there is a 0 in the accumulator, the result of the operation is FFh. (255 dec.)
Syntax: DEC A
STATUS register flags: No flags are affected
EXAMPLE:

```
0322     ...
0323     DEC      A
0324     ...
```

Before execution: A=E4h
After execution: A=E3h

DEC Rn - Decrements the register Rn by 1
Description: Instruction decrements the value in the Rn register by 1. If there is a 0 in the register, the result of the operation will be FFh. (255 dec.)
Syntax: DEC Rn
STATUS register flags: No flags are affected
EXAMPLE:

```
0322    ...
0323    DEC     R3
0324    ...
```

Before execution: R3=B0h
After execution: R3=AFh

DEC direct - Decrements the direct byte by 1
Direct: arbitrary register with address 0-255 (0-FFh)
Description: Instruction decrements the value of directly addressed register by 1. As it is direct addressing, the register must be within the first 255 locations of RAM. If there is a 0 in the register, the result will be FFh.
Syntax: DEC [register address]
STATUS register flags: No flags are affected
EXAMPLE:

```
0322
0323   DEC      CNT
0324
```

Before execution: CNT=0
After execution: CNT=FFh

DIV AB - Divides the accumulator by the register B
A: accumulator, B: Register B
Description: Instruction divides the value in the accumulator by the value in the B register. After division the integer part of result is stored in the accumulator while the register contains the remainder. In case of dividing by 1, the flag OV is set and the result of division is unpredictable. The 8-bit quotient is stored in the accumulator and the 8-bit remainder is stored in the B register.
Syntax: DIV AB
STATUS register flags: C, OV

EXAMPLE:

```
0222     ...
0223     (Label)    DIV     AB
0224     ...
```

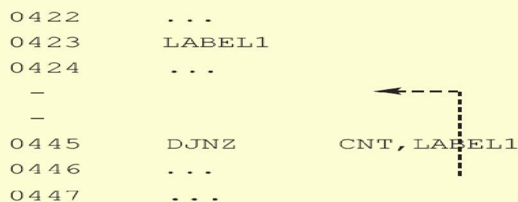
Before execution: A=FBh (251dec.) B=12h (18 dec.)
After execution: A=0Dh (13dec.) B=11h (17dec.) 13·18 + 17 =251

DEC @Ri - Decrements the indirect RAM by 1
Ri: Register R0 or R1
Description: This instruction decrements the value in the indirectly addressed register of RAM by 1. The register address is stored in the Ri register (R0 or R1). If there is a 0 in the register, the result will be FFh.
Syntax: DEC @Ri
STATUS register flags: No flags are affected
EXAMPLE:

```
0222     ...
0223     Label)    DEC     @R0
0224     ...
```

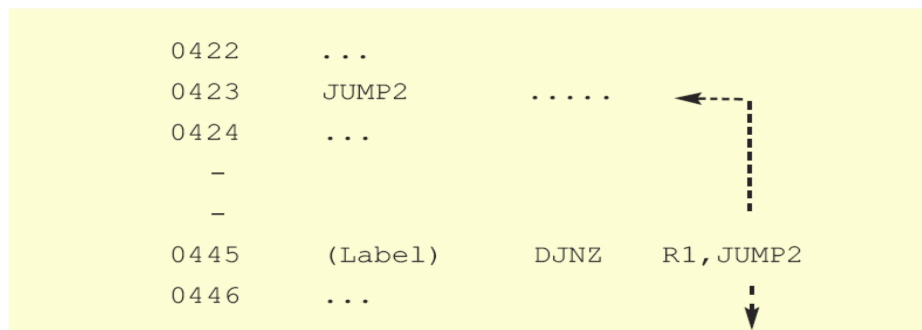
Register Address CNT = 4Fh R0=4Fh
Before execution: CNT=35h
After execution: CNT= 34h

DJNZ direct,rel - Decrements direct byte by 1 and jumps if not 0
Direct: arbitrary register with address 0-255 (0-FFh), addr: Jump address.
Description: This instruction first decrements value in the register. If the result is 0, the program proceeds with execution. Otherwise, a jump to the specified address in the program will be executed. As it is direct addressing, the register must be within the first 255 locations of RAM. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-128 to +127 locations relative to the first following instruction).
Syntax: DJNZ direct,[jump address]
STATUS register flags: No flags are affected
EXAMPLE:



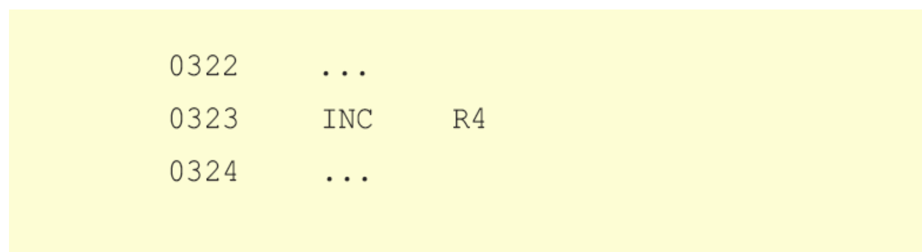
Before execution: PC=0445h
After execution: If CNT≠0: PC=0423h
If CNT=0: PC=0446h

DJNZ Rn,rel - Decrements the Rn register by 1 and jumps if not 0
Rn: any R register (R0-R7), addr: jump address
Description: This instruction first decrements the value in the Rn register. If the result is 0, the program proceeds with execution. Otherwise, a jump to the specified address in the program will be executed. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (- 128 to +127 locations relative to the first following instruction).
Syntax: DJNZ Rn, [jump address]
STATUS register flags: No flags are affected
EXAMPLE:



Before execution: PC=0445h
After execution: If R1≠0: PC=0423h
If R1=0: PC=0446h

INC Rn - Increments the Rn register by 1
Rn: any R register (R0-R7)
Description: Instruction increments the value in the Rn register by 1. If the register includes the number 255, the result of the operation will be 0.
Syntax: INC Rn
STATUS register flags: No flags are affected
EXAMPLE:



Before execution: R4=18h
After execution: R4=19h

INC A - Increments the accumulator by 1
A: accumulator
Description: This instruction increments the value in the accumulator by 1. If the accumulator includes the number 255, the result of the operation will be 0.
Syntax: INC A
STATUS register flags: No flags are affected
EXAMPLE:

```
0322     ...
0323     INC     A
0324     ...
```

Before execution: A=E4h
After execution: A=E5h

INC @Ri - Increments the value of indirectly addressed register of RAM by 1
Ri: Register R0 or R1
Description: This instruction increments the value in the directly addressed register of RAM by 1. The register address is stored in the Ri Register (R0 or R1). If register includes the number 255, the result of the operation will be 0.
Syntax: INC @Ri
STATUS register flags: No flags are affected
EXAMPLE:

```
0222    ...
0223    (Label)   INC   @R1
0224    ...
```

Register Address CNT = 4Fh
Before execution: CNT=35h R1=4Fh
After execution: CNT=36h

INC direct - Increments the direct byte by 1
Direct: arbitrary register with address 0-255 (0-FFh)
Description: Instruction increments the direct byte by 1. If the register includes the number 255, the result of the operation will be 0. As it is direct addressing, the register must be within the first 255 RAM locations.
Syntax: INC direct
STATUS register flags: No flags are affected
EXAMPLE:

```
0322    ...
0323    INC    CNT
0324    ...
```

Before execution: CNT=33h
After execution: CNT=34h

JB bit,rel - Jump if direct bit is set
addr: Jump address, Bit: any bit of RAM
Description: If the bit is set, a jump to the specified address will be executed. Otherwise, if the value of bit is 0, the program proceeds with the next instruction. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-128 to + 127 locations relative to the first following instruction).
Syntax: JB bit, [jump address]
STATUS register flags: No flags are affected
EXAMPLE:

```
0322    ...
0323    (Label)  JB    P0.5,PUSH BUTTON
0324    ...
 -
 -
0345    PUSH BUTTON   .......
0346    ...
```

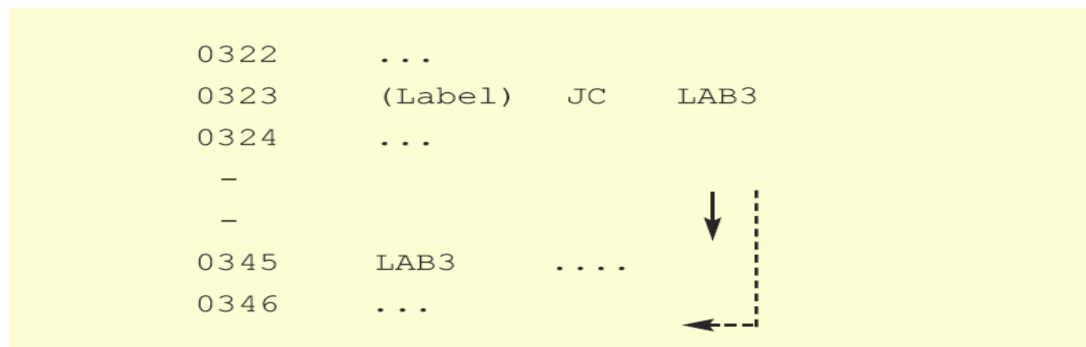
Before execution: PC=0323h
After execution: If P0.5=0: PC=0324h
If P0.5=1: PC=0345h

INC DPTR - Increments the Data Pointer by 1
DPTR: Data Pointer
Description: Instruction increments the value of the 16-bit data pointer by 1. This is the only 16-bit register upon which this operation can be performed.
Syntax: INC DPTR
STATUS register flags: No flags are affected
EXAMPLE:

```
0222    ...
0223    INC    DPTR
0224    ...
```

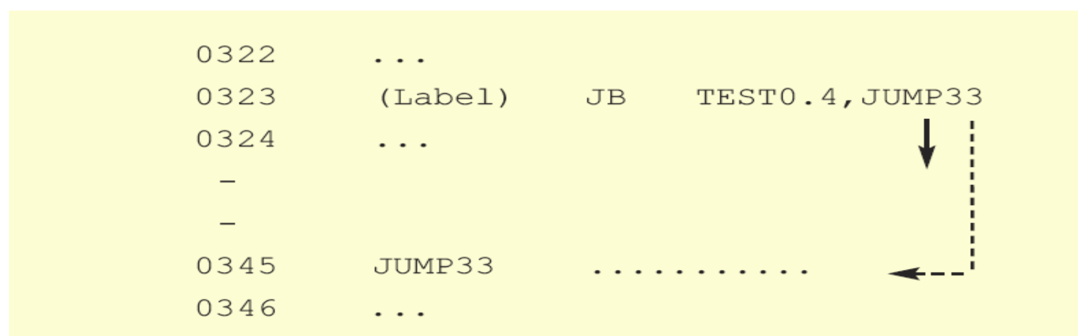
Before execution: DPTR = 13FF (DPH = 13h DPL = FFh)
After execution: DPTR = 1400 (DPH = 14h DPL = 0)

JC rel - Jump if carry flag is set
addr: Jump address
Description: Instruction first checks if the carry flag is set. If set, a jump to the specified address is executed. Otherwise, the program proceeds with the next instruction. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-129 to + 127 locations relative to the first following instruction).
Syntax: JC [jump address]
STATUS register flags: No flags are affected
EXAMPLE:



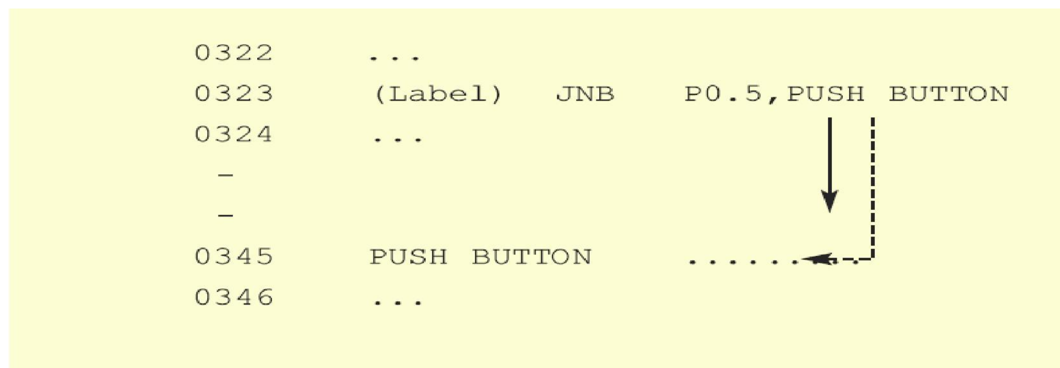
Before instruction: PC=0323h
After instruction: If C=0: PC=0324h
If C=1: PC=0345h

JBC bit,rel - Jump if direct bit is set
Bit: any bit of RAM, addr: Jump Address
Description: This instruction first checks if the bit is set. If set, a jump to the specified address is executed and the bit is cleared. Otherwise, the program proceeds with the first following instruction. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-129 to + 127 locations relative to the first following instruction).
Syntax: JBC bit, [jump address]
STATUS register flags: No flags are affected
EXAMPLE:

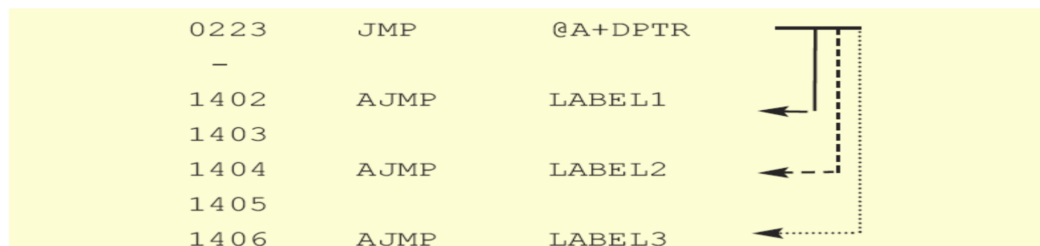


Before execution: PC=0323h
After execution: If TEST0.4=1: PC=0345h, TEST0.4=0
If TEST0.4=0: PC=0324h, TEST0,4=0

JNB bit,rel - Jump if direct bit is not set
addr: Jump address, Bit: any bit of RAM
Description: If the bit is cleared, a jump to the specified address will be executed. Otherwise, if the bit value is 1, the program proceeds with the first following instruction. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-129 to + 127 locations relative to the first following instruction).
Syntax: JNB bit,[jump address]
Bytes: 3 (instruction code, bit address, jump address)
STATUS register flags: No flags are affected
EXAMPLE:

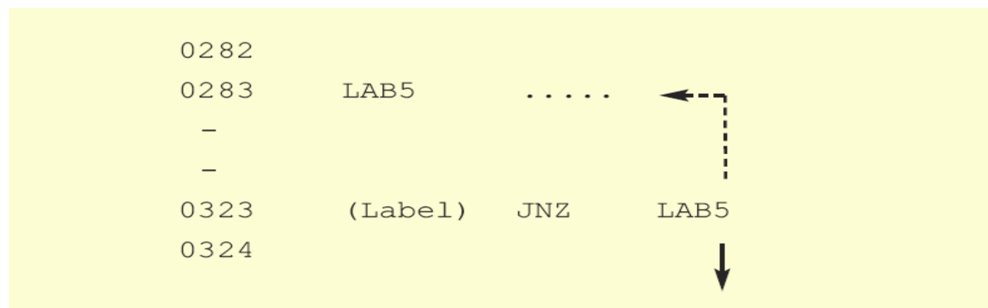


Before execution: PC=0323h
After execution: If P0.5=1: PC=0324h
If P0.5=0: PC=0345h
JMP @A+DPTR - Jump indirect relative to the DPTR
A: accumulator, DPTR: Data Pointer
Description: This instruction causes a jump to the address calculated by adding value stored in the accumulator to the 16-bit number in the DPTR Register. It is used with complex program branching where the accumulator affects jump address, for example when reading a table. Neither accumulator nor DPTR register are affected.
Syntax: JMP @A+DPTR
STATUS register flags: No flags are affected
EXAMPLE:



Before execution: PC=223 DPTR=1400h
After execution: PC = 1402h if A=2
PC = 1404h if A=4
PC = 1406h if A=6
Note: As instructions AJMP LABELS occupy two locations each, the values in the accumulator specifying them must be different from each other by 2.

JNZ rel - Jump if accumulator is not zero
addr: Jump Address
Description: This instruction checks if the value stored in the accumulator is 0. If not, a jump to the specified address will be executed. Otherwise, the program proceeds with the first following instruction. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-129 to + 127 locations relative to the first following instruction).
Syntax: JNZ [jump address]
STATUS register flags: No flags are affected
EXAMPLE:



Before execution: PC=0323h
After execution: If A=0: PC=324h
If A≠0: PC=283h

JNC rel - Jump if carry flag is not set
addr: Jump Address
Description: This instruction first checks whether the carry flag is set. If not, a jump to the specified address will be executed. Otherwise, the program proceeds with the first following instruction. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-129 to + 127 locations relative to the first following instruction).
Syntax: JNC [jump address]
STATUS register flags: No flags are affected
EXAMPLE:

```
0322
0323     (Label)  JNC    LAB4
0324
   -
   -
0360     LAB4    ...
```

Before execution: PC=0323h
After execution: If C=0: PC=360h
If C=1: PC=324h

LCALL addr16 - Long subroutine call
addr16: Subroutine Address
Description: This instruction unconditionally calls a subroutine located at the specified address. The current address and the start of the subroutine called can be located anywhere within the memory space of 64K.
Syntax: LCALL [subroutine name]
STATUS register flags: No flags are affected
EXAMPLE:

```
0123     (Label)   LCALL   SUBRTN 0124
 -
 -
1234      SUBRTN      ........
```

Before execution: PC=0123h
After execution: PC=1234h

JZ rel - Jump if accumulator is zero
addr: Jump Address
Description: The instruction checks whether the value stored in the accumulator is 0. If yes, a jump to the specified address will be executed. Otherwise, the program proceeds with the following instruction. This is a short jump instruction, which means that the address of a new location must be relatively near the current one (-129 to + 127 locations relative to the first following instruction).
Syntax: JZ [jump address]
STATUS register flags: No flags are affected
EXAMPLE:

```
0282
0283     LAB6          ...
 -
 -
0323     (Label)      JZ    LAB6
0324
```

Before execution: PC=0323h
After execution: If A0: PC=324h
If A=0: PC=283h

MOV A,Rn - Moves the Rn register to the accumulator
Rn: any R register (R0-R7), A: accumulator
Description: The instruction moves the Rn register to the accumulator. The Rn register is not affected.
Syntax: MOV A,Rn
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)   MOV    A,R3
0125    ...
```

Before execution: R3=58h
After execution: R3=58h A=58h

LJMP addr16 - Long jump
addr16: jump address
Description: Instruction causes a jump to the specified 16-bit address.
Syntax: LJMP [jump address]
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    (Label)   LJMP   JUMP33
0124    ...
 -
 -
1234     JUMP33   .....
```

Before execution: PC=0123h
After execution: PC=1234h

MOV A,@Ri - Moves the indirect RAM to the accumulator
Ri: Register R0 or R1, A: accumulator
Description: Instruction moves the indirectly addressed register of RAM to the accumulator. The register address is stored in the Ri register (R0 or R1). The result is stored in the accumulator. The register is not affected.

Syntax: MOV A,@Ri
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)   MOV   A,@R0
0125    ...
```

Register Address SUM=F2h R0=F2h
Before execution: SUM=58h
After execution: A=58h SUM=58h

MOV A,direct - Moves the direct byte to the accumulator
Direct: arbitrary register with address 0-255 (0-FFh), A: accumulator
Description: Instruction moves the direct byte to the accumulator. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh. (0-127 dec.). After executing the instruction, the register is not affected.
Syntax: MOV A,direct
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)   MOV   A,Rx
0125    ...
```

Before execution: Rx=68h
After execution: Rx=68h A=68h

MOV Rn,A - Moves the accumulator to the Rn register
Rn: any R register (R0-R7), A: accumulator
Description: Instruction moves the accumulator to the Rn register. The accumulator is not affected.
Syntax: MOV Rn,A
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)   MOV   R3,A
0125    ...
```

Before execution: A=58h
After execution: R3=58h A=58h

MOV A,#data - Moves the immediate data to the accumulator
A: accumulator
Data: Constant in the range of 0-255 (0-FFh)
Desription: Instruction moves the immediate data to the accumulator.
Syntax: MOV A,#data;
STATUS register flags: No flags are affected;
EXAMPLE:

```
0123    ...
0124    (Label)  MOV   A,#28
0125    ...
```

After execution: A=28h
MOV Rn,#data - Moves the immediate data to the Rn register
Rn: any R register (R0-R7) Data: Constant in the range of 0-255 (0-FFh)
Description: Instruction moves the immediate data to the Rn register.
Syntax: MOV Rn,#data
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)  MOV   R5,#32
0125    ...
```

After execution: R5=32h

MOV Rn,direct - Moves the direct byte to the Rn register
Rn: Any R registar (R0-R7), Direct: arbitrary register with address 0-255 (0-FFh)
Description: Instruction moves the direct byte to the Rn register. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh. (0-127 dec.). After executing the instruction, the register is not affected.
Syntax: MOV Rn,direct
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)   MOV   R3,SUM
0125    ...
```

Before execution: SUM=58h
After execution: SUM=58h R3=58h

MOV direct,Rn - Moves the Rn register to the direct byte
Rn: any R register (R0-R7), Direct: arbitrary register with address 0-255(0-FFh)
Description: Instruction moves the Rn register to the direct byte. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh. (0-127 dec.). After executing the instruction, the register is not affected.
Syntax: MOV direct,Rn
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)   MOV    CIF,R3
0125    ...
```

Before execution: R3=18h
After execution: R3=18h CIF=18h

MOV direct,A - Moves the accumulator to the direct byte
Direct: arbitrary register with address 0-255 (0 - FFh), A: accumulator
Description: Instruction moves the accumulator to the direct byte. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh. (0-127 dec.). After executing the instruction, the register is not affected.
Syntax: MOV direct,A
STATUS register flags: No flags are affected

EXAMPLE:

```
0723    ...
0724    (Label)    MOV    REG,A
0725    ...
```

Before execution: A=98h
After execution: A=98h REG=98h

MOV direct,@Ri - Moves the indirect RAM to the direct byte
Direct: arbitrary register with address 0-255 (0 - FFh), Ri: Register R0 or R1
Description: Instruction moves the indirectly adressed register of RAM to the direct byte. The register is not affected.
Syntax: MOV direct,@Ri
STATUS register flags: No flags are affected

EXAMPLE:

```
0123     ...
0124     (Label)    MOV    TEMP,@R1
0125     ...
```

Register Address SUM=F3
Before execution: SUM=58h R1=F3
After execution: SUM=58h TEMP=58h

MOV direct1,direct2 - Moves the direct byte to the direct byte
Direct: Arbitrary register with address 0-255 (0-FFh), Direct: Arbitrary register with address 0-255 (0-FFh)
Description: Instruction moves the direct byte to another direct byte. As it is direct addressing, both registers can be any SFRs or general-purpose registers with address 0-7Fh. (0-127 dec.). The direct1 is not affected.
Syntax: MOV direct1,direct2
STATUS register flags: No flags are affected.
EXAMPLE:

```
0223     ...
0224     (Label)    MOV    SUM,TEMP
0225     ...
```

