Microcontroller and Embedded Programming

(Code : ITC501)

Semester V - Information Technology
(Mumbai University)

Strictly as per the Choice Based Credit and Grading System (Revise 2016) of Mumbai University w.e.f. academic year 2018-2019

Harish G. Narula

Formerly, Assistant Professor (Senior),
Department of Computer Engineering
D. J. Sanghvi College of Engineering,
Mumbai.
Maharashtra, India.

Tech-Max Publications, Pune
Innovation Throughout
Engineering Division

MO242A

Chapter 1 Introduction to Embedded Systems 1-1 to 1-19

Chapter 2 8051 Microcontroller 2-1 to 2-8

LIST OF 8051 PROGRAMS

□□□

CHAPTER 1

Introduction to Embedded Systems

1.1 Embedded Systems and General Purpose Computers

→ **(MU - Dec. 14, Dec. 15, May 16, Dec.16, May 17, Dec. 17)**

Q. 1.1.1 Define embedded systems . (Ref. Sec. 1.1) **Dec. 14. Dec.16. May 17. 2 Marks**

Q. 1.1.2 What is embedded system ? (Ref. Sec. 1.1) **Dec.15. May 16. 2 Marks**

Q. 1.1.3 Define and classify the embedded systems also list major application areas of embedded systems. (Ref. Sec. 1.1) **Dec.17. 3 Marks**

Q. 1.1.4 Compare general purpose computer and embedded system. (Ref. Sec. 1.1) **(6 Marks)**

☞ **Definition**

- "A system that has embedded software and hardware, that makes it a dedicated system for a specific application or a part of an application or product is called as an Embedded system."
- Another way of defining the embedded system is "Any system that has a microprocessor or a microcontroller embedded into the system to control the system is called as Embedded System".
- Another definition of Embedded system is "It is a combination of a computer hardware with the software to control, monitor, communicate or do multiple operations".
- We have seen and used many devices in our daily life that have a microprocessor or a microcontroller embedded into it for controlling the system. For example, mobile, washing machine, microwave oven etc. These systems have a microprocessor or a microcontroller in it to control the system and are called as embedded system. The general purpose computers, as the name suggests, are such designed that they can perform multiple tasks, while the embedded systems are designed such that they can perform a specific task. Another major difference noted is that a general purpose computer normally uses a general purpose microprocessor, while an embedded system uses a microcontroller.
- A computer consists of microprocessor, large memory (ROM, RAM, secondary memories like hard disk etc.), I/O units (keyboard, mice, screen, modem, fax, etc.), networking units (Ethernet card, bus driver etc.), Operating System (OS) and other application software. An Embedded system on the other hand has hardware quite similar to a computer (usually no hard disk or secondary memory), but the OS of the embedded system is of a special type (mostly RTOS) and the other application software (that are also different than the ones in the computer). Table 1.1.1 shows the differences of a general purpose computer and an Embedded system.

Table 1.1.1 : Comparison of general purpose computer and embedded system

Sr. No.	General Purpose Computer	Embedded System
1.	It is designed using a microprocessor as the main processing unit.	It is mostly designed using a microcontroller as the main processing unit
2.	It contains a large memory semiconductor memories like cache and RAM. It also contains secondary storage like hard disks etc	It uses semiconductor memories, but normally doesn't require secondary memories like hard disks, CD etc. It sometimes has a special memory called as flash memory.
3.	It is designed such that it can cater to multiple tasks as per requirement.	It is designed such that it can cater to a particular predefined task.
4.	It is mostly costlier compared to the embedded systems	It is mostly cheaper compared to a computer.
5.	It requires huge number of peripheral devices and their controllers.	It requires less number of peripheral devices and their controllers are mostly on the microcontroller chip itself, hence further lower cost as well as space requirement.
6.	The Operating system and other software for the general purpose computers, are normally complicated and occupy more memory space	The operating system (mostly RTOS i.e. Real Time Operating System) and other software occupy less memory space.

Syllabus Topic : Overview of Embedded System Architecture

1.2 Overview of Embedded System Architecture

➔ **(MU - May 15, May 17)**

Q. 1.2.1 Discuss various components of embedded system. (Ref. Sec. 1.2) **May 15, 3 Marks**

Q. 1.2.2 Explain the embedded system architecture in detail. (Ref. Sec. 1.2) **May 17, 10 Marks**

- The embedded system components include its hardware and the software. We will first see the hardware components of the embedded system.
- The most important units in the embedded system is the processor. The processor can be said to be the heart of the embedded system. The various units in the embedded system includes the following :

1. Microprocessor or Microcontroller
2. Timers / Counters
3. Serial and parallel communication ports
4. Interrupt controller
5. Input devices with the driver circuits
6. Output devices with the driver circuits
7. Power supply, Reset and Oscillator circuits
8. Any application specific circuits.

- Fig. 1.2.1 shows the components of the embedded system hardware.
- We will discuss each of the components in short in the subsequent sub-sections. The details of these will be discussed in this book.

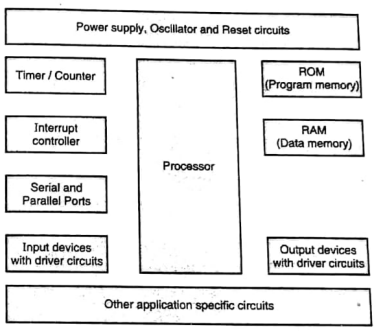


Fig. 1.2.1 : Hardware components of an embedded system

Syllabus Topic : Specialities of Embedded System

1.2.1 Specialities and Design Metrics of Embedded System

➔ **(MU - Dec. 15, Dec. 17)**

Q. 1.2.3 What are the specialties of Embedded Systems? (Ref. Sec. 1.2.1) **Dec. 15, 4 Marks**

Q. 1.2.4 What are the design metrics of Embedded Systems? (Ref. Sec. 1.2.1) **Dec. 17, 5 Marks**

When designing an embedded system the following points are to be considered.

➔ **1. Unit cost**

This is the cost of each unit. The cost includes the components cost and the man cost involved in it. It excludes the cost of development of the system for the first time.

➔ **2. Non Recurring Expenditure (NRE) cost**

This includes the costs that are incurred only once and are not for each system. This includes the development cost of the system and other investments that are to be done.

➔ **3. Size**

The software and the hardware size are also important parameters. It involves the memory space required for storing the program and data. It also involves the physical space required for the circuits required for the system.

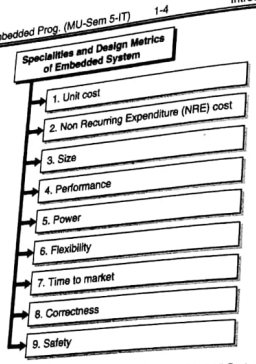


Fig. 1.2.2 : Specialities and Design Metrics of Embedded System

➔ **4. Performance**

Performance measure is a comparative term. The performance of the system is to be compared with the already existing systems. The response time and the speed are the important parameters to measure the performance of the system.

➔ **5. Power**

Power consumption of an embedded system is a very important parameter of the system. Most of the embedded systems are battery operated. In these cases the power consumption is more important parameter for the system design.

➔ **6. Flexibility**

It is a measure of cost involved in changing certain operational behaviour of the system. This cost is normally the non-recurring cost.

➔ **7. Time to market**

Time to market = Time to prototype + Time to refine + Time to produce in bulk

➔ **8. Correctness**

Test and Validation of the system is important.

➔ **9. Safety**

Is also an important aspect for design. Often these metrics are contradictory, hence calls for optimization. Processor choice, partitioning decisions, compilation knowledge and expertise in hardware and software both are required.

Syllabus Topic : Brief Introduction to Embedded Microcontroller Cores CISC and RISC

1.2.2 Embedded Microcontroller Core: Microprocessor and Microcontroller, RISC and CISC

Q. 1.2.5 Compare RISC and CISC processors. (Ref. Sec. 1.2.2) **(4 Marks)**

- There are various processors like general purpose processor (GPP or MP), microcontrollers (MC), digital signal processors (DSP), application specific processors (ASP), etc.
- We will go through these processors in this section.
- The microprocessor is a central processing unit of a general purpose digital computer. They can address megabytes of memory and operate on 8, 16, or 32 bit data.
- It consists of an ALU, accumulator, working registers, program counter, stack pointer, clock and interrupt circuits. Fig. 1.2.3 shows general architecture of a microprocessor.
- The microprocessor alone is not a complete digital computer. In order to make it a complete computers one should add memory devices (ROM, RAM, EEPROM, EPROM, PROM), memory decoders, I/O devices (keyboard and display controller i.e. IC 8279, Timer/Counter i.e. IC 8253/8254, programmable peripheral interface i.e. IC 8255, programmable interrupt controller i.e. IC 8259 etc).
- Due to varieties of memory and I/O devices, the hardware design of a microprocessor is arranged such that a very small or very large system can be configured around the CPU depending on the application of the user.

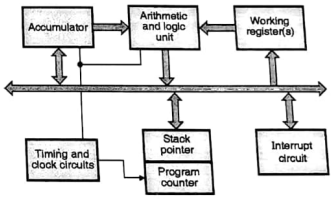


Fig. 1.2.3 : General architecture of a microprocessor

- Even for a small application the minimum size of memory and I/O (s) must be interfaced to the CPU. This increases the hardware. In turn the PCB size and cost of the system also increases.
- In order to avoid these drawbacks **Microcontrollers** were developed. The Microcontroller is an on-chip true computer. It is optimized for specific applications. It consists all the features of a microprocessor as well as the features required to build a complete computer.
- The microcontroller consists of RAM, ROM, I/O devices, counters and clock circuit etc.
- The first 8 bit microcontroller, 8048 was introduced by Intel in 1976 with a view to control general tasks.
- Later on, high performance microcontroller families like MCS51, MCS96 were developed. The MCS51 family includes a whole family of microcontrollers that have numbers ranging from 8031 to 8751. They are available in MOS or CMOS technology with different packages.

– The 8051 microcontroller in the MCS51 family was designed for 8 bit mathematical and single bit boolean operations. This family provides separate program and data memory. The program or data memory may be internal or external. They provide high speed of operation and have low system cost.

Thus, microprocessor is a chip which is dependent on other chips for many functions and the most important of them to interface I/O ports to the outside world data, while a microcontroller is as good as a single chip computer which has everything in-built. Table 1.2.1 shows the comparison of microprocessors and microcontrollers.

Table 1.2.1 : Comparison of microprocessors and microcontrollers

Sr. No.	Microprocessor (MP)	Microcontroller (MC)
1.	Microprocessor is an integrated circuit (IC) that integrates a number of useful functions lie execution of stored instructions, access data from memory and I/O devices etc.	Microcontroller is an integrated circuit (IC) that integrates a number of components i.e. microprocessor, memory and I/O controller.
2.	It is suited to processing information in computer systems.	It suited to control of I/O devices requiring a minimum component count.
3.	Microprocessor has instructions with powerful addressing modes and complex operations. It is capable of working on large data and hence the processing capability of microprocessor is much better compared to microcontroller.	MC have input and output instructions to set/clear bits Boolean operations (AND, OR, XOR, NOT, jump if a bit is set/cleared), etc. It has extremely compact instructions, many implemented in one byte (Control program must often fit in the small, on-chip ROM)
4.	They usually require external circuitry to do similar things (e.g. 8255 PPI, 8254 PIT, 8259 PIC)	They have built-in I/O operations, event timing, enabling and setting up priority levels for interrupts caused by external stimuli. Hence the circuit size reduces.
5.	Microprocessors have very wide large memory address spaces (> 4 Gbytes) lots of data (Data bus : 32, 64, 128 bits wide)	Microcontrollers have narrow relatively small memory address spaces (typically Kbytes) less data (Data bus typically 8, 16 bits wide).
6.	MP are very fast (> 1 GHz)	MC are Relatively slow (typically 10-20 MHz) since most I/O devices being controlled are relatively slow
7.	Microprocessors are comparatively expensive	Microcontrollers are comparatively cheaper

The microprocessors and the microcontrollers available are mainly of two types namely the RISC (Reduced Instruction Set Computing) and the CISC (Complex Instruction Set Computing). These processors are used according to their applications. Most of the CISC processors are used in computers while the RISC processors in Embedded System. Table 1.2.2 shows the comparison of these two types of processors i.e. the RISC and CISC processors.

Table 1.2.2 : Comparison of RISC and CISC processors

Sr. No.	Properties	RISC	CISC
1.	Number of Instructions	Less	More
2.	Addressing Modes	Less	More
3.	Instruction Formats	Less	More
4.	Instruction Size	Fixed	Variable
5.	Control Unit	Hardwired	Micro-programmed
6.	Number of Bus Cycles to execute an instruction	Single CPU cycle (for 80% Instructions)	Multiple CPU cycles
7.	Control Logic And Decoding Subsystem	Simple	Complex
8.	Pipelining	Huge no. of stages of Pipelining	Difficulty in efficient implementation
9.	Design time and Probability of Design Errors	Smaller time and less probable	Long time and Significant probability
10.	Complexity of Compiler	Simpler	More complex and the results of "optimization" may not be most efficient and the fastest machine language code
11.	HLL instructions	Supported	Not Supported

Syllabus Topic : Brief Introduction to Microcontroller Core DSP

1.2.2.1 Embedded Microcontroller Core: Digital Signal Processors (DSP) and Application Specific Processor (ASP)

→ **(MU - Dec. 15)**

Q. 1.2.6 Explain in brief features of DSP. (Ref. Sec. 1.2.2.1) **Dec. 15 , 3 Marks**

– There are some special processors that are specialized for signal processing applications. The DSP processors have instructions that perform complex instructions specialized to perform various signal processing operations.

– The differences of the general purpose processors with the digital signal processor is listed in the Table 1.2.3.

Table 1.2.3 : Comparison of DSP and GPP

Sr. No.	General Purpose Processor (GPP)	Digital Signal Processor(DSP)
1.	It has general purpose instructions	It has complex specialized instructions
2.	It has normally single operation in an instruction	It has multiple operations in a single instruction.
3.	These are mostly superscalar processors with high performance. Superscalar processors refer to processors with multiple ALUs and hence multiple instructions executed simultaneously.	These are normally VLIW (Very Large Instruction Width) processors with high performance. VLIW processors have multiple execution units for different types of instruction.

- Application specific processor (ASP) or application specific system processor (ASSP) are designed specially to cater the needs of a special task related embedded system. For example if a processor is specially designed to be used in the camera, then it becomes a ASP or ASSP.
- The ASP has the peripheral controllers as specifically required for the application. The instructions of the ASP are also specific to the application.
- The networking or interfacing standards are also as per the application. For example in the camera, image processing, data storage, controlling of the peripherals of camera, etc. are to be done.
- Finally let us compare all these types of processors. The comparison of all the types of processors is given in Table 1.2.4.

Table 1.2.4 : Comparison of various processors

Sr. No.	General Purpose Processor (GPP) or Microprocessor	Microcontroller (MC)	Digital Signal Processor (DSP)	Application Specific System Processor (ASSP) or single purpose processor
1. Application	GPPs find application in cases where intensive computations are required. The tasks are general purpose and will keep on varying from system to system. They are mainly used in computers.	It is used in systems that require memory and peripheral controller integrated in the processor chip itself. They are used in instruments which are not manufactured in bulk, for example a digital clock made by a student.	It is used in applications with signal processing capabilities like audio or video processing. It is used in embedded systems where signal processing is to be done for example digital camera.	As the name says, ASSP are application specific and hence the application of a particular ASSP will depend on the application. It is used mainly in embedded systems where the systems are manufactured in bulk, for example microwave oven, washing machine etc.
2. Advantage	The main advantage is that the processors are readily available and are manufactured in bulk and hence have less designing and manufacturing costs.	The main advantage is that the microcontrollers are also available readily and manufactured is bulk. Hence the cost of microcontrollers is very less. Also the peripheral controllers are on chip and hence the space required is less as well as the cost of the overall system is less.	The main advantage is that the standard DSP processors available have very well designed instruction set and hence a lot of DSP related operations can be easily carried out. Also since the DSP are available easily, they are not very costly.	The main advantage of the ASSP is that the processors are application specific and hence the resources are exactly as required.

Sr. No.	General Purpose Processor (GPP) or Microprocessor	Microcontroller (MC)	Digital Signal Processor (DSP)	Application Specific System Processor (ASSP) or single purpose processor
3. Disadvantage	The disadvantage is that there are many resources available on the chip that may not be required for a particular application	The disadvantage is again similar to the processor i.e. there are some resources in the microcontroller that are not useful for a particular application.	The disadvantage is that since these are the standard available processors, they have certain attributes included in it that may not be useful for a particular application	The only disadvantage of ASP is that the initial design cost is very high and hence the processors are costlier. But when the products are to be developed in bulk, these processors also become cheaper.

Syllabus Topic : Brief Introduction to Embedded Microcontroller Core(SoC)

1.2.2.2 Embedded Microcontroller Core: System On Chip (SOC)

→ **(MU - Dec. 15, May 16, Dec. 16)**

Q. 1.2.7 Explain in brief features of SOC. (Ref. Sec. 1.2.2.2) **Dec. 15. 3 Marks**

Q. 1.2.8 Explain SoC in detail. (Ref. Sec. 1.2.2.2) **May 16. Dec. 16. 2.5 Marks**

Q. 1.2.9 Explain various embedded microcontroller cores in detail. (Ref. Sec. 1.2.2.2) **Dec. 16. 10 Marks**

A SOC may be embedded with the following components and features:

1. It is an Embedded processor that may be General Purpose Processor (GPP) or Application Specific Instruction set Processor (ASIP)
2. It is a Single purpose processing core or multiple processor cores
3. It has a network bus protocol implementing core
4. It has an encryption unit
5. It has a Signal processing unit
6. It normally also has a Field Programmable Gate Array (FPGA) to implement some logical circuit
7. It will also definitely have Memory
8. It also mostly has Analog units

One of the widely used place for SOC is mobile phones. Single purpose processors are configured for dialing, modulating, demodulating, deciphering, interfacing keypad and multi line LCD or graphic LCD, touch screen, storing data etc. An example of the SOC is shown in Fig. 1.2.4.

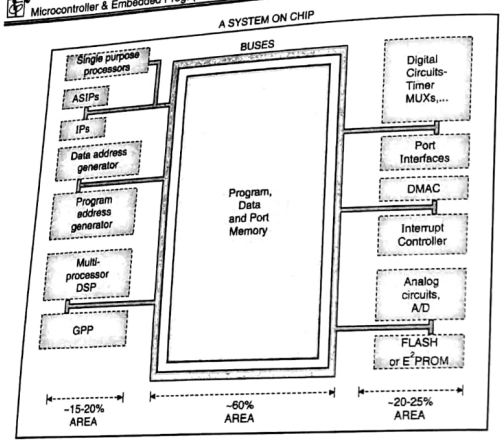


Fig. 1.2.4 : System on chip (SOC)

1.2.2.3 Embedded Microcontroller Core: FPGA and CPLD

→ (MU - Dec.16)

Q. 1.2.10 Explain various embedded microcontroller cores in detail. (Ref. Sec.1.2.2.3) **Dec.16, 10 Marks**

- FPGA (Field Programmable Grid Array) and CPLD (Complex Programmable Logic Device) have a huge number of gates that are programmable in terms of their connection and hence establish the required digital circuit on a single IC without a specially designed ASP nor the huge circuits that occupy more space.
- The Fig 1.2.5 shows a simple CPLD, that is made up of many PLDs and their programmable interconnect.
- The PLD is an array of NOT gates, OR gates and AND gates that can be connected in the required manner to achieve the required circuit. The pins on the four sides of the architecture are used to connect the CPLD with the system and these pins are called as I/O ports.
- FPGAs have more PLDs and also have storage i.e. flip-flops. They have a huge number of programmable logic blocks and a lots of programmable interconnect. Fig. 1.2.6 shows the architecture of a FPGA. FPGA also has I/O ports to connect itself in the system.

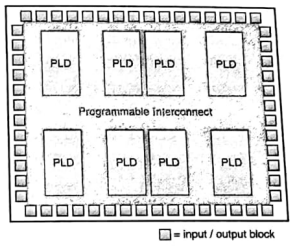


Fig. 1.2.5 : CPLD architecture

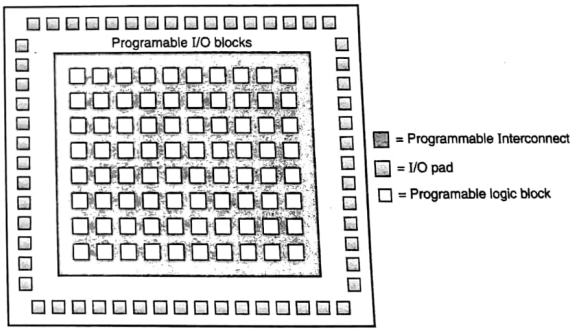


Fig. 1.2.6 : FPGA Architecture

1.2.3 Peripheral and Memory components of an Embedded System

☞ **Timer / Counter**

- In Microprocessor System, we come across two important operations i.e. timer which is used to provide a delay and counter which is used to count incoming pulses.
- In large systems, where microprocessor's wastage time is critical, a separate timer IC like IC8254 can be used. These counters can work as counter or can provide accurate time delays. Microcontrollers have these timers on-chip.

☞ ***Interrupt Controller***

The various peripheral controllers on-chip in case of a microcontroller need to be properly prioritised and controlled properly. The microprocessors require external interrupt controller, while most of the microcontrollers have on-chip interrupt controller.

☞ ***Serial and Parallel ports***

The embedded system needs to interface with the external devices for which it needs ports for interfacing. The serial ports give an advantage of long distance data transfer at lower costs, while parallel communication gives faster data transfer for shorter distance. We will see various communication protocols in the further chapters.

☞ ***Memory***

In an embedded system, ROM (Read only memory) is used for storing program or code while RAM (Random Access Memory) is used for storing data. The ROM used is normally the EEPROM (Electrically Erasable and Programmable ROM). Some data which is a constant value is also sometimes stored in ROM instead of RAM. There are flash memories that are high speed memory.

1.2.4 Power Supply, Oscillator and RESET Circuits

Besides the processor, peripherals and memory, the support circuits i.e. the power supply, oscillator and reset circuit are also very important. In this section we will study these three modules of an embedded system.

1.2.4.1 Power Supply

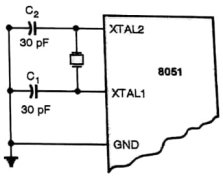
- Any electronic circuit requires a power supply. The power supply circuit that consists of a transformer, rectifier and regulator is also an important ingredient of the embedded system. Various voltages are required for different circuits in the system.
- Most of the microcontrollers require +5V or +3V as Vcc. Other peripherals may require different voltages according to the application. Hence the power supply is suppose to provide the power to various circuits in the system.
- The power supply is to be properly designed considering the current requirements of the blocks in the system.
- Many embedded systems are battery operated and hence the source of power is the battery. Certain embedded systems use a special technique called as charge pump.
- They utilize the charge on the bus and hence save power. It uses a set of capacitors and diodes. The capacitors are charged by the signals given on the bus. The capacitors store this charge and is utilised later as a power source.
- This can be also used to convert the voltage to a higher level or a lower level. The simple connection and charge storage system of these charge pumps enables them to be easily implementable and very effective for applications where low power consumption is required.

1.2.4.2 Oscillators

Clock pulses are required for any digital system. Microprocessor based systems or the embedded systems are digital systems. Some microcontrollers have on chip oscillator circuit and require only a crystal to be connected externally, while some require entire oscillator circuit to be external. Let us take an example of 8051 microcontroller.

The 8051 has an on-chip oscillator, but needs an external clock to run it.

- The oscillator circuit that is used in any microcontroller based system that generates the clock pulses for the internal operations to be synchronized.
- For 8051, there are two pins named as XTAL1 and XTAL2 to connect a resonating network that comprises of a crystal and capacitors to form an oscillator. Quartz crystal has an advantage better performance and hence it is used. Fig. 1.2.7 shows the oscillator circuit. Generally the capacitors C_1 and C_2 as shown in the figure are of 30 pF.
- The operating frequency and the crystal frequency for an 8051 based system is same.



m(19.7)**Fig. 1.2.7 : Connection of external crystal to 8051**

- 8051 microcontroller are available with the operating frequency ranging from 1MHz to 16 MHz. 8051 can also be sourced with external clock signal. Fig. 1.2.8 shows how 8051 is driven by an external clock signal.

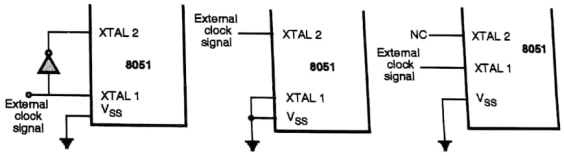


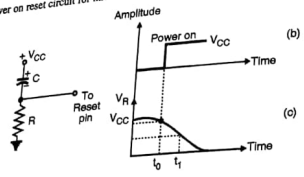
Fig. 1.2.8 : Using an external clock

- Generally 11.0592 MHz crystal oscillators are used so that the 8051 microcontroller systems are compatible with the serial ports. One machine cycle comprises of 12 oscillator periods. Hence, to find the time for 1 machine cycle we need to consider $\frac{1}{12}^{th}$ of the crystal frequency.

1.2.4.3 Reset Circuit

- The Reset pin of a microcontroller is either active high or active low. Let us consider the case of 8051 microcontroller.
- The Reset pin for 8051 microcontroller is active HIGH. Whenever power is switched ON, positive going pulse should be present for two machine cycles (The smallest time interval of time that is required to execute an instruction is called as a machine cycle) on this pin.
- The Reset pin can also be considered as an interrupt because the program cannot block the signal on reset pin.

– *Fig. 1.2.9 shows power on reset circuit for microcontroller 8051.*



(a) Reset circuit (b) Power on pulse (c) Across resistor

Fig. 1.2.9

– *At time t_0, the power supply is switched ON. The supply voltage V_{CC} appears across the RC network. The entire voltage appears across the resistor R, so V_R is approximately equal to V_{CC}. This resets the 8051.*

– *Once the capacitor charges, then V_R starts reducing and reaches approximately 0V. This removes reset signal. The oscillator $t_1 - t_0$ is reset time.*

Syllabus Topic : Recent Trends in Embedded Systems

1.3 Recent Trends and Design Process of Embedded System

Q. 1.3.1 Explain the design process of Embedded System. (Ref. Sec. 1.3) **(8 Marks)**

The following are the major Subtasks of Embedded System Design:

1. **Modelling the system to be designed and finding the constraints :** The block diagram of the system is to implemented. The various circuits required to implement these blocks and their availability and cost constraints are to be considered. In the software front we need to experiment and analyse with various algorithms and their space and time complexities. The huge tasks of the system are to be divided into smaller subtasks and their interactions are to be modelled.
2. **Refinement of the hardware and software :** The hardware and the software architecture designed are to be refined to give better performance. The power consumption and the time for computation are to be considered.
3. **Hardware-Software partitioning :** The hardware and software are finally to be partitioned i.e. separate working on the designed hardware and the software is to be done. Allocating the tasks into hardware, software running on custom hardware or general purpose hardware. For example, the timer is to be implemented by hardware or software is to be decided at this level and accordingly the partitioning of the same is to be done.
4. **Scheduling :** Once the partitioning is done, the scheduling is to be done. Scheduling refers to allocation of time steps for several modules sharing the same resource. Thus if a resource sharing is required for various stages of the design then that can be done by the scheduling i.e. the required task should be completed as and when required of the various parallel working tasks.

5. **Implementation :** Actual hardware building and software code generation. The hardware to be designed is implemented in this stage. The code generation also begins parallel to the hardware implementation.
6. **Simulation and Validation :** Once the implementation of the software is done, its simulation is done. Then it is interfaced with the hardware implemented and the system is simulated and validated for its working.
7. **Iterate :** If the validation of the system is not as expected then the above process is to be iterated until the expected results are obtained.

– Besides partitioning of Hardware and software it also requires Hardware Software Co-design. Traditional design is that software and hardware partitioning is done at an early stage and development henceforth proceeds independently.

– CAD tools are then focused towards hardware synthesis. For embedded systems we need several components like DSPs, microprocessors, network and bus interface etc.

– Hardware-Software co-design allows hardware and software design to proceed in parallel with interactions and feedback between the two processes. It also evaluates tradeoffs and hence performance yields ultimate result.

Syllabus Topic : Categories of Embedded System

1.4 Categories of Embedded System

➔ **(MU - Dec. 14,May 17, Dec. 17)**

Q. 1.4.1 Classify the embedded systems, give few examples of such systems. (Ref. Sec. 1.4) **Dec. 14, May 17, Dec. 17, 3 Marks**

The embedded systems are classified on the size of the system. The different types of embedded systems are:

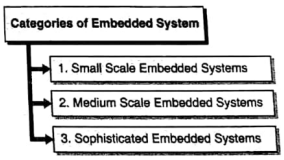


Fig. 1.4.1 : Categories of Embedded System

➔ **1. Small Scale Embedded Systems**

Systems that are designed using 8-bit microcontrollers like 8051 or 16-bit microcontrollers like 80196 fall under this category of embedded systems i.e. small scale embedded systems. The hardware and software complexities in these systems is very low. The power consumption is also not an issue and they are mostly battery operated. The coding can be done in simple embedded 'C'. Also the memory may be small and hence care has to be taken that the software is not very huge. An example of such a system could be a simple temperature measurement embedded system, a robotic arm controller, etc.

➔ **2. Medium Scale Embedded Systems**

Systems that are designed using 16-bit to 32-bit microcontrollers fall under this category. They may also have multiple such microcontrollers or DSPs. The hardware as well as software complexity of such systems is very high. They mostly have an operating system besides the applications required. The RTOS (Real Time Operating System) is used in such systems. These systems may also employ some ASSP processors to implement a special task if any. Various examples of medium scale embedded systems are routers for networking, ATM (i.e. Automated Teller Machine for bank transactions) machines etc

➔ **3. Sophisticated Embedded Systems**

These systems also have high end microcontrollers. The hardware and software complexities are also very high. The design tools required are also complicated and costly. The systems also use the RTOS and complicated time bound applications. The time bounding on the computations is a big task in such systems. These systems are used for cutting edge applications like smart phones, multimedia systems etc.

Syllabus Topic : Application Areas

1.5 Application Areas of Embedded System

➔ **(MU - Dec. 16, Dec. 17)**

Q. 1.5.1 Explain application areas of embedded system. (Ref Sec. 1.5) **Dec. 16, Dec. 17, 3 Marks**

There is a huge number of examples for embedded system. In this section we will see various examples of embedded systems based on the classification seen in the previous section.

1.5.1 Small Scale Embedded Systems

Some of the examples of small scale embedded applications:

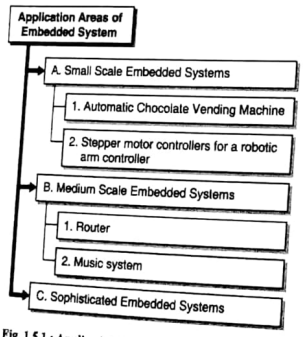


Fig. 1.5.1 : Application Areas of Embedded System

➔ **1. Automatic Chocolate Vending Machine**

This is a system that dispenses chocolates automatically. The system will require a microcontroller to control the system. A display will be required to display various messages for the customer. A mechanism will be required to accept the coins or notes and analyze the same to find their value. Based on the value, a decision is to be taken to dispense the chocolate and return the extra money if any. Thus the various tasks that are to controlled by the microcontroller are not very complicated, hence a small scale system can easily implement this system.

➔ **2. Stepper motor controllers for a robotic arm controller**

A robotic arm controller is a mechanism that is used to perform various movements like that of a human arm and do a particular operation mainly pick-and-place for an object. The stepper motor requires a special sequence of binary data to move in a particular direction. This 4-bit binary data can be given from the port of a microcontroller. There may be multiple stepper motors to implement various joints of the robotic arm. This can again be implemented easily by a small scale embedded system.

There can be various other small scale systems like digital multi-meter to measure voltage, current and resistance. Small toys, computer peripheral controllers, TV remote, power windows in vehicles, electronic instruments such as temperature controller etc. are also examples of small scale embedded systems.

1.5.2 Medium Scale Embedded Systems

Some of the examples of medium scale embedded systems:

➔ **1. Router**

A networking router used in a multi-computer system is a medium scale embedded system. It requires proper routing algorithm to be implemented. The protocol is to be implemented to have synchronization for communication. Various ports are required to interface the different computers in the multi-computer system.

➔ **2. Music system**

- Implementation of a music system that can play various formats of songs is also a medium scale embedded system. The codec (i.e. the coder and decoder) of these formats is to be implemented, which is a complicated task and requires a lot of algorithm development and programming. Also the hardware for such system has to be fast enough and hence requires high end microcontrollers or DSP processors.
- There can be many other medium scale embedded systems like ATM (i.e. Automated Teller Machine for bank transactions), FAX machine, pager, PDA (Personal Digital Assistant), etc.

