Embedded Systems

By
Chandra Sekhar G.
Assistant Professor
Electronics and Communication Engineering Dept.
Bharati Vidyapeeth's College of Engineering
New Delhi

UNIT - I

- **Overview of Embedded Systems**:
  - Characteristics of Embedded Systems.
  - Comparison of Embedded Systems with general purpose processors.
  - General architecture and functioning of micro controllers.
  - 8051 micro controllers.
- **PIC Microcontrollers:**
  - Architecture, Registers, memory interfacing, interrupts, instructions.
  - Programming and peripherals.

Note: Prerequisites: Clear Conceptual knowledge in Microprocessors, Microcontrollers and their Functional Units.

This Notes is prepared mostly using NPTEL Videos of Embedded Systems, IIT Delhi. Some concepts are taken from Internet.

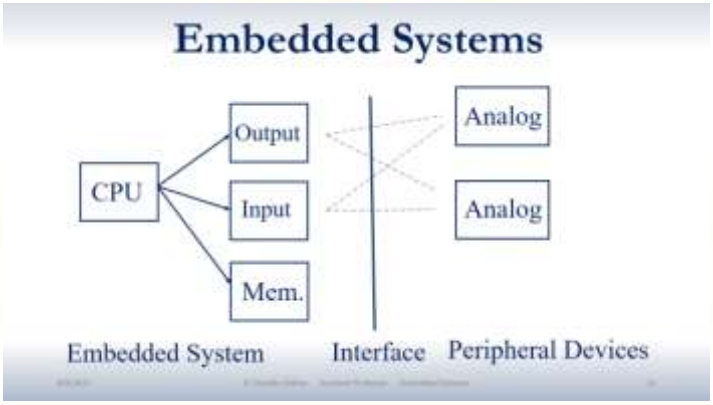
This notes is prepared to complete the syllabus in the given number of lectures and in the exam point of view. For more and detailed theory go through Mazidi(For overview of Embedded Systems) and Peatman(For PIC).

Embedded Systems - Overview

System: A system is an arrangement in which all its units assembled work together according to a set of rules. It can also be defined as a way of working, organizing or doing one or many tasks according to a fixed plan. For example, a watch is a time displaying system. Its components follow a set of rules to show time. If one of its parts fails, the watch will stop working. So we can say, in a system, all its subcomponents depend on each other.

What is an Embedded System? Most Important**********

- Any device that includes a computer, but is not itself a general- purpose computer.
- It includes hardware as well as software and it is a part of a larger system and is expected to function without human intervention.
- An embedded system is expected to respond, monitor as well as control external environment using sensors and actuators.
- So, basically what we are talking about is embedding a computer; embedding a computer into an appliance and, that computer is not expected to be used for general purpose computing.
- Since it is embedded into an appliance, it needs to interact with the external world with analog interfaces.
- And the model that I am showing is the simplest possible model of an embedded system.

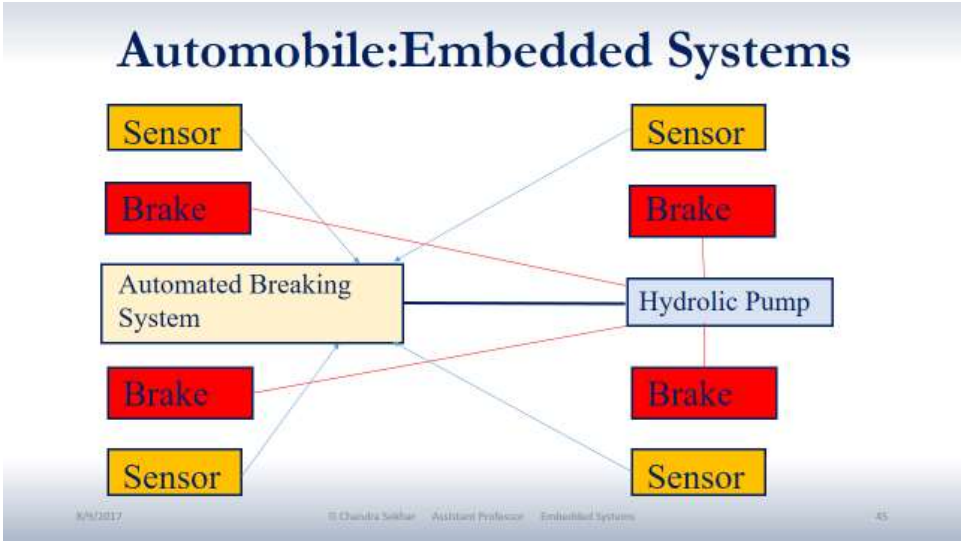


Examples of Embedded Systems:********

- Personal Digital Assistance (PDA) & Printer
- Cell Phone & Camera
- Automobile: Engine, Brakes, Dash etc.
- Television & Household Appliances

Automobile Embedded System example :

- Let us look an architecture of automatic breaking system; this is another aspect of an automobile.
- We have got sensors here, these sensors actually sensors the speed.
- And the breaks which are controlled by hydraulic pump and the embedded system in the automated breaking system which receives input from the sensors and then depending on the software that is running in the automated breaking system, actuates the hydraulic pump to control the break.
- So this is an example of a control system being implemented through the help of microcontrollers in an automobile.



Characteristics of Embedded Systems: Most Important**********

- Sophisticated functionality.
- Real time operation.
- Low manufacturing cost
- Low power Consumption
- Application dependent processors and not general purpose processors which we find in computers
- Restricted memory

Explanation:

What are the characteristics of embedded systems?

- First thing is, they all implement sophisticated functionality. The degree of sophistication can vary from appliance to appliance.
- They satisfy real time operation. Is it always true? It is not necessarily true and what is the real time operation, we shall comeback to this point slightly later on.
- They should have in many cases, low manufacturing cost, but cost itself is an issue which requires further closer examination.
- In many cases the appliances uses application dependent processors and not general purpose processors which we find in computers.
- They need to work with restricted memory

Manufacturing cost: There are two aspects;

- First aspect is what we call non- recurring engineering cost, which is actually the development cost into that system. The other aspect of the cost is production and marketing each unit. If we are targeting a mass market then what we need to optimize is a production marketing cost.
- But if you are trying to develop a very specialized application then we can invest in NRE as well as we may compensate set for high production cost.
- Say, for example, if I am designing an automated system for an aircraft, I can invest money for its development, I can use highly sophisticated equipments, but the same flexibility is not with me when I am designing a cell phone, a low cost cell phone aiming to serve a mass market. So the best technology choice will therefore depend on the number of units we plan to produce.

Real time operation: What is the real time operation?

- The basic definition is that operations must be completed by deadlines.
- So if I have a deadline, a real time operation must be completed within deadline.

- We have two kinds of real time deadlines;
    - hard real time deadlines and soft real time headlines and accordingly also, we classify real time systems.
- In a hard real time systems, we cannot really miss a deadline. If you are talking about an atomic reactor control, if I miss a deadline then there can be a catastrophe.
- On the other hand, for a soft real time systems, we can at times miss deadline. See for example, when we are playing a video on a laptop even if we cannot decode a frame in time, nothing catastrophic happens, only it disturbs you viewing experience.
- Many systems are also multi-rate that means those embedded systems are receiving inputs from the external worlds and inputs can come at different rates.
- So they need to handle different rate inputs and we call them therefore multi-rate systems.
- There also various **application dependent requirements**, in many cases, just take for example, an aircraft system with definitely need fault-tolerance
- Also for medical equipment when we are monitoring a critical patient using an embedded system, we do need fault-tolerance and reliability. Further, the systems must be safe; systems must avoid physical or economic damage to person as well as property.
- Further, if they are dedicated systems, then the design consideration is obviously different because they are not expected to be programmed on a regular basis. So what we say, the programmability of the systems would be really used during the right lifetime of the system.
- That means once programmed the systems are expected to execute infinitely for a large duration of time without users intervention. And they are expected to be programmed or designed for specific tasks and therefore they are basically what we call dedicated systems.

Different Types of Embedded Systems:

- Let us classify the examples we have seen into different types. General Computing Type, Control System Type, Signal Processing Type and Communication and Networking Type.
- Some are similar to general computing, like PDA, video games, set top boxes, automatic teller machines.
- Why they are similar to general computing?
- They are similar to general computing simply because if you take PDA, the majority of the tasks that you do is a restricted form of the task that you do on a computer.
- Similar thing is with the video games, you provide the input; the user provides the input and it expects some output.
- They are not really sensing external environment on its own as well as they are not activating any actuator on its own that would influence or change the external world.
- So, those devices are more like a general purpose computing machines; they respond to users input.
- Others, on the others side have got control systems whose basic job is that of sensing and actuating.
- The feedback control of real time system, various real time systems, I need a feedback control depending on the external input, I need the control to take some actions.
- And examples of these are vehicles engine fuel injections to be controlled, flight control, nuclear reactors. These are all examples of embedded systems which belong to the category of control systems.
- Next, we have signal processing because here the core job or basic focus is signal processing.

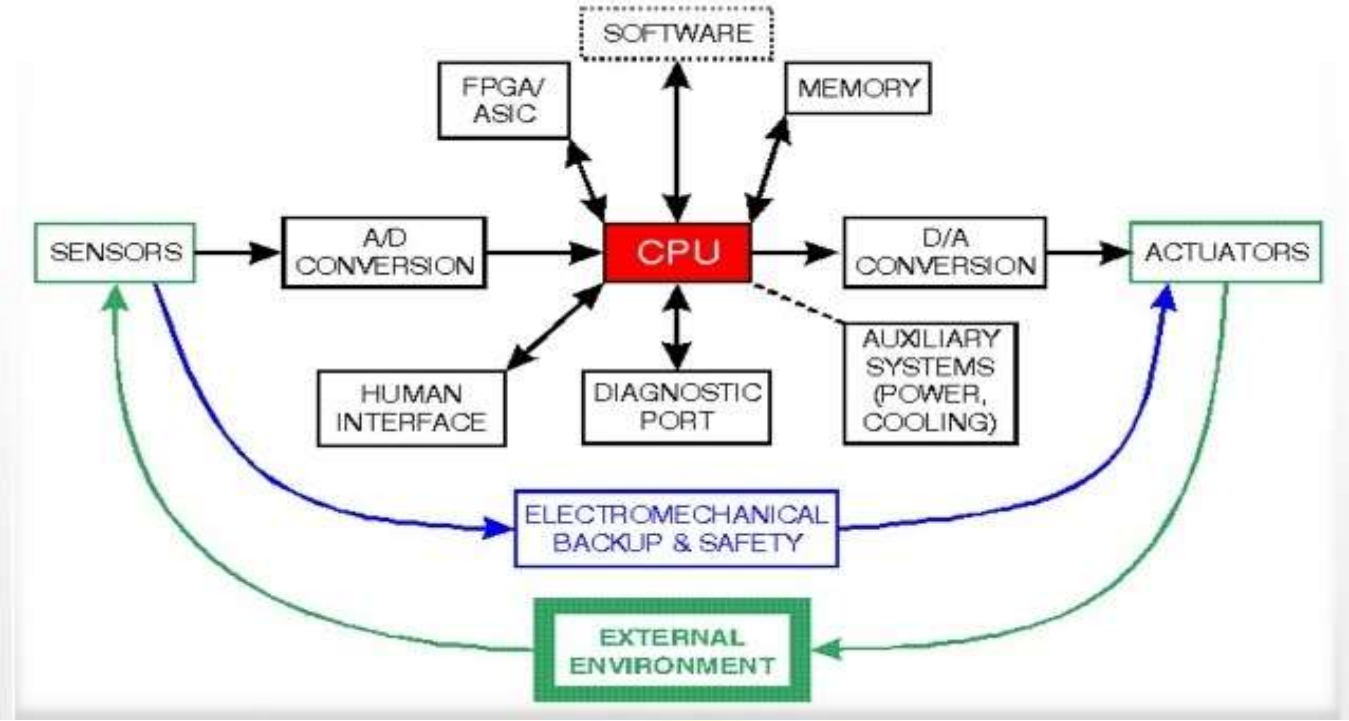
- Your MP3 players, your DVD players, radar control system; because in radar although there is a control system the basic job is processing of the data. Similarly, a sonar system; they are all example the signal processing systems.
- And communication and networking is another category of which the most common example is your cellular phones. And now, we are getting a number of internet appliance, in fact, the web enabled vending machine is an example of this kind of an internet appliance.

Nature of Embedded System Functions:

Control Laws, sequencing logic, signal processing, application specific interfacing, fault response

- So what are the different kinds of functions that an embedded system is expected to implement?
- First is, if it has got the actuation, sensing an actuation as a basic task it must realize some control law; it has to realize a control law.
- Second important issue is that there has to be a sequencing logic. This sequencing logic is obviously task specific and it is not a general purpose sequencing logic; it could have a task specific sequencing logic implemented into it.
- Third thing is that it should have signal processing if it is required and wherever and where we are interfacing an embedded system with external sensory input we need signal processing. So, in many cases, even when the signal processing is not core activity we need signal processing ability to deal with sensory inputs.
- Fourth one is application specific interfacing because application will tell us what kind of sensors and what kind of actuators to be interconnected and accordingly we should have that interfacing. This interfacing implies both hardware as well as software.
- Next thing is fault response; what happens when a fault occurs. A basic issue or basic design philosophy for fault response is what we know as, what we call graceful degradation. Catastrophic failure should not happen. The system should tell users that things are failing and gracefully it would degrade.
- Say, for example battery failure, there should be a message to the user saying that battery is low. So user can take some action. It should not suddenly stop its activity all of a sudden. So graceful degradation is another important function which is to be implemented.

Structure of an Embedded System****



- Let us look at now, a more complex structure of the embedded system.
- We have seen the simplest model and now we shall make it more complex because we have now understood; what are its requirements, we have also reviewed some of the examples.
- Now, let us look at a more complex example. So, what are the things which are involved here? What I have shown if you go back to the previous model; I have now expanded the basic block and have added something more to the basic block.
- In the basic block, earlier I had just shown the CPU, along with CPU now we are showing obviously the memory because memory will have the software to control the system. Also we are showing analog to digital converters and digital to analog converters.
- This AD conversion blocks actually provides an interface to the sensor here and DA conversion block actually provides interface to the actuators, because an embedded system which is situated in an environment is expected to receive sensor inputs and actuate the actuators to change the external environment.
- So, these two are very essential and integral component in majority of your embedded systems. Here I have shown an FPGA or ACIC block. Why? Because in many cases; my CPU may not have the ability to execute my software, satisfying real time constraints. Under those circumstances I might need special hardware to come or interfaced with my CPU. So that can be implemented on an FPGA and ACIC can be used along with the CPU.
- Now, let us look at other issues; one is obviously, on the CPU we need to implement, with the CPU with human interface; if you are talking about any kind of reading to be obtained through the embedded systems, any control functions to be altered by the human users I need a human interface. So, this interface becomes an important component. So you will find in many cases, you have an LCD display panel or a simple LED based informative colour codes by which the user can be informed of what is happening inside. Also there are diagnostic tools, why diagnostic tools? Because this, although this

systems are expected to work forever there are obviously probabilities of failure and if a failure occurs, how to trace that failure? Can it be repaired or simply it has to be taken out and thrown away? So, if it has to be repaired I need to have diagnostic tools to interface and check whether it is working. Second important thing why diagnostic tools are important; that when the system is starting up or system is working on, even on a continuous basis, it should do some self-check to know whether all the parts of it is functioning properly or not. Because if all the parts are not functioning properly, what can happen? It can actually do damage to the users, because of malfunctioning of some hardware components. So, it is also expected to do some self-checks at regular basics. So, diagnostics tools form also an essential component and you have the auxiliary systems which are to be dealt with power, because if there is a power dissipation, then cooling becomes an essential component.

- So, how to design the cooling circuit, how to take care of extra heat dissipations and this mechanical aspects of this design becomes important. Obviously the casing; the casing the whole system should be properly packaged. If it is not properly packaged and the packaging should be as per the requirements of the external environment in which the embedded system is expected to be pleased. So, this packaging becomes a very-very important issue and in fact, if you look at this packaging issue; this if your packaging is not properly designed even a good well design system can fail. Because then, because of the bad packaging the system can get, say for example, moisture. That moisture can go in and affect the electronics, then heat can affect electronics, so all these mechanical aspect of the design become extremely important. Although, in the course we shall not discuss those mechanical aspects of an embedded system design. But, please be conscious about the fact that these aspects are very-very important for any kind an appliances design and implementation.
- So now, if you know these as an architecture, so how to implement an embedded system? And this is exactly, will be the focus of our course. We shall learn more about what is presented in the slide. We shall discuss obviously the processing elements. The processing elements are basically your microprocessors and microcontrollers. We shall look at the peripheral devices because input and output devices become a critical component in this context. And also how to interface sensors and actuators and there also there are various kinds of interfacing protocols which can vary from one sensor to another sensor. Then you have got memory, also the bus design. So these are different aspects of an hardware of an embedded system and if you look into it, these aspects of the hardware are very-very similar to a general purpose computer. There is no basic difference conceptually from that of a general purpose computer. The only issue which is of importance is, these aspects: in a general purpose computer we tend to talk about standard input output devices although that set is getting expanded day by day, but we tend to talk about standard input output devices. But here, the set of input output devices are large because there can be different kinds of sensors and each sensor has got it's own characteristics.
- And therefore, I need to, you will find that particularly the processors which are targeted for embedded applications will have very sophisticated and a complex mechanisms; in many cases even simpler mechanisms not only complex mechanisms to interface with external devices and external IO devices in particular.

Advantages

- Easily Customizable
- Low power consumption
- Low cost
- Enhanced performance

Disadvantages

- High development effort
- Larger time to market

Embedded Systems - Processors

Processor is the heart of an embedded system. It is the basic unit that takes inputs and produces an output after processing the data. For an embedded system designer, it is necessary to have the knowledge of both microprocessors and microcontrollers.

Processors in a System

A processor has two essential units −

- Program Flow Control Unit (CU)
- Execution Unit (EU)

The CU includes a fetch unit for fetching instructions from the memory. The EU has circuits that implement the instructions pertaining to data transfer operation and data conversion from one form to another.

The EU includes the Arithmetic and Logical Unit (ALU) and also the circuits that execute instructions for a program control task such as interrupt, or jump to another set of instructions.

A processor runs the cycles of fetch and executes the instructions in the same sequence as they are fetched from memory.

Types of Processors

Processors can be of the following categories −

- General Purpose Processor (GPP)
  - Microprocessor
  - Microcontroller
  - Embedded Processor
  - Digital Signal Processor
  - Media Processor
- Application Specific System Processor (ASSP)
- Application Specific Instruction Processors (ASIPs)
- GPP core(s) or ASIP core(s) on either an Application Specific Integrated Circuit (ASIC) or a Very Large Scale Integration (VLSI) circuit.

Microprocessor*******

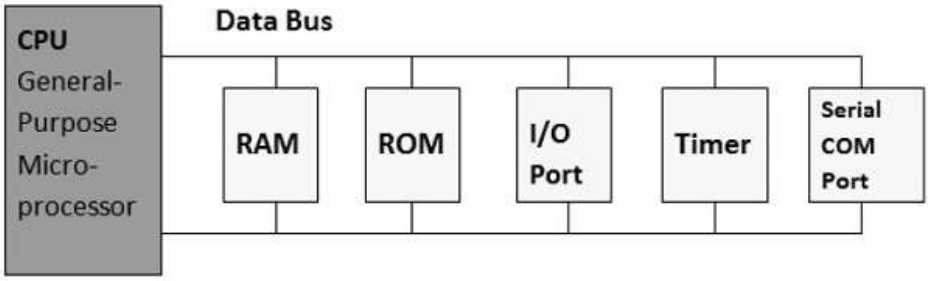
A microprocessor is a single VLSI chip having a CPU. In addition, it may also have other units such as coaches, floating point processing arithmetic unit, and pipelining units that help in faster processing of instructions.

Earlier generation microprocessors' fetch-and-execute cycle was guided by a clock frequency of order of ~1 MHz. Processors now operate at a clock frequency of 2GHz.

Microcontroller**********

A microcontroller is a single-chip VLSI unit (also called **microcomputer**) which, although having limited computational capabilities, possesses enhanced input/output capability and a number of on-chip functional units. Microcontrollers are particularly used in embedded systems for real-time control applications with on-chip program memory and devices.

CPU	RAM	ROM
I/O Port	Timer	Serial COM Port



A SIMPLE BLOCK DIAGRAM OF A MICROPROCESSOR

Microprocessor vs Microcontroller**********

Let us now take a look at the most notable differences between a microprocessor and a microcontroller.

Microprocessor	Microcontroller
Microprocessors are multitasking in nature. Can perform multiple tasks at a time. For example, on computer we can play music while writing text in text editor.	Single task oriented. For example, a washing machine is designed for washing clothes only.
RAM, ROM, I/O Ports, and Timers can be added externally and can vary in numbers.	RAM, ROM, I/O Ports, and Timers cannot be added externally. These components are to be embedded together on a chip and are fixed in numbers.
Designers can decide the number of memory or I/O ports needed.	Fixed number for memory or I/O makes a microcontroller ideal for a limited but specific task.
External support of external memory and I/O ports makes a microprocessor-based system heavier and costlier.	Microcontrollers are lightweight and cheaper than a microprocessor.
External devices require more space and their power consumption is higher.	A microcontroller-based system consumes less power and takes less space.

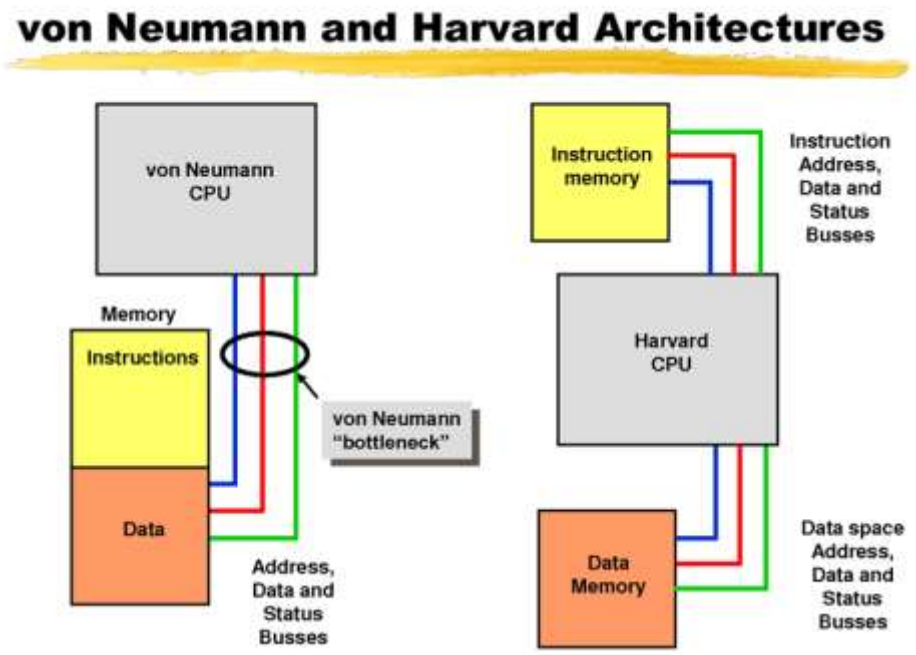
Embedded Systems - Architecture Types****

When data and code lie in different memory blocks, then the architecture is referred as **Harvard architecture**. In case data and code lie in the same memory block, then the architecture is referred as **Von Neumann architecture**.

Von Neumann Architecture:****

The Von Neumann architecture was first proposed by a computer scientist John von Neumann. In this architecture, one data path or bus exists for both instruction and data. As a result, the CPU does one operation at a time. It either fetches an instruction from memory, or performs read/write operation on data. So an instruction fetch and a data operation cannot occur simultaneously, sharing a common bus. Von-Neumann architecture supports simple hardware. It allows the

use of a single, sequential memory. Today's processing speeds vastly outpace memory access times, and we employ a very fast but small amount of memory (cache) local to the processor.



Harvard Architecture******

The Harvard architecture offers separate storage and signal buses for instructions and data. This architecture has data storage entirely contained within the CPU, and there is no access to the instruction storage as data. Computers have separate memory areas for program instructions and data using internal data buses, allowing simultaneous access to both instructions and data. Programs needed to be loaded by an operator; the processor could not boot itself. In a Harvard architecture, there is no need to make the two memories share properties.

Von-Neumann Architecture vs Harvard Architecture******

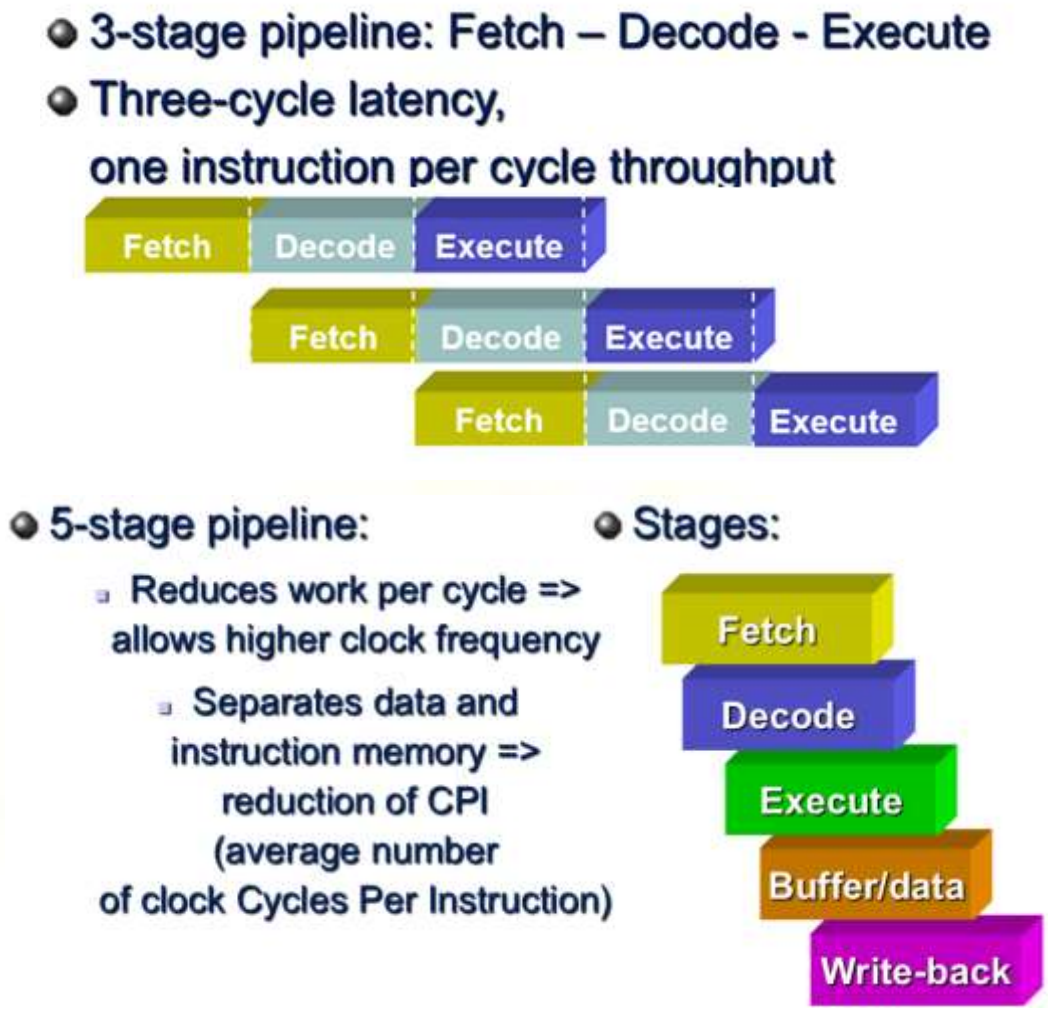
The following points distinguish the Von Neumann Architecture from the Harvard Architecture.

Von-Neumann Architecture	Harvard Architecture
Single memory to be shared by both code and data.	Separate memories for code and data.
Processor needs to fetch code in a separate clock cycle and data in another clock cycle. So it requires two sets of clock cycles.	Single set of clock cycles is sufficient, as separate buses are used to access code and data.
Pipelining is not Possible	Pipelining is Possible
Simple in design.	Complex in design.

Pipeline Organization:*****

- Pipelining involves the concept of overlapped execution of instructions
- Increases speed – most instructions executed in single cycle.

- We can perform pipelining if the instruction has more than 2 stages for the completion of executing and instruction.
  - 3-stage (ARM7TDMI and earlier)
  - 5-stage (ARMS, ARM9TDMI)
  - 6-stage (ARM10TDMI)
- Pipeline flushed and refilled on branch, causing execution to slow down
- Special features in instruction set eliminate small jumps in code to obtain the best flow through pipeline



CISC and RISC*****

CISC is a Complex Instruction Set Computer. It is a computer that can address a large number of instructions. In the early 1980s, computer designers recommended that computers should use fewer instructions with simple constructs so that they can be executed much faster within the CPU without having to use memory. Such computers are classified as Reduced Instruction Set Computer or RISC.

CISC vs RISC*****

The following points differentiate a CISC from a RISC −

CISC	RISC

Larger set of instructions. Easy to program	Smaller set of Instructions. Difficult to program.
Simpler design of compiler, considering larger set of instructions.	Complex design of compiler.
Many addressing modes causing complex instruction formats.	Few addressing modes, fix instruction format.
Instruction length is variable.	Instruction length varies.
Higher clock cycles per second.	Low clock cycle per second.
Emphasis is on hardware.	Emphasis is on software.
Control unit implements large instruction set using micro-program unit.	Each instruction is to be executed by hardware.
Slower execution, as instructions are to be read from memory and decoded by the decoder unit.	Faster execution, as each instruction is to be executed by hardware.
Pipelining is not possible.	Pipelining of instructions is possible, considering single clock cycle.

Embedded Systems - Tools & Peripherals

Compilers and Assemblers

Compiler: A compiler is a computer program (or a set of programs) that transforms the source code written in a programming language (the source language) into another computer language (normally binary format). The most common reason for conversion is to create an executable program. The name "compiler" is primarily used for programs that translate the source code from a highlevel programming language to a low-level language (e.g., assembly language or machine code).

Cross-Compiler: If the compiled program can run on a computer having different CPU or operating system than the computer on which the compiler compiled the program, then that compiler is known as a cross-compiler.

Decompiler: A program that can translate a program from a low-level language to a high-level language is called a decompiler.

Language Converter: A program that translates programs written in different high-level languages is normally called a language translator, source to source translator, or language converter.

A compiler is likely to perform the following operations −

- Preprocessing
- Parsing
- Semantic Analysis (Syntax-directed translation)
- Code generation
- Code optimization

Assemblers: An assembler is a program that takes basic computer instructions (called as assembly language) and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. An assembler creates object code by translating assembly instruction mnemonics into opcodes, resolving symbolic names to memory locations. Assembly language uses a mnemonic to represent each low-level machine operation (opcode).

Debugging Tools in an Embedded System

Debugging is a methodical process to find and reduce the number of bugs in a computer program or a piece of electronic hardware, so that it works as expected. Debugging is difficult when subsystems are tightly coupled, because a small change in one subsystem can create bugs in another. The debugging tools used in embedded systems differ greatly in terms of their development time and debugging features. We will discuss here the following debugging tools −

- Simulators
- Microcontroller starter kits
- Emulator

Simulators

Code is tested for the MCU / system by simulating it on the host computer used for code development. Simulators try to model the behavior of the complete microcontroller in software.

Functions of Simulators

A simulator performs the following functions −

- Defines the processor or processing device family as well as its various versions for the target system.
- Monitors the detailed information of a source code part with labels and symbolic arguments as the execution goes on for each single step.
- Provides the status of RAM and simulated ports of the target system for each single step execution.
- Monitors system response and determines throughput.
- Provides trace of the output of contents of program counter versus the processor registers.
- Provides the detailed meaning of the present command.
- Monitors the detailed information of the simulator commands as these are entered from the keyboard or selected from the menu.
- Supports the conditions (up to 8 or 16 or 32 conditions) and unconditional breakpoints.
- Provides breakpoints and the trace which are together the important testing and debugging tool.
- Facilitates synchronizing the internal peripherals and delays.

Microcontroller Starter Kit

A microcontroller starter kit consists of −

- Hardware board (Evaluation board)
- In-system programmer
- Some software tools like compiler, assembler, linker, etc.
- Sometimes, an IDE and code size limited evaluation version of a compiler.

A big advantage of these kits over simulators is that they work in real-time and thus allow for easy input/output functionality verification. Starter kits, however, are completely sufficient and the cheapest option to develop simple microcontroller projects.

Emulators:

An emulator is a hardware kit or a software program or can be both which emulates the functions of one computer system (the guest) in another computer system (the host), different from the first one, so that the emulated behavior closely resembles the behavior of the real system (the guest).

Emulation refers to the ability of a computer program in an electronic device to emulate (imitate) another program or device. Emulation focuses on recreating an original computer environment. Emulators have the ability to maintain a closer connection to the authenticity of the digital object. An emulator helps the user to work on any kind of application or operating system on a platform in a similar way as the software runs as in its original environment.

Peripheral Devices in Embedded Systems:*******

Embedded systems communicate with the outside world via their peripherals, such as following &mins;

- Serial Communication Interfaces (SCI) like RS-232, RS-422, RS-485, etc.
- Synchronous Serial Communication Interface like I2C, SPI, SSC, and ESSI
- Universal Serial Bus (USB)
- Multi Media Cards (SD Cards, Compact Flash, etc.)
- Networks like Ethernet, LonWorks, etc.
- Fieldbuses like CAN-Bus, LIN-Bus, PROFIBUS, etc.
- Timers like PLL(s), Capture/Compare and Time Processing Units.
- Discrete IO aka General Purpose Input/Output (GPIO)
- Analog to Digital/Digital to Analog (ADC/DAC)
- Debugging like JTAG, ISP, ICSP, BDM Port, BITP, and DP9 ports

Criteria for Choosing Microcontroller:*******

While choosing a microcontroller, make sure it meets the task at hand and that it is cost effective. We must see whether an 8-bit, 16-bit or 32-bit microcontroller can best handle the computing needs of a task. In addition, the following points should be kept in mind while choosing a microcontroller −

- **Speed** − What is the highest speed the microcontroller can support?
- **Packaging** − Is it 40-pin DIP (Dual-inline-package) or QFP (Quad flat package)? This is important in terms of space, assembling, and prototyping the end-product.
- **Power Consumption** − This is an important criteria for battery-powered products.
- **Amount of RAM and ROM** on the chip.
- **Count of I/O pins and Timers** on the chip.
- **Cost per Unit** − This is important in terms of final cost of the product in which the microcontroller is to be used.
- **Total Silicon Area** – Area of size of the chip needed to fabricate all the components should be minimized

Further, make sure you have tools such as compilers, debuggers, and assemblers, available with the microcontroller. The most important of all, you should purchase a microcontroller from a reliable source.

Embedded Systems - 8051 Microcontroller

Brief History of 8051:

The first microprocessor **4004** was invented by Intel Corporation. **8085** and **8086** microprocessors were also invented by Intel. In 1981, Intel introduced an 8-bit microcontroller called the **8051**. It is a basic type of micro controller based on Harvard & RISC Architectures and developed primarily for use in embedded systems technology. Normally, this microcontroller was developed using NMOS technology, which requires more power to operate. Therefore, Intel redesigned Microcontroller 8051 using CMOS technology and their updated versions came with a letter C in their name, for instance an 80C51 it is an 8 bit microcontroller. These latest Microcontrollers requires less power to operate as compared to their previous versions. The 8051 Microcontroller has two buses and can have two memory spaces of 64K X 8 size for program and data units. It has an 8 bit processing unit and 8 bit accumulator units. It was referred as **system on a chip** because it had 128 bytes of RAM, 4K byte of on-chip ROM, two timers, one serial port, and 4 ports (8-bit wide), all on a single chip. When it became widely popular, Intel allowed other manufacturers to make and market different flavors of 8051 with its code compatible with 8051. It means that if you write your program for one flavor of 8051, it will run on other flavors too, regardless of the manufacturer. This has led to several versions with different speeds and amounts of on-chip RAM. The 8051 microcontrollers work with 8-bit data bus. So they can support external data memory up to 64K and external program memory of 64k at best. Collectively, 8051 microcontrollers can address 128k of external memory.

8051 Flavors / Members

- **8052 microcontroller** − 8052 has all the standard features of the 8051 microcontroller as well as an extra 128 bytes of RAM and an extra timer. It also has 8K bytes of on-chip program ROM instead of 4K bytes.
- **8031 microcontroller** − It is another member of the 8051 family. This chip is often referred to as a ROM-less 8051, since it has 0K byte of on-chip ROM. You must add external ROM to it in order to use it, which contains the program to be fetched and executed. This program can be as large as 64K bytes. But in the process of adding external ROM to the 8031, it lost 2 ports out of 4 ports. To solve this problem, we can add an external I/O to the 8031

Comparison between 8051 Family Members

The following table compares the features available in 8051, 8052, and 8031.