Before execution: TEMP=58h
After execution: TEMP=58h SUM=58h

MOV @Ri,A - Moves the accumulator to the indirect RAM
A: accumulator, Ri: register R0 or R1
Description: Instruction moves the accumulator to the indirectly addressed register of RAM. The register address is stored in the Ri register (R0 or R1). After executing the instruction, the accumulator is not affected.
Syntax: MOV @Ri,A
STATUS register flags: No flags are affected
EXAMPLE:

```
0123     ...
0124     (Label)   MOV    @R0,A
0125     ...
```

Register Address SUM=F2h
Before execution: R0=F2h A=58h
After execution: SUM=58h A=58h

MOV direct,#data - Moves the immediate data to the direct byte
Direct: Arbitrary register with address 0-255 (0-FFh), Data: Constant in the range of 0-255 (0-FFh)
Description: Instruction moves the immediate data to the direct byte. As it is direct addressing, the direct byte can be any SFRs or general-purpose register with address 0-7Fh. (0-127 dec.).
Syntax: MOV direct,#data
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)  MOV   TEMP,#22
0125    ...
```

After execution: TEMP=22h

MOV @Ri,#data - Moves the immediate data to the indirect RAM
Ri: Register R0 or R1, Data: Constant in the range of 0-255 (0-FFh)
Description: Instruction moves the immediate data to the idirectly addressed register of RAM. The register address is stored in the Ri register (R0 or R1).
Syntax: MOV @Ri,#data
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)  MOV   @R1,#44
0125    ...
```

Register address TEMP=E2h
Before execution: R1=E2h
After execution: TEMP=44h

MOV @Ri,direct - Moves the direct byte to the indirect RAM
Direct: Arbitrary register with address 0-255 (0-FFh), Ri: Register R0 or R1
Description: Instruction moves the direct byte to a register the address of which is stored in the Ri register (R0 or R1). After executing the instruction, the direct byte is not affected.
Syntax: MOV @Ri,direct
STATUS register flags: No flags are affected
EXAMPLE:

```
0123    ...
0124    (Label)  MOV   @R1,ZBIR
0125    ...
```

Register address TEMP=E2h
Before execution: SUM=58h R1=E2h
After execution: SUM=58h TEMP=58h

MOV bit,C - Moves the carry flag to the direct bit
C: Carry flag, Bit: any bit of RAM
Description: Instruction moves the carry flag to the direct bit. After executing the instruction, the carry flag is not affected.
Syntax: MOV bit,C
STATUS register flags: No flags are affected
EXAMPLE:

```
0123     ...
0124     (Label)   MOV    P1.2,C
0125     ...
```

After execution: If C=0 P1.2=0
If C=1 P1.2=1

MOV C,bit - Moves the direct bit to the carry flag
C: Carry flag, Bit: any bit of RAM
Description: Instruction moves the direct bit to the carry flag. After executing the instruction, the bit is not affected.
Syntax: MOV C,bit
STATUS register flags: C
EXAMPLE:

```
0123     ...
0124     (Label)   MOV    C,P1.4
0125     ...
```

After execution: If P1.4=0 C=0
If P1.4=1 C=1

MOVC A,@A+DPTR - Moves the code byte relative to the DPTR to the accumulator
A: accumulator, DPTR: Data Pointer
Description: Instruction first adds the 16-bit DPTR register to the accumulator. The result of addition is then used as a memory address from which the 8-bit data is moved to the accumulator.
Syntax: MOVC A,@A+DPTR
STATUS register flags: No flags affected

EXAMPLE:

```
0100                          MOVC  A,@A+DPTR
 -
1100      Tabela              DB     66h
1101                          DB     77h
1102                          DB     88h
1103                          DB     99h
```

Before execution:
DPTR=1000:
A=0
A=1
A=2
A=3
After execution:
A=66h
A=77h
A=88h
A=99h
Note: DB (Define Byte) is a directive in assembly language used to define constant.

MOV DPTR,#data16 - Loads the data pointer with a 16-bit constant
Data: constant in the range of 0-65535 (0-FFFFh), DPTR: Data Pointer
Description: Instruction stores a 16-bit constant to the DPTR register. The 8 high bits of the constant are stored in the DPH register, while the 8 low bits are stored in the DPL register.
Syntax: MOV DPTR,#data
STATUS register flags: No flags affected
EXAMPLE:

```
0123     ...
0124     (Labela)   MOV    DPTR,#1234
0125     ...
```

After execution: DPH=12h DPL=34h

MOVX A,@Ri - Moves the external RAM (8-bit address) to the accumulator
Ri: register R0 or R1, A: accumulator
Description: Instruction reads the content of a register in external RAM and moves it to the accumulator. The register address is stored in the Ri register (R0 or R1).
Syntax: MOVX A,@Ri
STATUS register flags: No flags affected

EXAMPLE:

```
0123     ...
0124     (Labela)   MOVX    A,@R0
0125     ...
```

Register Address: SUM=12h
Before execution: SUM=58h R0=12h
After execution: A=58h
Note: SUM Register is stored in external RAM which is 256 bytes in size.

MOVC A,@A+PC - Moves the code byte relative to the PC to the accumulator
A: accumulator, PC: Program Counter
Description: Instruction first adds the 16-bit PC register to the accumulator (the current program address is stored in the PC register). The result of addition is then used as a memory address from which the 8-bit data is moved to the accumulator.
Syntax: MOVC A,@A+PC
STATUS register flags: No flags affected
EXAMPLE:

```
0100    (Tabela)      INC     A
0101                  MOVC    A,@A+PC
0102                  RET
0103                  DB      66h
0104                  DB      77h
0105                  DB      88h
0106                  DB      99h
```

After the subroutine "Table" has been executed, one of four values is stored in the accumulator:
Before execution:
A=0
A=1
A=2
A=3
After execution:
A=66h
A=77h
A=88h
A=99h
Note: DB (Define Byte) is a directive in assembly language used to define constant.

MOVX @Ri,A - Moves the accumulator to the external RAM (8-bit address)
Ri: register R0 or R1, A: accumulator

Description: Instruction moves the accumulator to a register stored in external RAM. Its address is stored in the Ri register.
Syntax: MOVX @Ri,A
STATUS register flags: No flags affected
EXAMPLE:

```
0123    ...
0124    (Label)  MOVX   @R1,A
0125    ...
```

Register address: SUM=34h
Before execution: A=58 R1=34h
After execution: SUM=58h
NOTE:
Register SUM is located in external RAM which is 256 bytes in size.
MOVX A,@DPTR - Moves the external memory (16-bit address) to the accumulator
A: accumulator, DPRTR: Data Pointer
Description: Instruction moves the content of a register in external memory to the accumulator. The 16-bit address of the register is stored in the DPTR register (DPH and DPL).
Syntax: MOVX A,@DPTR
STATUS register flags: No flags affected
EXAMPLE:

```
0123    ...
0124    (Label) MOVX  A,@DPTR
0125    ...
```

Register address: SUM=1234h
Before execution: DPTR=1234h SUM=58
After execution: A=58h
Note: Register SUM is located in external RAM which is up to 64K in size.

MUL AB - Multiplies A and B
A: accumulator, B: Register B
Description: Instruction multiplies the value in the accumulator with the value in the B register. The low-order byte of the 16-bit result is stored in the accumulator, while the high byte remains in the B register. If the result is larger than 255, the overflow flag is set. The carry flag is not affected.
Syntax: MUL AB
STATUS register flags: No flags affected

EXAMPLE:

```
0123    ...
0124    (Label)  MUL   AB
0125    ...
```

Before execution: A=80 (50h) B=160 (A0h)
After execution: A=0 B=32h
A·B=80·160=12800 (3200h)

MOVX @DPTR,A - Moves the accumulator to the external RAM (16-bit address)
A: accumulator, DPTR: Data Pointer
Description: Instruction moves the accumulator to a register stored in external RAM. The 16-bit address of the register is stored in the DPTR register (DPH and DPL).
Syntax: MOVX @DPTR,A
STATUS register flags: No flags affected
EXAMPLE:

```
0123    ...
0124    (Labela) MOVX   @DPTR,A
0125    ...
```

Register address: SUM=1234h
Before execution: A=58 DPTR=1234h
After execution: SUM=58h
Note: Register SUM is located in RAM which is up to 64K in size.

ORL A,Rn - OR register to the accumulator
Rn: any R register (R0-R7), A: accumulator
Description: Instruction performs logic OR operation between the accumulator and Rn register. The result is stored in the accumulator.
Syntax: ORL A,Rn;
STATUS register flags: No flags affected
EXAMPLE:

```
0123    ...
0124    (Label)  ORL   A,R5
0125    ...
```

Before execution: A= C3h (11000011 Bin.)
R5= 55h (01010101 Bin.)
After execution: A= D7h (11010111 Bin.)

NOP - No operation
Description: Instruction doesn't perform any operation and is used when additional time delays are needed.
Syntax: NOP
STATUS register flags: No flags affected
EXAMPLE:

```
0123
0124    (Label)    CLR    P2.3
0125               NOP
0126               NOP
0127               NOP
0128               NOP
0129               SETB   P2.3
```

Such a sequence provides a negative pulse which lasts exactly 5 machine cycles on the P2.3. If a 12 MHz quartz crystal is used then 1 cycle lasts 1uS, which means that this output will be a low-going output pulse for 5 uS.

ORL A,@Ri - OR indirect RAM to the accumulator
Ri: register R0 or R1, A: accumulator
Description: Instruction performs logic OR operation between the accumulator and a register. As it is indirect addressing, the register address is stored in the Ri register (R0 or R1). The result is stored in the accumulator.
Syntax: ANL A,@Ri
STATUS register flags: No flags affected
EXAMPLE:

```
0123    ...
0124    (Label)  ORL   A,@R1
0125    ...
```

Register address: TEMP=FAh
Before execution: R1=FAh
TEMP= C2h (11000010 Bin.)
A= 54h (01010100 Bin.)
After execution: A= D6h (11010110 Bin.)

ORL A,direct - OR direct byte to the accumulator
Direct: arbitrary register with address 0-255 (0-FFh), A: accumulator
Description: Instruction performs logic OR operation between the accumulator and a register. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh (0-127 dec.). The result is stored in the accumulator.
Syntax: ORL A,direct
STATUS register flags: No flags affected
EXAMPLE:

```
0123    ...
0124    (Label)  ORL   A,LOG
0125    ...
```

Before execution: A= C2h (11000010 Bin.)
LOG= 54h (01010100 Bin.)
After execution: A= D6h (11010110 Bin.)

ORL direct,A - OR accumulator to the direct byte
Direct: arbitrary register with address 0-255 (0-FFh), A: accumulator
Description: Instruction performs logic OR operation between a register and accumulator. As it is direct addressing, the register can be any SFRs or general- purpose register with address 0-7Fh (0-127 dec.). The result is stored in the register.
Syntax: ORL [register address], A
STATUS register flags: No flags affected
EXAMPLE:

```
0123    ...
0124    (Label)  ORL   TEMP,A
0125    ...
```

Before execution: TEMP= C2h (11000010 Bin.)
A= 54h (01010100 Bin.)
After execution: A= D6h (11010110 Bin.)

ORL A,#data - OR immediate data to the accumulator
Data: constant in the range of 0-255 (0-FFh), A: accumulator
Description: Instruction performs logic OR operation between the accumulator and the immediate data. The result is stored in the accumulator.
Syntax: ORL A, #data
STATUS register flags: No flags affected

EXAMPLE:

```
0123    ...
0124    (Label)  ORL   A,11
0125    ...
```

Before execution: A= C2h (11000010 Bin.)
After execution: A= C3h (11000011 Bin.)

ORL C,bit - OR direct bit to the carry flag
C: Carry flag, Bit: any bit of RAM
Description: Instruction performs logic OR operation between the direct bit and the carry flag. The result is stored in the carry flag.
Syntax: ORL C,bit
STATUS register flags: No flags affected
EXAMPLE:

```
0123    ...
0124    (Label)  ORL   C,ACC.3
0125    ...
```

Before execution: ACC= C6h (11001010 Bin.) C=0
After execution: C=1

ORL direct,#data - OR immediate data to direct byte
Direct: arbitrary register with address 0-255 (0-FFh), Data: constant in the range of 0-255 (0-FFh)
Description: Instruction performs logic OR operation between the immediate data and the direct byte. As it is direct addressing, the direct byte can be any SFRs or general-purpose register with address 0-7Fh (0-127 dec.). The result is stored in the direct byte.
Syntax: ORL [register address],#data
STATUS register flags: No flags affected
EXAMPLE:

```
0123    ...
0124    (Label)   ORL   TEMP,#12
0125    ...
```

Before execution: TEMP= C2h (11000010 Bin.)
After execution: A= D2h (11010010 Bin.)

POP direct - Pop the direct byte from the stack
Direct: arbitrary register with address 0-255 (0-FFh)

Description: Instruction first reads data from the location being pointed to by the Stack. The data is then copied to the direct byte and the value of the Stack Pointer is decremented by 1. As it is direct addressing, the direct byte can be any SFRs or general-purpose register with address 0-7Fh. (0-127 dec.).
Syntax: POP direct;
Bytes: 2 (instruction code, direct byte address);
STATUS register flags: No flags affected;
EXAMPLE:

```
0123    ...
0124    (Label)    POP   DPH
0125    ...        POP   DPL
```

Before execution: Address Value
030h 20h
031h 23h
SP==> 032h 01h
DPTR=0123h (DPH=01, DPL=23h)
After execution: Address Value
SP==> 030h 20h
031h 23h
032h 01h

ORL C,/bit - OR complements of direct bit to the carry flag
C: carry flag
Bit: any bit of RAM
Description: Instruction performs logic OR operation between the addressed inverted bit and the carry flag. The result is stored in the carry flag.

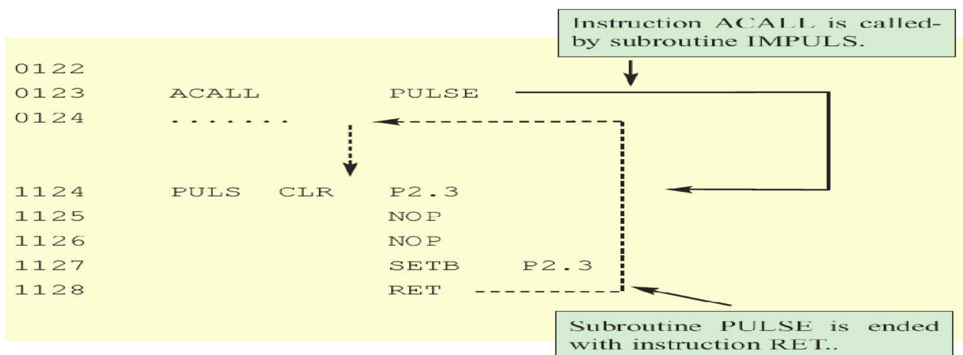
BIT	BIT	C	C AND BIT
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0

Syntax: ORL C,/bit;
Bytes: 2 (instruction code, bit address);
STATUS register flags: No flags affected;

EXAMPLE:

```
0123    ...
0124    (Label)  ORL   C,ACC.3
0125    ...
```

Before execution: ACC= C6h (11001010 Bin.)
C=0
After execution: C=0
RET - Return from subroutine
Description: This instruction ends every subroutine. After execution, the program proceeds with the instruction following an ACALL or LCALL.
Syntax: RET;
Byte: 1 (instruction code);
STATUS register flags: No flags affected;
EXAMPLE:



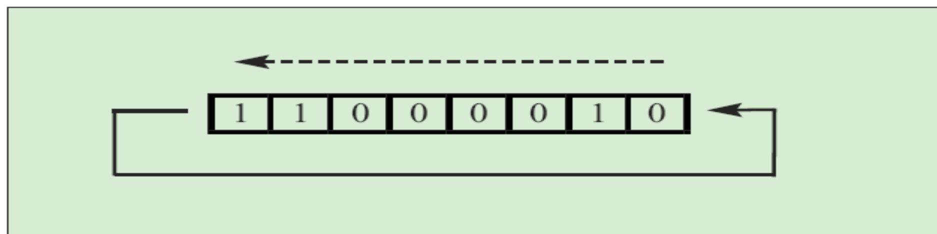
PUSH direct - Pushes the direct byte onto the stack
Data: Arbitrary register with address 0-255 (0-FFh)
Description: Address currently pointed to by the Stack Pointer is first incremented by 1 and afterwards the data from the register Rx is copied to it. As it is direct addressing, the direct byte can be any SFRs or general-purpose register with address 0-7Fh. (0-127 dec.)
Syntax: PUSH direct;
Bytes: 2 (instruction code, direct byte address);
STATUS register flags: No flags affected;
EXAMPLE:

```
0123
0124    (Label)    PUSH    DPL
0125               PUSH    DPH
```

Before execution: Address Value
SP==> 030h 20h
DPTR=0123h (DPH=01, DPL=23h)
After execution: Address Value
030h 20h
031h 23h
SP==> 032h 01h
RL A - Rotates the accumulator one bit left
A: accumulator
Description: Eight bits in the accumulator are rotated one bit left, so that the bit 7 is rotated into the bit 0 position.
Syntax: RL A;
Byte: 1 (instruction code);
STATUS register flags: No flags affected;
EXAMPLE:

```
0123    ...
0124    (Label)   RL  A
0125    ...
```

Before execution: A= C2h (11000010 Bin.)
After execution: A=85h (10000101 Bin.)

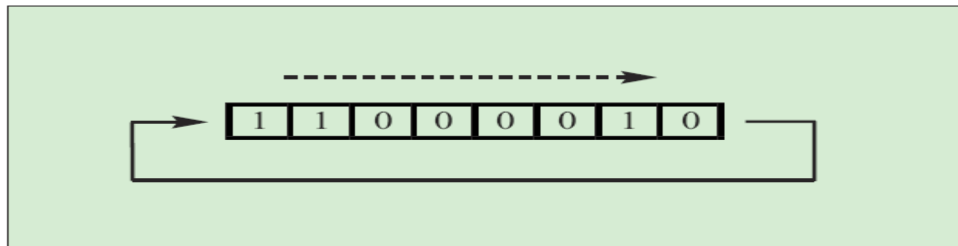


RETI - Return from interrupt
Description: This instruction ends every interrupt routine and informs processor about it. After executing the instruction, the program proceeds from where it left off. The PSW is not automatically returned its pre-interrupt status.
Syntax: RETI;
Byte: 1 (instruction code);
STATUS register flags: No flags affected;
RR A - Rotates the accumulator one bit right
A: accumulator
Description: All eight bits in the accumulator are rotated one bit right so that the bit 0 is rotated into the bit 7 position.

Syntax: RR A;
Byte: 1 (instruction code);
STATUS register flags: No flags affected;
EXAMPLE:

```
0123    ...
0124    (Label)   RR    A
0125    ...
```

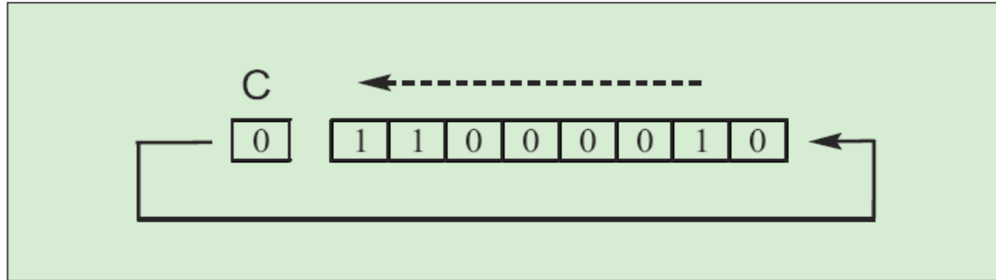
Before execution: A= C2h (11000010 Bin.)
After execution: A= 61h (01100001 Bin.)