1.5.3 Sophisticated Embedded Systems

Some of the examples of sophisticated embedded systems:

The systems like wireless LAN, real time video and speech processing systems, high speed LAN and WAN, Network security systems etc are examples of sophisticated embedded systems.

The list of embedded systems in very huge. Here is a list of some more embedded systems that is a mixture of the above categories.

1. Washing Machine
2. Digital Camera
3. Power windows of a vehicle
4. Power steering of a vehicle
5. Fuel injection system of a vehicle
6. Air conditioner
7. Smart Phone
8. Music Player
9. Automatic Temperature controller
10. MODEM
11. Mouse
12. Keyboard controller
13. LAN card
14. Printer
15. Scanner
16. Router
17. Digital watch
18. Digital rolling display for advertisements
19. Digital Locker
20. Set top box and many others.

1.6 Exam Pack (University and Review Questions)

Q. 1 Define embedded systems. **(Refer section 1.1)** **(D-14, D-16, M-17, 2 Marks)**

Q. 2 What is embedded system ? **(Refer section 1.1)** **(D-15, M-16, 2 Marks)**

Q. 3 Define and classify the embedded systems also list major application areas of embedded systems. **(Refer section 1.1)** **(D-17, 3 Marks)**

Q. 4 Compare general purpose computer and embedded system **(Refer section 1.1)** **(6 Marks)**

☞ **Syllabus Topic : Overview of Embedded System Architecture**

Q. 5 Discuss various components of embedded system. **(Refer section 1.2)** **(M-15, 3 Marks)**

Q. 6 Explain the embedded system architecture in detail. **(Refer section 1.2)** **(M-17, 10 Marks)**

☞ **Syllabus Topic : Specialities of Embedded System**

Q. 7 What are the specialties of Embedded Systems? **(Refer section 1.2.1)** **(D-15, 4 Marks)**

Q. 8 What are the design metrics of Embedded Systems? **(Refer section 1.2.1)** **(D-17, 5 Marks)**

☞ **Syllabus Topic : Brief Introduction to Embedded Microcontroller Cores CISC and RISC**

Q. 9 Compare RISC and CISC processors. **(Refer section 1.2.2)** **(4 Marks)**

☞ **Syllabus Topic : Brief Introduction to Microcontroller Core DSP**

Q. 10 Explain in brief features of DSP. **(Refer section 1.2.2.1)** **(D-15, 3 Marks)**

☞ **Syllabus Topic : Brief Introduction to Embedded Microcontroller Core(SoC)**

Q. 11 Explain in brief features of SOC. **(Refer section 1.2.2.2)** **(D-15, 3 Marks)**

Q. 12 Explain SoC in detail. **(Refer section 1.2.2.2)** **(M-16, D-16, 2.5 Marks)**

Q. 13 Explain various embedded microcontroller cores in detail. **(Refer section 1.2.2.2)** **(D-16, 10 Marks)**

Q. 14 Explain various embedded microcontroller cores in detail. **(Refer section1.2.2.3)** **(D-16, 10 Marks)**

☞ **Syllabus Topic : Recent Trends in Embedded Systems**

Q. 15 Explain the design process of Embedded System. **(Refer section 1.3)** **(8 Marks)**

☞ **Syllabus Topic : Categories of Embedded System**

Q. 16 Classify the embedded systems, give few examples of such systems. **(Refer section 1.4)** **(D-14, M-17, D-17, 3 Marks)**

☞ **Syllabus Topic : Application Areas**

Q. 17 Explain application areas of embedded system. **(Refer section 1.5)** **(D-16, D-17, 3 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 2

8051 Microcontroller

Syllabus Topic : Introduction to 8051 Microcontroller

2.1 Introduction

- The microprocessor is a central processing unit of a general purpose digital computer. They can address megabytes of memory and operate on 8, 16, or 32 bit data.
- It consists of an ALU, accumulator, working registers, program counter, stack pointer, clock and interrupt circuits. Fig. 2.1.1 shows general architecture of a microprocessor.
- The microprocessor alone is not a complete digital computer. In order to make it a complete computers one should add memory devices (ROM, RAM, EEPROM, EPROM, PROM), memory decoders, I/O devices (keyboard and display controller i.e. IC 8279, Timer/Counter i.e. IC 8253/8254, programmable peripheral interface i.e. IC 8255, programmable interrupt controller i.e. IC 8259 etc).

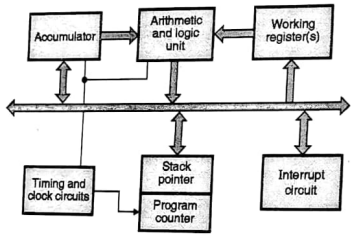


Fig. 2.1.1 : General architecture of a microprocessor

- Due to varieties of memory and I/O devices, the hardware design of a microprocessor is arranged such that a very small or very large system can be configured around the CPU depending on the application of the user.
- Even for a small application the minimum size of memory and I/O (s) must be interfaced to the CPU. This increases the hardware. In turn the PCB size and cost of the system also increases.

In order to avoid these drawbacks **Microcontrollers** were developed. The Microcontroller is an on-chip true computer. It is optimized for specific applications. It consists all the features of a microprocessor as well as the features required to build a complete computer.

- The microcontroller consists of RAM, ROM, I/O devices, counters and clock circuit etc.
- The first 8 bit microcontroller, 8048 was introduced by Intel in 1976 with a view to control general tasks.
- Later on, high performance microcontroller families like MCS51, MCS96 were developed. The MCS51 family includes a whole family of microcontrollers that have numbers ranging from 8031 to 8751. They are available in MOS or CMOS technology with different packages.
- The 8051 microcontroller in the MCS51 family was designed for 8 bit mathematical and single bit boolean operations. This family provides separate program and data memory. The program or data memory may be internal or external. They provide high speed of operation and have low system cost.

2.2 Comparison of Microcontroller and Microprocessor

Q. 2.2.1 Differentiate between a microprocessor and a microcontroller. (Ref. Sec. 2.2) **(4 Marks)**

A microprocessor is a chip which is dependent on other chips for many functions and the most important of them to interface I/O ports to the outside world data, while a microcontroller is as good as a single chip computer which has everything in-built.

☞ **What is a microprocessor (MP) ?**

A device that integrates a number of useful functions into a single IC package. Some functions are :

- Ability to execute a stored set of instructions to carry out user defined tasks.
- Ability to access external memory chips to read/write data from/to memory.
- Ability to interface with I/O devices

☞ **What is a microcontroller (MC)?**

Basically, a device which integrates a number of the components of a microprocessor system onto a single chip. Only need to be supplied power and clocking. Microcontroller combines on the same chip :

- The CPU core
- I/O
- Memory

Some differences between microprocessors and microcontrollers that make the microcontroller fit for embedded system

MP : suited to processing information in computer systems.

MC : suited to control of I/O devices requiring a minimum component count.

☞ **Instruction sets**

- **MP** have processing intensive powerful addressing mode instructions to perform complex operations and manipulate large volumes of data. Processing capability of MCs never approaches those of MPs large instructions.

- *MC have input and output instructions to set/clear bits Boolean operations (AND, OR, XOR, NOT, jump if a bit is set/cleared), etc. It has extremely compact instructions, many implemented in one byte (Control program must often fit in the small, on-chip ROM)*

☞ ***Hardware and Instruction set support***

- *MC : built-in I/O operations, event timing, enabling and setting up priority levels for interrupts caused by external stimuli. Hence the circuit size reduces.*
- *MP : usually require external circuitry to do similar things (e.g. 8255 PPI, 8254 PIT, 8259 PIC)*

☞ ***Bus widths***

- *MP : very wide large memory address spaces (> 4 Gbytes) lots of data (Data bus : 32, 64, 128 bits wide)*
- *MC : narrow relatively small memory address spaces (typically Kbytes) less data (Data bus typically 8, 16 bits wide).*

☞ ***Clock rates***

- *MP very fast (> 1 GHz)*
- *MC : Relatively slow (typically 10-20 MHz) since most I/O devices being controlled are relatively slow*

☞ ***Cost***

- *MPs expensive*
- *MCs cheap. Hence the effective cost of the embedded system is reduced*

2.3 Features of 8051

Q. 2.3.1 Describe the features of microcontroller 8051. (Ref. Sec. 2.3) **(2 Marks)**

(1) 8 bit CPU optimized for control applications.

(2) 4 KB of on chip program memory.

(3) 128 bytes of on chip data memory.

(4) 64 KB program external ROM and 64 KB external RAM addressability.

(5) 32 bidirectional and individually addressable I/O lines arranged as four 8 bit ports P0-P3.

(6) Two 16 bit timer/counters.

(7) Full duplex serial data transmitter/receiver.

(8) Four register banks.

(9) 8 bit program status word and stack pointer.

(10) Interrupt structure with two priority levels.

(11) On chip oscillator and clock circuits.

(12) Direct bit and byte addressability.

(13) Binary or decimal arithmetic.

(14) Signed-overflow detection and parity computation.

(15) Integrated Boolean processor for control applications.

(16) Full depth stack for subroutine return linkage and data storage.

Syllabus Topic : Pin Configuration

2.4 Pin Functions of 8051

➔ **(MU - Dec. 14, Dec. 16)**

Q. 2.4.1 Explain function of $\overline{PSEN}$ and $\overline{EA}$ pins of 8051. (Ref. Sec. 2.4) **Dec. 14, 4 Marks**

Q. 2.4.2 Explain the pin configuration of 8051 microcontroller. (Ref. Sec. 2.4) **Dec. 16, 5 Marks**

Q. 2.4.3 Explain the function of the following pins :

(i) ALE/$\overline{PROG}$ (ii) RST (Ref. Sec. 2.4) **(4 Marks)**

- The pin diagram of 8051 is shown in Fig. 2.4.1. It is available in a standard 40 pin dual in line package.
- The devices 8031, 8751 have the same pin-out, same timing and same electrical characteristics. The difference lies in the on chip program memory, that is different for different user requirements.

☞ **V_{CC} :** Supply voltage

☞ **GND :** Ground

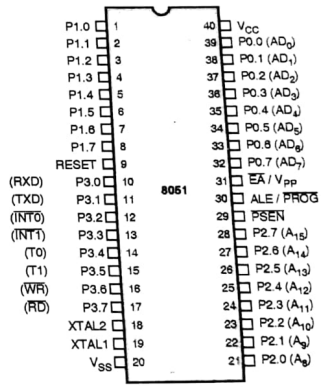


Fig. 2.4.1 : Pin diagram of 8051

☞ **Port 0**

- Port 0 is an 8-bit open drain bidirectional I / O port.
- As an output port each pin can sink eight TTL inputs.
- When 1s are written to port 0 pins, the pins can used as high-impedance inputs.

- Port 0 can also be configured to be the multiplexed low order address/ data bus during accesses to external program and data memory.
- Port 0 also receives the code bytes during Flash programming and outputs the code bytes during program verification.

☞ **Port 1**

- Port 1 is an 8-bit bidirectional I/O port with internal pull-ups.
- The Port 1 output buffers can sink/source four TTL inputs.
- When 1s are written to port 1 pins, they are pulled high by the internal pull-ups and can be used as inputs. As inputs, port 1 pins that are externally being pulled low will source current because of the internal pull-ups.

☞ **Port 2**

- Port 2 is an 8-bit bidirectional I/O port with internal pull-ups.
- The port 2 output buffers can sink/source four TTL inputs.
- When 1s are written to port 2 pins they are pulled high by the internal pull-ups and can be used as inputs. As inputs, port 2 pins that are externally being pulled low will source current due to internal pull-ups.

☞ **Port 3**

- Port 3 is an 8-bit bidirectional I/O port with internal pull-ups.
- The port 3 output buffers can sink/source four TTL inputs.
- When 1s are written to port 3 pins they are pulled high by the internal pull-ups and can be used as inputs. As inputs, port 3 pins that are externally being pulled low will source current due to internal pull-ups.
- Port 3 also serves various other functions as listed below:

P3.0	RXD (serial input port)
P3.1	TXD (serial output port)
P3.2	$\overline{INT0}$ (external interrupt)
P3.3	$\overline{INT1}$ (external interrupt)
P3.4	T0 (Timer / Counter 0 external input)
P3.5	T1 (Timer / Counter 1 external input)
P3.6	$\overline{WR}$ (external data memory write strobe)
P3.7	$\overline{RD}$ (external data memory read strobe)

- Port 3 also receives some control signals for flash programming and program verification.

☞ **RST (Reset input)**

- A HIGH on this pin for two machine cycles while the oscillator is running resets the device.

☞ **ALE / $\overline{PROG}$**

- Address latch enable output pulse for latching the low byte of address during the access to external memory.

- This pin is also the program pulse input ($\overline{PROG}$) during Flash programming. When 8051 is reset/ON, it checks P1.0 & P1.1. If P1.0 & P1.1 = 0 & 1 respectively, it enters into Flash programming mode else it enters into normal operation mode.
- In normal operation, ALE is emitted at constant rate of 1/6 of the oscillator frequency and may be used for external clocking or timing purposes.
- If desired, ALE operation can be disabled by setting bit 0 of SFR location 8EH. When disabled, ALE is active only during some instructions, like MOVX and MOVC.
- Setting the ALE-disable bit has no effect if the microcontroller is in external execution mode.

☞ **$\overline{PSEN}$**

- Program store enable is the read strobe to external program memory.
- When 8051 is executing code from external program memory, $\overline{PSEN}$ is activated twice each machine cycle except that two $\overline{PSEN}$ activations are skipped during each access to external data memory.

☞ **$\overline{EA}$/V_{PP}**

- External access enable. $\overline{EA}$ must be strapped to ground in order to enable the device to fetch code from external program memory locations starting from 0000H to 0FFFH.
- $\overline{EA}$ should be strapped to V_{CC} for fetching from internal program memory.
- This pin also receives 12V, programming enable voltage (V_{PP}) during flash programming.
- The effect of $\overline{EA}$ pin is as shown in Fig. 2.4.2.

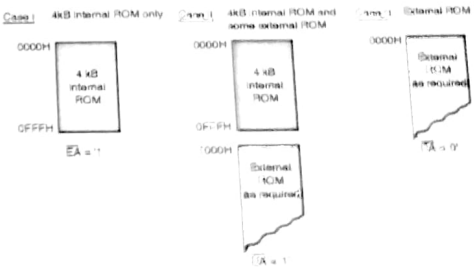


Fig. 2.4.2

☞ **XTAL1**

- Input to the inverting oscillator amplifier and input to the internal clock operating circuit.

☞ **XTAL2**

- Output from the inverting oscillator amplifier. A crystal can only be connected between XTAL1 and XTAL2 pins.

2.5 CPU Timing and Machine Cycle

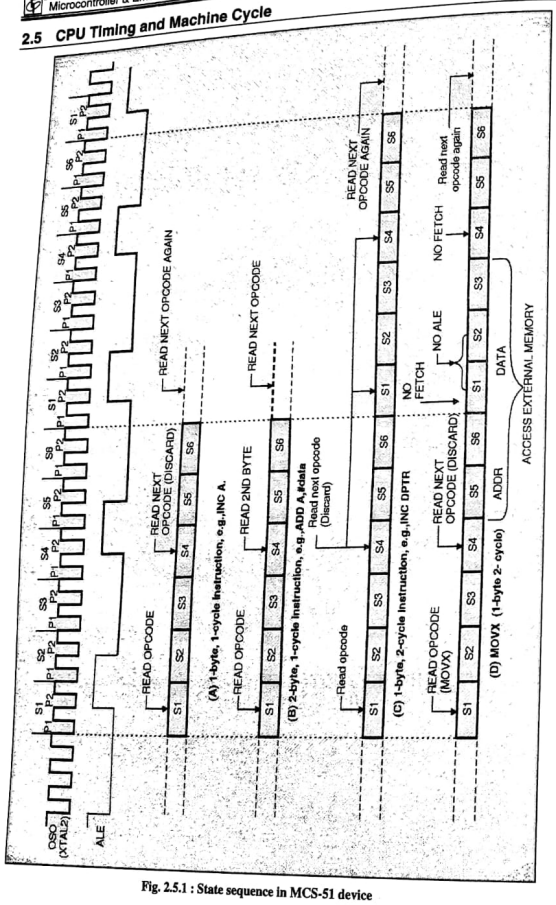


Fig. 2.5.1 : State sequence in MCS-51 device

- A machine cycle consists of sequence of 6 states, numbered S1 through S6. Each state time lasts for two oscillator periods i.e. each state has period 1 and period 2. Therefore, one machine cycle takes 12 oscillator clock periods.
- Each machine state is divided into two period : period 1 and period 2. During period 1, period 1 clock is active and period 2 clock is active during period 2. The machine cycle states will be numbered as S1P1 (State 1, period 1), S1P2 (State 1, period 2) ..... S6P2 (State 6, period 2).
- Each period lasts for one oscillator period. Typically, arithmetic and logical operations occur during period 1 and register to register transfers occur during period 2. Fig. 2.5.1 shows the fetch 1 execute sequence in states and phases for different instructions.
- In case of one byte two cycle instruction e.g. MOVX. The MOVX instruction accesses external data memory. Two fetches are skipped while the external data memory is being addressed and strobed. Fig. 2.5.1 shows the timing diagram.

2.6 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : Introduction to 8051 Microcontroller**

Q. 1 Differentiate between a microprocessor and a microcontroller. **(Refer Section 2.2)** **(4 Marks)**

Q. 2 Describe the features of microcontroller 8051. **(Refer Section 2.3)** **(2 Marks)**

☞ **Syllabus Topic : Pin Configuration**

Q. 3 Explain function of $\overline{\text{PSEN}}$ and $\overline{\text{EA}}$ pins of 8051. **(Refer Section 2.4)** **(D-14, 4 Marks)**

Q. 4 Explain the pin configuration of 8051 microcontroller. **(Refer Section 2.4)** **(D-16, 5 Marks)**

Q. 5 Explain the function of the following pins :

(i) ALE/$\overline{\text{PROG}}$ (ii) RST **(Refer Section 2.4)** **(4 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 3

Architecture of 8051

Syllabus Topic : Architecture

3.1 Architecture of 8051

→ **(MU - Dec. 16)**

Q. 3.1.1 Explain the architecture of 8051 microcontroller. (Ref. Sec. 3.1) **Dec. 16, 10 Marks**

Q. 3.1.2 Explain the organization of 8051 microcontroller with proper block diagram. (Ref. Sec. 3.1) **(5 Marks)**

Fig. 3.1.1 shows the architectural block diagram of 8051. It consists of

(i) Eight bit register A (Accumulator) and register B.

(ii) Arithmetic and Logic Unit (ALU).

(iii) 16 bit Program counter (PC).

(iv) 16 bit Data pointer (DPTR).

(v) 8 bit Program Status Word (PSW).

(vi) 8 bit stack pointer (SP).

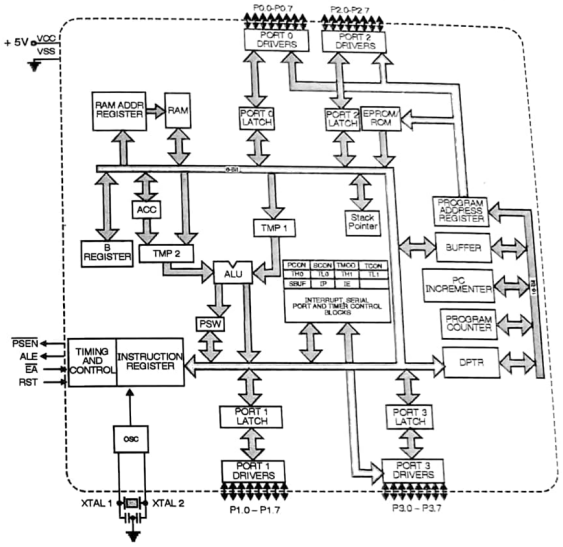
(vii) 128 byte Internal RAM and 4 KB Internal ROM.

(viii) Four 8 bit ports : Port 0, Port 1, Port 2, Port 3.

(ix) Two 16 bit Timer/counters, serial port and interrupt control.

(x) Control Registers.

(xi) On chip oscillator.



m(19.5)**Fig. 3.1.1 : Architectural block diagram**

3.1.1 Important Features of 8051 Architecture

→ **(MU - Dec. 15, May 17)**

Q. 3.1.3 What are the major components of 8051 microcontroller ? (Ref. Sec. 3.1.1) **Dec. 15, 3 Marks**

Q. 3.1.4 State the features of 8051 microcontroller. (Ref. Sec. 3.1.1) **May 17, 5 Marks**

- **Harvard Architecture :** Separate Program and Data Memory.
- **High Integration :** Built-in Program (ROM) and Data (RAM) Memory, Timers, Interrupt Control, Serial and Parallel Ports.
- The Important components into the 8051 internal architecture can be expressed as follows :
  - Internal Clock Oscillator driving the system clock generated from the external Crystal Oscillator connected across XTAL1, XTAL2 pins.

- Core MCS-51 CPU : The Processor logic that executes the MCS-51 instruction set Instructions and drive the processor engine.
- Interrupt control logic : Manages Interrupts from 5 Interrupt sources (2 External : #INT0 and #INT1 and 3 Internal – Timer 0, Timer 1 overflow and Serial Interrupt) based on the IE and IP - SFR configured by user.
- 4KB internal ROM used as Internal program memory.
- 128 Byte RAM : Used as internal data memory – Constitutes Register Banks(00-1Fh), Bit Addressable Memory(20-2Fh), User Memory (30-7Fh)
- Two 8/16 Bit Timers : Timer 0 and Timer 1 – Operating in 4 possible modes (Mode 0 to Mode 3) as per the TCON, TMOD, TL0, TH0, TL1, TH1 – SFR configured by the user.
- Asynchronous full duplex Serial Port compliant to UART (Universal Asynchronous Receiver/ Transmitter) operating 1 of 4 Modes based on SCON and SBUF – SFR registers configured by the user.
- 4 Parallel Ports of 8 Bits each – A total of 32 Digital I/O lines – Quasi– bi-directional, General Purpose – some of them with Alternate functions – Accessed by SFR – Port 0, Port 1, Port 3, Port 4.
- Bus Control Logic that controls the operations on the buses and helps 8051 access the external memory.

3.1.2 Accumulator (ACC)

ACC is the accumulator register. It is an 8 bit register.

Function

It is most versatile and holds source operand and receives the result of arithmetic operations including addition, subtraction, integer multiplication, division and Boolean bit manipulations.

Uses

(i) It is also used for data transfer between 8051 and any external memory.

(ii) Several functions like rotate, swap etc. apply specifically on the accumulator.

3.1.3 B Register

Function and use

- The B register is used with the A register for multiplication and division operations.
- For other instructions it is treated as a scratch pad register. (i.e. it has no other function other than as a location where data can be stored.)

3.1.4 Arithmetic and Logic Unit (ALU)

Function

The ALU can perform arithmetic and logic operations on eight bit data. It can perform arithmetic operations like addition, subtraction, multiplication, division and logical operations like AND, OR, EX-OR, complement, rotate etc.

- The ALU also takes care of branching instructions.

3.1.5 Program Status Word (PSW) and Flags

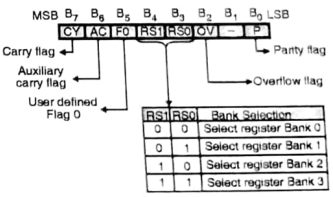
→ (MU - May 15, Dec. 15)

Q. 3.1.5 Explain PSW register of 8051. (Ref. Sec. 3.1.5) **May 15. 5 Marks**

Q. 3.1.6 Explain functions of Program Status Word Register in 8051 microcontroller (Ref. Sec. 3.1.5) **Dec. 15. 6 Marks**

Many instructions affect the status flags. In order to address these flags conveniently they are grouped to form the program status word.

- Flags are 1 bit registers provided to store the status of the result of some instructions.
- A Flag is a flip flop that indicates some condition produced by the execution of an instruction.
  e.g. : The carry flag (CY) will be set if there is a carry or borrow out of the MSB of the result.
- Fig. 3.1.2 shows the PSW. It contains math flags user program flag F0, register select bits that identify the register bank that is currently in use.
- The 8051 has four math flags that include carry (CY), Auxiliary carry (AC), Overflow (OV) and Parity (P). The carry flag will be set when there will be a carry or borrow out of the MSB (D7 bit) of result.
- The auxiliary carry flag is set, whenever there is a carry out of the lower nibble into the higher nibble or whenever there is a borrow from higher nibble into the lower nibble.
- The overflow flag will be set, if an arithmetic overflow has occurred i.e. a significant bit has been lost because the size of the result exceeded the capacity of its destination location.
- The parity flag is set when the result has even parity, i.e. even number of 1's.
- The bits RS0 and RS1 are used to change the bank registers as shown in Fig. 3.1.2.



RS1	RS0	Bank Selection
0	0	Select register Bank 0
0	1	Select register Bank 1
1	0	Select register Bank 2
1	1	Select register Bank 3

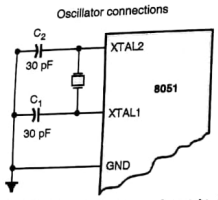
m(19.6)**Fig. 3.1.2 : Program status word (PSW)**

- The 8051 also has three general purpose user flags that can be set to 1 or cleared to 0 by the programmer as desired. The user flags are named F0, GF0 and GF1. The PSW contains user program flag F0, while GF0 and GF1 flags are stored in the PCON register.

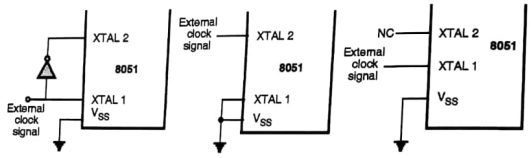
3.1.6 Clock and Oscillator

- The 8051 has an on-chip oscillator, but needs an external clock to run it.
- The oscillator circuit that generates the clock pulses so that all the internal operations are synchronized.

- The pins XTAL 1 and XTAL 2 are provided to connect a resonating network comprising of a crystal and capacitors to form an oscillator. Normally quartz crystal is used. The capacitors are used inorder to stabilize the network. Fig. 3.1.3 shows the oscillator circuit. Generally the capacitors C_1 and C_2 are of 30 pF.
- The crystal frequency is the internal clock frequency of the microcontroller.
- The manufacturers provide 8051 microcontroller designs that run a frequencies ranging from 1 MHz to 16 MHz.
- Fig. 3.1.4 shows how 8051 is driven by an external clock signal.
- Generally 11.0592 MHz crystal oscillators are used so that the 8051 microcontroller systems are compatible with the serial ports. One machine cycle comprises of 12 oscillator periods. Hence, to find the time for 1 machine cycle we need to consider $\frac{1}{12}^{th}$ of the crystal frequency.



m(19.7)**Fig. 3.1.3 : Connection of external crystal to 8051**



m(19.8)**Fig. 3.1.4 : Using an external clock**

3.1.7 Program Counter (PC)

- It is a 16 bit register.

Uses

It is used to hold the address of a instruction in the memory.

Function

Its function is to keep the track of the execution of the program. The program instruction bytes are fetched from locations in memory that are addressed by the Program counter.

- Whenever the power supply is switched on, the PC resets to 0000H. The PC is automatically incremented after fetching a byte from the program memory.

- In case of instructions like jump, call, interrupt the contents of PC may change.
- The PC is only a register. The PC does not have an internal address.

3.1.8 Data Pointer (DPTR)

- The data pointer is a 16 bit register.

Uses

(i) It is used to hold the address of data in the memory.

(ii) It can be used as a 16 bit data register or two independent 8-bit registers.

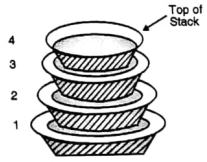
Function

- It serves as a base register in case of instructions handling look up tables and external data transfer.
- The DPTR register can be accessed separately as lower eight bits (DPL) and higher eight bits (DPH).
- The DPTR does not have a single internal address instead DPL and DPH are each assigned a separate address.

3.1.9 The Stack and Stack Pointer

Function

- The stack is a reserved area of the memory in RAM where temporary information may be stored. An 8-bit stack pointer is used to hold the address of the most recent stack entry. This location which has the most recent entry is called as the **top of the stack.**
- When the information is written on the stack, the operation is called **PUSH.** When the information is read from the stack, the operation is called **POP.** The stack works on the principle of **Last In First Out** or **First In Last Out.**
- The microcontroller stores the information/data like stacking plates. Fig. 3.1.5 shows stacked plates. If we want to remove the first stack plate, then we have to remove all the plates above the first i.e. we have to remove the fourth plate, third plate, second plate and then finally the first plate. This indicates that the first plate pushed onto the stack is the last one to be popped from the stack. This operation is called as **First In Last Out (FILO).**
- The stack is implemented with the help of special memory pointer register called as the **stack pointer**. It is of 8 bit.
- This indicates that it can take values from 00 to FFH. On power up, SP contains the value 07H.
- The stack pointer's contents are automatically adjusted to top of the stack. The memory location that is currently pointed by the stack pointer is called as top of stack. As shown in Fig. 3.1.5 the 4^{th} stacked plate represents top of stack.



m(19.9)**Fig. 3.1.5 : Stacked plates**

- When data is to be stored on the stack, the SP increments before storing the data on stack and when the data is to be retrieved/popped from the stack the SP decrements to point next available byte of stored data.
- The stack array can reside anywhere in the on-chip RAM. However, its height is limited to the size of internal RAM.
- The locations 08H to 1FH in the 8051 RAM can be used for the stack. This is because the locations 20-2FH of RAM are reserved for bit-addressable memory and cannot be used by the stack.
- If in a program we need more than 2A bytes of stack, we can change SP to point to RAM locations 30-7FH
- The stack can overwrite data in the register banks, bit-addressable RAM, scratch-pad RAM areas. So, normally the stack is placed at a higher location in internal RAM to avoid conflict with the register and bit addressable internal RAM areas.

☞ **Uses**

(i) The stack can be used to save the register contents.

(ii) The CPU uses the stack to save the address of the instruction just below the CALL instruction. This will indicate the CPU where to resume after it returns from the called subroutine.

3.1.9(A) Stack and Bank 1 Conflict

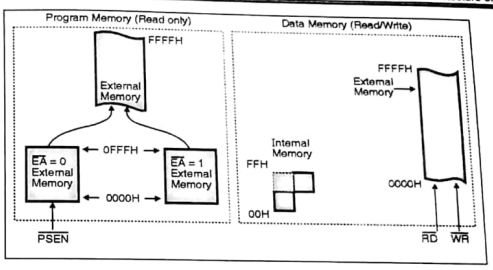
- After a reset, the stack pointer is initialized to 07H. So, the stack will begin at location 08H.
- The stack pointer register points to the current RAM location available from the stack. As data is pushed onto the stack, SP is incremented. Conversely it is decremented as data is popped off the stack into the registers. This is because SP is incremented after the push instruction to confirm that the stack is growing towards RAM location 7FH from lower address to the upper address.
- If the stack pointer is decremented after the push instruction we would be using RAM locations 7, 6, 5 that belong to registers R7 to R0 of bank 0.
- The incrementing of the stack pointer for push instructions confirms that the stack will not reach location 0 at bottom of RAM and stack will run out of space.
- However there is a conflict with default setting of stack. As SP = 07H when 8051 is powered up, the starting location of stack is RAM location 08H. This location belongs to register R0 of register bank 1 i.e. the register bank 1 and stack are using same memory space.

Syllabus Topic : Memory Organisation

3.2 Memory Organisation

☞ **Structure of internal memory**

- Fig. 3.2.1 shows the basic memory structure for microcontroller 8051. It can access upto **64 KB of program memory and 64 KB of data memory.** It has 4 KB of internal program memory and 256 bytes of internal data memory.
- Each memory has a separate addressing mechanism. It also has different control signals and different functions. Each memory will use the same address and data bus, but with different control signals.



m(19.10)**Fig. 3.2.1 : Memory structure of 8051**

☞ **Program memory**

- The microcontroller 8051 has 64 KB external and 64 KB internal program memory. Fig. 3.2.1(a) shows the memory map of program memory.
- If the $\overline{EA}$ pin is connected to V_{cc} then the program fetches the addresses from 0000H through 0FFFH are directed to the internal memory and the addresses from 1000H through FFFFH through the external ROM
- If the $\overline{EA}$ pin is grounded then all the addresses from 0000H through FFFFH are fetched by the 64 KB external ROM/EPROM. The $\overline{PSEN}$ signal activates the output enable signal for ROM/EPROM.

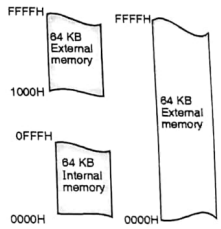


Fig. 3.2.1(a) : 8051 Program memory

☞ **Data memory**

- 8051 microcontroller can address upto 64 KB of external data memory and 256 bytes of internal data memory.
- Fig. 3.2.1(b) shows a memory map of the 8051 data memory.