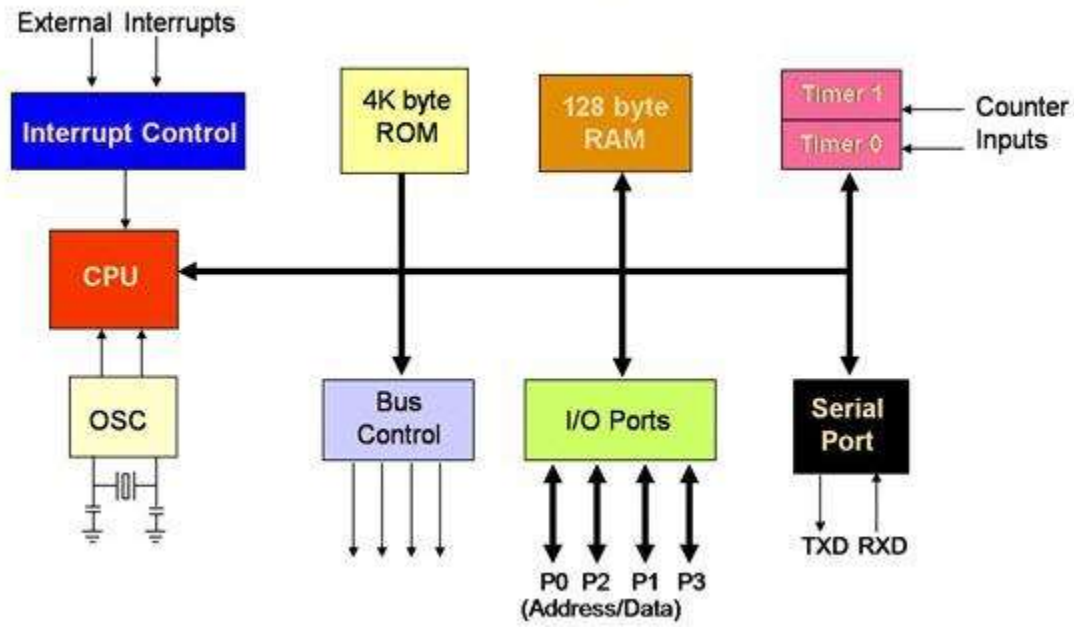
Feature	8051	8052	8031
ROM(bytes)	4K	8K	0K
RAM(bytes)	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

Features of 8051 Microcontroller

An 8051 microcontroller comes bundled with the following features −

- 64K bytes on-chip program memory (ROM)
- 128 bytes on-chip data memory (RAM)
- Four register banks
- 128 user defined software flags
- 8-bit bidirectional data bus
- 16-bit unidirectional address bus
- 32 general purpose registers each of 8-bit
- 16 bit Timers (usually 2, but may have more or less)
- Three internal and two external Interrupts
- Four 8-bit ports,(short model have two 8-bit ports)
- 16-bit program counter and data pointer
- 8051 may also have a number of special features such as UARTs, ADC, Op-amp, etc.

Architecture of 8051 Microcontroller: Most Imp.*******



Description of Functional Blocks:*******

Let us have a look at each part or block of this Architecture of microcontroller.

Central Processor Unit (CPU): As we know that the CPU is the brain of any processing device of the microcontroller. It monitors and controls all operations that are performed on the Microcontroller units. The User has no control over the work of the CPU directly. It reads program written in ROM memory and executes them and do the expected task of that application.

Interrupts: As its name suggests, Interrupt is a subroutine call that interrupts of the microcontrollers main operations or work and causes it to execute any other program, which is more important at the time of operation. The feature of Interrupt is very useful as it helps in case of emergency operations. An Interrupts gives us a mechanism to put on hold the ongoing operations, execute a subroutine and then again resumes to another type of operations. The Microcontroller 8051 can be configured in such a way that it temporarily terminates or pause the main program at the occurrence of interrupts. When a subroutine is completed, then the execution of main program starts. Generally five interrupt sources are there in 8051 Microcontroller. There are 5 vectored interrupts are shown in below

- INTO
- TFO
- INT1
- TF1
- R1/T1

Out of these, $(INT0)^{-}$ and $(INT1)^{-}$ are external interrupts that could be negative edge triggered or low level triggered. When All these interrupts are activated, set the corresponding flags except for serial interrupts. The interrupt flags are cleared when the processor branches to the interrupt service routine (ISR). The external interrupt flags are cleared when the processor branches to the interrupt service routine, provides the interrupt is a negative edge triggered whereas the timers and serial port interrupts two of them are external interrupts, two of them are timer interrupts and one serial port interrupt terminal in general.

Memory: Microcontroller requires a program which is a collection of instructions. This program tells microcontroller to do specific tasks. These programs require a memory on which these can be saved and read by Microcontroller to perform specific operations of a particular task. The memory which is used to store the program of the microcontroller is known as code memory or Program memory of applications. It is known as ROM memory of microcontroller also requires a memory to store data or operands temporarily of the micro controller. The data memory of the 8051 is used to store data temporarily for operation is known RAM memory. 8051 microcontroller has 4K of code memory or program memory, that has 4KB ROM and also 128 bytes of data memory of RAM.

BUS: Basically Bus is a collection of wires which work as a communication channel or medium for transfer of Data. These buses consists of 8, 16 or more wires of the microcontroller. Thus, these can carry 8 bits, 16 bits simultaneously. Hire two types of buses that are shown in below

- Address Bus
- Data Bus

Address Bus: Microcontroller 8051 has a 16 bit address bus for transferring the data. It is used to address memory locations and to transfer the address from CPU to Memory of the microcontroller. It has five addressing modes that are

- Immediate addressing modes.
- Bank address (or) Register addressing mode.
- Direct Addressing mode.
- Register indirect addressing mode.
- Indexed or memory indirect Addressing mode

Data Bus: Microcontroller 8051 has 8 bits of the data bus, which is used to carry data of particular applications.

Oscillator: Generally, we know that the microcontroller is a device, therefore it requires clock pulses for its operation of microcontroller applications. For this purpose, microcontroller 8051 has an on-chip oscillator which works as a clock source for Central Processing Unit of the microcontroller. The output pulses of oscillator are stable. Therefore, it enables synchronized work of all parts of the 8051 Microcontroller. We have to connect a crystal between XTAL1 and XTAL2 pins of the 8051 which are directly connected to the on-chip oscillator. Typical frequency of 8051 MC is 11.0592MHz

Input/Output Port: Normally microcontroller is used in embedded systems to control the operation of machines in the microcontroller. Therefore, to connect it to other machines, devices or peripherals we require I/O interfacing ports in the microcontroller interface. For this purpose microcontroller 8051 has 4 input, output ports to connect it to the other peripherals. Out of these 4 ports, 2 ports are used for address and data where 8bits of data and 8bits of lower byte of address are multiplexed.

Timers/Counters: 8051 microcontroller has two 16 bit timers and counters. These counters are again divided into a 8 bit register. The timers are used for measurement of intervals to determine the pulse width of pulses.

Applications of 8051 Microcontroller:

Some of the applications of 8051 is mainly used in daily life & industrial applications also some of that applications are shown below

- Light sensing and controlling devices
- Temperature sensing and controlling devices
- Fire detections and safety devices
- Automobile applications
- Defense applications

Some industrial applications of micro controller and its applications

- Industrial instrumentation devices
- Process control devices

Some of 8051 microcontroller devices are used in measurement applications

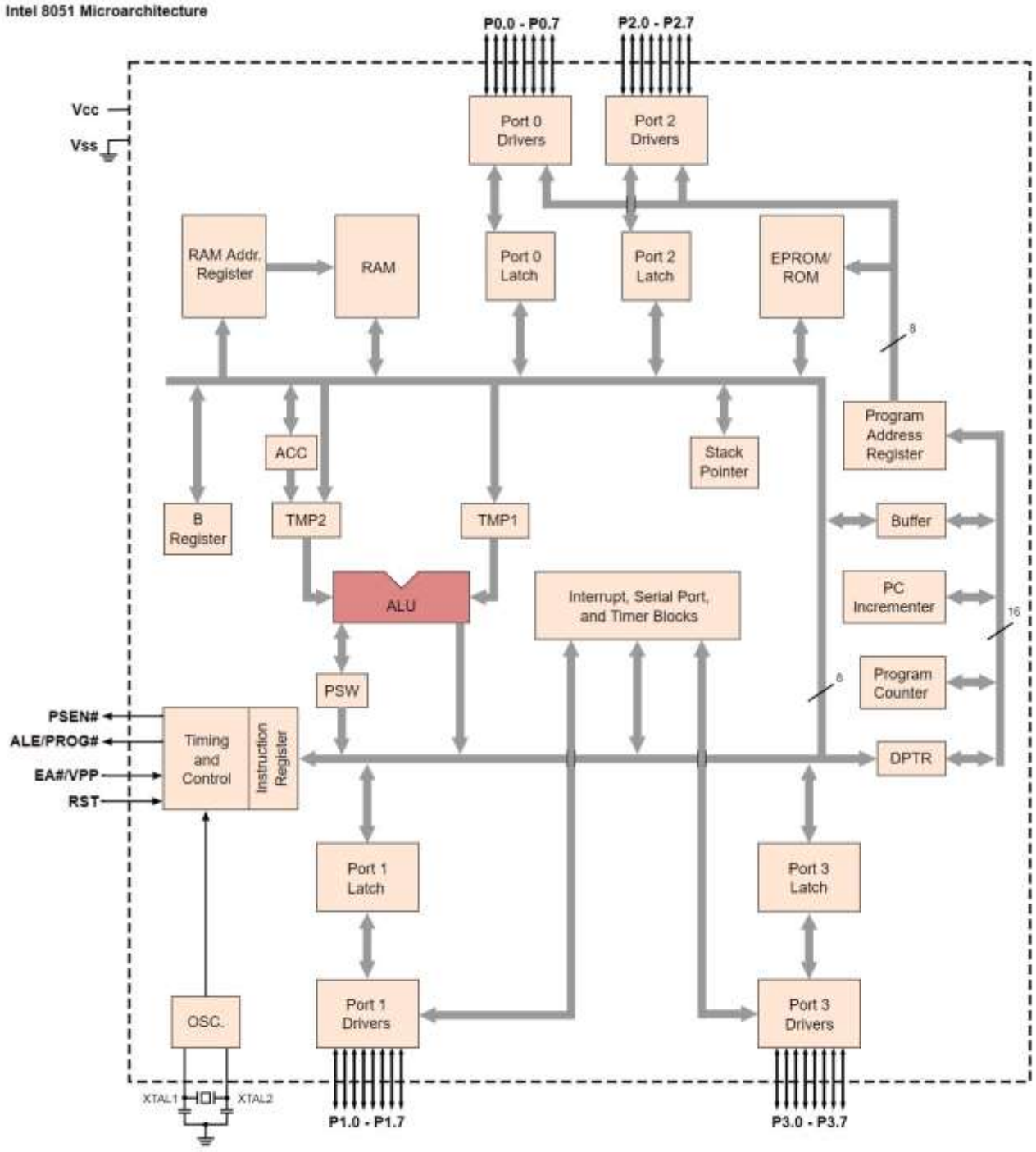
- Voltmeter applications
- Measuring and revolving objects
- Current meter objects
- Hand held metering system

8051 Microcontroller Applications in Embedded Systems

The applications of 8051 microcontroller involves in 8051 based projects. The list of 8051 projects is listed below.

- Arduino Managed High Sensitive LDR based Power Saver for Street Light Control System
- The Temperature Humidity Monitoring System of Soil Based on Wireless Sensor Networks using Arduino
- RFID based Electronic Passport System for Easy Governance using Arduino
- Arduino based RFID Sensed Device Access
- Arduino based DC Motor Speed Control
- Arduino Based Line Following Robot
- Zigbee based Automatic Meter Reading System
- GSM based Electricity Energy Meter Billing with Onsite Display
- Android Phone Speech Recognition Sensed Voice Command based Notice Board Display
- Parking Availability Indication System
- Voice Controlled Home Appliances
- Remote Control Home Appliances
- PC Mouse operated Electrical Load Control Using VB Application
- Solar Highway Lighting System with Auto Turn Off in Daytime
- 8051 Microcontroller based Wireless Energy Meter
- Farmer Friendly Solar Based Electric Fence for Deterring Cattles
- Vehicle Movement Sensed Streetlight with Daytime auto off Features

Detailed Architechture of 8051 Microcontroller:



From this we have to explain how CPU will work for Arithmetic and logic operations and also how it will generate timing & control signals for the execution of instructions. We will see how an addition operation will be performed by the CPU.

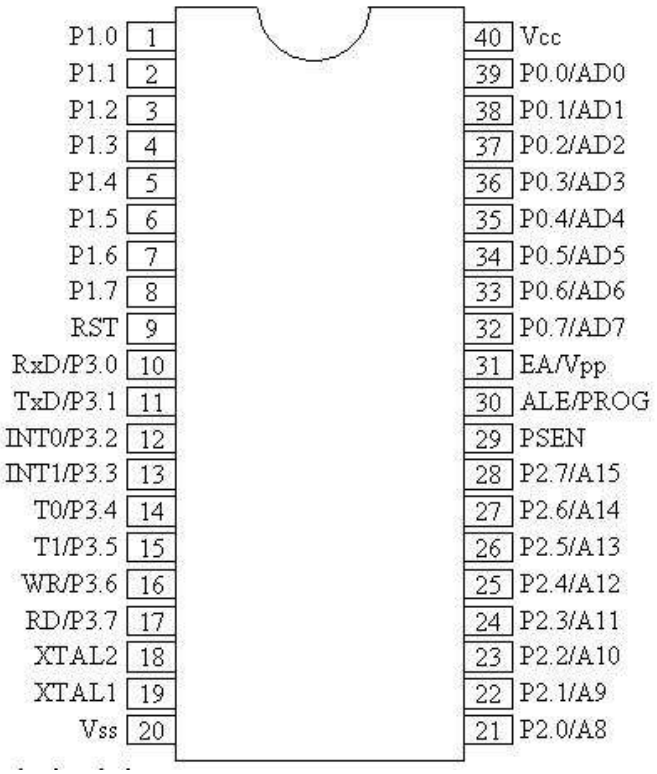
2008 XXH ADD A , B ; When 8051 start executing this instruction PC will change from 2008 to 2009 and opcode (XXH) loaded at ROM or program memory location will be loaded into Instruction register. Then the contents or opcode present in the IR are decoded and the corresponding timing and control signals are generated. Here timing and control signals are generated for addition of contents in A & B as follows:

1. Transfer the data or content in Accumulator to Temporary register 2, i.e. Temp2 ← (A)

2. Transfer the data or content in B register to Temporary register 1, i.e. Temp1 ← (B)
3. Perform the addition operation in ALU (Arithmetic and Logic Unit) and store the result in the Accumulator. And also set the flags in the PSW (Program status word) based on the result obtained. (A) ← Temp1+Temp2
4. For all the above steps required timing and control signals are generated and the signals will be transmitted to the registers, memory blocks and peripherals through the address and data path.

Embedded System - I/O Programming

In 8051, I/O operations are done using four ports and 40 pins. The following pin diagram shows the details of the 40 pins. I/O operation port reserves 32 pins where each port has 8 pins. The other 8 pins are designated as V_{cc}, GND, XTAL1, XTAL2, RST, EA (bar), ALE/PROG (bar), and PSEN (bar).
It is a 40 Pin PDIP (Plastic Dual Inline Package)



Note − In a DIP package, you can recognize the first pin and the last pin by the cut at the middle of the IC. The first pin is on the left of this cut mark and the last pin (i.e. the 40th pin in this case) is to the right of the cut mark.

I/O Ports and their Functions

The four ports P0, P1, P2, and P3, each use 8 pins, making them 8-bit ports. Upon RESET, all the ports are configured as inputs, ready to be used as input ports. When the first 0 is written to a port, it becomes an output. To reconfigure it as an input, a 1 must be sent to a port.

Port 0 (Pin No 32 – Pin No 39)

It has 8 pins (32 to 39). It can be used for input or output. Unlike P1, P2, and P3 ports, we normally connect P0 to 10K-ohm pull-up resistors to use it as an input or output port being an open drain.
It is also designated as AD0-AD7, allowing it to be used as both address and data. In case of 8031 (i.e. ROMless Chip), when we need to access the external ROM, then P0 will be used for both Address and Data Bus. ALE

(Pin no 31) indicates if P0 has address or data. When ALE = 0, it provides data D0-D7, but when ALE = 1, it has address A0-A7. In case no external memory connection is available, P0 must be connected externally to a 10K-ohm pull-up resistor.

```
MOV A,#0FFH  ;(comments: A=FFH(Hexadecimal  i.e. A=1111 1111)
MOV P0,A     ;(Port0 have 1's on every pin so that it works as Input)
```

Port 1 (Pin 1 through 8)

It is an 8-bit port (pin 1 through 8) and can be used either as input or output. It doesn't require pull-up resistors because they are already connected internally. Upon reset, Port 1 is configured as an input port. The following code can be used to send alternating values of 55H and AAH to Port 1.

```
; Toggle alternate bits of P1 continuously
START: MOV     A,#55
BACK:  MOV     P1,A
       ACALL   DELAY
       CPL     A      ; Complement(invert) reg. A
       SJMP    BACK
; Delay Subroutine
Delay:   MOV     R1, #0FFH          ; R1 = Counter 1 = 255
LOOP2:  MOV     R2, #0FFH          ; R2 = Counter 2 = 255
LOOP1:  DJNZ    R2, LOOP1
        DJNZ    R1, LOOP2
        RET
```

If Port 1 is configured to be used as an output port, then to use it as an input port again, program it by writing 1 to all of its bits as in the following code.

```
;Toggle all bits of continuously
MOV     A ,#0FFH    ;A = FF hex
MOV     P1,A        ;Make P1 an input port
MOV     A,P1        ;get data from P1
MOV     R7,A        ;save it in Reg R7
ACALL   DELAY       ;wait
MOV     A,P1        ;get another data from P1
MOV     R6,A        ;save it in R6
ACALL   DELAY       ;wait
MOV     A,P1        ;get another data from P1
MOV     R5,A        ;save it in R5
```

Port 2 (Pins 21 through 28)

Port 2 occupies a total of 8 pins (pins 21 through 28) and can be used for both input and output operations. Just as P1 (Port 1), P2 also doesn't require external Pull-up resistors because they are already connected internally. It must be used along with P0 to provide the 16-bit address for the external memory. So it is also designated as (A0–A7), as shown in the pin diagram. When the 8051 is connected to an external memory, it provides path for upper 8-bits of 16-bits address, and it cannot be used as I/O. Upon reset, Port 2 is configured as an input port. The following code can be used to send alternating values of 55H and AAH to port 2.

```
;Toggle all bits of continuously
MOV     A,#55
BACK:
MOV     P2,A
ACALL   DELAY
CPL     A        ; complement(invert) reg. A
SJMP    BACK
```

If Port 2 is configured to be used as an output port, then to use it as an input port again, program it by writing 1 to all of its bits as in the following code.

```
;Get a byte from P2 and send it to P1
MOV    A,#0FFH    ;A = FF hex
MOV    P2,A       ;make P2 an input port
BACK:
MOV    A,P2       ;get data from P2
MOV    P1,A       ;send it to Port 1
SJMP   BACK       ;keep doing that
```

Port 3 (Pins 10 through 17)

It is also of 8 bits and can be used as Input/Output. This port provides some extremely important signals. P3.0 and P3.1 are RxD (Receiver) and TxD (Transmitter) respectively and are collectively used for Serial Communication. P3.2 and P3.3 pins are used for external interrupts. P3.4 and P3.5 are used for timers T0 and T1 respectively. P3.6 and P3.7 are Write (WR) and Read (RD) pins. These are active low pins, means they will be active when 0 is given to them and these are used to provide Read and Write operations to External ROM in 8031 based systems.

P3 Bit	Function	Pin
P3.0	RxD	10
P3.1 <	TxD	11
P3.2 <	Complement of INT0	12
P3.3 <	INT1	13
P3.4 <	T0	14
P3.5 <	T1	15
P3.6 <	WR	16
P3.7 <	Complement of RD	17

Dual Role of Port 0 and Port 2

- **Dual role of Port 0** − Port 0 is also designated as AD0–AD7, as it can be used for both data and address handling. While connecting an 8051 to external memory, Port 0 can provide both address and data. The 8051 microcontroller then multiplexes the input as address or data in order to save pins.

- **Dual role of Port 2** − Besides working as I/O, Port P2 is also used to provide 16-bit address bus for external memory along with Port 0. Port P2 is also designated as (A8– A15), while Port 0 provides the lower 8-bits via A0–A7. In other words, we can say that when an 8051 is connected to an external memory (ROM) which can be maximum up to 64KB and this is possible by 16 bit address bus because we know 216 = 64KB. Port2 is used for the upper 8-bit of the 16 bits address, and it cannot be used for I/O and this is the way any Program code of external ROM is addressed.

Hardware Connection of Pins

- **V_{cc}** − Pin 40 provides supply to the Chip and it is +5 V.
- **Gnd** − Pin 20 provides ground for the Reference.
- **XTAL1, XTAL2 (Pin no 18 & Pin no 19)** − 8051 has on-chip oscillator but requires external clock to run it. A quartz crystal is connected between the XTAL1 & XTAL2 pin of the chip. This crystal also needs two capacitors of 30pF for generating a signal of desired frequency. One side of each capacitor is connected to ground. 8051 IC is available in various speeds and it all depends on this Quartz crystal, for example, a 20 MHz microcontroller requires a crystal with a frequency no more than 20 MHz.
- **RST (Pin No. 9)** − It is an Input pin and active High pin. Upon applying a high pulse on this pin, that is 1, the microcontroller will reset and terminate all activities. This process is known as **Power-On Reset**. Activating a power-on reset will cause all values in the register to be lost. It will set a program counter to all 0's. To ensure a valid input of Reset, the high pulse must be high for a minimum of two machine cycles before it is allowed to go low, which depends on the capacitor value and the rate at which it charges. (**Machine Cycle** is the minimum amount of frequency a single instruction requires in execution).
- **EA or External Access (Pin No. 31)** − It is an input pin. This pin is an active low pin; upon applying a low pulse, it gets activated. In case of microcontroller (8051/52) having on-chip ROM, the EA (bar) pin is connected to V_{cc}. But in an 8031 microcontroller which does not have an on-chip ROM, the code is stored in an external ROM and then fetched by the microcontroller. In this case, we must connect the (pin no 31) EA to Gnd to indicate that the program code is stored externally.
- **PSEN or Program store Enable (Pin No 29)** − This is also an active low pin, i.e., it gets activated after applying a low pulse. It is an output pin and used along with the EA pin in 8031 based (i.e. ROMLESS) Systems to allow storage of program code in external ROM.
- **ALE or (Address Latch Enable)** − This is an Output Pin and is active high. It is especially used for 8031 IC to connect it to the external memory. It can be used while deciding whether P0 pins will be used as Address bus or Data bus. When ALE = 1, then the P0 pins work as Data bus and when ALE = 0, then the P0 pins act as Address bus.

I/O Ports and Bit Addressability

It is a most widely used feature of 8051 while writing code for 8051. Sometimes we need to access only 1 or 2 bits of the port instead of the entire 8-bits. 8051 provides the capability to access individual bits of the ports. While accessing a port in a single-bit manner, we use the syntax "SETB X. Y" where X is the port number (0 to 3), and Y is a bit number (0 to 7) for data bits D0-D7 where D0 is the LSB and D7 is the MSB. For example, "SETB P1.5" sets high bit 5 of port 1.

The following code shows how we can toggle the bit P1.2 continuously.

```
AGAIN:
SETB    P1.2
```

```
ACALL   DELAY
CLR     P1.2
ACALL   DELAY
SJMP    AGAIN
```

Single-Bit Instructions

Instructions	Function
SETB bit	Set the bit (bit = 1)
CLR bit	clear the bit (bit = 0)
CPL bit	complement the bit (bit = NOT bit)
JB bit, target	jump to target if bit = 1 (jump if bit)
JNB bit, target	jump to target if bit = 0 (jump if no bit)
JBC bit, target	jump to target if bit = 1, clear bit (jump if bit, then clear)

Embedded Systems - Terms

Program Counter

The Program Counter is a 16- or 32-bit register which contains the address of the next instruction to be executed. The PC automatically increments to the next sequential memory location every time an instruction is fetched. Branch, jump, and interrupt operations load the Program Counter with an address other than the next sequential location.

Activating a power-on reset will cause all values in the register to be lost. It means the value of the PC (program counter) is 0 upon reset, forcing the CPU to fetch the first opcode from the ROM memory location 0000. It means we must place the first byte of upcode in ROM location 0000 because that is where the CPU expects to find the first instruction

Reset Vector

The significance of the reset vector is that it points the processor to the memory address which contains the firmware's first instruction. Without the Reset Vector, the processor would not know where to begin execution. Upon reset, the processor loads the Program Counter (PC) with the reset vector value from a predefined memory location. On CPU08 architecture, this is at location $FFFE:$FFFF.

When the reset vector is not necessary, developers normally take it for granted and don’t program into the final image. As a result, the processor doesn't start up on the final product. It is a common mistake that takes place during the debug phase.

Stack Pointer:

Stack is implemented in RAM and a CPU register is used to access it called SP (Stack Pointer) register. SP register is an 8-bit register and can address memory addresses of range 00h to FFh.

Initially, the SP register contains value 07 to point to location 08 as the first location being used for the stack by the 8051.

When the content of a CPU register is stored in a stack, it is called a PUSH operation. When the content of a stack is stored in a CPU register, it is called a POP operation. In other words, a register is pushed onto the stack to save it and popped off the stack to retrieve it.

Infinite Loop

An infinite loop or an endless loop can be identified as a sequence of instructions in a computer program that executes endlessly in a loop, because of the following reasons −

- loop with no terminating condition.
- loop with a terminating condition that can never be met.
- loop with a terminating condition that causes the loop to start over.

Such infinite loops normally caused older operating systems to become unresponsive, as an infinite loop consumes all the available processor time. I/O operations waiting for user inputs are also called "infinite loops". One possible cause of a computer "freezing" is an infinite loop; other causes include **deadlock** and **access violations**.

Embedded systems, unlike a PC, never "exit" an application. They idle through an Infinite Loop waiting for an event to take place in the form of an interrupt, or a **pre-scheduled task**. In order to save power, some processors enter special **sleep** or **wait modes** instead of idling through an Infinite Loop, but they will come out of this mode upon either a timer or an External Interrupt.

Interrupts

Interrupts are mostly hardware mechanisms that instruct the program that an event has occurred. They may occur at any time, and are therefore asynchronous to the program flow. They require special handling by the processor, and are ultimately handled by a corresponding Interrupt Service Routine (ISR). Interrupts need to be handled quickly. If you take too much time servicing an interrupt, then you may miss another interrupt.

Little Endian Vs Big Endian

Although numbers are always displayed in the same way, they are not stored in the same way in memory. Big-Endian machines store the most significant byte of data in the lowest memory address. A Big-Endian machine stores 0x12345678 as −

Big-Endian	Little-Endian
ADDRESS+0: 0x12 ADDRESS+1: 0x34 ADDRESS +2: 0x56 ADDRESS +3: 0x78	ADDRESS+0: 0x78 ADDRESS +1: 0x56 ADDRESS +2: 0x34 ADDRESS +3: 0x12

Little-Endian machines, on the other hand, store the least significant byte of data in the lowest memory address. A Little-Endian machine stores 0x12345678 as −

Embedded Systems - Assembly Language

Assembly languages were developed to provide **mnemonics** or symbols for the machine level code instructions. Assembly language programs consist of mnemonics, thus they should be translated into machine

code. A program that is responsible for this conversion is known as **assembler**. Assembly language is often termed as a low-level language because it directly works with the internal structure of the CPU. To program in assembly language, a programmer must know all the registers of the CPU.

Different programming languages such as C, C++, Java and various other languages are called high-level languages because they do not deal with the internal details of a CPU. In contrast, an assembler is used to translate an assembly language program into machine code (sometimes also called **object code** or **opcode**). Similarly, a compiler translates a high-level language into machine code. For example, to write a program in C language, one must use a C compiler to translate the program into machine language.

Structure of Assembly Language:

An assembly language program is a series of statements, which are either assembly language instructions such as ADD and MOV, or statements called **directives (Like ORG, END,EQU)**.An **instruction** tells the CPU what to do, while a **directive** (also called **pseudo-instructions**) gives instruction to the assembler. For example, ADD and MOV instructions are commands which the CPU runs, while ORG and END are assembler directives. The assembler places the opcode to the memory location 0 when the ORG directive is used, while END indicates to the end of the source code. A program language instruction consists of the following four fields −

[label:] mnemonics [operands] [;comment]

A square bracket ([ ]) indicates that the field is optional.

- The **label field** allows the program to refer to a line of code by name. The label fields cannot exceed a certain number of characters.
- The **mnemonics** and **operands fields** together perform the real work of the program and accomplish the tasks. Statements like ADD A , C & MOV C, #68 where ADD and MOV are the mnemonics, which produce opcodes ; "A, C" and "C, #68" are operands. These two fields could contain directives. Directives do not generate machine code and are used only by the assembler, whereas instructions are translated into machine code for the CPU to execute.

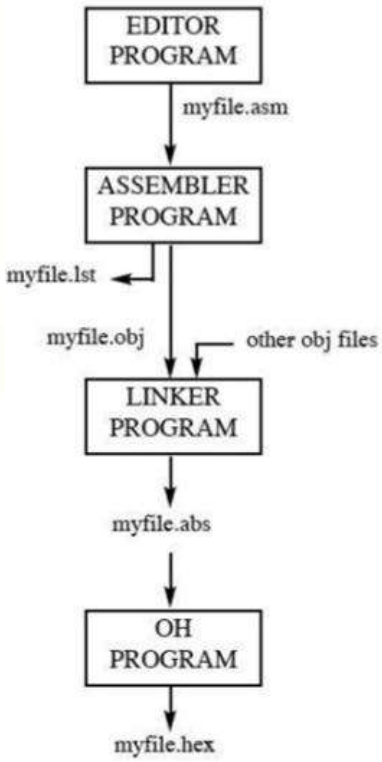
S.No.	ROM Location	OPCODE	ASSEMBLY CODE	Comments
1.	0000		ORG 0H	;start (origin) at location 0
2	0000	7D25	MOV R5,#25H	;load 25H into R5 – 2Byte ins.
3.	0002	7F34	MOV R7,#34H	;load 34H into R7 – 2Byte ins.
4.	0004	7400	MOV A,#0	;load 0 into A – 2Byte ins.
5.	0006	2D	ADD A,R5	;add contents of R5 to A – 1B ins.
6.	0007	2F	ADD A,R7	;add contents of R7 to A – 1B ins.
7.	0008	2412	ADD A,#12H	;add to A value 12 H – 2B ins.
8.	000A	80FE	HERE: SJMP HERE	;stay in this loop – 2B ins.
9.	000C		END	;end of asm source file

- The **comment field** begins with a semicolon which is a comment indicator.
- Notice the Label "HERE" in the program. Any label which refers to an instruction should be followed by a colon.

Assembling and Running an 8051 Program:

Here we will discuss about the basic form of an assembly language. The steps to create, assemble, and run an assembly language program are as follows −

- First, we use an editor to type in a program similar to the above program. Editors like MS-DOS EDIT program that comes with all Microsoft operating systems can be used to create or edit a program. The Editor must be able to produce an ASCII file. The "asm" extension for the source file is used by an assembler in the next step.
- The "asm" source file contains the program code created in Step 1. It is fed to an 8051 assembler. The assembler then converts the assembly language instructions into machine code instructions and produces an **.obj file** (object file) and a **.lst file** (list file). It is also called as a **source file**, that's why some assemblers require that this file have the "src" extensions. The "lst" file is optional. It is very useful to the program because it lists all the opcodes and addresses as well as errors that the assemblers detected.
- Assemblers require a third step called **linking**. The link program takes one or more object files and produces an absolute object file with the extension "abs".
- Next, the "abs" file is fed to a program called "OH" (object to hex converter), which creates a file with the extension "hex" that is ready to burn in to the ROM.



Data Type

The 8051 microcontroller contains a single data type of 8-bits, and each register is also of 8-bits size. The programmer has to break down data larger than 8-bits (00 to FFH, or to 255 in decimal) so that it can be processed by the CPU.

DB (Define Byte): The DB directive is the most widely used data directive in the assembler. It is used to define the 8-bit data. It can also be used to define decimal, binary, hex, or ASCII formats data. For decimal, the "D" after the decimal number is optional, but it is required for "B" (binary) and "Hl" (hexadecimal).

To indicate ASCII, simply place the characters in quotation marks ('like this'). The assembler generates ASCII code for the numbers/characters automatically. The DB directive is the only directive that can be used to define ASCII strings larger than two characters; therefore, it should be used for all the ASCII data definitions. Some examples of DB are given below −

```
        ORG  500H
DATA1:  DB   28              ;DECIMAL (1C in hex)
DATA2:  DB   00110101B          ;BINARY  (35 in hex)
DATA3:  DB   39H             ;HEX
        ORG  510H
DATA4:  DB   "2591"           ;ASCII  NUMBERS
        ORG  520H
DATA6:  DA   "MY NAME IS Michael"   ;ASCII CHARACTERS
```

Either single or double quotes can be used around ASCII strings. DB is also used to allocate memory in byte-sized chunks.

Assembler Directives

Some of the directives of 8051 are as follows −

- **ORG (origin)** − The origin directive is used to indicate the beginning of the address. It takes the numbers in hexa or decimal format. If H is provided after the number, the number is treated as hexa, otherwise decimal. The assembler converts the decimal number to hexa.
- **EQU (equate)** − It is used to define a constant without occupying a memory location. EQU associates a constant value with a data label so that the label appears in the program, its constant value will be substituted for the label. While executing the instruction "MOV R3, #COUNT", the register R3 will be loaded with the value 25 (notice the # sign). The advantage of using EQU is that the programmer can change it once and the assembler will change all of its occurrences; the programmer does not have to search the entire program.
- **END directive** − It indicates the end of the source (asm) file. The END directive is the last line of the program; anything after the END directive is ignored by the assembler.

Labels in Assembly Language

All the labels in assembly language must follow the rules given below −

- Each label name must be unique. The names used for labels in assembly language programming consist of alphabetic letters in both uppercase and lowercase, number 0 through 9, and special characters such as question mark (?), period (.), at the rate @, underscore (_), and dollar($).
- The first character should be in alphabetical character; it cannot be a number.
- Reserved words cannot be used as a label in the program. For example, ADD and MOV words are the reserved words, since they are instruction mnemonics.

Embedded Systems - Registers

Registers are used in the CPU to store information on temporarily basis which could be data to be processed, or an address pointing to the data which is to be fetched. In 8051, there is one data type is of 8-bits, from the MSB (most significant bit) D7 to the LSB (least significant bit) D0. With 8-bit data type, any data type larger than 8-bits must be broken into 8-bit chunks before it is processed.

The most widely used registers of the 8051 are A (accumulator), B, R0-R7, DPTR (data pointer), and PC (program counter). All these registers are of 8-bits, except DPTR and PC.

Storage Registers in 8051

We will discuss the following types of storage registers here −

- Accumulator
- B register
- R register (R0 – R7)
- Data Pointer (DPTR)
- Program Counter (PC)
- Stack Pointer (SP)

Accumulator

The accumulator, register A, is used for all arithmetic and logic operations. If the accumulator is not present, then every result of each calculation (addition, multiplication, shift, etc.) is to be stored into the main memory. Access to main memory is slower than access to a register like the accumulator because the technology used for the large main memory is slower (but cheaper) than that used for a register.

The "R" Registers

The "R" registers are a set of eight registers, namely, R0, R1 to R7. These registers function as auxiliary or temporary storage registers in many operations. Consider an example of the sum of 10 and 20. Store a variable 10 in an accumulator and another variable 20 in, say, register R4. To process the addition operation, execute the following command −

```
ADD A,R4
```

After executing this instruction, the accumulator will contain the value 30. Thus "R" registers are very important auxiliary or **helper registers**. The Accumulator alone would not be very useful if it were not for these "R" registers. The "R" registers are meant for temporarily storage of values.

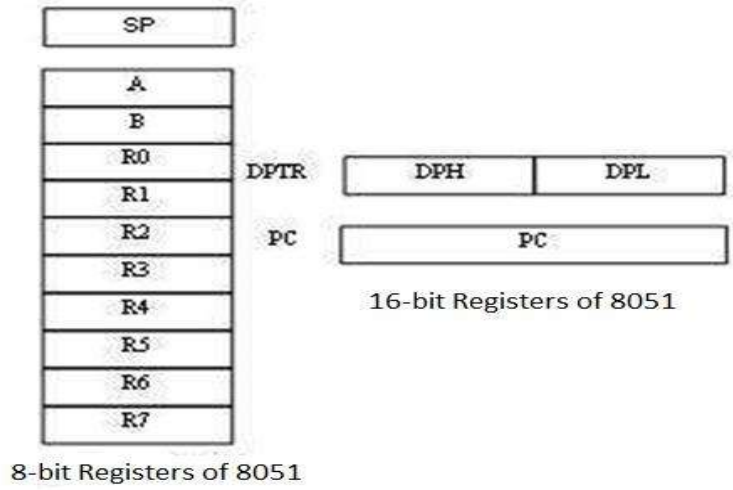
Let us take another example. We will add the values in R1 and R2 together and then subtract the values of R3 and R4 from the result.

```
MOV A,R3   ;Move the value of R3 into the accumulator
ADD A,R4   ;Add the value of R4
MOV R5,A   ;Store the resulting value temporarily in R5
MOV A,R1   ;Move the value of R1 into the accumulator
ADD A,R2   ;Add the value of R2
SUBB A,R5  ;Subtract the value of R5 (which now contains R3 + R4)
```

As you can see, we used R5 to temporarily hold the sum of R3 and R4. Of course, this is not the most efficient way to calculate (R1 + R2) – (R3 + R4), but it does illustrate the use of the "R" registers as a way to store values temporarily.

The "B" Register: The "B" register is very similar to the Accumulator in the sense that it may hold an 8-bit (1-byte) value. The "B" register is used only by two 8051 instructions: **MUL AB** and **DIV AB**. To quickly

and easily multiply or divide A by another number, you may store the other number in "B" and make use of these two instructions. Apart from using MUL and DIV instructions, the "B" register is often used as yet another temporary storage register, much like a ninth R register.