RLC A - Rotates the accumulator one bit left through the carry flag
A: accumulator
Description: All eight bits in the accumulator and carry flag are rotated one bit left. After this operation, the bit 7 is rotated into the carry flag position and the carry flag is rotated into the bit 0 position.
Syntax: RLC A;
Byte: 1 (instruction code);
STATUS register flags: C;
EXAMPLE:

```
0123    ...
0124    (Label)    RLC   A
0125    ...
```

Before execution: A= C2h (11000010 Bin.)
C=0
After execution: A= 85h (10000100 Bin.)
C=1



SETB C - Sets the carry flag
C: Carry flag
Description: Instruction sets the carry flag.
Syntax: SETB C;
Byte: 1 (instruction code);
STATUS register flags: C;
EXAMPLE:

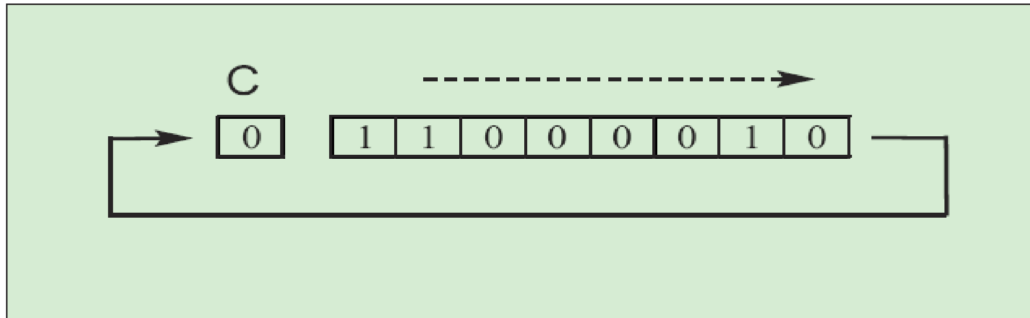
```
0123    ...
0124    (Label)  SETB   C
0125    ...
```

After execution: C=1

RRC A - Rotates the accumulator one bit right through the carry flag
A: accumulator
Description: All eight bits in the accumulator and carry flag are rotated one bit right. After this operation, the carry flag is rotated into the bit 7 position and the bit 0 is rotated into the carry flag position.
Syntax: RRC A;
Byte: 1 (instruction code);
STATUS register flags: C;
EXAMPLE:

```
0123    ...
0124    (Label)  RRC  A
0125    ...
```

Before execution: A= C2h (11000010 Bin.)
C=0
After execution: A= 61h (01100001 Bin.)
C=0



SJMP rel - Short Jump (relative address)
addr: Jump Address
Description: Instruction enables jump to the new address which should be in the range of -128 to +127 locations relative to the first following instruction.
Syntax: SJMP [jump address];
Bytes: 2 (instruction code, jump value);
STATUS register flags: No flags are affected;
EXAMPLE:

```
0322     ...
0323     (Label)   SJMP   PUSH BUTTON
0324     ...
 -
 -
0345   PUSH BUTTON        ......
```

Before execution: PC=323
After execution: PC=345

SETB bit - Sets the direct bit
Bit: any bit of RAM
Description: Instruction sets the specified bit. The register containing that bit must belong to the group of the so called bit addressable registers.
Syntax: SETB [bit address];
Bytes: 2 (instruction code, bit address);
STATUS register flags: No flags affected;
EXAMPLE:

```
0123     ...
0124     (Label)   SETB    P1.0
0125     ...
```

Before execution: P0.1 = 34h (00110100)
pin 1 is configured as an output
After execution: P0.1 = 35h (00110101)
pin 1 is configured as an input

SUBB A,direct - Subtracts the direct byte from the accumulator with a borrow
Direct: arbitrary register with address 0-255 (0-FFh)
A: accumulator
Description: Instruction subtracts the direct byte from the accumulator with a borrow. If the higher bit is subtracted from the lower bit then the carry flag is set. As it is direct addressing, the direct byte can be any SFRs or general-purpose register with address 0-7Fh. (0-127 dec.). The result is stored in the accumulator.
Syntax: SUBB A,direct;
Bytes: 2 (instruction code, direct byte address);
STATUS register flags: C, OV, AC;
EXAMPLE:

```
0322    ...
0323    (Label)  SUBB   A,DIF
0324    ...
```

Before execution: A=C9h, DIF=53h, C=0
After execution: A=76h, C=0

SUBB A,Rn - Subtracts the Rn register from the accumulator with a borrow
Rn: any R register (R0-R7)
A: accumulator
Description: Instruction subtracts the Rn register from the accumulator with a borrow. If the higher bit is subtracted from the lower bit then the carry flag is set. The result is stored in the accumulator.
Syntax: SUBB A,Rn;
Byte: 1 (instruction code);
STATUS register flags: C, OV, AC;
EXAMPLE:

```
0322    ...
0323    (Label)  SUBB    A,R4
0324    ...
```

Before execution: A=C9h, R4=54h, C=1
After execution: A=74h, C=0
Note:
The result is different (C9 - 54=75) because the carry flag is set (C=1) before the instruction starts execution.
SUBB A,#data - Subtracts the immediate data from the accumulator with a borrow
A: accumulator
Data: constant in the range of 0-255 (0-FFh)
Description: Instruction subtracts the immediate data from the accumulator with a borrow. If the higher bit is subtracted from the lower bit then the carry flag is set. The result is stored in the accumulator.
Syntax: SUBB A,#data;
Bytes: 2 (instruction code, data);
STATUS register flags: C, OV, AC;
EXAMPLE:

```
0322    ...
0323    (Label)   SUBB    A,#22
0324    ...
```

Before execution: A=C9h, C=0
After execution: A=A7h, C=0

SUBB A,@Ri - Subtracts the indirect RAM from the accumulator with a borrow
Ri: register R0 or R1
A: accumulator
Description: Instruction subtracts the indirectly addressed register of RAM from the accumulator with a borrow. If the higher bit is subtracted from the lower bit then the carry flag is set. As it is indirect addressing, the register address is stored in the Ri register (R0 or R1). The result is stored in the accumulator.
Syntax: SUBB A,@Ri;
Byte: 1 (instruction code);
STATUS register flags: C, OV, AC;
EXAMPLE:

```
0322    ...
0323    (Label)   SUBB     A,@R1
0324    ...
```

Register address: MIN=F4
Before execution: A=C9h, R1=F4h, MIN=04, C=0
After execution: A=C5h, C=0

XCH A,Rn - Exchanges the Rn register with the accumulator
Rn: any R register (R0-R7)
A: accumulator
Description: Instruction causes the accumulator and Rn register to exchange data. The content of the accumulator is moved to the Rn register and vice versa.
Syntax: XCH A,Rn;
Byte: 1 (instruction code);
STATUS register flags: No flags are affected;
EXAMPLE:

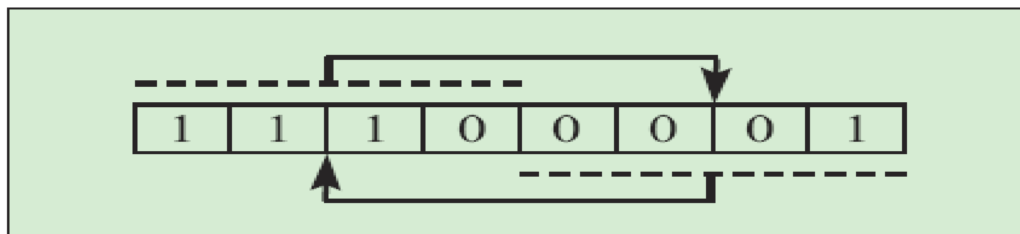
```
0322    ...
0323    (Labela)  XCH   A,R3
0324    ...
```

Before execution: A=C6h, R3=29h
After execution: R3=C6h, A=29h

SWAP A - Swaps nibbles within the accumulator
A: accumulator
Description: A nibble refers to a group of 4 bits within one register (bit0-bit3 and bit4-bit7). This instruction interchanges high and low nibbles of the accumulator.
Syntax: SWAP A;
Byte: 1 (instruction code);
STATUS register flags: No flags are affected;
EXAMPLE:

```
0322    ...
0323    (Label)  SWAP  A
0324    ...
```

Before execution: A=E1h (11100001)bin.
After execution: A=1Eh (00011110)bin.



XCH A,@Ri - Exchanges the indirect RAM with the accumulator
Ri: register R0 or R1
A: accumulator
Description: Instruction moves the contents of accumulator to the indirectly addressed register of RAM and vice versa. As it is indirect addressing, the register address is stored in the register Ri (R0 or R1).
Syntax: XCH A,@Ri;
Byte: 1 (instruction code);
STATUS register flags: No flags are affected;
EXAMPLE:

```
0322    ...
0323    (Label)   XCH    A,@R0
0324    ...
```

Register address: SUM=E3
Before execution: R0=E3, SUM=29h, A=98h
After execution: A=29h, SUM=98h

XCH A,direct - Exchanges the direct byte with the accumulator
Direct: arbitrary register with address 0-255 (0-FFh)
A: accumulator
Description: Instruction moves the contents of the accumulator into the direct byte and vice versa. As it is direct addressing, the direct byte can be any SFRs or general-purpose register with address 0-7Fh (0-127 dec.).
Syntax: XCH A,direct;
Bytes: 2 (instruction code, direct byte address);
STATUS register flags: No flags are affected;
EXAMPLE:

```
0322    ...
0323    (Label)   XCH    A,SUM
0324    ...
```

Before execution: A=FFh, SUM=29h
After execution: SUM=FFh A=29h

XRL A,Rn - Exclusive OR register to accumulator
Rn: any R register (R0-R7)
A: accumulator

Description: Instruction performs exclusive OR operation between the accumulator and the Rn register. The result is stored in the accumulator.
Syntax: XRL A,Rn;
Byte: 1 (instruction code);
STATUS register flags: No flags are affected;
EXAMPLE:

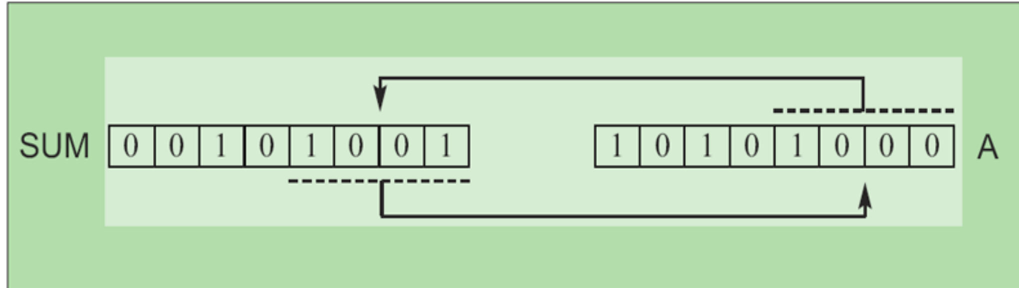
```
0123    ...
0124    (Label)  XRL   A,R3
0125    ...
```

Before execution: A= C3h (11000011 Bin.)
R3= 55h (01010101 Bin.)
After execution: A= 96h (10010110 Bin.)

XCHD A,@Ri - Exchanges the low-order nibble indirect RAM with the accumulator
Ri: register R0 or R1
A: accumulator
Description: This instruction interchanges the low-order nibbles (bits 0-3) of the accumulator with the low-order nibbles of the indirectly addressed register of RAM. High-order nibbles of the accumulator and register are not affected. This instruction is mainly used when operating with BCD values. As it is indirect addressing, the register address is stored in the register Ri (R0 or R1).
Syntax: XCHD A,@Ri;
Byte: 1 (instruction code);
STATUS register flags: No flags are affected;
EXAMPLE:

```
0322    ...
0323    (Label)   XCHD    A,@R0
0324    ...
```

Register address: SUM=E3
Before execution: R0=E3 SUM=29h A=A8h,
After execution: A=A9h, SUM=28h



XRL A,@Ri - Exclusive OR indirect RAM to the accumulator
Ri: Register R0 or R1
A: accumulator
Description: Instruction performs exclusive OR operation between the accumulator and the indirectly addressed register. As it is indirect addressing, the register address is stored in the Ri register (R0 or R1). The result is stored in the accumulator.
Syntax: XRL A,@Ri;
Byte: 1 (instruction code);
STATUS register flags: No flags are affected;
EXAMPLE:

```
0123    ...
0124    (Label)  XRL   A,@R1
0125    ...
```

Register address: TEMP=FAh, R1=FAh
Before execution: TEMP= C2h (11000010 Bin.)
A= 54h (01010100 Bin.)
After execution: A= 96h (10010110 Bin.)

XRL A,direct - Exclusive OR direct byte to the accumulator
Direct: Arbitrary register with address 0-255 (0-FFh)
A: accumulator
Description: Instruction performs exclusive OR operation between the accumulator and the direct byte. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh (0-127 dec.). The result is stored in the accumulator.
Syntax: XRL A,direct;
Bytes: 2 (instruction code, direct byte address);
STATUS register flags: No flags are affected;

EXAMPLE:

```
0123     ...
0124     (Label)   XRL    A,LOG
0125     ...
```

Before execution: A= C2h (11000010 Bin.)
LOG= 54h (01010100 Bin.)
After execution: A= 96h (10010110 Bin.)

XRL direct,A - Exclusive OR accumulator to the direct byte
Direct: arbitrary register with address 0-255 (0-FFh)
A: accumulator
Description: Instruction performs exclusive OR operation between the direct byte and the accumulator. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh (0-127 dec.). The result is stored in the register.
Syntax: XRL direct,A;
Bytes: 2 (instruction code, direct byte address);
STATUS register flags: No flags affected;
EXAMPLE:

```
0123     ...
0124     (Label)   XRL    TEMP,A
0125     ...
```

Before execution: TEMP= C2h (11000010 Bin.)
A= 54h (01010100 Bin.)
After execution: A= 96h (10010110 Bin.)
XRL A,#data - Exclusive OR immediate data to the accumulator
Data: constant in the range of 0-255 (0-FFh)
A: accumulator
Description: Instruction performs exclusive OR operation between the accumulator and the immediate data. The result is stored in the accumulator.
Syntax: XRL A,#data;
Bytes: 2 (instruction code, data);
STATUS register flags: No flags are affected;

EXAMPLE:

```
0123    ...
0124    (Label)  XRL   A,11
0125    ...
```

Before execution: A= C2h (11000010 Bin.)
X= 11h (00010001 Bin.)
After execution: A= D3h (11010011 Bin.)

XRL direct,#data - Exclusive OR immediate data to direct byte
Direct: arbitrary register with address 0-255 (0-FFh)
Data: constant in the range of 0-255 (0-FFh)
Description: Instruction performs exclusive OR operation between the immediate data and the direct byte. As it is direct addressing, the register can be any SFRs or general-purpose register with address 0-7Fh (0-127 dec.). The result is stored in the register.
Syntax: XRL direct,#data;
Bytes: 3 (instruction code, direct byte address, data);
STATUS register flags: No flags affected;
EXAMPLE:

```
0123    ...
0124    (Label)  XRL   TEMP,#12
0125    ...
```

Before execution: TEMP= C2h (11000010 Bin.)
X=12h (00010010 Bin.)
After execution: A= D0h (11010000 Bin.)

1. To initialise data to registers and memory using Immediate, Register, Direct and Indirect addressing modes.

To initialise the following Registers and memory contents as specified below.
(R0) = 9A
(B) = 1F
(DPTR) = 4500
(R6) = 9A

PROGRAM:

MOV R0,#9A ; (R0) = 9A - Immediate
MOV B,#1F ; (B) = 1F - Immediate

```
MOV DPTR,#4500 ; (DPL) = 00;
; (DPH) = 45 - Immediate
MOV R6,A ; (R6) <)) (A) - Register
MOV A,20 ; (A) <)) (20) - Direct
MOVX A,@DPTR ; (A) <)) ((DPTR)) - Indirect
XCH A,R0 ; (A) <))> (R0) - Register
HERE: SJMP HERE
```

2. To move a block (array) of data to another block to show how to employ indexed addressing to access data stored in the form of a continuous array in memory.

PROGRAM:

```
MOV DPTR,#4500
MOVX A,@DPTR
MOV R0,A
INC DPTR
MOV R1,DPL
MOV R2,DPH
MOV DPTR,#4510
REPT: PUSH DPL
PUSH DPH
MOV DPL,R1
MOV DPH,R2
MOVX A,@DPTR
INC DPTR
MOV R1,DPL
MOV R2,DPH
POP DPH
POP DPL
MOVX @DPTR,A
INC DPTR
DJNZ R0,REPT
HERE: SJMP HERE
```

3. To perform 16-bit addition of two 16-bit data using immediate addressing and store the result in memory.

The program is to add the 16-bit data 1234 with the data 5678 and store the result at the locations 4150 and 4151 using immediate addressing.

RESULT: [4150] = AC (LSB); [4151] = 68 (MSB).

PROGRAM:

```
CLR C
MOV A,#DATAL1
ADDC A,#DATAL2
MOV DPTR,#4150
MOVX @DPTR,A
INC DPTR
MOV A,#DATAM1
ADDC A,#DATAM2
MOVX @DPTR,A
HLT: SJMP HLT
```

4. To perform subtraction of two 8-bit data using immediate addressing and store the result in memory.

EXAMPLE:

Sample data: DATA1 = 20

DATA2 = 10

Result: [4500] = 10

PROGRAM:

```
CLR C
MOV A,#DATA1
SUBB A,#DATA2
MOV DPTR,#4500
MOVX @DPTR,A
HERE: SJMP HERE
```

5. Subtract the contents of location 4500 from the contents of location 4501 and store the result at location 4600.

EXAMPLE:

Sample data: [4500] = 56

[4501] = 6A

Result: [4600] = 14

PROGRAM:

```
MOV A,#DATA1
MOV B,#DATA2
MUL AB
MOV DPTR,#4500
MOVX @DPTR,A
```

```
INC DPTR
MOV A,B
MOVX @DPTR,A
HERE: SJMP HERE
```

6. Obtain the square of a number stored in memory.

Sample: [4500] = 0A
Result: [4600] = 64

7. To obtain the one's and two's complement of an 8-bit number in Register A.

The number whose complements are needed is in register A. The result is in location 4200 and 4201.
Sample data: DATA = CC
Result: [4200] = 33 - One's Complement
[4201] = 34 - Two's Complement

PROGRAM:

```
MOV A,#DATA
CPL A
MOV DPTR,#4200
MOVX @DPTR,A
INC A
INC DPTR
MOVX @DPTR,A
HERE: SJMP HERE
```

8. To set specific bits of an 8-bit number.

THEORY:

Setting bits can be done by ORing that particular bit by 1. The following program explains how to set a particular bit in an 8-bit number by using the ORL instruction of 8051.

EXAMPLE:

In the following program, the contents of the Accumulator is ORed with an immediate data accordingly to set the required bits.
Sample Data : DATA1 = 2F
DATA2 = 45
Result : [4500] = 6F

PROGRAM:

```
MOV A,#DATA1
ORL A,#DATA2
MOV DPTR,#4500
```

```
MOVX @DPTR,A
HERE: SJMP HERE
```

9. To mask bits 0 and 7 of an 8-bit number and store the result in memory.

THEORY:

The ANL instruction of 8051 can be used to reset bits. ANDing with zero produces a cleared bit. ANDing with one does not change the status of the bit.

EXAMPLE:

To mask bits 0 and 7, the 8-bit data has to be ANDed with 7E, which is 01111110 in binary.

Sample data: DATA1 = 87

DATA2 = 7E

Result: [4500] = 06

PROGRAM:

```
MOV A,#DATA1
ANL A,#DATA2
MOV DPTR,#4500
MOVX @DPTR,A
HERE: SJMP HERE
```

10. To perform various Arithmetic Operations using Bit addressable Special Function Registers of 8051.

THEORY:

Specific bits of various bit addressable registers can be ANDed, ORed with Carry flag, CLeaRed, ComPLemented, SET and can be moved to Carry flag.

EXAMPLE:

In the following program, the contents of C(carry) is ANDed with ACC.7, ORed with IE.2. Also, the contents of SCON.5 is set and that of SCON.1 is cleared and the contents of SCON.1 is moved to the carry flag.

PROGRAM:

```
MOV A,#0FFH
SETB C
ANL C,ACC.7
ORL C,IE.2
SETB SCON.5
CLR SCON.1
MOV C,SCON.1
HLT: SJMP HLT
```

11. To add the numbers of an 8-bit array, the array length being the first element in the array and store the result in memory.

THEORY:

The numbers are stored in consecutive locations in memory. Addition has to be performed for the number of times decided by the length of the array. Since the sum may exceed 8-bits, it is necessary to monitor the carry flag status. Indexed addressing employing the 16-bit Base Register DPTR is used in this program.

EXAMPLE:

The array starts at 4201. The address location 4200 contains the number of elements in that array. The result is to be stored at 4500 and 4501.

Sample Data: [4200] = 03
[4201] = 01
[4202] = 02
[4203] = 03
Result: [4500] = 00 (MSB)
[4501] = 06 (LSB)

PROGRAM:

```
MOV DPTR,#4200 ;DPTR points to base of array
MOVX A,@DPTR
MOV R0,A ;R0 has the length of array
MOV B,#0
MOV R1,B
ADD: CLR C
INC DPTR ;Increment the Pointer
MOVX A,@DPTR ;Get the element
ADD A,B ;Do the partial addition
MOV B,A
JNC NC ;Check for carry
INC R1 ;If carry set, then inc. MSB
NC: DJNZ R0,ADD ;Repeat it till count = 0
MOV DPTR,#4500
MOV A,R1 ;Store the MSB first
MOVX @DPTR,A
INC DPTR
MOV A,B ;Store the LSB at 4501
MOVX @DPTR,A
HLT: SJMP HLT
```

12. To add two numbers that are more than 8-bit in length, the length being given as the number of 8-bits in that number.

THEORY:

The multi-byte addition program adds only in sets of 8-bits. The LSD of the two numbers are added first. Now, the next set of 8-bits is added, taking into consideration the status of carry due to the previous addition. Addition is done till the length of the number specified becomes zero.

EXAMPLE:

The length of both the numbers is 3 bytes. The length is loaded in R4. The first data is stored from 4200 with the LSB taking the first position. The second number starts at 4210.

Data: [4200] = 29 - First number
[4201] = A4
[4202] = 02
[4210] = FB - Second number
[4211] = 37
[4212] = 28

```
02 A4 29 (+)
28 37 FB
--------------
2A DC 24
--------------
```

Result: [4500] = 24
[4501] = DC
[4502] = 2A

PROGRAM:

```
CLR C ;Carry flag = 0
MOV R4,#3 ;Count
MOV DPTR,#4200 ;Pointer for first no.
MOV R0,DPL
MOV R1,DPH
MOV DPTR,#4210 ;Pointer for second no.
MOV R2,DPL
MOV R3,DPH
MOV DPTR,#4500 ;Pointer to store result
REPT: PUSH DPL
PUSH DPH
MOV DPL,R0
MOV DPH,R1
MOVX A,@DPTR
MOV B,A
```

```
INC DPTR
MOV R0,DPL
MOV R1,DPH
MOV DPL,R2
MOV DPH,R3
MOVX A,@DPTR
ADDC A,B
INC DPTR
MOV R2,DPL
MOV R3,DPH
POP DPH
POP DPL
MOVX @DPTR,A
INC DPTR
DJNZ R4,REPT
HLT: SJMP HLT
```

13. To divide an 8-bit number by another 8-bit number and store the quotient and remainder in memory.

THEORY:

The 8051 has a "DIV" instruction unlike many other 8-bit processors. DIV instruction divides the unsigned eight-bit integer in A by the unsigned 8-bit integer in register B. The Accumulator receives the integer part of the quotient and register B receives the integer remainder. The carry and OV flags will be cleared.

EXAMPLE:

Let the divisor and dividend be in registers B and A respectively.

Data : DATA1 = 65 - Dividend

DATA2 = 08 - Divisor

Result : [4500] = 0C - Quotient

[4501] = 05 – Remainder

PROGRAM:

```
MOV A,#DATA1 ;Load Acc. with Dividend.
MOV B,#DATA2 ;Load Reg. B with Divisor.
DIV AB
MOV DPTR,#4500
MOVX @DPTR,A ;Store quotient at 4500
INC DPTR
MOV A,B ;Store remainder at 4501
```

```
MOVX @DPTR,A
HLT: SJMP HLT
```

14. To convert the ASCII number in the accumulator to its equivalent decimal number.

THEORY:

Conversion of an ASCII number to decimal is very simple because all the decimal numbers form a sequence in ASCII. Any ASCII number can be converted to decimal just by subtracting 30 from it.

ASCII Number(Hex) Decimal Equivalent

ASCII Number(Hex)	Decimal Equivalent
30	00
31	01
32	02
33	03
34	04
35	05
36	06
37	07
38	08
39	09

EXAMPLE:

Let the ASCII number be in Register A and the result be stored at 4500.

Data : DATA = 35

Result : [4500] = 05

Data : DATA =3B

Result : [4500]= FF

PROGRAM:

```
MOV DPTR,#4500
MOV  A,#DATA  ;Get ASCIInumber
CLR  C  ;Clear carryflag
SUBB  A,#30  ;Subtract [A] by30
CLR  C
SUBB  A,#0A  ;Check if the no. is decimal
JC  STR
MOV  A,#0FF  ;Else storeFFat 4500
SJMP  L1
STR: ADD  A,#0A
L1:  MOVX  @DPTR,A ;Storetheresult if valid
```

```
HLT:  SJMP  HLT
```

15. Split the contents of 4500 into two nibbles (4-bitnumbers) and store the HN at 4501 and the LN at 4502.

THEORY:

The data is fetched from memory and SWAP instruction of 8051 isused which will interchange The low and high-order nibbles of the Accumulator. The HN of the result is masked off and is stored in memory. Then, the HN in the original data is masked off and is stored in memory.

EXAMPLE:

The contents of 4500 is 56. The result stored at 4501 is 05 and at 4502 is 06.