– The instruction "MOVX" is used to access the external data memory. The internal data memory space is divided into three blocks lower 128 bytes (00H to 7FH), upper 128 bytes (80H to FFH) and the SFRs. The upper 128 bytes and SFRs occupy the same block of address space, although both are separate.

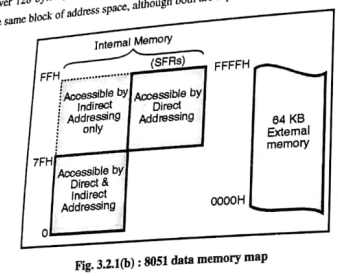


Fig. 3.2.1(b) : 8051 data memory map

3.2.1 Internal Memory Organization

→ **(MU - Dec. 14)**

Q. 3.2.1 Explain internal memory organization of 8051. **Dec. 14, 10 Marks**
(Ref. Sec. 3.2.1)

– For the proper operation of a computer system, the system should have memory for program code. This memory for program code is normally in the ROM.

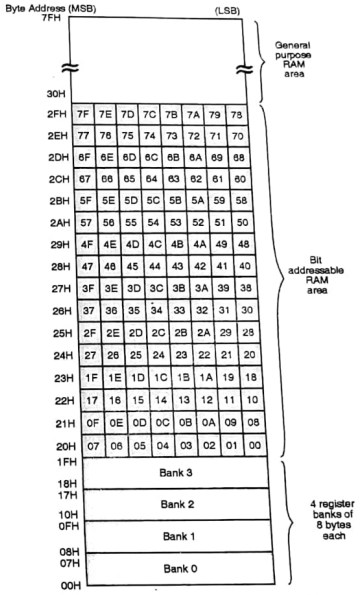
– The 8051 microcontroller has an internal RAM and ROM. If this memory is insufficient, then additional memory is externally added using suitable circuits.

3.2.1.1 Internal RAM

– The 8051 microcontroller has a **128 byte Internal RAM**. This internal RAM is organized in three distinct areas. They are :

(i) Four register banks of 8 bytes each
(ii) Bit addressable area of 16 bytes
(iii) General purpose RAM area (scratch pad area)

Fig. 3.2.2 shows the internal RAM structure with memory map.



m(19.11)**Fig. 3.2.2 : Internal RAM structure**

3.2.1.1(A) Four Register Banks of 8 Bytes each

→ **(MU - Dec. 16)**

Q. 3.2.2 Write note on 8051 register bank. **Dec. 16, 10 Marks**
(Ref. Sec. 3.2.1.1(A))

– There are four register banks which are numbered 0 to 3. Each bank is made up of eight registers named R0 to R7. In total 32 bytes or 32 working registers from address 00H to 1FH organized as four register banks.

– RAM locations 00H to 07H are set aside for Bank 0 of R0 – R7, R0 is RAM location 00H, R1 is RAM location 01H, R2 is RAM location 02H and so on.

– The second bank of registers begins at 08H – 0FH. The third bank of R0 – R7 begins at 10H – 17H and finally the fourth bank of R0 – R7 at RAM locations 18H – 1FH as shown in Fig. 3.2.3.

– For selecting a register bank two bits *RS0* and *RS1* are provided in the program status word (PSW) as shown in Fig. 3.1.2.

– If any of the register banks is not selected, then the programmer can use the area from 00H to 1FH as simple scratch pad RAM or general purpose RAM.

– Upon power on or reset BANK 0 is selected by default.

☞ **Disadvantage of register banks**

– We know that the register bank 1 uses the same RAM space as the stack.

– This is a major problem while programming 8051. We must either not use the register bank 1 or allocate another area of RAM for stack.

☞ **Advantage of register banks**

– The advantage of Register banks is that it reduces the latency period for a subroutine call or an interrupt.

– When the programmer is using a set of registers R0 to R7 of bank 0, and an interrupt occurs :

(i) For a normal processor (without register banks), the ISR or subroutine begins with pushing the contents of all the registers onto the stack. This avoids overwriting of the data of the caller function. At the end of ISR or subroutine all the contents of these registers are to be popped. This push and pop of data is the extra work the processor has to do and the time required to do this is called as latency period.

(ii) In case of 8051, the programmer need not push the data when an interrupt occurs or when a subroutine is called. Instead the programmer can switch to another set of registers (register bank). And hence the programmer need not even pop the data from stack. This saves the latency period and hence improves efficiency of the system.

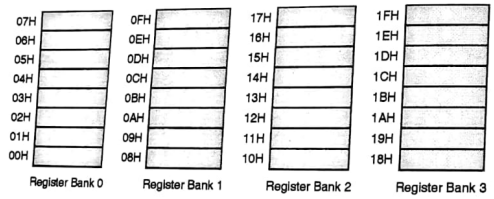


Fig. 3.2.3 : 8051 register banks and their RAM addresses

3.2.1.1(B) Bit Addressable Area of 16 Bytes

– The microcontroller has reserved 16 bytes of internal RAM whose address ranges from 20H to 2FH. These 16 bytes provide us with $16 \times 8 = 128$ bits forming addressable bits.

– The microcontroller has given addresses to these bits ranging from 0 to 127 (decimal) or 00H to 7FH. Hence, these locations are called **bit addressable locations** as shown in Fig. 3.2.2.

☞ **Use**

– The addressable bits are useful if a binary event in the program needs to be remembered. e.g. open switch, close switch etc.

– Inorder to access the 128 bits of RAM locations and other bit addressable space of 8051, we can use only the single bit instructions. All the single bit instructions support only direct addressing mode.

– For single bit instructions there is no indirect addressing mode.

3.2.1.1(C) General Purpose RAM Area

– This RAM area is also called as **scratch pad area**.

– It lies above the bit addressable area and has address 30H to 7FH. This RAM area can be used as data RAM. The memory in this area is byte addressable.

– The programmer may declare stack in this area, provided sufficient number of bytes are available.

3.2.1.2 Uses of Internal RAM

– It is not essential to utilise the entire area of internal RAM of the microcontroller as scratch pad.

– The microcontroller has instructions that allow the internal memory locations to be used as data pointers.

– The registers R0 to R1 in the four registers banks (Bank 0 to Bank 3) can be used as a pointer to point the internal RAM as well as the external RAM. When the registers R0 or R1 are used to access the external RAM, these 8 bits will be treated as the LSB of 16 bit address and the MSB is taken to be 00H by default.

3.2.1.3 Internal ROM

The microcontroller 8051 has a separate memory for the program and data. Both these memories have same address ranges.

☞ **Function**

– The internal ROM is used to store the internal program code. It occupies the address space ranging from address 0000 H to 0FFF H.

– The PC can access program code bytes from 0000 H to FFFF H. The capacity of internal ROM is from 0000 H to 0FFF H. For addresses higher than 0FFF H the 8051 will automatically fetch the program code bytes from an external memory.

– The PC is not bothered of about where the program code resides, the programmer decides whether the code resides in the internal memory, external memory or combination of the internal and external memory.

3.2.2 External Memory

– External memory is used in cases when the internal ROM and RAM memory available on chip is not sufficient. Two separate external memory spaces are made available by the 16-bit PC and the DPTR and by different control pins for enabling external ROM and RAM chips.

– If the 128 bytes of internal RAM is insufficient, then external RAM is accessed by the DPTR. In the 8051 family, external RAM of upto 64 KB can be added to any chip.

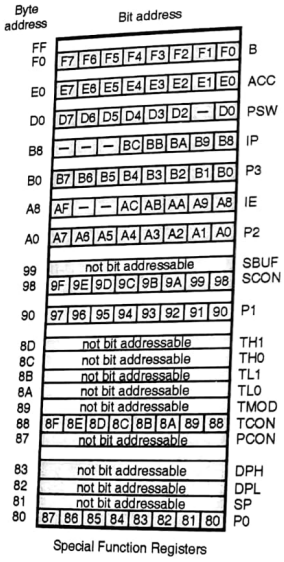
– The external ROM of upto 64 KB can be added to any chip in the 8051 family.

3.2.3 Special Function Registers (SFRs)

Q. 3.2.3 What are special function registers ? Explain their utility with a few examples. **(5 Marks)**
(Ref. Sec. 3.2.3)

Q. 3.2.4 What are 'SFRs' in 8051 microcontroller ? Explain their utility. (Ref. Sec. 3.2.3) **(5 Marks)**

- **Special function registers are placed in the address space immediately above the 128 bytes of RAM, from address 80H to FFH.**
- **The SFR memory consists of important registers like accumulator, B register, interrupt control registers, PSW, timer / counter, power control, four I/O ports, serial control. Some of these registers are bit addressable while remaining are byte addressable.**
- Fig. 3.2.4 shows the SFR structure.



m(19.12)**Fig. 3.2.4 : SFR Structure**

- Some of the addresses i.e. locations in between 80H and FFH are not used. If we try to use one of these unused locations that are not defined or are empty, then we may get unpredictable results. When reading from such an unused location a random data will be given. When writing the data to such unused location the data will be lost, i.e. not stored anywhere.

- The PC is not a part of the SFR. The PC (program counter) does not have an internal RAM address.
- SFRs are referenced by their addresses such as 0E0 H, 87 H, 90 H, 0A0 H, 80 H etc. or by names.
  e.g. Accumulator has address 0E0H, register B has address 0F0H, SCON register has address 98H etc.

3.2.4 CPU Timing and Machine Cycle

- A machine cycle consists of sequence of 6 states, numbered S1 through S6. Each state time lasts for two oscillator periods i.e. each state has period 1 and period 2. Therefore, one machine cycle takes 12 oscillator clock periods.
- Each machine state is divided into two period : period 1 and period 2. During period 1, period 1 clock is active and period 2 clock is active during period 2. The machine cycle states will be numbered as S1P1 (State 1, period 1), S1P2 (State 1, period 2) ..... S6P2 (State 6, period 2).
- Each period lasts for one oscillator period. Typically, arithmetic and logical operations occur during period 1 and register to register transfers occur during period 2. Fig. 3.2.5 shows the fetch 1 execute sequence in states and phases for different instructions.

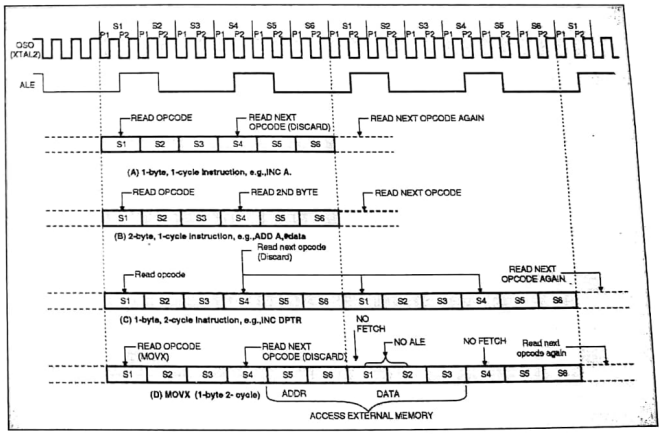


Fig. 3.2.5 : State sequence in MCS-51 device

- In case of one byte two cycle instruction e.g. MOVX. The MOVX instruction accesses external data memory. Two fetches are skipped while the external data memory is being addressed and strobed. Fig. 3.2.5 shows the timing diagram.

Program 3.2.1 : If oscillator frequency of 8051 is 10 MHz, then what is the time required for one machine cycle ?

Solution : Clock frequency = $\frac{1}{12}$ × oscillator frequency

$\therefore$ Clock frequency $= \frac{10}{12} = 0.833$ MHz.

$\therefore$ Time required for one machine cycle $= \frac{1}{0.833\ MHz} = 1.2\ \mu s$.

$\therefore$ **One machine cycle = 1.2 μs.**

3.2.5 Interfacing of External Memories in 8051

- External memories are interfaced with 8051 using the four control signals ($\overline{PSEN}$, ALE, $\overline{RD}$ and $\overline{WR}$) and $AD_0 - AD_7$ bus at port 0 and $A_8 - A_{15}$ at port 2.
- Fig. 3.2.6 shows the interfacing circuit.

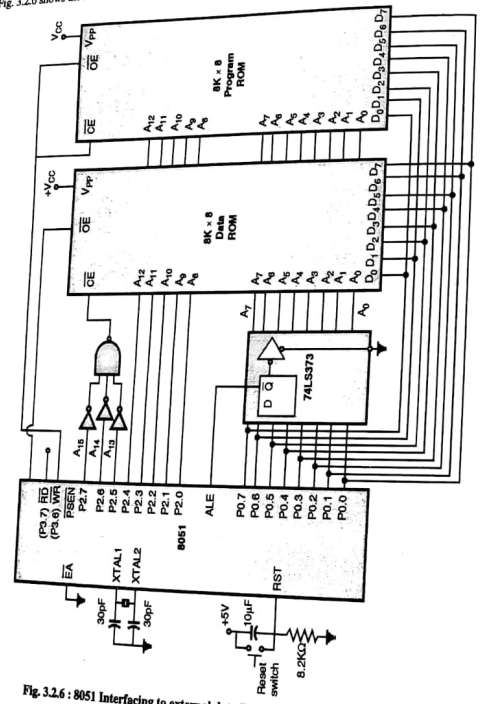


Fig. 3.2.6 : 8051 Interfacing to external data ROM and external program ROM

Fig. 3.2.7 shows the clock states.

(i) ALE signal (1) separates the lower address bus $A_0 - A_7$ for program and data memories. An 8 bit latch 74LS37S is used for demultiplexing the lower address bus and data bus $AD_0 - AD_7$.

(ii) $\overline{PSEN}$ signal when low, uses the program memory code bank 2 to 31 for code reading operation. It uses bank 0 and bank 1. $\overline{EA}$ is low during the microcontroller reset.

(iii) $\overline{RD}$ when low, uses data memory for external data reading operation.

(iv) $\overline{WR}$ when low, uses data memory for data write operation.

(v) When $\overline{PSEN}$ and $\overline{RD}$ are short circuited, the program memory and the data memory areas overlap. The codes and data can reside in unified space.

3.2.6 Timing Diagrams for Memory Interfacing

- We know that an instruction cycle there are 12 clock states that are spread over two cycles S_1 - S_6.
- A state is for a period of 12 times the oscillator clock period that is equal to the reciprocal of XTAL frequency.

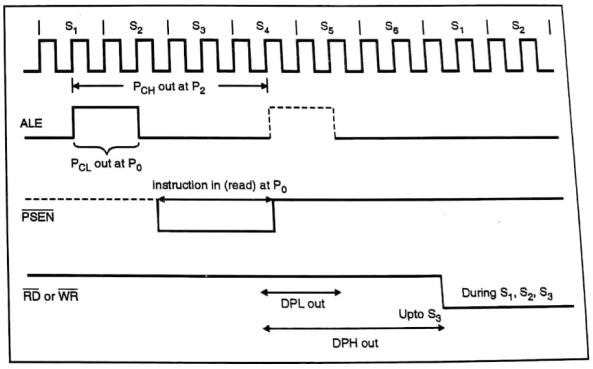


Fig. 3.2.7 : Timing diagram when external data and program memory interfaces with 8051

Fig. 3.2.7 shows the clock states and timing diagram to reflect the sequence of bus operations.

(1) During the period between the middle of two clock states S_1 and S_2, period first ALE signal demultiplexes the lower order address bus $A_0 - A_7$.

(2) During a period nearly end of clock state, S_3 and middle of S_4, $\overline{PSEN}$ signal when low, microcontroller reads the program memory code.

(3) During three clock periods between beginning of S_1 and end of S_3, the $\overline{RD}$ when 0, uses data memory for read. This S_1 and S_2 are after the first cycle of six clock states $S_1 - S_6$.

(4) During the three clock periods between the beginning of S_1 and end of S_3, $\overline{WR}$ when low uses data memory for write. This S_1 and S_2 are after the first cycle of six clock states $S_1 - S_6$.

(5) Just before the ALE is high and just after ALE becomes low, P_0 has PCL buffer bits. Just before the ALE is 1 at S_1 and upto just before the next time, another ALE activation at S_1, the P_2 has PCH buffer bits.

(6) An instruction is read is between the period of $\overline{PSEN} = 0$ and between the middle of S_3 to just before the start of next ALE at S_1.

(7) Just after beginning of S_4, the DPL is out for one state period before ALE = 1 at S_4 (middle of S_4 to middle of S_5). During the write, the data is at P_0 after S_5 upto just after the end of S_2 in next S_1, S_2 and S_3 when $\overline{RD}$ or $\overline{WR}$ is low.

(8) Just after the beginning of S_4, the DPH is out for periods (middle of S_4 to end of S_3) is next S_1, S_2 and S_3 when $\overline{RD}$ or $\overline{WR}$ is low.

3.2.7 Time for Execution of an Instruction

Instruction cycle helps in calculating the time required for executing an instruction. e.g. : A two cycle instruction will need 2 μs. It is because one-cycle frequency is $\left(\frac{1}{12}\right) f_{xtal}$, where f_{xtal} is the XTAL oscillation frequency. (Assuming crystal frequency 12 MHz)

Adding the total number of instruction cycles in a program that will take on execution gives the total time.

Syllabus Topic : Input / Output Ports

3.3 Input / Output Ports

Q. 3.3.1 Explain the physical structure of I/O ports of the 8051 microcontroller. (Ref. Sec. 3.3) **(5 Marks)**

- As functions are multiplexed on same port pins, in order to decide which function is supported we need to see how the circuit is connected and what software commands are used to "program" the pin.
- The microcontroller has four ports named P0, P1, P2 and P3. All these ports are bi-directional. Each of these 8 bit ports consists of a D-type output latch, an output driver and an input buffer.

Now let us see the ports one by one.

3.3.1 Port 0

- Port 0 is a multifunctioned port of microcontroller 8051. It can be used as simple input / output mode or for generating data and lower order address bus for external memory ($AD_0 - AD_7$). Fig. 3.3.1 shows port 0 circuit.
- Port 0 does not have an internal pull up, hence whenever port 0 is configured as an output port external pull up is required.

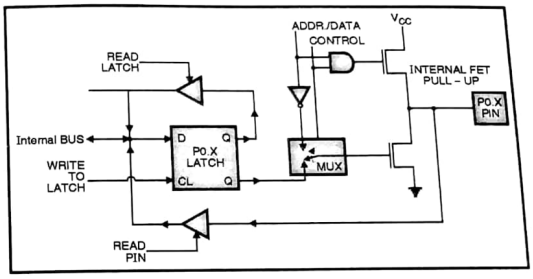


Fig. 3.3.1 : Port 0 circuit

☞ **Port 0 as simple input port**

- When port 0 is used as an input port, a '1' must be written to the corresponding port 0 latch that will cause both the output transistors to switch off and the pin **"floats"** in a high impedance state. Hence it is called as **"true bidirectional"** port because when configured as an input port it floats. When configured as input port the microcontroller provides two facilities :
  - (i) Read logic level on physical pin by asserting READ PIN signal.
  - (ii) Read the contents of internal latch by asserting the READ LATCH signal. The latch is read when the instruction is a Read-Modify-Write type of instruction. A Read-Modify-Write instruction is one, wherein the instruction **Reads** the data from the port (latch) **Modifies** (performs some operation on it) and **Writes** to the port (latches and hence pins).

☞ **Port 0 as simple output port**

- When port 0 is configured as an output port, the latch pins that are programmed to 0 will cause the lower FET to turn on and pin is grounded.
- If a '1' is written on to the latch pin the FET will turn off and the pin is pulled HIGH by external pull up resistors.

☞ **Port 0 used as multiplexed address / data bus ($AD_0 - AD_7$) for external memory**

- When micro-controller accesses external program memory or data memory the address for memory is generated by port 0 and port 2.
- Port 0 generates the lower order address $A_7 - A_0$, while port 2 generates the higher order address $A_8 - A_{15}$.
- When port 0 is used as an address bus to the external memory, the internal control signals switch the address lines to the gate of FETs.

– If a logic 1 is written onto the address bit, then the upper FET will turn on and the lower FET will turn off providing a logic HIGH at the pin. If a logic 0 is written onto the address bit, then the upper FET will turn off and the lower FET will turn on providing a logic LOW at the pin. Once, the 8 bit address is been formed and latched by the Address Latch Enable (ALE) pulse into the external circuits, the bus turns around to data bus. Port 0 can now read data from the external memory. For reading data it must be configured as a input port. Fig. 3.3.2 shows multiplexed address / data bus.

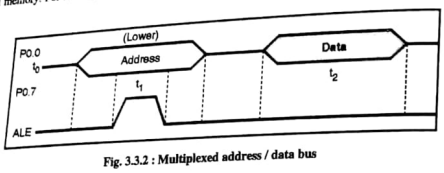


Fig. 3.3.2 : Multiplexed address / data bus

3.3.2 Port 1

– Port 1 is a simple I/O port of microcontroller. It does not have any extra function. Hence, the output latch is connected directly to the gate of the lower FET, consisting of a circuit labelled internal pull up. Fig. 3.3.3 shows port 1 circuit.

– When port 1 is used as an input port, a "1" must be written to the corresponding port 1 latch bit. This causes the lower FET to turn off. The pin and input to pin buffer are pulled to logic HIGH by the internal pull up load.

– Port 1 is called as **"quasi-bidirectional"** port as its output is pulled high with pull up resistors.

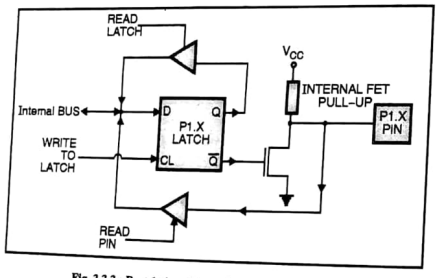


Fig. 3.3.3 : Port 1 circuit Port 1 as simple input port

☞ Port 1 as simple output port

– When port 1 is used as an output port, the latch pins that are programmed to 0, will cause the lower FET to turn on, the internal pull up to turn off and input to the circuit is logic 0.

– If a '1' is written onto the latch pin then it will drive the input of external circuit high through the pull up. The lower FET turns off.

– The internal FET pull up is used to help the port 1 to speed up when it is used as an output port. The internal FET pull up has another FET that is in parallel to lower FET. This FET is turned on for two clock periods when the transition on the pin is low to high. This arrangement provides a low impedance path to the positive supply voltage.

3.3.3 Port 2

– Port 2 of the microcontroller 8051 is a multifunctional port. It can be used as a simple input / output port or for generating the upper order address bus for external memory ($A_8 - A_{15}$) Fig. 3.3.4 shows port 2 circuit.

Port 2 circuit contains D-type latch, multiplexer, two unidirectional buffer and FET output stage with pull up.

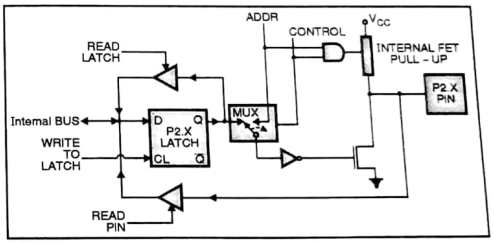


Fig. 3.3.4 : Port 2 circuit

☞ Port 2 as simple input port

– When port 2 is used as an input port, a "1" must be written to the corresponding port 2 latch bit. This causes the FET to turn off. The pin and input to pin buffer are pulled to logic HIGH by the internal pull up load.

– Port 2 is called as **"quasi-bi-directional port"** as its output is pulled high with pull up resistors.

☞ Port 2 as simple output port

– When port 2 is used as an output port, the latch pins that are programmed to 0, will cause the lower FET to turn on, the internal pull up to turn off and input to the circuit is logic 0.

– If a "1" is written onto the latch pin then it will drive the input of external circuit high through the pull up. The lower FET turns off.

☞ Port 2 used as higher order address bus ($A_8 - A_{15}$) for external memory

– The port 2 pins are momentarily changed, due to the address control signals when it is supplying the higher order address. The latch remains stable because it does not have to turn around for data input as in port 0.

– Fig. 3.3.5 shows how port 2 generates address in conjunction with ALE.

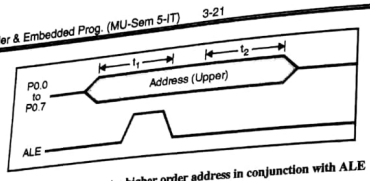


Fig. 3.3.5 : Port 2 generates higher order address in conjunction with ALE

3.3.4 Port 3

- Port 3 is a multifunctioned port it can be used as a simple input / output port. The port 3 pins have special functions. These functions are listed in Table 3.3.1.

Table 3.3.1 : Function of port 3 pins

PIN	Symbol	SFR	Significance
P 3.0	RXD	SBUF	It is the receive data pin for serial port in UART mode.
P 3.1	TXD	SBUF	It is the transmit data pin for serial port in UART mode. It works as the clock output in the shift register mode
P 3.2	$\overline{INT0}$	TCON.1	It is an external interrupt. It is low level or falling edge triggered.
P 3.3	$\overline{INT1}$	TCON.3	It is an external interrupt. It is low level or falling edge triggered.
P 3.4	T0	TL0	External Timer / Counter 0 input pin, gives pulses to TL0 register of the timer 0 to increment by 1
P 3.5	T1	TL1	External Timer / Counter 1 input pin, gives pulses to TL1 register of the timer 0 to increment by 1
P 3.6	$\overline{WR}$	-	It is external memory write pulse. It is an active low pulse.
P 3.7	$\overline{RD}$	-	It is an external memory read pulse. Whenever data from memory is read this pulse is active low.

- Unlike the ports 0 and 2, where all the 8 bits simultaneously change for alternate use, each bit of port 3 can be programmed as I/O of to perform one of the functions listed above.
- Fig. 3.3.6 shows port 3 circuit.
- Port 3 bit contains D type latch, three unidirectional buffer, FET with internal pull-up. As the internal pull up is fixed port 3 is called as **"quasi-bidirectional"** port.

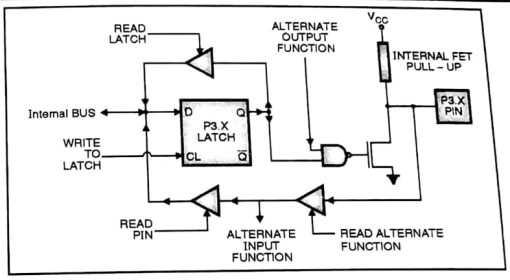


Fig. 3.3.6 : Port 3 circuit

☞ **Port 3 as simple input port**

When port 3 is used as an input port, a "1" must be written to the corresponding port 3 latch bit. This causes the FET to turn off. The pin and input to pin buffer are pulled to logic HIGH by internal pull up load.

☞ **Port 3 as simple output port**

- When port 3 is used as an output port, the latch pins that are programmed to 0, will cause the lower FET to turn on, the internal pull up to turn off and input to the circuit is logic 0.
- If a "1" is written onto the latch pin then it will drive the input of external circuit high through the pull up. The lower FET turns off.

☞ **Port 3 as one of the alternate functions**

- For achieving any one of the alternate output functions, another control signal called as "alternate output function" is available on port 3. Depending on the logic level present on line "alternate output function" the FET will turn ON or OFF.
- When the latch bit of port 3 is at logic 1, the output level is controlled by the control input.

☞ **Port loading and interfacing**

- The output buffers of ports 1,2 and 3 can each drive 4 LS TTL inputs. Port 0 when used in external bus mode output buffer can drive 8 LS TTL inputs.
- In order to drive inputs these ports as port pins require external pull ups to drive any inputs.

Syllabus Topic : Counters and Timers

3.4 Counters and Timers

➔ **(MU - Dec. 15, May 16, Dec. 16, May 17)**

Q. 3.4.1 Explain functions of Timer Control Register in 8051 microcontroller. (Ref. Sec. 3.4) **Dec. 15. 3 Marks**

Q. 3.4.2 Explain the Timer/Counter modes of 8051 microcontroller. (Ref. Sec. 3.4) **May 16, May 17. 10 Marks**

Q. 3.4.3 Explain the following SFR's of 8051 : TCON, TMOD (Ref. Sec. 3.4) **Dec. 16. 5 Marks**

Q. 3.4.4 Discuss the internal timer / counter of 8051 microcontroller with appropriate diagrams. (Ref. Sec. 3.4) **(5 Marks)**

- The microcontroller 8051 has two 16 bit Timer / Counter registers namely Timer 0 and Timer 1. Both these registers can be configured independently to operate as timer or an event counter.
- When used as a "Timer" the register is programmed to count the internal clock pulse. The internal clock pulses are generated from a constant clock generator, the count loaded in the register gives constant time. The register is incremented every machine cycle. One machine cycle consists of 12 oscillator periods and so the counting rate is 1/12 of the oscillator frequency. When used as a "Counter", the microcontroller is programmed to count external pulses. The register is incremented in response to a high to low transition (↘) of the corresponding external input pin, T0 and T1. The external input does not have a constant frequency and hence it is not used for timing reference.
- The T0 and T1 pins are sampled during S5P2 of every machine cycle. When the processor finds it to be '0' in one machine cycle and '1' in another machine cycle, then it increments the timer/counter register in S3P1 of the next machine cycle.
- Hence, in order to recognize the high-low transition the microcontroller requires two machine cycles i.e. 24 oscillator periods. The maximum count rate is 1/24 of the oscillator frequency.
- There are no restrictions on the duty cycle of the external input signal, but it should be held high atleast for one machine cycle, to ensure that the input is sampled atleast once before it changes.
- The timer mode can also be used for pulse width measurement. When the gate bit is kept '1', the timer runs until the $\overline{INTx}$ pin is high. Hence the timer is counting the number of internal clock pulses for which the pulse on $\overline{INTx}$ pin is at logic '1'. For e.g. if the timer counts from 1 to 10 and the crystal is of 12 MHz (i.e. one machine cycle is 1μsec), it indicates the pulse on $\overline{INTx}$ pin was at logic '1' for 10 μsecs.
- The counter / timer registers are divided into 8 bit registers called the timer low (TL0 and TL1) and timer high (TH0 and TH1). TL0 and TH0 together form the 16 bit Timer 0 and TL1 and TH1 forms the 16 bit Timer 1.
- The counter action is controlled by the bits in the timer mode control register (TMOD) and the timer / counter control register (TCON) and some instructions.

- Fig. 3.4.1 shows TMOD register and Fig. 3.4.2 shows the TCON register

(MSB)							(LSB)
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
Timer 1				Timer 0			

GATE — Gating control. When set Timer/Counter "x" is enabled only while "$\overline{INTx}$" pin is high and "TRx" control pin is set. When cleared, Timer "x" is enabled whenever "TRx" control bit is set. It is used for Pulse width measurement (PWM) of the pulse on $\overline{INTx}$ pin.

C/$\overline{T}$ — Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin)

M1	M0	Operating Mode
0	0	8-bit Timer/Counter "THx" with "TLx" as 5-bit prescalar
0	1	16-bit Timer/Counter "THx" and "TLx" are cascaded, there is no prescaler
1	0	8-bit auto-reload Timer/Counter. "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
1	1	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits. TH0 is an 8-bit timer only controlled by Timer 1 control bits.
1	1	(Timer 1) Timer/Counter 1 stopped

GATE is used to program timers in PWM mode.

C/$\overline{T}$ will select either counter operation or timer operation.

C/$\overline{T}$ = 0 → Timer; C/$\overline{T}$ = 1 → Counter.

Mode bits are going to select different modes of operation of the timer.

Fig. 3.4.1 : Timer / Counter mode control register

(MSB)							(LSB)
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
Timer 1		Timer 0		$\overline{INT1}$		$\overline{INT0}$	

Symbol	Position	Name and Significance
TF1	TCON.7	Timer 1 overflow Flag. Set by hardware on timer Counter overflow. Cleared by hardware when processor vectors to interrupt routine

Symbol	Position	Name and Significance
IE1	TCON.3	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.

Symbol	Position	Name and Significance
TR1	TCON.6	Timer 1 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TF0	TCON.5	Timer 0 overflow Flag. Set by hardware on Timer counter overflow. Cleared by hardware when processor vectors to interrupt routine.
TR0	TCON.4	Timer 0 Run control bit. Set/ cleared by software to turn Timer/Counter on/off.
IT1	TCON.2	Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
IE0	TCON.1	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed
IT0	TCON.0	Interrupt 0 Type control bit Set/ cleared by software to specify falling edge/low level triggered external interrupts.

Fig. 3.4.2 : TCON Timer / Counter control register

– As shown in Fig. 3.4.2(a), the Timer is enabled when TR = 1 and GATE = 0. If the GATE = 1, then the timer is controlled by external input $\overline{INTx}$, to allow pulse width measurement. TR is control bit in the TCON register.

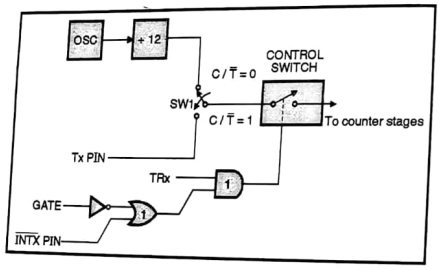


Fig. 3.4.2(a) : Timer / Counter control circuit

Timer modes of operation

Depending on the mode control bits M1 and M0 in the Timer / Counter mode control (TMOD) register can operate in any one of the four modes : Mode 0, Mode 1, Mode 2, and Mode 3.