The Data Pointer

The Data Pointer (DPTR) is the 8051's only user-accessible 16-bit (2-byte) register. The Accumulator, R0–R7 registers and B register are 1-byte value registers. DPTR is meant for pointing to data. It is used by the 8051 to access external memory using the address indicated by DPTR. DPTR is the only 16-bit register available and is often used to store 2-byte values.

The Program Counter

The Program Counter (PC) is a 2-byte address which tells the 8051 where the next instruction to execute can be found in the memory. PC starts at 0000h when the 8051 initializes and is incremented every time after an instruction is executed. PC is not always incremented by 1. Some instructions may require 2 or 3 bytes; in such cases, the PC will be incremented by 2 or 3.

Branch, jump, and **interrupt** operations load the Program Counter with an address other than the next sequential location. Activating a power-on reset will cause all values in the register to be lost. It means the value of the PC is 0 upon reset, forcing the CPU to fetch the first opcode from the ROM location 0000. It means we must place the first byte of upcode in ROM location 0000 because that is where the CPU expects to find the first instruction.

The Stack Pointer (SP)

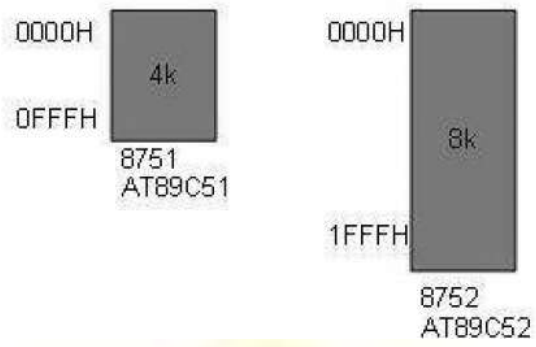
The Stack Pointer, like all registers except DPTR and PC, may hold an 8-bit (1-byte) value. The Stack Pointer tells the location from where the next value is to be removed from the stack. When a value is pushed onto the stack, the value of SP is incremented and then the value is stored at the resulting memory location. When a value is popped off the stack, the value is returned from the memory location indicated by SP, and then the value of SP is decremented.

This order of operation is important. SP will be initialized to 07h when the 8051 is initialized. If a value is pushed onto the stack at the same time, the value will be stored in the internal RAM address 08h because the 8051 will first increment the value of SP (from 07h to 08h) and then will store the pushed value at that memory

address (08h). SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI.

ROM Space in 8051

Some family members of 8051 have only 4K bytes of on-chip ROM (e.g. 8751, AT8951); some have 8K ROM like AT89C52, and there are some family members with 32K bytes and 64K bytes of on-chip ROM such as Dallas Semiconductor. The point to remember is that no member of the 8051 family can access more than 64K bytes of opcode since the program counter in 8051 is a 16-bit register (0000 to FFFF address).
The first location of the program ROM inside the 8051 has the address of 0000H, whereas the last location can be different depending on the size of the ROM on the chip. Among the 8051 family members, AT8951 has $k bytes of on-chip ROM having a memory address of 0000 (first location) to 0FFFH (last location).



8051 Flag Bits and PSW Register

The program status word (PSW) register is an 8-bit register, also known as **flag register**. It is of 8-bit wide but only 6-bit of it is used. The two unused bits are **user-defined flags**. Four of the flags are called **conditional flags**, which means that they indicate a condition which results after an instruction is executed. These four are **CY** (Carry), **AC** (auxiliary carry), **P** (parity), and **OV**(overflow). The bits RS0 and RS1 are used to change the bank registers. The following figure shows the program status word register.
The PSW Register contains that status bits that reflect the current status of the CPU.

CY	CA	F0	RS1	RS0	OV	-	P

CY	PSW.7	Carry Flag
AC	PSW.6	Auxiliary Carry Flag
F0	PSW.5	Flag 0 available to user for general purpose.
RS1	PSW.4	Register Bank selector bit 1
RS0	PSW.3	Register Bank selector bit 0
OV	PSW.2	Overflow Flag
-	PSW.1	User definable FLAG
P	PSW.0	Parity FLAG. Set/ cleared by hardware during instruction cycle to indicate even/odd number of 1 bit in accumulator.

We can select the corresponding Register Bank bit using RS0 and RS1 bits.

RS1	RS2	Register Bank	Address
0	0	0	00H-07H
0	1	1	08H-0FH
1	0	2	10H-17H
1	1	3	18H-1FH

- **CY, the carry flag** − This carry flag is set (1) whenever there is a carry out from the D7 bit. It is affected after an 8-bit addition or subtraction operation. It can also be reset to 1 or 0 directly by an instruction such as "SETB C" and "CLR C" where "SETB" stands for set bit carry and "CLR" stands for clear carry.
- **AC, auxiliary carry flag** − If there is a carry from D3 and D4 during an ADD or SUB operation, the AC bit is set; otherwise, it is cleared. It is used for the instruction to perform binary coded decimal arithmetic.
- **P, the parity flag** − The parity flag represents the number of 1's in the accumulator register only. If the A register contains odd number of 1's, then P = 1; and for even number of 1's, P = 0.
- **OV, the overflow flag** − This flag is set whenever the result of a signed number operation is too large causing the high-order bit to overflow into the sign bit. It is used only to detect errors in signed arithmetic operations.

Example

Show the status of CY, AC, and P flags after the addition of 9CH and 64H in the following instruction.

```
MOV A, #9CH
ADD A, # 64H
```

```
Solution:   9C   10011100
           +64   01100100
           100   00000000

CY = 1 since there is a carry beyond D7 bit
AC = 0 since there is a carry from D3 to D4
P  = 0 because the accumulator has even number of 1's
```

Embedded Systems - Registers Bank/Stack

The 8051 microcontroller has a total of 128 bytes of RAM. We will discuss about the allocation of these 128 bytes of RAM and examine their usage as stack and register.

RAM Memory Space Allocation in 8051: The 128 bytes of RAM inside the 8051 are assigned the address 00 to 7FH. They can be accessed directly as memory locations and are divided into three different groups as follows −

- 32 bytes from 00H to 1FH locations are set aside for register banks and the stack.
- 16 bytes from 20H to 2FH locations are set aside for bit-addressable read/write memory.

- 80 bytes from 30H to 7FH locations are used for read and write storage; it is called as **scratch pad**. These 80 locations RAM are widely used for the purpose of storing data and parameters by 8051 programmers.

Register Banks in 8051

A total of 32 bytes of RAM are set aside for the register banks and the stack. These 32 bytes are divided into four register banks in which each bank has 8 registers, R0–R7. RAM locations from 0 to 7 are set aside for bank 0 of R0–R7 where R0 is RAM location 0, R1 is RAM location 1, R2 is location 2, and so on, until the memory location 7, which belongs to R7 of bank 0.

The second bank of registers R0–R7 starts at RAM location 08 and goes to locations OFH. The third bank of R0–R7 starts at memory location 10H and goes to location to 17H. Finally, RAM locations 18H to 1FH are set aside for the fourth bank of R0–R7.

Default Register Bank

If RAM locations 00–1F are set aside for the four registers banks, which register bank of R0–R7 do we have access to when the 8051 is powered up? The answer is register bank 0; that is, RAM locations from 0 to 7 are accessed with the names R0 to R7 when programming the 8051. Because it is much easier to refer these RAM locations by names such as R0 to R7, rather than by their memory locations.

How to Switch Register Banks

Register bank 0 is the default when the 8051 is powered up. We can switch to the other banks using PSW register. D4 and D3 bits of the PSW are used to select the desired register bank, since they can be accessed by the bit addressable instructions SETB and CLR. For example, "SETB PSW.3" will set PSW.3 = 1 and select the bank register 1.

Stack and its Operations

Stack in the 8051: The stack is a section of a RAM used by the CPU to store information such as data or memory address on temporary basis. The CPU needs this storage area considering limited number of registers.

How Stacks are Accessed: As the stack is a section of a RAM, there are registers inside the CPU to point to it. The register used to access the stack is known as the stack pointer register. The stack pointer in the 8051 is 8-bits wide, and it can take a value of 00 to FFH. When the 8051 is initialized, the SP register contains the value 07H. This means that the RAM location 08 is the first location used for the stack. The storing operation of a CPU register in the stack is known as a **PUSH**, and getting the contents from the stack back into a CPU register is called a **POP**.

Pushing into the Stack

In the 8051, the stack pointer (SP) points to the last used location of the stack. When data is pushed onto the stack, the stack pointer (SP) is incremented by 1. When PUSH is executed, the contents of the register are saved on the stack and SP is incremented by 1. To push the registers onto the stack, we must use their RAM addresses. For example, the instruction "PUSH 1" pushes register R1 onto the stack.

Popping from the Stack

Popping the contents of the stack back into a given register is the opposite to the process of pushing. With every pop operation, the top byte of the stack is copied to the register specified by the instruction and the stack pointer is decremented once.

Embedded Systems - Instructions

The flow of program proceeds in a sequential manner, from one instruction to the next instruction, unless a control transfer instruction is executed. The various types of control transfer instruction in assembly language include conditional or unconditional jumps and call instructions.

Loop and Jump Instructions

Looping in the 8051

Repeating a sequence of instructions a certain number of times is called a **loop**. An instruction **DJNZ reg, label** is used to perform a Loop operation. In this instruction, a register is decremented by 1; if it is not zero, then 8051 jumps to the target address referred to by the label.

The register is loaded with the counter for the number of repetitions prior to the start of the loop. In this instruction, both the registers decrement and the decision to jump are combined into a single instruction. The registers can be any of R0–R7. The counter can also be a RAM location.

Example

Multiply 25 by 10 using the technique of repeated addition.

Solution − Multiplication can be achieved by adding the multiplicand repeatedly, as many times as the multiplier. For example,

25 * 10 = 250(FAH)

25 + 25 + 25 + 25 + 25 + 25 + 25 + 25 + 25 + 25 = 250

```
MOV A,#0                    ;A = 0,clean ACC
MOV R2,#10                  ; the multiplier is replaced in R2
```

```
  ADD A,#25                ;add the multiplicand to the ACC
AGAIN:DJNZ R2, AGAIN       :repeat  until R2 = 0 (10 times)
  MOV R5 , A               ;save A in R5 ;R5 (FAH)
```

Drawback in 8051 − Looping action with the instruction **DJNZ Reg label** is limited to 256 iterations only. If a conditional jump is not taken, then the instruction following the jump is executed.

Looping inside a Loop

When we use a loop inside another loop, it is called a **nested loop**. Two registers are used to hold the count when the maximum count is limited to 256. So we use this method to repeat the action more times than 256.

Example

Write a program to −

- Load the accumulator with the value 55H.
- Complement the ACC 700 times.

Solution − Since 700 is greater than 255 (the maximum capacity of any register), two registers are used to hold the count. The following code shows how to use two registers, R2 and R3, for the count.

```
        MOV A,#55H        ;A = 55H
        MOV R3,#10      ;R3 the outer loop counter
NEXT:   MOV R2,#70      ;R2 the inner loop counter
AGAIN:  CPL A           ;complement
        DJNZ R2,AGAIN
        DJNZ R3, NEXT
```

Other Conditional Jumps

The following table lists the conditional jumps used in 8051 −

Instruction	Action
JZ	Jump if A = 0
JNZ	Jump if A ≠ 0
DJNZ	Decrement and Jump if register ≠ 0
CJNE A, data	Jump if A ≠ data
CJNE reg, #data	Jump if byte ≠ data
JC	Jump if CY = 1
JNC	Jump if CY ≠ 1
JB	Jump if bit = 1
JNB	Jump if bit = 0
JBC	Jump if bit = 1 and clear bit

- **JZ (jump if A = 0)** − In this instruction, the content of the accumulator is checked. If it is zero, then the 8051 jumps to the target address. JZ instruction can be used only for the accumulator, it does not apply to any other register.
- **JNZ (jump if A is not equal to 0)** − In this instruction, the content of the accumulator is checked to be non-zero. If it is not zero, then the 8051 jumps to the target address.
- **JNC (Jump if no carry, jumps if CY = 0)** − The Carry flag bit in the flag (or PSW) register is used to make the decision whether to jump or not "JNC label". The CPU looks at the carry flag to see if it is raised (CY = 1). If it is not raised, then the CPU starts to fetch and execute instructions from the address of the label. If CY = 1, it will not jump but will execute the next instruction below JNC.
- **JC (Jump if carry, jumps if CY = 1)** − If CY = 1, it jumps to the target address.
- **JB (jump if bit is high)**
- **JNB (jump if bit is low)**

Note − It must be noted that all conditional jumps are short jumps, i.e., the address of the target must be within –128 to +127 bytes of the contents of the program counter.

Unconditional Jump Instructions

There are two unconditional jumps in 8051 −

- **LJMP (long jump)** − LJMP is 3-byte instruction in which the first byte represents opcode, and the second and third bytes represent the 16-bit address of the target location. The 2-byte target address is to allow a jump to any memory location from 0000 to FFFFH.
- **SJMP (short jump)** − It is a 2-byte instruction where the first byte is the opcode and the second byte is the relative address of the target location. The relative address ranges from 00H to FFH which is divided into forward and backward jumps; that is, within –128 to +127 bytes of memory relative to the address of the current PC (program counter). In case of forward jump, the target address can be within a space of 127 bytes from the current PC. In case of backward jump, the target address can be within –128 bytes from the current PC.

Calculating the Short Jump Address

All conditional jumps (JNC, JZ, and DJNZ) are short jumps because they are 2-byte instructions. In these instructions, the first byte represents opcode and the second byte represents the relative address. The target address is always relative to the value of the program counter. To calculate the target address, the second byte is added to the PC of the instruction immediately below the jump. Take a look at the program given below −

```
Line    PC      Op-code         Mnemonic   Operand
1       0000                    ORG        0000
2       0000    7800            MOV        R0,#003
3       0002    7455            MOV        A,#55H0
4       0004    6003            JZ         NEXT
5       0006    08              INC        R0
6       0007    04              AGAIN:  INC        A
7       0008    04              INC        A
8       0009    2477    NEXT:   ADD        A, #77h
9       000B    5005            JNC        OVER
10      000D    E4              CLR        A
11      000E    F8              MOV        R0, A
```

```
12      000F    F9          MOV     R1, A
13      0010    FA          MOV     R2, A
14      0011    FB          MOV     R3, A
15      0012    2B    OVER: ADD     A, R3
16      0013    50F2        JNC     AGAIN
17      0015    80FE  HERE: SJMP    HERE
18      0017                END
```

Backward Jump Target Address Calculation

In case of a forward jump, the displacement value is a positive number between 0 to 127 (00 to 7F in hex). However, for a backward jump, the displacement is a negative value of 0 to –128.

CALL Instructions

CALL is used to call a subroutine or method. Subroutines are used to perform operations or tasks that need to be performed frequently. This makes a program more structured and saves memory space. There are two instructions − LCALL and ACALL.

LCALL (Long Call)

LCALL is a 3-byte instruction where the first byte represents the opcode and the second and third bytes are used to provide the address of the target subroutine. LCALL can be used to call subroutines which are available within the 64K-byte address space of the 8051.

To make a successful return to the point after execution of the called subroutine, the CPU saves the address of the instruction immediately below the LCALL on the stack. Thus, when a subroutine is called, the control is transferred to that subroutine, and the processor saves the PC (program counter) on the stack and begins to fetch instructions from the new location. The instruction RET (return) transfers the control back to the caller after finishing execution of the subroutine. Every subroutine uses RET as the last instruction.

ACALL (Absolute Call)

ACALL is a 2-byte instruction, in contrast to LCALL which is 3 bytes. The target address of the subroutine must be within 2K bytes because only 11 bits of the 2 bytes are used for address. The difference between the ACALL and LCALL is that the target address for LCALL can be anywhere within the 64K-bytes address space of the 8051, while the target address of CALL is within a 2K-byte range.

Embedded Systems - Addressing Modes****

An **addressing mode** refers to how you are addressing a given memory location. There are five different ways or five addressing modes to execute this instruction which are as follows −

- Immediate addressing mode
- Direct addressing mode
- Register direct addressing mode
- Register indirect addressing mode
- Indexed addressing mode

Immediate Addressing Mode

Let's begin with an example.

```
MOV A, #6AH
```

In general, we can write,

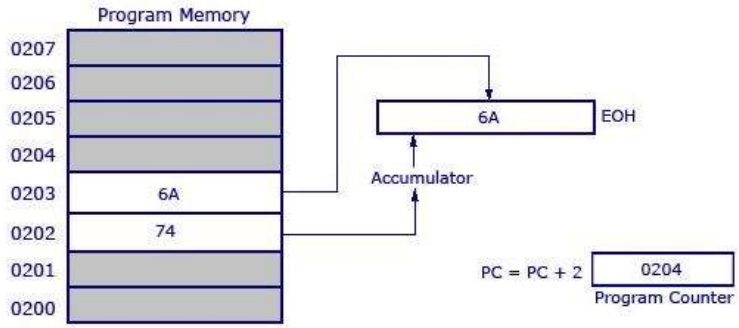
MOV A, #data

It is termed as **immediate** because 8-bit data is transferred immediately to the accumulator (destination operand).

The following illustration describes the above instruction and its execution. The opcode 74H is saved at 0202 address. The data 6AH is saved at 0203 address in the program memory. After reading the opcode 74H, the data at the next program memory address is transferred to accumulator A (E0H is the address of accumulator). Since the instruction is of 2-bytes and is executed in one cycle, the program counter will be incremented by 2 and will point to 0204 of the program memory.

Immediate Addressing Mode

Instruction	Opcode	Bytes	Cycles
MOV A, #6AH	74H	2	1



Note − The '#' symbol before 6AH indicates that the operand is a data (8 bit). In the absence of '#', the hexadecimal number would be taken as an address.

Direct Addressing Mode

This is another way of addressing an operand. Here, the address of the data (source data) is given as an operand. Let's take an example.

MOV A, 04H

The register bank#0 (4th register) has the address 04H. When the MOV instruction is executed, the data stored in register 04H is moved to the accumulator. As the register 04H holds the data 1FH, 1FH is moved to the accumulator.

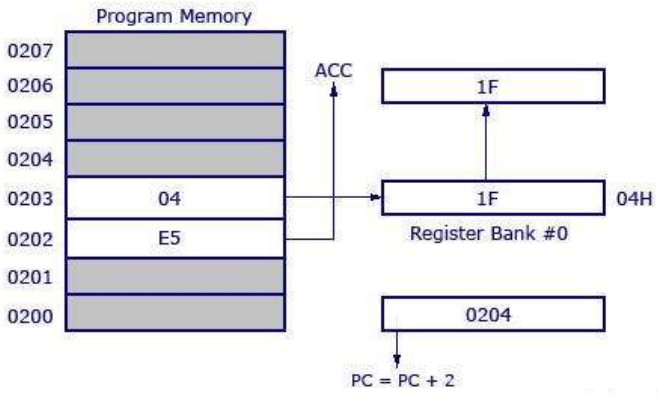
Note − We have not used '#' in direct addressing mode, unlike immediate mode. If we had used '#', the data value 04H would have been transferred to the accumulator instead of 1FH.

Now, take a look at the following illustration. It shows how the instruction gets executed.

As shown in the below illustration, this is a 2-byte instruction which requires 1 cycle to complete. The PC will be incremented by 2 and will point to 0204. The opcode for the instruction MOV A, address is E5H. When the instruction at 0202 is executed (E5H), the accumulator is made active and ready to receive data. Then the PC goes to the next address as 0203 and looks up the address of the location of 04H where the source data (to be transferred to accumulator) is located. At 04H, the control finds the data 1F and transfers it to the accumulator and hence the execution is completed.

Direct Addressing Mode

Instruction	Opcode	Bytes	Cycles
MOV A, #04H	E5	2	1



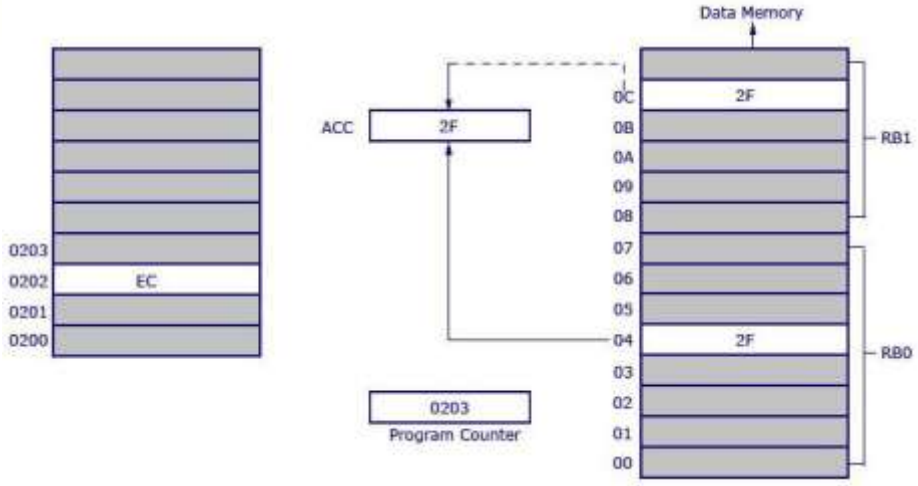
Register Direct Addressing Mode

In this addressing mode, we use the register name directly (as source operand). Let us try to understand with the help of an example.

MOV A, R4

Register Direct Addressing Mode

Instruction	Opcode	Bytes	Cycles
MOV A, R4	ECH	1	1



At a time, the registers can take values from R0 to R7. There are 32 such registers. In order to use 32 registers with just 8 variables to address registers, register banks are used. There are 4 register banks named from 0 to 3. Each bank comprises of 8 registers named from R0 to R7.

At a time, a single register bank can be selected. Selection of a register bank is made possible through a **Special Function Register** (SFR) named **Processor Status Word** (PSW). PSW is an 8-bit SFR where each bit can be

programmed as required. Bits are designated from PSW.0 to PSW.7. PSW.3 and PSW.4 are used to select register banks.

Now, take a look at the above illustration to get a clear understanding of how it works.

Opcode EC is used for MOV A, R4. The opcode is stored at the address 0202 and when it is executed, the control goes directly to R4 of the respected register bank (that is selected in PSW). If register bank #0 is selected, then the data from R4 of register bank #0 will be moved to the accumulator. Here 2F is stored at 04H. 04H represents the address of R4 of register bank #0.

Data (2F) movement is highlighted in bold. 2F is getting transferred to the accumulator from data memory location 0C H and is shown as dotted line. 0CH is the address location of Register 4 (R4) of register bank #1.

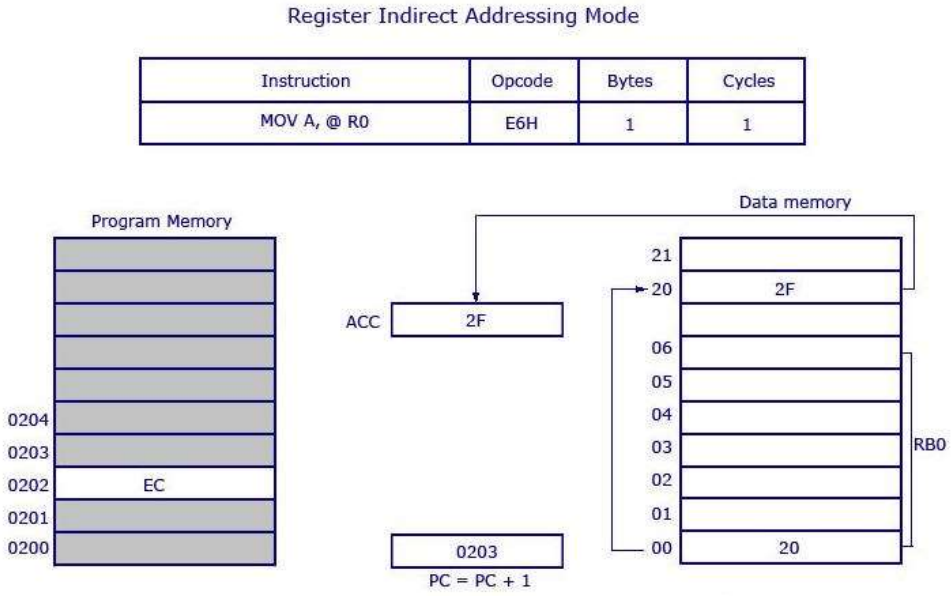
The instruction above is 1 byte and requires 1 cycle for complete execution. What it means is, you can save program memory by using register direct addressing mode.

Register Indirect Addressing Mode

In this addressing mode, the address of the data is stored in the register as operand.

MOV A, @R0

Here the value inside R0 is considered as an address, which holds the data to be transferred to the accumulator. **Example**: If R0 has the value 20H, and data 2FH is stored at the address 20H, then the value 2FH will get transferred to the accumulator after executing this instruction. See the following illustration.



So the opcode for **MOV A, @R0** is E6H. Assuming that the register bank #0 is selected, the R0 of register bank #0 holds the data 20H. Program control moves to 20H where it locates the data 2FH and it transfers 2FH to the accumulator. This is a 1-byte instruction and the program counter increments by 1 and moves to 0203 of the program memory.

Note − Only R0 and R1 are allowed to form a register indirect addressing instruction. In other words, the programmer can create an instruction either using @R0 or @R1. All register banks are allowed.

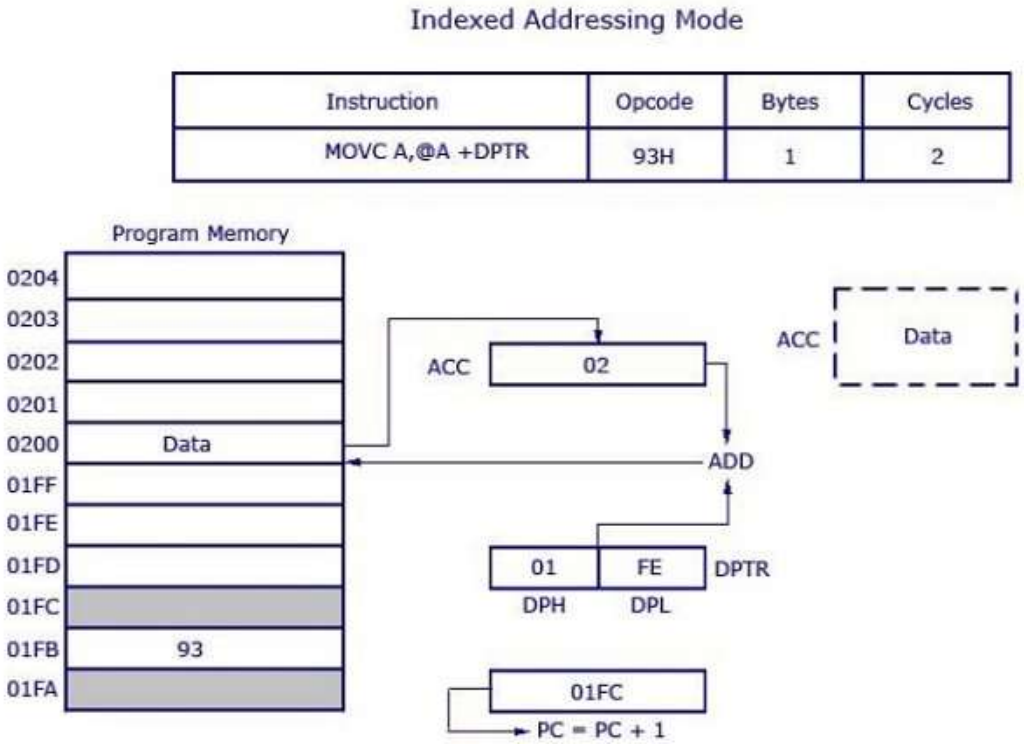
Indexed Addressing Mode: We will take two examples to understand the concept of indexed addressing mode. Take a look at the following instructions −

MOVC A, @A+DPTR and **MOVC A, @A+PC**

where DPTR is the data pointer and PC is the program counter (both are 16-bit registers). Consider the first example.

MOVC A, @A+DPTR

The source operand is @A+DPTR. It contains the source data from this location. Here we are adding the contents of DPTR with the current content of the accumulator. This addition will give a new address which is the address of the source data. The data pointed by this address is then transferred to the accumulator.



The opcode is 93H. DPTR has the value 01FE, where 01 is located in DPH (higher 8 bits) and FE is located in DPL (lower 8 bits). Accumulator has the value 02H. Then a 16-bit addition is performed and 01FE H+02H results in 0200 H. Data at the location 0200H will get transferred to the accumulator. The previous value inside the accumulator (02H) will be replaced with the new data from 0200H. The new data in the accumulator is highlighted in the illustration.

This is a 1-byte instruction with 2 cycles needed for execution and the execution time required for this instruction is high compared to previous instructions (which were all 1 cycle each).

The other example **MOVC A, @A+PC** works the same way as the above example. Instead of adding DPTR with the accumulator, here the data inside the program counter (PC) is added with the accumulator to obtain the target address.

Embedded Systems - SFR Registers

A Special Function Register (or Special Purpose Register, or simply Special Register) is a register within a microprocessor that controls or monitors the various functions of a microprocessor. As the special registers are closely tied to some special function or status of the processor, they might not be directly writable by normal instructions (like add, move, etc.). Instead, some special registers in some processor architectures require special instructions to modify them.

In the 8051, register A, B, DPTR, and PSW are a part of the group of registers commonly referred to as SFR (special function registers). An SFR can be accessed by its name or by its address.
The following table shows a list of SFRs and their addresses.

Byte Address	Bit Address								SFR
FF									
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
D0	D7	D6	D5	D4	D3	D2	-	D0	PSW
B8	-	-	-	BC	BB	BA	B9	B8	IP
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
A2	AF	-	-	AC	AB	AA	A9	A8	IE
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
99	Not bit Addressable								SBUF
98	9F	9E	9D	9C	9B	9A	99	98	SCON
90	97	96	95	94	93	92	91	90	P1
8D	Not bit Addressable								TH1
8C	Not bit Addressable								TH0
8B	Not bit Addressable								TL1
8A	Not bit Addressable								TL0
89	Not bit Addressable								TMOD

88	8F	8E	8D	8C	8B	8A	89	88	TCON
87	Not bit Addressable								PCON
83	Not bit Addressable								DPH
82	Not bit Addressable								DPL
81	Not bit Addressable								SP
80	87	87	85	84	83	82	81	80	P0

Consider the following two points about the SFR addresses.

- A special function register can have an address between 80H to FFH. These addresses are above 80H, as the addresses from 00 to 7FH are the addresses of RAM memory inside the 8051.
- Not all the address space of 80 to FF are used by the SFR. Unused locations, 80H to FFH, are reserved and must not be used by the 8051 programmer.

The Program Status Word (PSW) contains status bits to reflect the current state of the CPU. The 8051 variants provide one special function register, PSW, with the status information. The 8251 provides two additional status flags, Z and N, which are available in a second special function register called PSW1.

Embedded Systems - Timer/Counter

A **timer** is a specialized type of clock which is used to measure time intervals. A timer that counts from zero upwards for measuring time elapsed is often called a **stopwatch**. It is a device that counts down from a specified time interval and used to generate a time delay, for example, an hourglass is a timer.

A **counter** is a device that stores (and sometimes displays) the number of times a particular event or process occurred, with respect to a clock signal. It is used to count the events happening outside the microcontroller. In electronics, counters can be implemented quite easily using register-type circuits such as a flip-flop.

Difference between a Timer and a Counter

The points that differentiate a timer from a counter are as follows −

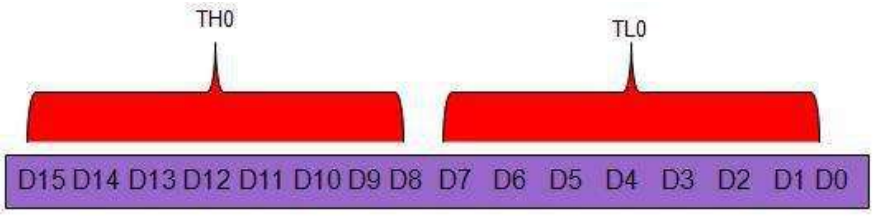
Timer	Counter
The register incremented for every machine cycle.	The register is incremented considering 1 to 0 transition at its corresponding to an external input pin (T0, T1).
Maximum count rate is 1/12 of the oscillator frequency.	Maximum count rate is 1/24 of the oscillator frequency.
A timer uses the frequency of the internal clock, and generates delay.	A counter uses an external signal to count pulses.

Timers of 8051 and their Associated Registers:

The 8051 has two timers, Timer 0 and Timer 1. They can be used as timers or as event counters. Both Timer 0 and Timer 1 are 16-bit wide. Since the 8051 follows an 8-bit architecture, each 16 bit is accessed as two separate registers of low-byte and high-byte.

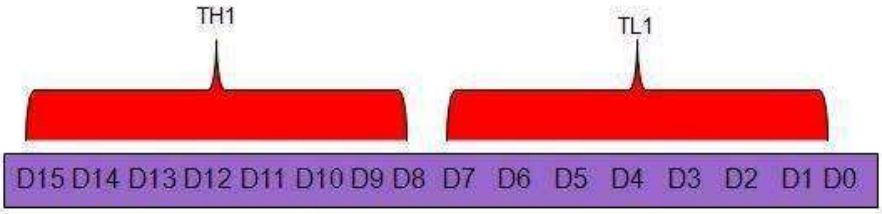
Timer 0 Register

The 16-bit register of Timer 0 is accessed as low- and high-byte. The low-byte register is called TL0 (Timer 0 low byte) and the high-byte register is called TH0 (Timer 0 high byte). These registers can be accessed like any other register. For example, the instruction **MOV TL0, #4H** moves the value into the low-byte of Timer #0.



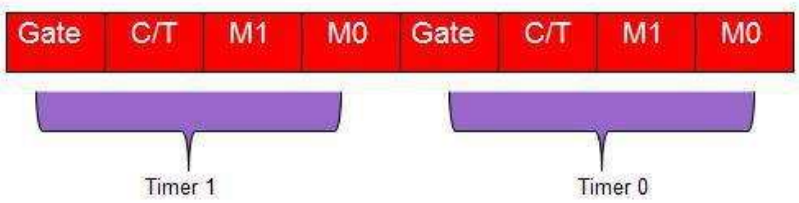
Timer 1 Register

The 16-bit register of Timer 1 is accessed as low- and high-byte. The low-byte register is called TL1 (Timer 1 low byte) and the high-byte register is called TH1 (Timer 1 high byte). These registers can be accessed like any other register. For example, the instruction **MOV TL1, #4H** moves the value into the low-byte of Timer 1.



TMOD (Timer Mode) Register

Both Timer 0 and Timer 1 use the same register to set the various timer operation modes. It is an 8-bit register in which the lower 4 bits are set aside for Timer 0 and the upper four bits for Timers. In each case, the lower 2 bits are used to set the timer mode in advance and the upper 2 bits are used to specify the location.



Gate − When set, the timer only runs while INT(0,1) is high.

C/T − Counter/Timer select bit.

M1 − Mode bit 1.

M0 − Mode bit 0.

GATE: Every timer has a means of starting and stopping. Some timers do this by software, some by hardware, and some have both software and hardware controls. 8051 timers have both software and hardware controls. The start and stop of a timer is controlled by software using the instruction **SETB TR1** and **CLR TR1** for timer 1, and **SETB TR0** and **CLR TR0** for timer 0.

The SETB instruction is used to start it and it is stopped by the CLR instruction. These instructions start and stop the timers as long as GATE = 0 in the TMOD register. Timers can be started and stopped by an external source by making GATE = 1 in the TMOD register.

C/T (CLOCK / TIMER)

This bit in the TMOD register is used to decide whether a timer is used as a **delay generator** or an **event manager**. If C/T = 0, it is used as a timer for timer delay generation. The clock source to create the time delay is the crystal frequency of the 8051. If C/T = 0, the crystal frequency attached to the 8051 also decides the speed at which the 8051 timer ticks at a regular interval.

Timer frequency is always 1/12th of the frequency of the crystal attached to the 8051. Although various 8051 based systems have an XTAL frequency of 10 MHz to 40 MHz, we normally work with the XTAL frequency of 11.0592 MHz. It is because the baud rate for serial communication of the 8051.XTAL = 11.0592 allows the 8051 system to communicate with the PC with no errors.

M1 / M2

M1	M2	Mode
0	0	13-bit timer mode.
0	1	16-bit timer mode.
1	0	8-bit auto reload mode.
1	1	Spilt mode.

Different Modes of Timers

Mode 0 (13-Bit Timer Mode): Both Timer 1 and Timer 0 in Mode 0 operate as 8-bit counters (with a divide-by-32 prescaler). Timer register is configured as a 13-bit register consisting of all the 8 bits of TH1 and the lower 5 bits of TL1. The upper 3 bits of TL1 are indeterminate and should be ignored. Setting the run flag (TR1) does not clear the register. The timer interrupt flag TF1 is set when the count rolls over from all 1s to all 0s. Mode 0 operation is the same for Timer 0 as it is for Timer 1.

Mode 1 (16-Bit Timer Mode): Timer mode "1" is a 16-bit timer and is a commonly used mode. It functions in the same way as 13-bit mode except that all 16 bits are used. TLx is incremented starting from 0 to a maximum 255. Once the value 255 is reached, TLx resets to 0 and then THx is incremented by 1. As being a full 16-bit timer, the timer may contain up to 65536 distinct values and it will overflow back to 0 after 65,536 machine cycles.