Data : [4500] = 56

Result : [4501] = 05

[4502] = 06

PROGRAM:

```
MOV DPTR,#4500
MOVX  A,@DPTR ;Get the data from memory
MOV  B,A  ;B has the original data
SWAP A
ANL  A,#0F
INC  DPTR
MOVX @DPTR,A ;Store HN
MOV  A,B          ;Get original data
ANL  A,#0F  ;Mask HN of original data
INC  DPTR
MOVX @DPTR,A; Store LN
HLT: SJMP HLT
```

16. To obtain the decimal equivalent of an 8-bit hex number stored in memory.

THEORY:

In this program, the hex number is converted to its equivalent decimal number. The algorithm followed is very simple. The hex number to be converted is brought to the accumulator and is divided by100 D to find the number of hundreds in it. DIV instruction of 8051 isused in this program. The remainder is now divided by10 D to count the number of tens in it. Finally, the remainder obtained from the above division gives the number of units in the given hex number. The result is stored in memory in the unpacked form.

EXAMPLE:

Let us work with FF.

Data : [4500] = FF

Result : [4501] = 02

[4502] = 05
[4503] = 05

PROGRAM:

```
MOV DPTR,#4500
MOVX  A,@DPTR  ;Get thedata
MOV  B,#64
DIV  AB  ;Find no. of100s
MOV  DPTR,#4501
MOVX  @DPTR,A
MOV  A,B
MOV  B,#0A
DIV  AB  ;Find no. of10s
INC  DPTR
MOVX  @DPTR,A
INC  DPTR
MOV  A,B
MOVX  @DPTR,A
HLT: SJMP  HLT
```

17. To convert BCD digits in memory to the equivalent hex number.

THEORY:

Considering that out of the two unpacked BCD digits at 4200 and 4201, the digit at 4200 is the MSD, the logic is to multiply this by0A (10D) and then add the LSD at 4201 to the product.

EXAMPLE:

The digits are at 4200 and 4201 and let the result be stored at 4202.
Data : [4200] = 03
[4201] = 06
Result : [4202] = 24

PROGRAM:

```
MOV DPTR,#4200
MOVX A,@DPTR
MOV  B,#0AH
MUL  AB
MOV  B,A
INC  DPTR
MOVX A,@DPTR
ADD  A,B
INC  DPTR
MOVX @DPTR,A
HLT: SJMP HLT
```

18. To find the biggest number in an arrayof 8-bit unsigned numbers of predetermined length.

THEORY:

To find the Biggest number in any given array, the contents of the array must be compared with an arbitrary Biggest number. In this experiment, since all numbers are said to be unsigned 8-bit numbers, let Internal memory location (say 40H) has the biggest number i.e. zero. Now compare the first number with Internal memory location. I f it is greater, move it to Internal memory location. Further comparison is with this biggest number and this comparison is done till the end of the array. Now the biggest number in Internal memory location is stored in memory as the result.

EXAMPLE:

The length of the array is specified in the Register R5.The array itself starts at 4200. The largest number of the array is stored at location 420A.

Data : [4200] = 05
[4201] = 67
[4202] = 76
[4203] = 89
[4204] = 98
[4205] = 49
[4206] = 45
[4207] = 9F
[4208] = 57
[4209] = 7A
Result :[420A] = 9F

PROGRAM:

```
MOV  DPTR,#4200
MOV  40H,#00
MOV  R5,#0AH
LOOP2: MOVX  A,@DPTR
CJNE  A,40H,LOOP1
LOOP3:  INC  DPTR
DJNZ  R5,LOOP2
MOV  A,40H
MOVX @DPTR,A
HLT: SJMP  HLT
LOOP1: JC  LOOP3
MOV  40H,A
SJMP LOOP3
```

19. To arrange an array of 8-bit unsigned numbers of known length in ascending order.

THEORY:

The sorting technique used here is relatively simple. First consider the first two numbers of the array. See if this pair of numbers is out of order. Similarly examine successively each pair of numbers in the array. I f any pair is out of order interchange the numbers in the pair. Perform this step for count times, count being one less than array length.

EXAMPLE:

The length of the array is in register R3. The array starts from 4501.

Data : [4500] = 2A
[4501] = 1C
[4502] = F9
[4503] = 88
[4504] = 76

Result : [4500] = 1C
[4501] = 2A
[4502] = 76
[4503] = 88
[4504] = F9

PROGRAM:

```
MOV R3,#4 ;Count-1
MOV   R4,#4
MOV   DPTR,#4500
REPT1: MOV  R5,DPL
MOV   R6,DPH
MOVX  A,@DPTR
MOV  B,A
REPT: INC  DPTR
MOVX A,@DPTR
MOV  R0,A
CLR  C
SUBB  A,B
JNC  CHKNXT
EXCH: PUSH  DPL
PUSH  DPH
MOV  DPL,R5
MOV  DPH,R6
MOV  A,R0
MOVX  @DPTR,A
POP  DPH
POP  DPL
```

```
MOV  A,B
MOVX  @DPTR,A
MOV  B,R0
CHKNXT: DJNZ  R3,REPT
DEC   R4
MOV  A,R4
MOV  R3,A
INC  R 4
MOV  DPL,R5
MOV  DPH,R6
INC   DPTR
DJNZ  R4,REPT1
HLT: SJMP  HLT
```

20. To show the stack pointer contents and the contents of the stack itself areas depicted below after the execution of each instruction

Instruction	Register Contents			Stack Contents
	A	B	DPL	
MOV SP,#10	XX	XX	XX	SP initialised to 10 of internal RAM
MOV A,#88	88	XX	XX	A has 88
MOV B,#67	88	67	XX	B has 67
MOV DPL,#43	88	67	43	DPL has 43
PUSH A	88	67	43	SP incremented by 1 and (A) is stored here
PUSH B	88	67	43	SP incremented by 1 and (B) is stored here
PUSH DPL	88	67	43	SP Incremented by 1 and (DPL) is stored here.

STACK CONTENTS AFTER INSTRUCTION PUSII A

10	XX
11	88

STACK CONTENTS AFTER INSTRUCTION PUSII B

10	XX
11	88
12	67

STACK CONTENTS AFTER INSTRUCTION PUSH DPL

10	XX
11	88
12	67
13	43

XX = USER DEFINED DATA

21. To write a program that generates delay of 0.5ms with the 8051 at the Clock rate of 12MHz.

THEORY:

Timing intervals or delays can be generated by either of the following three modes.

i. Hardware delays employing monostable multivibrators which produce a pulse output as decided by the trigger.

ii. Programmable timers which are the combination of hardware and software.

iii. Software loops employing register(s) as a counter

PROGRAM:

```
MOV R0,#0FB
REPT:  DJNZ  R0,REPT
```

DELAYCALCULATION:

The MOV R0,#COUNTinstruction takes execution time of 1 Microsec and DJNZ instruction takes 2Microsecs.

The count to be loaded into register R0 is as calculated below:

1 + 2 x count = Execution time

Required Delay = 500 microseconds.

500 = 1 + 2 x count

$$\text{Count} = \frac{500 - 1}{2}$$

= 249.5 D.

In hex the count = FB in hex.

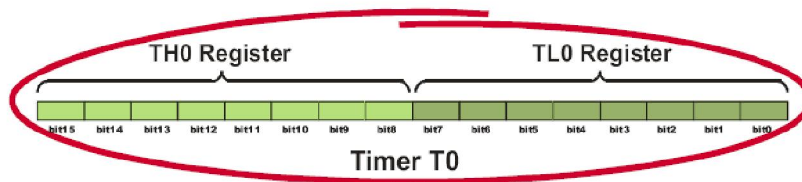
The value FB has to be loaded in register R0 to obtain a delay of 0.5 millisecond.

COUNTERS AND TIMERS

As you already know, the microcontroller oscillator uses quartz crystal for its operation. As the frequency of this oscillator is precisely defined and very stable, pulses it generates are always of the same width, which makes them ideal for time measurement. Such crystals are also used in quartz watches. In order to measure time between two events it is sufficient to count up pulses coming from this oscillator. That is exactly what the timer does. If the timer is properly programmed, the value stored in its register will be incremented (or decremented) with each coming pulse, i.e. once per each machine cycle. A single machine-cycle instruction lasts for 12 quartz oscillator periods, which means that by embedding quartz with oscillator frequency of 12MHz, a number stored in the timer register will be changed million times per second, i.e. each microsecond. The 8051 microcontroller has 2 timers/counters called T0 and T1. As their names suggest, their main purpose is to measure time and count external events. Besides, they can be used for generating clock pulses to be used in serial communication, so called Baud Rate.

TIMER T0

As seen in figure below, the timer T0 consists of two registers – TH0 and TL0 representing a low and a high byte of one 16-digit binary number.



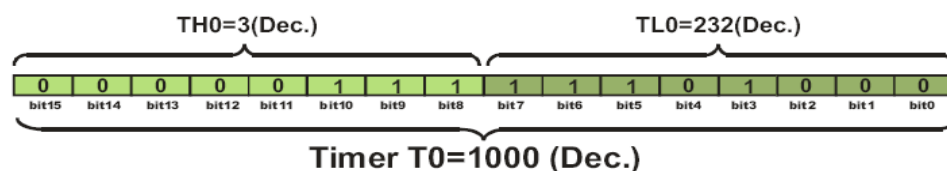
Accordingly, if the content of the timer T0 is equal to 0 (T0=0) then both registers it consists of will contain 0. If the timer contains for example number 1000 (decimal), then the TH0 register (high byte) will contain the number 3, while the TL0 register (low byte) will contain decimal number 232.

Formula used to calculate values in these two registers is very simple:

TH0 × 256 + TL0 = T

Matching the previous example it would be as follows:

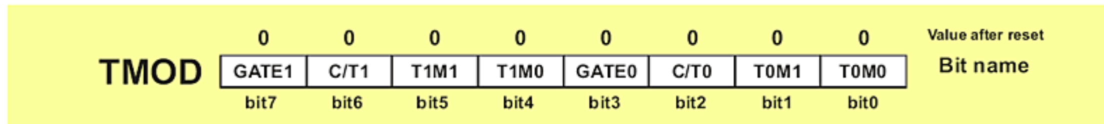
3 × 256 + 232 = 1000



Since the timer T0 is virtually 16-bit register, the largest value it can store is 65 535. In case of exceeding this value, the timer will be automatically cleared and counting starts from 0. This condition is called an overflow. Two registers TMOD and TCON are closely connected to this timer and control its operation.

TMOD REGISTER (TIMER MODE)

The TMOD register selects the operational mode of the timers T0 and T1. As seen in figure below, the low 4 bits (bit0 - bit3) refer to the timer 0, while the high 4 bits (bit4 - bit7) refer to the timer 1. There are 4 operational modes and each of them is described herein.



Bits of this register have the following function:

- ***GATE1*** enables and disables Timer 1 by means of a signal brought to the INT1 pin (P3.3):
  - **1** - Timer 1 operates only if the INT1 bit is set.
  - **0** - Timer 1 operates regardless of the logic state of the INT1 bit.

- ***C/T1*** selects pulses to be counted up by the timer/counter 1:
  - **1** - Timer counts pulses brought to the T1 pin (P3.5).
  - **0** - Timer counts pulses from internal oscillator.

- ***T1M1,T1M0*** These two bits select the operational mode of the Timer 1.

T1M1	T1M0	MODE	DESCRIPTION
0	0	0	13-bit timer
0	1	1	16-bit timer
1	0	2	8-bit auto-reload
1	1	3	Split mode

- ***GATE0*** enables and disables Timer 1 using a signal brought to the INT0 pin (P3.2):
  - 1 - Timer 0 operates only if the INT0 bit is set.
- 0 - Timer 0 operates regardless of the logic state of the INT0 bit.

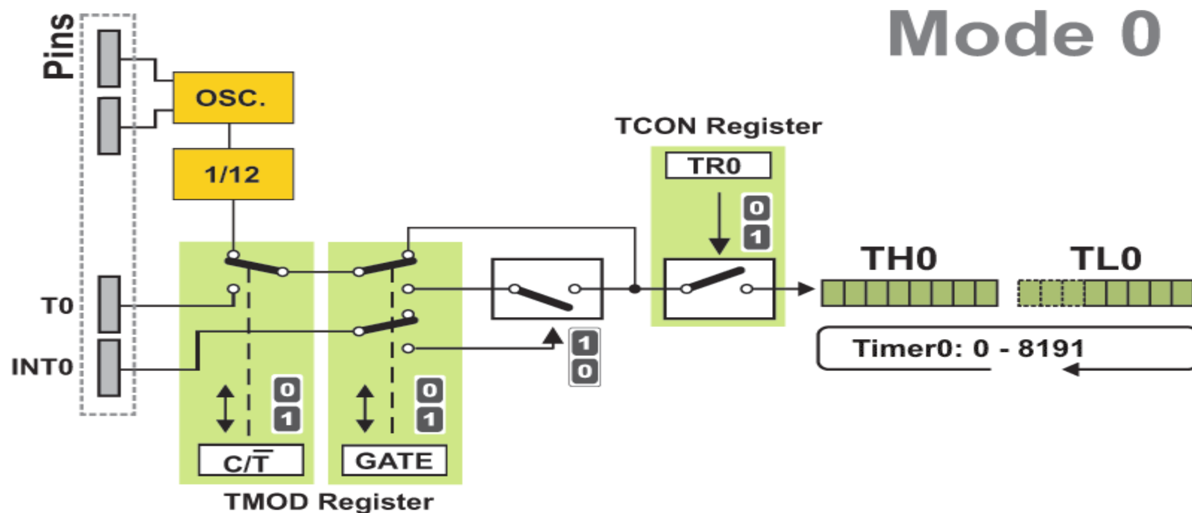
- ***C/T0*** selects pulses to be counted up by the timer/counter 0:
  - **1** - Timer counts pulses brought to the T0 pin (P3.4).
  - **0** - Timer counts pulses from internal oscillator.

- ***T0M1,T0M0*** These two bits select the oprtaional mode of the Timer 0.

T0M1	T0M0	MODE	DESCRIPTION
0	0	0	13-bit timer
0	1	1	16-bit timer
1	0	2	8-bit auto-reload
1	1	3	Split mode

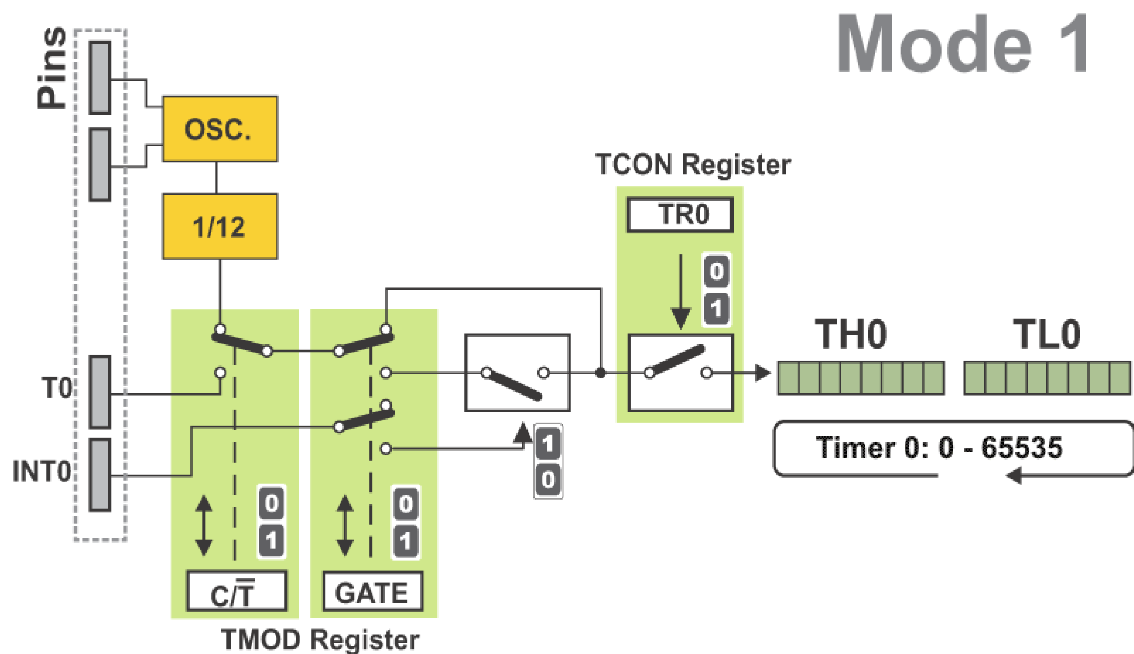
TIMER 0 IN MODE 0 (13-BIT TIMER)

This is one of the rarities being kept only for the purpose of compatibility with the previuos versions of microcontrollers. This mode configures timer 0 as a 13-bit timer which consists of all 8 bits of TH0 and the lower 5 bits of TL0. As a result, the Timer 0 uses only 13 of 16 bits. How does it operate? Each coming pulse causes the lower register bits to change their states. After receiving 32 pulses, this register is loaded and automatically cleared, while the higher byte (TH0) is incremented by 1. This process is repeated until registers count up 8192 pulses. After that, both registers are cleared and counting starts from 0.



TIMER 0 IN MODE 1 (16-BIT TIMER)

Mode 1 configures timer 0 as a 16-bit timer comprising all the bits of both registers TH0 and TL0. That's why this is one of the most commonly used modes. Timer operates in the same way as in mode 0, with difference that the registers count up to 65 536 as allowable by the 16 bits.

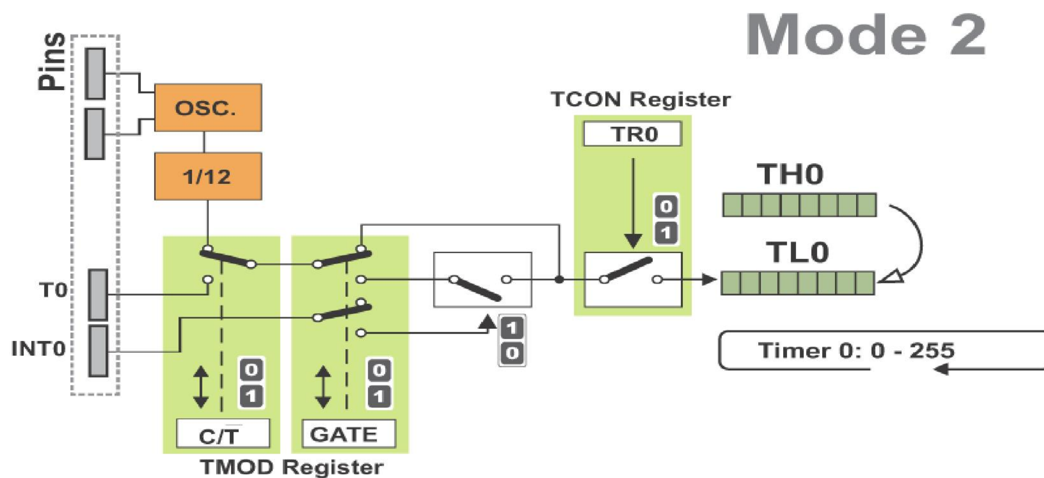


TIMER 0 IN MODE 2 (AUTO-RELOAD TIMER)

Mode 2 configures timer 0 as an 8-bit timer. Actually, timer 0 uses only one 8-bit register for counting and never counts from 0, but from an arbitrary value (0-255) stored in another (TH0) register.

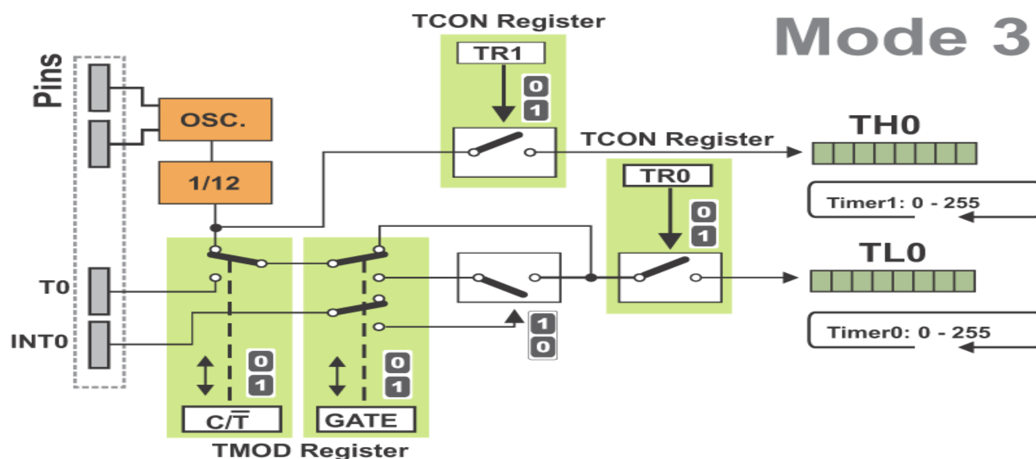
The following example shows the advantages of this mode. Suppose it is necessary to constantly count up 55 pulses generated by the clock.

If mode 1 or mode 0 is used, It is necessary to write the number 200 to the timer registers and constantly check whether an overflow has occured, i.e. whether they reached the value 255. When it happens, it is necessary to rewrite the number 200 and repeat the whole procedure. The same procedure is automatically performed by the microcontroller if set in mode 2. In fact, only the TL0 register operates as a timer, while another (TH0) register stores the value from which the counting starts. When the TL0 register is loaded, instead of being cleared, the contents of TH0 will be reloaded to it. Referring to the previous example, in order to register each 55th pulse, the best solution is to write the number 200 to the TH0 register and configure the timer to operate in mode 2.



TIMER 0 IN MODE 3 (SPLIT TIMER)

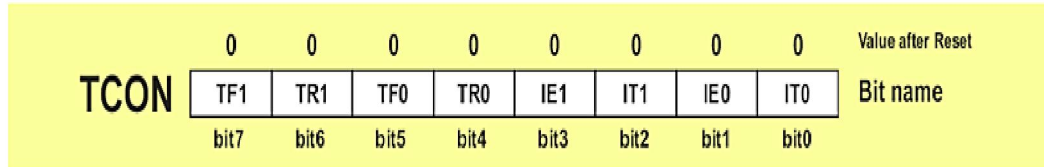
Mode 3 configures timer 0 so that registers TL0 and TH0 operate as separate 8-bit timers. In other words, the 16-bit timer consisting of two registers TH0 and TL0 is split into two independent 8-bit timers. This mode is provided for applications requiring an additional 8-bit timer or counter. The TL0 timer turns into timer 0, while the TH0 timer turns into timer 1. In addition, all the control bits of 16-bit Timer 1 (consisting of the TH1 and TL1 register), now control the 8-bit Timer 1. Even though the 16-bit Timer 1 can still be configured to operate in any of modes (mode 1, 2 or 3), it is no longer possible to disable it as there is no control bit to do it. Thus, its operation is restricted when timer 0 is in mode 3.



The only application of this mode is when two timers are used and the 16-bit Timer 1 the operation of which is out of control is used as a baud rate generator.