Mode 0

– In this mode the timer operates as a 13 bit register (TL – 5 bits and TH – 8 bits). The 5 bits in the TL sets the divide by 32 prescalar to the TH as an 8 bit counter.

– Fig. 3.4.3 shows the Timer / Counter in mode 0.

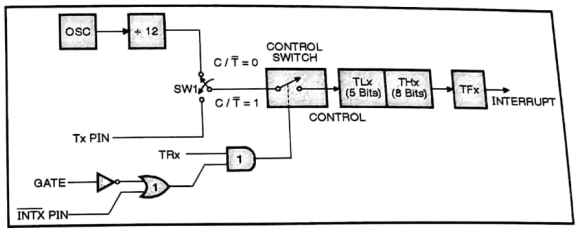


Fig. 3.4.3 : Timer / Counter mode 0 : 13 bit counter

Interrupt

– If the user desires some count in the register and gives command by TRx bit the timer / counter will start. If the timer count exceeds 13 bit i.e. 1FFFH the next count will be 0000H.

– It causes the microcontroller to generate an interrupt in order to inform the programmer whether one cycle is over.

– To indicate it, Timer overflow bit (TF1 and TF0) will be set.

– If the timer interrupt is enabled using the Interrupt Enable register (IE), the controller will vector to ISR routine.

Mode 1

Mode 1 of the Timer / Counter is same as mode 0. The only difference is that in mode 0 the timer / counter was 13 bit timer / counter, but in mode 1 it is a 16 bit timer / counter.

Mode 2

– In this mode the Timer / Counter register is configured as an 8 bit counter (TL) with auto reload facility. Fig. 3.4.4 shows Timer / Counter 1 in mode 2.

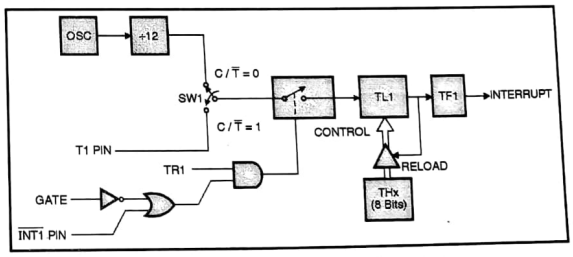


Fig. 3.4.4 : Timer / Counter 1 mode 2 – 8 bit auto reload

- TL1 acts as the basic timer / counter. When the timer starts working, it keeps on incrementing and it will overflow. This will set the timer interrupt flag TF1 and it also reloads the contents of TH1 to TL1. The reloading operation will not alter the contents of TH1.

Mode 3

- In mode 3 Timer 1 is not used. Timer 1 therefore simply holds its count. The timer 0 registers TL0 and TH0 are configured as two separate 8 bit counters. Fig. 3.4.5 shows Timer / Counter 0 mode 3.
- The TL0 register uses the Timer 0 control bits $C/\overline{T}$, GATE, TR0, $\overline{INT0}$ and TF0. TH0 counts the machine cycles. It takes over the use of TR1 and TF1 from Timer 1.
- When the Timer 0 is in mode 3, the Timer 1 may be used in modes 0, 1, and 2. In this case no interrupts will be generated because Timer 0 is using the TF1 flag.
- When the Timer 0 is in mode 3, Timer 1 can be used as a baud rate generator for serial port.

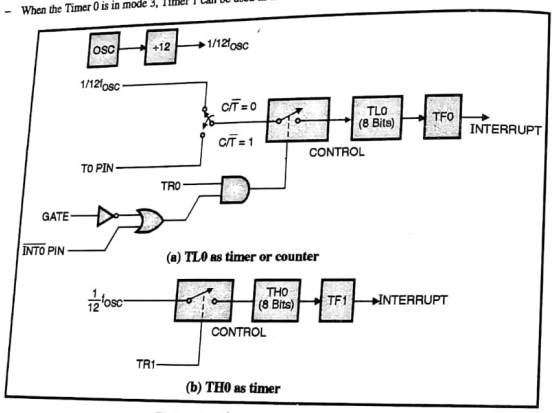


Fig. 3.4.5 : Timer / Counter 0 mode 3

Syllabus Topic : Serial Communication

3.5 Serial Data Input and Output

→ **(MU - May 15, Dec. 15,May 16, Dec. 16, May 17)**

Q. 3.5.1 Explain various modes of operation of serial port in 8051. (Ref. Sec. 3.5) **May 15, 10 Marks**

Q. 3.5.2 Write short note on : Serial Port Communication in 8051. (Ref. Sec. 3.5) **Dec. 15,Dec. 16, May 17, 6 Marks**

Q. 3.5.3 Explain various serial modes of 8051 microcontroller. (Ref. Sec. 3.5) **May 16, 10 Marks**

Q. 3.5.4 Explain the following SFR's of 8051 : SCON. (Ref. Sec. 3.5) **Dec. 16, 3 Marks**

Q. 3.5.5 Discuss the modes of operation of the serial port of the 8051 microcontroller. (Ref. Sec. 3.5) **(5 Marks)**

- For communication between two computer systems we can send and receive the data bits serially.
- The microcontroller 8051 supports full duplex serial communication. A full duplex asynchronous serial interface can be implemented on using serial interface control circuitry.
- It has serial data communication circuit which uses a register in the SFR called SBUF to hold the data. The register SCON controls the data communication, the register PCON along with timer 1 controls the data rates.
- The SBUF register comprises of two registers physically; one of them is write only and is used to hold the data that is to be transmitted out from the microcontroller via the TXD pin, while the other is read only and holds the data that is received from the external sources via the RXD pin.
- A double buffered receiver is used in the circuit so that the receiver can receive a second character when the first one is in an intermediate register. Double buffering reduces the chances of an overrun error and complexity. But if the previous byte is not read when and the next byte is completed its reception, the first byte received will be lost.
- Serial data communication is a relatively slow process. In order not to tie up with valuable processor time, serial data flags are included in SCON to aid in efficient data transmission and reception. The serial data flags in SCON, TI and RI, are set whenever a data byte is received (RI) or transmitted (TI).
- These flags are ORed together to produce an interrupt to the program, indicating that the byte is received / transmitted and hence to get ready for next byte (i.e. read the byte from SBUF in case of reception or write the next byte into the SBUF for serial transmission). The program must read these flags to find out which bit has created the interrupt and clear the bit.
- Transmission of serial data bits begins anytime data is written in to SBUF. TI is set to a 1 when the data has been transmitted and signifies that the SBUF is empty and another byte can be sent.
- Reception of serial data will begin if the receive enable bit (REN) in SCON is set to 1 for all modes. In addition, for Mode 0 only, RI must be cleared to 0. RI is set to 1 when a byte is received in all modes. REN is the only program control to prevent or to receive the serial data.
- The serial interface can be operated in four different modes that can be configured using SCON (Serial port control status) register. Fig. 3.5.1 shows the SCON register.
- The 4 modes behave as

Mode 0 : Shift Register with fixed baud rate
Mode 1 : 8 bit UART with variable baud rate
Mode 2 : 9 bit UART with fixed baud rate.
Mode 3 : 9 bit UART with variable baud rate.

(MSB)							(LSB)
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Where SM0, SM1 specify the serial port mode, as follows :

SM0	SM1	Mode	Description	Baud Rate
0	0	0	Shift register	f_{osc} / 12
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	f_{osc} / 64 or f_{osc} / 32
1	1	3	9-bit UART	variable

- *SM2 enables the multiprocessor communication feature in Modes 2 and 3. In Mode 2 or 3, if SM2 is set to 1 then RI will not be activated if the received 9th data bit (RB8) is 0. In Mode 1, if SM2 = 1 then RI will not be activated if a valid stop bit was not received. In mode 0, SM2 should be 0.*
- *REN enables serial reception. Set by software to enable reception. Clear by software to disable reception.*
- *TB8 is the 9th data bit that will be transmitted in modes 2 and 3. Set or clear by software as desired.*
- *RB8 in modes 2 and 3, is the 9th data bit that was received. In Mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, RB8 is not used. (concept of stop bit will be dealt with later in this section).*
- **TI** is transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must be cleared by software.
- **RI** is receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

Fig. 3.5.1 : SCON – serial port control / status register

- The modes 2 and 3 have special provision for multiprocessor communication i.e. we can have master / slave configuration.
- Baud rate for serial mode can be adjusted and the clock source required for setting baud rate is provided on chip. The Baud rate is fixed for mode 0, while it is variable for modes1, 2 and 3. Variable baud rates are determined by the Timer 1 overflow rate.

Mode 0

- This mode is a half duplex **synchronous** mode. It is referred as the shift register. The shift register has shift left, shift right operation. The shifting operation of data bits is sequentially done.
- The input to the shift register is data and clock. The serial data is transmitted and received through the RXD line. The clock is generated by the TXD line. The TXD shift clock is a square wave, low for states S3, S4 and S5 of the machine cycle, while high for S6, S1 and S2 as shown in Fig. 3.5.2.
- As eight bit are transmitted at a time, the start and stop bits are not required. The LSB of data is transmitted and received first.
- The Baud rate for mode 0 is fixed.

$$\text{Baud rate} = \frac{\text{oscillator frequency}}{12}$$

- The microcontroller chip has limited number of I/O lines available. In order to increase the I/O lines we can provide external I/O chips like shift register.

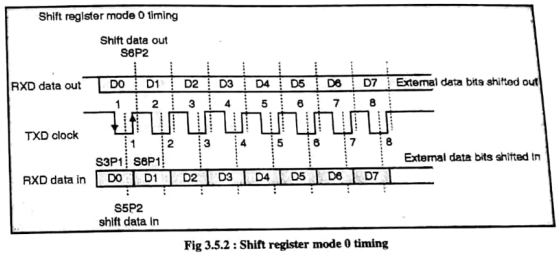


Fig 3.5.2 : Shift register mode 0 timing

Mode 1

- It is a full duplex mode. It supports 8 bit asynchronous communication. Whenever there is a change in data 1 is transmitted, otherwise 0 is transmitted.
- Although the data bits are 8 bit, the number of transmitted bits are 10 i.e. 1 start bit, 1 stop bit and 8 data bits.
- Whenever a character is serially transmitted the transmitting and receiving device should satisfy this communication protocol.
- Fig. 3.5.3 shows the UART data word.

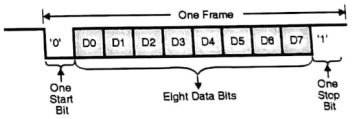


Fig. 3.5.3 : Transmission on of 1 byte, formed by 1 start and 1 stop bit

- The baud rate for mode 1 is variable. The baud rate is determined by timer 1 overflow rate. Typically Timer 1 is used in mode 2 as an auto reload 8 bit timer.

$$\text{Baud rate} = \frac{2^{SMOD}}{32} \times \frac{\text{oscillator frequency}}{12 \times (256 - (TH1))}$$

SMOD is a control bit in the PCON register. It can be '0 ' or '1'.

- If the timer 1 is not in mode 2 then the baud rate is

$$\text{baud rate} = \frac{2^{SMOD}}{32} \times (\text{timer 1 overflow rate})$$

- The Table 3.5.1 shows the commonly used baud rates and the way in which they can be obtained from Timer 1. The oscillator frequency is specified.

Table 3.5.1

Baud Rate	f_{osc}	SMOD	Timer		
			C/T	Mode	Reload value
Mode 0 maximum = 1 Mbps	12 MHz	×	×	×	×
Mode 1,3 maximum = 62.5 kbps	12 MHz	1	0	2	FF H
Mode 2 maximum = 375 kbps	12 MHz	1	×	×	×
19.2 kbps	11.059 MHz	1	0	2	FE H
9.6 kbps	11.059 MHz	0	0	2	FD H
4.8 kbps	11.059 MHz	0	0	2	FA H
2.4 kbps	11.059 MHz	0	0	2	F4 H
1.2 kbps	11.059 MHz	0	0	2	E8 H
137.5 bps	11.986 MHz	0	0	2	ID H
110 bps	6 MHz	0	0	2	72 H
110 bps	12 MHz	0	0	1	FEEB H

Mode 2

- It supports full duplex transmission it supports 11 bit asynchronous communication. Whenever there is change in data '1' is transmitted, otherwise zero is transmitted. Such an operation is called non-return to zero (NRZ) operation.
- In this mode the number of transmitted bits are 11 i.e. one start bit, nine data bits and one stop bit. The 9th data bit to be transmitted is to be stored in the TB8 bit in the SCON register and is stored in the RB8 bit of SCON when data is received. The start and stop bits are discarded.
- The baud rate for mode 2 is fixed. It can be programmed as,

$$\text{baud rate} = \frac{2^{SMOD}}{64} \times \text{oscillator frequency}$$

In comparison to mode 0, the baud rate in mode 2 is higher than standard communication rates. This is done because multiprocessor systems demand high data rates.

- Mode 2 is mainly used for multiprocessor communication. Multiprocessor communication means that 'N' number of microcontroller based systems are interfaced to each other through the serial communication channel. This helps to transfer the data from one system to another, like a LAN network.

Mode 3

Mode 3 is similar to mode 2 except that in mode 3 the baud rate is determined as in mode 1 using Timer 1.

Syllabus Topic : Interrupts

3.6 Interrupts

→ (MU - Dec. 14, May 17, Dec. 17)

Q. 3.6.1 Explain interrupt structure of 8051 in detail. (Ref. Sec. 3.6) Dec. 14, Dec. 17, 10 Marks

Q. 3.6.2 Explain the hardware and software interrupts of 8051 microcontroller. (Ref. Sec. 3.6) May 17, 10 Marks

Q. 3.6.3 Write a detailed note on the interrupt structure and priorities of the 8051 microcontroller. (Ref. Sec. 3.6) (5 Marks)

Q. 3.6.4 Explain the interrupt structure for 8051 microcontroller along with priorities. (Ref. Sec. 3.6) (5 Marks)

- Whenever the microcontroller is executing a program and if a user wants service to an I/O device then an external asynchronous input would inform the microcontroller that it should complete the execution of current instruction and then fetch a new routine that will service the requesting I/O device. Once, the I/O device is serviced, the microcontroller resumes operation from the point whenever it had stopped. The external asynchronous input applied to the microcontroller is termed as an **Interrupt**.
- The Interrupts may be generated by internal chip operations or they may be provided by external sources. An interrupt causes the microcontroller to enter an interrupt service routine. The interrupt handling routine is located at a predetermined absolute address in the program memory.
- The microcontroller 8051 supports five interrupts. Three interrupts are automatically generated by internal operations and two interrupts are generated by external signals provided. The three interrupts that are automatically generated by internal operations are Timer flag 0 (TF0), Timer flag 1 (TF1) and serial port interrupt (RI or TI). The two interrupts that are triggered by external signals are $\overline{INT0}$ and $\overline{INT1}$.
- All the interrupt functions are under the program control. The programmer is able to change the control bits in the Interrupt Enable register (IE), the interrupt priority register (IP), and the Timer control register (TCON). By setting or clearing the bits in these registers the program can block any or all of the interrupts.
- Fig. 3.6.1 shows the 8051 Interrupt structure and control system.

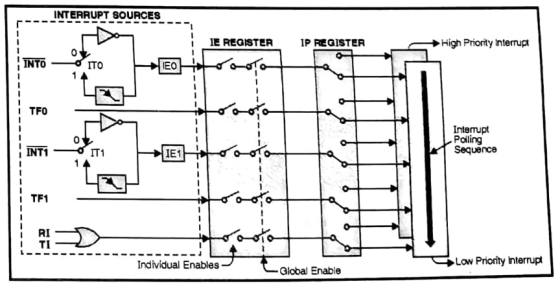


Fig. 3.6.1 : 8051 Interrupt structure and control system

3.6.2 The Interrupt Enable Register (IE)

(LSB)	
	EX0

Function
Disable external interrupt 0
Disable Timer 0 interrupt
Disable external interrupt 1
Disable Timer 1 interrupt
Disable serial port interrupt
Disable Timer 2 interrupt
Disable external interrupt 2
Disable external interrupt 3
Disable external interrupt 4
Disable external interrupt 5
Disable external interrupt 6
Disable all interrupts

Fig. 3.6.2: Interrupt Enable register (IE)

3.6.3 The Interrupt Priority Register (IP)

→ (MU - Dec. 15)

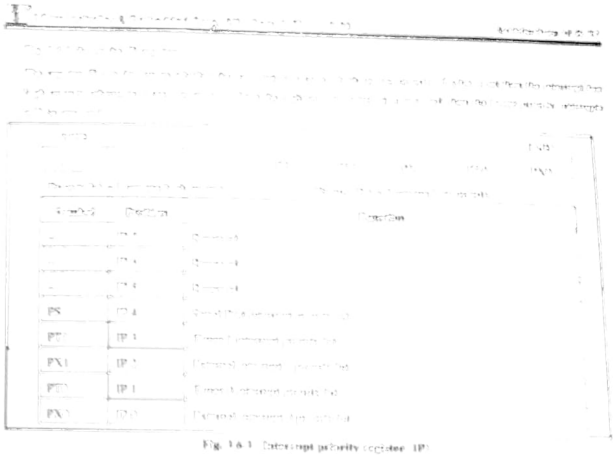


Fig. 3.6.1: Interrupt priority register IP

Table 3.6.1

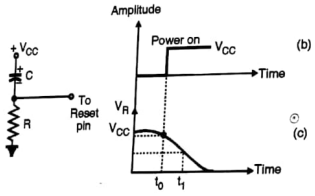
Table 3.6.2

Source	Vector Address
IE0	0003 H
TF0	000B H
IE1	0013 H
TF1	001B H
RI or TI	0023 H

3.7 Reset

- The Reset pin for microcontroller is active HIGH. Whenever power is switched ON, positive going pulse should be present for two machine cycles (The smallest time interval of time that is required to execute an instruction is called as a machine cycle) on this pin.
- The Reset pin can also be considered as an interrupt because the program cannot block the signal on reset pin.

Fig. 3.7.1 shows power on reset circuit for microcontroller



(a) Reset circuit (b) Power on pulse (c) Across resister

Fig. 3.7.1

- At time t_0, the power supply is switched ON. The supply voltage V_{CC} appears across the RC network. The entire voltage appears across the resistor R, so V_R is approximately equal to V_{CC}. This resets the 8051.
- Once the capacitor charges, then V_R starts reducing and reaches approximately 0V. This removes reset signal. The oscillator $t_1 - t_0$ is reset time.
- The reset will force all the SFRs to 00 H, port latches are initialised to FF H, SP to 07 H and SBUF is undetermined. The internal RAM is not affected by reset. The internal RAM content is indeterminate.

Table 3.7.1 lists the values of Registers on Reset.

Table 3.7.1

Registers	Reset Value
PC	0000 H
ACC	00 H
B	00 H

Registers	Reset Value
PSW	00 H
SP	07 H
DPTR	0000 H
P 0 – P 3	FF H
IP	xxx00000B
IE	0xx00000B
TMOD	00 H
TCON	00 H
TH0	00 H
TL0	00 H
TH1	00 H
TL1	00 H
SCON	00 H
SBUF	Indeterminate
PCON	0×××××00 B

3.8 Power Saving Modes of Operation

→ **(MU - Dec. 16)**

Q. 3.8.1 Explain the following SFR's of 8051: PCON. (Ref. Sec. 3.8) Dec. 16, 3 Marks

Q. 3.8.2 Discuss power down mode of 8051 microcontroller. (Ref. Sec. 3.8) **(5 Marks)**

Q. 3.8.3 Discuss the power saving modes of operation of 8051 microcontroller with appropriate diagram (Ref. Sec. 3.8) **(5 Marks)**

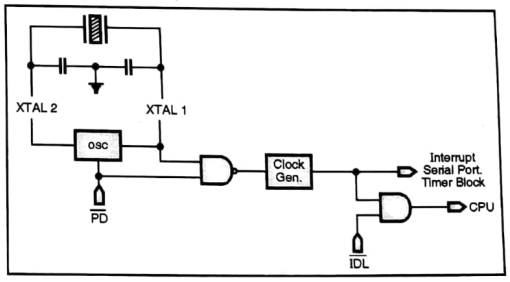


Fig. 3.8.1 : Hardware to implement idle and power down modes

– Power saving feature is available in CHMOS version of the microcontroller. The question now arises that in microcontroller how can one reduce power consumption. The power consumption reduces if some part of the IC is kept in working condition and some part of IC retains previous status and indefinitely stops.

– For applications where power consumption is critical the CHMOS version provides power reduced modes of operation as a standard feature.

– The advantages of reduced power consumption are :

(i) It allows the microcontroller to put more functionality into smaller space.

(ii) It allows the use of smaller and lighter power supplies. As less heat is generated, the packaging can be made dense. Also, the use of expensive fans and blowers for cooling is avoided.

(iii) The cooler running chip is more reliable, as most of the random and wear out failures of the microcontroller are related to temperature.

– The microcontroller has two power saving features. They are idle and power down modes of operation. Fig. 3.8.1 shows the on chip hardware that implements the reduced power modes.

3.8.1 Idle Mode

– As shown in Fig. 3.8.1, the $\overline{IDL}$ is connected to the input of AND gate. The second input to the AND gate is from the clock generator. The output of the AND gate is given to the CPU.

– If $\overline{IDL}$ = 1, then the output of clock generator is given to CPU. But if $\overline{IDL}$ = 0, then output of clock generator will not be passed to the CPU, as output of the AND gate will be zero. So, in the idle mode pulse are not given to the CPU and hence the CPU is at standstill.

– The on chip peripherals i.e. timers, serial port, interrupts, RAM etc. continue to function as normal in the Idle mode.

– The stack pointer, program counter, program status word, accumulator, B register and all other registers maintain their data during idle state.

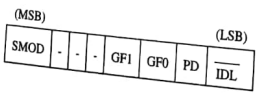
– The ALE and $\overline{PSEN}$ signals are at logic level high when the microcontroller operates in the Idle mode. Due to this external EPROM can be deselected, if its output is disabled.

– The Idle mode is invoked by setting $\overline{IDL}$ = 0. $\overline{IDL}$ is idle mode bit. It resides in the PCON register.

– The PCON register is not bit addressable, so the $\overline{IDL}$ bit has to be set with a byte operation like. ORL PCON, #01H.

– The general purpose flags GF0 and GF1 in the PCON register, give an indication about the interrupt, whether the interrupt occurred during normal operation or idle mode operation. An instruction that invokes idle mode operation causes either of the flags bits GF0 or GF1 to be set.

☞ **PCON register**



Symbol	Position	Name and Function
SMOD	PCON.7	Double Baud rate bit. When set to a 1 and Timer 1 is used to generate baud rate, and the Serial Port is used in modes 1,2 or 3.
—	PCON.6	(Reserved)
—	PCON.5	(Reserved)
—	PCON.4	(Reserved)
GF1	PCON.3	General-purpose flag bit.
GF0	PCON.2	General-purpose flag bit.
PD	PCON.1	Power Down bit. Setting this bit activates power down operation.
$\overline{IDL}$	PCON.0	Idle mode bit. Setting this bit activates idle mode operation.

3.8.2 Termination / Exit from Idle Mode

There are two ways to terminate the Idle Mode.

(1) The activation of any enabled interrupt will cause the PCON.0 (i.e. 0^{th} bit of PCON) to be cleared by the hardware, terminating the idle mode. After clearing the bit the CPU will be activated. This causes an interrupt service routine to be executed. After the execution of interrupt service routine, the program execution will continue from the next instruction that invoked the idle mode.

(2) The other way of termination from the Idle Mode is to Reset the system. As the clock oscillator is running, the hardware reset needs to be held active for two machine cycles (i.e. 24 oscillator periods) to complete the reset. The signal at RST pin of microcontroller clears the $\overline{IDL}$ bit. Finally, CPU resumes program execution from where it was left off. i.e. at the instruction that follows the instruction that invoked the idle mode.

Note : The termination from Idle Mode writes 1s to all the ports, initializes all SFRs to their reset values and it restarts program execution from location 0000H.

3.8.3 Power Down Mode

– To enter the power down mode, we have to set bit 1 (PCON.1) in the PCON register. This will cause the oscillator operation to stop. If the microcontroller was running from an external oscillator, it gates off the path to internal phase generators and no internal clock is generated even if the external oscillator is running. i.e. with the clock frozen / stopped all the functions are stopped.

– However as long as the supply voltage V_{CC} is maintained the contents of internal RAM and SFRs are held.

– In power down mode the ALE and $\overline{PSEN}$ signals are at logic level low.

– The port pins output the values that are held by their respective SFRs.

– In power down mode each and every activity is stopped / frozen. So, an instruction that sets PCON.1 causes that to be the last instruction to be executed before going into the power down mode.

Table 3.8.1 summarizes the status of pins in Idle and power down modes.

Table 3.8.1

Pin	Internal Execution		External Execution	
	Idle	Power down	Idle	Power down
ALE	1	0	1	0
$\overline{PSEN}$	1	0	1	0
P0	SFR data	SFR data	High impedance	High impedance
P1	SFR data	SFR data	SFR data	SFR data
P2	SFR data	SFR data	PCH	SFR data
P3	SFR data	SFR data	SFR data	SFR data

SFR data : internal register data.

PCH : higher byte of program counter.

3.8.4 Termination from Power Down Mode

- The only way to come out of power down mode is hardware reset. As the oscillator was stopped / frozen in the power down mode the *RST* needs to be active for a long time for the oscillator to restart and stabilise.
- The SFRs are initialised to their reset values and program execution begins from 0000H. The contents on the on chip internal RAM are retained.

3.9 EPROM Versions

- The EPROM versions of 8051 are listed in Table 3.9.1. The 8751H programs at V_{PP} = 21V using one 50 msec $\overline{PROG}$ pulse per byte programmed. This results in a total programming time (4K bytes) of approximately 4 minutes.
- The 8752BH and 87C51 use the faster Quick-Pulse algorithm. These devices program at V_{PP} = 12.75 V using a series of twenty-five 100 pulse per byte programmed. This results in a total programming time of approximately 26 seconds for the 8752BH (8K bytes) and 13 seconds for the 87C51 (4K bytes).

Table 3.9.1 : EPROM versions of the 8051 and 8052

Device name	EPROM version	EPROM bytes	Circuit type	VPP	Time required to program entire array
8051	(8751)	4K	HMOS	21.0V	4 minutes
8051AH	8751H	4K	HMOS	21.0V	4 minutes
80C51BH	87C51	4K	CHMOS	12.75V	13 seconds
8052AH	8752BH	8K	HMOS	12.75V	26 seconds

3.10 Single Stepping Operation of 8051

Q. 3.10.1 Explain how single stepping can be done in the 8051 microcontroller. (Ref. Sec. 3.10) **(5 Marks)**

- Beginners, are bound to make mistakes while writing Assembly Language Program (ALP). Single step allows beginner to *step through* instructions one by one. After execution of each instruction, refer to contents of A, B, banks of internal RAM, scratch pad area and so on. Looking at the contents of mentioned register, he will come to know what is exactly going on, and can make some conclusion or decision if going wrong at a particular point. Single stepping allows you to Debug your program and helps you to find out *logical mistakes*.
- MCS-51 interrupt structure allows single step execution with very little software overhead. To understand full operation remember following points :

In 8051, an interrupt request will not be responded to,

- While an interrupt of equal priority level is still in progress.
- After RETI until at least one other Instruction has been executed. Thus once an interrupt routine has been entered, it cannot be re entered until at least one instruction of the interrupted program is executed.

To understand the full scheme, first learn two instruction.

(1) JNB → Jump if bit is '0'.
Syntax → JNB <bit>, <Address>
This instruction will check specified 'bit' and if 'bit' is logical '0', it will jump to specified address.

(2) JB → Jump if bit is '1'.
Syntax → JB <bit>, <Address>
This instruction will check specified 'bit' and if 'bit' is logical '1', it will jump to specified address.

SCHEME

Step 1 : Program one of the external interrupts to be level activated. Let's select $\overline{INT0}$.

Step 2 : Now write service routine for $\overline{INT0}$ interrupt.

The code is as follows :

```
A1 :   JNB P 3.2, A1   ; Wait here till INT0 goes high
A2 :   JB P 3.2, A2    ; Wait here till it goes low
       RETI            ; Go back and execute one instruction.
```

Step 3 : Let's say we have

Main Routine	**ISR Subroutine ($\overline{INT0}$)**
Instruction 1	A1 : JNB P3.2, A1
Instruction 2	A2 : JB P3.2, A2
Instruction 3	RETI
Instruction 4	
Instruction ;	

Instruction ;

Instruction ;

Instruction N

Step 4 : Connect switch to $\overline{INT0}$ pin in such way that it should provide normally LOW, and when pressed should give HIGH.

Step 5 : Whenever we switch on power supply, micro-controller diverts itself to ISR of $\overline{INT0}$, as $\overline{INT0}$ = 0. [Refer Fig. 3.10.1]

Step 6 : 1st instruction of ISR i.e. JNB will force micro-controller to wait till INT0 = 0 . In short it will wait till switch is pressed.

Step 7 : Now press switch ∴ $\overline{INT0}$ = 1. Now ISR will loop in, 2nd Instruction i.e. JB, It will wait till $\overline{INT0}$ = 1

Step 8 : Now release switch ∴ $\overline{INT0}$ = 0. But micro-controller will now execute RETI instruction, micro-controller will not enter into ISR. It will go to main routine, execute one Instruction and after that Interrupt will be sensed and micro-controller will diversify itself to ISR of $\overline{INT0}$. ∴We will be back to step-6. Therefore micro-controller will again expect keyboard or switch depression. For each key depression one instruction from main routine will be executed.

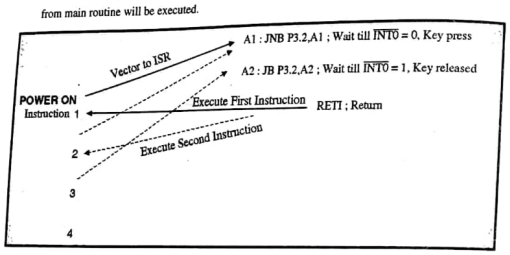


Fig. 3.10.1

3.11 Solved Example

Example 3.11.1 : Suggest the hardware scheme to increase the number of interrupting sources, to be handled by $\overline{INT0}$ or $\overline{INT1}$ pin of 8051 and explain the working of it.

Solution : The number of interrupts can be increased using priority encoder as shown in Fig. P. 3.11.1.

As shown in Fig. P. 3.11.1, 7 interrupt inputs can be connected to the priority encoder. It will give binary output '000', when none of the interrupt occurs. And hence the output of the NOR gate will be '1'. This indicates no interrupt. Whenever an interrupt occurs, the corresponding code will be generated on the select lines (binary output) and hence making the NOR gate output to be '0' and in turn generating interrupt. In the ISR P0 pins 0, 1 and 2 can be read to indicate the interrupt source. To increase the numbers of interrupt sources, the size of the encoder can be increased and also a similar system can be implemented an $\overline{INT1}$ (P3.3). Assuming D_0 has lowest priority and D_7 has highest priority.