Mode 2 (8 Bit Auto Reload): Both the timer registers are configured as 8-bit counters (TL1 and TL0) with automatic reload. Overflow from TL1 (TL0) sets TF1 (TF0) and also reloads TL1 (TL0) with the contents of Th1 (TH0), which is preset by software. The reload leaves TH1 (TH0) unchanged. The benefit of auto-reload mode is that you can have the timer to always contain a value from 200 to 255. If you use mode 0 or 1, you would have to check in the code to see the overflow and, in that case, reset the timer to 200. In this case, precious instructions check the value and/or get reloaded. In mode 2, the microcontroller takes care of this. Once you have configured a timer in mode 2, you don't have to worry about checking to see if the timer has overflowed, nor do you have to worry about resetting the value because the microcontroller hardware will do it all for you. The auto-reload mode is used for establishing a common baud rate.

Mode 3 (Split Timer Mode): Timer mode "3" is known as **split-timer mode**. When Timer 0 is placed in mode 3, it becomes two separate 8-bit timers. Timer 0 is TL0 and Timer 1 is TH0. Both the timers count from 0 to 255 and in case of overflow, reset back to 0. All the bits that are of Timer 1 will now be tied to TH0. When Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be set in modes 0, 1 or 2, but it cannot be started/stopped as the bits that do that are now linked to TH0. The real timer 1 will be incremented with every machine cycle.

Initializing a Timer: Decide the timer mode. Consider a 16-bit timer that runs continuously, and is independent of any external pins. Initialize the TMOD SFR. Use the lowest 4 bits of TMOD and consider Timer 0. Keep the two bits, GATE 0 and C/T 0, as 0, since we want the timer to be independent of the external pins. As 16-bit mode is timer mode 1, clear T0M1 and set T0M0. Effectively, the only bit to turn on is bit 0 of TMOD. Now execute the following instruction −

```
MOV TMOD,#01h
```

Now, Timer 0 is in 16-bit timer mode, but the timer is not running. To start the timer in running mode, set the TR0 bit by executing the following instruction −

```
SETB TR0
```

Now, Timer 0 will immediately start counting, being incremented once every machine cycle.

Reading a Timer: A 16-bit timer can be read in two ways. Either read the actual value of the timer as a 16-bit number, or you detect when the timer has overflowed.

Detecting Timer Overflow: When a timer overflows from its highest value to 0, the microcontroller automatically sets the TFx bit in the TCON register. So instead of checking the exact value of the timer, the TFx bit can be checked. If TF0 is set, then Timer 0 has overflowed; if TF1 is set, then Timer 1 has overflowed.

Embedded Systems – Interrupt*****

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an **Interrupt Service Routine** (ISR) or **Interrupt Handler**. ISR tells the processor or controller what to do when the interrupt occurs. The interrupts can be either hardware interrupts or software interrupts.

Hardware Interrupt: A hardware interrupt is an electronic alerting signal sent to the processor from an external device, like a disk controller or an external peripheral. For example, when we press a key on the keyboard or move the mouse, they trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

Software Interrupt: A software interrupt is caused either by an exceptional condition or a special instruction in the instruction set which causes an interrupt when it is executed by the processor. For example, if the processor's arithmetic logic unit runs a command to divide a number by zero, to cause a divide-by-zero exception, thus causing the computer to abandon the calculation or display an error message. Software interrupt instructions work similar to subroutine calls.

What is Polling?

The state of continuous monitoring is known as **polling**. The microcontroller keeps checking the status of other devices; and while doing so, it does no other operation and consumes all its processing time for monitoring. This problem can be addressed by using interrupts.

In the interrupt method, the controller responds only when an interruption occurs. Thus, the controller is not required to regularly monitor the status (flags, signals etc.) of interfaced and inbuilt devices.

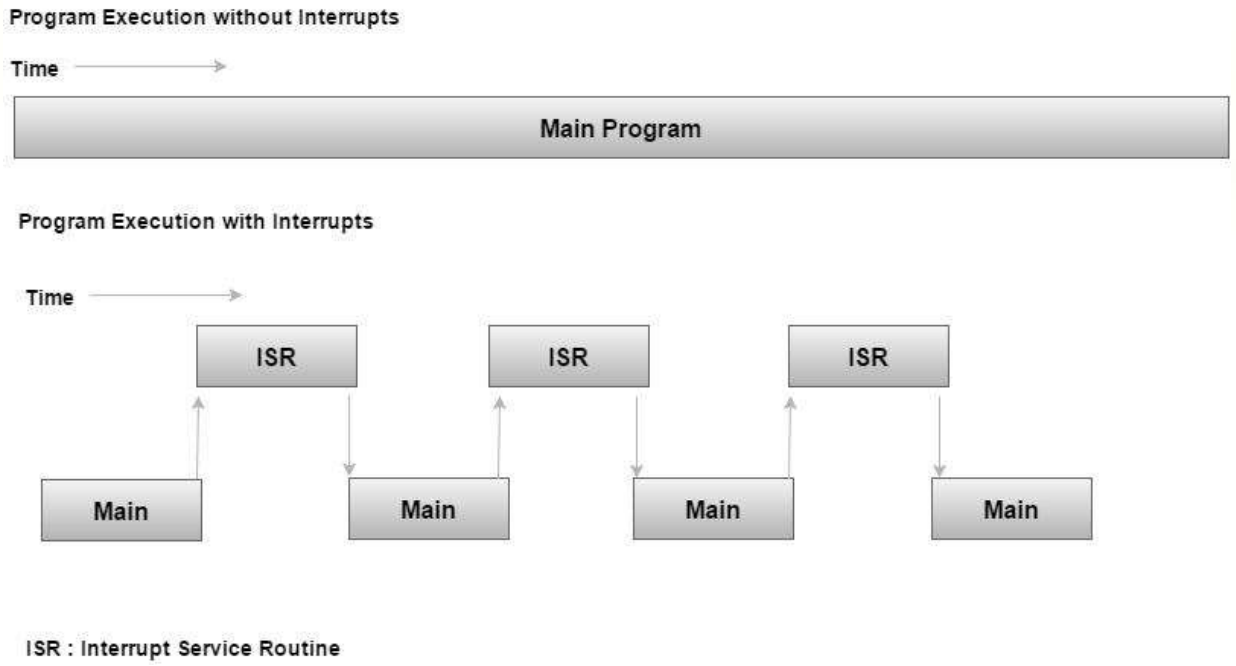
Interrupts v/s Polling

Here is an analogy that differentiates an interrupt from polling −

Interrupt	Polling
An interrupt is like a **shopkeeper**. If one needs a service or product, he goes to him and apprises him of his needs. In case of interrupts, when the flags or signals are received, they notify the controller that they need to be serviced.	The polling method is like a **salesperson**. The salesman goes from door to door while requesting to buy a product or service. Similarly, the controller keeps monitoring the flags or signals one by one for all devices and provides service to whichever component that needs its service.

Interrupt Service Routine

For every interrupt, there must be an interrupt service routine (ISR), or **interrupt handler**. When an interrupt occurs, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its interrupt service routine, ISR. The table of memory locations set aside to hold the addresses of ISRs is called as the Interrupt Vector Table.



Interrupt Vector Table

There are six interrupts including RESET in 8051.

Interrupts	ROM Location (Hex)	Pin
Interrupts	ROM Location (HEX)	

Serial COM (RI and TI)	0023	
Timer 1 interrupts(TF1)	001B	
External HW interrupt 1 (INT1)	0013	P3.3 (13)
External HW interrupt 0 (INT0)	0003	P3.2 (12)
Timer 0 (TF0)	000B	
Reset	0000	9

- When the reset pin is activated, the 8051 jumps to the address location 0000. This is power-up reset.
- Two interrupts are set aside for the timers: one for timer 0 and one for timer 1. Memory locations are 000BH and 001BH respectively in the interrupt vector table.
- Two interrupts are set aside for hardware external interrupts. Pin no. 12 and Pin no. 13 in Port 3 are for the external hardware interrupts INT0 and INT1, respectively. Memory locations are 0003H and 0013H respectively in the interrupt vector table.
- Serial communication has a single interrupt that belongs to both receive and transmit. Memory location 0023H belongs to this interrupt.

Steps to Execute an Interrupt

When an interrupt gets active, the microcontroller goes through the following steps −

- The microcontroller closes the currently executing instruction and saves the address of the next instruction (PC) on the stack.
- It also saves the current status of all the interrupts internally (i.e., not on the stack).
- It jumps to the memory location of the interrupt vector table that holds the address of the interrupts service routine.
- The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine, which is RETI (return from interrupt).
- Upon executing the RETI instruction, the microcontroller returns to the location where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then, it start to execute from that address.

Edge Triggering vs. Level Triggering

Interrupt modules are of two types − level-triggered or edge-triggered.

Level Triggered	Edge Triggered
A level-triggered interrupt module always generates an interrupt whenever the level of the interrupt source is asserted.	An edge-triggered interrupt module generates an interrupt only when it detects an asserting edge of the interrupt source. The edge gets detected when the interrupt source level actually changes. It can also be detected by periodic sampling and detecting an asserted level when the previous sample was de-asserted.

If the interrupt source is still asserted when the firmware interrupt handler handles the interrupt, the interrupt module will regenerate the interrupt, causing the interrupt handler to be invoked again.	Edge-triggered interrupt modules can be acted immediately, no matter how the interrupt source behaves.
Level-triggered interrupts are cumbersome for firmware.	Edge-triggered interrupts keep the firmware's code complexity low, reduce the number of conditions for firmware, and provide more flexibility when interrupts are handled.

Enabling and Disabling an Interrupt

Upon Reset, all the interrupts are disabled even if they are activated. The interrupts must be enabled using software in order for the microcontroller to respond to those interrupts.

IE (interrupt enable) register is responsible for enabling and disabling the interrupt. IE is a bitaddressable register.

Interrupt Enable Register

EA	-	ET2	ES	ET1	EX1	ET0	EX0

- **EA** − Global enable/disable.
- **-** − Undefined.
- **ET2** − Enable Timer 2 interrupt.
- **ES** − Enable Serial port interrupt.
- **ET1** − Enable Timer 1 interrupt.
- **EX1** − Enable External 1 interrupt.
- **ET0** − Enable Timer 0 interrupt.
- **EX0** − Enable External 0 interrupt.

To enable an interrupt, we take the following steps −

- Bit D7 of the IE register (EA) must be high to allow the rest of register to take effect.
- If EA = 1, interrupts will be enabled and will be responded to, if their corresponding bits in IE are high.
  If EA = 0, no interrupts will respond, even if their associated pins in the IE register are high.

Interrupt Priority in 8051

We can alter the interrupt priority by assigning the higher priority to any one of the interrupts. This is accomplished by programming a register called **IP**(interrupt priority).

The following figure shows the bits of IP register. Upon reset, the IP register contains all 0's. To give a higher priority to any of the interrupts, we make the corresponding bit in the IP register high.

-	-	-	-	PT1	PX1	PT0	PX0

-	IP.7	Not Implemented.
-	IP.6	Not Implemented.
-	IP.5	Not Implemented.
-	IP.4	Not Implemented.

PT1	IP.3	Defines the Timer 1 interrupt priority level.
PX1	IP.2	Defines the External Interrupt 1 priority level.
PT0	IP.1	Defines the Timer 0 interrupt priority level.
PX0	IP.0	Defines the External Interrupt 0 priority level.

Interrupt inside Interrupt

What happens if the 8051 is executing an ISR that belongs to an interrupt and another one gets active? In such cases, a high-priority interrupt can interrupt a low-priority interrupt. This is known as **interrupt inside interrupt**. In 8051, a low-priority interrupt can be interrupted by a high-priority interrupt, but not by any another low-priority interrupt.

Triggering an Interrupt by Software

There are times when we need to test an ISR by way of simulation. This can be done with the simple instructions to set the interrupt high and thereby cause the 8051 to jump to the interrupt vector table. For example, set the IE bit as 1 for timer 1. An instruction **SETB TF1** will interrupt the 8051 in whatever it is doing and force it to jump to the interrupt vector table.

PIC Microcontrollers

PIC microcontroller was developed in the year 1993 by microchip technology. The term PIC stands for Peripheral Interface Controller. Initially this was developed for supporting PDP computers to control its peripheral devices, and therefore, named as a peripheral interface device. These microcontrollers are very fast and easy to execute a program compared with other microcontrollers. PIC Microcontroller architecture is based on Harvard and RISC architecture. PIC microcontrollers are very popular due to their ease of programming, wide availability, easy to interfacing with other peripherals, low cost, large user base and serial programming capability (reprogramming with flash memory), etc. PIC mainly targeted for Low End Applications

We know that the microcontroller is an integrated chip which consists of CPU, RAM, ROM, timers, and counters, etc. In the same way, PIC microcontroller architecture consists of RAM, ROM, CPU, timers, counters and supports the protocols such as SPI, CAN, and UART for interfacing with other peripherals. At present PIC microcontrollers are extensively used for industrial purpose due to low power consumption, high performance ability and easy of availability of its supporting hardware and software tools like compilers, debuggers and simulators.

What is a PIC Microcontroller?******

PIC (Programmable Interface Controllers) microcontrollers are the worlds smallest microcontrollers that can be programmed to carry out a huge range of tasks. These microcontrollers are found in many electronic devices such as phones, computer control systems, alarm systems, embedded systems, etc. Various types of microcontrollers exist, even though the best are found in the GENIE range of programmable microcontrollers. These microcontrollers are programmed and simulated by a circuit-wizard software.

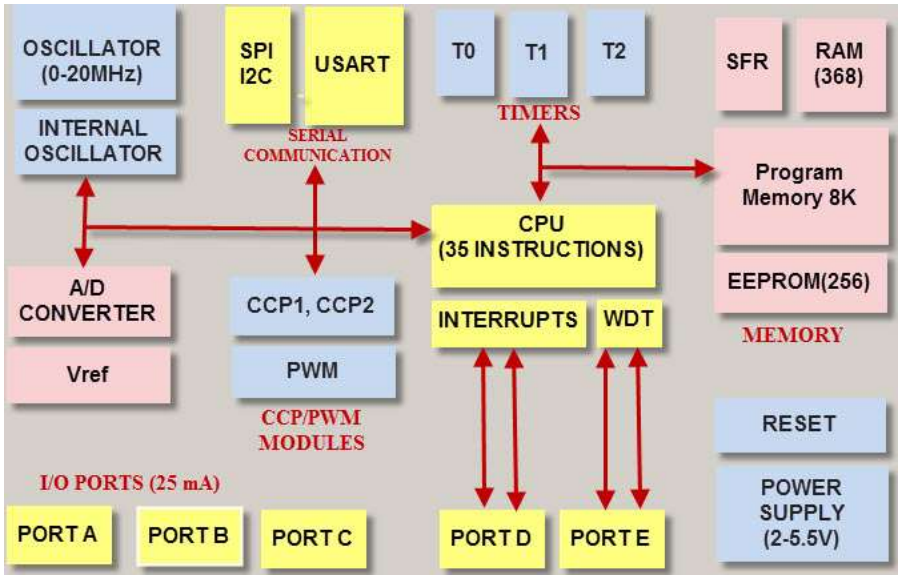
Every PIC microcontroller architecture consists of some registers and stack where registers function as Random Access Memory(RAM) and stack saves the return addresses. The main features of PIC microcontrollers are RAM, flash memory, Timers/Counters, EEPROM, I/O Ports, USART, CCP (Capture/Compare/PWM module), SSP, Comparator, ADC (analog to digital converter), PSP(parallel slave port), LCD and ICSP (in circuit serial programming) The 8-bit PIC microcontroller is classified into four types on the basis of internal architecture such as Base Line PIC, Mid Range PIC, Enhanced Mid Range PIC and PIC18

Architecture of PIC Microcontroller: Most Important*****

The PIC microcontroller architecture comprises of CPU, I/O ports, memory organization, A/D converter, timers/counters, interrupts, serial communication, oscillator and CCP module which are discussed in detailed below.

CPU (Central Processing Unit)

It is not different from other microcontrollers CPU and the PIC microcontroller CPU consists of the ALU, CU, MU and accumulator, etc. Arithmetic logic unit is mainly used for arithmetic operations and to take logical decisions. Memory is used for storing the instructions after processing. To control the internal and external peripherals, control unit is used which are connected to the CPU and the accumulator is used for storing the results and further process.



Architecture of PIC Microcontroller

Memory Organization

The memory module in the PIC microcontroller architecture consists of RAM (Random Access Memory), ROM (Read Only Memory) and STACK.

Random Access Memory (RAM)

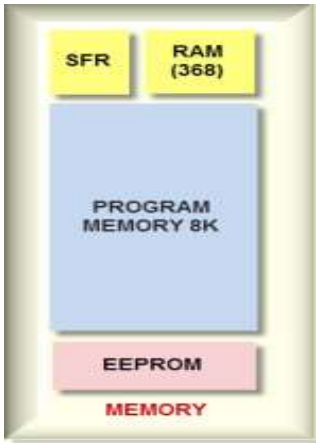
RAM is an unstable memory which is used to store the data temporarily in its registers. The RAM memory is classified into two banks, and each bank consists of so many registers. The RAM registers are classified into two types: Special Function Registers (SFR) and General Purpose Registers (GPR).

- ***General Purpose Registers (GPR)***

These registers are used for general purpose only as the name implies. For example, if we want to multiply two numbers by using the PIC microcontroller. Generally, we use registers for multiplying and storing the numbers in other registers. So these registers don't have any special function - CPU can easily access the data in the registers.

- ***Special Function Registers***

These registers are used for special purposes only as the name SFR implies. These registers will perform according to the functions assigned to them, and they cannot be used as normal registers. For example, if you cannot use the STATUS register for storing the data, these registers are used for showing the operation or status of the program. So, user cannot change the function of the SFR; the function is given by the retailer at the time of manufacturing.



Memory Organization

Read Only Memory (ROM)

Read only memory is a stable memory which is used to store the data permanently. In PIC microcontroller architecture, the architecture ROM stores the instructions or program, according to the program the microcontroller acts. The ROM is also called as program memory, wherein the user will write the program for microcontroller and saves it permanently, and finally the program is executed by the CPU. The microcontrollers performance depends on the instruction, which is executed by the CPU.

Electrically Erasable Programmable Read Only Memory (EEPROM)

In the normal ROM, we can write the program for only once we cannot use again the microcontroller for multiple times. But, in the EEPROM, we can program the ROM multiple times.

Flash Memory

Flash memory is also programmable read only memory (PROM) in which we can read, write and erase the program thousands of times. Generally, the PIC microcontroller uses this type of ROM.

Stack

When an interrupt occurs, first the PIC microcontroller has to execute the interrupt and the existing process address. Then that is being executed is stored in the stack. After completing the execution of the interrupt, the microcontroller calls the process with the help of address, which is stored in the stack and get executes the process.

I/O Ports

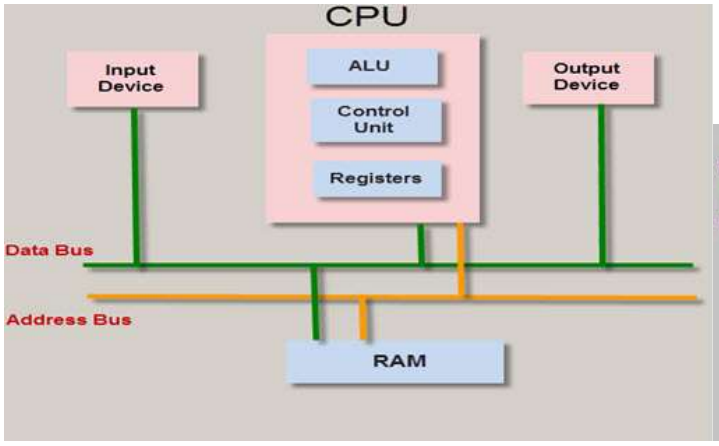
- The series of PIC16 consists of five ports such as Port A, Port B, Port C, Port D & Port E.
- Port A is an 16-bit port that can be used as input or output port based on the status of the TRISA (Tradoc Intelligence Support Activity) register.
- Port B is an 8- bit port that can be used as both input and output port.
- Port C is an 8-bit and the input of output operation is decided by the status of the TRISC register.
- Port D is an 8-bit port acts as a slave port for connection to the microprocessor BUS.
- Port E is a 3-bit port which serves the additional function of the control signals to the analog to digital converter.

BUS

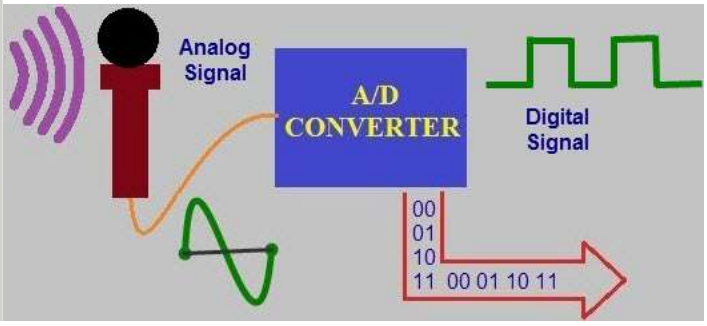
BUS is used to transfer and receive the data from one peripheral to another. It is classified into two types such as data bus and address.

Data Bus: It is used for only transfer or receive the data.

Address Bus: Address bus is used to transmit the memory address from the peripherals to the CPU. I/O pins are used to interface the external peripherals; UART and USART both are serial communication protocols which are used for interfacing serial devices like GSM, GPS, Bluetooth, IR , etc.



BUS



A/D CONVERTER

A/D converters

The main intention of this analog to digital converter is to convert analog voltage values to digital voltage values. A/D module of PIC microcontroller consists of 5 inputs for 28 pin devices and 8 inputs for 40 pin devices. The operation of the analog to digital converter is controlled by ADCON0 and ADCON1 special registers. The upper bits of the converter are stored in register ADRESH and lower bits of the converter are stored in register ADRESL. For this operation, it requires 5V of an analog reference voltage.

Timers/ Counters

PIC microcontroller has four timer/counters wherein the one 8-bit timer and the remaining timers have the choice to select 8 or 16-bit mode. Timers are used for generating accuracy actions, for example, creating specific time delays between two operations.

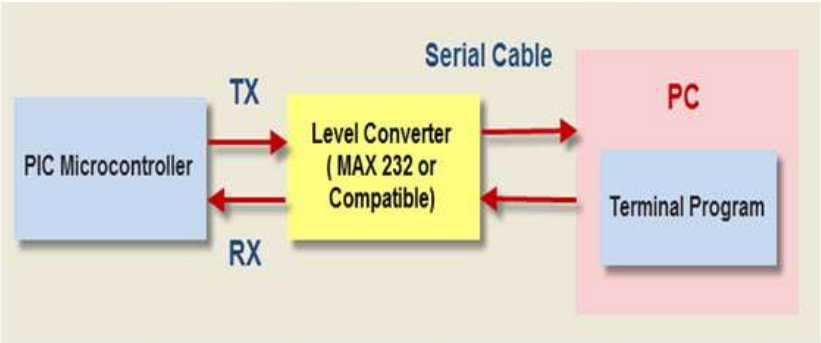
Interrupts

PIC microcontroller consists of 20 internal interrupts and three external interrupt sources which are associated with different peripherals like ADC, USART, Timers, and so on.

Serial Communication

Serial communication is the method of transferring data one bit at a time sequentially over a communication channel.

- **USART:** The name USART stands for Universal synchronous and Asynchronous Receiver and Transmitter which is a serial communication for two protocols. It is used for transmitting and receiving the data bit by bit over a single wire with respect to clock pulses. The PIC microcontroller has two pins TXD and RXD. These pins are used for transmitting and receiving the data serially.
- **SPI Protocol:** The term SPI stands for Serial Peripheral Interface. This protocol is used to send data between PIC microcontroller and other peripherals such as SD cards, sensors and shift registers. PIC microcontroller support three wire SPI communications between two devices on a common clock source. The data rate of SPI protocol is more than that of the USART.
- **I2C Protocol:** The term I2C stands for Inter Integrated Circuit , and it is a serial protocol which is used to connect low speed devices such as EEPROMS, microcontrollers, A/D converters, etc. PIC microcontroller support two wire Interface or I2C communication between two devices which can work as both Master and Slave device.



Serial Communication

Oscillators

Oscillators are used for timing generation. Pic microcontroller consist of external oscillators like RC oscillators or crystal oscillators. Where the crystal oscillator is connected between the two oscillator pins. The value of the capacitor is connected to every pin that decides the mode of the operation of the oscillator. The modes are crystal mode, high-speed mode and the low-power mode. In case of RC oscillators, the value of the resistor & capacitor determine the clock frequency and the range of clock frequency is 30KHz to 4MHz.

CCP module: The name CCP module stands for capture/compare/PWM where it works in three modes such as capture mode, compare mode and PWM mode.

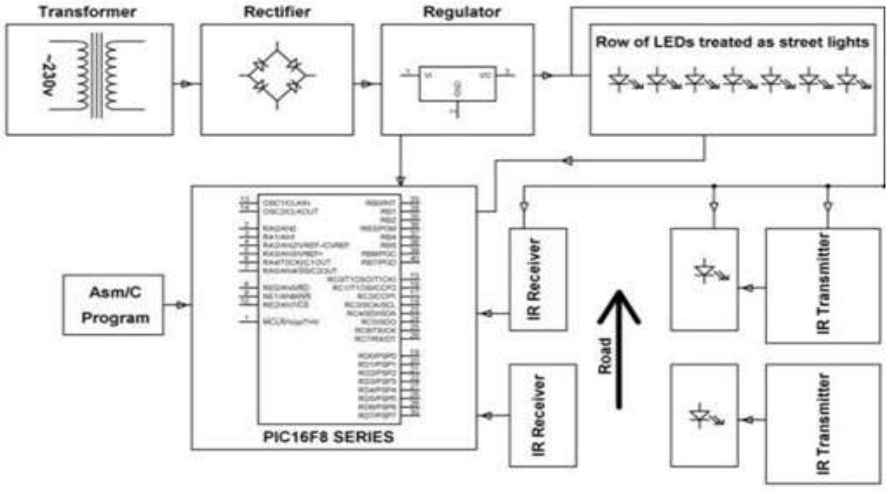
- **Capture Mode:** Capture mode captures the time of arrival of a signal, or in other words, when the CCP pin goes high, it captures the value of the Timer1.
- **Compare Mode:** Compare mode acts as an analog comparator. When the timer1 value reaches a certain reference value, then it generates an output.
- **PWM Mode:** PWM mode provides pulse width modulated output with a 10-bit resolution and programmable duty cycle.

PIC Microcontroller Applications

The PIC microcontroller projects can be used in different applications, such as peripherals, audio accessories, video games, etc. For better understanding of this PIC microcontroller, the following project demonstrates PIC microcontroller's operations. PIC is mainly targeted for Low End Applications

Street Light that Glows on Detecting Vehicle Movement:

The main intention of this project is to detect the movement of vehicles on highways to switch on a block of street lights ahead of it, and also switch off the trailing lights to conserve energy. In this project, a PIC microcontroller is done by using assembly language or embedded C.



Street Light that Glows on Detecting Vehicle Movement

The power supply gives the power to the total circuit by stepping down, rectifying, filtering and regulating AC mains supply. When there are no vehicles on highway, then all lights will turn OFF so that the power can be conserved. The IR sensors are placed on the road to sense the vehicle movement. When there are vehicles on highway, then the IR sensor senses the vehicle movement immediately, it sends the commands to the PIC microcontroller to switch ON/OFF the LEDs. A bunch of LEDS will be turned on when a vehicle come near to the sensor and once the vehicle passes away from the sensor the intensity will become lower than the LEDs will turn OFF

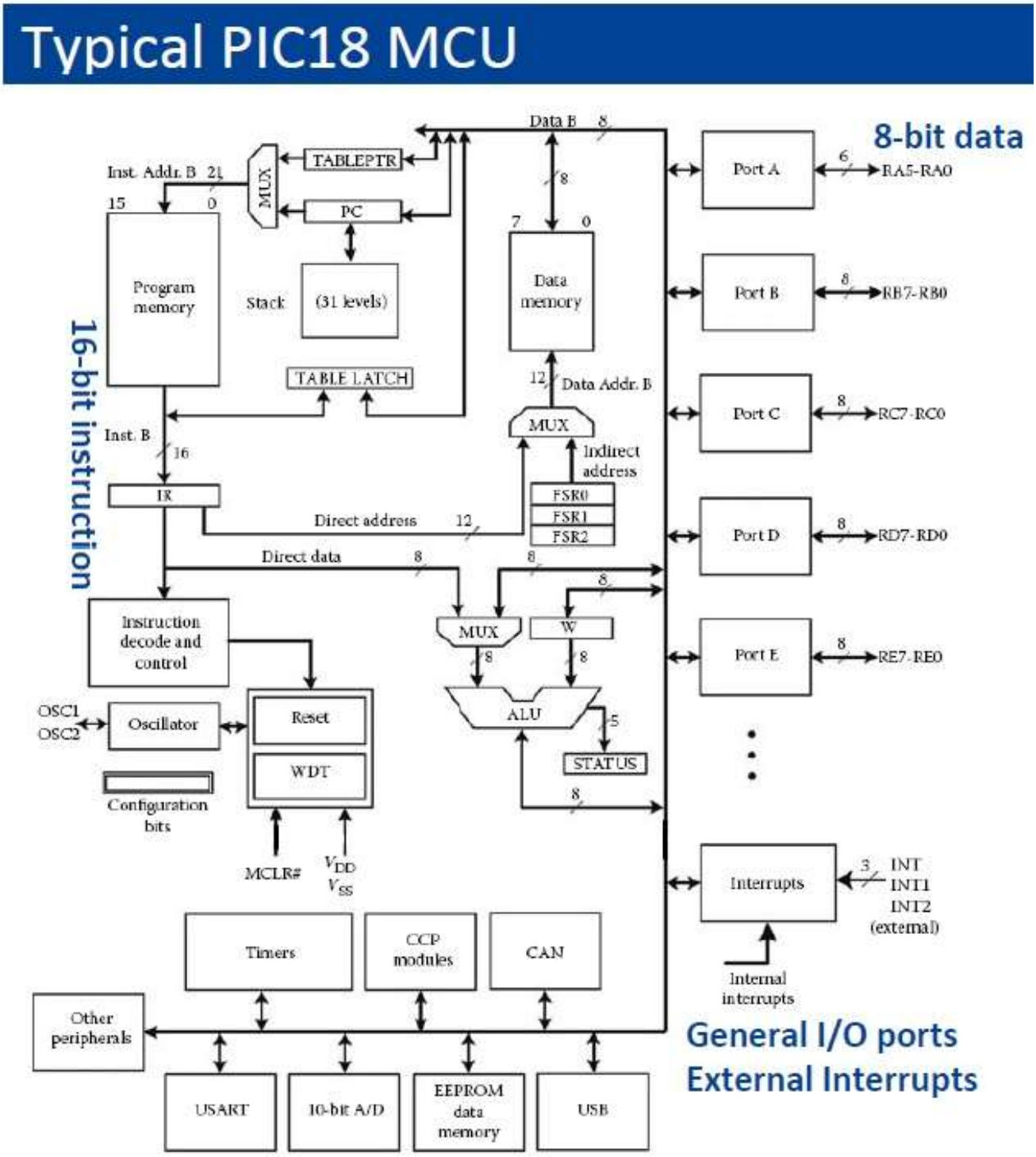
Advantages of PIC Microcontroller:

- PIC microcontrollers are consistent and faulty of PIC percentage is very less. The performance of the PIC microcontroller is very fast because of using RISC architecture.
- When comparing to other microcontrollers, power consumption is very less and programming is also very easy.
- Interfacing of an analog device is easy without any extra circuitry

Disadvantages of PIC Microcontroller:

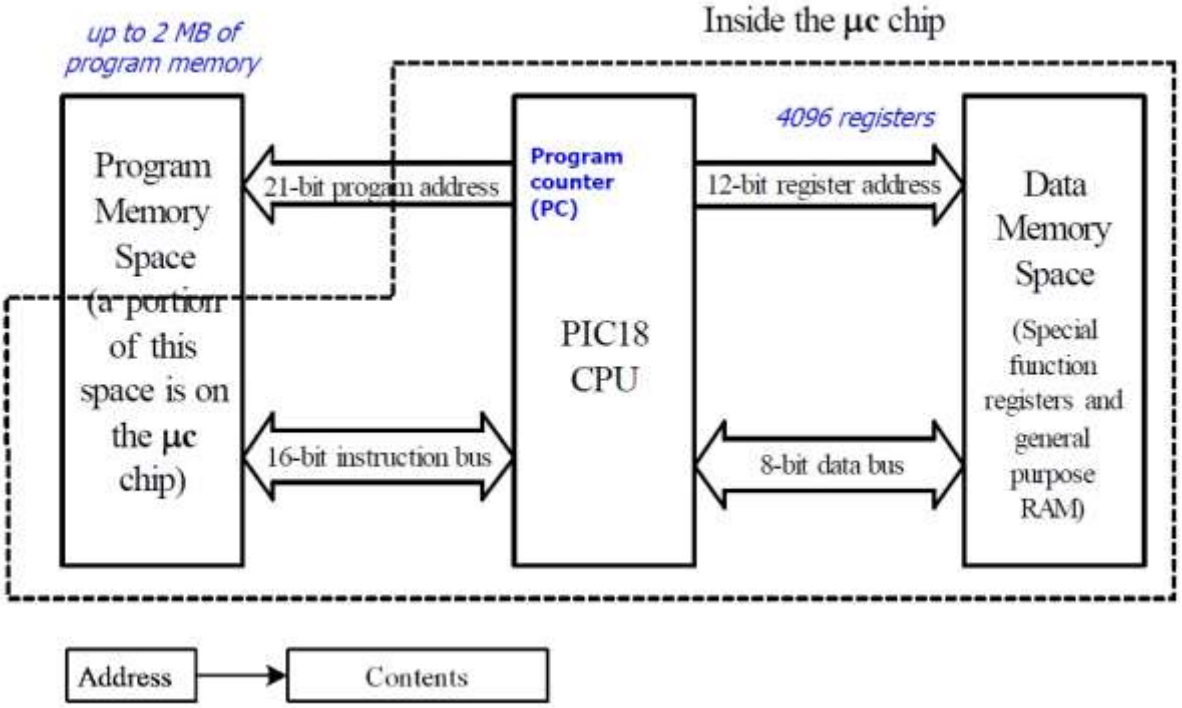
- The length of the program is high due to using RISC architecture (35 instructions)
- One single accumulator is present and program memory is not accessible

PIC18 MCU Detailed Architecture:

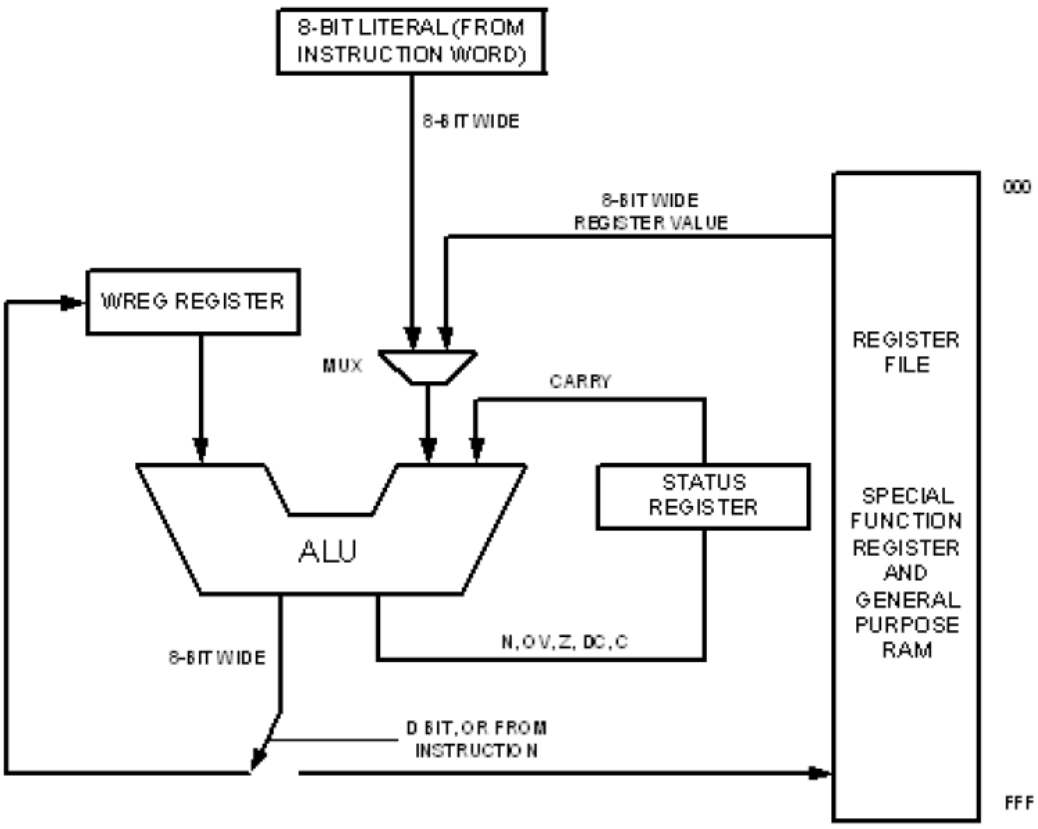


Using this detailed architecture we can explain the data moves, arithmetic and logical operations takes place and how the timing & control signals will be generated after instruction fetch and decode. From the above diagram, as the working register or W register is connected to ALU directly, one operand is always present in W and the 2nd operand will come from either instruction register or from the memory. After specified operation performed ALU will set the flags and stored the result as specified in the instruction.