TIMER CONTROL (TCON) REGISTER

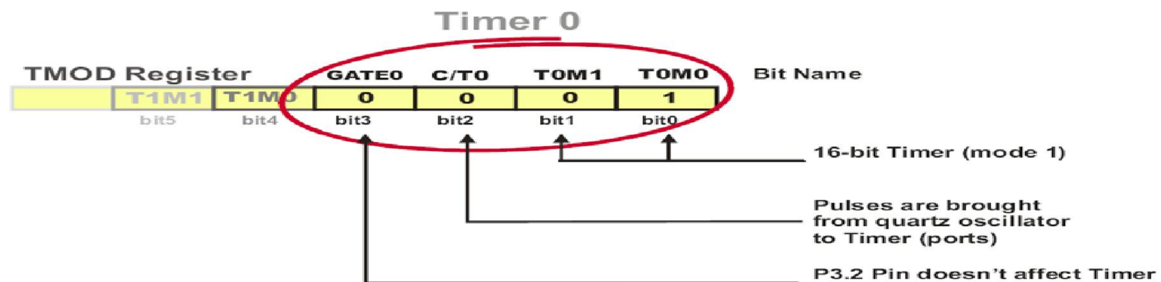
TCON register is also one of the registers whose bits are directly in control of timer operation. Only 4 bits of this register are used for this purpose, while rest of them is used for interrupt control to be discussed later.



- ***TF1*** bit is automatically set on the Timer 1 overflow.
- ***TR1*** bit enables the Timer 1.
  - **1** - Timer 1 is enabled.
  - **0** - Timer 1 is disabled.
- ***TF0*** bit is automatically set on the Timer 0 overflow.
- ***TR0*** bit enables the timer 0.
  - **1** - Timer 0 is enabled.
  - **0** - Timer 0 is disabled.

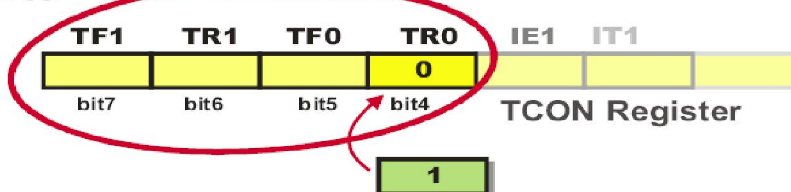
HOW TO USE THE TIMER 0 ?

In order to use timer 0, it is first necessary to select it and configure the mode of its operation. Bits of the TMOD register are in control of it:



Referring to figure above, the timer 0 operates in mode 1 and counts pulses generated by internal clock the frequency of which is equal to 1/12 the quartz frequency.

Turn on the timer:



The TR0 bit is set and the timer starts operation. If the quartz crystal with frequency of 12MHz is embedded then its contents will be incremented every microsecond. After 65.536 microseconds, the both registers the timer consists of will be loaded. The microcontroller automatically clears them and the timer keeps on repeating procedure from the beginning until the TR0 bit value is logic zero (0).

HOW TO 'READ' A TIMER?

Depending on application, it is necessary either to read a number stored in the timer registers or to register the moment they have been cleared.

I. It is extremely simple to read a timer by using only one register configured in mode 2 or 3. It is sufficient to read its state at any moment.

II. It is somehow complicated to read a timer configured to operate in mode 2. Suppose the lower byte is read first (TL0), then the higher byte (TH0). The result is:

TH0 = 15 TL0 = 255

Everything seems to be ok, but the current state of the register at the moment of reading was:

TH0 = 14 TL0 = 255

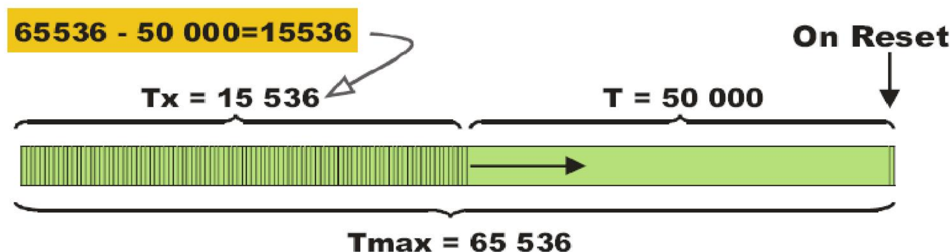
In case of negligence, such an error in counting (255 pulses) may occur for not so obvious but quite logical reason. The lower byte is correctly read (255), but at the moment the program counter was about to read the higher byte TH0, an overflow occurred and the contents of both registers have been changed (TH0: 14→15, TL0: 255→0). This problem has a simple solution. The higher byte should be read first, then the lower byte and once again the higher byte. If the number stored in the higher byte is different then this sequence should be repeated. It's about a short loop consisting of only 3 instructions in the program.

There is another solution as well. It is sufficient to simply turn the timer off while reading is going on (the TR0 bit of the TCON register should be cleared), and turn it on again after reading is finished.

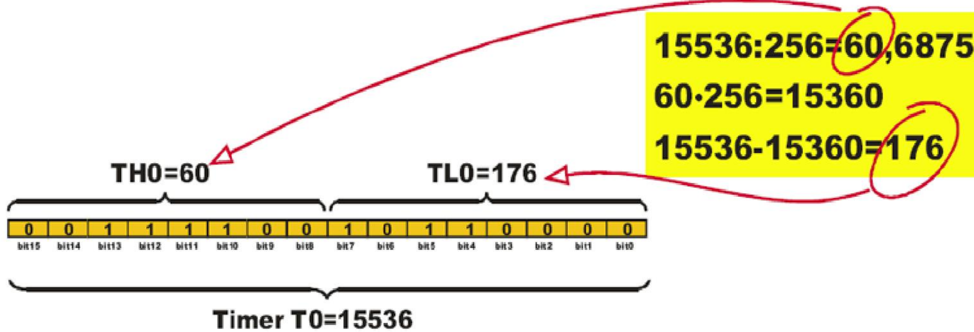
TIMER 0 OVERFLOW DETECTION

Usually, there is no need to constantly read timer registers. It is sufficient to register the moment they are cleared, i.e. when counting starts from 0. This condition is called an overflow. When it occurrs, the TF0 bit of the TCON register will be automatically set. The state of this bit can be constantly checked from within the program or by enabling an interrupt which will stop the main program execution when this bit is set. Suppose it is necessary to provide a program delay of 0.05 seconds (50 000 machine cycles), i.e. time when the program seems to be stopped:

First a number to be written to the timer registers should be calculated:

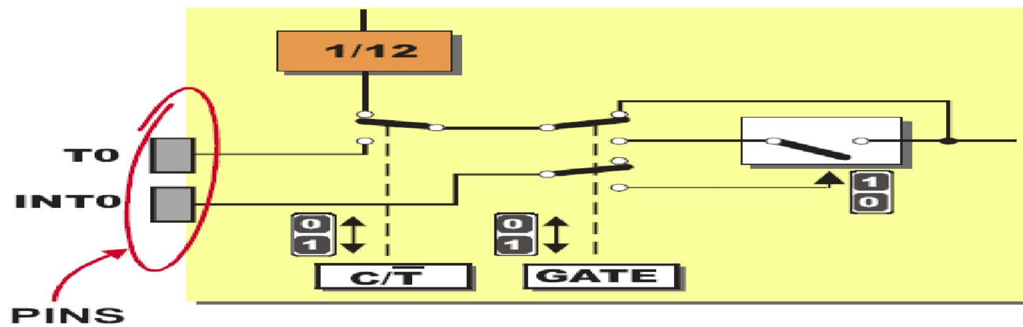


Then it should be written to the timer registers TH0 and TL0:



When enabled, the timer will resume counting from this number. The state of the TF0 bit, i.e. whether it is set, is checked from within the program. It happens at the moment of overflow, i.e. after exactly 50.000 machine cycles or 0.05 seconds.

HOW TO MEASURE PULSE DURATION?



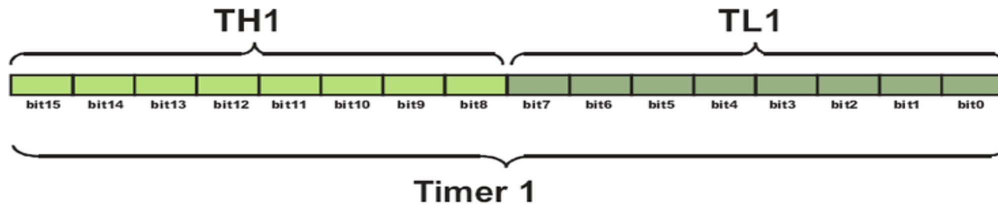
Suppose it is necessary to measure the duration of an operation, for example how long a device has been turned on? Look again at the figure illustrating the timer and pay attention to the function of the GATE0 bit of the TMOD register. If it is cleared then the state of the P3.2 pin doesn't affect timer operation. If GATE0 = 1 the timer will operate until the pin P3.2 is cleared. Accordingly, if this pin is supplied with 5V through some external switch at the moment the device is being turned on, the timer will measure duration of its operation, which actually was the objective.

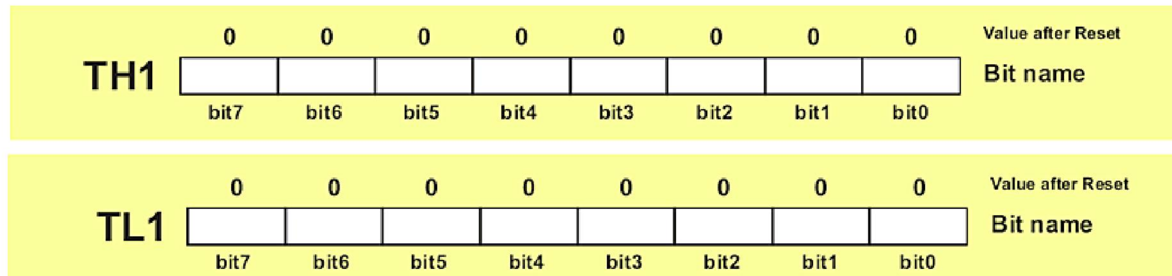
HOW TO COUNT UP PULSES?

Similarly to the previous example, the answer to this question again lies in the TCON register. This time it's about the C/T0 bit. If the bit is cleared the timer counts pulses generated by the internal oscillator, i.e. measures the time passed. If the bit is set, the timer input is provided with pulses from the P3.4 pin (T0). Since these pulses are not always of the same width, the timer cannot be used for time measurement and is turned into a counter, therefore. The highest frequency that could be measured by such a counter is 1/24 frequency of used quartz-crystal.

TIMER 1

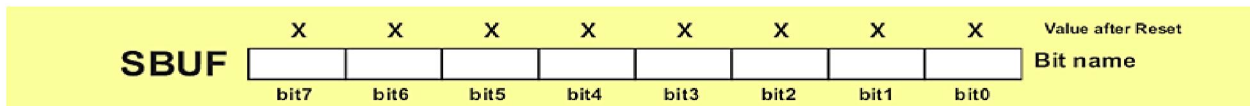
Timer 1 is identical to timer 0, except for mode 3 which is a hold-count mode. It means that they have the same function, their operation is controlled by the same registers TMOD and TCON and both of them can operate in one out of 4 different modes.





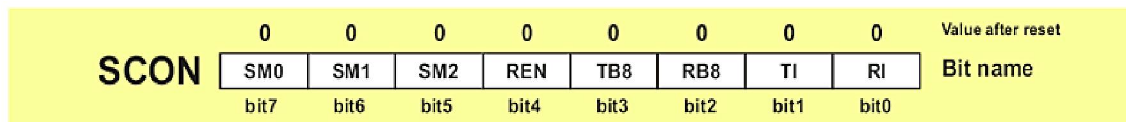
UART (UNIVERSAL ASYNCHRONOUS RECEIVER AND TRANSMITTER)

One of the microcontroller features making it so powerful is an integrated UART, better known as a serial port. It is a full-duplex port, thus being able to transmit and receive data simultaneously and at different baud rates. Without it, serial data send and receive would be an enormously complicated part of the program in which the pin state is constantly changed and checked at regular intervals. When using UART, all the programmer has to do is to simply select serial port mode and baud rate. When it's done, serial data transmit is nothing but writing to the SBUF register, while data receive represents reading the same register. The microcontroller takes care of not making any error during data transmission.



Serial port must be configured prior to being used. In other words, it is necessary to determine how many bits is contained in one serial "word", baud rate and synchronization clock source. The whole process is in control of the bits of the SCON register (Serial Control).

SERIAL PORT CONTROL (SCON) REGISTER

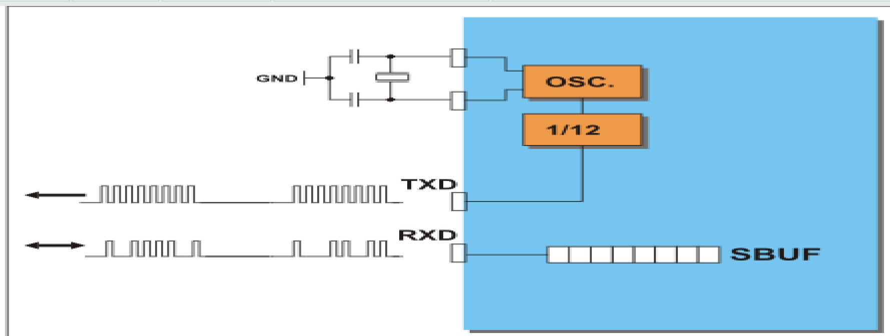


- ***SM0*** - Serial port mode bit 0 is used for serial port mode selection.
- ***SM1*** - Serial port mode bit 1.

- ***SM2*** **-** Serial port mode 2 bit, also known as multiprocessor communication enable bit. When set, it enables multiprocessor communication in mode 2 and 3, and eventually mode 1. It should be cleared in mode 0.
- ***REN*** **-** Reception Enable bit enables serial reception when set. When cleared, serial reception is disabled.
- ***TB8*** **-** Transmitter bit 8. Since all registers are 8-bit wide, this bit solves the problem of transmiting the 9th bit in modes 2 and 3. It is set to transmit a logic 1 in the 9th bit.
- ***RB8*** **-** Receiver bit 8 or the 9th bit received in modes 2 and 3. Cleared by hardware if 9th bit received is a logic 0. Set by hardware if 9th bit received is a logic 1.
- ***TI*** **-** Transmit Interrupt flag is automatically set at the moment the last bit of one byte is sent. It's a signal to the processor that the line is available for a new byte transmite. It must be cleared from within the software.
- ***RI*** **-** Receive Interrupt flag is automatically set upon one byte receive. It signals that byte is received and should be read quickly prior to being replaced by a new data. This bit is also cleared from within the software.

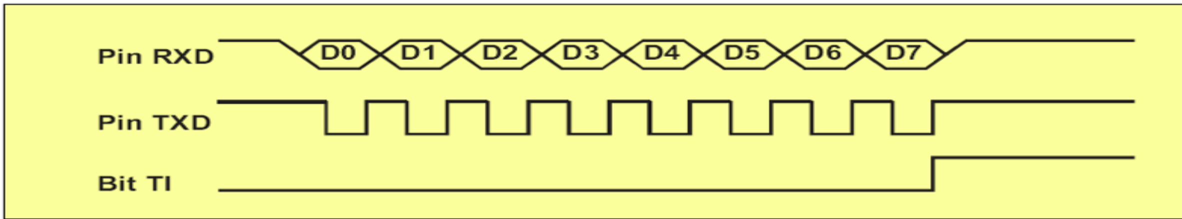
As seen, serial port mode is selected by combining the SM0 and SM2 bits:

SM0	SM1	MODE	DESCRIPTION	BAUD RATE
0	0	0	8-bit Shift Register	1/12 the quartz frequency
0	1	1	8-bit UART	Determined by the timer 1
1	0	2	9-bit UART	1/32 the quartz frequency (1/64 the quartz frequency)
1	1	3	9-bit UART	Determined by the timer 1

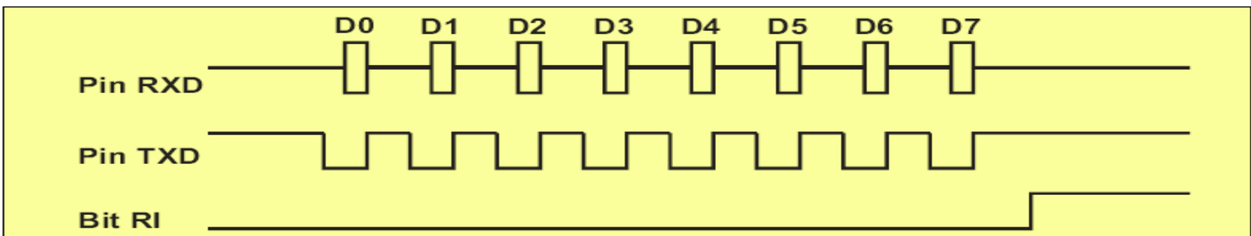


In mode 0, serial data are transmitted and received through the RXD pin, while the TXD pin output clocks. The bout rate is fixed at 1/12 the oscillator frequency. On transmit, the least significant bit (LSB bit) is sent/received first.

TRANSMIT - Data transmit is initiated by writing data to the SBUF register. In fact, this process starts after any instruction being performed upon this register. When all 8 bits have been sent, the TI bit of the SCON register is automatically set.

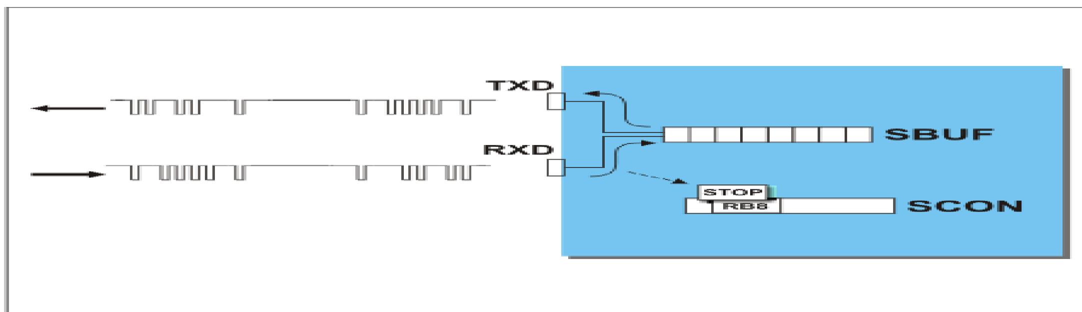


RECEIVE - Data receive through the RXD pin starts upon the two following conditions are met: bit REN=1 and RI=0 (both of them are stored in the SCON register). When all 8 bits have been received, the RI bit of the SCON register is automatically set indicating that one byte receive is complete.



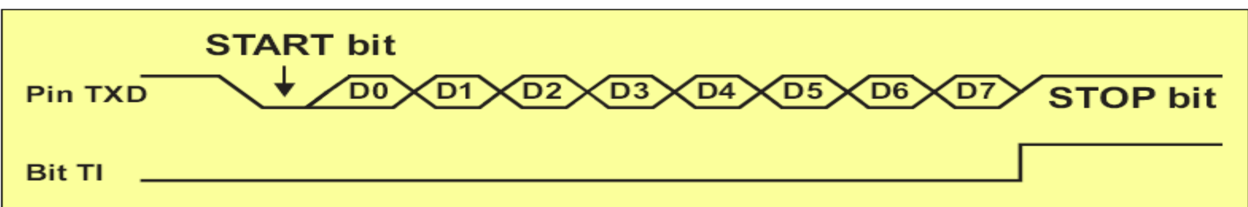
Since there are no START and STOP bits or any other bit except data sent from the SBUF register in the pulse sequence, this mode is mainly used when the distance between devices is short, noise is minimized and operating speed is of importance. A typical example is I/O port expansion by adding a cheap IC (shift registers 74HC595, 74HC597 and similar).

MODE 1

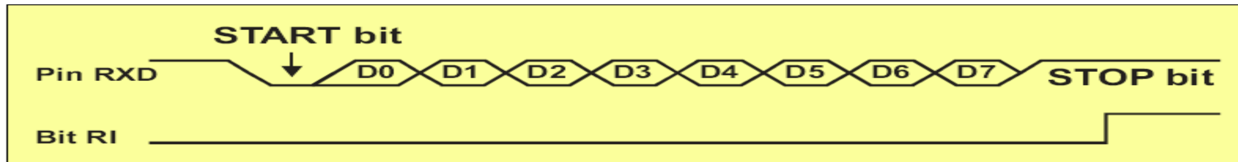


In mode 1, 10 bits are transmitted through the TXD pin or received through the RXD pin in the following manner: a START bit (always 0), 8 data bits (LSB first) and a STOP bit (always 1). The START bit is only used to initiate data receive, while the STOP bit is automatically written to the RB8 bit of the SCON register.

TRANSMIT - Data transmit is initiated by writing data to the SBUF register. End of data transmission is indicated by setting the TI bit of the SCON register.

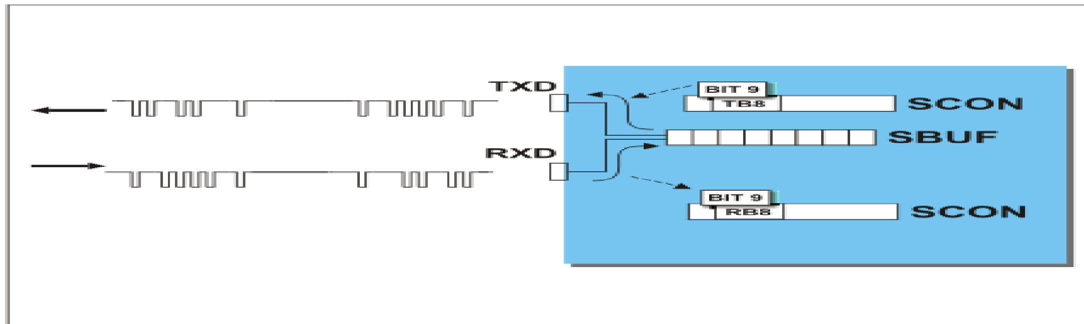


RECEIVE - The START bit (logic zero (0)) on the RXD pin initiates data receive. The following two conditions must be met: bit REN=1 and bit RI=0. Both of them are stored in the SCON register. The RI bit is automatically set upon data reception is complete.



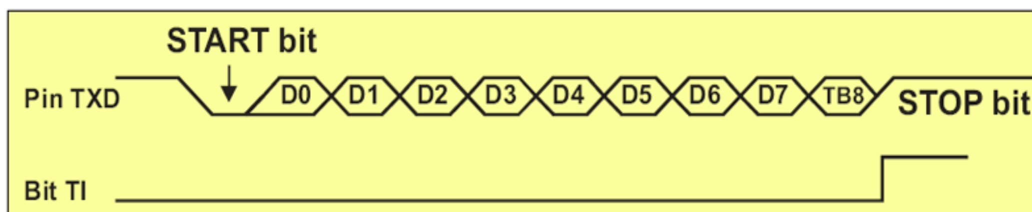
The Baud rate in this mode is determined by the timer 1 overflow.