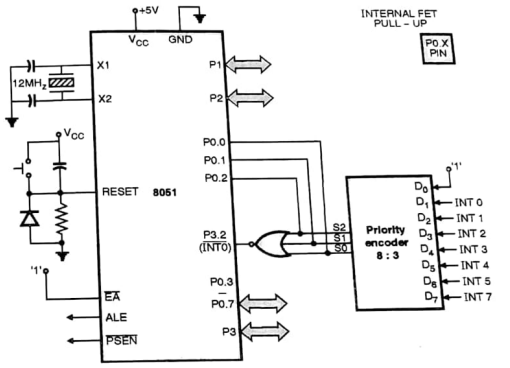


Fig. P. 3.11.1

3.12 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : Architecture**

Q. 1 Explain the architecture of 8051 microcontroller. **(Refer Section 3.1)** **(D-16, 10 Marks)**

Q. 2 Explain the organization of 8051 microcontroller with proper block diagram. **(Refer Section 3.1)** **(5 Marks)**

Q. 3 What are the major components of 8051 microcontroller ? **(Refer Section 3.1.1)** **(D-15, 3 Marks)**

Q. 4 State the features of 8051 microcontroller. **(Refer Section 3.1.1)** **(M-17, 5 Marks)**

Q. 5 Explain PSW register of 8051. **(Refer Section 3.1.5)** **(M-15, 5 Marks)**

Q. 6 Explain functions of Program Status Word Register in 8051 microcontroller. **(Refer Section 3.1.5)** **(D-15, 6 Marks)**

☞ **Syllabus Topic : Memory Organisation**

Q. 7 Explain internal memory organization of 8051. **(Refer Section 3.2.1)** **(D-14, 10 Marks)**

Q. 8 Write note on 8051 register bank. **(Refer Section 3.2.1.1(A))** **(D-16, 10 Marks)**

Q. 9 What are special function registers ? Explain their utility with a few examples. **(Refer Section 3.2.3)** **(5 Marks)**

Q. 10 What are 'SFRs' in 8051 microcontroller ? Explain their utility. **(Refer Section 3.2.3)** **(5 Marks)**

Q. 11 Explain the physical structure of I/O ports of the 8051 microcontroller. **(Refer Section 3.3)** **(5 Marks)**

☞ **Syllabus Topic : Counters and Timers**

Q. 12 Explain functions of Timer Control Register in 8051 microcontroller. **(Refer Section 3.4)** **(D-15, 3 Marks)**

Q. 13 Explain the Timer/Counter modes of 8051 microcontroller. **(Refer Section 3.4)** **(M-16, M-17, 10 Marks)**

Q. 14 Explain the following SFR's of 8051 : TCON, TMOD. **(Refer Section 3.4)** **(D-16, 5 Marks)**

Q. 15 Discuss the internal timer / counter of 8051 microcontroller with appropriate diagrams. **(Refer Section 3.4)** **(5 Marks)**

☞ **Syllabus Topic : Serial Communication**

Q. 16 Explain various modes of operation of serial port in 8051. **(Refer Section 3.5)** **(M-15, 10 Marks)**

Q. 17 Write short note on : Serial Port Communication in 8051. **(Refer Section 3.5)** **(D-15, D-16, M-17, 6 Marks)**

Q. 18 Explain various serial modes of 8051 microcontroller. **(Refer Section 3.5)** **(M-16, 10 Marks)**

Q. 19 Explain the following SFR's of 8051 : SCON. **(Refer Section 3.5)** **(D-16, 3 Marks)**

Q. 20 Discuss the modes of operation of the serial port of the 8051 microcontroller. **(Refer Section 3.5)** **(5 Marks)**

☞ **Syllabus Topic : Interrupts**

Q. 21 Explain interrupt structure of 8051 in detail. **(Refer Section 3.6)** **(D-14, D-17, 10 Marks)**

Q. 22 Explain the hardware and software interrupts of 8051 microcontroller. **(Refer Section 3.6)** **(M-17, 10 Marks)**

Q. 23 Write a detailed note on the interrupt structure and priorities of the 8051 microcontroller. **(Refer Section 3.6)** **(5 Marks)**

Q. 24 Explain the interrupt structure for 8051 microcontroller along with priorities. **(Refer Section 3.6)** **(5 Marks)**

Q. 25 Explain functions of Interrupt Priority Register in 8051 microcontroller. **(Refer Section 3.6.3)** **(D-15, 3 Marks)**

Q. 26 Explain the following SFR's of 8051 : PCON. **(Refer Section 3.8)** **(D-16, 3 Marks)**

Q. 27 Discuss power down mode of 8051 microcontroller. **(Refer Section 3.8)** **(5 Marks)**

Q. 28 Discuss the power saving modes of operation of 8051 microcontroller with appropriate diagram. **(Refer Section 3.8)** **(5 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 4

Instruction Set of 8051

Syllabus Topic : Instruction Set

4.1 Introduction

- In this chapter we will study the instruction set of Microcontroller 8051. These instructions treat different types of operands uniformly. Register, memory and immediate operands may be specified interchangeably in most instructions.
- The instruction set is divided into number of groups of functionally related instructions. The different groups are :
  (1) Data transfer group
  (2) Arithmetic group
  (3) Bit manipulation group
  (4) Program transfer instruction group
  (5) Processor control group
- Before studying the instruction set, it is essential to know how 8051 accesses the instruction operands in different ways i.e. the Addressing modes.

Syllabus Topic : Addressing Modes

4.2 Addressing Modes

➔ **(MU - May 15, Dec. 15, Dec. 16, May 17, Dec. 17)**

Q. 4.2.1 What are the Addressing Modes of 8051 microcontroller ? Explain with example in each addressing mode. (Ref. Sec. 4.2) May 15, Dec. 15, Dec. 16, May 17, Dec. 17, 10 Marks

Q. 4.2.2 Explain the different addressing modes of 8051. (Ref. Sec. 4.2) **(8 Marks)**

Q. 4.2.3 How will an instruction distinguish between internal and external memory reference ? (Ref. Sec. 4.2) **(5 Marks)**

- When the microcontroller executes an instruction, it performs specific function on data. The data is stored at some source location. This data must be moved or copied to destination location. The ways by which these addresses locations are specified are called as **Addressing Modes.**
- The Addressing modes for 8051 can be given as follows :

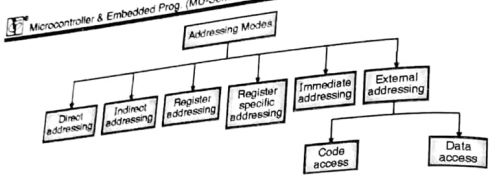


Fig. 4.2.1

4.2.1 Direct Addressing Mode

- In this mode, the operand is specified by an 8 bit address field in the instruction. One can access all the 128 bytes of internal RAM and the SFRs directly, using the single byte address that is assigned to each RAM location and each special function register.
- The most significant bit in the address decides whether the location is within the on chip internal RAM or in the special function register. If MSB = 0, then the location is within onchip internal RAM. If MSB = 1, then the location is in the special function register.
- The internal RAM uses addresses from 00 H to 7FH to address each byte. The SFR addresses are from 80H to FFH.

Example :

MOV A, 40 H : Copy data from address 40 H into register A

MOV R0, 14 H : Copy the contents of memory location 14H, to register R0 of the selected bank.

4.2.2 Indirect Addressing Mode

- In this addressing mode, the instruction specifies a register which contains address of an operand i.e. the register holds the actual address that will be used in the data move operation. This address may be 8 bit or a 16 bit address.
- The R0 and R1 of each register bank can be used as an index or pointer register. R0 and R1 point to the contents in the RAM.
- The @ sign indicates the register acts as a pointer to memory location.

Example

MOV A, @ R1 : Copy the contents of memory location, whose address is specified in R1 register of selected bank to the accumulator.

Note : @ indicates that the register acts like a pointer.

MOV @R0, 85 H : Copy the data of address 85H to the memory location whose address is specified by the R0 register of selected bank.

Only registers R0 and R1 can be used for indirect addressing. If registers R2 to R7 are used, then in that case the instruction becomes an invalid instruction

eg :- MOV @R2, A ; invalid instruction.

4.2.3 Register Addressing Mode

Each register bank consists of registers R0 to R7. To access these registers there are special instructions. In the instruction Opcode, 3 bits are reserved for specifying one of the eight registers from the selected register bank. For selecting the register bank, the user has to modify two bits in the PSW.

Example

MOV A, R2 ; copy the data from register R2 of the selected register bank to register A.

4.2.4 Register Specific Addressing Mode

In this addressing mode the instructions refer to a specific register such as accumulator or data pointer DPTR.

Example

DA A ; Decimal adjust accumulator for addition.

RR A ; Rotate the contents of accumulator to the right

SWAP A ; Swap the nibbles within the accumulator.

4.2.5 Immediate Addressing Mode

- This method is the simplest method to get the data. In this addressing mode the source operand is a constant rather than a variable. As the data source is a part of the instruction it is immediately available.
- The # sign indicates that the data followed is immediate operand.

Example :

MOV A, #30H ; Copy 30H immediately to accumulator.

MOV P1, #0FFH ; Copy FFH immediately to port 1.

MOV DPTR, #1234H ; Copy 1234H immediately to data pointer (DPTR).

4.2.6 External Addressing Mode

Q. 4.2.4 Describe the following instructions : MOVC A, @A + DPTR. (Ref. Sec. 4.2.6) **(2 Marks)**

☞ **(a) Code access (External ROM access)**

- Using these instructions only external program memory can be accessed.
- This addressing mode is preferred for reading look up tables in the program memory. Either the DPTR or PC (Program Counter) can be used as pointer.

Example :

MOVC A, @A + DPTR ; This instruction will load the accumulator with the byte from program memory. The byte from program memory is fetched from the sum of unsigned eight bit accumulator contents and contents of the DPTR.

☞ **(b) Data access (External RAM access)**

- Using this addressing mode the programmer can access the external data memory.

Example :

MOVX @R0, A : This instruction will copy the data from accumulator to the external memory location, whose address is given by register R0. Using Register R0 or R1, the programmer can access external data memory from location 00H to FFH. To access the memory beyond this, DPTR is used.

4.3 Data Movement, Exchange and PUSH/POP Instructions

4.3.1 MOV <dest – byte>, <src–byte>

Mnemonic :	MOV <Dest–byte>, <src–byte>
Algorithm:	destination = source.
Function :	Move byte variable.
Operation : MOV <dest–byte>, <src–byte>	– This instruction copies the contents of the source location to the destination location. The contents of source location are unchanged. – It is the most flexible operation. It allows fifteen combinations of source and destination addressing modes. – Depending on the type of transfer the number of bytes required may be 1, 2, or 3 and number of cycles required are 1 or 2.

Let us see the different combinations :

Case (i) : *If the destination byte is Accumulator, then source byte may be*

☞ **(1) MOV A, Rn**

Mnemonic	MOV A, Rn	**Function**	Move byte from register to accumulator.
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	A = Rn
Addr. Mode	Register Addr.Mode.	**Flags**	No flags are affected.

Operation	MOV A, Rn.	This instruction copies the contents of the register Rn to the accumulator.
Example	MOV A, R1.	This instruction will copy the contents of register R1 of the selected register bank to the accumulator. Refer Fig. 4.3.1

Note : *Rn refers to any register R0 to R7 for all instructions of 8051.*

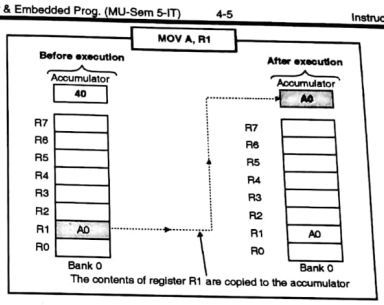


Fig. 4.3.1

☞ **(2) MOV A, direct**

Mnemonic	MOV A, direct	**Function**	Move byte from direct address to accumulator.
Machine cycles	1	**Clock Pulses**	12
Bytes	2	**Algorithm**	A = [direct]
Addr. Mode	direct Addr. mode.	**Flags**	No flags are affected.

Operation	MOV A, direct	This instruction will copy the contents of direct address given in the instruction to the accumulator.
Example	MOV A, 40H	This instruction will copy the contents from memory location whose address is 40H to the accumulator. Fig. 4.3.2 shows this.

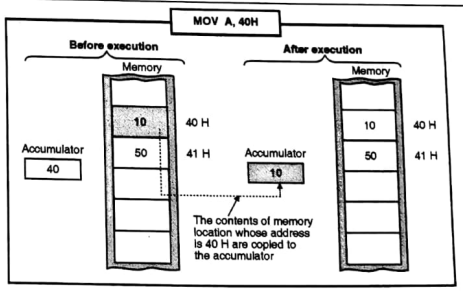


Fig. 4.3.2

(3) MOV A, @Ri

Mnemonic	MOV A, @Ri	Function	Move the contents of memory location pointed by Ri to accumulator.
Machine cycles	1	Clock Pulses	12
Bytes	1	Algorithm	A = [Ri]
Addr. Mode	Register indirect addressing mode.	Flags	No flags are affected.
Operation	MOV A, @Ri	This instruction will copy the contents of memory location whose address is specified in the register Ri of the selected bank to the accumulator.	
Example	MOV A, @R0	This instruction will copy the contents of memory location whose address is specified in the R0 register of the selected bank to accumulator. Let R0 = 40H. Contents of location 40H = 10H, Fig. 4.3.3 shows this.	

Note : Ri refers to R0 or R1 for all instructions of 8051.

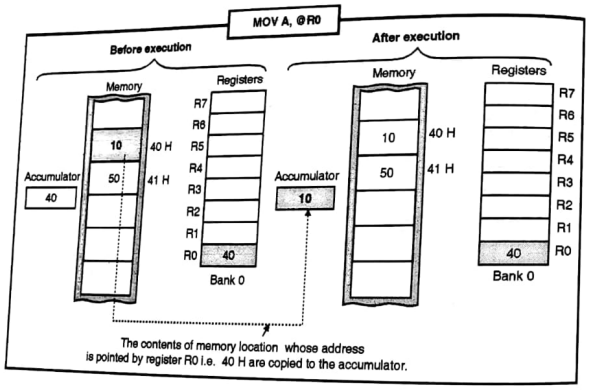


Fig. 4.3.3

(4) MOV A, #data

Mnemonic	MOV A, #data	Function	Move the immediate data to accumulator.
Machine cycles	1	Clock Pulses	12
Bytes	2	Algorithm	A = data
Addr. Mode	Immediate Addr. mode.	Flags	No flags are affected.
Operation	MOV A, #data	This instruction will copy the immediate data to accumulator.	
Example	MOV A, #31H	This instruction will move the data 31H immediately to the accumulator. Fig. 4.3.4 shows this.	

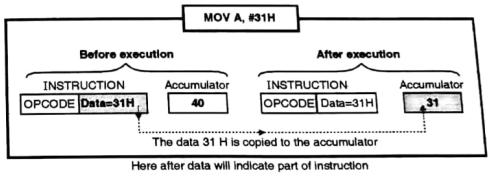


Fig. 4.3.4

Case (ii) : If the destination is a register, then the source may be

(5) MOV Rn, A

Mnemonic	MOV Rn, A	Function	Move data from accumulator to register.
Machine cycles	1	Clock Pulses	12
Bytes	1	Algorithm	Rn = A
Addr. Mode	Register Addr. mode.	Flags	No flags are affected.
Operation	MOV Rn, A	This instruction will copy the contents of accumulator to the register Rn of selected register bank.	
Example	MOV R5, A	This instruction will copy the contents of accumulator to register R5 of selected register bank. Let contents of register R5 = A0H, A = 40H. Fig. 4.3.5 shows this operation.	

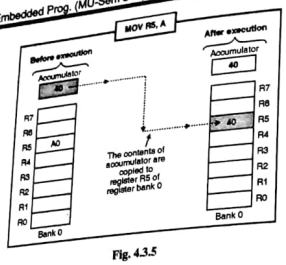


Fig. 4.3.5

☞ **(6) MOV Rn, direct**

Mnemonic	MOV Rn, direct.	**Function**	Move data from direct address to register.
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	Rn = direct
Addr. Mode	Direct Addr. mode.	**Flags**	No flags are affected.

Operation	MOV Rn, direct ↑	This instruction will copy contents from direct address specified in the instruction to register Rn of selected register bank.
Example	MOV R3, 30H	This instruction will copy the contents from address 30 H to the register R3 of selected register bank. Fig. 4.3.6 shows this.

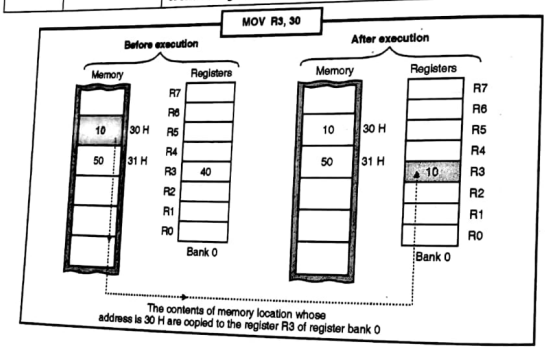


Fig. 4.3.6

☞ **(7) MOV Rn, # data**

Mnemonic	MOV Rn, # data	**Function**	Move immediate data to register.
Machine cycles	1	**Clock Pulses**	12
Bytes	2	**Algorithm**	Rn = data
Addr. Mode	Immediate Addr. mode.	**Flags**	No flags are affected.

Operation	MOV Rn, # data ↑	This instruction will copy the immediate data to the register Rn of selected register bank.
Example	MOV R7, # 20H	This instruction will copy immediate data 20H to the register R7 of the selected register bank. Fig. 4.3.7 shows this.

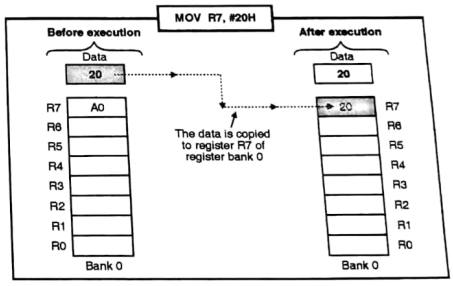


Fig. 4.3.7

Case (iii) : If the destination is a direct address, then source may be.

☞ **(8) MOV direct, A**

Mnemonic	MOV direct, A	**Function**	Move data from accumulator to direct address.
Machine cycles	1	**Clock Pulses**	12
Bytes	2	**Algorithm**	direct = A
Addr. Mode	direct addressing mode.	**Flags**	No flags are affected.

Operation	MOV direct, A ↑	This instruction will copy the contents of accumulator to the direct address specified in the instruction.
Example	MOV 80H, A	80H is the address of port 0. This instruction will copy the contents of accumulator to the port 0 latch. Fig. 4.3.8 illustrates this.

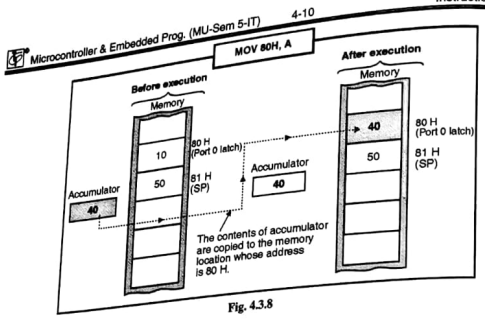


Fig. 4.3.8

(9) MOV direct, Rn

Mnemonic	MOV direct, Rn	**Function**	Move data from register to direct address.
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	direct = Rn.
Addr. Mode	direct addressing mode.	**Flags**	No flags are affected.

Operation	MOV direct, Rn	This instruction will copy the contents of register Rn of the selected register bank to the direct address specified in the instruction.
Example	MOV 30H, R5	This instruction will copy the contents of register R5 of the selected register bank to the memory location whose address is 30H.Fig. 4.3.9 shows this.

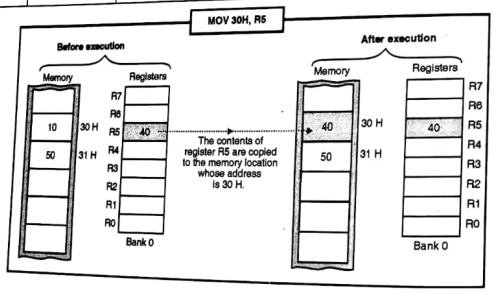


Fig. 4.3.9

(10) MOV direct, direct

Mnemonic	MOV direct, direct.	**Function**	Move data from source direct address to destination direct address.
Machine cycles	2	**Clock Pulses**	24
Bytes	3	**Algorithm**	direct = direct
Addr. Mode	direct addressing mode.	**Flags**	No flags are affected.

Operation	MOV direct, direct	This instruction will copy the contents of source direct address to the destination direct address.
Example	MOV 20H, 30H	This instruction will copy the contents of memory location whose address is 30H to the memory location whose address is 20H. Fig. 4.3.10 shows this.

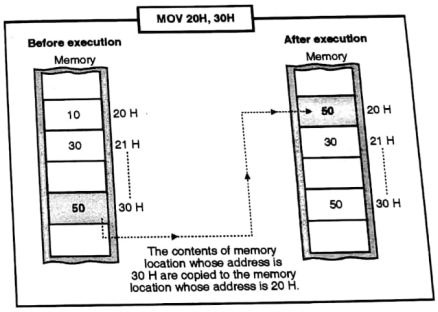


Fig. 4.3.10

(11) MOV direct, @Ri

Mnemonic	MOV direct, @Ri	**Function**	Move data from address specified in register Ri to direct address.
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	direct = ((Ri))
Addr. Mode	register indirect addressing mode.	**Flags**	No flags are affected.

Operation	MOV direct, @Ri	This instruction will copy the data from memory location whose address is specified in the Ri register of the selected register bank to the direct address.
Example	MOV 20H, @R1	This instruction will copy the data from memory location whose address is given in R1 register of selected register bank to memory location whose address is 20H. Fig. 4.3.11 shows this.

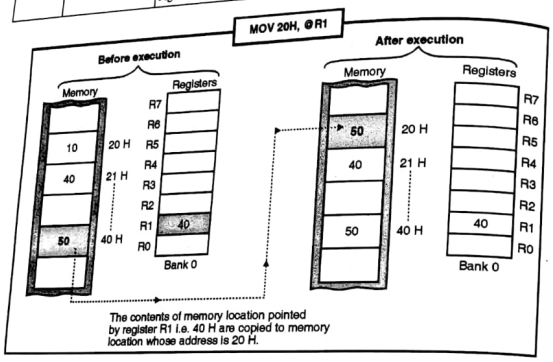


Fig. 4.3.11

(12) MOV direct, # data

Mnemonic	MOV direct, # data	**Function**	Move immediate data to direct address.
Machine cycles	2	**Clock Pulses**	24
Bytes	3	**Algorithm**	direct = # data
Addr. Mode	immediate addressing mode.	**Flags**	No flags are affected.

Operation	MOV direct, # data	This instruction will copy the immediate data to direct address specified in the instruction.
Example	MOV 10H, # 10H	This instruction will copy immediate data 10H to memory location whose address is 10H. Fig. 4.3.12 shows this.

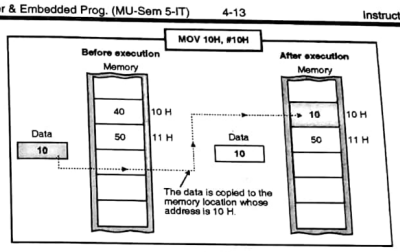


Fig. 4.3.12

Case (iv) : If the destination byte is a memory location on pointed by Ri, then the source may be

(13) MOV @RI, A

Mnemonic	MOV @Ri, A	**Function**	Move data from accumulator to memory location pointed by Ri.
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	((Ri)) = A
Addr. Mode	indirect addressing mode.	**Flags**	No flags are affected.

Operation	MOV @Ri, A	This instruction will copy the contents of accumulator to the memory location pointed by Ri.
Example	MOV @ R0, A	This instruction will copy the contents of accumulator to memory location whose address is pointed by the R0 register of selected register bank. Fig. 4.3.13 shows this.

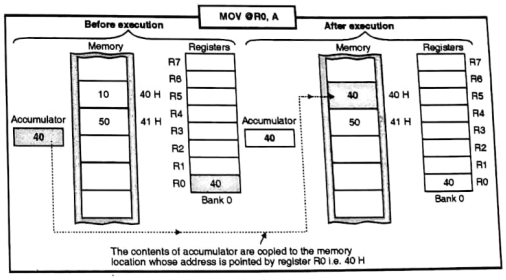


Fig. 4.3.13

(14) MOV @Ri, Direct

Mnemonic	MOV @Ri, direct
Machine cycles	2
Bytes	2
Addr. Mode	direct addressing mode.

Function	Move data from direct address to memory location pointed by Ri.
Clock Pulses	24
Algorithm	((Ri)) = direct
Flags	No flags are affected.

Operation	MOV @Ri, direct	This instruction will copy the contents of direct address specified in the instruction to the memory location pointed by Ri register of selected register bank.
Example	(1) MOV @R0, 30H	This instruction will copy data from memory location whose address is 30 H to the memory location pointed by register R0 of selected register bank. Fig. 4.3.14 shows this.
	(2) MOV @R0, iram addr	This instruction will copy the contents of direct RAM address specified in the instruction to the memory location pointed by R0 register of the selected register bank.

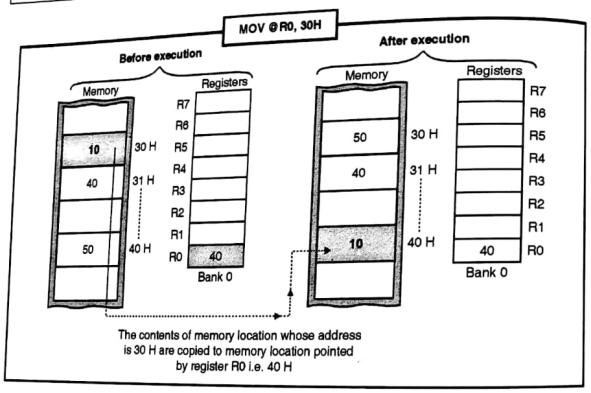


Fig. 4.3.14

(15) MOV @Ri, # data

Mnemonic	MOV @Ri, #data
Machine cycles	1
Bytes	2
Addr. Mode	immediate addressing mode.

Function	Move immediate data to memory location pointed by the Ri register of selected register bank.
Clock Pulses	12
Algorithm	((Ri)) = # data
Flags	No flags are affected.

Operation	MOV @Ri, #data	This instruction will copy the immediate data to memory location pointed by register Ri of selected register bank.
Example	MOV @ R0, # 30H	This instruction will copy data immediate data 30H to memory location pointed by R0 register of selected register bank. Fig. 4.3.15 shows this.

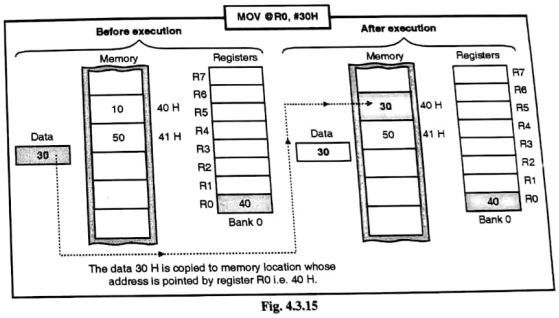


Fig. 4.3.15

4.3.2 MOV DPTR, # data 16

Mnemonic	MOV DPTR, # data 16
Machine cycles	2
Bytes	3
Addr. Mode	Immediate Addressing mode.

Function	Load data pointer with a 16-bit constant.
Clock Pulses	24
Algorithm	(DPTR) = # $data_{15-0}$ (DPH) = # $data_{15-8}$, (DPL) = # $data_{7-0}$
Flags	No flags are affected.

Operation	MOV (DPTR) ← R#data 15-0 OR (DPH) ← # data15-8, (DPL) ← # data 7-0	– This instruction will load the data pointer with the 16 bit constant indicated. – The 16 bit constant is loaded into the second and third bytes of the instruction. The second bytes (DPH) holds the high-order byte. While the third byte (DPL) holds low-order byte.
Example	MOV DPTR, # 2476 H	This instruction will load the immediate data 2476H into the Data Pointer. DPH will hold 24H, while DPL will hold 76H. Fig. 4.3.16 shows this.

Note : This is the only instruction which moves 16 bits of data at once.

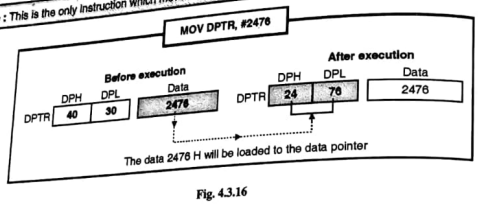


Fig. 4.3.16

4.3.3 MOVC A, @ A+ <base register>

Mnemonic	MOVC A, @ A + <base register>	**Function**	Move code byte.
Machine cycles	2	**Clock Pulses**	24
Bytes	1	**Algorithm**	A = ((A) + ((DPTR)) OR (PC) = (PC) + 1, A = (A) + (PC)
	Indirect addressing mode.	**Flags**	No flags are affected.

Operation	MOVC A, @ A + DPTR (A) ← ((A) + (DPTR)) OR MOVC A, @ A + PC (PC) ← (PC) + 1 (A) ← (A) + (PC)	– This instruction will load the accumulator with a code byte or constant from the program memory. – Hence, it is essential to generate 16 bit address, so that data can be fetched from that address. The address of the byte fetched is the sum of the original unsigned eight-bit accumulator contents and the contents of a sixteen bit base register. The 16 bit base register may be the Data pointer or the program counter (PC).
		– If the base register used is PC, then the PC is incremented to the address of the following instruction i.e. next instruction before being added with the Accumulator, otherwise the base register remains unaltered.

Example	MOVC A, @ A + DPTR	Let (DPTR) = 1000 H, (A) = 8H Contents of memory location (1008H) = 22H This instruction will copy the contents of address A + DPTR → 8H+ 1000H → 1008H to the accumulator i.e. after execution of this instruction (A) = 22 H. Fig. 4.3.17 shows this.

Note : The DPTR and PC remain unchanged, the accumulator contains the code byte fetched from program memory.

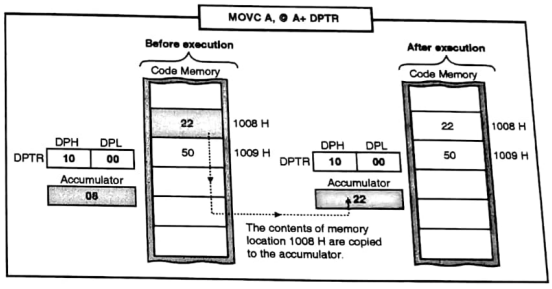


Fig. 4.3.17

Another example, MOVC A, @ A + PC

Let (PC) = 4000 H and (A) = 50 H.

Let contents of memory location 4051H = 52H.

Initially the 16 bit address is computed.

(PC) = (PC) + 1 (PC) = 4000 H + 1H

(PC) = 4001 H (A) + (PC) → 50 H + 4001H → 4051 H.

This instruction will copy the contents of memory location 4051H i.e. 52H to the accumulator.

4.3.4 MOVX <dest-byte>, <src-byte>

→ **(MU - May 15)**

Q. 4.3.1 Describe the instructions of 8051,MOVX @ DPTR, A with one example. (Ref. Sec. 4.3.4) **May 15, 2 Marks**

Mnemonic	MOVX <dest-byte>, <src-byte>	**Function**	Move External.
Machine cycles	2	**Clock Pulses**	24

Bytes	1	**Algorithm**	dest-byte = src-byte
Addr. Mode	register indirect addressing mode.	**Flags**	No flags are affected.

Operation	MOVX <dest-byte>, <src-byte>	– The MOVX instructions transfer data between the accumulator and a byte of external data memory, hence "X" is appended to MOV. – Depending on whether the indirect address provided to the external data RAM is eight bit or 16 bit, there are two types of instructions. – In the first type, the contents of register R0 or R1 of current register bank provide an 8 bit address that is multiplexed with data on port 0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. – In the second type, the Data Pointer is used for generating 16 bit address. This is done for larger RAM array. Port 2 outputs the high-order address bits (the contents of DPH), while Port 0 output the low-order address bits (contents of DPL) multiplexed with data.
Example	MOVX A, @ R0	This instruction will copy the data from the 8 bit address pointed by register R0 of the selected register bank to the accumulator. Fig. 4.3.18 shows this.

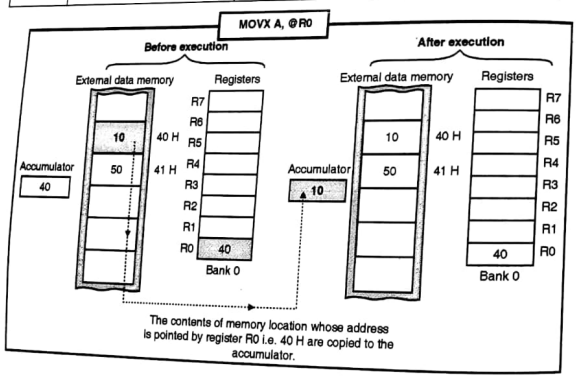


Fig. 4.3.18

(1) MOVX A, @ DPTR

This instruction will copy the contents of external data memory location, pointed by DPTR to the accumulator. Fig. 4.3.19 shows this.

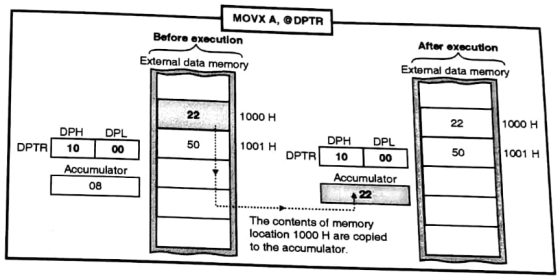


Fig. 4.3.19

(2) MOVX @ RI, A

This instruction will copy the contents of accumulator to the external data memory location pointed by register Ri of selected register bank. Fig. 4.3.20 shows this.

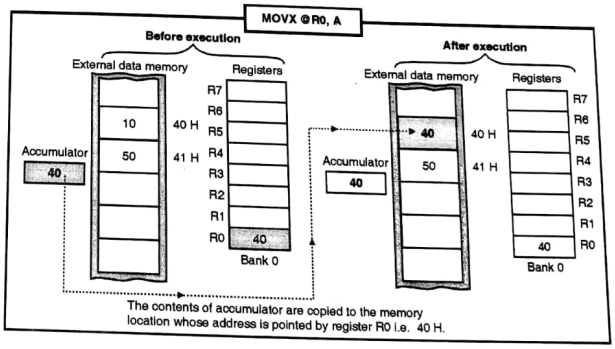


Fig. 4.3.20

☛ **(3) MOVX @ DPTR, A**

This instruction will copy the contents of accumulator to the 16 bit address, pointed by DPTR Fig. 4.3.21 shows this.