CPU Architecture of PIC18 Microcontroller:******



Instruction Datapath PIC18 Microcontroller:******



PIC Microcontrollers

1 : Introduction to PIC Microcontrollers

PIC Microcontrollers********

PIC stands for Peripheral Interface Controller given by Microchip Technology to identify its single-chip microcontrollers. These devices have been very successful in 8-bit microcontrollers. The main reason is that Microchip Technology has continuously upgraded the device architecture and added needed peripherals to the microcontroller to suit customers' requirements. The development tools such as assembler and simulator are freely available on the internet at www.microchip.com .

The architectures of various PIC microcontrollers can be divided as follows.

Low - end PIC Architectures :*******

Microchip PIC microcontrollers are available in various types. When PIC microcontroller MCU was first available from General Instruments in early 1980's, the microcontroller consisted of a simple processor executing 12-bit wide instructions with basic I/O functions. These devices are known as low-end architectures. They have limited program memory and are meant for applications requiring simple interface functions and small program & data memories. Some of the low-end device numbers are

12C5XX
16C5X
16C505

Mid range PIC Architectures

Mid range PIC architectures are built by upgrading low-end architectures with more number of peripherals, more number of registers and more data/program memory. Some of the mid-range devices are

16C6X
16C7X
16F87X

Program memory type is indicated by an alphabet.

C = EPROM
F = Flash
RC = Mask ROM

Popularity of the PIC microcontrollers is due to the following factors.

1. Speed: Harvard Architecture, RISC architecture, 1 instruction cycle = 4 clock cycles.
2. Instruction set simplicity: The instruction set consists of just 35 instructions (as opposed to 111 instructions for 8051).
3. Power-on-reset and brown-out reset. Brown-out-reset means when the power supply goes below a specified voltage (say 4V), it causes PIC to reset; hence malfunction is avoided. A watch dog timer (user programmable) resets the processor if the software/program ever malfunctions and deviates from its normal operation.
4. PIC microcontroller has four optional clock sources.
    - Low power crystal
    - Mid range crystal
    - High range crystal
    - RC oscillator (low cost).
5. Programmable timers and on-chip ADC.
6. Up to 12 independent interrupt sources.
7. Powerful output pin control (25 mA (max.) current sourcing capability per pin.)
8. EPROM/OTP/ROM/Flash memory option.
9. I/O port expansion capability.
10. Free assembler and simulator support from Microchip at www.microchip.com

CPU Architecture: *******

The CPU uses Harvard architecture with separate Program and Variable (data) memory interface. This facilitates instruction fetch and the operation on data/accessing of variables simultaneously.

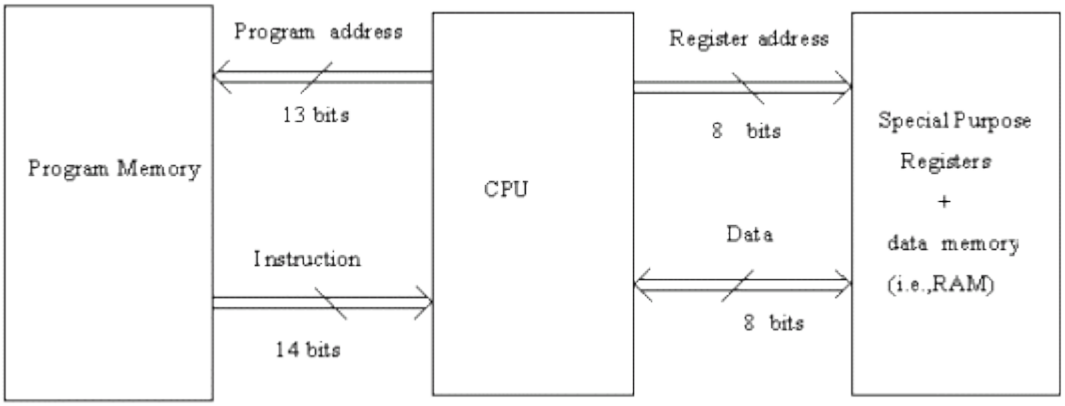


Fig 16.1 CPU Architecture of PIC microcontroller

PIC Memory Organisation:*******

PIC microcontroller has 13 bits of program memory address. Hence it can address up to 8k of program memory. The program counter is 13-bit. PIC 16C6X or 16C7X program memory is 2k or 4k. While addressing 2k of program memory, only 11- bits are required. Hence two most significant bits of the program counter are ignored. Similarly, while addressing 4k of memory, 12 bits are required. Hence the MSb of the program counter is ignored.

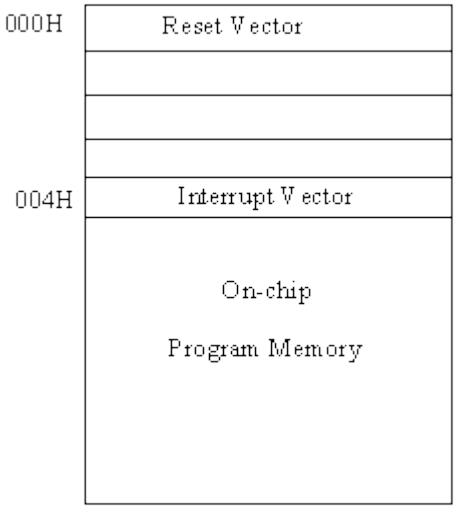


Fig 16.2 Program Memory map

The program memory map of PIC16C74A is shown in Fig 16.2.
On reset, the program counter is cleared and the program starts at 00H. Here a 'goto' instruction is required that takes the processor to the mainline program.

When a peripheral interrupt, that is enabled, is received, the processor goes to 004H. A suitable branching to the interrupt service routine (ISR) is written at 004H.

Data memory (Register Files):

Data Memory is also known as Register File. Register File consists of two components.

1. General purpose register file (same as RAM).
2. Special purpose register file (similar to SFR in 8051).

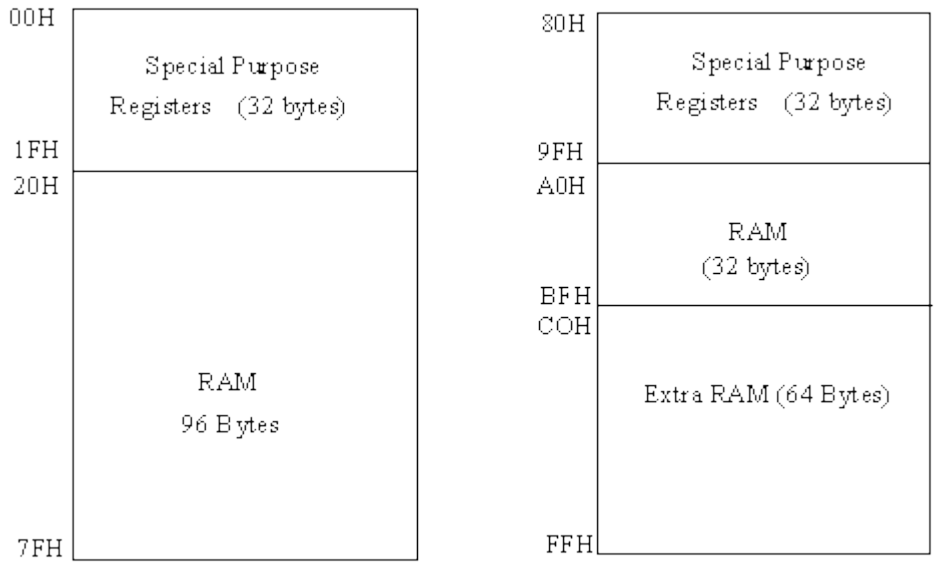


Fig 16.3 Data Memory map

The special purpose register file consists of input/output ports and control registers. Addressing from 00H to FFH requires 8 bits of address. However, the instructions that use direct addressing modes in PIC to address these register files use 7 bits of instruction only. Therefore the register bank select (RP0) bit in the STATUS register is used to select one of the register banks.

In indirect addressing FSR register is used as a pointer to anywhere from 00H to FFH in the data memory.

2 : Basic Architecture of PIC Microcontrollers

Specifications of some popular PIC microcontrollers are as follows:

Device	Program Memory (14bits)	Data RAM (bytes)	I/O Pins	ADC	Timers 8/16 bits	CCP (PWM)	USART SPI / I2C
16C74A	4K EPROM	192	33	8 bits x 8 channels	2/1	2	USART SPI / I^2C
16F877	8K Flash	368 (RAM) 256 (EEPROM)	33	10 bits x 8 channels	2/1	2	USART SPI / I^2C

Device	Interrupt Sources	Instruction Set
16C74A	12	35
16F877	15	35

PIC Microcontroller Clock

Most of the PIC microcontrollers can operate upto 20MHz. One instructions cycle (machine cycle) consists of four clock cycles.

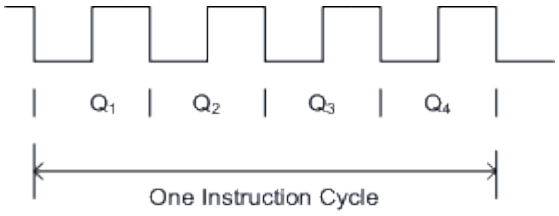


Fig 17.1 Relation between instruction cycles and clock cycles for PIC microcontrollers

Instructions that do not require modification of program counter content get executed in one instruction cycle.

Although the architectures of various midrange 8 - bit PIC microcontroller are not the same, the variation is mostly interns of addition of memory and peripherals. We will discuss here the architecture of a standard mid-range PIC microcontroller, 16C74A. Unless mentioned otherwise, the information given here is for a PIC 16C74A microcontroller Chip.

Architecture of PIC16C74A**********

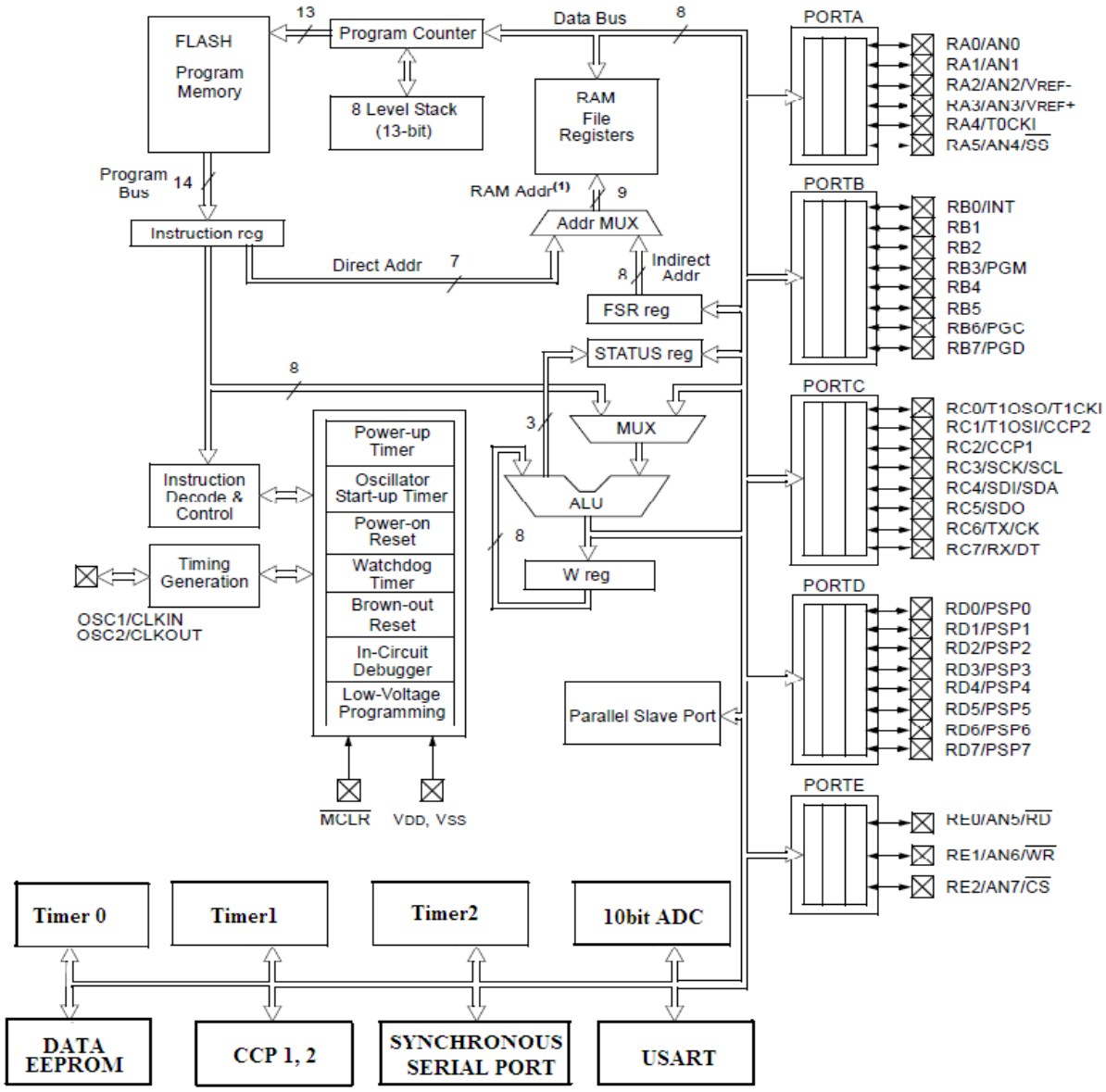


Fig 17.2 Basic Architecture of PIC 16C74A

The basic architecture of PIC16C74A is shown in fig 17.2. The architecture consists of Program memory, file registers and RAM, ALU and CPU registers. It should be noted that the program Counter is 13 - bit and the program memory is organised as 14 - bit word. Hence the program Memory capacity is 8k x 14 bit. Each instruction of PIC 16C74A is 14 - bit long. The various CPU registers are discussed here.

CPU registers (registers commonly used by the CPU)

W, the working register, is used by many instructions as the source of an operand. This is similar to accumulator in 8051. It may also serve as the destination for the result of the instruction execution. It is an 8 - bit register.

W, Working register

Fig 17.3 W register

STATUS Register

The STATUS register is a 8-bit register that stores the status of the processor. This also stores carry, zero and digit carry bits.

STATUS - address 03H, 83H

7	6	5	4	3	2	1	0
0	0	RP0	NOT_TO	NOT_PD	Z	DC	C

Fig 17.4 STATUS register

C = Carry bit
DC = Digit carry (same as auxiliary carry)
Z = Zero bit
NOT_TO and NOT_PD - Used in conjunction with PIC's sleep mode
RP0- Register bank select bit used in conjunction with direct addressing mode.

FSR Register

(File Selection Register, address = 04H, 84H)
FSR is an 8-bit register used as data memory address pointer. This is used in indirect addressing mode.

INDF Register

(INDirect through FSR, address = 00H, 80H)
INDF is not a physical register. Accessing INDF access is the location pointed to by FSR in indirect addressing mode.

PCL Register

(Program Counter Low Byte, address = 02H, 82H)
PCL is actually the lower 8-bits of the 13-bit program counter. This is a both readable and writable register.

PCLATH Register

(Program Counter Latch, address = 0AH, 8AH)
PCLATH is a 8-bit register which can be used to decide the upper 5bits of the program counter. PCLATH is not the upper 5bits of the program counter. PCLATH can be read from or written to without affecting the program counter. The upper 3bits of PCLATH remain zero and they serve no purpose. When PCL is written to, the lower 5bits of PCLATH are automatically loaded to the upper 5bits of the program counter, as shown in the figure.

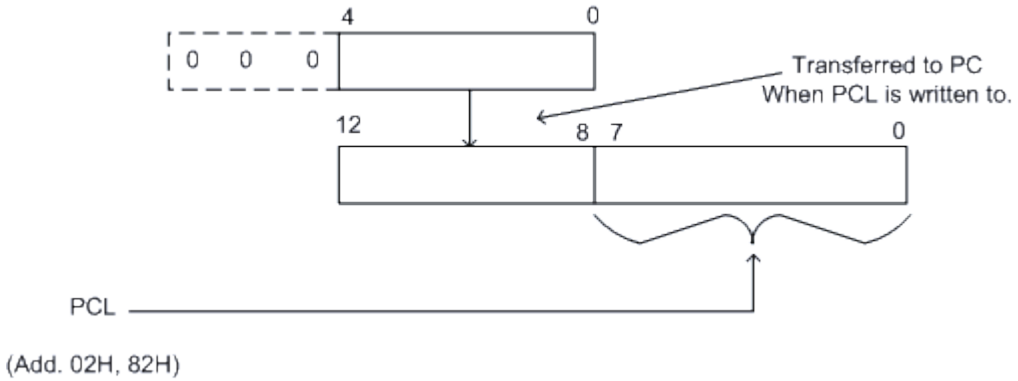


Fig 17.5 Schematic of how PCL is loaded from PCLATH

Program Counter Stack

An independent 8-level stack is used for the program counter. As the program counter is 13bit, the stack is organized as 8x13bit registers. When an interrupt occurs, the program counter is pushed onto the stack. When

the interrupt is being serviced, other interrupts remain disabled. Hence, other 7 registers of the stack can be used for subroutine calls within an interrupt service routine or within the mainline program.

Register File Map

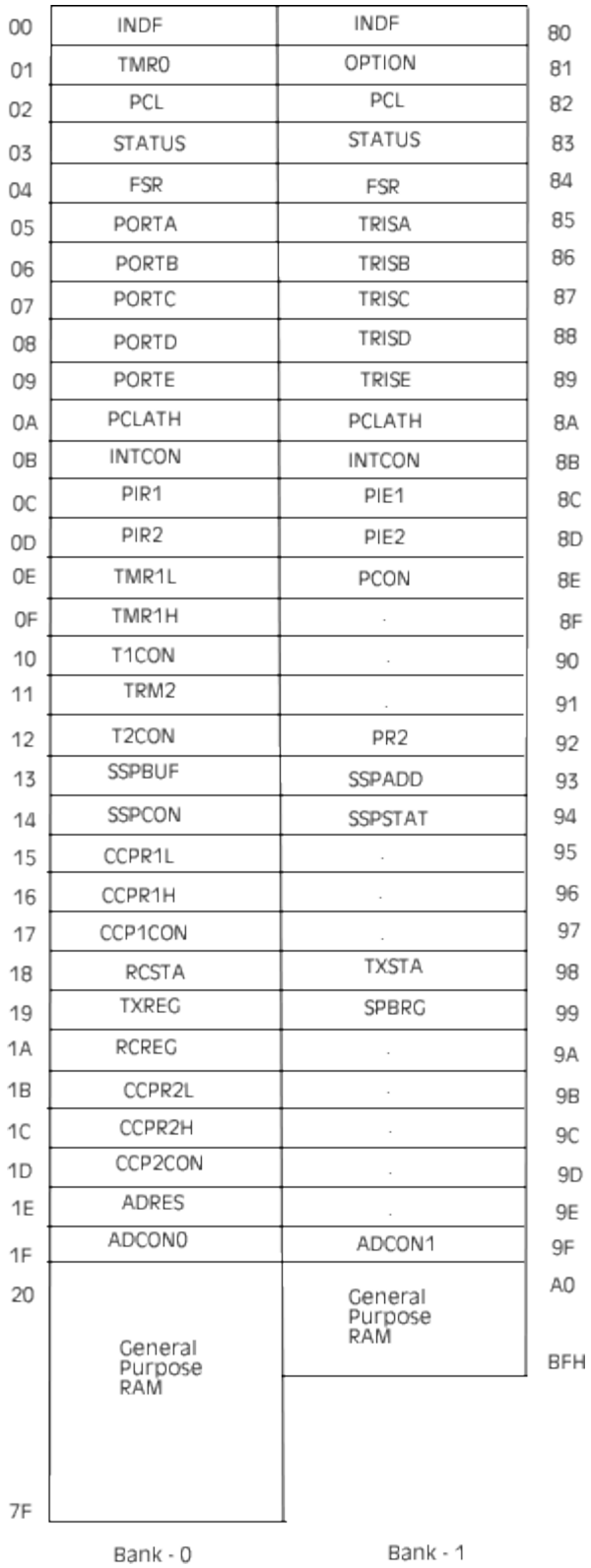


Fig 17.6 Register File Map

It can be noted that some of the special purpose registers are available both in Bank-0 and Bank-1. These registers have the same value in both banks. Changing the register content in one bank automatically changes its content in the other bank.

Port Structure and Pin Configuration of PIC 16C74A

As mentioned earlier, there is a large variety of PIC microcontrollers. However, the midrange architectures are widely used. Our discussion will mainly confine to PIC16C74A whose architecture has most of the required features of a mid-range PIC microcontroller. Study of any other mid-range PIC microcontroller will not cause much variation from the basic architecture of PIC 16C74A ..

PIC 16C74A has 5 I/O Ports. Each port is a bidirectional I/O port. In addition, they have the following alternate functions.

Port	Altenative uses of I/O pins	No.of I/O pins
Port A	A/D Converter inputs	6
Port B	External interrupt inputs	8
Port C	Serial port, Timer I/O	8
Port D	Parallel slave port	8
Port E	A/D Converter inputs	3
	Total I/O pins	33
	Total pins	40

In addition to I/O pins, there is a Master clear pin (MCLR) which is equivalent to reset in 8051. However, unlike 8051, MCLR should be pulled low to reset the micro controller. Since PIC16C74Ahas inherent power-on reset, no special connection is required with MCLR pin to reset the micro controller on power-on.

There are two V_{DD} pins and two V_{SS} pins. There are two pins (OSC1 and OSC2) for connecting the crystal oscillator/ RC oscillator. Hence the total number of pins with a 16C74A is 33+7=40. This IC is commonly available in a dual-in-pin (DIP) package.

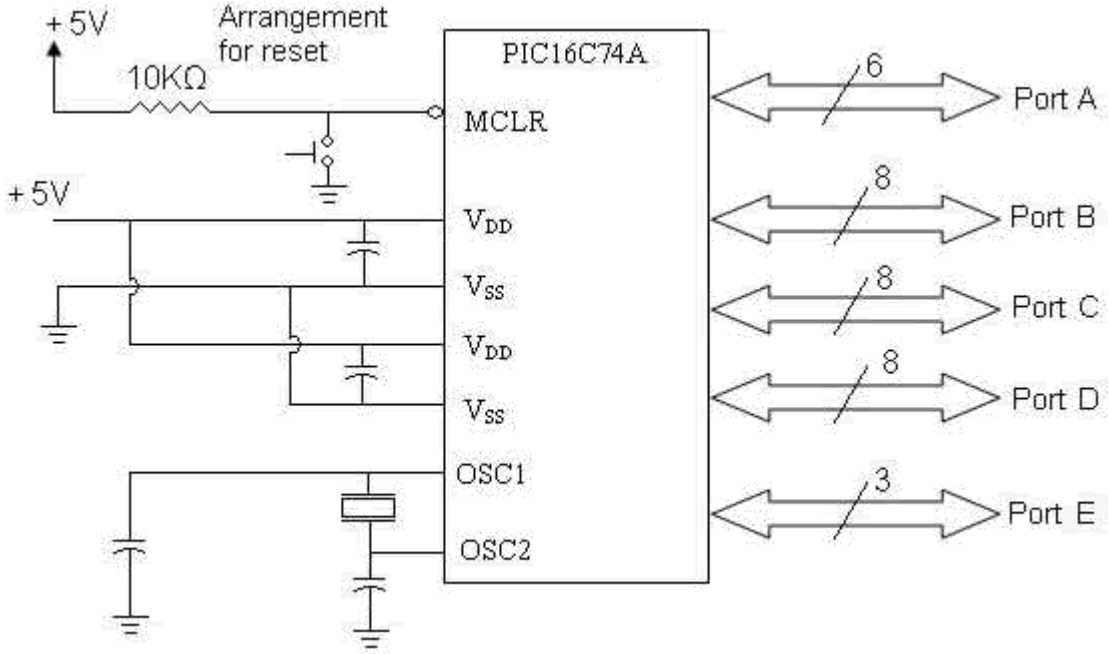


Fig 17.7 Pin configuration of PIC 16C74A

3 : Instruction Set of PIC Microcontroller

Guidelines from Microchip Technology

For writing assembly language program Microchip Technology has suggested the following guidelines.

1. Write instruction mnemonics in lower case. (e.g., movwf)
2. Write the special register names, RAM variable names and bit names in upper case. (e.g., PCL, RP0, etc.)
3. Write instructions and subroutine labels in mixed case. (e.g., Mainline, LoopTime)

ADDRESSING MODES:**********

To know the working principal and data handling we need to have clear knowledge on addressing modes of pic microcontroller. The PIC microcontrollers support three addressing modes .They are

(i) **Immediate** or Literal Addressing Mode

(ii) Register Direct or Memory **Direct** Addressing Mode

(iii) **Indirect** Addressing Mode

Immediate addressing mode:

In this addressing mode, the operand is a number or constant not an address as MOVLW 43h, the operand here is data not address. So in this addressing mode of pic microcontroller data is direct transferred and data is immediate after opcode that is why this type of addressing is called immediate addressing. This way is fast in execution.

Memory operand addressing mode :

In this addressing mode, the operand is an address of Memory location which holds the data to be executed. Again memory operand addressing mode is fallen under two categories

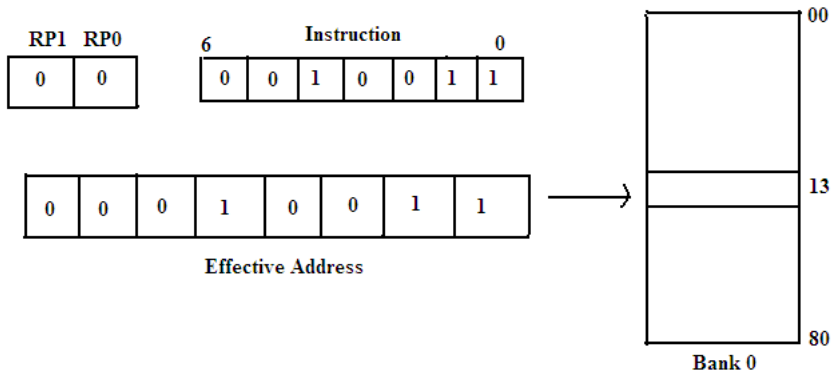
1) Direct addressing like CLRF 13h. We deal with the address or the memory location.
2) Indirect addressing in which we use INDF and FSR registers.

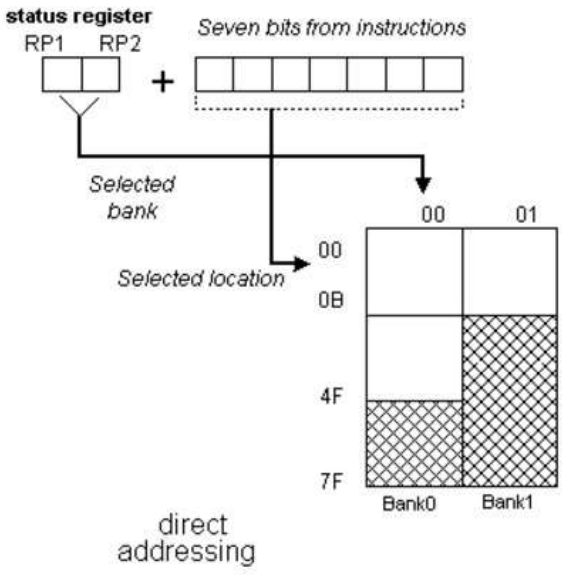
Direct addressing:

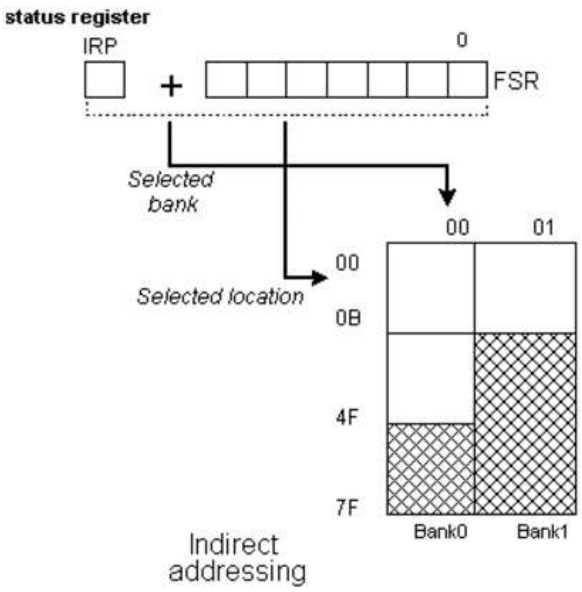
Direct Addressing is done through a 9-bit address. In direct addressing mode, 7 bits (0-6) of the instruction identify the register file address and two bits (RP1, RP0) from STATUS register as is shown in bellow Figure. Any access to SFR(Special Function Registers) registers can be an example of direct addressing.

Register operand addressing mode: it also same as the Direct addressing mode as the register are memory locations in the Data memory or RAM. In this addressing mode, the operand is a Register which holds the data to be execute. Register operand addressing mode deals with the registers like: CLR W

The below diagram explains the method of accessing register file address 13H by direct addressing method







Indirect addressing:

It does not take an address from an instruction, but it derives it from IRP bit of STATUS and FSR(File Selection Register) registers. Addressed location is accessed through INDF register which in fact holds the address indicated by the FSR. Indirect addressing is very convenient for manipulating data arrays located in GPR registers. In this case, it is necessary to initialize FSR register with a starting address of the array, and the rest of the data can be accessed by incrementing the FSR register. Above Figure shows the indirect addressing concept.

Instruction Set:

The instruction set for PIC16C74A consists of only 35 instructions. Some of these instructions are byte oriented instructions and some are bit oriented instructions.

The **byte oriented instructions** that require two parameters (For example, movf f, F(W)) expect the f to be replaced by the name of a special purpose register (e.g., PORTA) or the name of a RAM variable (e.g., NUM1), which serves as the source of the operand. 'f' stands for file register. The F(W) parameter is the destination of the result of the operation. It should be replaced by:
F, if the destination is to be the source register.
W, if the destination is to be the working register (i.e., Accumulator or W register).
The **bit oriented instructions** also expect parameters (e.g., btfsc f, b). Here 'f' is to be replaced by the name of a special purpose register or the name of a RAM variable. The 'b' parameter is to be replaced by a bit number ranging from 0 to 7.

For Example:

```
Z equ 2
```

Z has been equated to 2. Here, the instruction will test the Z bit of the STATUS register and will skip the next instruction if Z bit is clear.
The **literal instructions** require an operand having a known value (e.g., 0AH) or a label that represents a known value.

For example:

```
NUM equ 0AH ;          Assigns 0AH to the label NUM ( a constant )
movlw NUM ;            will move 0AH to the W register.
```

Every instruction fits in a single 14-bit word. In addition, every instruction also executes in a single cycle, unless it changes the content of the Program Counter. These features are due to the fact that PIC micro controller has been designed on the principles of RISC (Reduced Instruction Set Computer) architecture.

Instruction set:********

Mnemonics	Description	Instruction Cycles
bcf f, b	Clear bit b of register f	1
bsf f, b	Set bit b of register f	1
clrw	Clear working register W	1
clrf f	Clear f	1
movlw k	Move literal 'k' to W	1
movwf f	Move W to f	1
movf f, F(W)	Move f to F or W	1
swapf f, F(W)	Swap nibbles of f, putting result in F or W	1
andlw k	And literal value into W	1
andwf f, F(W)	And W with F and put the result in W or F	1
andwf f, F(W)	And W with F and put the result in W or F	1
iorlw k	inclusive-OR literal value into W	1
iorwf f, F(W)	inclusive-OR W with f and put the result in F or W	1
xorlw k	Exclusive-OR literal value into W	1
xorwf f, F(W)	Exclusive-OR W with f and put the result in F or W	1
addlw k	Add the literal value to W and store the result in W	1
addwf f, F(W)	Add W to f and store the result in F or W	1
sublw k	Subtract the literal value from W and store the result in W	1
subwf f, F(W)	Subtract f from W and store the result in F or W	1
rlf f, F(W)	Copy f into F or W; rotate F or W left through the carry bit	1
rrf f, F(W)	Copy f into F or W; rotate F or W right through the carry bit	1
btfsc f, b	Test 'b' bit of the register f and skip the next instruction if bit is clear	1 / 2
btfss f, b	Test 'b' bit of the register f and skip the next instruction if bit is set	1 / 2
decfsz f, F(W)	Decrement f and copy the result to F or W; skip the next instruction if the result is zero	1 / 2
incfcz f, F(W)	Increment f and copy the result to F or W; skip the next instruction if the result is zero	1 / 2
goto label	Go to the instruction with the label "label"	2
call label	Go to the subroutine "label", push the Program Counter in the stack	2
retrun	Return from the subroutine, POP the Program Counter from the stack	2
retlw k	Retrun from the subroutine, POP the Program Counter from the stack; put k in W	2
retie	Return from Interrupt Service Routine and re-enable interrupt	2

clrwdt	Clear Watch Dog Timer	1
sleep	Go into sleep/ stand by mode	1
nop	No operation	1

Encoding of instruction:

As has been discussed, each instruction is of 14-bit long. These 14-bits contain both op-code and the operand. Some examples of instruction encoding are shown here.

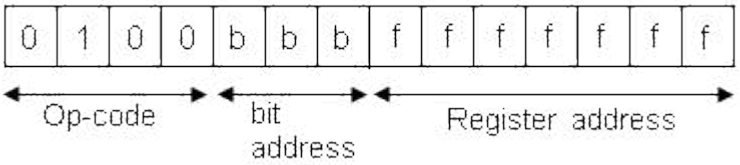
Example-1:

bcf f, b Clear 'b' bit of register 'f'

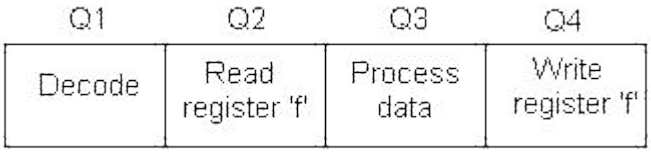
Operands: $0 \leq f \leq 127$

$0 \leq b \leq 7$

Encoding:



The instruction is executed in one instruction cycle, i.e., 4 clock cycles. The activities in various clock cycles are as follows.



Example-2:

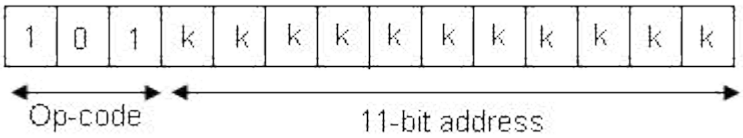
goto K Go to label 'k' instruction

Operand: $0 \leq K \leq 2047$ (11-bit address is specified)

Operation: K ⟶ PC <10:0>

PCLATH <4:3> ⟶ PC <12:11>

Encoding:



Since this instruction requires modification of program Counter, it takes two instruction cycles for execution.

Q-Cycle activities are shown as follows.

	Q1	Q2	Q3	Q4
1st instruction cycle	Decode	Read literal 'k'	Process data	Write to PC
2nd Instruction cycle	No-Operation	No-Operation	No-Operation	No-Operation

4 : I/O Port Configuration

Discussion on I/O ports of PIC16C74A:****

PIC16C74A has five I/O ports. Port-B, Port-C and Port-D have 8 pins each. Port-A and Port-E have 6 and 3 pins respectively. Each port has bidirectional digital I/O capability. In addition, these I/O ports are multiplexed with alternate functions for the peripheral devices on the microcontroller. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin. Each port latch has a corresponding TRIS (Tri-state Enable) register for configuring the port either as an input or as an output. The port pins are designated by the alphabet R, followed by the respective port (viz. A, B, C, D or E) and the pin number. For example, Port-A pins are named as RA0, RA1, etc.

Port-A: Port-A pins RA0-RA3 and RA5 are similar. These pins function (alternate function) as analog inputs to the analog-to-digital converter.

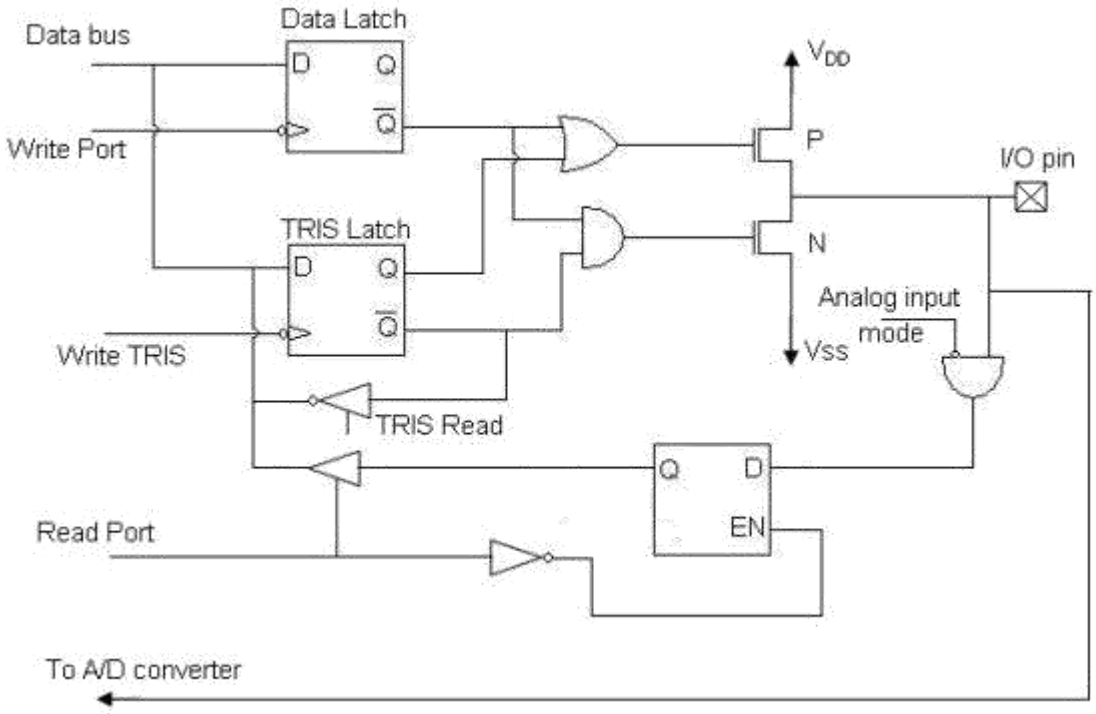


Fig 19.1 RA0-RA3 and RA5 pin of Port-A

The structure of Port-A pins RA0-RA3 and RA5 is shown in the figure. TRISA register decides whether the port-pin is configured as an input or as an output (digital) pin. Setting a TRISA register bit puts the corresponding output driver in high impedance mode. In this mode, the pin can be used as a digital or analog input. Clearing a bit in the TRISA register puts the contents of the data latch on the selected pins, i.e., the pin functions as a digital output. Pins RA0-RA and RA5 have current sourcing capability of 25mA.