MODE 2

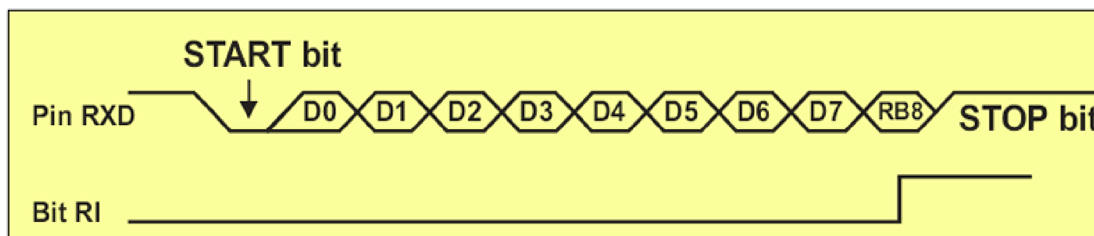


In mode 2, 11 bits are transmitted through the TXD pin or received through the RXD pin: a START bit (always 0), 8 data bits (LSB first), a programmable 9th data bit and a STOP bit (always 1). On transmit, the 9th data bit is actually the TB8 bit of the SCON register. This bit usually has a function of parity bit. On receive, the 9th data bit goes into the RB8 bit of the same register (SCON).The baud rate is either 1/32 or 1/64 the oscillator frequency.

TRANSMIT - Data transmit is initiated by writing data to the SBUF register. End of data transmission is indicated by setting the TI bit of the SCON register.



RECEIVE - The START bit (logic zero (0)) on the RXD pin initiates data receive. The following two conditions must be met: bit REN=1 and bit RI=0. Both of them are stored in the SCON register. The RI bit is automatically set upon data reception is complete.



MODE 3

Mode 3 is the same as Mode 2 in all respects except the baud rate. The baud rate in Mode 3 is variable.

The parity bit is the P bit of the PSW register. The simplest way to check correctness of the received byte is to add a parity bit to it. Simply, before initiating data transmit, the byte to transmit is stored in the accumulator and the P bit goes into the TB8 bit in order to be "a part of the message". The procedure is opposite on receiver, received byte is stored in the accumulator and the P bit is compared with the RB8 bit. If they are the same everything is fine.

BAUD RATE

Baud Rate is a number of sent/received bits per second. In case the UART is used, baud rate depends on: selected mode, oscillator frequency and in some cases on the state of the SMOD bit of the SCON register. All the necessary formulas are specified in the table:

	BAUD RATE	BITSMOD
Mode 0	Fosc. / 12	
Mode 1	1 Fosc. 16 12 (256-TH1)	BitSMOD
Mode 2	Fosc. / 32 Fosc. / 64	1 0
Mode 3	1 Fosc. 16 12 (256-TH1)	

TIMER 1 AS A CLOCK GENERATOR

Timer 1 is usually used as a clock generator as it enables various baud rates to be easily set. The whole procedure is simple and is as follows:

- First, enable Timer 1 overflow interrupt.
- Configure Timer T1 to operate in auto-reload mode.
- Depending on needs, select one of the standard values from the table and write it to the TH1 register.

BAUD RATE	FOSC. (MHZ)					BIT SMOD
	11.0592	12	14.7456	16	20	
150	40 h	30 h	00 h			0
300	A0 h	98 h	80 h	75 h	52 h	0
600	D0 h	CC h	C0 h	BB h	A9 h	0
1200	E8 h	E6 h	E0 h	DE h	D5 h	0
2400	F4 h	F3 h	F0 h	EF h	EA h	0

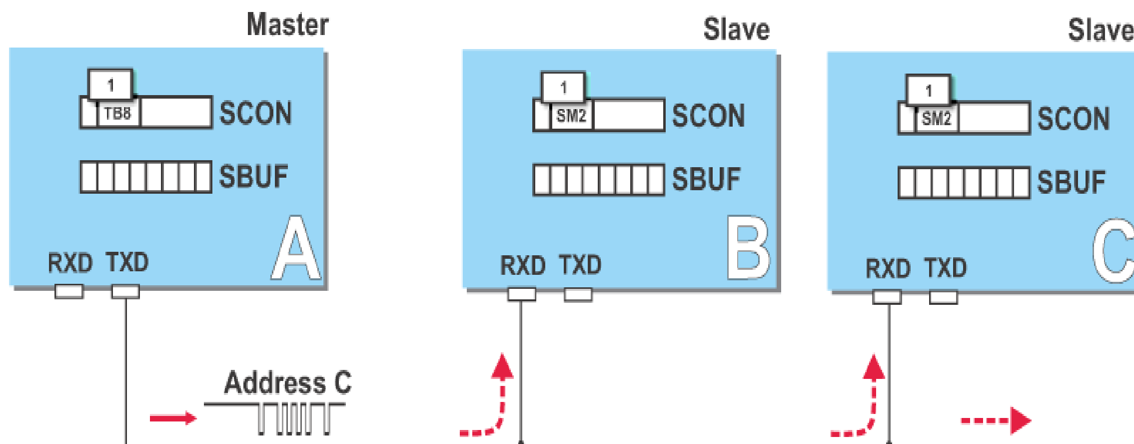
4800		F3 h	EF h	EF h		1
4800	FA h		F8 h		F5 h	0
9600	FD h		FC h			0
9600					F5 h	1
19200	FD h		FC h			1
38400			FE h			1
76800			FF h			1

MULTIPROCESSOR COMMUNICATION

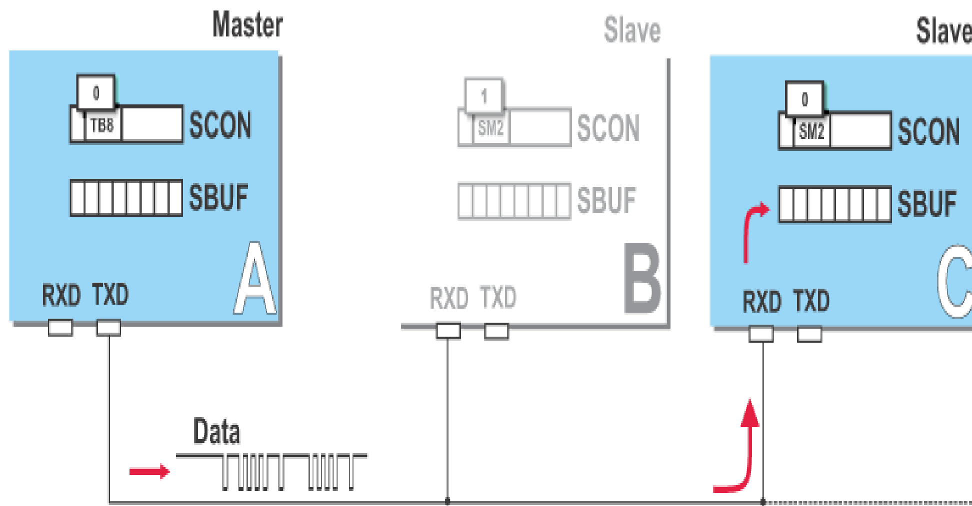
As you may know, additional 9th data bit is a part of message in mode 2 and 3. It can be used for checking data via parity bit. Another useful application of this bit is in communication between two or more microcontrollers, i.e. multiprocessor communication. This feature is enabled by setting the SM2 bit of the SCON register. As a result, after receiving the STOP bit, indicating end of the message, the serial port interrupt will be generated only if the bit RB8 = 1 (the 9th bit).

This is how it looks like in practice:

Suppose there are several microcontrollers sharing the same interface. Each of them has its own address. An address byte differs from a data byte because it has the 9th bit set (1), while this bit is cleared (0) in a data byte. When the microcontroller A (master) wants to transmit a block of data to one of several slaves, it first sends out an address byte which identifies the target slave. An address byte will generate an interrupt in all slaves so that they can examine the received byte and check whether it matches their address.



Of course, only one of them will match the address and immediately clear the SM2 bit of the SCON register and prepare to receive the data byte to come. Other slaves not being addressed leave their SM2 bit set ignoring the coming data bytes.



PERIPHERALS

TIMERS:-

- Timer is a very common and useful peripheral.
- It is used to generate events at specific times or measures the duration of specific events which are external to the processor.
- It is a programmable device, i.e. the time period can be adjusted by writing specific bit patterns to some of the registers called timer-control registers.
- A timer measures time by counting pulses that occur on an input clock signal having a known period.

COUNTERS:-

- A counter is nearly identical to a timer except that instead of counting the clock cycle, a counter counts the number of pulses of input signal.
- It can be free running device with a clock input pulse and for comparing the counts with one which is preloaded in the register.
- This device is used for the alarm and the other processors.
- It is useful for the processor interrupt at the preset time.

WATCHDOG TIMERS:-

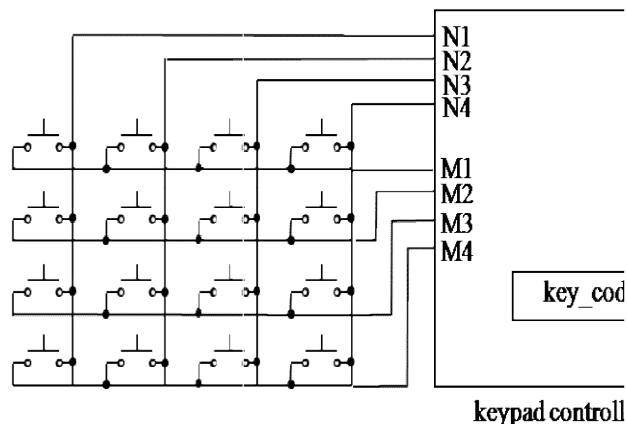
- A watchdog timer (WDT; sometimes called a computer operating properly or COP timer, or simply a watchdog) is an electronic timer that is used to detect and recover from computer malfunctions.
- During normal operation, the computer regularly restarts the watchdog timer to prevent it from elapsing, or "timing out".
- If, due to a hardware fault or program error, the computer fails to restart the watchdog, the timer will elapse and generate a timeout signal.
- The timeout signal is used to initiate corrective action or actions. The corrective actions typically include placing the computer system in a safe state and restoring normal system operation.
- Watchdog timers are commonly found in embedded systems and other computer-controlled equipment where humans cannot easily access the equipment or would be unable to react to faults in a timely manner.
- In such systems, the computer cannot depend on a human to reboot it if it hangs; it must be self-reliant.

LCD CONTROLLERS:-

- A liquid crystal display (LCD) is a low cost, low power device capable of displaying text and images.
- LCDs are extremely common in embedded systems.
- LCDs can be found in numerous common devices like watches, fax and copy machines and calculators.
- Each type of LCD may be able to display multiple characters.
- The LCD may permit a character to be blinking or may permit display of a cursor indicating the "current" character. Such functionality would be difficult for us to implement using software.
- Thus we use an LCD controller to provide us with a simple interface to an LCD with eight data input and one enable input.
- To send a byte to the LCD, we provide a value to the eight inputs and one enable input.
- This byte may be a control word, which instructs the LCD controller to initialize the LCD, clear the display, select the position of the cursor, brighten the display and so on.

KEYPAD CONTROLLERS:-

- A keypad consists of a set of buttons that may be pressed to provide input to an embedded system.
- A simple keypad has buttons arranged in an N- column by M-row grid as shown in the figure below.



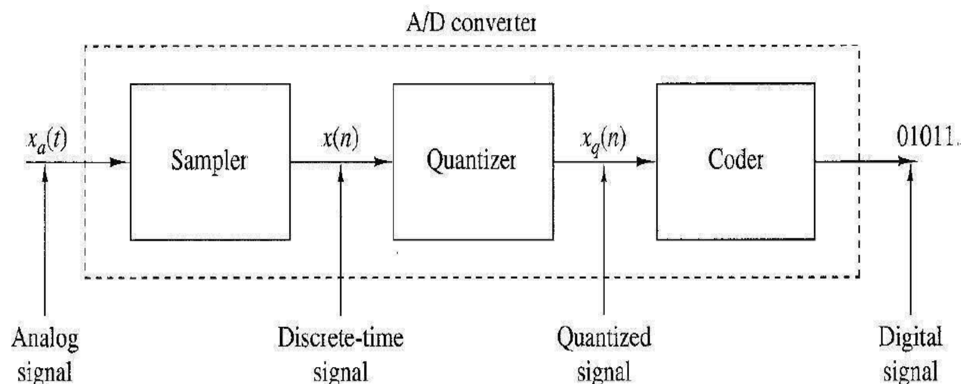
N=4, M=4

- The device has N outputs, each output corresponding to a column, and another M outputs, each output corresponding to a row. When a button is pressed one column output and one row output go high, uniquely identifying the pressed button.
- To read such a keypad from software, scan the column and row outputs. The scanning is performed by keypad controller.
- The controller scans the column and row outputs of the keypad. When the controller detects a press button, it stores a code corresponding to that button into a register, key_code, and sets an output high, k_ pressed, indicating that a button has been pressed.
- The software may poll this output every 100 milliseconds or so, and read the register when the output is high.

ANALOG TO DIGITAL CONVERTERS:-

- An ADC is a device which converts the analog signal to a digital signal and a digital signal to analog signal by the help of digital to analog converter.
- Such conversion are necessary because the embedded system deals with digital signal and is surrounded by system involving analog signal.
- The analog to digital conversion goes under the following 3 process:-
    1. Sampling
    2. Quantization
    3. Coding

Block diagram of ADC:-



Basic parts of an analog-to-digital (A/D) converter.

1. **Sampling:-**
   It is the process in which a continuous time signals or analog signal is converted into discrete time signal. This is done by the device sampler.

2. **Quantization:-**
   It is a process as in which a discrete time signal is converted into quantized signal. It can also defined as the process of conversion from continuous value to discrete value. This process is done by the device called quantizer.
3. **Coding:-**
   - Coding is the process in which the discrete valued signal is converted into corresponding binary form.
   - Analog always refers to continuous valued signal and digital always refers to discrete valued signal.
   - By converting from analog to digital we use a device called digital processor to compute the analog value to digital value.
     For this the formula is
     $e/V_{max} = d / (2^n – 1)$

     where V_{max} = Maximum voltage of analog signal

     n = no. of bits available for digital encoding

     d = the present digital encoding

     e = the present analog voltage

   - The resolution of the digital signal can be found by the formula $V_{max} / (2^n – 1)$.

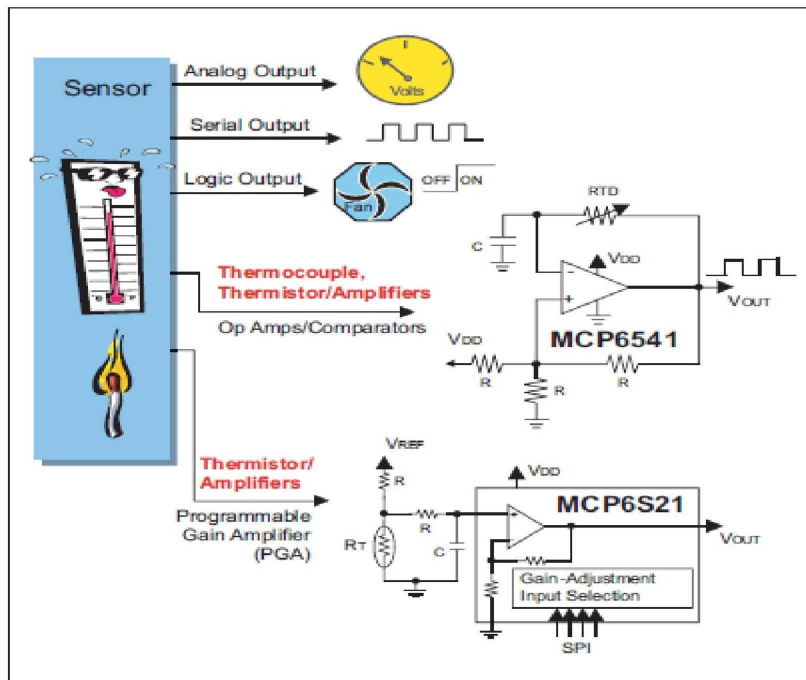
REAL - TIME CLOCKS:-

- A real – time clock (RTC) keeps the time and date in an embedded system.
- Real – time clocks are typically composed of a crystal – controlled oscillator, cascade counters and battery backup.
- The crystal – controlled oscillator generates a very consistent high-frequency digital pulse that feeds the cascaded counters.
- The first counter counts these pulses up to the oscillator frequency, which corresponds to exactly one second.
- At this point it generates a pulse that feeds the next counter. This counter counts up to 59, at which point it generates a pulse feeding the minute counter. The hour, date, month and year counters work in the same manner. Real- time clocks adjust for the leap years.
- The rechargeable back-up battery is used to keep the real-time clock running the system is powered off.

APPLICATION OF EMBEDDED SYSTEMS

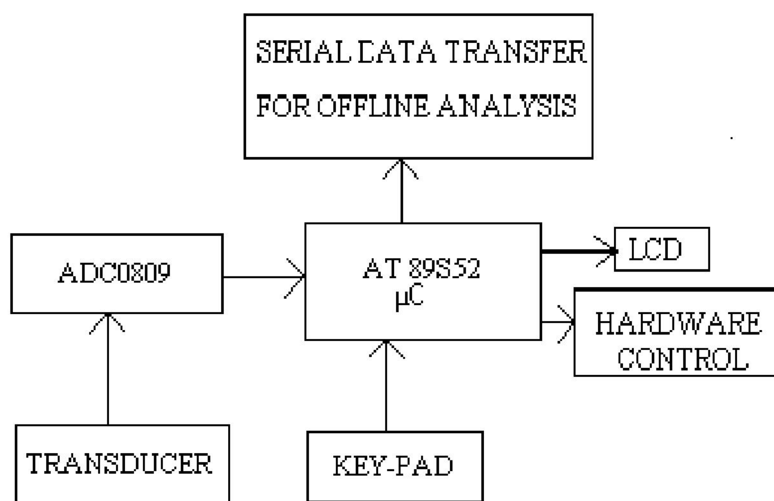
TEMPERATURE MEASURING SYSTEM:-

In many systems, temperature control is fundamental. There are a number of passive and active temperature sensors that can be used to measure system temperature, including: thermocouple, resistive temperature detector, thermistor and silicon temperature sensors. These sensors provide temperature feedback to the system controller to make decisions such as, over-temperature shutdown, turn-on/off cooling fan, temperature compensation or general purpose temperature monitor.



Temperature Measurement Applications

- Computing:
- CPU overtemperature protection
- Fan control
- Cellular/PCS:
- Power amplifier temperature compensation
- Thermal sensing of display for contrast control
- Power Supply Embedded Systems:
- Overtemperature shutdown
- Battery management



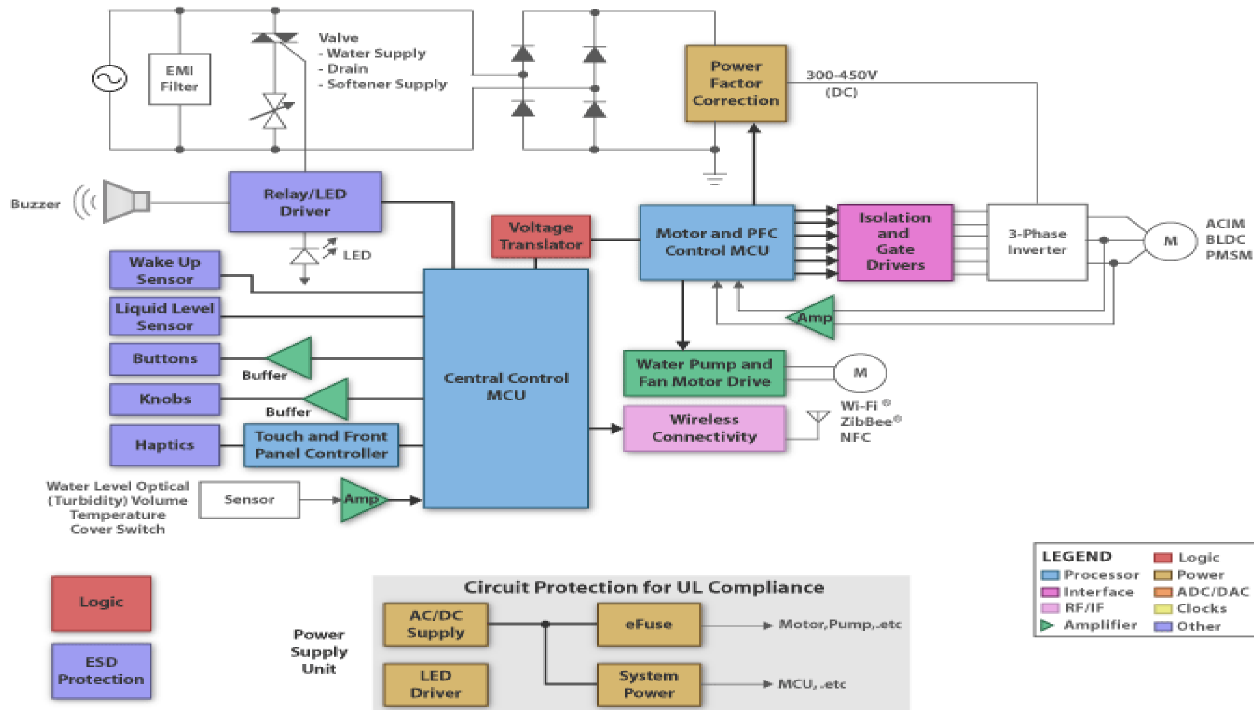
Hardware Description

Whole circuit can be divided into following sections:-

(a) **Power supply section:** The regulated power supply section made with full wave rectifier (with IN 4007 diodes) using voltage regulator IC 7805 and IC 7812 which provide a constant voltage of 5V to the circuit as well as constant 12V to relays.

(b) Analog to digital conversion section: Since we have to sense analog parameters i.e. temperature and light hence we have to use any analog to digital converter. We have opted for ADC 0809 as it has 8 channels and is microprocessor compatible ADC which is easily available. It will convert the analog signal of the transducer to digital value with respect to the reference voltage which in our case is 2.5V. This reference voltage is obtained using TL431, which is a programmable shunt voltage reference with output voltage range of 2.5V to 36V and works like zener diode. For the conversion ADC requires a reference frequency which is supplied from 555 IC in the form of astable oscillator. The conversion frequency is kept around 150 kHz. Sensor used for temperature measurement is LM 35 and for light intensity is LDR. LM 35 is calibrated in ºC and is linear in +10 mV/ ºC scale factor with 0.5ºC accuracy [5]. The calibration curve given here with will make the scenario clear.

Washing Machine



Description

Today's state-of-the-art high-end appliances are more energy efficient, consume less water, and are smarter than the machines developed in years past. Texas Instruments has developed products to meet the challenges and requirements for these new sophisticated systems.

Energy Efficiency: Home appliance (a.k.a. white goods) motors are often oversized to account for the load torque changes and transients. Scalar techniques for control can result in inefficient systems and noisy operation. This, in turn, leads to a mediocre energy efficiency that hovers in the 40% to 50% range. By implementing the control system with TI's digital signal controllers, designers are able to implement smaller, quieter motors with energy efficiency as high as 85% - 90%. A high efficiency is necessary to receive a stamp of approval from a governing body such as the US Environmental Protection Agency and Department of Energy's STAR rating.