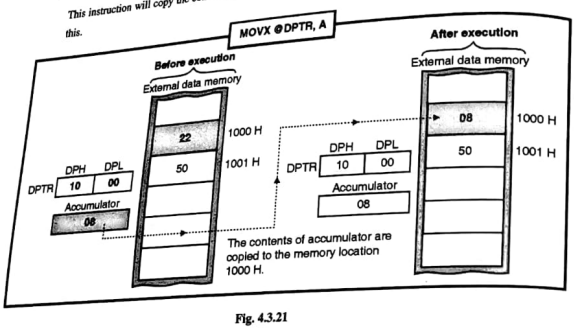


Fig. 4.3.21

4.3.5 PUSH <direct>

Mnemonic	PUSH <direct>
Machine cycles	2
Bytes	2
Addr. Mode	direct addressing mode.

Function	Push onto stack.
Clock Pulses	24
Algorithm	(SP) = (SP) + 1 ((SP)) = direct
Flags	No flags are affected.

Operation	PUSH (SP) ← (SP) + 1 ((SP)) ← direct	– This instruction copies the data from the source address onto the stack. – The stack pointer (SP) is incremented by 1 before the data in copied, to the internal RAM location addressed by the stack pointer. – The stack will grow up in memory as data is pushed onto the stack. If the stack exceeds 7F H (i.e. top of internal RAM), then it results in errors.
Example	PUSH DPL PUSH DPH	Let SP = 0AH and Data Pointer=1234 H. The first instruction PUSH DPL will set the SP=0BH and store 34H in internal Ram location 0BH. The second instruction PUSH DPH will set the SP = 0CH and store 12H in internal RAM location 0BH. The stack pointer will remain at 0CH. Fig. 4.3.22 shows this.

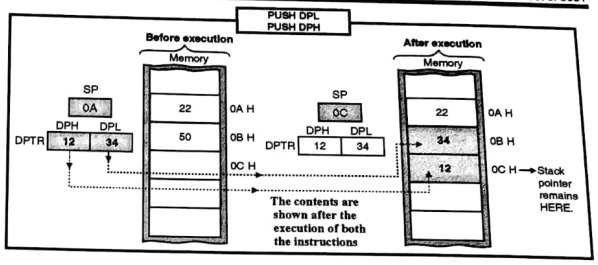


Fig. 4.3.22

4.3.6 POP <direct>

Mnemonic	POP <direct>
Machine cycles	2
Bytes	2
Addr. Mode	Direct addressing mode.

Function	Pop from the stack.
Clock Pulses	24
Algorithm	direct = ((SP)) (SP) = (SP) – 1
Flags	No flags are affected.

Operation	POP direct (direct) ← ((SP)) (SP) ← (SP) – 1	– This instruction copies data from the stack to the destination location. – The SP is decremented by 1 after data is copied from the stack. RAM address to the direct destination address. This is done to ensure that the data placed on the stack is retrieved in the same order as it was stored.
Example	POP DPL	Let SP = 34H and data at internal RAM locations 34H be 52H. The instruction POP DPL will make the stack pointer to value 33H and DPL = 52 H. Fig. 4.3.23 shows this.

Note : The Push and Pop operation can Push /Pop a single byte only.

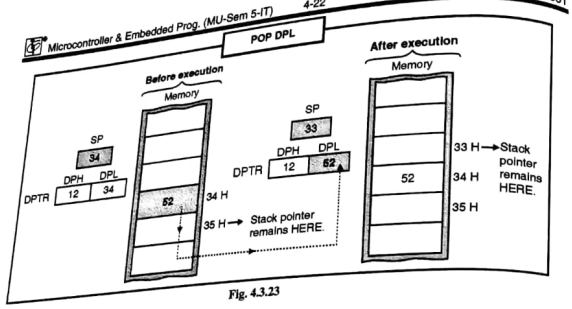


Fig. 4.3.23

4.3.7 XCH A, <byte variable>

Mnemonic : XCH A, <byte variable >	Function : Exchange accumulator with byte variable.
Algorithm : (A) = (byte variable) (byte variable) = (A)	Operation : (A) ↔ (Byte variable)

☞ **Operation**

- This instruction will load the accumulator with the contents of byte variable. At the same time the original accumulator contents are written to the byte variable.
- The source / destination operand can use register, direct, or register indirect addressing. Let us see the different combinations depending on the different addressing modes.

☞ **(1) XCH A, Rn**

Mnemonic	XCH A, Rn
Machine cycles	1
Bytes	1
Addr. Mode	register addressing mode.

Function	Exchange accumulator with byte in register.
Clock Pulses	12
Algorithm	(A) = (byte variable in register) (byte variable in register) = (A)
Flags	No flags are affected.

Operation	(A) ↔ (Rn)	This instruction will exchange the contents of accumulator with the contents of register Rn of selected register bank.
Example	XCH A, R1	This instruction will load the contents of register R1 of selected register bank in the accumulator and at the same time the contents of original accumulator will be copied in register R1. Fig. 4.3.24 shows this.

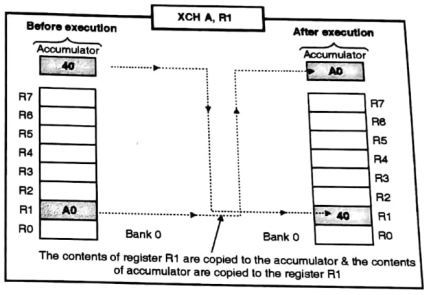


Fig. 4.3.24

☞ **(2) XCH A, direct**

Mnemonic	XCH A, direct
Machine cycles	1
Bytes	2
Addr. Mode	direct addressing mode.

Function	Exchange accumulator with byte in memory.
Clock Pulses	12
Algorithm	(A) = (direct) (direct) = (A)
Flags	No flags are affected.

Operation	(A) ↔ (direct)	This instruction will exchange the contents of accumulator with the direct address.
Example	XCH A, 10H	This instruction will load the contents of memory location whose address is 10H to the accumulator and at the same time the contents of accumulator are transferred to memory location whose address is 10H. Fig. 4.3.25 shows this.

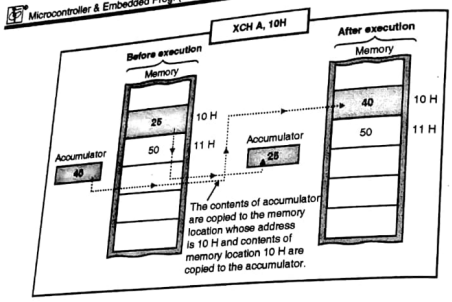


Fig. 4.3.25

☞ (3) XCH A, @Ri

Mnemonic	XCH A, @Ri
Machine cycles	1
Bytes	1
Addr. Mode	Register indirect addressing mode.

Function	Exchange accumulator with byte variable.
Clock Pulses	12
Algorithm	(A) = ((Ri)) ((Ri)) = (A)
Flags	No flags are affected.

Operation	XCH (A) ↔ ((Ri))	This instruction will exchange the contents of accumulator with the contents of memory pointed by register Ri i.e. contents of memory location pointed by Ri will be transferred to accumulator and at the same time contents of the accumulator are transferred to memory location pointed by Ri.
Example	XCH A, @R0	This instruction will load the contents of memory location pointed by register R0 of selected register bank to the accumulator and at the same time the contents of accumulator are copied to the memory location pointed by R0 register of the selected register bank.

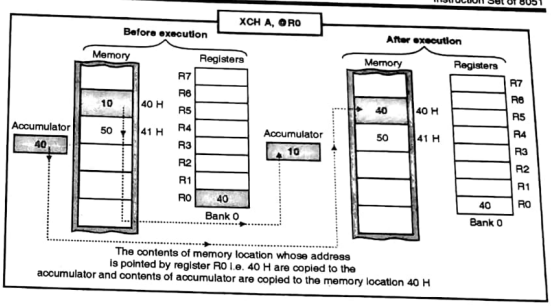


Fig. 4.3.26

4.3.8 XCHD A, @Ri

Q. 4.3.2 Describe the following instruction : XCHD A, @R1. (Ref. Sec. 4.3.8) **(2 Marks)**

Mnemonic	XCHD @ Ri
Machine cycles	1
Bytes	1
Addr. Mode	Register indirect addressing mode.

Function	Exchange Digit.
Clock Pulses	12
Algorithm	$(A_{(3\text{-}0)}) = ((Ri_{(3\text{-}0)}))$
Flags	No flags are affected.

Operation	XCHD $(A)_{(3-0)} \leftrightarrow ((Ri_{(3\text{-}0)}))$	– This instruction exchanges the lower nibble of accumulator (bits 3-0) with the lower nibble of the memory location indirectly addressed by the specified register R0/R1 of the specified register bank. – The upper nibble (bits 7-4) of each register remain unchanged.
Example	XCHD A, @R0	Let R0 contain address 37 H, accumulator contain 25H, and the internal RAM location 37H contain 47H, then the instruction XCHD A, @ R0 will leave RAM location 37H holding the value 45H and accumulator holding the value 27H.

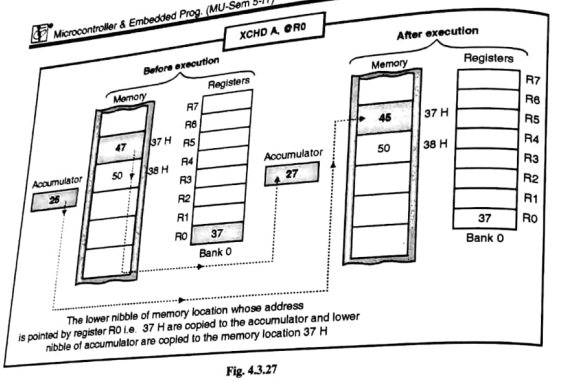


Fig. 4.3.27

Syllabus Topic : Programming Based on Arithmetic Operations

4.4 Arithmetic Instructions

4.4.1 ADD A, <src-byte>

Mnemonic : ADD A, <src-byte>	**Function :** Add
Algorithm : (A) = (A) + <src-byte>	**Operation :** (A) ← (A) + <src-byte>

- *This instruction adds the byte variable indicated to the accumulator. The result is contained in the accumulator.*
- All the addressing modes can be used for source : an immediate number, a register, direct address, indirect address.
- Depending on the addressing modes, let us see the different combinations :

☞ **(1) ADD A, Rn**

Mnemonic	ADD A, Rn
Machine cycles	1
Bytes	1
Addr. Mode	Register addressing mode.

Function	ADD
Clock Pulses	12
Algorithm	(A) = (A) + (Rn)
Flags	Flags are affected.

Operation	(A) ← (A) + (Rn)	This instruction will add the byte in register Rn of the selected register bank with the byte in accumulator. The result is contained in the accumulator.
Example	ADD A, R0	Let A = 42H (0100 0010 B) and R0 = 91 H (1001 0001B). then, ADD A, R0 will leave A = D3H (1101 0011 B) with the AC flag cleared, carry flag and overflow flag also cleared. Fig. 4.4.1 shows this. – Carry flag is set if there is carry-out from bit 7 – Auxiliary carry flag is set if there is carry-out from bit 3. – The overflow flag is set if there is a carry out of bit 6 but not out of bit 7 or a carry out of bit 7 bit not from bit 6 otherwise overflow is cleared.

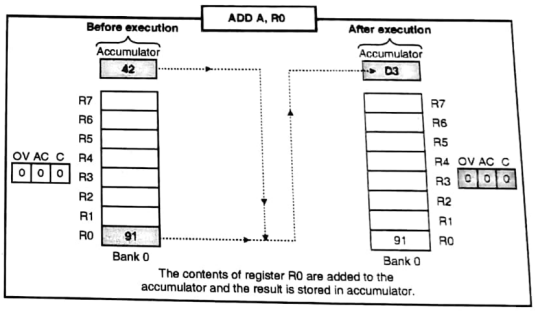


Fig. 4.4.1

☞ **(2) ADD A, direct**

Mnemonic	ADD A, direct
Machine cycles	1
Bytes	2
Addr. Mode	Direct addressing mode.

Function	ADD
Clock Pulses	12
Algorithm	(A) = (A) + direct
Flags	Flags are affected

Operation	(A) ← (A) + (direct)	This instruction will add the contents of the memory to location whose direct address is specified in the instruction with the accumulator contents. The result of addition will be stored in the accumulator.
Example	ADD A, 20H	Let the contents at memory location 20H be 45H (0100 0101B) and contents of accumulator be 77H (0111 0111 B), then the instruction ADD A, 20H will leave. A = BC H (1011 1100 B) with auxiliary flag cleared, carry flag cleared and overflow flag is set. Fig. 4.4.2 shows this.

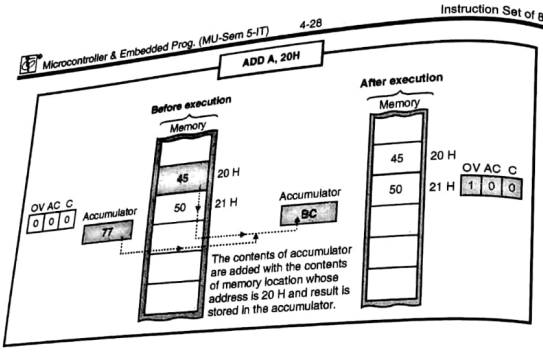


Fig. 4.4.2

(3) ADD A, @Ri

Mnemonic	ADD A, @Ri
Machine cycles	1
Bytes	1
Addr. Mode	register indirect addressing mode.

Function	ADD
Clock Pulses	12
Algorithm	(A) = (A) + ((Ri))
Flags	Flags are affected.

Operation	ADD (A) ← (A) + ((Ri))	This instruction will add the contents of memory location whose address is pointed by register Ri of the selected register bank with contents of the accumulator. The result of addition is stored in the accumulator.
Example	ADD A, @R0	Let R0 contain 35H. Let the contents of memory location 35H be 89H (1000 1001 B) and contents of accumulator be 95 H (1001 0101 B). The instruction ADD A, @R0 will add the contents of memory location pointed by register R0 i.e. 35H with the contents of accumulator i.e. 95H. The result stored in accumulator is 1EH (0001 1110B) with the auxiliary flag cleared and both the overflow and carry flag set.

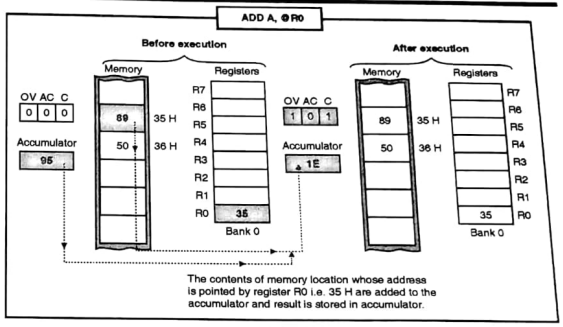


Fig. 4.4.3

(4) ADD A, # data

Mnemonic	ADD A, # data
Machine cycles	1
Bytes	2
Addr. Mode	immediate addressing mode

Function	ADD
Clock Pulses	12
Algorithm	(A) = (A) + data
Flags	Flags are affected.

Operation	ADD (A) ← (A) + data	This instruction will add the immediate 8 bit data with data in the accumulator. The result of addition is stored in the accumulator.
Example	ADD A, # 40H	Let A = 29 H (0010 1001 B) The instruction ADD A, # 40H will add the data 40H to contents of accumulator (29H). The result is stored accumulator (69 H) with auxiliary carry, carry and overflow flag cleared.

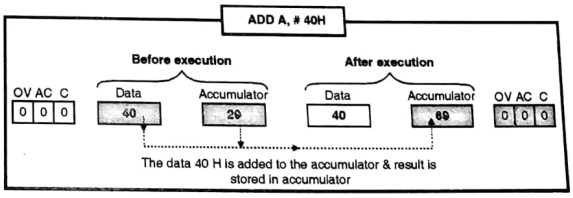


Fig. 4.4.4

4.4.2 ADDC A, <src-byte>

Mnemonic : ADDC A ,<src-byte>	Function : Add with carry
Algorithm : (A) = (A) + <src-byte> + carry	Operation : (A) ← (A) + <src-byte> + carry

- This instruction will add the byte variable indicated, the carry flag and the accumulator contents. The result of addition is stored in the accumulator.
- All the addressing modes can be used for source: an immediate number, a register, direct address, indirect address.
- Depending on the addressing modes, let us see the different combinations.

☞ **1. ADDC A, Rn**

Mnemonic	ADDC A, Rn	Function	Add with carry.
Machine cycles	1	Clock Pulses	12
Bytes	1	Algorithm	(A) = (A) + (Rn) + (CY)
Addr. Mode	Register addressing mode.	Flags	Flags are affected.

Operation	(A) ←(A) + (Rn) + (CY)	This instruction will add the contents of accumulator with the contents of register Rn of the selected register bank and carry flag. The result of addition is stored in accumulator.
Example	ADDC A, R1	Let R1 = 54H, A = 99H and CF = 1 then the instruction ADDC A, R1 will add the contents of accumulator with the contents of register R1 of the selected register bank and carry flag. The result stored in accumulator is EEH with the auxiliary carry, carry and overflow flag cleared.

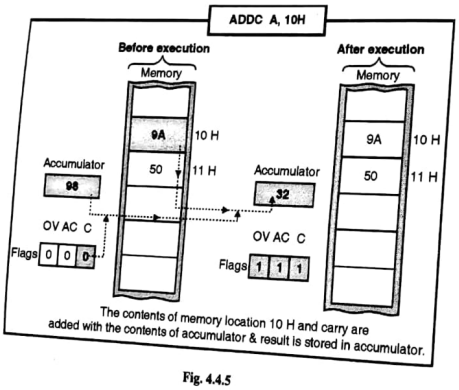


Fig. 4.4.5

☞ **2. ADDC A, direct**

Mnemonic	ADDC A, direct.	Function	Add with carry.
Machine cycles	1	Clock Pulses	12
Bytes	2	Algorithm	(A) = (A) + (direct) + CY.
Addr. Mode	Direct addressing mode.	Flags	Flags are affected.

Operation	(A) ← (A)+(direct) + CY	This instruction will add the contents of memory location whose direct address is specified in the instruction with the contents of accumulator and carry. The result of addition is stored in the accumulator.
Example	ADDC A, 10 H.	Let contents of memory location whose address is 10H be 9AH, contents of accumulator be 98H and contents of carry flag be 0. The instruction ADDC A, 10H will leave the accumulator with contents 32H with the auxiliary carry and carry flag set and the overflow flag cleared.

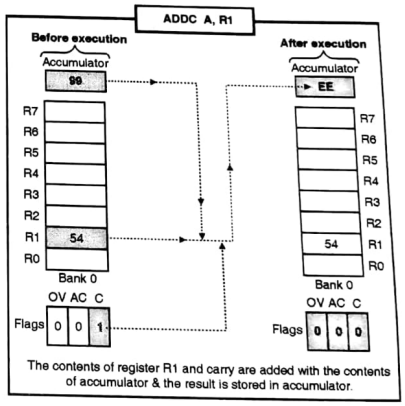


Fig. 4.4.6

☞ **3. ADDC A, @Ri**

Mnemonic	ADDC A, @Ri.	**Function**	Add with carry.
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	(A) = (A) + ((Ri)) + CY.
Addr. Mode	Register indirect addressing mode.	**Flags**	Flags are affected.

Operation	(A) ← (A) + ((Ri)) + CY.	– This instruction will add the contents of memory location pointed by register Ri of selected register bank with the accumulator and carry flag. – The result of addition is stored in the accumulator.
Example	ADDC A, @R0.	– Let R0 hold 30H, the contents of memory location 30H be 89H, contents of accumulator be 45H and contents of carry flag be 1, then the instruction ADDC A, @R0 will leave the accumulator with CFH and the carry, auxiliary carry and overflow flag will be cleared.

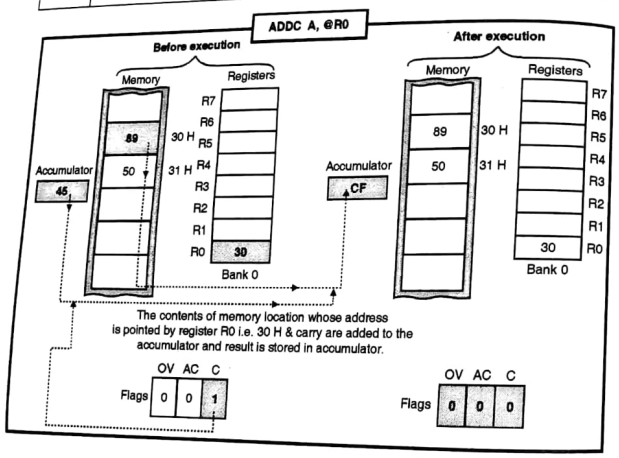


Fig. 4.4.7

☞ **4. ADDC A, # data**

Mnemonic	ADDC A, # data.	**Function**	Add with carry.
Machine cycles	1	**Clock Pulses**	12
Bytes	2	**Algorithm**	(A) = (A) + data + CY
Addr. Mode	Immediate addressing mode.	**Flags**	Flags are affected.

Operation	(A) ← (A) + data + CY.	This instruction will add the contents of accumulator with immediate data specified in the instruction along with carry.
Example	ADDC A, # 40H.	Let A = 50H, CY = 1 then the instruction ADDC A, # 40H will leave the accumulator with 91H and the carry, overflow and auxiliary flags are cleared.

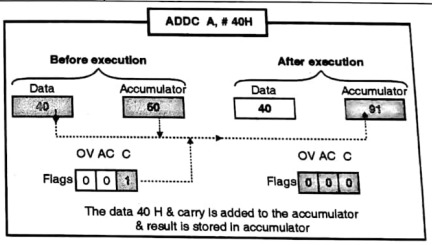


Fig. 4.4.8

4.4.3 SUBB A, <src-byte>

Mnemonic : SUBB A, <src-byte>	**Function :** Subtract with borrow.
Algorithm : (A) = (A) – <src-byte> – CY	**Operation :** (A) ← (A) – <src-byte> – CY

This instruction subtracts the indicated byte variable and the carry flag contents together, from the accumulator. The result is stored in the accumulator.

Note : The carry flag is treated as the borrow flag.

– All the addressing modes can be used as source : an immediate number, a register, direct address, indirect address.

– Depending on the addressing-modes, let us see the different combinations.

1. SUBB A, Rn

Mnemonic	SUBB A, Rn
Machine cycles	1
Bytes	1
Addr. Mode	register addressing mode.

Function	Subtract with borrow.
Clock Pulses	12
Algorithm	A = (A) – (Rn) – CY.
Flags	Flags are affected.

Operation	SUBB. (A) ← (A) – (Rn) – CY	This instruction will subtract the contents of register Rn of the current register bank and the contents of carry flag together, from the accumulator. The result is stored in the accumulator.
Example	SUBB A, R3	Let A = C9 H = (1100 1001) R3 = 54 H = (0101 0100 B) CY = 1 (set) The instruction SUBB A, R3 will subtract the contents of register R3 of selected register bank and contents of carry flag from the accumulator. So, the accumulator contains 74H with the carry and auxiliary flag cleared, while the overflow flag is set.

- Carry (borrow) flag is set if a borrow is needed for bit 7 and clears CY otherwise.
- Auxiliary carry flag is set if a borrow is needed for bit 3 otherwise it is cleared.
- The overflow flag is set if a borrow is needed for bit 6, but not into bit 7 or if a borrow is needed into bit 7, but not bit 6.
- When signed integers are subtracted the overflow flag indicates a negative number produced when negative value is subtracted from a positive value, or a positive number produced when a positive number is subtracted from a negative number.

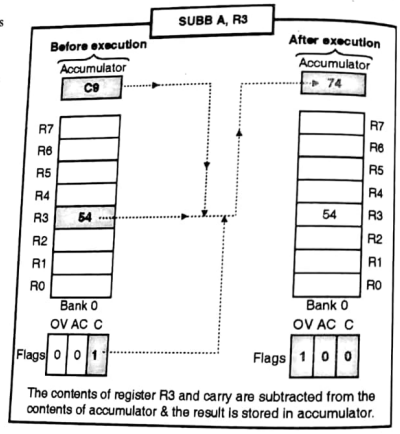


Fig. 4.4.9

2. SUBB A, direct

Mnemonic	SUBB A, direct.
Machine cycles	1
Bytes	2
Addr. Mode	Direct addressing mode.

Function	Subtract with borrow.
Clock Pulses	12
Algorithm	(A) = (A) – (direct) – CY
Flags	Flags are affected.

Operation	SUBB. (A) ← (A) – (direct) – CY.	This instruction will subtract the contents of memory location whose direct address is specified in the instruction and the contents of carry flag from the contents of accumulator. The result is stored in the accumulator.
Example	SUBB A, 45H.	Let contents at memory location 45 = C9 H (1100 1001 B) A = 54 H (0101 0100 B) CY = 1 Then the instruction SUBB A, 45H will leave the accumulator with 8A, the carry, auxiliary flag and the overflow flag are set.

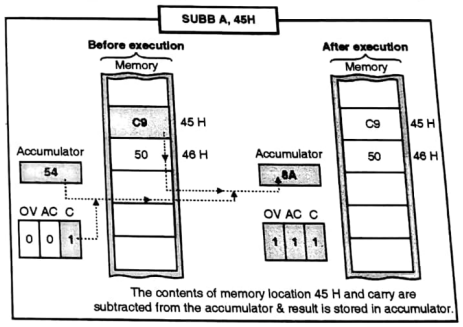


Fig. 4.4.10

3. SUBB A, @RI

Mnemonic	SUBB A, @Ri
Machine cycles	1
Bytes	1
Addr. Mode	Register Indirect Addressing mode.

Function	Subtract with borrow.
Clock Pulses	12
Algorithm	(A) = (A) – ((Ri)) – CY
Flags	Flags are affected.

Operation	(A) ← (A) – ((Ri)) – CY.	This instruction will subtract the contents of memory location pointed to by register Ri and contents of carry flag from the accumulator. The result of subtraction is stored in the accumulator.
Example	SUBB A, @ R1.	Let R1 = 30H and the contents of memory location 30H be 54H (0101 0100 B), and contents of accumulator = C9H (1100 1001 B) and CY (borrow) = 1, then the instruction SUBB A, @ R1 will leave the accumulator with 74H with the carry and auxiliary carry flag cleared and the overflow flag set.

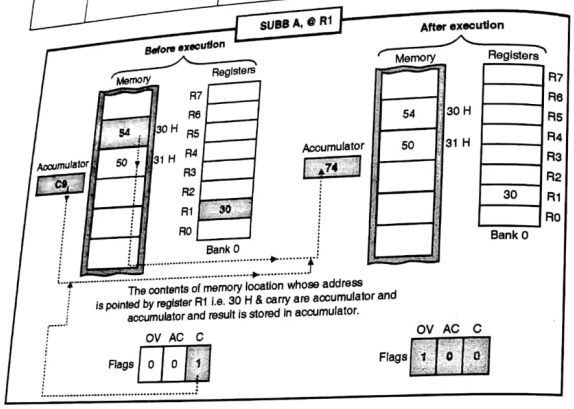


Fig. 4.4.11

4. SUBB A, # data

Mnemonic	SUBB A, # data.
Machine cycles	1
Bytes	2
Addr. Mode	Immediate addressing mode.

Function	Subtract with borrow.
Clock Pulses	12
Algorithm	(A) = (A) – data – CY.
Flags	Flags are affected.

Operation	SUBB (A) ← (A) – data – CY.	This instruction will subtract the data specified in the instruction and contents of carry flag from the contents of accumulator. The result will be stored in the accumulator.
Example	SUBB A, # 40H.	Let A = 50H, CY = 1 then the instruction SUBB A, # 40H will leave the accumulator with 0FH.

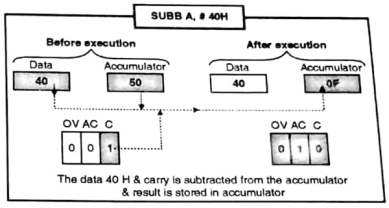


Fig. 4.4.12

4.4.4 INC <byte>

Mnemonic : INC <byte>	**Function :** Increment
Algorithm : <byte> = <byte> + 1	**Operation :** byte ← byte + 1.

- This instruction will increment the indicated variable by 1.
- If the byte value is FFH and if it is incremented, then the result will overflow to 00H.
- It supports three addressing modes, Register, direct and register-indirect.

Note : When the increment instruction operates on a port direct address, alter the latch of that port. (Since it is a read-modify-write operation)

Depending on different addressing modes let us see the different combinations.

1. INC Rn

Mnemonic	INC Rn
Machine cycles	1
Bytes	1
Addr. Mode	Register Addressing mode.

Function	Increment.
Clock Pulses	12
Algorithm	(Rn) = (Rn) + 1
Flags	No flags are affected.

Operation	(Rn) ← (Rn) + 1.	– This instruction will increment the contents of register Rn of selected register bank by 1. – The register Rn can also be accumulator.
Example	INC R5.	Let R5 = 0EH then instruction INC R5 will leave R5 = 0FH.

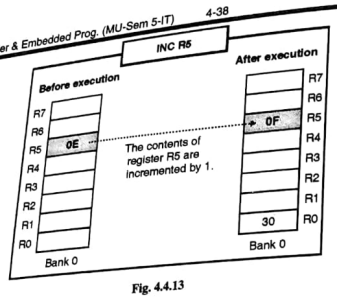


Fig. 4.4.13

☞ 2. **INC <direct>**

Mnemonic	INC <direct>.
Machine cycles	1
Bytes	2
Addr. Mode	Direct addressing mode.

Function	Increment.
Clock Pulses	12
Algorithm	<direct> = <direct> + 1
Flags	No flags are affected.

Operation	<direct> ← <direct> + 1.	This instruction will increment the contents of memory location whose direct address is specified in the instruction by 1.
Example	INC 40H.	Let the contents of memory location 40H be 22H, then the instruction INC 40H will increment the contents of memory location whose direct address is 40H by 1. i.e. location 40H = 23H.

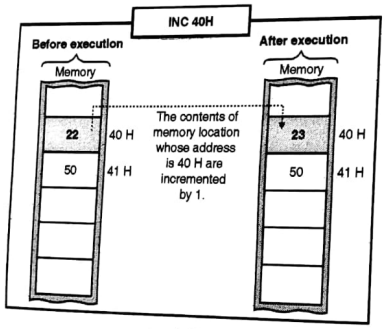


Fig. 4.4.14

☞ 3. **INC @RI**

Mnemonic	INC @Ri
Machine cycles	1.
Bytes	1.
Addr. Mode	Register-indirect addressing mode.

Function	Increment.
Clock Pulses	12
Algorithm	((Ri)) = ((Ri)) + 1.
Flags	No flags are affected.

Operation	((Ri)) ← ((Ri)) + 1.	This instruction will increment the contents of memory location that is pointed by register Ri by 1.
Example	INC @R1.	Let the contents of R1 = 45H, and contents of memory location 45H = 56H, then the instruction INC @ R1 will increment the contents of memory location pointed by register R1 i.e. 45H by 1. i.e. now memory location 45H will contain 57H.

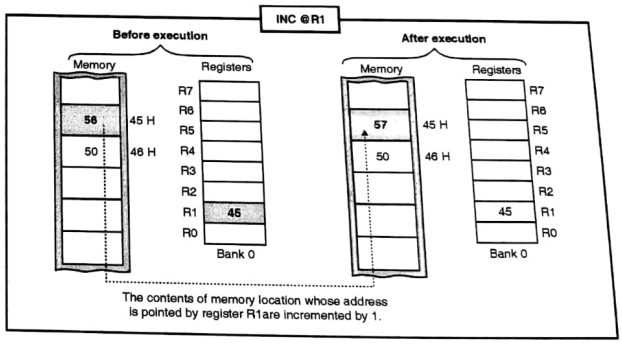


Fig. 4.4.15

4.4.5 INC DPTR

Mnemonic	INC DPTR.
Machine cycles	2
Bytes	1
Addr. Mode	Register addressing mode.

Function	Increment Data Pointer.
Clock Pulses	24
Algorithm	(DPTR) = (DPTR) + 1.
Flags	No flags are affected.

Operation	(DPTR) ← (DPTR) + 1.	– This instruction will increment the contents of Data Pointer by 1. – A 16 bit increment is performed. An overflow of the low-order byte of the data pointer (DPL) from 0FF H to 00 H will increment the high order byte (DPH). – DPTR is the only 16 bit register that is incremented.
Example	Let contents of DPH = 15 H and contents of DPL = FFH.	The instruction INC DPTR will cause DPH = 16H and DPL = 00H.