The alternate function of RA4 pin is Timer-0 clock input (T0CKI). RA4 pin is an open drain pin and hence requires external pull-up when configured as output pin. It is shown in the following figure.

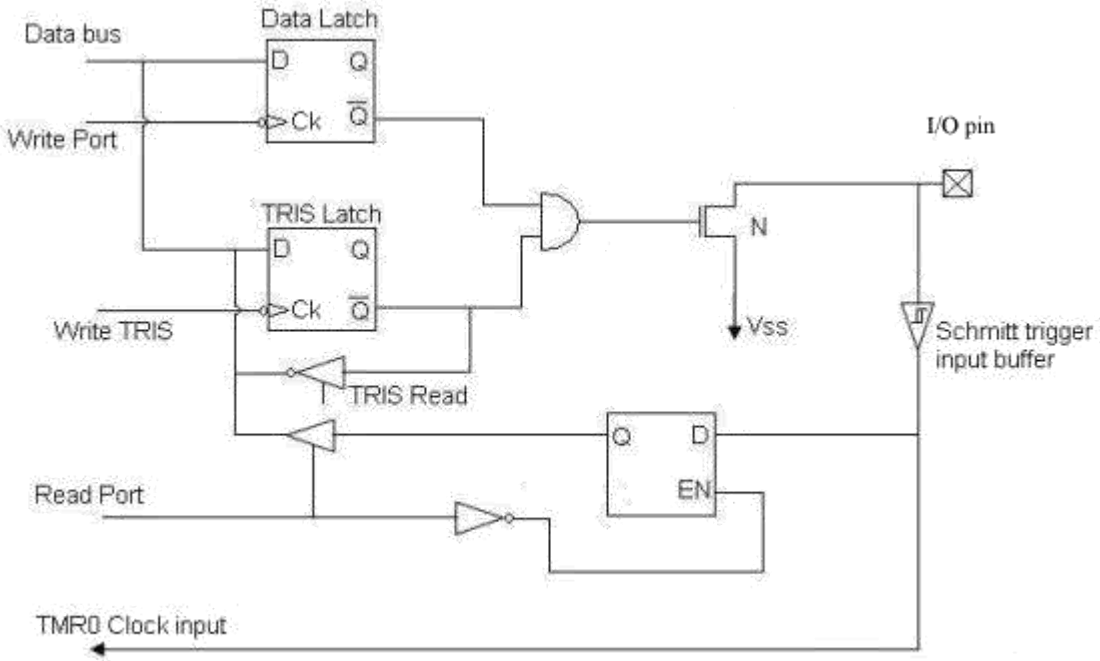


Fig 19.2 RA4 pin Configuration

Configuration of Port-A pins

Example : Set RA0-RA3 as outputs and RA4 - RA5 as inputs.

```
bcf STATUS, RP0 ;        Select Bank-0
clrf PORTA ;              Clears the data latch
bsf STATUS, RP0 ;        Select Bank-1
movlw 30H ;               W ←03H ( data direction )
movwf TRISA ;             Set RA0-RA3 as outputs, RA4-RA5 as inputs
```

Port-B

Port-B is an 8-bit bidirectional I/O port. The data direction in Port-B is controlled by TRISB register. Setting a bit in TRISB register puts the corresponding output in high impedance input mode. When a bit in TRISB is made zero, the corresponding pin in Port-B outputs the content of the latch (output mode).

Each port pin has a weak internal pull-up that can be enabled by clearing bit $\overline{\text{RBPU}}$ of OPTION register (bit-7). When a pin is configured in the output mode, the weak pull-up is automatically turned off. Internal pull-up is used so that we can directly drive a device from the pins.

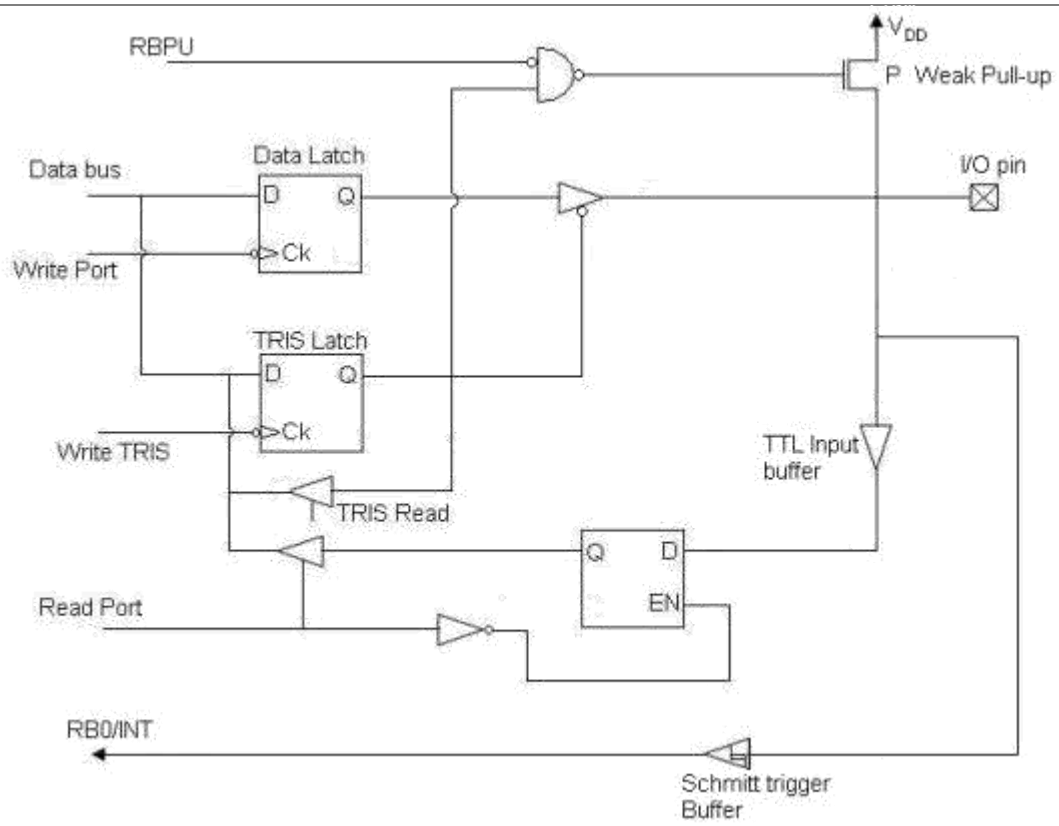


Fig 19.3 Pins RB0-RB3 of Port-B

Configuration of Port-B pins

Example : Set RB0-RB3 as outputs, RB4-RB5 as inputs, RB7 as output.

```
bcf STATUS, RP0
clrf PORTB
bsf STATUS, RP0
movlw 70H
movwf TRISB
```

5:Timer modules in PIC Microcontroller

Overview of Timer Modules :**********

PIC 16C74A has three modules, viz., Timer-0, Timer-1 and Timer-2. Timer-0 and Timer-2 are 8-bit timers. Timer-1 is a 16-bit timer. Each timer module can generate an interrupt on timer overflow.

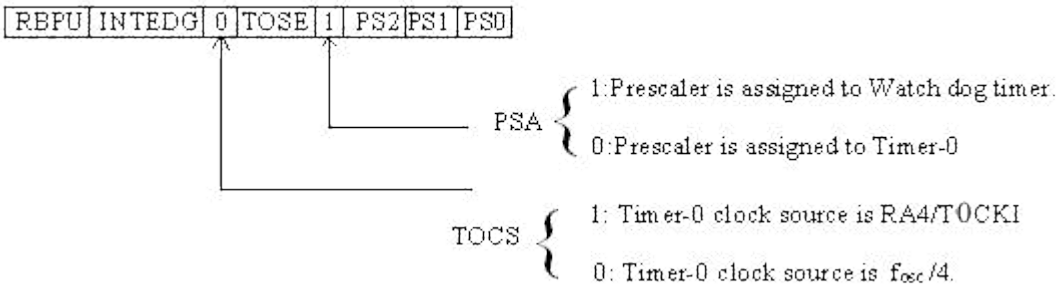
Timer-0 Overview:

The timer-0 module is a simple 8-bit UP counter. The clock source can be either the internal clock (f_{osc} /4) or an external clock. When the clock source is external, the Timer-0 module can be programmed to increment on either the rising or falling clock edge. Timer-0 module has a programmable pre-scaler option. This pre-scaler can be assigned either to Timer-0 or the Watch dog timer, but not to both.

The Timer-0 Counter sets a flag T0IF (Timer-0 Interrupt Flag) when it overflows and can cause an interrupt at that time if that interrupt source has been enabled, (T0IE = 1), i.e., timer-0 interrupt enable bit = 1.

OPTION Register Configuration :

Option Register (Addr: 81H) Controls the prescaler and Timer -0 clock source. The following OPTION register configuration is for clock source = f_{osc} /4 and no Watchdog timer.



Timer-0 use without pre-scalar

Internal clock source of f $_{osc}$ /4. (External clock source, if selected, can be applied at RA4/TOCKI input at PORTA).

The following diagram shows the timer use without the prescaler.

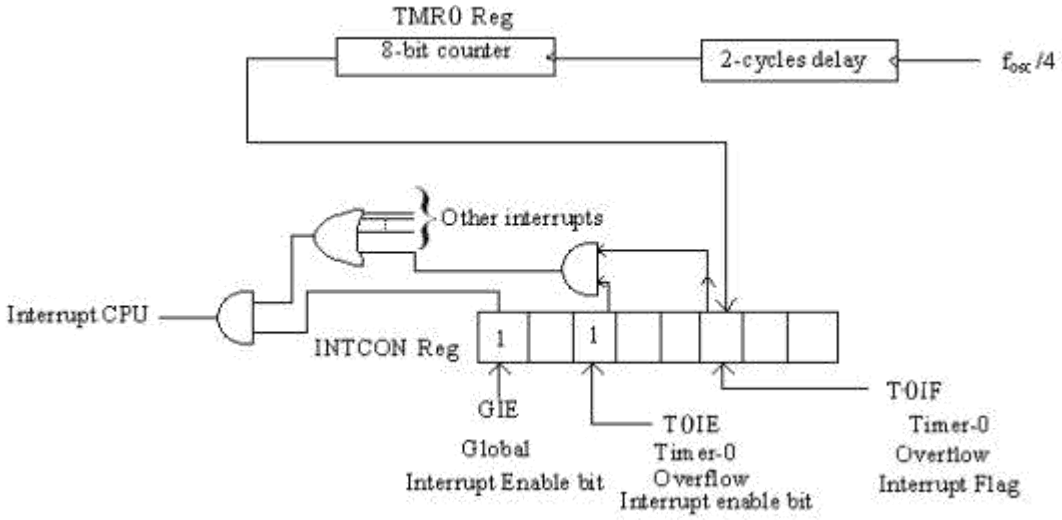


Fig 20.1 Timer - 0 operation without prescaler

Timer-0 use with pre-scalar:

The pre-scalar can be used either with the Timer-0 module or with the Watchdog timer. The pre-scalar is available for Timer-0 if the pre-scalar assignment bit PSA in the OPTION register is 0. Pre-scalar is a

programmable divide by n counter that divides the available clock by a pre-specified number before applying to the Timer-0 counter.

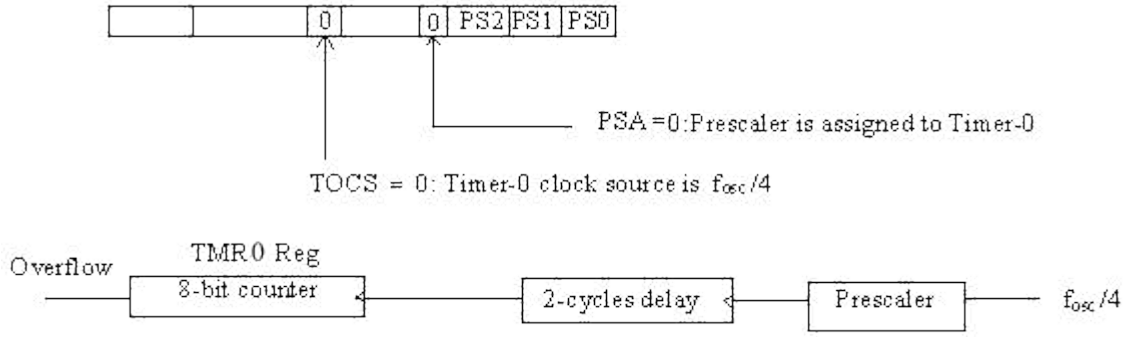


Fig 20.2 Timer - 0 with prescaler

Prescaler bits			Divide by N
PS2	PS1	PS0	
0	0	0	2
0	0	1	4
0	1	0	8
0	1	1	16
1	0	0	32
1	0	1	64
1	1	0	128
1	1	1	256

6 : Timer modules in PIC Microcontroller (contd.)

Timer - 1 Module

Timer 1 module is a 16-bit timer/counter consisting of two 8-bit registers (TMR1H and TMR1L) which are readable and writable. The TMR1 register pair (TMR1H:TMR1L) increments from 0000H to FFFFH and rolls over to 0000H. The TMR1 interrupt, if enabled, is generated on overflow, which sets the interrupt flag bit TMR1IF (bit-0 of PIR1 register). This interrupt can be enabled/disabled by setting/clearing TMR1 interrupt enable bit TMR1IE (bit-0 of the PIE1 register).

The operating and control modes of Timer1 are determined by the special purpose register T1CON.

Various bits of T1CON register are given as follows:-

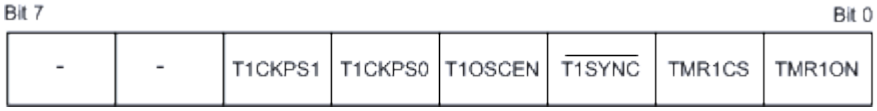


Fig 21.1 T1CON Register

TMR1 ON : Timer1 ON bit
0 = stops Timer 1; 1 = Enables Timer 1

TMR1CS : Timer 1 Clock source Select Bit
1 = External Clock (RCO/T1OSO/T1CKI)
0 = Internal Clock ($f_{osc}/4$)

$\overline{T1SYNC}$: Timer 1 External Clock Input Synchronization Bit
(Valid if TMR1CS = 1)
1 - Do not synchronize
0 - Synchronize

T1OSCEN: Oscillator enable control bit
1 = Oscillator is enabled
0 = Oscillator is shut off

Timer 1 Input Clock Prescaler

Select bits		Prescaler Value
T1CKPS1	**T1CKPS0**	
1	1	1:8
1	0	1:4
0	1	1:2
0	0	1:1

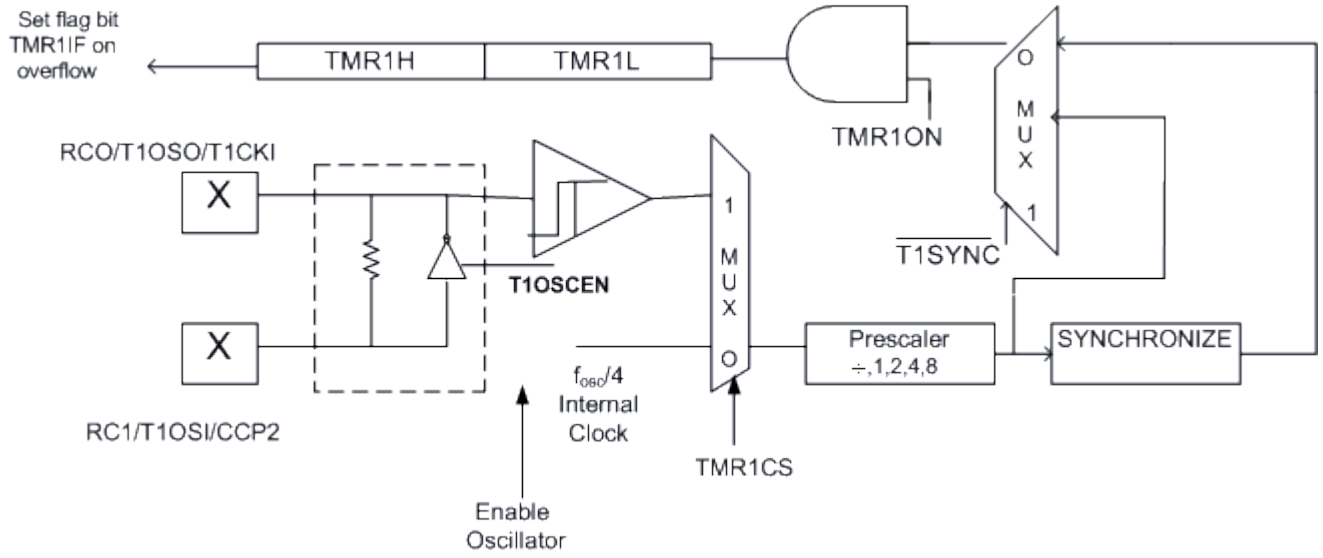


Fig 21.2 Operation of Timer 1

Timer 1 can operate in one of the two modes

- As a timer (TMR1CS = 0). In the timer mode, Timer 1 increments in every instruction cycle. The timer 1 clock source is $f_{osc}/4$. Since the internal clock is selected, the timer is always synchronized and there is no further need of synchronization.
- As a counter (TMR1CS = 1). In the counter mode, external clock input from the pin RCO/T1CKI is selected.

Reading and writing Timer 1

Reading TMR1H and TMR1L from Timer 1, when it is running from an external clock source, have to be done with care. Reading TMR1H or TMR1L for independent 8 - bit values does not pose any problem. When the 16-bit value of the Timer is required, the high byte (TMR1H) is read first followed by the low byte (THR1lL). It should be ensured that TMR1L does not overflow (that is goes from FFH to 00H) since THR1H was read. This condition is verified by reading TMR1H once again and comparing with previous value of TMR1H.

Example Program

Reading 16bit of free running Timer 1

```
movf TMR1H          ;          read high byte
movwf TMPH          ;          store in TMPH
movf TMR1L          ;          read low byte
movwf TMPL          ;          store in TMPL
movf TMR1H, W       ;          read high byte in W
subwf TMPH, W       ;          subtract 1 st read with 2 nd read
btfsc STATUS, Z     ;          and check for equality
goto next ;
;  if the high bytes differ, then there is an overflow
;  read the high byte again followed by the low byte
movf TMR1H, W       ;          read high byte
movwf TMPH
movf TMR1L, W       ;          read low byte
movwf TMPL
next : nop
```

Timer 2 Overview

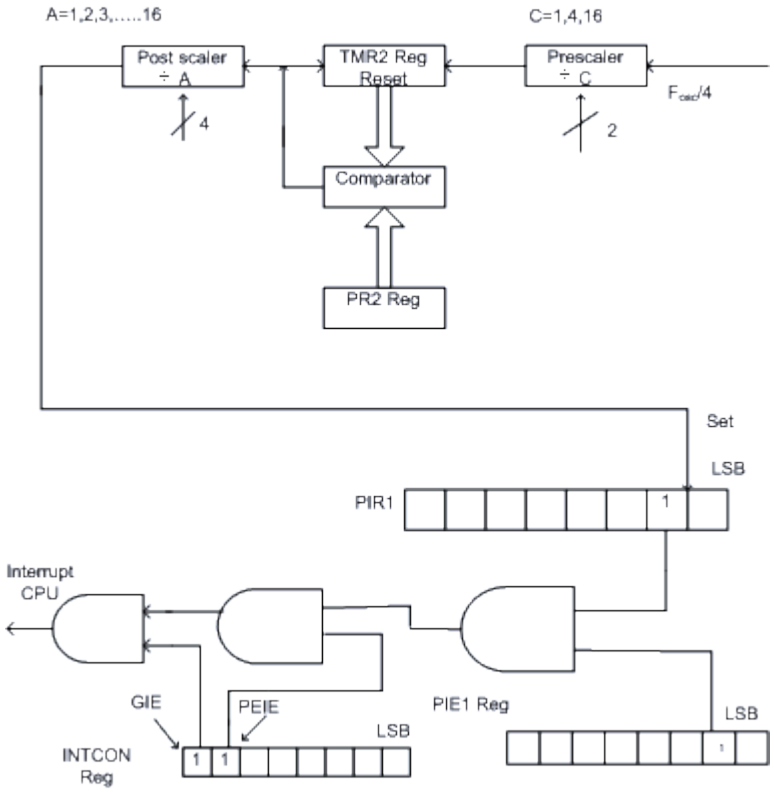


Fig 21.3 Schematic diagram showing operation of Timer 2

Timer 2 is an 8 - bit timer with a pre-scaler and a post-scaler. It can be used as the PWM time base for PWM mode of capture compare PWM (CCP) modules. The TMR2 register is readable and writable and is cleared on device reset.

The input clock ($f_{osc}/4$) has a pre-scaler option of 1:1, 1:4 or 1:16 which is selected by bit 0 and bit 1 of T2CON register respectively.

The Timer 2 module has an 8bit period register (PR2). Timer-2 increments from 00H until it is equal to PR2 and then resets to 00H on the next clock cycle. PR2 is a readable and writable register. PR2 is initailised to FFH on reset.

The output of TMR2 goes through a 4bit post-scaler (1:1, 1:2, to 1:16) to generate a TMR2 interrupt by setting TMR2IF.

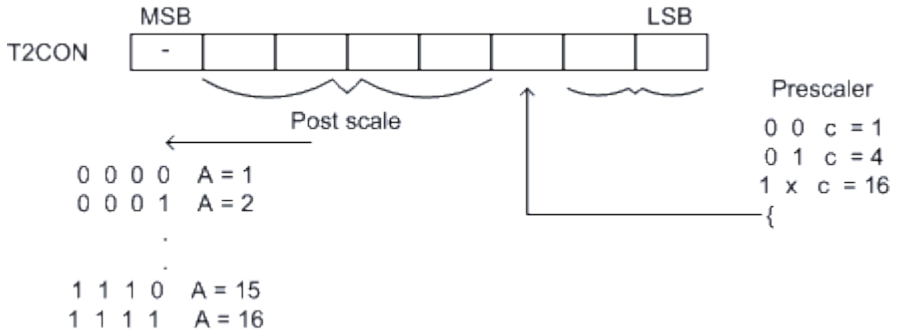


Fig 21.4 The T2CON Register

Interrupt Logic in PIC 16C74A

PIC 16C74A microcontroller has one vectored interrupt location (i.e., 0004H) but has 12 interrupt sources. There is no interrupt priority. Only one interrupt is served at a time. However interrupts can be masked. The interrupt logic is shown below :

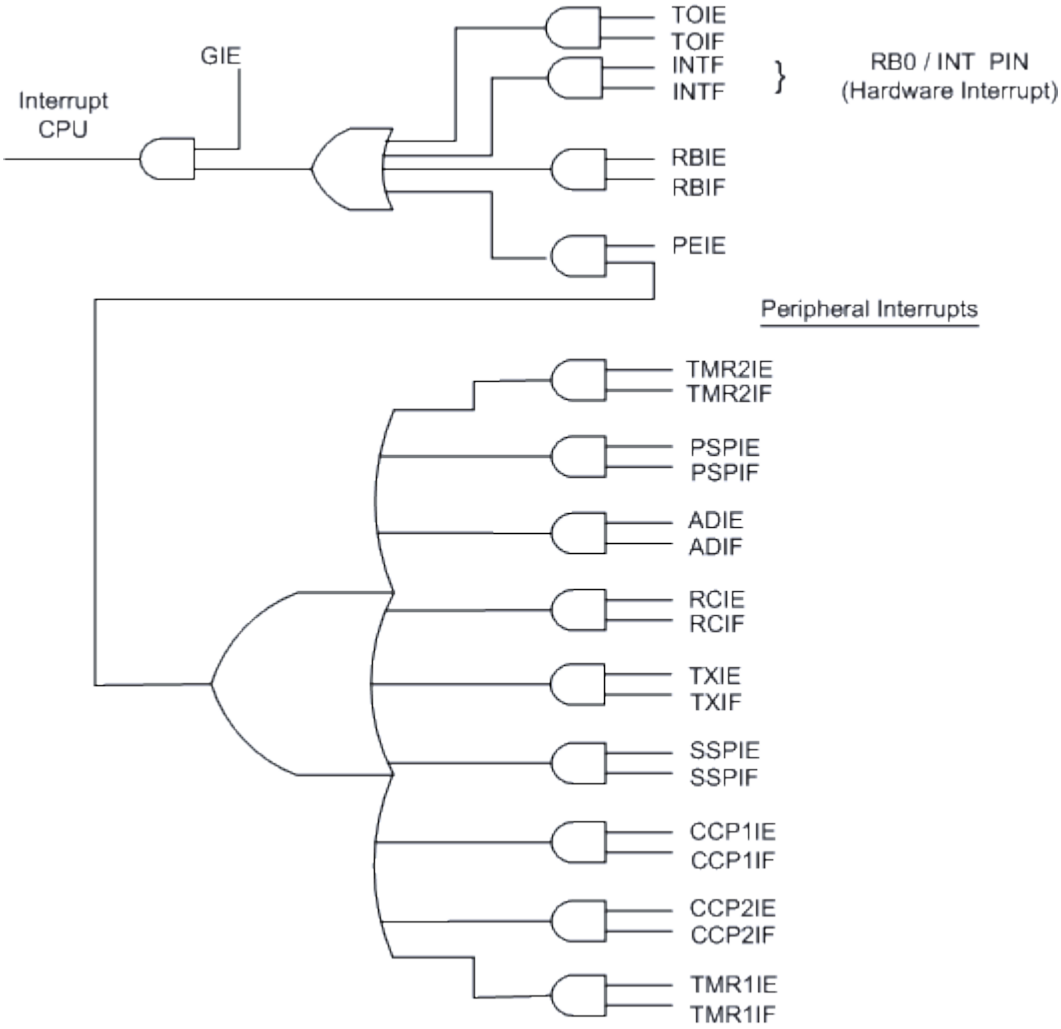


Fig 21.5 Schematic diagram showing the interrupt logic for PIC

7 : CCP Modules

Capture / Compare /PWM (CCP) Modules:**********

PIC16C74A has two CCP Modules. Each CCP module contains a 16 bit register (two 8-bit registers) and can operate in one of the three modes, viz., 16-bit capture, 16-bit compare, or up to 10-bit Pulse Width Modulation (PWM). The details of the two modules (CCP1 and CCp2) are given as follows.

CCP1 Module:

CCP1 Module consists of two 8-bit registers, viz., CCPR1L (low byte) and CCPR1H (high byte). The CCP1CON register controls the operation of CCP1 Module.

CCP2 Module:

CCP2 Module consists of two 8 bit registers, viz., CCPR2L (Low byte) and CCPR2H (high byte). The CCP1CON register controls the operation of CCP2 Module.

Both CCP1 and CCP2 modules are identical in operation with the exception of the operation of special event trigger.

The following table shows the timer resources for the CCP Mode.

CCP Mode	Timer Used
Capture	Timer 1
Compare	Timer 1
PWM	Timer 2

CCP1CON Register (Address 17H)

CCP2CON Register is exactly similar to CCP1CON register. CCP2CON Register address is 1DH. CCP1CON controls CCP module1 where as CCP2CON controls CCP Module2.

Bit 7 bit 0

—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0

Bit 5-4:

CCP1X CCP1Y: PWM least significant bits. These bits are of no use in Capture mode. In PWM Mode, these bits are the two Lsbs of the PWM duty cycle. The eight Msbs are found in CCPR1L. Thus the PWM mode operates in 10-bit mode.

Bit 3-0:

CCP1M3:CCP1MO (CCP1 Mode select bits)
0000=Capture/Compare/PWM Mode off
0100=Capture mode, every falling edge
0101=Capture mode, every rising edge
0110=Capture mode, every 4 th rising edge
0111=Capture mode, every 16 th rising edge
1000=Compare mode, set output on match (CCP1IF bit is set)
1001=Compare mode, clear output on match (CCP1IF bit is set)
1010=Compare mode, generate software interrupt on match (CCP1IF bit is set, CCP1 pin unaffected)
1011=Compare mode, trigger special event (CCP1IF bit is set;CCP1 resets Tmr1; CCP2 resets TMR1 and starts A/D conversion if A/D module is Enabled)
11XX=PWM mode.

Capture Mode (CCP1):

Capture Mode captures the 16-bit value of TMR1 into CCPR1H:CCPR1L register pair in response to an event occurring on RC2/CCP1 pin. Capture Mode for CCP2 is exactly similar to that of CCP1.

An event on RC2/CCP1 pin is defined as follows:

- Every falling edge
- Every rising edge.
- Every 4 th rising edge.
- Every 16 th rising edge.

As mentioned earlier, this event is decided by bit 3-0 of CCP1CON register.

Schematic diagram for capture mode of operation

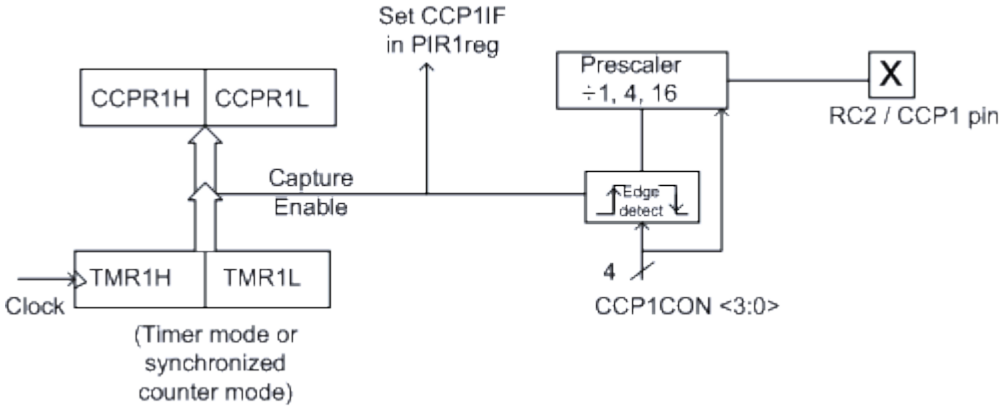


Fig 22.1 Capture operation

Required condition for capture mode:

1. RC2/CCP1 pin should be configured as an input by setting TRISC (bit 2).
2. Timer 1 should be operated from the internal clock (fosc/4), i.e., timer mode or in synchronized counter mode.

Compare Mode (CCP1)

Compare mode for CCP2 is similar to that of CCP1, except that in special event trigger mode, CCP1 resets TMR1 only, whereas CCP2 resets TMR1 and starts A/D conversion if A/D module is enabled.

In compare mode, the 16-bit CCPR1 register value is compared against TMR1 register pair (TMR1H and TMR1L) value. When a match occurs, the RC2/CCP1 pin is driven high or driven low or remains unchanged as decided by CCP1CON<3:0> bits.

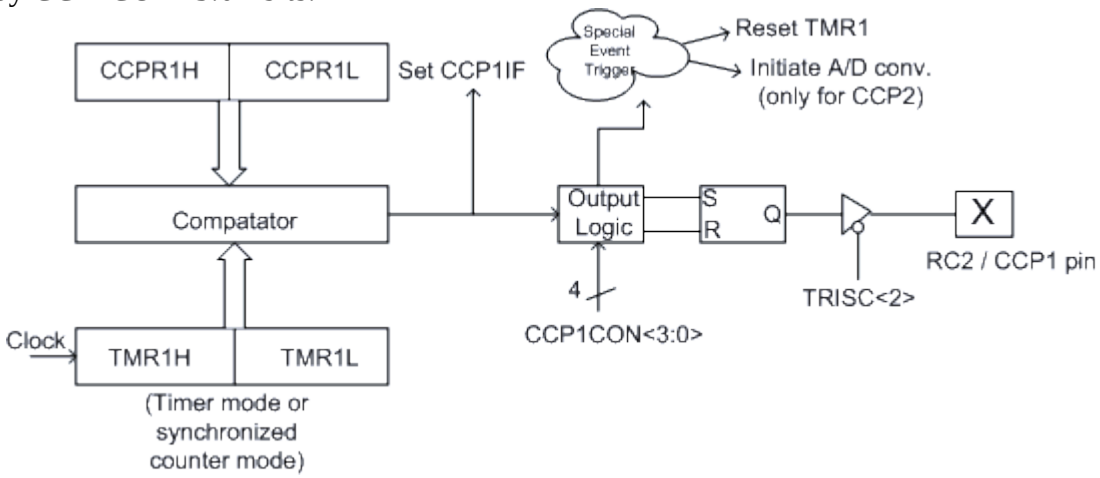


Fig 22.2 Compare Operation

Required conditions for compare mode

1. RC2/CCP1 pin must be configured as an output by clearing TRISC<2> bit.

2. Timer-1 should be operated in timer mode (i.e., internal clock source of $f_{osc}/4$) or in synchronized counter mode.

In software interrupt mode, CCP1IF bit is set but CCP1 pin in unaffected.

As shown in the figure, in special event trigger mode, both CCP1 and CCP2 intiates an A/D conversion.

PWM mode (CCP1)**********

Both CCP1 and CCP2 have similar operation in PWM mode. Here we will discuss PWM with respect to CCP1. In PWM mode, the CCP1 pin produces upto a 10-bit resolution Pulse Width Modulation (PWM) output. RC2/CCP1 pin should be configured in the uotput mode by clearing TRISC<2> bit. The schematic block diagram of CCP1 module in PWM mode is shown in the figure.

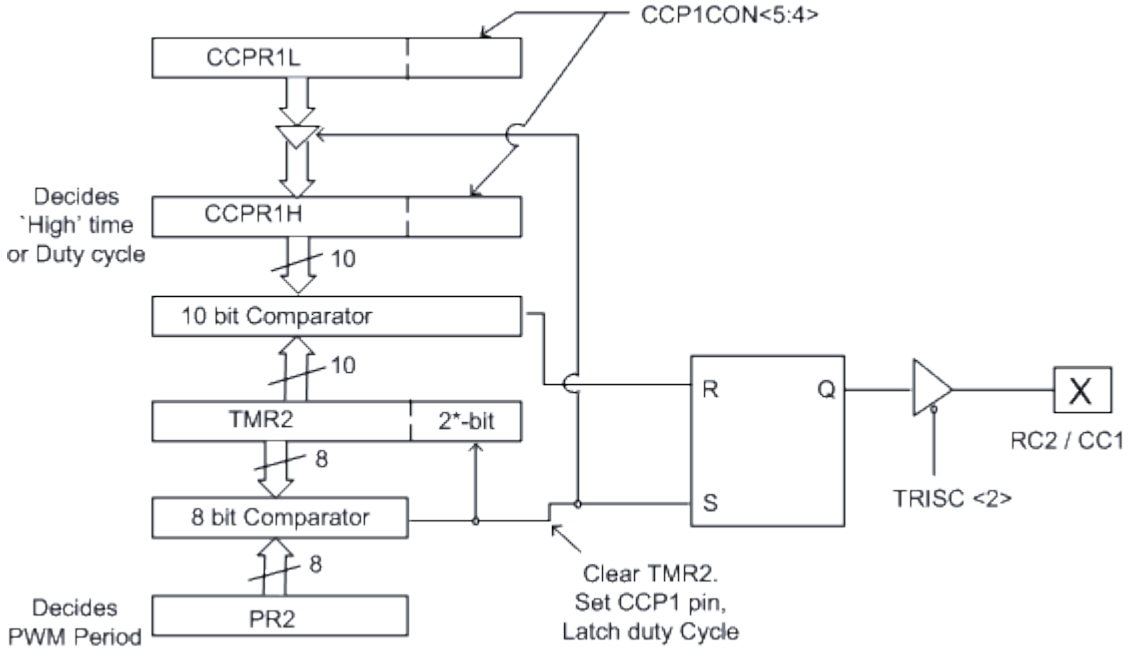
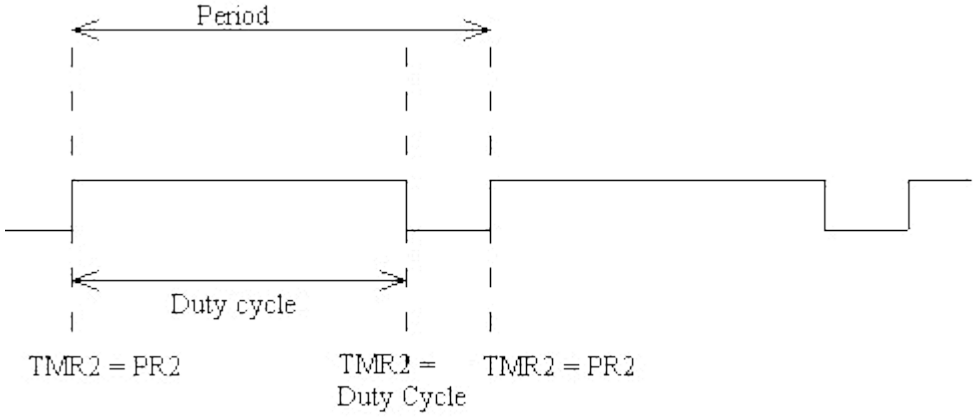


Fig 22.3 PWM Operation

It can be noted that PR2 (Period Register, 8 bit) decides the PWM period where CCPR1L (8-bits) and CCP1CON <5:4> (2-bits) decide the PWM duty cycle. When TMR2 equals PR2, the SR latch is set and RC2/CCP1 pin is pulled high. In the same time, TMR2 is cleared and the duty cycle value available in CCPR1L is latched to CCPR1H. CCPR1H, CCP1CON <5:4> decide the duty cycle and when this 10-bit ewquals the TMR2+2 prescaler or Q-bits, the SR latch is set and RC2/CCP1 pin is driven low.