Power Factor Correction (PFC): PFC is a technique of counteracting the undesirable effects of electric loads that create a power factor that is less than 1. In washing machines, PFC is needed because of the continuous transients and surge currents exhibited by the electric motor during the wash cycle, for example. It is also used to boost the rectified mains voltage up to 300 V to 450 V, which is then used to power the 3-phase inverters which ultimately operate the electric motor.

With TI products, PFC can be performed externally with a separate integrated circuit or it can be done in the digital signal controller eliminating the need for a separate external PFC controller.

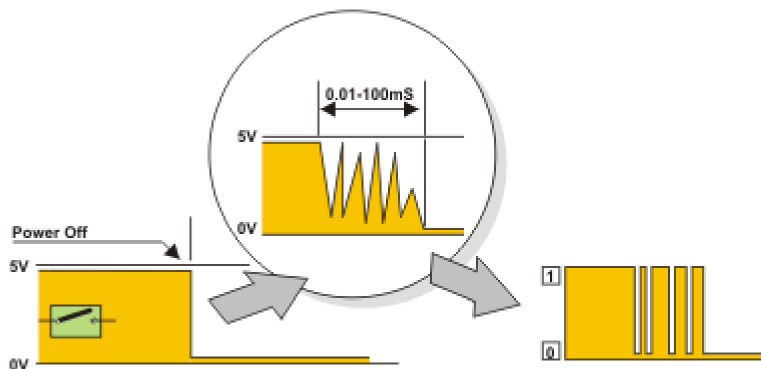
IEC 60730 Compliance: Home appliance manufacturers continually introduce new design enhancements to their automatic electronic controls that ensure safe, reliable and efficient operation of the equipment. Among other things, the IEC 60730 specification discusses mechanical, electrical, electronic, EMC, and abnormal operation of AC appliances. For microcontrollers, the specification details new test and diagnostic methods for the real-time embedded software to ensure the safe operation of embedded control hardware and software. All TI TMSxx24xxxx and TMSxx28xxxx digital signal controllers support IEC 60730 compliance.

High-Voltage Isolation: For larger, higher-performance products where reliability and motor-control accuracy are key concerns, TI offers isolation products that block high voltage, isolate grounds, and prevent noise currents from entering the local ground and interfering with or damaging sensitive circuitry.

Integration: Texas Instruments provides fully-integrated solutions such as the digital signal controllers (for digital motor control, PFC, and other system functions), and relay drivers that provide up to 8 channels, zero-volt detection, and 5 V linear regulation for 5 V logic that may reside on the board.

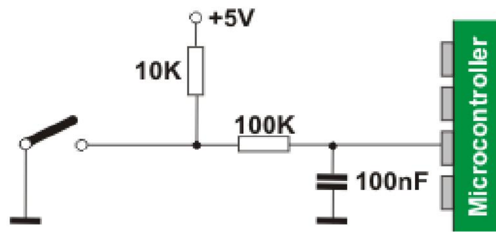
Switches and Push buttons

There are no simpler devices than switches and push-buttons. This is the simplest way of detecting appearance of a voltage on the microcontroller input pin.



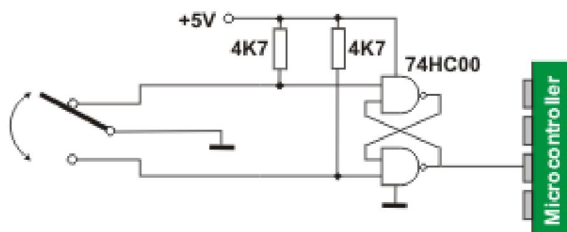
Nevertheless, it is not so simple in practice... It is about contact bounce- a common problem with m e c h a n i c a l switches. When the contacts strike together, their momentum and elasticity act together to cause bounce. The result is a rapidly pulsed electrical current instead of a clean transition from zero to full current. It mostly occurs due to vibrations, slight rough spots and dirt between contacts. This effect is usually unnoticeable when using these components in everyday life because the bounce happens too quickly. In other words, the whole this process does not last

long (a few micro- or miliseconds), but it is long enough to be registered by the microcontroller. When using only a push-button as a pulse counter, errors occur in almost 100% of cases.



The simplest solution to this problem is to connect a simple RC circuit to suppress quick voltage changes. Since the bounce period is not defined, the values of components are not precisely determined. In most cases, it is recomended to use the values shown in figure below.

If complete stability is needed then radical measures should be taken. The output of the circuit, shown in figure (RS flip-flop), will change its logic state only after detecting the first pulse triggered by contact bounce. This solution is expensive (SPDT switch), but effecient, the problem is definitely solved. Since the capacitor is not used, very short pulses can also be registered in this way.



In addition to these hardware solutions, there is also a simple software solution. When a program tests the state of an input pin and detects a change, the check should be done one more time after a certain delay. If the change is confirmed, it means that a switch or push button has changed its position. The advantages of such solution are obvious: it is free of charge, effects of noises are eliminated and it can be applied to the poorer quality contacts as well. Disadvantage is the same as when using RC filter, i.e. pulses shorter than program delay cannot be registered.

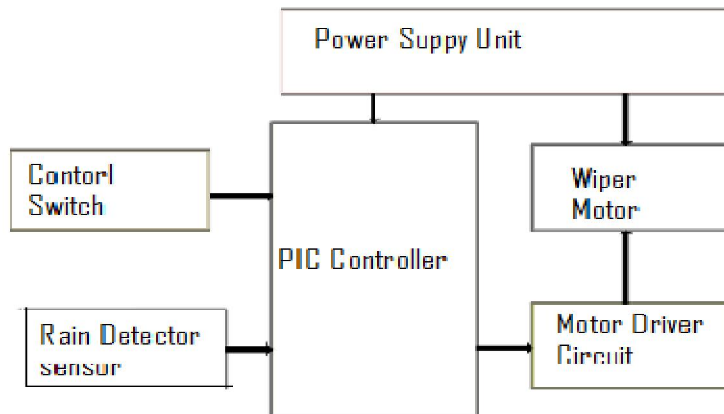
Working of windscreen wiper:-

The working of the wiper is based on the conversion of the wiper motors linear motion into linear back and forth movement of the wiper blades. The wiper combines two mechanical technologies to perform their tasks:

A combination of electric motor and worm gear reduction power to the wipers and a neat linkage

converts the rotational output of the motor into the back and forth motion of the wiper.

Motor takes a lot of force to accelerate the wiper blades back and forth across the windshield so quickly. In order to generate this type of force, a worm gear is used on the output of a small electric motor. The worm gear reduction can multiply the torque of the motor by about 50 times and can slow it down with the same force. The output of the gear reduction operates a linkage that moves the wipers back and forth. Inside the motor/gear assembly is the electronic circuit which senses the wipers are in their down position. The circuit maintains power to the wipers until they are parked at bottom of the windshield, and then cuts the power to the motor. A short cam is attached to the output shaft of the gear reduction. This cam spins around as the wiper motor turns. The cam is connected to a long road; as the cam spins, it moves the rod back and forth. The long rod is connected to a short rod that actuates the wiper blade on the driver's side. Another long rod transmits the force from the driver's side to the passenger's side wiper blade.



The basic control units of the hardware comprises of power supply unit, control switch, wiper motor, rain detector sensor ,motor driver circuit and the most important of all pic controller. Power supply unit maintains the continuous power to the controller and the wiper motor. Control switch is directly connected to the controller. Motor driver circuit is linked with the wiper motor and the controller. The command it gets from the controller is used to either drive the wiper motor or switch it off. Rain detection sensor detects the amount of moisture on the windscreen and accordingly sends the signal to the controller. A wiper motor control using a load sense resistor to monitor conditions confronting the wiper operation.

The control switch can detect if the wiper is frozen to the window, moving too slowly, moving too quickly, etc. The power to the motor is then adjusted to achieve a desired speed, or, otherwise adjusted, based upon the detected conditions. Further, the direction of the motor may be reversed based upon the detected conditions. In a preferred embodiment, the load sense resistor is provided by a defroster filament strip. Alternatively, a temperature sensor may be used to detect the conditions confronting the wiper operation.

PROGRAMMABLE LOGIC CONTROLLERS

INTRODUCTION:-

A programmable logic controller (PLC) is a specialized computer used to control machines and processes. It uses a programmable memory to store instructions and execute specific functions that include on/off control, Timing, Counting, Sequencing, arithmetic and data handling. Programmable logic controllers are used for the control and operation of manufacturing process equipment and machinery.

PARTS OF A PLC:-

- A typical PLC can be divided into parts as illustrated in the Figure below. These components are the central processing unit (CPU), the input /output (I/O) section, the power supply and the programming device.
- The term architecture can refer to PLC hardware to PLC software, or to a combination of both.

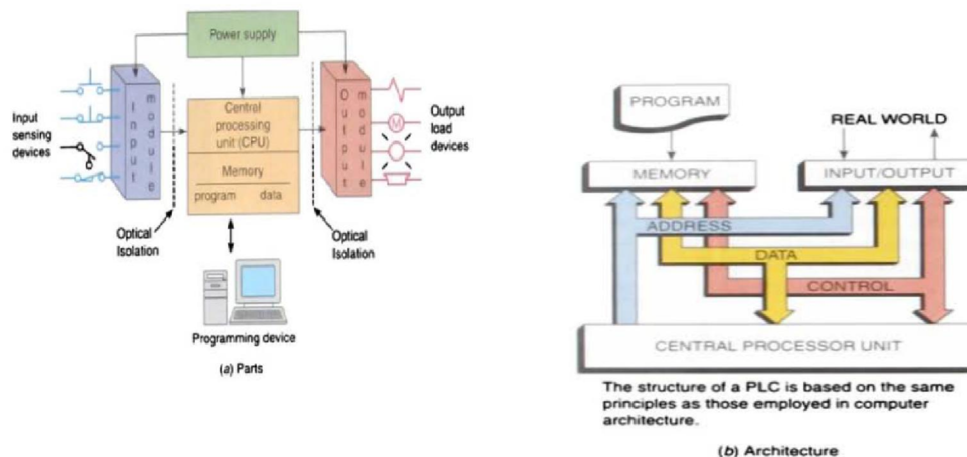


FIGURE: - PLC parts and architecture

- There are two ways in which I/O is incorporated into the PLC: fixed and modular.
- Fixed I/ O is typical of small PLCs that come in one package with no separate, removable units. The processor and I/O are packaged together, and the I/O terminals are available but cannot be changed.
- The main advantage of this type of packaging is lower cost.
- One disadvantage of fixed I/O is its lack of flexibility.

(b)

FIGURE: I/O configurations: (a) fixed I/O (b) modular I/O

- Modular I/O in figure shown above is divided by compartments into which separate modules can be plugged. This feature greatly increases your options and the unit's flexibility. The basic modular controller consists of a rack, power supply, processor module (CPU), input/output (l /O modules), and an operator interface for programming and monitoring. The modules plug into a rack. When a module is slid into the rack, it makes an electrical connection with a series of contacts called the backplane, located at the rear of the rack.
- The power supply supplies dc power to other modules that plug into the rack. For large PLC systems, this power supply does not normally supply power to the field devices. With larger systems, power to field devices is provided by external alternating current (ac) or direct current (dc) supplies.
- The processor (CPU) is the "brain" of the PLC. A processor usually consists of a microprocessor for implementing the logic and controlling the communications among the modules. The processor requires memory for storing the results of the logical operations performed by the microprocessor. Memory is also required for the program EPROM or EEPROM plus RAM.
- The I/O section consists of input modules and output modules (Fig. shown below). The I/O system forms the interface by which field devices are connected to the controller. The purpose of this interface is to condition the various signals received from or sent to external field devices. Input devices such as pushbuttons, limit switches, sensors, selector switches, and thumbwheel switches are hardwired to terminals on the input modules. Output devices such as small motors, motor starters, solenoid valves, and indicator lights are hardwired to the terminals on the output modules.

FIGURE: (a) Typical input module (b) typical output module

- The programming device, or terminal, is used to enter the desired program into the memory of the processor. This program is entered using relay ladder logic, which is the most popular programming language used by all major manufacturers of PLCs. Ladder logic programming language uses instead of words, graphic symbols that show their intended outcome. It is a special language written to make it easy for people familiar with relay logic control to program the PLC.

THE I/O SECTION:-

- The I/O section consists of an I/O rack and individual I/O modules similar to that shown in Figure shown below. Input interface modules accept signals from the machine or process devices and convert them into signals that can be used by the controller. Output interface modules convert controller signals into external signals used to control the machine or process.

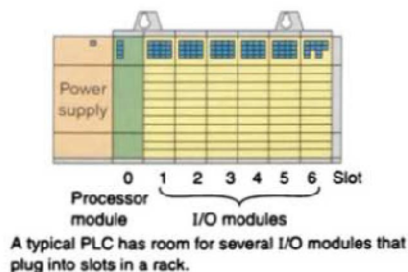
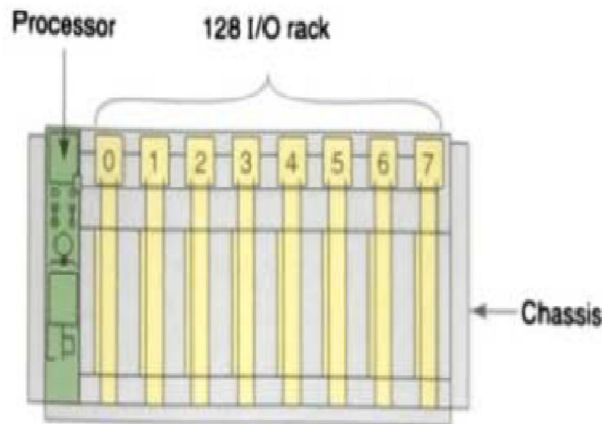


Figure: I/O section

- The I/O system provides an interface between the hardwired components in the field and the CPU. The input interface allows status information regarding processes to be communicated to the CPU, and thus allows the CPU to communicate operating signals through the output interface to the process devices under its control.

- A chassis is a physical hardware assembly that houses devices such as I/O modules, processor modules, and power supplies. Chassis come in different sizes according to the number of slots they contain. In general, they can have 4, 8, 12, or 16 slots.
- A logical rack is an addressable unit consisting of 128 input points and 128 output points. A rack uses 8 words in the input image table file and 8 words in the output image table file. A word in the output image table file and its corresponding word in the input image table file are called an I/O group. A rack can contain a maximum of 8 I/O groups (numbered from 0 through 7) for up to 128 discrete I/O (Fig. shown below). There can be more than one rack in a chassis and more than one chassis in a rack.



Output image table

Words																
O:00	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
O:01	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
O:02	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
O:03	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
O:04	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
O:05	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
O:06	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
O:07	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17

Input image table

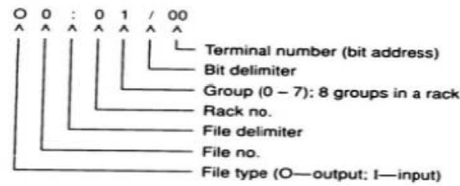
Words																
I:00	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
I:01	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
I:02	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
I:03	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
I:04	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
I:05	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
I:06	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
I:07	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17

Allen-Bradley PLC-5 family uses an octal numbering system to address bit locations. Notice that the 16 bits are numbered 0 through 7 and 10 through 17. In the octal numbering system, the numbers 8 and 9 are never used.

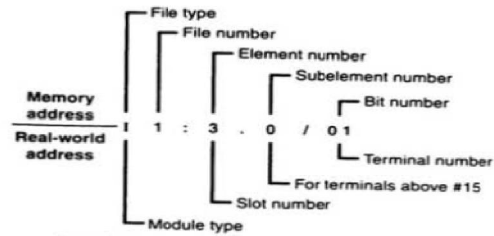
Figure: Logical rack

- One benefit of a PLC system is the ability to locate the I/O modules near the field devices to minimize the amount of wiring required. This rack is referred to as a remote rack when it is located away from the processor module.
- The location of a module within a rack and the terminal number of a module to which an

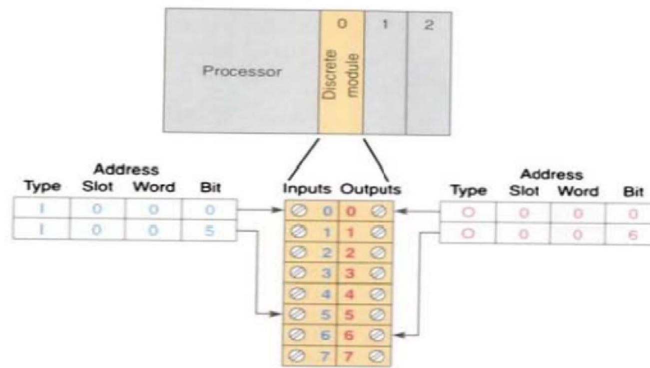
input or output device is connected will determine the device's address (Fig. shown below). Each input and output device must have a specific address. This address is used by the processor to identify where the device is located to monitor or control it. In addition there is some means of connecting field wiring on the I/O module housing. Connecting the field wiring to the I/O housing allows easier disconnection and reconnection of the wiring to change modules. Lights are also added to each module to indicate the ON or OFF status of each I/O circuit. Most output modules also have blown fuse indicators.



(a) Allen-Bradley PLC 5 addressing format.



(b) Allen-Bradley SLC 500 addressing format.



(c) Discrete I/O module addressing.

In general, basic addressing elements include:

• **TYPE**

The type determines if an input or output is being addressed.

• **SLOT**

The slot number is the physical location of the I/O module. This may be a combination of the rack number and the slot number when using expansion racks.

• **WORD AND BIT**

The word and bit are used to identify the actual terminal connection in a particular I/O module. A discrete module usually uses only one word, and each connection corresponds to a different bit that makes up' the word.

PLC MEMORY ORGANIZATION:-

- The term processor memory organization refers to how certain areas of memory in a given PLC are used.

Figure given below shows an illustration of the Allen-Bradley PLC-2 memory organization, known as a memory map. Every PLC has a memory map, but it may not be like the one illustrated. The memory space can be divided into two broad categories: the user program and the data table. The individual sections, their order, and the sections' length will vary and may be fixed or variable, depending on the manufacturer and model.

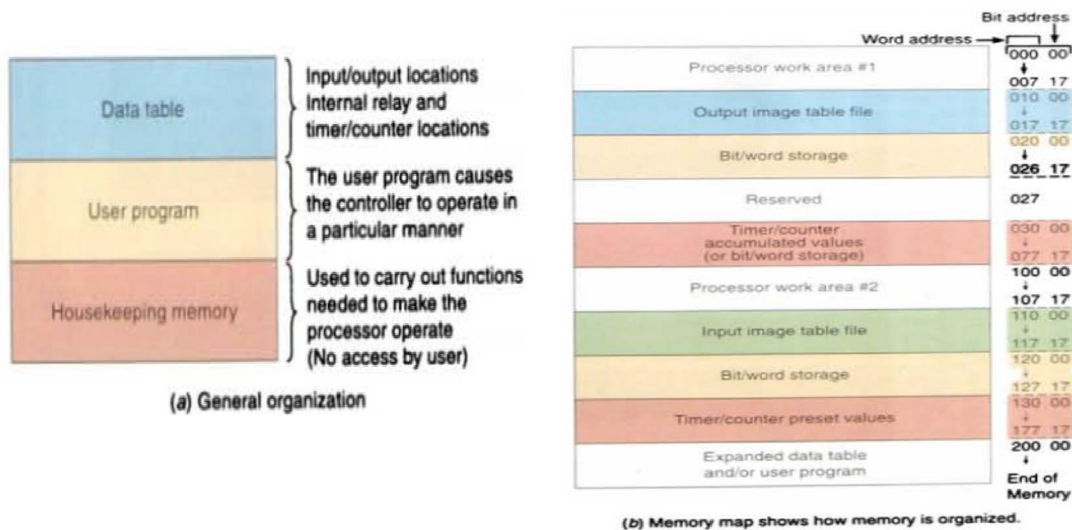


Figure: Memory map for an Allen-Bradley PLC-2

- The user program is where the logic ladder program is entered and stored. The user program will account for most of the total memory of a given PLC system. It contains the logic that controls the machine operation. This logic consists of instructions that are programmed in a ladder logic format. Most instructions require one word of memory.
- The data table stores the information needed to carry out the user program. This includes information such as the status of input and output devices, timer and counter values, data storage, and so on. Contents of the data table can be divided into two categories: status data and numbers or codes. Status is ON/OFF type of information represented by 1s and 0s, stored in unique bit locations. Number or code information is represented by groups of bits that are stored in unique byte or word locations.
- A processor file is the collection of program files and data files created under a particular processor file name. It contains all the instructions, data, and configuration information pertaining to a user program.

Figure shown below shows typical program and data file memory organization for an Allen-Bradley SLC-500 controller. The contents of each file are outlined in the sections that follow.

Program files:-

Program files are the areas of processor memory where ladder logic programming is stored. They may include:

- **System functions (file 0)** - This file is always included and contains various system-related information and user programmed information such as processor type. I/O configuration processor files name, and password.
- **Reserved (file 1)-**This file is reserved by the processor and is not accessible to the user.
- **Main ladder program (file 2)** - This file is always included and contains user programmed instructions that define how the controller is to operate.
- **Subroutine ladder program (files3- 255)-**These files are user-created and are activated according to subroutine instructions residing in the main ladder program file.

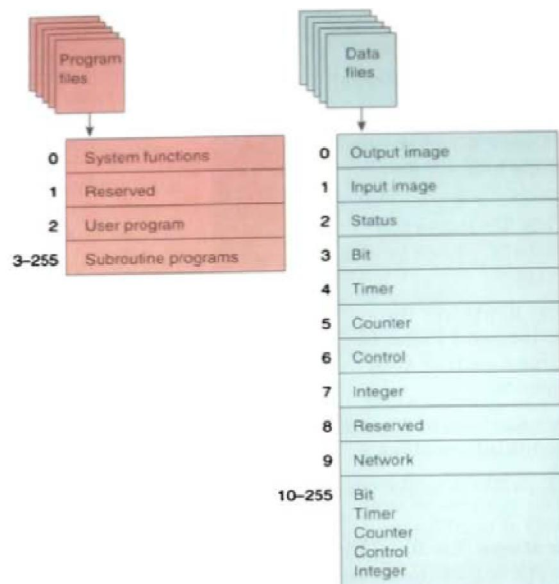


Figure: Program and data file memory organization for an Allen-Bradley SLC-500 controller

Data Files:-

The data file portion of the processor's memory stores input and output status, processor status, the status of various bits, and numerical data. All this information is accessed via the ladder logic program. These files are organized by the type of data they contain and may include:

- **Output (file 0) -** This file stores the state of the output terminals for the controller.
- **Input (file 1)**-This file stores the status of the input terminals for the controller.
- **Status (file 2)** - This file stores controller operation information. This file is useful for troubleshooting controller and program operation.
- **Bit (file 3)**-This file is used for internal relay logic storage.

- **Timer (file 4)**-This file stores the timer accumulated and preset values and status bits.
- **Counter (file 5)**-This file stores the counter accumulated and preset values and status bits.
- **Control (file 6)-**This file stores the length, pointer position, and status bit for specific instructions such as shift registers and sequencers.
- **Integer (file 7)-**This file is used to store numerical values or bit information.
- **Reserved (file 8)** - This file is not accessible to the user.
- **Network communications (file 9)** - This file is used for network communications if installed or used like files 10-255.
- **User-defined (files 10-255)**-These files are user-defined as bit, timer, counter, control, and/or integer data storage.