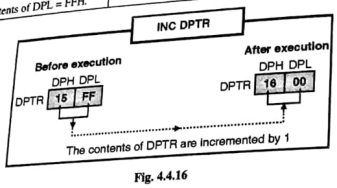


Fig. 4.4.16

4.4.6 DEC <byte>

Mnemonic :	DEC <byte>.	**Function :** Decrement.	
Algorithm :	<byte> = 0<byte> – 1. <byte> ← <byte> – 1.	**Operation :** DEC	

– This instruction will decrement the indicated variable by 1.

– An original value of 00H will underflow to FFH.

– Three operand addressing modes are allowed register, direct and register indirect addressing modes.

Note : The decrement instruction when used to modify an output port, alters the latch for that port. Depending on different addressing mode, let us see the different combinations.

☞ **1. DEC Rn**

Mnemonic	DEC Rn.	**Function**	Decrement.
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	(Rn) = (Rn) – 1.
Addr. Mode	Register addressing mode.	**Flags**	No flags are affected.

Operation	(Rn) ← (Rn) – 1.	– This instruction will decrement the contents of register Rn of the selected register bank by 1. – The registers can be any of the registers R0 – R7 of the selected register bank or the accumulator.
Example	DEC R5	Let R5 = 0EH, DEC R5 will leave R5 = 0DH.

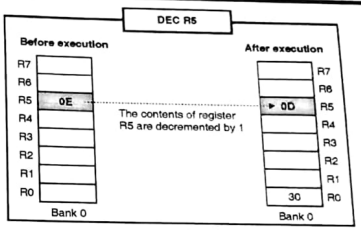


Fig. 4.4.17

☞ **2. DEC <direct>**

Mnemonic	DEC <direct>.	**Function**	Decrement.
Machine cycles	1	**Clock Pulses**	12
Bytes	2	**Algorithm**	<direct> = <direct> – 1.
Addr. Mode	Direct addressing mode.	**Flags**	No flags are affected.

Operation	<direct> ← <direct> – 1.	– This instruction will decrement the contents of memory location whose direct address is specified in the instruction by 1.
Example	DEC 55H	Let the contents of memory location 55H = 14H, then the instruction DEC 55H will decrement the contents of memory location 55H by 1 i.e., contents of memory location 55H = 13H.

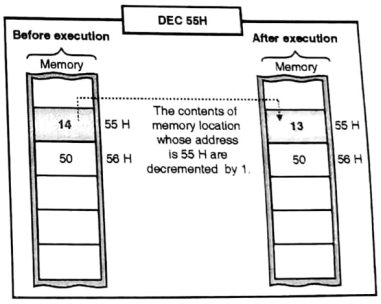


Fig. 4.4.18

3. **DEC @ Ri**

Mnemonic	DEC @ Ri	**Function**	Decrement.
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	((Ri)) = ((Ri)) – 1.
Addr. Mode	register-indirect addressing mode.	**Flags**	No flags are affected.

Operation	((Ri)) ← ((Ri)) – 1.	– This instruction will decrement the contents of memory location pointed by register Ri by 1.
Example	DEC @R0.	Let R0 = 51H and contents of memory location 51 = 1AH, then the instruction DEC @ R0 will leave the memory location 51H = 19H.

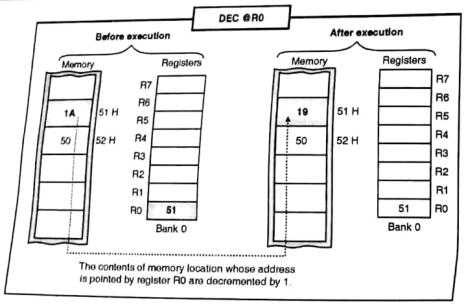


Fig. 4.4.19

4.4.7 MUL AB

➔ **(MU - Dec. 17)**

Q. 4.4.1 Describe the instruction of 8051, JNC and MUL with one example. (Ref. Secs. 4.7.12 and 4.4.7) **Dec. 17, 5 Marks**

Mnemonic	MUL AB.	**Function**	Multiply.
Machine cycles	4	**Clock Pulses**	48
Bytes	1	**Algorithm**	$(A)_{7-0} = (A) \times (B)$ $(B)_{15-8}$
Addr. Mode	Register Addressing mode.	**Flags**	Flags are affected.

Operation	MUL $(A)_{7-0} \leftarrow (A) \times (B)$ $(B)_{15-8}$	– This instruction multiplies an eight bit unsigned integer in the Accumulator and the B register. The low-order byte of the sixteen bit product is left in the accumulator, and the high-order byte in B. – The largest possible product is FE01 H when A = FFH and B = FFH. Then A = 01H and B = FEH after multiplication.
Example	MUL AB.	Let A = 50H, B = A0H MUL AB (50H) × (A0H) = (3200H) B = 32H, A = 00H with the overflow flag set and carry flag cleared.

Note : (1) There is no comma between A and B in the MUL instruction.
(2) Original contents of registers A and B are lost.

– The overflow flag is set if the product is greater than 255 decimal (FFH), otherwise it is cleared.
– The carry flag is always cleared.

4.4.8 DIV AB

Mnemonic	DIV AB	**Function**	Divide.
Machine cycles	4	**Clock Pulses**	48
Bytes	1	**Algorithm**	(A ÷ B) A = quotient B = Remainder.
Addr. Mode	Register Addressing mode.	**Flags**	Flags are affected.

Operation	DIV A (quotient) ← A ÷ B B(remainder)	– This instruction divides the unsigned number in accumulator with the unsigned number in register B. – Accumulator contains the quotient of the result and register B contains the remainder. – The contents of A and B, when division by 0 is attempted, are undefined.
Example	DIV AB	Let A = FBH and B = 12H then DIV AB will result. A = $(13)_{10}$ = 0DH (quotient), B = $(17)_{10}$ = 11H (remainder).

Note : (1) The original contents of A and B are lost. The registers A and B are used as source and destination for the division operation.
(2) There is no comma between A and B in the DIV instruction.

– The carry and overflow flags are always cleared. But if, A contains some number, B contains 00H. Then if an attempt is done DIV AB then the values contained in register A and B are undefined. The overflow flag will be set and carry flag will be cleared. i.e. overflow flag is set when a divide by zero is attempted.

4.4.9 DAA

Mnemonic	DA A
Machine cycles	1
Bytes	1
Addr. Mode	Register Specific Addressing mode.

Function	Decimal Adjust Accumulator for Addition
Clock Pulses	12
Algorithm	Contents of Accumulator are BCD if $[(A_{3-0}) > 9]$ then $(A_{3-0}) = (A_{3-0}) + 6$ AND if $[(A_{7-4}) > 9]$ then $(A_{7-4}) =$ $(A_{7-4}) + 6$
Flags	Flags are affected.

Operation	**DA A**	– This instruction adjusts the sum of two packed BCD numbers to an eight bit value i.e. producing two four bit digits. – To perform the addition any ADD or ADDC instruction can be used. – The rules of BCD addition are (i) if the number is greater than 9, add 6 (ii) if the auxiliary carry or carry is generated, add 6. This is done in order to produce a valid BCD result.
Example	**DA A**	Let A = 56H (packed BCD) R3 = 67H (packed BCD) CY = 1 then, ADDC A, R3 A → 5 6 H R3 → + 6 7 H CY → + 1 B E H DA A As B > 9, E > 9, 6 should be added to both B E H 6 6 H 2 4 with CY = 1.

– If $(A_{3-0} > 9)$ or AC = 1 then 6 is added to accumulator, so that it produces a proper BCD digit in the lower nibble. This internal addition may set the auxiliary carry flag if there is a carry-out of the bit 3, propagating into higher order bits.

– If $(A_{7-4} > 9)$ or CY = 1 then 6 is added to produce proper BCD digit in the high-order nibble. If there is carry-out of high-order bits then carry flag will again be set.

– The carry flag if set thus, indicates whether the sum of two BCD numbers is greater than 99, allowing multiple precision decimal addition.

– The overflow flag is not affected.

Note : (1) All the above conversions are done in one instruction cycle.
(2) DA A performs decimal-conversions by adding 00H, 06H, 60H or 66H to the accumulator, depending on the accumulator and PSW (i.e. flags) conditions.
(3) DA A cannot simply convert a hexadecimal number in the accumulator to BCD notation.
(4) DA A does not apply to decimal subtraction.

Syllabus Topic : Programming Based on Logical Operations

4.5 Logical Instructions

4.5.1 ANL <dest-byte>, <src-byte>

Mnemonic	ANL <dest-byte>, <src-byte>
Function	Logical AND for byte variables.
Algorithm	<dest-byte> = <dest-byte> ∧ <src-byte>
Operation	<dest-byte> ← <dest-byte> ∧ <src-byte>

– This instruction performs bit wise logical AND operation between the destination and the source byte. The result is stored at the destination byte.

– The source and destination support four addressing modes : register, direct, register-indirect and immediate addressing modes. These 4 addressing modes support six combinations.

– Let us see these combinations.

Note : When this instruction is used to modify an output port, the value used as original port data will be read from the output data latch, not the input pins, as it will be a read-modify-write instruction.

☞ **1. ANL A, Rn**

Mnemonic	ANL A, Rn
Machine cycles	1
Bytes	1
Addr. Mode	Register addressing mode.

Function	Logical AND for byte variables
Clock Pulses	12
Algorithm	(A) = (A) ∧ (Rn)
Flags	No flags are affected.

Operation	ANL (A) ← (A) ∧ (Rn)	This instruction will perform bitwise logical AND operation between the contents of accumulator and register Rn of the selected register bank. The result will be stored in the accumulator.
Example	ANL A, R5	Let A = FFH, R5 = 15H, then the instruction ANL A, R5 will logically AND contents of accumulator with the contents of register R5. The result of ANDing is stored in accumulator i.e. 15H.

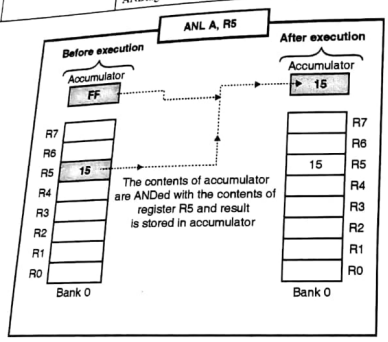


Fig. 4.5.1

2. ANL A, Direct

Mnemonic	ANL A, Direct	**Function**	Logical AND for byte variables.
Machine cycles	1	**Clock Pulses**	12
Bytes	2	**Algorithm**	(A) = (A) ∧ (Direct)
Addr. Mode	Direct Addressing mode.	**Flags**	No flags are affected.

Operation	ANL (A) ← (A) ∧ (direct)	– This instruction will bitwise logically AND the contents of accumulator with the contents of memory location whose direct address is specified in the instruction. The result will be stored in the accumulator.
Example	ANL A, 50H.	Let A = 10H, contents of memory location 50H be 80H, then the instruction ANL A, 50H will logically AND the contents of accumulator bitwise with the contents of memory location 50H. The result in accumulator will be 00H.

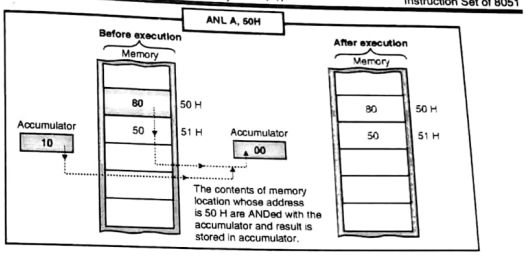


Fig. 4.5.2

3. ANL A, @ Ri

Mnemonic	ANL A, @ Ri	**Function**	Logical AND for byte variables
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	(A) = (A) ∧ ((Ri))
Addr. Mode	Register-indirect addressing mode.	**Flags**	No flags are affected

Operation	ANL (A) ← (A) ∧ ((Ri))	This instruction will bitwise logically AND the contents of accumulator with the contents of memory location pointed by register Ri of the selected register bank. The result will be stored in the accumulator.
Example	ANL A, @ R1.	Let A = 40H and R1 = 0AH, contents of memory location 0AH = CAH, then the instruction ANL A, @R1 will bitwise AND the contents of accumulator with the contents of memory location pointed by register R1 of the selected register bank i.e. CAH. The result in accumulator will be 40H.

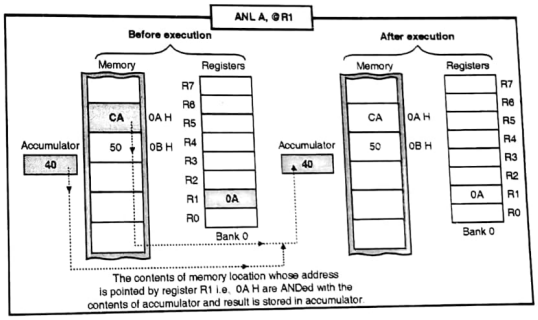


Fig. 4.5.3

4. ANL Direct, A

Mnemonic	ANL Direct, A
Machine cycles	1
Bytes	2
Addr. Mode	Direct Addressing mode.

Function	Logical AND for byte variables.
Clock Pulses	12
Algorithm	(direct) = (direct) ∧ (A)
Flags	No flags are affected.

Operation	ANL (direct) ← (direct) ∧ A	– This instruction will bitwise logically AND the contents of memory location whose direct address is specified in the instruction with the contents of the accumulator. The result will be stored in the memory location whose direct address is specified in the instruction.
Example	ANL 30H, A.	Let contents of memory location 30H = 57H, contents of accumulator = 10H then the instruction ANL 30H, A will bitwise logically AND the contents of memory location whose direct address is 30H with the contents of accumulator. The result in memory location 30H will be 10H.

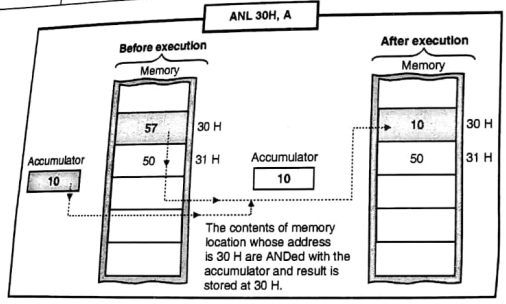


Fig. 4.5.4

5. ANL A, # data

Mnemonic	ANL A, # data.
Machine cycles	1
Bytes	2
Addr. Mode	Immediate Addressing mode.

Function	Logical AND for byte variables.
Clock Pulses	12
Algorithm	(A) = (A) ∧ data
Flags	No flags are affected.

Operation	ANL (A) ← (A) ∧ (data)	– This instruction will bitwise logically AND the contents of accumulator with the immediate data specified in the instruction. The result will be stored in the accumulator.
Example	ANL A, # 57H.	Let A = 22H (0010 0010 B), then the instruction ANL A, #57H will bitwise logically AND the contents of accumulator 22H with immediate data 57H. The result in accumulator is 02H.

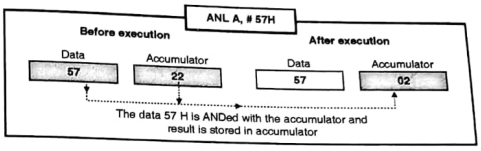


Fig. 4.5.5

6. ANL Direct, # data

Mnemonic	ANL Direct, # data.
Machine cycles	2
Bytes	3
Addr. Mode	Immediate Addressing mode.

Function	Logical AND for byte variables.
Clock Pulses	24
Algorithm	(direct) = (direct) ∧ data
Flags	No flags are affected.

Operation	ANL (direct) ← (direct) ∧ data.	This instruction will bitwise logically AND the contents of memory location whose direct address is specified in the instruction with the immediate data. The result will be stored in the memory location whose direct address is specified.
Example	ANL 54H, # 33H	Let the contents of memory location 54H be 25H. The instruction ANL 54H, #33H will logically AND the contents of memory location 54H i.e. 25H with immediate data i.e. 33H. The result will be stored in the memory location 54H i.e. 21H.

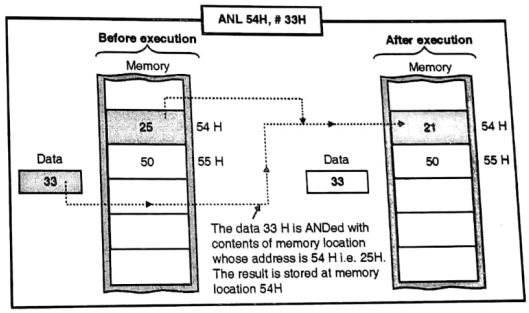


Fig. 4.5.6

4.5.2 ORL <dest-byte>, <src-byte>

Mnemonic : ORL <dest-byte>, <src-byte>	**Function :** Logical OR for byte variables.
Algorithm : <dest-byte> = <dest-byte> ∨ <src-byte>	**Operation :** ORL <dest-byte> ← <dest-byte> ∨ <src-byte>

- This instruction will bitwise logically OR the contents of source byte with the contents of the destination byte. The result of logical ORing operation will be stored in the destination byte.
- The source and destination support four addressing modes : register, direct, register-indirect and immediate addressing modes. These four addressing modes support six combinations. Let us see these combinations.

Note : When this instruction is used to modify an output port, the value used as original port data will be read from the output data latch, not the input pins. Since, it is a read-modify-write operation.

1. ORL A, Rn

Mnemonic	ORL A, Rn
Machine cycles	1
Bytes	1
Addr. Mode	Register addressing mode.

Function	Logical OR for byte variables.
Clock Pulses	12
Algorithm	(A) = (A) ∨ (Rn)
Flags	No flags are affected.

Operation	ORL (A) ← (A) ∨ (Rn)	This instruction will perform bitwise logical OR operation between the contents of accumulator and the register Rn of the selected register bank. The result will be stored in the accumulator.
Example	ORL A, R5	Let A = FFH, R5 = 15H, then the instruction ORL A, R5 will logically OR the contents of accumulator with the contents of register R5. The result of ORing stored in the accumulator is FFH.

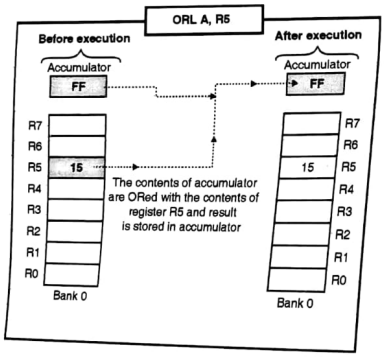


Fig. 4.5.7

2. ORL A, Direct

Mnemonic	ORL A, Direct.
Machine cycles	1
Bytes	2
Addr. Mode	Direct Addressing mode.

Function	Logical OR for byte variables.
Clock Pulses	12
Algorithm	(A) = (A) ∨ (Direct)
Flags	No flags are affected.

Operation	ORL (A) ← (A) ∨ (Direct)	This instruction will perform bitwise logical OR operation between the contents of accumulator and the contents of memory location whose direct address is specified in the instruction. The result will be stored in the accumulator.
Example	ORL A, 50H.	Let A = 10H, contents of memory location 50H = 80H. The instruction ORL A, 50H will bitwise logically OR the contents of accumulator 10H with contents of memory location 50H i.e. 80H. The result stored in the accumulator will be 90H.

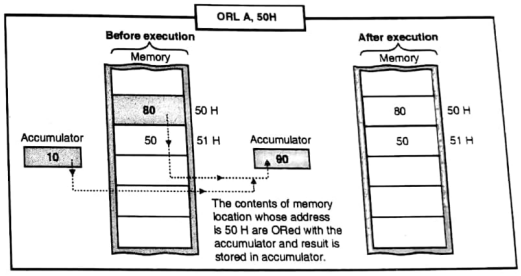


Fig. 4.5.8

3. ORL A, @Ri

Mnemonic	ORL A, @Ri
Machine cycles	1
Bytes	1
Addr. Mode	Register-indirect addressing mode.

Function	Logical OR for byte variables.
Clock Pulses	12
Algorithm	(A) = (A) ∨ ((Ri))
Flags	No flags are affected.

Operation	ORL (A)←(A) ∨ ((Ri))	This instruction will bitwise logically OR the contents of accumulator with the contents of memory location pointed by register Ri of the selected register bank. The result will be stored in the accumulator.

Example	ORL A, @R1	Let A = 40H, and R1 = 0AH. Let the contents of memory location 0AH = CAH. The instruction ORL A, @ R1 will bitwise logically OR the contents of accumulator (40H) with the contents of memory location pointed by register R1 i.e. CAH. The result in accumulator will be CAH.

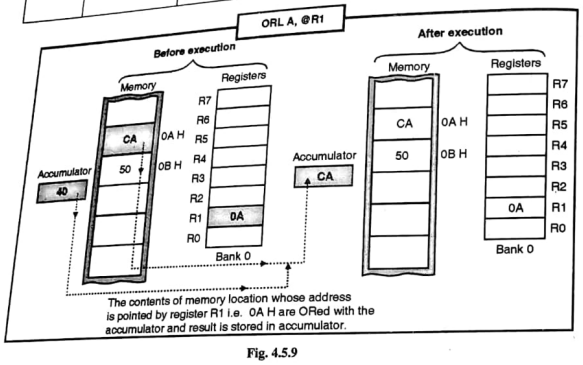


Fig. 4.5.9

4. ORL Direct, A

Mnemonic	ORL Direct, A
Machine cycles	1
Bytes	2
Addr. Mode	Direct Addressing mode.

Function	Logical OR for byte variables.
Clock Pulses	12
Algorithm	(Direct) = (Direct) ∨ (A)
Flags	No flags are affected.

Operation	ORL (Direct) ← (Direct) ∨ (A)	This instruction will bitwise logically OR the contents of memory location whose direct address is specified in the instruction with the contents of the accumulator. The result will be stored in the memory location whose direct address is specified in the instruction.
Example	ORL 30H, A	Let the contents of memory location 30H = 57H, contents of accumulator = 64H. The instruction ORL 30H, A, will bitwise logically OR the contents of memory location whose address is 30H i.e. 57H with the contents of accumulator (64H). The result in memory location 30H will be 77H.

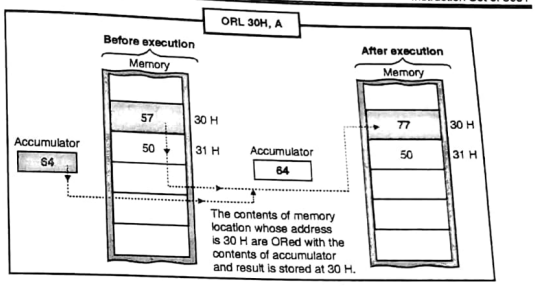


Fig. 4.5.10

5. ORL A, # data

Mnemonic	ORL A, # data
Machine cycles	1
Bytes	2
Addr. Mode	Immediate Addressing mode.

Function	Logical OR for byte variables.
Clock Pulses	12
Algorithm	(A) = (A) ∨ data
Flags	No flags are affected.

Operation	ORL (A) ← (A) ∨ data	This instruction will bitwise logically OR the contents of accumulator with the immediate data specified in the instruction. The result will be stored in the accumulator.
Example	ORL A, # 57H	Let A = 22H. The instruction ORL A, # 57H will bitwise logically OR the contents of accumulator 22H with the immediate data 57H. The result in accumulator will be 77H.

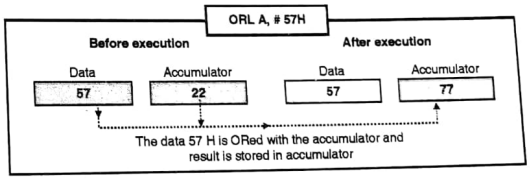


Fig. 4.5.11

☞ **6. ORL Direct, # data**

Mnemonic	ORL Direct, # data
Machine cycles	2
Bytes	3
Addr. Mode	Immediate addressing mode.

Function	Logical OR for byte variables.
Clock Pulses	24
Algorithm	(Direct) = (Direct) ∨ data
Flags	No flags are affected.

Operation	ORL (Direct) ← (Direct) ∨ data	This instruction will bitwise logically OR the contents of memory location whose direct address is specified in the instruction with the immediate data. The result will be stored in the memory location whose direct address is specified.
Example	ORL 54H, # 33H	Let the contents of memory location 54H be 25H. The instruction ORL 54H, # 33H will logically OR the contents of memory location 54H i.e. 25H with immediate data i.e. 33H. The result stored in memory location 54H is 37H.

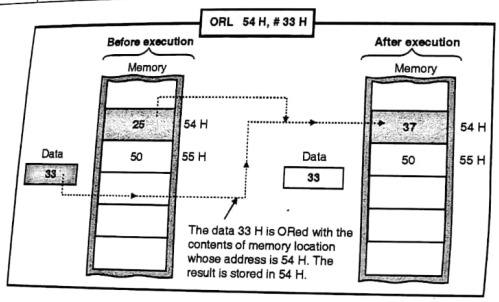


Fig. 4.5.12

4.5.3 XRL <dest-byte>, <src-byte>

Mnemonic : XRL <dest-byte>, <src-byte>	Function : Logical Exclusive- OR for byte variables.
Algorithm : <dest-byte> = <dest-byte> ⊕ <src-byte>	Operation : XRL <dest-byte> ← <dest-byte> ⊕ <src-byte>

– This instruction performs bitwise Exclusive-OR operation between the destination byte and the source byte. The result of operation will be stored in the destination byte.

– The source and destination support four addressing : register, direct, register-indirect and immediate addressing modes. The four addressing modes support six combinations. Let us see these combinations.

Note : When this instruction, is used to modify an output port, the value used as original port data will be read from the output data latch, not the input pins. Since it is read-modify-write operation.

☞ **1. XRL A, Rn**

Mnemonic	XRL A, Rn
Machine cycles	1
Bytes	1
Addr. Mode	Register addressing mode.

Function	Logical Exclusive-OR for byte variables.
Clock Pulses	12
Algorithm	(A) = (A) ⊕ (Rn)
Flags	No flags are affected.

Operation	XRL (A) ← (A) ⊕ (Rn)	This instruction will perform bitwise logical Exclusive-OR operation between the contents of accumulator and the contents of register Rn of the selected register bank. The result will be stored in the accumulator.
Example	XRL A, R5	Let A = FFH, R5 = 15H, then the instruction XRL A, R5 will logically EX–OR the contents of accumulator with the contents of register R5. The result stored in accumulator is EAH.

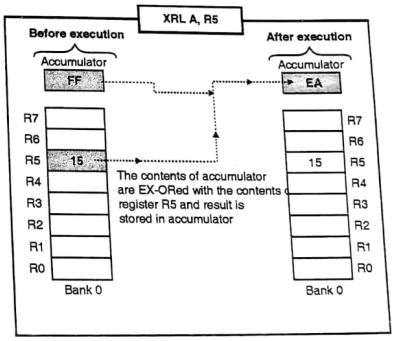


Fig. 4.5.13

☞ **2. XRL A, Direct**

Mnemonic	XRL A, Direct
Machine cycles	1
Bytes	2
Addr. Mode	Direct Addressing mode.

Function	Logical Exclusive-OR for byte variables.
Clock Pulses	12
Algorithm	(A) = (A) ⊕ (Direct)
Flags	No flags are affected.

Operation	XRL (A) ← (A) ⊕ (Direct)	This instruction will perform bitwise logical EX-OR operation between the contents of accumulator and the contents of memory location whose direct address is specified in the instruction. The result will be stored in the accumulator.
Example	XRL A, 50H	Let A = 10H, contents of memory location 50H = 80H. The instruction XRL A, 50H will bitwise logically EX-OR the contents of accumulator 10H with the contents of memory location 50H i.e. 80H. The result stored in the accumulator will be 90H.

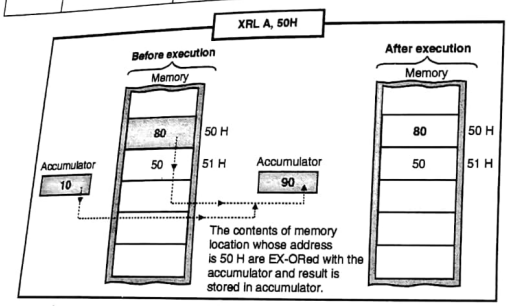


Fig. 4.5.14

3. XRL A, @Ri

Mnemonic	XRL A, @Ri
Machine cycles	1
Bytes	1
Addr. Mode	Register-indirect addressing mode.

Function	Logical X-OR for byte variables.
Clock Pulses	12
Algorithm	(A) = (A) ⊕ ((Ri))
Flags	No flags are affected.

Operation	XRL (A) ← (A) ⊕ ((Ri))	This instruction will bitwise logically EX-OR the contents of accumulator with the contents of memory location pointed by register Ri of the selected register bank. The result will be stored in the accumulator.
Example	XRL A, @R1	Let A = 40H and R1 = 0AH. Let the contents of memory location 0A = CAH. The instruction XRL A, @ R1 will bitwise logically EX-OR the contents of accumulator (40H) with the contents of memory location pointed by register R1 i.e. CAH. The result stored in the accumulator will be 8AH.

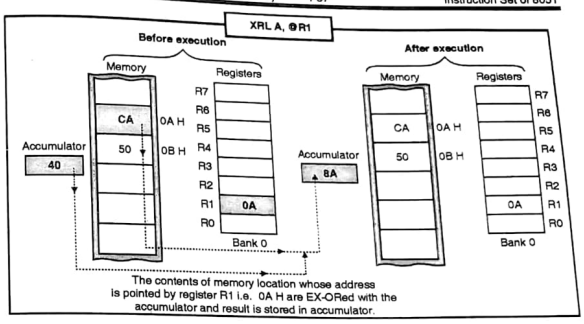


Fig. 4.5.15

4. XRL Direct, A

Mnemonic	XRL Direct, A
Machine cycles	1
Bytes	2
Addr. Mode	Direct Addressing mode.

Function	Logical Exclusive OR for byte variables.
Clock Pulses	12
Algorithm	(Direct) = (Direct) ⊕ (A)
Flags	No flags are affected.

Operation	XRL (Direct) ← (Direct) ⊕ (A)	This instruction will bitwise logically EX-OR the contents of memory location whose direct address is specified in the instruction with the contents of the accumulator. The result will be stored in the memory location whose direct address is specified in the instruction.
Example	XRL 30H, A	Let contents of memory location 30H = 57H, A = 64H. The instruction XRL 30H, A will bitwise logically EX-OR the contents of memory location whose address is 30H i.e. 57H with the contents of accumulator (64H). The result in memory location 30H will be 33H.

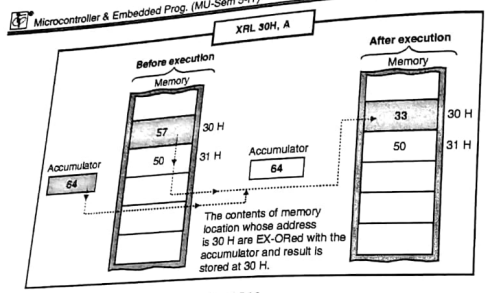


Fig. 4.5.16

☛ **5. XRL A, #data**

Mnemonic	XRL A, #data
Machine cycles	1
Bytes	2
Addr. Mode	Immediate Addressing mode.

Function	Logical Exclusive OR for byte variables.
Clock Pulses	12
Algorithm	(A) = (A) ⊕ (Data)
Flags	No flags are affected.

Operation	XRL (A) ← (A) ⊕ Data
	This instruction will bitwise logically EX-OR the contents of accumulator with the immediate data specified in the instruction. The result will be stored in the accumulator.
Example	XRL A, #57H
	Let A = 22H The instruction XRL A, #57H will bitwise logically EX-OR the contents of accumulator 22H with the immediate data 57H. The result in accumulator will be 75H.

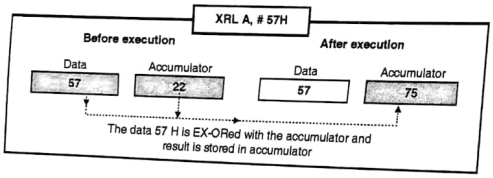


Fig. 4.5.17

☛ **6. XRL Direct, #data**

Mnemonic	XRL Direct, #data
Machine cycles	2
Bytes	3
Addr. Mode	Immediate Addressing mode.

Function	Logical Exclusive-OR for byte variables.
Clock Pulses	24
Algorithm	(Direct) = (Direct) ⊕ Data
Flags	No flags are affected.

Operation	XRL (Direct) ← (Direct) ⊕ Data
	This instruction will bitwise logically EX-OR the contents of memory location whose direct address is specified in the instruction with the immediate data. The result will be stored in the memory location whose direct address is specified.
Example	XRL 54H, #33H
	Let the contents of memory location 54H be 25H. The instruction XRL 54H, # 33H will logically EX-OR the contents of memory location 54H i.e. 25H with immediate data i.e. 33H. The result stored in memory location 54H is 16H.

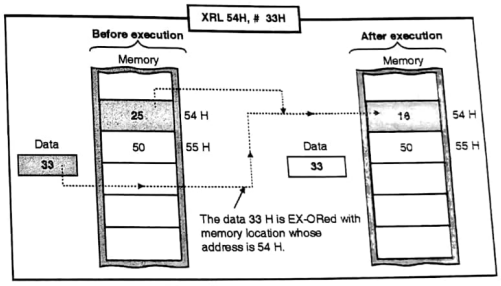


Fig. 4.5.18

4.5.4 CLR A

Mnemonic	CLR A
Machine cycles	1
Bytes	1
Addr. Mode	Register specific addressing mode.