A PWM output as shown has a time period. The time for which the output stays high is called duty cycle.

PWM Period

The PWM period is specified by writing to PR2 register. The PWM period can be calculated using the following formula:
PWM period = [(*PR* 2) + 1] × 4 × *T osc* × (TMR2 prescale value)
PWM frequency = 1/ PWM period
When TMR2 is equal to PR2, the following events occur on the next increment cycle.

- TMR2 is cleared
- the CCP1 pin is set (if PWM duty cycle is 0
- The PWM duty cycle is latched from CCPR1L into CCPR1H

PWM duty cycle

The PWM duty cycle is specified by writing to the CCPR1L register and to CCP1CON < 5 : 4 >

bits. Up to 10-bit resolution is available where CCPR1L contains the eight MSBs and CCP1CON < 5 : 4 > contains the two LSB's. The 10-bit value is represented by CCPR1L : CCP1CON < 5 : 4 >.
The PWM duty cycle is given by
PWM duty cycle = (CCPR1L : CCP1CON < 5 : 4 >). *T osc* . (TMR2 prescale value)
To understand the 10-bit counter configuration from Timer-2, let us first see the counting mechanism of Timer-2, as shown in Fig 22.4.

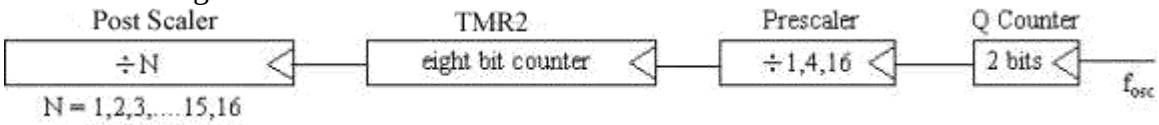


Fig 22.4 Counting mechanism in Timer - 2

If the prescaler is 1, the 10-bit counter is configured as follows

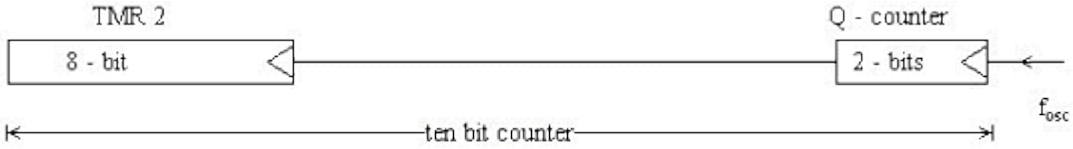


Fig 22.5 Prescaler set to divide by one

If the prescaler is 4, the 10-bit counter is configured as follows.

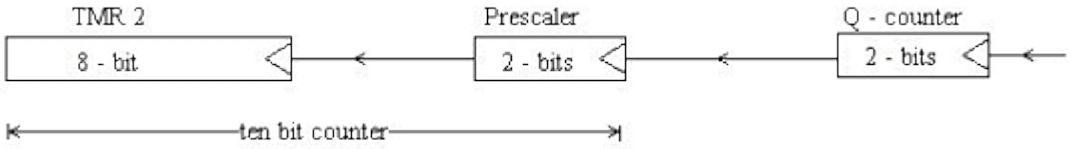


Fig 22.6 Prescaler programed to divide by four

If the prescaler is 16, the 10-bit counter is realized as follows.

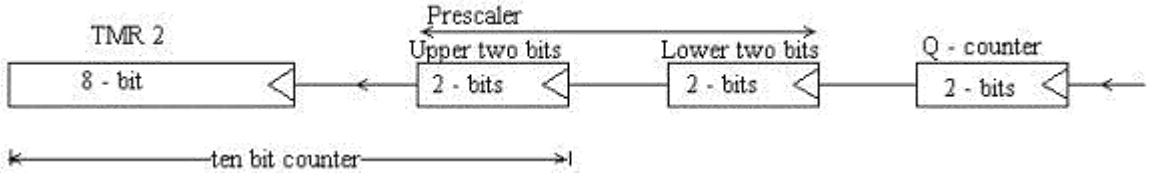


Fig 22.7 Prescaler programed to divide by 16

Although CCPR1L and CCP1CON < 5 : 4 > can be written to at anytime, the duty cycle value is not latched into CCPR1H until a match between PR2 and TMR2 occurs. In PWM mode, CCPR1H is a read-only register.

The CCPR1H register and a 2-bit internal latch are used to double buffer the PWM duty cycle. This double buffering is essential for glitchless PWM operation. When the CCPR1H and 2-bit latch match TMR2

concatenated with an internal 2-bit Q clock or 2-bits of prescaler, the CCP1 pin is cleared. Maximum PWM resolution (bits) for a given PWM frequency can be calculated as

$$\frac{\log\left(\frac{f_{osc}}{f_{PWM}}\right)}{\log 2}$$

If the PWM duty cycle is longer than the PWM period, then the CCP1 pin will not be cleared.

PWM Period and duty cycle calculation

Example:

Desired PWM frequency = 78.125 kHz
f_{osc} = 20MHz
TMR2 Prescalar = 1

$$\frac{1}{78.125\times10^{3}} = (PR2+1)4\times\frac{1}{20\times10^{6}}$$

$$PR2 = 63$$

Find the maximum resolution of duty cycle that can be used with a 78.124 kHz frequency and 20 MHz oscillator.

$$\frac{1}{78.125\times10^{3}} = 2^{PWMResolution}\cdot\frac{1}{20\times10^{6}}\cdot 1$$

$256 = 2^{PWM}$ Resolution

At most, an 8-bit resolution duty cycle can be obtained from a 78.125 kHz frequency and 20 MHz oscillator ie, 0 CCPR1L : CCP1CON <5 : 4> ≤ 255 .

Any value greater than 255 will result in a 100 % duty cycle. The following table gives the PWM frequency f_{PWM} if f_{osc} = 20MHz

Duty cycle resolution	10-Bit counter scale	PR2 value	Prescaler 1	Prescaler 4	Prescaler 16
10 bit	1024	255	19.53 KHz	4.88 kHz	1.22 kHz
≈ 10 bit	1000	249	20kHz	5kHz	1.25kHz
8 bit	256	63	78.125kHz	19.53kHz	4.88kHz
6 bit	64	15	312.5kHz	78.125kHz	19.53kHz

8 : Analog to Digital Convertor Module

ADC Module**********

An analog-to-digital converter (ADC) converts an analog signal into an equivalent digital number. PIC 16C74A has an inbuilt ADC with the following features -

- 8-bit conversion
- 8 analog input channels
- An analog multiplexer
- A sample and hold circuit for signal on the selected input channel
- Alternative clock sources for carrying out conversion
- Adjustable sampling rate
- Choice of an internal or external reference voltage
- Interrupt to microcontroller on end of conversion

Port A and Port E pins are used for analog inputs/reference voltage for ADC. In A/D conversion, the input analog voltage is digitized and an equivalent digital output is generated as shown in the figure.

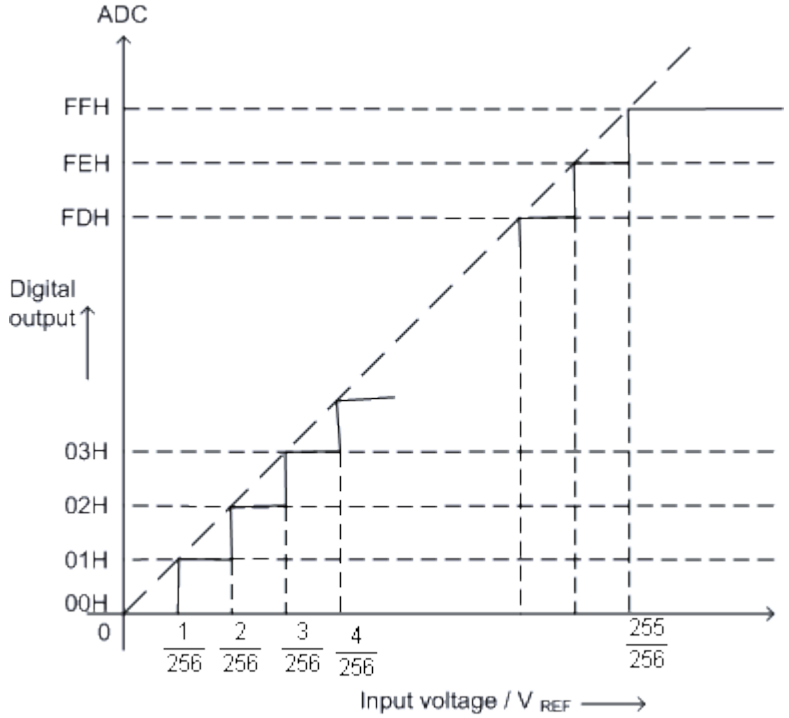


Fig 23.1 Digital output versus analog input

Port-A pins (Alternate functions)

RA0/AN0 - can be used as analog input-0
RA1/AN1 - can be used as analog input-1
RA2/AN2 - can be used as analog input-2
RA3/AN3/V_{ref} - can be used as analog input-3 or analog reference voltage
RA4/TOCKI - clock input to Timer-0
RA5/ /AN4 - can be used for analog input 4 or slave select for the synchronized serial port.

Port-E pins (Alternate functions)

RE0/ $\overline{RD}$ /AN5 - can be used as analog input-5
RE1/ $\overline{WR}$/AN6 - can be used as analog input-6
RE2/ $\overline{CS}$ /AN7 - can be used as analog input-7

PIC microcontroller has internal sample and hold circuit. The input signal should be stable across the capacitor before the conversion is initiated.

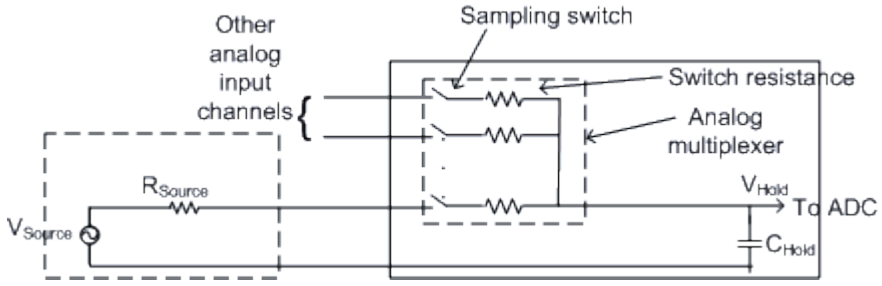


Fig 23.2 Sample and Hold Circuit

After waiting for the sampling time, a conversion can be initiated. The ADC Circuit will open the sampling switch and carry out the conversion of the input voltage as it was at the moment of opening of the switch. Upon completion of the conversion, the sampling switch is again closed and V_{Hold} once again tracks V_{Source}.

Using the A/D Converter

Registers ADCON1, TRISA, and TRISE must be initialized to select the reference voltage and input channels. The first step selects the ADC clock from among the four choices ($f_{osc}/2$, $f_{osc}/8$, $f_{osc}/32$, and RC). The constraint for selcting clock frequency is that the ADC clock period must be 1.6micro seconds or greater.

The A/D module has 3registers. These registers are:-

- A/D result register (ADRES)
- A/D control register 0 (ADCON 0)
- A/D control register 1 (ADCON 1)

The ADCON0 register, which is shown below, controls the operation of A/D module.

7	6	5	4	3	2	1	0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{DONE}$	-	ADON

Fig 23.3 ADCON0 register

Bit 7-6 - A/D Clock select bits
ADCS1:ADCS0
00 = $f_{osc}/2$
01 = $f_{osc}/8$
10 = $f_{osc}/32$
11 = f_{RC}- clock derived from an internal RC oscillator

Bit 5-3 - A/D Channel Select
CHS2:CHS0
000 - Channel 0 - AN0
001 - Channel 1 - AN1
010 - Channel 2 - AN2
011 - Channel 3 - AN3
100 - Channel 4 - AN4
101 - Channel 5 - AN5
110 - Channel 6 - AN6
111 - Channel 7 - AN7

Bit 2 - A/D conversion status bit

GO / $\overline{DONE}$

if A/D Converter is enabled (ie. ADON = 1) then

If GO / $\overline{DONE}$ = 1, A/D conversion is in progress
(setting this bit starts A/D conversion)

If GO / $\overline{DONE}$ = 0, A/D conversion is not in progress
(This bit is automatically cleared by hardware when A/D conversion is complete)

Bit1 - Unimplemented

Bit 0 - ADON: A/D On bit

1. A/D Converter module is ON
2. A/D Converter module is OFF

ADCON1 Register

This register specifies the analog inputs

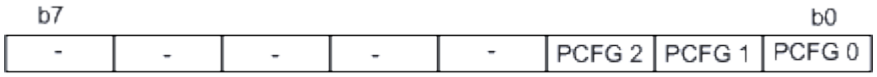


Fig 23.4 ADCON1 register

PCFG2:PCFG0	RA0	RA1	RA2	RA5	RA3	RE0	RE1	RE2	V_{REF}
000	A	A	A	A	A	A	A	A	V_{DD}
001	A	A	A	A	V_{REF}	A	A	A	RA_3
010	A	A	A	A	A	D	D	D	V_{DD}
011	A	A	A	A	V_{REF}	D	D	D	RA_3
100	A	A	D	D	A	D	D	D	V_{DD}
101	A	A	D	D	V_{REF}	D	D	D	RA_3
11X	D	D	D	D	D	D	D	D	-

Fig 23.5 PCFG2:PCFG0 = A/D Port configuration control bits

A = Analog input
D = Digital I/O

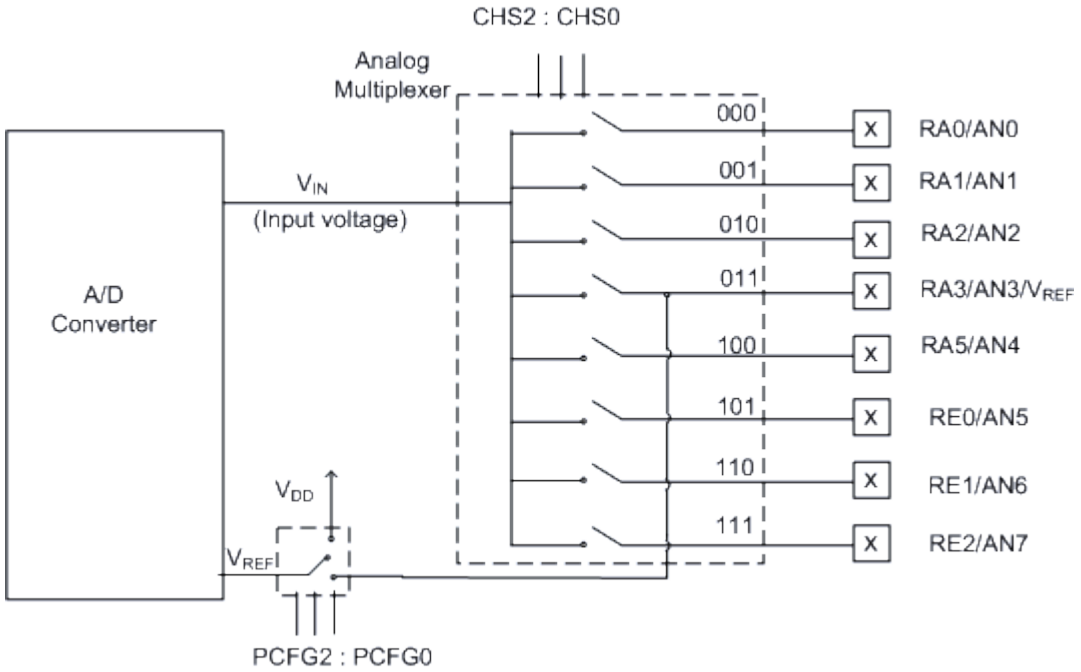


Fig 23.6 Schematic diagram of A/D convertor analog inputs and reference voltage

Steps for A/D conversion

1. Configure A/D module
   - Configure analog inputs/voltage reference and digital I/O (ADCON1)
   - Select A/D Channel (ADCON0)
   - Select A/D Conversion Clock (ADCON0)
   - Turn on A/D Module (ADCON0)

2. Configure A/D Interrupt (Optional)
    - Clear ADIF bit in PIR1 register
    - Set ADIE bit in PIE1 register
    - Set GIE bit
3. Wait for required acquisition time
4. Start Conversion - set GO/ $\overline{DONE}$ bit (ADCON0)
5. Wait for A/D conversion to complete, by either polling GO/ $\overline{DONE}$ bit or by waiting for the A/D interrupt
6. Read A/D result registers (ADRES). Clear ADIF if required.

Example Program

```
A/D conversion with interrupt
org 000H
goto Mainline
org 020H
bsf   STATUS, RP0              ;        Select Bank 1
clrf   ADCON 1                  ;        Configure A/D inputs
bsf   PIE1, ADIE                ;        Enable A/D interrupt
bcf   STATUS, RP0               ;        Select Bank 0
movlw   081H                    ;        Select fosc/32, channel 0, A/D on
movwf   ADCON0
bcf       PIR1,   ADIF
bsf       INTCON,   PEIE        ;        Enable peripheral and global interrupt bits
bsf       INTCON,   GIE         ;        interrupt bits
; Ensure that the required sampling time of the selected input channel has been elapsed.
; Then conversion may be started.
; bsf ADCON0, GO            ;  Start A/D conversion.
                            ;  ADIF bit will be set and GO/ DONE
                            ;  bit is cleared upon completion of A/D conversion.
```

Interrupt Service Routine

```
Org 004H
Movf ADRES, W ; Result of A/D conversion in W
```

Consideration of Sampling Time

When a channel is selected (writing to ADCON0), the switch 'SW' in Fig 23.8 is closed, changing C_{HOLD} to V_{Source} . When A/D conversion is started (setting Go bit in ADCON0), SW is opened. The time from the closure of 'SW' till the voltage across C_{HOLD} (V_o) reaches V_{Source} is the minimum sampling time T_s .
The actual sampling time can be higher than T_s .
The graph between T_s and source resistance R_{Source} is shown in Fig 23.7.+

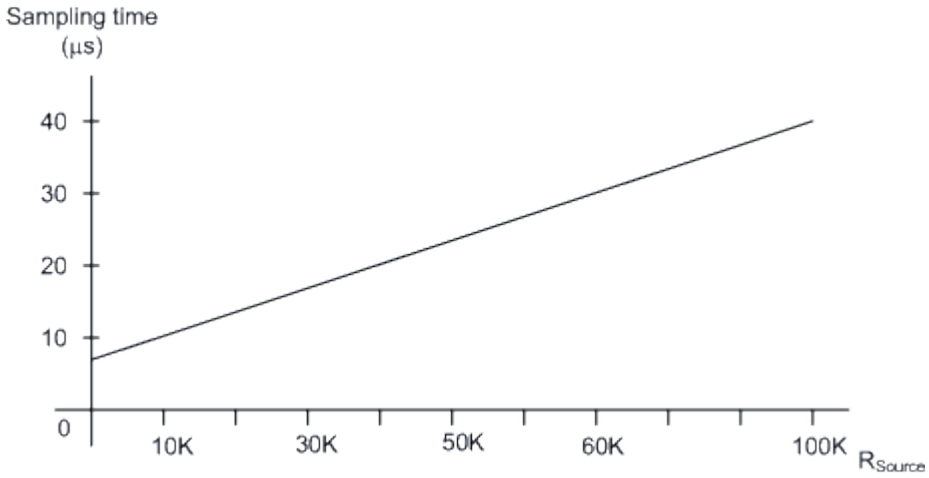


Fig 23.7 Relation between sampling time and source resistance

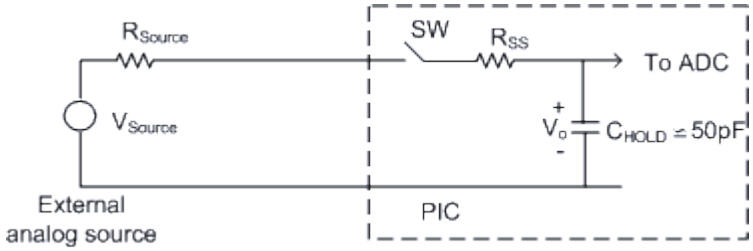


Fig 23.8 Sampling circuit in the PIC μC

R_{ss} is the resistance of the sampling switch 'SW' and C_{Hold} is the charge holding capacitance. C_{Hold} is nearly 50pF. R_{Source} is the impedance of the external analog source Vsource. Once the switch 'SW' is closed, the capacitor Chold takes some time to charge up. This time it is called the sampling time (T_s). This time varies linearly with R_{Source} as shown. The recommended value of impedance of the external analog source, Vsource, is less than 10kΩ.

The circuit in Fig 23.8 is a first order RC circuit. When SW is closed, V_o varies as shown in Fig 23.9.

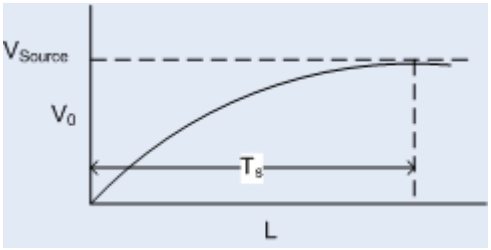


Fig 23.9

From Fig 23.9,

$T_s = 5\,\tau = 5\,(R_{source} + R_{ss})\,C_{HOLD}$

$= 5\,R_{ss}\,C_{HOLD} + 5\,R_{source}\,C_{HOLD}$

Hence sampling time T_s varies linearly with R_{Source} as shown in Fig 23.7.

9 : Synchronous Serial Port (SSP) Module:**********

Most of mid range PIC microcontrollers include a Synchronous Serial Port (SSP) Module. The discussion in this section is relevant to PIC16C74A only. SSP Module section can be configured in either of the following two modes.

- Serial Peripheral Interface (SPI)
- Inter Integrated Circuit (I^2C)

Either of these modes can be used to interconnect two or more PIC chips to each other using a minimal number of wires for communication. Alternatively, either can be used to connect a PIC microcontroller to a peripheral chip. When I^2C mode is selected, the peripheral chip must also have an I^2C interface. On the other hand, the SPI mode provides the clock and serial data lines for direct connection to shift registers. This leads to increased I/O interface capability and an arbitrary number of I/O devices can be connected to a PIC microcontroller. SPI can also achieve data rate significantly higher than I^2C. Both the communication methods are synchronous, i.e., the data transfer is synchronized with an explicit clock signal.

Two special purpose registers control the synchronous serial port (SSP) operations. These registers are:

- SSPCON (Synchronous Serial Port Control Register), Address: 14H
- SSPSTAT(Synchronous Serial Port status Register), Address: 94H

Serial Peripheral Interface (SPI)

Port-C three pins, viz., RC5/SDO, RC4/SDI and RC3/SCK/SCL are mainly used for SPI mode. In addition, one Port-A pin, viz., RA5/$\overline{SS}$ /AN4 is used for slave select. The schematic block diagram of SPI is shown in the figure

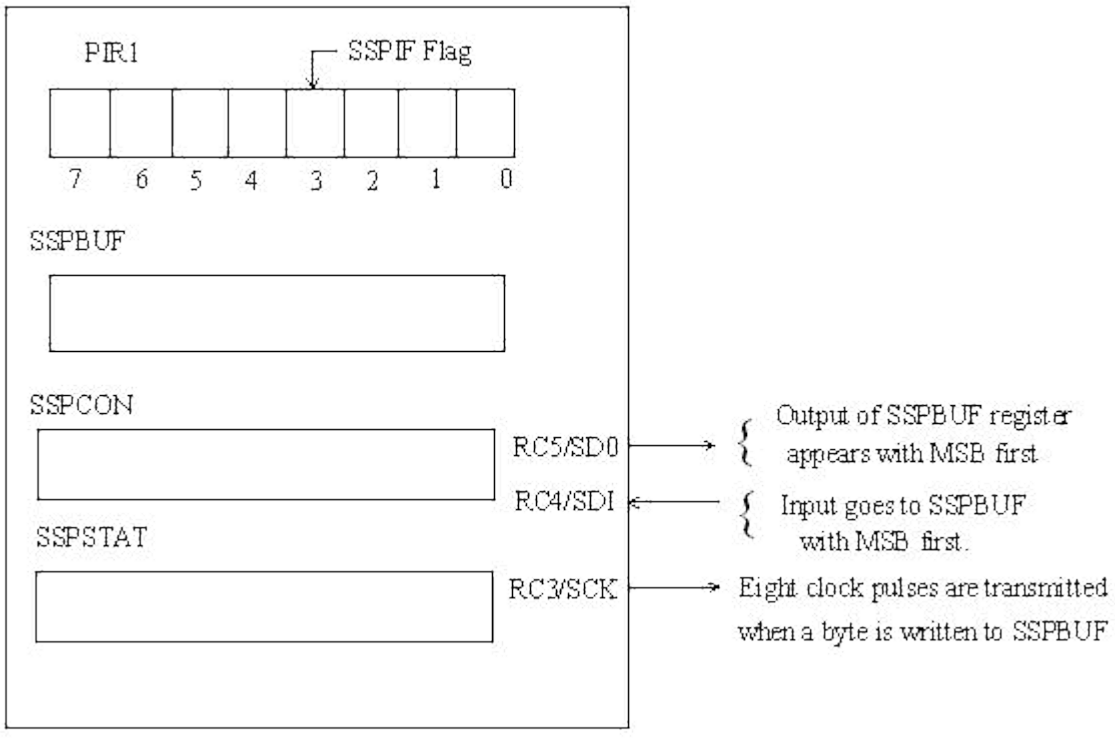


Fig 24.1 Schematic diagram under SPI Mode

The SPI port requires RC3/SCK pin to be an output that generates the clock signal used by the external shift registers. When SPI is configured in the slave mode, RC3/SCK pin works as the input for the clock.

When a byte is written to SSPBUF register, it is shifted out of RC5/SDO pin in synchronous with the emitted clock pulses on RC3/SCK pin. The MSB of SSPBUF is the first bit to appear on RC5/SDO pin. Simultaneously, the same write to SSPBUF also initiates the 8 bit data reception into SSPBUF of whatever appears on RC4/SDI

pin at the time of rising edges of the clock on SCK pin. Hence shifting-in and shifting-out of data occur simultaneously.

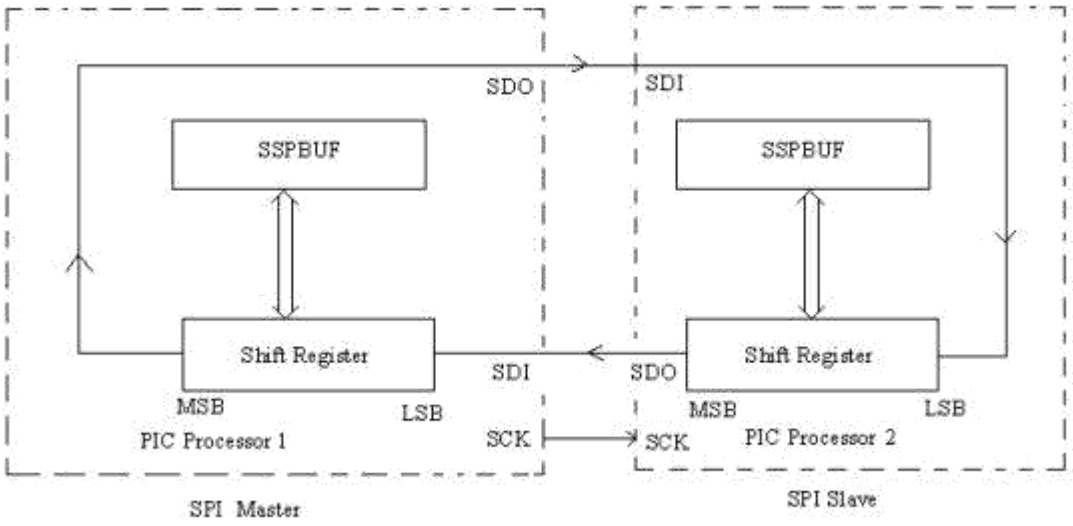
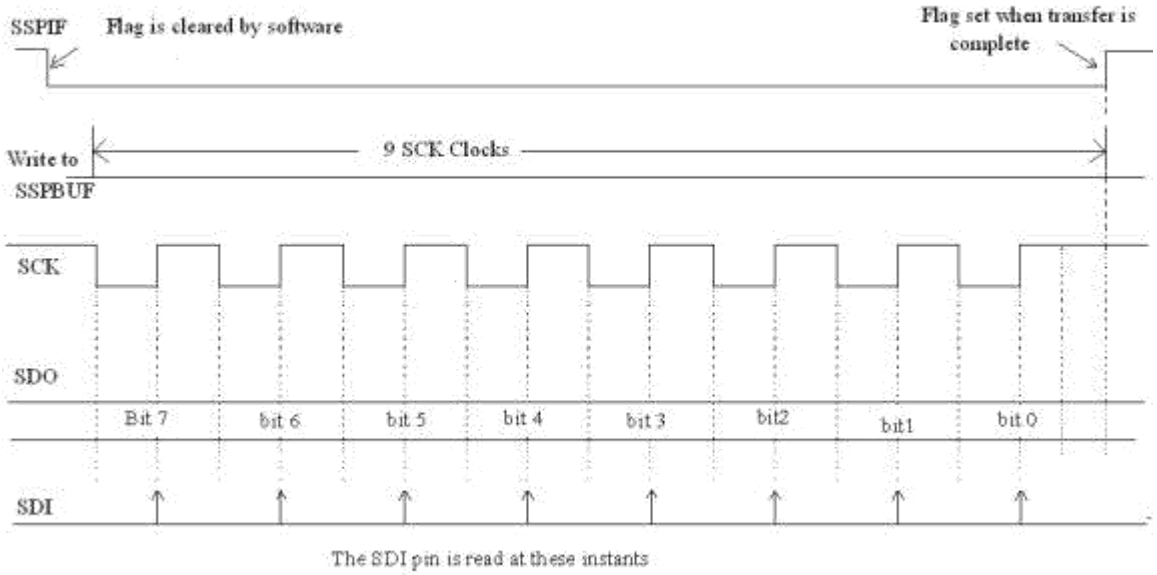


Fig 24.2 SPI Master / Slave Connection

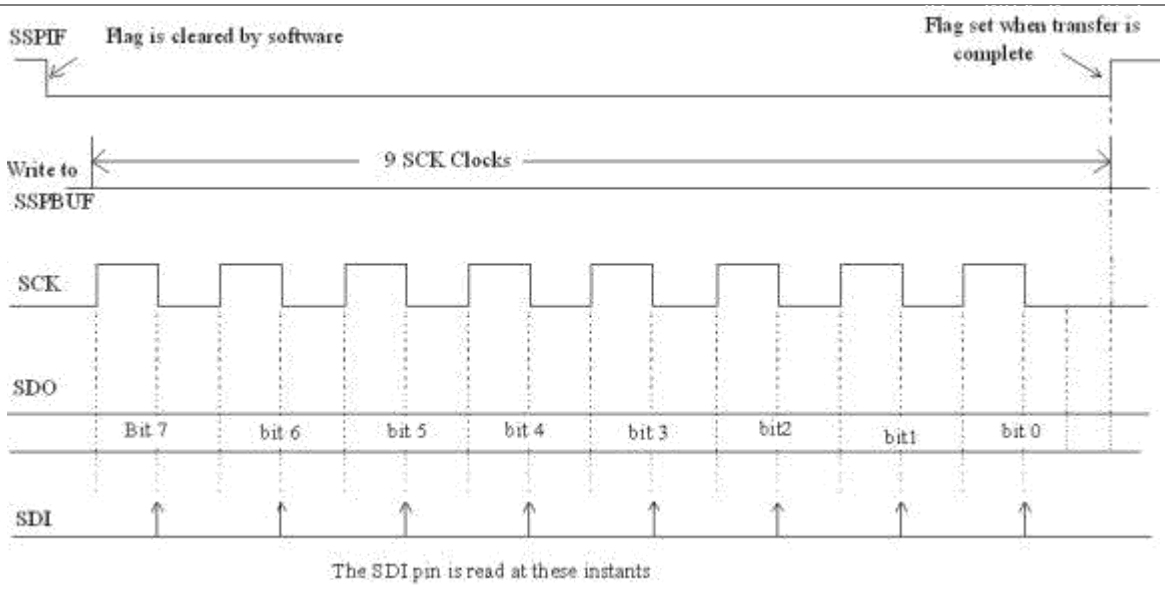
The schematic diagram of SPI Master/Slave connection is shown in the figure.

Timing diagram for data transfer in 'Master mode' :

SSPIF interrupt flag is cleared by the user software if already in the set mode. The interrupt is enabled. Any write to SSPBUF initiates the data transfer, i.e., transmission and reception. The clock pulses (8 clock pulses) are output through SCK pin. The data is received through SDI. When CKP=1 (SSPCON<4>), data changes at SDO at negative clock transition and is read through SDI at positive clock transition. The idle state of clock is high. If CKP=0, data appears at SDO at positive clock transition and is read through SDI at negative clock transition. The idle state of the clock is low. These are shown in the following diagrams.



(i) Timing diagram for CKP=1



(ii) Timing diagram for CKP=0

Fig 24.3 Timing Diagram under SPI mode

10 : I/O Port Expansion using Serial Peripheral Interface (SPI)

Though SPI is a serial communication interface, it can be used to realize multiple output parallel ports and multiple input parallel ports. We will consider this realization of an output parallel port and an input port separately.

Parallel Output Port Realization

A parallel 8-bit output port can be realized through SPI with the help of a shift register chip (74HC595) as shown in Fig 25.1. RC5/SD0 pin outputs serial data while RC3/SCK oin outputs the serial clock. Since input data transfer is not required, port pin RC4/SDI is used to latch the shift register data to the output pins of the shift register. Hence RC4 is configured as an output pin.