PROGRAM SCAN OF A PLC:-

- During each operating cycle, the processor reads all the inputs, takes these values, and energizes or de-energizes the outputs according to the user program. This process is known as a scan. Figure shown below shows a single PLC scan, which consists of the I/O scan and the program scan. Because the inputs can change at any time, the PLC must carry on this process continuously.

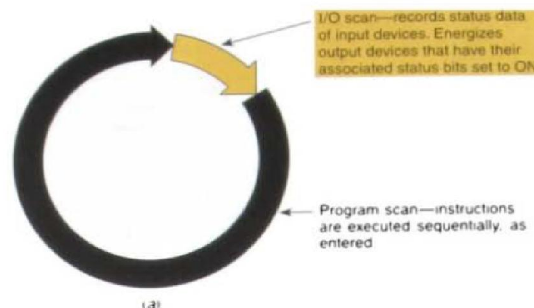


Figure: Single PLC scan

- The PLC scan time specification indicates how fast the controller can react to changes in inputs. Scan time varies with program content and length. The time required to make a single scan can vary from about 1 ms to 20 ms. If a controller has to react to an input signal that changes states twice during the scan time, it is possible that the PLC will never be able to detect this change.
- For example, if it takes 8 ms for the CPU to scan a program, and an input contact is opening and closing every 4 ms, the program may not respond to the contact changing state. The CPU will detect a change if it occurs during the update of the input image table file, but the CPU will not respond to every change. The scan is normally a continuous and sequential process of reading the status of inputs, evaluating the control logic, and updating the outputs.

Figure shown below illustrates this process.

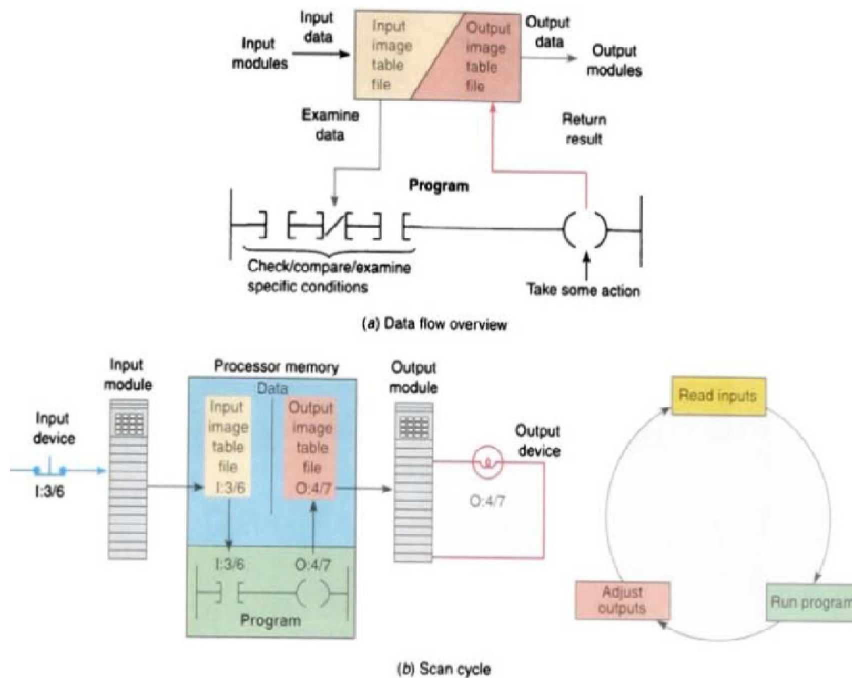


Figure: Scan Process

- When the input device connected to address I:3/6 is closed, the input module circuitry module circuitry senses a voltage and a 1 (ON) condition is entered into the input image table bit I:3/6. During the program scan, the processor examines bit I: 3/6 for a 1 (ON) condition.
- In this case, because input I: 3/6 is 1, the rung is said to be TRUE. The processor then sets the output image table bit O: 4/7 to 1. The processor turns on output O: 4/7 during the next I/O scan, and the output device (light) wired to this terminal becomes energized. This process is repeated as long as the processor is in the RUN mode. If the input device were to open, a 0 would be placed in the input image table. As a result, the rung would be called FALSE. The processor would then set the output image table bit O: 4/7 to 0, causing the output device to turn off.

CONTROL INSTRUCTIONS OF PLC:-

- Figure given below shows typical program control instructions.
- Instructions comprising the override instruction group include the master control reset (MCR) and jump (JMP) instructions.
- These operations are accomplished by using a series of conditional and unconditional branches and return instructions.

Command	Name	Description
JMP	Jump to Label	Jump forward/backward to a corresponding label instruction
LBL	Label	Specifies label location
JSR	Jump to Subroutine	Jump to a designated subroutine instruction
RET	Return from Subroutine	Exits current subroutine and returns to previous condition
SBR	Subroutine	Identifies the subroutine program
TND	Temporary End	Makes a temporary end that halts program execution
MCR	Master Control Reset	Clears all set outputs between the paried MCR instructions
SUS	Suspend	Identifies specific conditions for program debugging and system troubleshooting

- Hardwired master control relays are used in relay circuitry to provide input/output power shutdown of an entire circuit. Figure shown below is a typical hardwired master control relay circuit. In this circuit, unless the master control relay coil is energized, there is no power flow to the load side of the MCR contacts. The master control relay circuit shown in Figure below could not be programmed into the PLC as it appears because it contains two vertical contacts. For this reason, most PLC manufacturers include some form of master control relay as part of their instruction set. These instructions function in a similar manner to the hardwired master control relay; that is, when the instruction is true, the circuit functions normally, and when the instruction is false, outputs are switched off. Because these instructions are not hardwired but programmed, for safety reasons they should not be used as a substitute for a hard· wired master control relay, which provides emergency I/O power shutdown

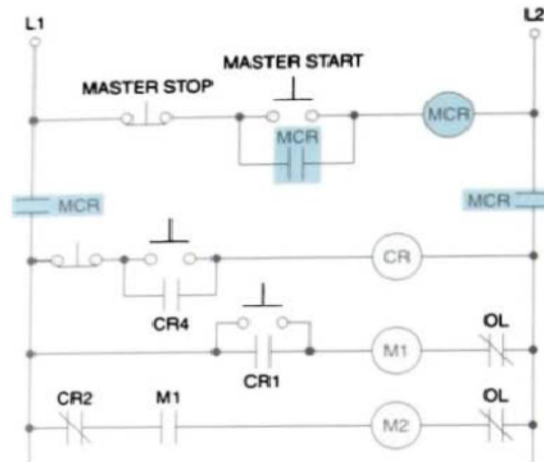


Figure: Hardwired master control relay circuit

- The master control reset instruction can be programmed to control an entire circuit or to control only selected rungs of a circuit. In the program of figure shown below, the MCR is programmed to control an entire circuit. When the MCR instruction is false, or de-energized, all non retentive (nonlatched) rungs below the MCR will be de-energized even if the programmed logic for each rung is true. All retentive rungs will remain in their last state. The MCR instruction establishes a zone in the user program in which all nonretentive outputs can be turned off simultaneously. Therefore, retentive instructions should not normally be placed within an MCR zone because the MCR zone maintains retentive instructions in the last active state when the instruction goes false.

DIFFERENCE BETWEEN PLC AND PC:-

PLC:-

- The input- output capabilities are large and can be used in various industrial processes.
- The input-output cards are mounted on racks and are visible.
- The noise is more.
- The possibility of mishap exists as input-output cards are on the rack.
- The connections are rugged, accessible and organized.
- The processor architecture is simple.
- It is used for controlling industrial processes and control logic programming.
- The interface is by using simple devices such as indicators push buttons etc.
- It uses ladder programming.
- It has limited expandability.

PC:-

- Limited input-output capabilities interms of analog and digital cards which are suited for laboratory purposes.

- The input-output cards are mounted on board and are not visible.
- Noise is less
- No possibility of mishap
- There exists a bunch of cables and very small room available for cable connections.
- The processor architecture is complicated.
- It is used for many advanced computation and high level programming language is used.
- The easy and good interface is not possible.
- It uses microprocessor programming
- It has great flexibility and high reliability.

LADDER RUNG DIAGRAM:-

- The main function of the ladder logic diagram program is to control outputs based on input conditions. This control is accomplished through the use of what is referred to as a ladder rung. In general, a rung consists of a set of input conditions, represented by contact instructions, and an output instruction at the end of the rung, represented by the coil symbol (Fig. given below).

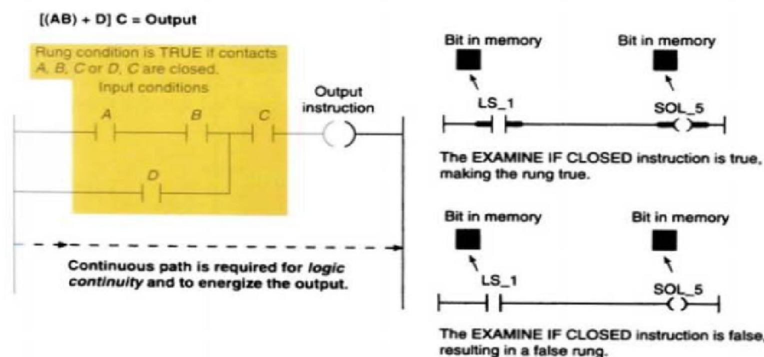


Figure: Ladder rung

- Each contact or coil symbol is referenced with an address number that identifies what is being evaluated and what is being controlled. The same contact instruction can be used throughout the program whenever that condition needs to be evaluated.
- For an output to be activated or energized, at least one left-to-right path of contacts must be closed. A complete closed path is referred to as having logic continuity.
- When logic continuity exists in at least one path, the rung condition is said to be TRUE. The rung condition is FALSE if no path has continuity.
- During controller operation, the processor determines the ON/OFF state of the bits in the data files, evaluates the rung logic, and changes the state of the outputs according to the logical continuity of rungs. More specifically, input instructions set up the conditions under which the processor will make an output instruction true or false.
- These conditions are as follows:
    1. When the processor finds a continuous path of true input instructions in a rung, the

OUTPUT ENERGIZE [OTE] output instruction will become (or remain) true. We then say that rung conditions are TRUE.

2. When the processor does not find a continuous path of true input instructions in a rung, the OTE input instruction will become (or remain) false. We then say that rung conditions are FALSE.

TIMER:-

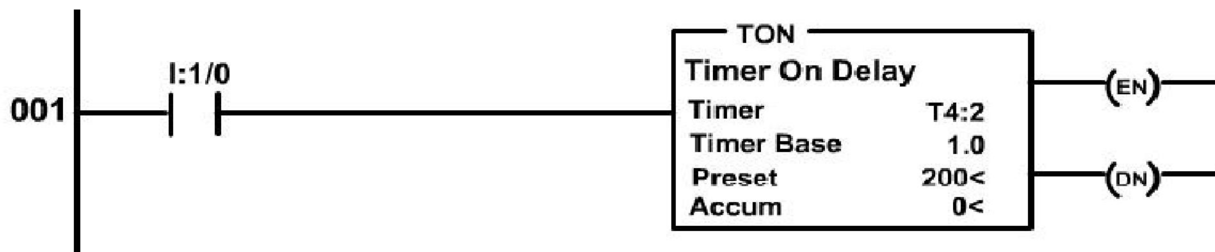
All PLC's have timer instructions. Timers are output instructions that are internal to the programmable logic controller. Timers provide timed control of the devices that they activate or de-activate.

Basic functions of timer

- Timers are used to delay an action.
- Timers are used to run an operation for a predetermined period of time.
- Timers are also used to record the total accumulated time of continuous or intermediate events.

Timer's Instructions:-

Timers consists of following parts: timer address, preset value, timer base, and accumulated value, as shown in figure below.



In the above figure, term instruction name is timer on delay (TON), timer base is 1.0 seconds, timer address is T4:0, accumulated value of zero (0) and a preset value of 200. Each timer instruction has three very useful status bits. These bits are timer enable (E N), timer timing (TT), and timer done (DN). There are 3 types of timers: On- delay timer, Off-delay timer, and retentive timer.

On delay timer:-

- Use this instruction to program a time delay after instructions become true.
- On – delay timers are used when an action is to begin a specified time after the input becomes true. For example, a certain step in the manufacturing is to begin 45 seconds after a signal is received from a limit switch. The 45- seconds delay is the on-delay timers preset value.

Off- delay timer:-

- Off- delay timer instructions is used to program a time delay to begin after rung input goes false.
- As an example, when an external cooling fan on a motor is provided, the fan has to run all the time the motor is running and also for certain time (say 10min) after the motor is turned off. This is a ten minute off- delay timer. The ten-minute timing period begins as soon as the motor is turned off.

Retentive timer:-

- Retentive timer is a timer which retains the accumulated value in case of power loss, change of processor mode or rung state going from true to false (rung state transition).
- Retentive timer can be used to track the running time of a motor for its maintenance purpose. Each time the motor is turned off, the timer will remember the motor's elapsed running time. The next time the motor is turned on, the time will increase from there. This timer can be reset by using a reset instruction.

Reset:-

- This instruction is used to reset the accumulated value of counter or timer.
- It is used to reset a retentive timer's accumulated value to zero.

A typical timer element:-

A timer element is made up of three 16 bit words:

- Word 0: 3 status bits (EN, TT, DN).
- Word 1: Preset values.
- Word 2: Accumulated value.

<table>
<tr><td>EN</td><td>TT</td><td>DN</td><td>Reserved Bits</td></tr>
<tr><td colspan="4">Preset Value</td></tr>
<tr><td colspan="4">Accumulated Value</td></tr>
</table>

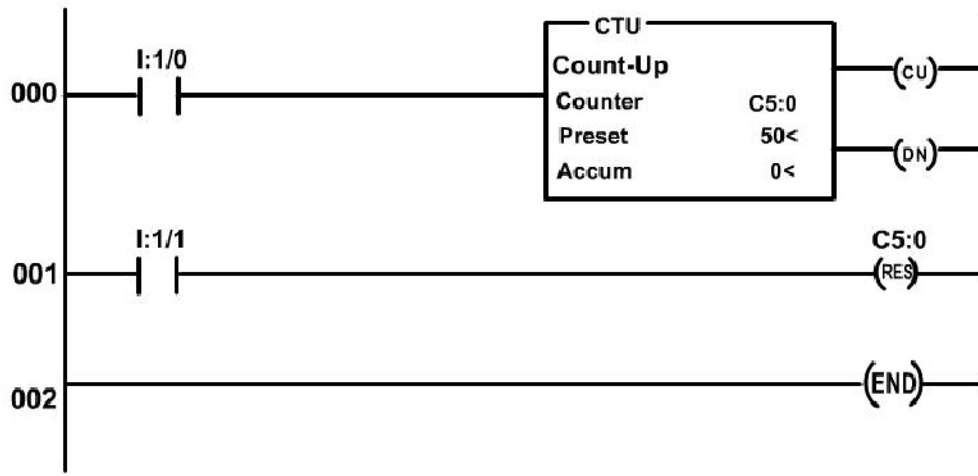
COUNTER:-

A counter is a simple device intended to do one simple thing-count. Every PLC has counter instructions. Using counters sometimes be little challenging because many manufacturers seem to use them different way. In other words, the instruction symbol used and method of programming will change for different manufacturers.

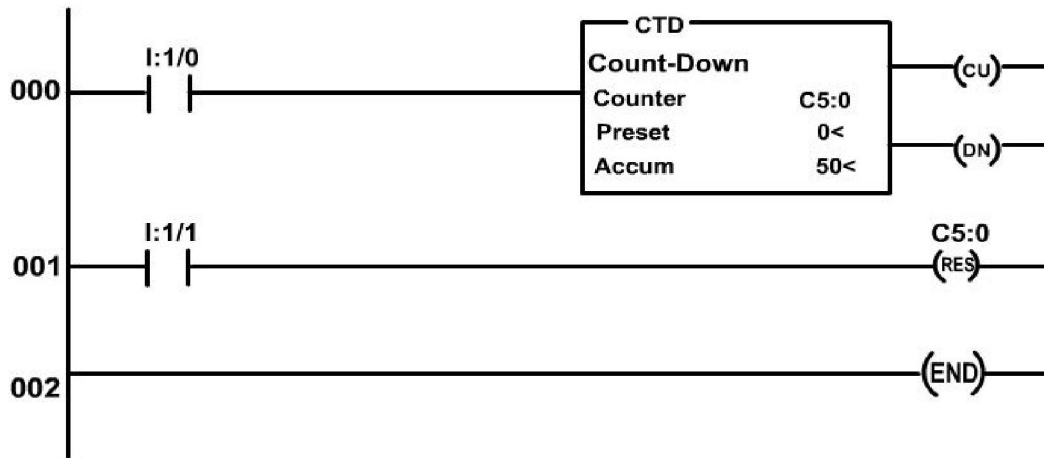
A typical counter counts from 0 up to a predetermined value, called the "preset" value. For example, if you wanted to count from certain value, say from 0 to 50, you would be counting

up using a count-up or up-counter. Here, the number "50" is the predetermined value, which is nothing but preset value. The current count or accumulated count is called as the "accumulated value". If our counter had counted 25 pieces that had passed on the conveyor, the accumulated count would be 25. When all 50 pieces had passed on the conveyor, the preset value and counter accumulated value would be equal. At this point the counter would signal other logic within the PLC program that the batch of 50 was completed and it should now take some action. The next action the PLC has to take is to move the box containing 50 parts on to the next station for carton sealing. To start counting the next batch, a reset instruction would be used to reset the counter's accumulated value back to zero.

The fig below shows an up-counter, counting from 0 to 50.



The fig below shows a down-counter, counting from 50 to 0.



PLC Counter Instructions

Instruction	This instruction is used to	Functional description
Count Down	Count down from a desired value to zero	An operator interface display shows the operator the number of parts remaining to be made for a lot of,say 50 parts ordered.
Count Up	Count from zero up to a desired value	Counting the number of parts produced during a specific work shift or batch.Also counting the number of rejects from a batch.
High-speed Counter	Count input pulses that are too fast,from normal input points and modules	Most fixed programmable logic controllers have a high-speed set of input points that allow interface to high-speed inputs. Signals from an incremental encoder would be a typical high-speed input.
Counter Reset	To reset a counter or timer	Used to reset a counter to zero so that another counting sequence can begin.

As for explanation, let us take ALLEN-BRADELY counters:

In Allen-bradely counter, the default counter file is file 5. The counter data is stored in counter file. Each counter consists of three 16-bit words and is known as “counter element”. In a single processor file, there can be many counter files. Any data file, which is greater than file 8 can be assigned as an additional counter file. Each counter file can have up to 256 counter elements. A counter instruction is one element. A counter element is made up of three 16-bit words. Thus, the counter instruction contains the three parts i.e. word0, word1 and word2.

- Word zero is for status bits. Status bits include CU, CD, DN, OV, UN, and UA. Along with their associated instructions.
- Word one is for the preset value.
- Word two is for accumulated value.

<table>
<tr><td>CU</td><td>CD</td><td>DN</td><td>OV</td><td>UN</td><td>UA</td><td>Reserved Bits</td></tr>
<tr><td colspan="7">Preset Value</td></tr>
<tr><td colspan="7">Accumulated Value</td></tr>
</table>

Addressing a counter

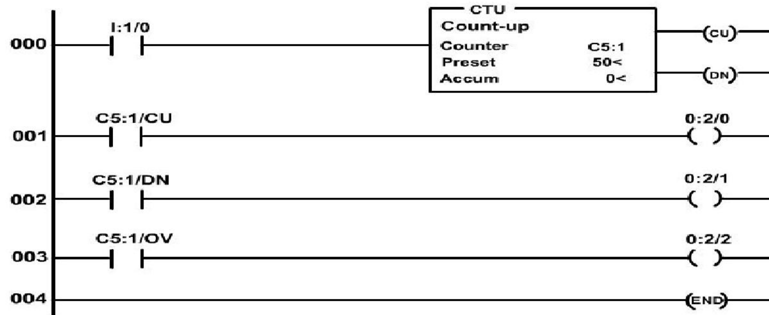
1) To address the counter as a unit the addressing format used is C5:4.
Where, C= C identifies this as a counter file.
5= This is counter file 4 which is default. Any unused file from 10 to 255 can be assigned to counters.
:4= The colon used here is called the file separator. It separates the file, file 5, from the specific counter, in this case, counter 4 in counter file 5.
2) To address the counter 14's accumulated value, the address used is C5:14.ACC.
Where, C= C identifies this as a counter file.
5= This represents the counter file 5.
:14= The colon is called the file separator. The colon separates the file, file 5, from the specific counter; in this case, counter 14 in counter file 5.
. = The point is called the word delimiter. The word delimiter is used to separate the counter number, called the structure, from the sub element. The sub element is ACC for the accumulated value. Similarly, preset value can be accessed as C5:14.PRE.
The addressing format of counter status bits is as follows:

- C5:14/DN is the address for counter file 5, counter 14's done bit.
- C5:14/CU is the address for counter file 5, counter 14's count-up-enable bit.
- C5:14/EN is the address for counter file 5, counter 14's enable bit.

Working of a counter:-

- A counter instruction is always an output instruction. The counter instruction counts each time the input logic changes the rung state from false to true. This input logic can be signal from an external device, for e.g. limit switch or sensor, or a signal from internal logic. Every time the counter instruction sees a false-to-true rung transition, a count-up counters accumulated value is incremented by one.
- The working of down-counter is little different. Each time when count-down counter sees a false-to-true rung transition, its accumulated value is decremented by one. Since the accumulated value gets decremented by 1 when each time the input logic changes the rung state from false to true, the accumulated value must be chosen as the starting point of the count.
- Counters are retentive in nature. The counter will retain its accumulated value or the on or off status of the done, overflow and underflow bits through a power loss.

The count-up instruction



The count-up instruction is used if we want a counter to increment one decimal value each time it register a rung transition from false to true. Each time input I:1/0 has a transition from off to on, counter C5:1 will increment its accumulated value by one decimal value. The count-up-enabled bit, on rung 001 is set when the rung conditions are true, or enabled. In rung 002, the done bit, DN, is set when the accumulated value is equal to or greater than the preset value. In the event of wrap from +32,767 to -32,768, the accumulated value becomes less than the preset value and the done bit will not be reset. In rung 003, the count-up overflow bit, OV, is set whenever the count-up counters accumulated value wraps from +32,767 to -32,768.

The count-down instruction

This instruction is used when we want to count down over the range of +32,767 to -32,768. Accumulated value will be decremented by one count, each time the instruction sees a false-to-true transition. Count-down instruction has many applications, for example: if we want to display the remaining number of parts to be filled for a specific order say 50 parts, then, a count-down instruction can be used. In this example, the accumulated value will be set as 50 and the preset value will be 0. Each time a part is completed and passes the sensor, the accumulated value will be decremented by one decimal value, as shown in the figure below.