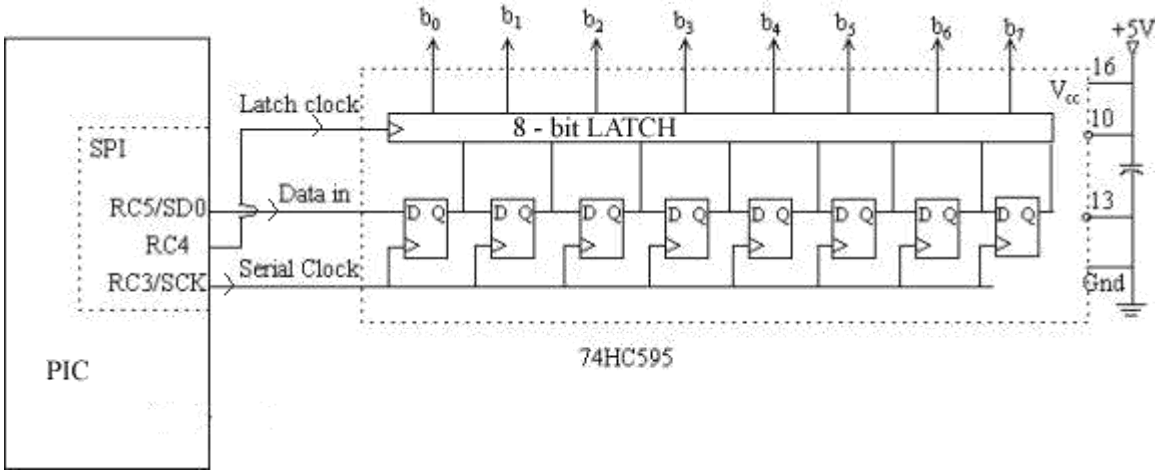


Fig 25.1 PIC connection (in SPI mode) with a shift register

When an 8-bit data is written to SSPBUF, the data is shifted out of RC5/SD0 pin. With CKP = 1, the data is stable at the positive transition but changes at the negative transition. The shift shifts the data at the positive clock transition. After 8 clock pulses, all 8-bits are shifted in the shift register. The completion of data transfer is indicated by SSPIF interrupt flag becoming ' 1' . The interrupt service routine make RC4 ' 1' , thus latching the 8-bit data to the output of the shift register. The configuration of various registers are shown in Fig 25.2

Port configurations

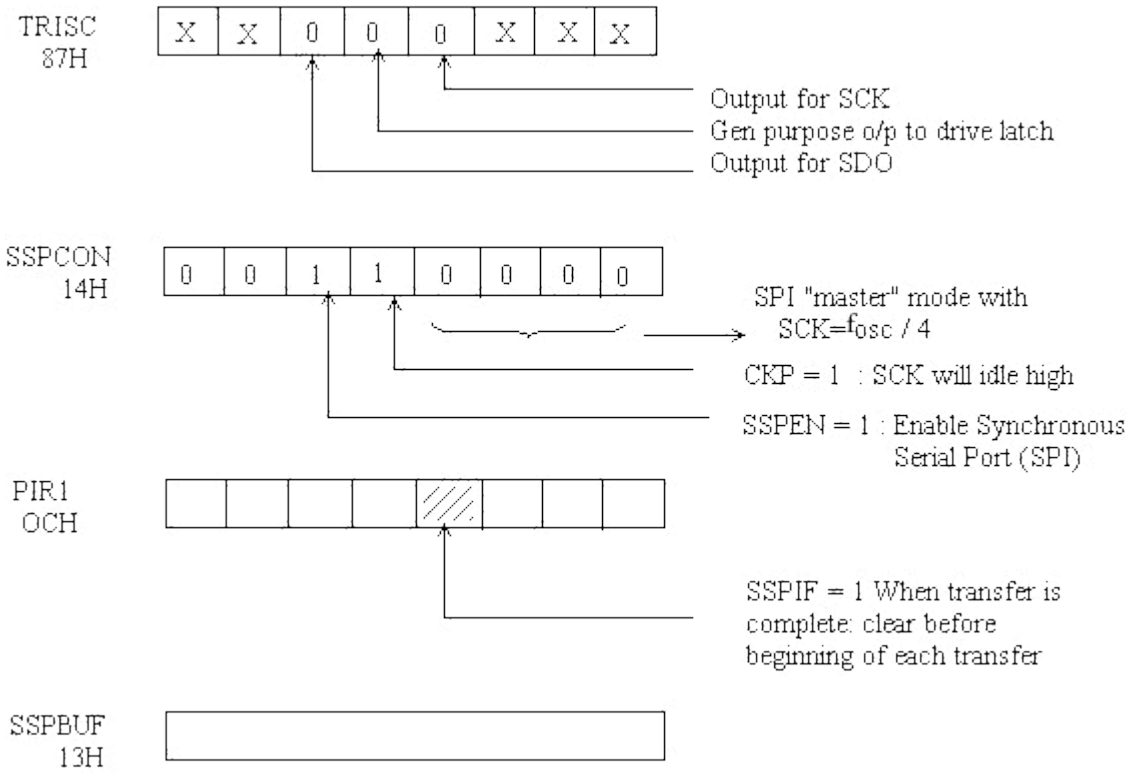
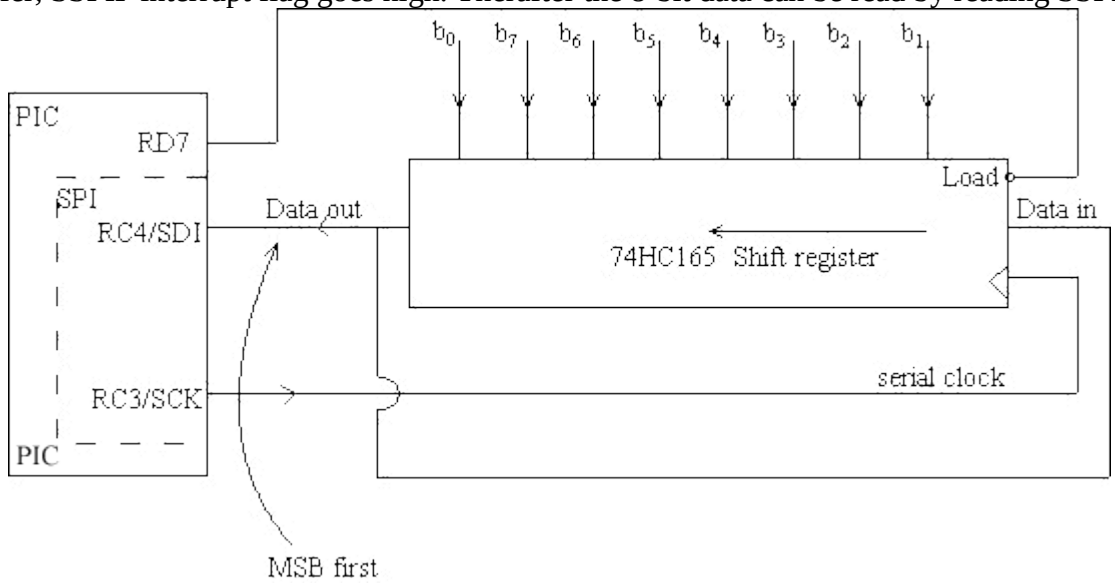
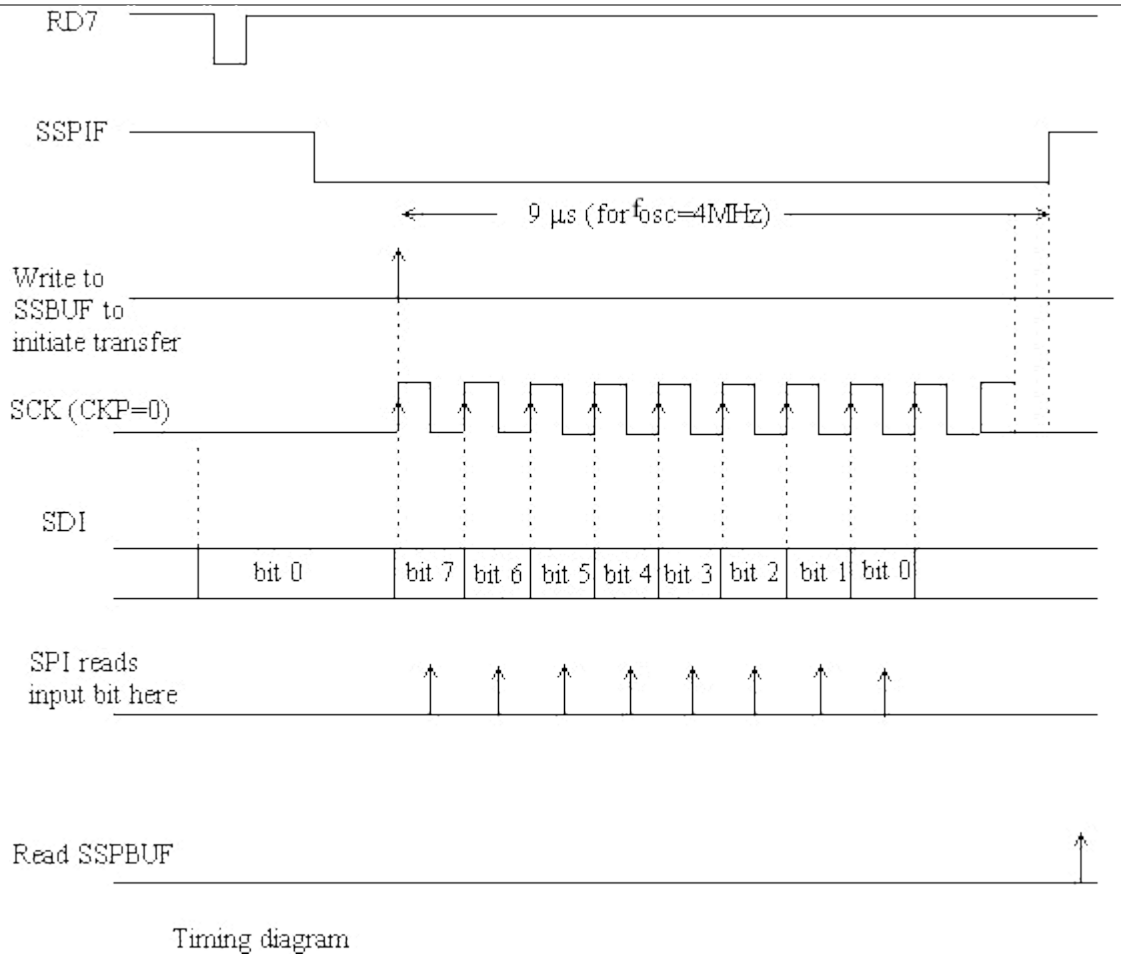


Fig 25.2 Various Register Configurations

Parallel Input Port Realization

A shift register (74HC165) is connected to the PIC microcontroller as shown in Fig 25.3. Pin RD7 is configured as an output and is used to load 8-bit data to the shift register. A dummy write to SSPBUF initiates data transfer. Data bit is read into RC4/SDI at the negative clock transition (CKP = 0) where the data bit is stable. Data is shifted in the shift register at the position clock transition as shown in the timing diagram. After the completion of data transfer, SSPIF interrupt flag goes high. Therafter the 8-bit data can be read by reading SSPBUF.





Timing diagram

Fig 25.3 Realization of an 8-bit parallel input port with PIC in SPI mode.

Port configurations

Fig 25.4 gives the configurations various registers for inputs parallel port realization.

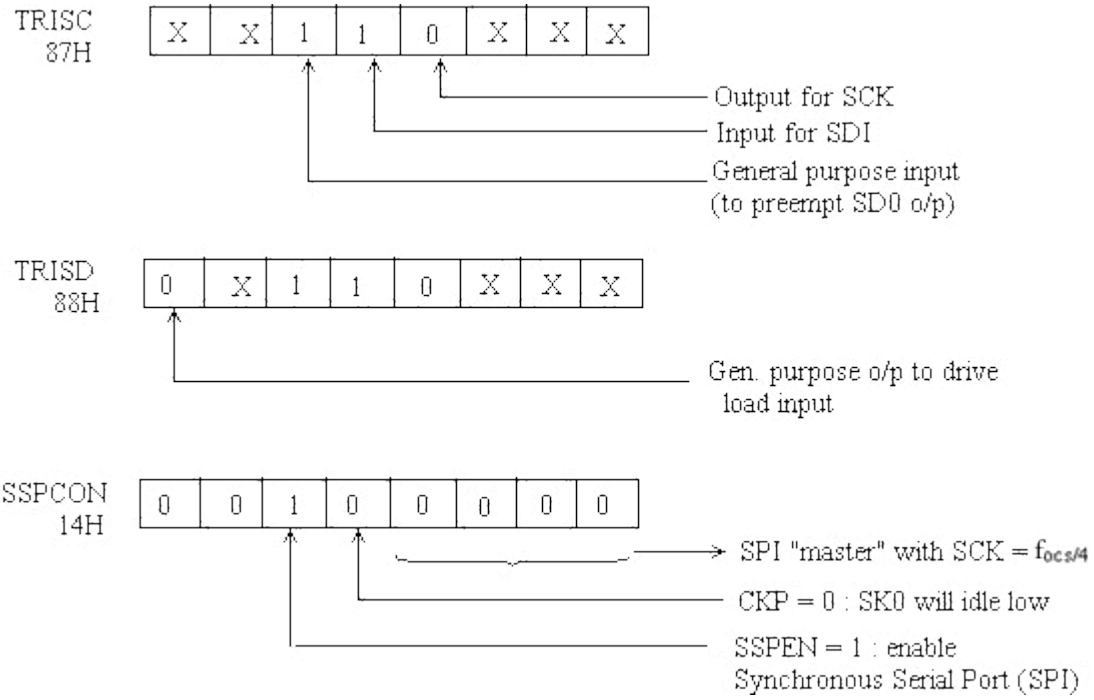


Fig 25.4 Configurations of various registers for parallel input port

11 : I ^{2}C Communication in PIC Microcontroller:********

I^{2}C Communication in PIC Microcontroller

I ^{2}C stands for Inter-Integrated circuit. I ^{2}C communication is a two wire bi-directional interface for connecting one or more master processors with one or more slave devices, such as an EEPROM, ADC, RAM, LCD display, DAC, etc. I ^{2}C interface requires two open drain I/O pins, viz. SDA (Serial Data) and SCL (Serial Clock).

The reason for open drain connection is that the data transfer is bi-directional and any of the devices connected to the I ^{2}C bus can drive the data line (SDA). The serial clock line (SCL) is usually driven by the master. Since SDA and SCL pins are open drain pins, external pull-up resistances are required for operation of I ^{2}C bus.

A typical I^{2}C bus showing the connection of multi-master and multi-slave configuration is shown in the following figure.

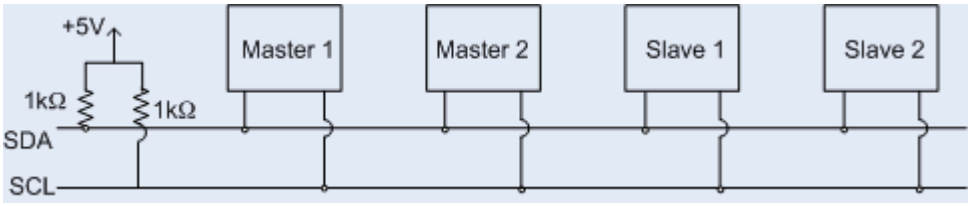


Fig 26.1 Multimaster Multislave Connection

Some conventions are followed in I^{2}C communication. Let us assume that there is one master and one slave and 8-data bits are sent. We will initially assume that the master is the transmitter and the slave is the receiver. The clock is driven by the master. On receiving 8-bits, an acknowledgement bit is driven by the receiver on SDA line. The acknowledgement bit is usually Low (0). The following diagram shows the data communication pattern having 8 data bits and one acknowledgement bit.

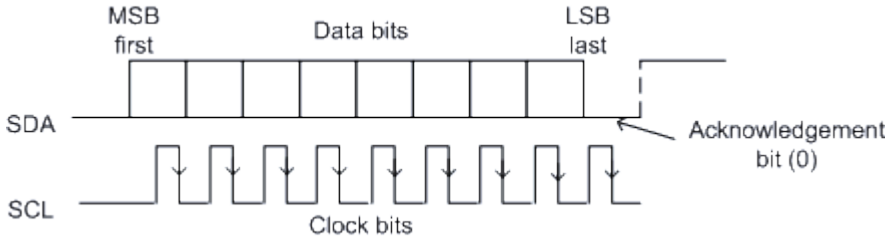


Fig 26.2 Timing diagram for data transfer

The following features are to be noted -

1. SDA line transmits/ receives data bits. MSB is sent first.
2. Data in SDA line is stable during clock (SCL) high. A new bit is initiated at the negative clock transition after a specified hold time.
3. Serial clock (SCL) is driven by the master.
4. An acknowledgement bit (0) is driven by the receiver after the end of reception. If the receiver does not acknowledge, SDA line remains high (1).

I^{2}C bus transfer consists of a number of byte transfers within a START condition and either another START condition or a STOP condition. During the idle state when no data transfer is taking place, both SDA and SCL lines are released by all the devices and remains high. When a master wants to initiate a data transfer, it pulls SDA low followed by SCL being pulled low. This is called START condition. Similarly, when the processor wants to terminate the data transfer it first releases SCL (SCL becomes high) and then SDA. This is called a STOP condition. START and STOP conditions are shown in the diagram as follows.

Fig 26.3 Timing diagram for START and STOP Conditions

START and STOP conditions are unique and they never happen within a data transfer.

Data Communication Protocol:

In I^2C communication both 7-bit and 10-bit slave addressing are possible. In 7-bit addressing mode 128 slaves can be interfaced with a single master. Similarly, in 10-bit addressing mode, 1024 slaves can be interfaced with the master. We will discuss here 7-bit addressing mode only. 10-bit addressing mode is similar to 7-bit addressing except from the fact that the number of address bits is more.

Following a 'start' condition, the master sends a 7-bit address of the slave on SDA line. The MSB is sent first. After sending 7-bit address of the slave peripheral, a R/$\overline{W}$ (8^{th} bit) bit is sent by the master. If R/$\overline{W}$ bit is '0', the following byte (after the acknowledgement bit) is written by the master to the addressed slave peripheral. If R/$\overline{W}$=1, the following byte (after the acknowledgement bit) has to be read from the slave by the master.

After sending the 7-bit address of the slave, the master sends the address (usually 8 bit) of the internal register of the slave wherefrom the data has to be read or written to. The subsequent access is automatically directed to the next address of the internal register.

The following diagrams give the general format to write and read from several peripheral internal registers.

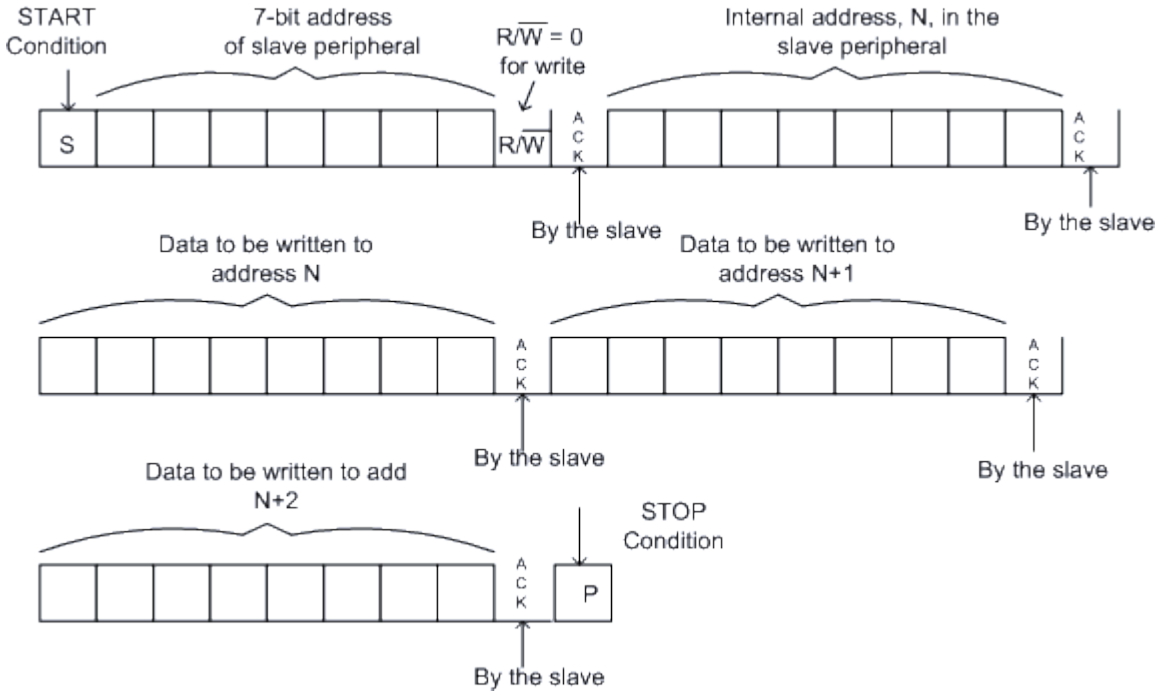


Fig 26.4 Data transfer protocol for writing to a slave device

R/$\overline{W}$ (Read / Write) bit indicates whether the data is to be written by the master or read by the master. If R/$\overline{W}$ is 1, the subsequent data are to be read by the master. If R/$\overline{W}$ = 0, the subsequent data are to be written by the master to the addressed slave. It has to be noted that the slave address is sent first, following a 'start' condition. The addressed slave responds by acknowledging and gets ready for data transfer.

If data has to be read from a specific address of the slave device, the master sends the 7-bit address of the slave first following a 'start' condition. R/$\overline{W}$ bit is sent as 'low'. The addressed slave acknowledges by pulling the ACK line low. The master then sends the 8-bit internal address of the slave from which data has to be read. The slave acknowledges. Since R/$\overline{W}$ bit was initially 0, the master is in the write mode. To change this to read

mode, the 'start' condition is again generated followed by 7-bit address of the slave with R/$\overline{W}$ = 1. The slave acknowledges. The slave then sends data from previously specified internal address to the master. The master acknowledges by pulling ACK bit low. The data transfer stops when the master does not acknowledge the data reception and a 'stop' condition is generated.

12 : Software for I ^{2}C Communication

Software for I^2 C Communication

The data transfer in I^{2}C mode is not automatically controlled by hardware unlike UART. The Master has to be programmmed by suitable software to generate 'Start' / 'Stop' conditions, various data bits from sending / receving , acknowledgement bit and clock signal. Here, we will discuss some examples of I^{2}C software.
Since SDA (RC4) and SCL (RC3) are both open drain pins, they can be configured either as an output or as an input. When a PIC Processor is configured is I^{2}C master, the SCL pin will function as open drain output while the SDA pin can be either an input or an open drain output. Hence, the software I ^{2}C will repeatedly access TRISC, the data direction register for PORT C. However, TRISC is located in bank-1 at an address 87H, which cannot be accessed by direct addressing without changing RP0 bit to 1 as given in the following instruction.

bsf STATUS, RP0

Then required bit of TRISC can be changed followed by clearing RP0 and reverting back to Bank-0.

bcf STATUS, RP0

Alternately, the indirect pointer FSR can have the address of TRISC and the required bit setting and bit clearing can be done indirectly.

Consider the following definitions.

```
SCL equ 3
SDA equ 4
```

The instruction bsf INDF, SDA will release the SDA line(as RC4/SDA pin is configured as an input, hence tristated), letting the external pullup resister pull it high or some I ^{2}C Slave device/Chip pull it low.
When FSR is used for indirect addressing, care should be taken to restore FSR value when subroutine is completed and the program returns to the main line program.

I ^{2} C Subroutine

```
SDA equ 4
SCL equ 3
```

The following subroutine DATA_OUT transfers out three bytes, i.e., ADDRDEV, ADDR8, and DATAWRTE

```
DATA_OUT:       call START          ; Generate start condition
                movf ADDRDEV, W     ; Sends 7-bit peripheral address with R/ W =0
```

```
                call TRBYTE          ; Transmit
                movf ADDR8, W        ; Send 8-bit internal address
                call TRBYTE
                movf DATAWRTE, W  ; Send data to be written
                call TRBYTE
                call STOP            ; Generate Stop condition
                return
```

The DATA_IN subroutine, which is given below transfers out ADDRDEV (with R/ $\overline{W}$ =0) and ADDR8, restarts and transfers out ADDRDEV (with R/ $\overline{W}$ =1) and read one byte back into RAM variable DATARD.

```
DATA_IN:        call START
                movf ADDRDEV, W      ;  Send 7-bit peripheral address R/ W =0
                call TRBYTE
                movf ADDR8, W        ;  Send int. address
                call TRBYTE
                call START1          ; Restart
                movf ADDRDEV, W      ;  Send 7-bit peripheral address R/ W =1
                iorlwl  01H
                call TRBYTE
                bsf  TRBUF, 7        ; Generate NO ACK
                call RCVBYTE
                movwf DATARD
                call STOP
                return
```

The 'START' subroutine initializes I^2C bus and then generates START condition on the I^2C bus. START1 bypasses the initialization of I^2C.

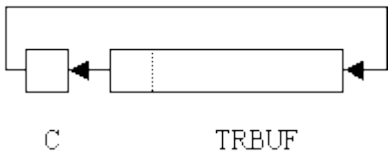
```
START:          movlw 3BH            ;enables I2C  master mode by programming SSPCON
                movwf SSPCON
                bcf PORTC, SDA       ; drive SDA low when it is an o/p
                movlw TRISC          ;set indirect pointer to TRISC
                movwf FSR
START1:
                bsf INDF, SDA        ; SDA=1
                bsf INDF , SCL       ; SCL=1
                call DELAY           ; Generates a suitable delay
                bcf INDF, SDA        ; SDA=0
                call DELAY           ; Generate a suitable delay
                bcf INDF, SCL        ;SCL=0
                return
STOP:
                bcf INDF, SDA        ;SDA=0
                bsf INDF, SCL        ; SCL=1
                call DELAY           ; Generate a suitable delay
                bsf INDF, SDA        ;SDA=1
                return
```

The subroutine 'TRBYTE' send out the byte available in w. It returns with Z=1 if ACK occurs. It returns with Z=0 if NOACK occurs.

TRBUF is an 8-bit RAM variable used for temporary storage. The bits are shifted to carry flag (C) and the carry bit transmitted successively. Data transfer is complete when all 8-bits are transmitted. Setting C = 1 initially sets an index for 8-bits to be transferred. C is rotated through TRBUF. After transmitting C, C-bit is cleared. When TRBUF is completely cleared, all 8-bis are transmitted.

```
TRBYTE:
```

```
movwf    TRBUF
bsf      STATUS,C
TR_1:
```

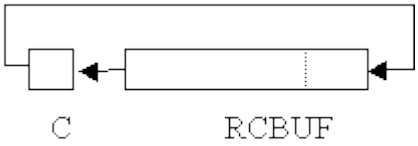



```
rlf      TRBUF, F
movf     RBUF,F
btfss    STATUS, Z
call     out_bit          ; Send a bit available in C
btfss    STATUS, Z
goto     TR_1
call     in_bit           ; Get the ACK bit in RCBUF<0>
movlw    01H              ;
andwf    RCBUF, W         ; Store the complement of ACK bit in Z flag
return
```

The RCVBYTE subroutine receives a byte from I^2 C into W using a RAM variable RCBUF buffer.

Call RCVBYTE with bit 7 of TRBUF clear for ACK
Call RCVBYTE with bit 7 of TRBUF set for NOACK
RCBUF is an 8-bit RAM variable used for recieving the data. the bit is recieved in the RCBUF<0> and is rotated successively through RCBUF as shown. The reception ends when all 8-bits are recieved.

```
RCVBYTE:
movlw    01H
movwf    RCBUF            ; Keep an index for 8-bits to be recieved.
```




```
RCV_1:
rlf      RCBUF, F
call     In_bit
btfss    STATUS, C
goto     RCV_1
rlf      TRBUF, F
call     Out_bit
movf     RCBUF,w
return
```

The out_bit subroutine transmits carry bit, then clears the carry bit.

```
Out_bit:
bcf      INDF, SDA
btfsc    STATUS, C
bsf      INDF, SDA        ; Send carry bit
bsf      INDF, SCL
call     DELAY
bcf      INDF, SCL
bcf      STATUS,C         ; Clear carry bit
return
```

The in_bit subroutine receives one bit into bit-0 of RCBUF.

In_bit:

```
bsf     INDF,SDA
bsf     INDF, SCL
bcf     RCBUF, 0
btfsc   PORTC, SDA        ; Check SDA line for data bit
bsf     RCBUF, 0
bcf     INDF, SCL
return
```

Example of I ^{2} C interfacing

DAC interfacing on I ^{2} C bus:

MAX518 is a dual 8-bit Digital to Analog Converter (DAC) with I^2C interface. The address of the device is selectable through two pins AD_1 and AD_0 . This device works in I^2C slave mode. The connection diagram is shown as follows.

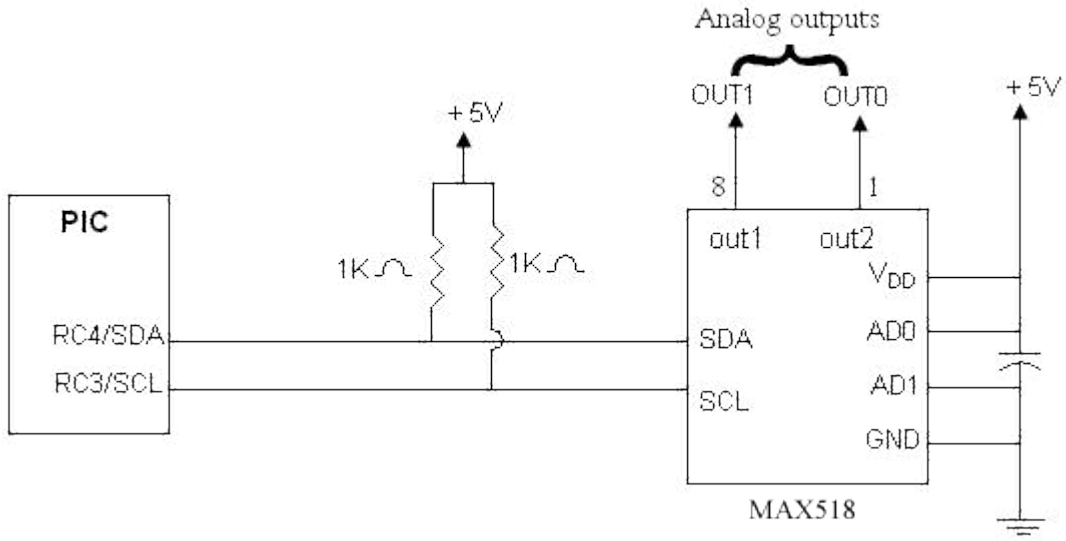


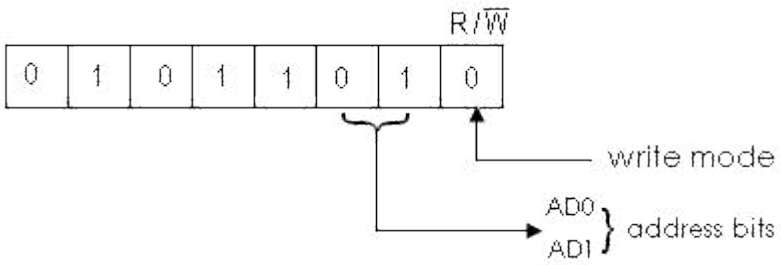
Fig 27.1 I^2C Interface for DAC

The 7-bit device address is given as

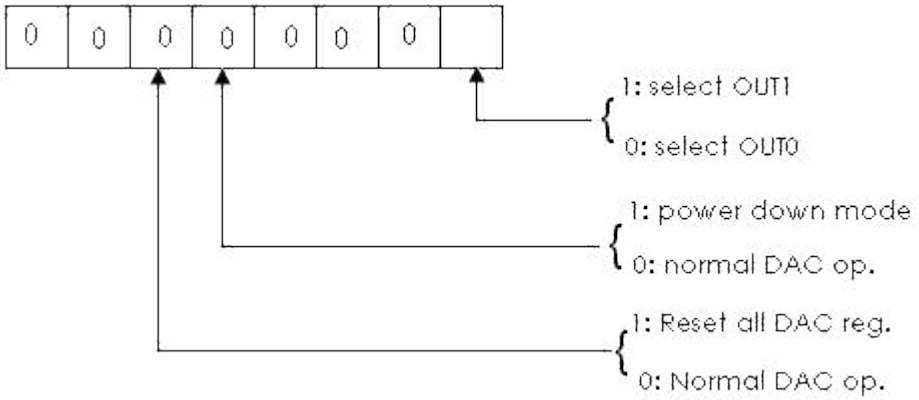
0	1	0	1	1	AD1	AD0

For the present connection $AD_1 = 0$ and $AD_0 = 1$
The device address is 010 1101
Three bytes are sent to output an analog voltage.

First byte (Address of the DAC and R/$\overline{W}$ bit)



Second byte (DAC Configuration)



Third byte (The 8-bit digital data(B) to be converted to analog voltage)
B

Analog output voltage = V_{DD} x B/256

13 : Parallel slave port

Parallel slave port (PSP)

PIC Microcontroller offers a mechanism by which an 8-bit parallel bidirectional data transfer can be achieved between a PIC Microcontroller and a PC. PIC Microcontroller's Port-D and Port-E are used in this data transfer. For this data transfer, Port-D of PIC Microcontroller is configured as a Parallel slave Port (PSP) by setting bit-4 of TRISE Register. The pins of Port-E function as control pins ($\overline{RD}$, $\overline{WR}$ and $\overline{CS}$) for data transfer. The connection diagram between PC and the PIC Microcontroller in PSP Mode is shown below.

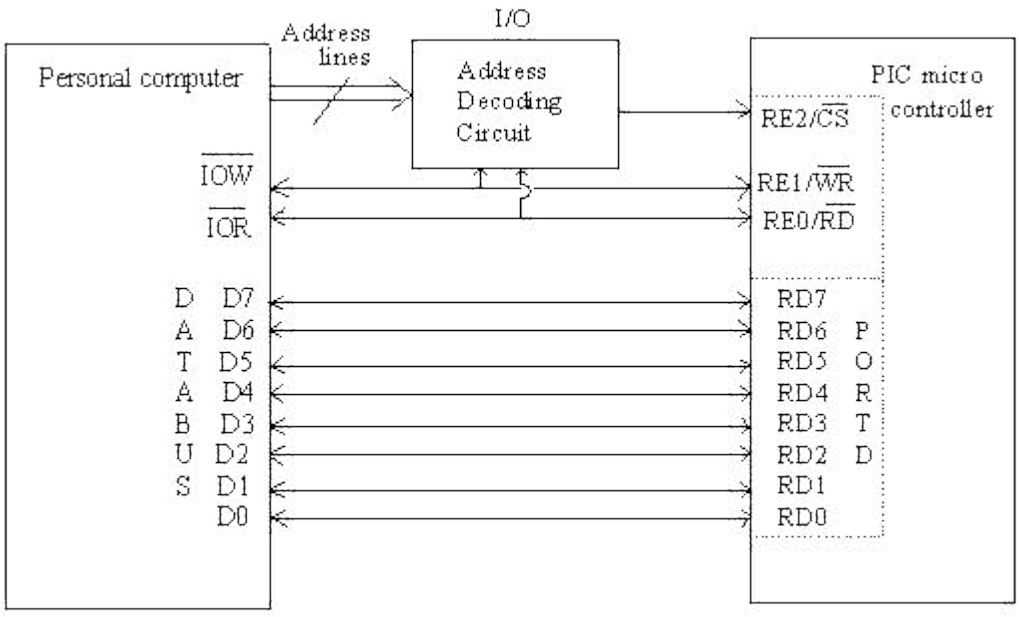


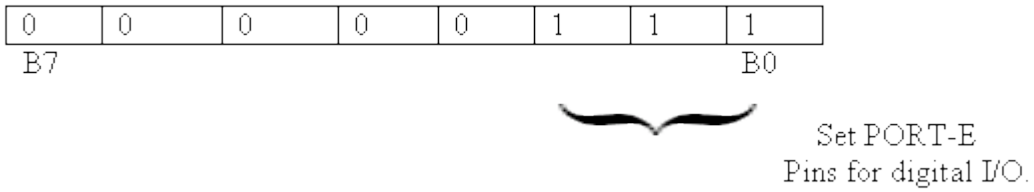
Fig 28.1 Interfacing a PIC-microcontroller with a PC using Parallel Slave Port

Registers used for PSP Mode

ADCON1:

Three low significant bits (PCFG2-PCFG0) are set to enable Port-E pins for digital I/O

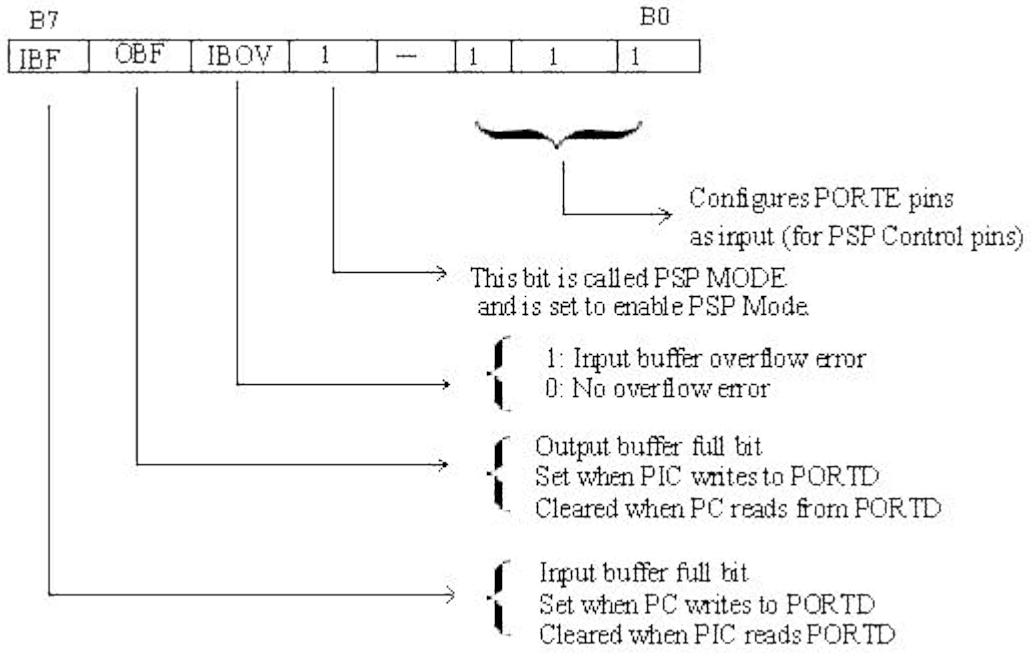
ADCON1, ADD: 9F H



TRISE:

This register plays a crucial role in PSP configuration and control. The lower three bits control the data direction of PortE. the upper four bits are used in conjunction with parallel slave port as shown here.

TRISE, ADD: 89 H



As explained, PSP Mode facilitates bidirectional 8-bit parallel data transfer. After ADCON1<b2-b0> and TRISE<b4, b2-b0> bits are set by the user program, PORTD and PORTE are configured for PSP. When PC wants to write an 8-bit data to PIC, it addresses the PIC microcontroller and the I/O address decoding circuit makes $\overline{CS}$ go low selecting the PIC chip. PC also makes $\overline{IOW}$ (I/O write) pin low and floats the data through its data bus (b7-b0). The data is written to PORTD and IBF flag in TRISE Register is set indicating that a byte is waiting at PORTD input buffer to be read by the PIC. Simultaneously PSPIF flag bit of PIR1 register is set and an interrupt is generated if PSPIE, PEIE and GIE bits have been set (i.e., the peripheral PSP interrupt is enabled.). After the data is read from PORTD, IBF bit automatically becomes zero; however PSPIF bit has to be cleared by software. If a second byte is written by the PC before the first byte is read, the second byte is lost and the IBOV flag in TRISE register is set indicating this loss.

Similarly a byte can also be read by the PC from the PIC microcontroller. When PIC writes a byte to PORTD, OBF flag is set indicating that the byte is waiting to be read by the PC. When the PC reads this bytes, OBF flag in TRISE Register is automatically cleared and the interrupt flag bit PSPIF is set indicating that the byte has been read by the PC from PIC microcontroller.