Function	Clear Accumulator
Clock Pulses	12
Algorithm	A = 0
Flags	No flags are affected.

Operation	CLR A ← 0	This instruction will clear all the bits of accumulator to zero.
Example	CLR A	Let A = 77H. The instruction CLR A will leave the accumulator set to 00H.

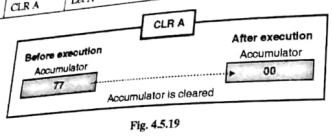


Fig. 4.5.19

4.5.5 CPL A

Mnemonic	CPL A
Machine cycles	1
Bytes	1
Addr. Mode	Register specific addressing mode.

Function	Complement Accumulator.
Clock Pulses	12
Algorithm	$A = \bar{A}$
Flags	No flags are affected.

Operation	CPL $A \leftarrow \bar{A}$	– This instruction will complement all the bits of the accumulator i.e. 1's complement of the number in accumulator is taken. – The bits which previously contained a one are changed to zero and vice versa.
Example	CPL A	Let A = 57H (0101 0111 B). The instruction CPL A will complement all the bits in accumulator. So, now the accumulator will contain (1010 1000 B) A8H.

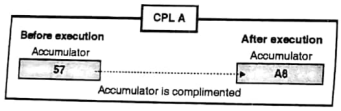


Fig. 4.5.20

4.5.6 RL A

Mnemonic	RL A
Machine cycles	1
Bytes	1
Addr. Mode	Register Specific Addressing mode.

Function	Rotate Accumulator left.
Clock Pulses	12
Algorithm	$(A_{n+1}) = (A_n)$ where n = 0 to 6 $(A_0) = (A_7)$
Flags	No flags are affected.

Operation	RL $(A_{n+1}) \leftarrow (A_n)$ where n = 0 to 6 $(A_0) \leftarrow A_7$	– This instruction will rotate the eight bits in the accumulator by one bit to the left. – Bit 7 is rotated into the Bit 0 position.
Example	RL A	Let A = 58H (0101 1000 B). The instruction RLA will rotate the accumulator by one bit to the left. So now accumulator contains (1011 0000 B) i.e. B0H

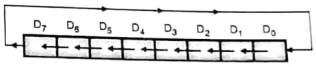


Fig. 4.5.21 : RL A

4.5.7 RLC A

Mnemonic	RLC A
Machine cycles	1
Bytes	1
Addr. Mode	Register specific addressing mode.

Function	Rotate Accumulator Left through Carry.
Clock Pulses	12
Algorithm	$(A_{n+1}) = (A_n)$ where n = 0 – 6 $(A_0) = (CY)$ $(CY) = A_7$
Flags	Except carry, no other flags are affected.

Operation	RLC $(A_{n+1}) \leftarrow (A_n)$ where n = 0 – 6 $(A_0) \leftarrow (CY)$ $(CY) \leftarrow (A_7)$	– This instruction will rotate the eight bits in the accumulator and the carry flag together by one bit to the left. – Bit 7 will move into carry and original carry will move to Bit 0 position. – Fig. 4.5.22 shows this
Example	RLC A	Let A = 72H (0111 0010 B), and CY = 1. The instruction RLC A leaves the accumulator holding the value (1110 0101 B) i.e. E5 H and CY = 0

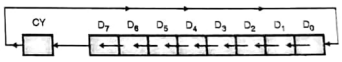


Fig. 4.5.22 : RLC A

4.5.8 RR A

Mnemonic	RR A
Machine cycles	1
Bytes	1
Addr. Mode	Register Specific Addressing mode.

Function	Rotate Accumulator Right.
Clock Pulses	12
Algorithm	$(A_n) = (A_{n+1})$ where n = 0 to 6 $(A_7) = (A_0)$
Flags	No flags are affected.

Operation	RR $(A_n) \leftarrow (A_{n+1})$ where n = 0 to 6 $(A_7) \leftarrow A_0$	– This instruction will rotate the eight bits in the accumulator by one position to the right. – Bit 0 is rotated into bit 7 position. – Fig. 4.5.23 shows this.
Example	RR A	– Let A = 75H (0111 0101 B). The instruction RR A will leave the accumulator holding value BAH (1011 1010 B)

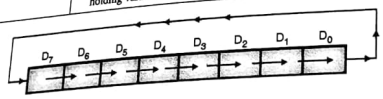


Fig. 4.5.23 : RR A

4.5.9 RRC A

Mnemonic	RRC A
Machine cycles	1
Bytes	1
Addr. Mode	Register Specific Addressing mode.

Function	Rotate Accumulator Right through carry.
Clock Pulses	12
Algorithm	$(A_n) = (A_{n+1})$ where n = 0 to 6 $(A_7) = (CY)$ $(CY) = A_0$
Flags	Except carry, no other flags are affected.

Operation	RRC $(A_n) \leftarrow (A_{n+1})$ where n = 0 to 6 $(A_7) \leftarrow (CY)$ $(CY) \leftarrow A_0$.	– This instruction will rotate the eight bits in accumulator and the carry flag together by one bit position to the right. – Bit 0 moves into the carry flag, original carry flag contents move into bit 7. – Fig. 4.5.24 shows this
Example	RRC A	Let A = 85H (1000 0101 B), and CY = 1 then the instruction RRC A will leave the accumulator holding C2H (1100 0010) and CY = 1

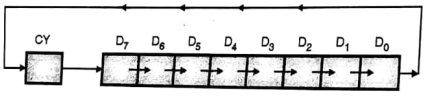


Fig. 4.5.24 : RRC A

4.5.10 SWAP A

→ (MU - May 15)

Q. 4.5.1 Describe the instruction of 8051, SWAP A with one example. (Ref. Sec. 4.5.10) **May 15, 2 Marks**

Mnemonic	SWAP A
Machine cycles	1
Bytes	1
Addr. Mode	Register Specific addressing mode.

Function	Swap nibbles within the Accumulator.
Clock Pulses	12
Algorithm	$(A_{3-0}) = (A_{7-4})$ $(A_{7-4}) = (A_{3-0})$
Flags	No flags are affected.

Operation	SWAP $(A_{3-0}) \leftrightarrow (A_{7-4})$	– This instruction interchanges the low order and high order nibbles of the accumulator. – The operation can also be thought of as a four bit rotate instruction.
Example	SWAP A	Let A = 87H (1000 0111 B). The instruction SWAP A leaves the accumulator holding the value 78H (0111 1000 B).

4.6 Bit Level Operations

4.6.1 CLR Bit

Mnemonic	CLR bit
Machine cycles	1
Bytes	1 if carry specific 2 if directly addressable bit.
Addr. Mode	If operated on carry flag then register addressing mode otherwise direct addressing mode.

Function	Clear bit
Clock Pulses	12
Algorithm	(Bit) = 0
Flags	Except carry, no other flags are affected if it is carry specific. If it is direct no flags are affected.

Operation	CLR (Bit) ← 0.	This instruction will clear the indicated bit. CLR can operate on carry flag or any directly addressable bit.
Example	CLR P2.3	Let Port 2 has previously been written with ADH (1010 1101 B). The instruction CLR P2.3 will clear the 3rd bit of Port 2 leaving Port 2 with A5H (1010 0101 B).

4.6.2 SETB Bit

Mnemonic	SETB bit
Machine cycles	1
Bytes	1 if carry specific 2 if directly addressable bit.
Addr. Mode	register addressing mode if carry specific otherwise direct addressing mode.

Function	Set Bit
Clock Pulses	12
Algorithm	(Bit) = 1
Flags	Except carry, no other flags are affected if it is carry specific. If it is direct no flags are affected.

Operation	SETB (Bit) ← 1	– This instruction will set the indicated bit. – SET can operate on carry flag or any directly addressable bit.
Example	SETB P2.3	Let Port 2 has previously been written by A5H (1010 0101 B). The instruction SETB P2.3 will set the 3rd bit of Port 2, leaving Port 2 with ADH (1010 1101 B).

4.6.3 CPL Bit

Mnemonic	CPL bit
Machine cycles	1
Bytes	1 if carry specific 2 if directly addressable bit.
Addr. Mode	register addressing mode if carry specific otherwise direct addressing mode.

Function	Complement bit
Clock Pulses	12
Algorithm	(Bit) = $(\overline{Bit})$
Flags	Except carry, no other flags are affected if it is carry specific. If direct no flags are affected.

Operation	CPL (Bit) ← $(\overline{Bit})$	– This instruction will complement the bit variable specified. A bit which had been a one is changed to zero and vice versa. – CPL can operate on carry flag or any directly addressable bit.
Example	CPL P2.5	Let Port 2 has previously been written by A5H (1010 0101 B). The instruction CPL P 2.5 will complement the 5th bit of Port 2. Port 2 will hold 85H (1000 0101 B).

4.6.4 ANL C, <src-bit>

Mnemonic	ANL C, <src-bit>
Machine cycles	2
Bytes	2
Addr. Mode	Direct Addressing mode.

Function	Logical AND for bit variables.
Clock Pulses	24
Algorithm	(C) = (C) ∧ (src-bit)
Flags	Except carry, no other flags are affected.

Operation	ANL (C) ← (C) ∧ <src-bit>	– This instruction will logically AND the specified bit with the carry bit. The result is stored in the carry bit. – If the Boolean value of the source bit is zero, then this instruction will clear the carry flag, otherwise it will leave the carry flag in its current state. – A slash ("/") preceding the operand (i.e. ANL C, /<src-bit>) in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is unaffected.
Example	ANL C, ACC.4 ANL C, / ACC.4	– Let C = 1, A = 11H (0001 0001 B) then the instruction ANL C, ACC.4 will logically AND the contents of carry with accumulator Bit 4 leaving the C = 1. – The instruction ANL C,/ACC. 4 will logically AND the contents of carry with complement of accumulator Bit 4 leaving the carry C = 0.

4.6.5 ORL C, <src-bit>

Mnemonic	ORL C, <src-bit>
Machine cycles	2
Bytes	2
Addr. Mode	Direct Addressing mode.

Function	Logical-OR for bit variables.
Clock Pulses	24
Algorithm	(C) = (C) ∨ <src-bit>
Flags	Except carry, no other flags are affected.

Operation	ORL (C) ← (C) ∨ <src-bit>	– This instruction will logically OR the specified bit with the carry bit. The result is stored in the carry bit. – If the Boolean value is a logical 1 then set the carry flag otherwise leave the carry flag in its current state.
		– A slash ("/") preceding the operand (i.e. ORL C, /<src-bit>) in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is unaffected.
Example	ORL C, ACC.4 ORL C, / ACC.4	– Let C = 1, A = 11H (0001 0001 B) then the instruction ORL C, ACC.4 will logically OR the contents of carry with accumulator Bit 4 leaving carry C = 1. – The instruction ORL C, / ACC.4 will logically OR the contents of carry with complement of accumulator Bit 4 leaving carry C = 1.

4.6.6 MOV <dest-bit>, <src-bit>

Mnemonic : MOV <dest-bit>, <src-bit>	Function : Move bit data
Algorithm : <dest-bit> = <src-bit>	Operation : MOV <dest-bit> ← <src-bit>

– This instruction will copy the source bit to the destination bit. One of the operands must be carry flag, the other operand may be any directly addressable bit. The different combinations are :

1. MOV bit, C

Mnemonic	MOV bit, C
Machine cycles	2
Bytes	2
Addr. Mode	Direct Addressing mode.

Function	Move bit data.
Clock Pulses	24
Algorithm	Bit = C
Flags	No flags are affected.

Operation	MOV (Bit) ← C	This instruction will copy the carry flag status into the Boolean variable whose address is specified in the instruction.
Example	MOV ACC.3,C	Let C = 1 A = 11H (0001 0001 B) then the instruction MOV ACC .3, C will copy the contents of carry to 3rd bit of accumulator leaving the accumulator with 19H (0001 1001 B).

2. MOV C, Bit

Mnemonic	MOV C, Bit
Machine cycles	1
Bytes	2
Addr. Mode	Direct Addressing mode.

Function	Move bit data.
Clock Pulses	12
Algorithm	C = (Bit)
Flags	Except carry, no other flags are affected.

Operation	MOV C ← (Bit)	This instruction will copy the data from Boolean variable whose address is specified in the instruction to the carry flag.
Example	MOV C, ACC.4	Let C = 1, A = 01H (0000 0001 B), then the instruction MOV C, ACC.4 will load the contents of 4th bit of accumulator to carry flag, leaving C = 0.

4.7 JUMP and CALL

Q. 4.7.1 Explain the following instructions.
(a) RETI (b) AJMP
(c) SJMP (d) DJNZ (Ref. Sec. 4.7) **(8 Marks)**

4.7.1 ACALL addr11

Mnemonic	ACALL addr11
Machine cycles	2
Bytes	2

Function	Absolute call.
Clock Pulses	24
Algorithm	(PC) = ((PC) (SP) = (SP)+ ((SP)) = (PC_{7-0}) (SP) = (SP) + 1 ((SP)) = (PC_{15-8}) (PC_{10-0}) = Page address.

Operation	ACALL (PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP)) ← (PC_{7-0}) (SP) ← (SP) + 1 ((SP)) ← (PC_{15-8}) (PC_{10-0}) ← Page address.	– This instruction unconditionally calls a subroutine at the indicated address. At the end of subroutine the program will resume operation at the opcode address following the call instruction. – ACALL can be located – anywhere in the program memory. It can be called/used a number of times in the same program. – The return address of the next instruction after the call instruction is in the program counter.
		– PC, 3 bits (A_{10}, A_9, A_8) from the first byte of instruction and the 8 bits (A_7 to A_0) second byte of instruction. The steps followed while executing ACALL instruction are : **Step 1:** The PC increments by twice in order to obtain the address of the next instruction after CALL.
		Step 2: It then pushes the 16 bit result onto the stack. Initially (SP) increments by 1. Then the low order byte of PC is pushed onto the stack (SP) again increments by 1. Now the high-order byte of PC is pushed onto the stack. **Step 3:** The destination address is computed by concatenating the high-order five bits of the incremented
Example	ACALL add	Let SP = 0AH. PC = 0239H. The label "add" is at program memory location 0435 H. After the execution of instruction, ACALL add at location 0237 H, SP will contain 0CH, the internal RAM location 0BH will contain 39H and 0CH will contain 02H and the PC will contain 0435H.

Fig. 4.7.1 shows how the destination address is computed.

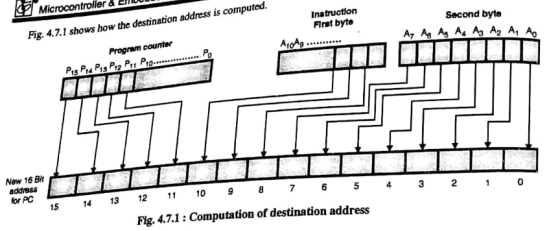


Fig. 4.7.1 : Computation of destination address

4.7.2 LCALL addr16

Mnemonic	LCALL addr16
Machine cycles	2
Bytes	3

Function	Long call
Clock Pulses	24
Algorithm	(PC) = (PC) + 3 (SP) = (SP) + 1 ((SP)) = (PC_{7-0}) SP) = (SP) ((SP)) = (PC_{15-8}) (PC) = $addr_{15-0}$

Operation	LCALL (PC) ← (PC) + 3 (SP) ← (SP) + 1 ((SP)) ← (PC_{7-0}) (SP) ← (SP) + 1 ((SP)) ← (PC_{15-8}) (PC) ← $addr_{15-0}$	– This instruction calls a subroutine that is located at the indicated address.
		The steps followed while executing LCALL instruction are: ***Step 1 :*** *The PC increments by thrice in order to generate the address of the next instruction.* ***Step 2 :*** *It then pushes the 16 bit PC onto the stack. Initially (SP) increments by 1. Then the low-order byte of PC is pushed onto the stack (SP) again increments by 1. Now the high-order byte of PC is pushed onto the stack.* ***Step 3 :*** The PC will be loaded with the second and third bytes of the instruction.

Example	LCALL add	Let the Stack Pointer SP = 07 H and PC = 023AH. The label "add" is at program memory location 0435H. After the execution of instruction. LCALL add at location 0237H, the stack pointer will contain 09H, internal RAM location 08H will contain 3DH and location 09H will contain 02H, and the PC will contain 0435H.

4.7.3 RET

Mnemonic	RET
Machine cycles	2
Bytes	1

Function	Return from Subroutine.
Clock Pulses	24
Algorithm	(PC_{15-8}) = ((SP)) (SP) = (SP) – 1 (PC_{7-0}) = ((SP)) (SP) = (SP) – 1

Operation	RET (PC_{15-8}) ← ((SP)) (SP) ← (SP) – 1 (PC_{7-0}) ← ((SP)) (SP) ← (SP) – 1.	– When this instruction is executed, it informs the microcontroller to return back to the program, from where subroutine was called. – This instruction Pops the high and low order bytes of the PC successively from the stack. The stack pointer is decremented by two. – The program execution will resume from the resulting address, generally the address of the next instruction that follows the LCALL or CALL instruction.
Example	RET	The Stack Pointer contains value 09H. Internal RAM locations 08H and 09H contain 53H and 27H. The instruction, RET will leave the stack pointer equal to the value 07H. Program execution will continue at location 2753H.

4.7.4 RETI

Q. 4.7.2 Explain the following instruction : RETI. (Ref. Sec. 4.7.4) **(2 Marks)**

Mnemonic	RETI
Machine cycles	2
Bytes	1

Function	Return from interrupt.
Clock Pulses	24
Algorithm	(PC_{15-8}) = ((SP)) (SP) = (SP) – 1 (PC_{7-0}) = ((SP)) (SP) = (SP) – 1

Operation	RETI (PC_{15-8}) ←((SP)) (SP) ← (SP – 1) (PC_{7-0}) ← ((SP)) (SP) ← (SP – 1)	– This instruction informs the microcontroller that it is returning from an ISR routine. – This instruction pops high and low order bytes of PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed.

		– The Stack Pointer is decremented by two. – The program execution resumes from the instruction that follows the point at which the interrupt request was detected. – If an interrupt of the same level is pending, then it is processed.
Example	*RETI*	Let SP = 09H, let us assume that an instruction RETI is detected at location 0322H. The internal RAM locations 08H contains 57H and location 09H contains 12H. The instruction RETI, will leave the SP = 07H and returns the program execution to location 1257H. i.e. PC = 1257H

Note : (i) IF the RETI instruction is used at the end of subroutine, then it may enable the interrupt logic erroneously.
(ii) No other registers are affected.
(iii) PSW is not automatically restored to its pre interrupt status.

– The only difference between the RET and RETI instructions is that whenever an interrupt logic is enabled RETI instruction is to be used.

4.7.5 AJMP addr11

Q. 4.7.3 Explain the following instruction : AJMP. (Ref. Sec. 4.7.5) **(2 Marks)**

Mnemonic	AJMP addr11	**Function**	Absolute jump
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	PC) = (PC) + 2 (PC_{10-0}) = Page address.

Operation	AJMP (PC) ← (PC) + 2 (PC_{10-0}) ← Page address	– This instruction transfers the program execution to the indicated address i.e. AJMP label. – The limitation is that the label (destination) must be within the same 2KB block of program memory. The destination address is calculated by concatenating the higher order five bits of the PC i.e.$(P_{15} - P_{11})$ after the PC is incremented by twice, bits 5 – 7 of the first byte of instruction i.e. (A_{10}, A_9, A_8) and the second byte of instruction. Fig. 4.7.1 shows the address computation.
Example	AJMP L7	The label "L7" is at program memory location 0537H. The instruction AJMP L7 is at location 0671H and will load the PC with 0537H.

4.7.6 LJMP addr16

Mnemonic	LJMP addr16	**Function**	Long Jump.
Machine cycles	2	**Clock Pulses**	24
Bytes	3	**Algorithm**	$(PC) = addr_{15-0}$

Operation	LJMP (PC) ← $addr_{15-0}$	– This instruction causes an unconditional branch to the indicated address, by loading the high order and low-order bytes of the PC respectively, with the second and third instruction bytes.
		– The destination may therefore be anywhere in the full 64K program memory address space, because it uses full 16 bits of two bytes i.e. 2nd and 3rd instruction bytes.
Example	LJMP L7	The label "L7" is assigned at memory location 5678H. The instruction LJMP L7 at location 0537H will load the program counter with 5678H.

4.7.7 SJMP rel

Mnemonic	SJMP rel	**Function**	Short jump
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	(PC) = (PC) + 2 (PC) = (PC) + rel

Operation	SJMP (PC) ← (PC) + 2 (PC) ← (PC) + rel	– The program control branches unconditionally to the address indicated. – The branch destination is computed by adding the signed i.e. relative displacement in the second instruction byte to the PC, after incrementing the PC by twice. – The only limitation is that the jump range is limited from – 128 to + 127 bytes. The destination range allowed is from 128 bytes preceding this instruction to 127 bytes following it.
Example	SJMP SUBA1	The label "SUBA1" is assigned to an instruction whose program memory location is 0478H. The instruction SJMP SUBA1 assembles into location 0401H. After the execution of this instruction the PC will contain 0478H.

4.7.8 JMP @A + DPTR

Mnemonic	JMP @ A + DPTR	**Function**	Jump indirect
Machine cycles	2	**Clock Pulses**	24
Bytes	1	**Algorithm**	(PC) = (A) + (DPTR)

Operation	JMP (PC ← A) + (DPTR)	– This instruction adds the eight bit unsigned contents of the accumulator with the contents of sixteen bit Data Pointer. The result of addition is loaded into the program counter. This is the address from where the microcontroller will begin execution. – Neither the accumulator nor the DPTR contents are altered.
Example	JMP @ A + DPTR	(i) Let (A) = 30H, (DPTR) = 1000 H, PC = 0500 H. The instruction JMP @ A + DPTR will the PC will contain 1030H, and execution will begin from location 1030H instead of location 0500H. (ii) Another application of this instruction is in CASE jumps. MOV DPTR, # JMP_TBL ; DPTR points to jumptable MOV A, INDEX – NO JMP @ A + DPTR JMP – TBL: AJMP CASE– 0 AJMP CASE – 1 AJMP CASE – 2 AJMP CASE – 3 AJMP CASE – 4 Let say Index – No = 1 then DPTR = 1000H, A = 01H ∴ Execution begins from address 1001H i.e. CASE – 1.

4.7.9 JZ rel

Mnemonic	JZ rel
Machine cycles	2
Bytes	2

Function	Jump if accumulator zero
Clock Pulses	24
Algorithm	PC = (PC) + 2 if A = 0 then (PC) = (PC) + rel

Operation	JZ PC ← (PC) + 2 if (A) = 0 then (PC) ← (PC) + rel	– This instruction will branch i.e. jump to indicated address if all the bits in the accumulator are zero otherwise it will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the second instruction byte to the PC, after incrementing the PC by two. – The accumulator remains unchanged.

Example	JZ LABEL5	Let A = 00H. The instruction, JZ LABEL 5 will cause the program execution to continue at the instruction identified by label LABEL 5

4.7.10 JNZ rel

Mnemonic	JNZ rel
Machine cycles	2
Bytes	2

Function	Jump if Accumulator Not Zero
Clock Pulses	24
Algorithm	(PC) = (PC) + 2 if A ≠ 0 then PC = PC + rel

Operation	JNZ (PC) ← (PC) + 2 if A ≠ 0 then PC = PC + rel.	– This instruction will branch i.e. jump to indicated address if the bits in the accumulator are nonzero otherwise it will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the second instruction byte to the PC, after incrementing the PC by two. The accumulator remains unchanged.
Example	JNZ L1	Let A = 05H. The instruction JNZ L1 will continue program execution at label L1

4.7.11 JC rel

Mnemonic	JC rel
Machine cycles	2
Bytes	2

Function	Jump if carry is set
Clock Pulses	24
Algorithm	(PC) = (PC) + 2 if C = 1 then PC = PC + rel.

Operation	JC (PC) ← (PC) + 2 if C = 1 then (PC) ← (PC) + rel	– This instruction will branch i.e. jump to the address indicated if the carry flag is set otherwise it will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the second instruction byte to the PC, after incrementing the PC twice.
Example	JC L1	The instruction JC L1 will cause the program execution to continue at the instruction identified by Label L1 if the carry flag is set.

4.7.12 JNC rel

Q. 4.7.4 Describe the instruction of 8051, JNC and MUL with one example. (Ref. Secs. 4.7.12 and 4.4.7) **(5 Marks)**

Mnemonic	JNC rel
Machine cycles	2
Bytes	2

Function	Jump if carry not set.
Clock Pulses	24
Algorithm	(PC) = (PC) + 2 if C ≠ 1 (i.e. C = 0) then (PC) = (PC) + rel.

Operation	JNC (PC) ← (PC + 2) if C ≠ 1 (i.e. C = 0) then (PC) ← (PC) + rel	– This instruction will branch i.e. jump to the address indicated if the carry flag is cleared otherwise continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the second byte of instruction to the PC, after the PC is incremented by twice.
Example	JNC	Let the carry flag is set C = 1 The instruction sequence JNC L1 CPL C JNC L2 will clear the carry flag and program execution will resume from the instruction identified by Label L2.

4.7.13 JB bit, rel

Mnemonic	JB bit, rel
Machine cycles	2
Bytes	3

Function	Jump if bit set.
Clock Pulses	24
Algorithm	(PC) = (PC) + 3 if (bit) = 1 then (PC) = PC + rel.

Operation	JB (PC) ← (PC) + 3 if (bit) = 1 then (PC) ← (PC) + rel	– This instruction will jump to the address indicated, if the indicated bit in the instruction is 1, otherwise program will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to the first byte of next instruction – The bit indicated is unchanged.
Example	JB P2.1, L1	Let port 2 = 73 H (0111 0011 B). The instruction JB P2.1, L1 will cause program execution to jump to the instruction at label L1.

4.7.14 JNB bit, rel.

Mnemonic	JNB bit, rel.
Machine cycles	2
Bytes	3

Function	Jump if bit not set.
Clock Pulses	24
Algorithm	(PC) = (PC) + 3 If (bit) = 0 then (PC) = (PC) + rel.

Operation	JNB (PC) ← (PC) + 3 if (bit) = 0 then (PC) ← (PC) + rel.	– This instruction will jump to the address indicated, if the indicated bit in the instruction is 0, otherwise program will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the next instruction. – The bit remains unchanged.
Example	JNB P1.3, L3	Let port 2 = 73 H (0111 0011 B). The instruction JNB P1.3, L3 will cause the program execution to resume at the instruction at label L3.

4.7.15 JBC bit, rel.

Mnemonic	JBC bit, rel.
Machine cycles	2
Bytes	3

Function	Jump if bit set and clear bit.
Clock Pulses	24
Algorithm	(PC) = (PC) + 3 If (bit) = 1 then bit = 0 and (PC) = (PC) + rel.

Operation	JBC (PC) ← (PC) + 3 if (bit = 1) then (bit = 0) and (PC) ← (PC) + rel.	– This instruction will jump to the address indicated if the indicated bit is 1, otherwise the program will continue with the next instruction. – The destination is computed by adding the signed relative displacement in the third instruction byte to the PC, once the PC in incremented to point the first byte of the next instruction. – The contents of the indicated bit are changed if the bit is set otherwise of bit are unaltered the contents.
Example	JBC P2.0, L1	Let port 2 = 75 H (0111 0101 B). The instruction JBC P2.0, L1 will cause program execution to resume form label L1 and port 2 = 74 H (0111 0100 B)

4.7.16 CJNE <dest-byte>, <src-byte>, rel.

Mnemonic :	CJNE <dest-byte>, <src-byte>, rel.

Algorithm:	(PC) = (PC) + 3
	if
	(dest-byte) ≠ (src-byte)
	then
	(PC) = (PC) + rel.
	if (dest-byte) < (src-byte)
	then
	C = 1
	else
	C = 0

Function :	Compare and Jump if not equal.
Operation :	– This instruction compares the magnitudes of the source bytes and the destination byte. If their values are unequal then it jumps to the address indicated otherwise program execution continues from the next instruction. – Neither source-byte nor the destination-byte is altered. The destination is calculated by adding the signed relative-displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the next instruction. – The carry flag is set if the dest-byte is less than the source byte, otherwise carry flag is cleared.

– This instruction supports different combinations. They are :

☛ **1. CJNE A, direct, rel.**

Mnemonic	CJNE A, direct, rel.	**Function**	Compare and Jump if not equal.
Machine cycles	2	**Clock Pulses**	24

Bytes	3	**Algorithm**	(PC) = (PC) + 3
			if
			(A) ≠ (direct)
			then
			(PC) = (PC) + rel.
			if (A) < (direct)
			then
			C = 1
			else
			C = 0.

Addr. Mode	Direct addressing mode.	**Flags**	Except carry, no other flags are affected.

Operation	CJNE A, direct, rel	– This instruction compares the magnitudes of accumulator and magnitude of memory location whose direct address is provided in the instruction and jumps to the indicated address if the magnitudes are unequal. – The carry flag is set if the contents of accumulator are smaller than the contents of memory location whose direct address is specified, otherwise it is cleared. – The destination address is calculated by adding the signed relative-displacement in the third instruction byte to the PC, once the PC is incremented to point to the first byte of the next instruction.
Example	CJNE A, 60 H, L5	Let A = 75H, contents of memory location 60H = 44 H then the instruction CJNE A, 60H, L5 Will clear the carry flag as contents of A > contents of memory location 60H and jump to instruction at label L5.

☛ **2. CJNE A, # data , rel.**

Mnemonic	CJNE A, # data, rel.	**Function**	Compare and jump if not equal.
Machine cycles	2	**Clock Pulses**	24

Bytes	3	**Algorithm**	(PC) = (PC) + 3
			If (A) ≠ (data)
			then
			(PC) = (PC) + rel.
			if (A) < (data)
			C = 1
			else
			C = 0

Addr. Mode	Immediate addressing mode.	**Flags**	Carry flag is affected.

Operation	(PC) ← (PC) + 3
	if (A) ≠ (data)
	(PC) ← (PC) + rel.
	if A < data
	C ← 1
	else
	C ← 0.
		– This instruction compares the magnitudes of accumulator with the magnitude of data specified in the instruction and jumps to the indicated address if the magnitudes are unequal.
		– The carry flag is set if the contents of accumulator are smaller than the data otherwise carry flag is cleared.
		– The destination address is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the point to the first byte of the next instruction.

Example	CJNE A, #60, L7	Let A = 44H, then the instruction CJNE A, # 60H, L7 will set the carry flag and jump to instruction at Label L7.

3. CJNE Rn, # data, rel.

Mnemonic	CJNE Rn, # data, rel	**Function**	Compare and Jump if not equal.
Machine cycles	2	**Clock Pulses**	24

Bytes	3	**Algorithm**	(PC) = (PC) + 3
			if (Rn) ≠ data
			then
			(PC) = (PC) + rel
			if (Rn) < data
			then
			C = 1
			else
			C = 0

Operation	CJNE Rn, # data, rel	– This instruction compares the magnitudes of register Rn of the selected register bank with the data specified in the instruction and jumps to the indicated address if the magnitudes are unequal. – The carry flag is set if the contents of register Rn are smaller than data, otherwise it is cleared. – The destination address is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the next instruction.
Example	CJNE R2, # 60 H, L8	Let R2 = 44H then the instruction CJNE R2, # 60H, L8 will set the carry flag and jump to instruction at Label L8.

4. CJNE @ Ri, # data, rel.

Mnemonic	CJNE @ Ri, # data, rel	**Function**	Compare and Jump if not equal.
Machine cycles	2	**Clock Pulses**	24

Bytes	3	**Algorithm**	(PC) = (PC) + 3
			if ((Ri)) ≠ data
			then
			(PC) = (PC) + rel
			if ((Ri)) < data
			then
			C = 1
			else
			C = 0.

Operation	CJNE @Ri, #data, rel.	– This instruction compares the magnitudes of contents pointed by register Ri of the selected register bank with the data specified in the instruction and jumps to the indicated address if the magnitudes are unequal.
		– The carry flag is set if the contents pointed by register Ri are smaller than the data, otherwise the carry flag is cleared. – The destination address is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the next instruction.
Example	CJNE @R1, #98, add	Let R1 = 57H and the contents of memory location 57H be 99H. then the instruction CJNE @R1, # 98H, add will set the carry flag and jump to instruction at label add.

4.7.17 DJNZ <byte>, <ret-addr>

Q. 4.7.5 Explain the following instruction : DJNZ. (Ref. Sec. 4.7.17) (2 Marks)

Mnemonic	DJNZ <byte>, <rel-addr>	**Function**	Decrement specified memory location/register by 1 and jump if not zero.
Machine cycles	2	**Clock Pulses**	24
Bytes	2 if register addressing is used 3 if direct addressing is used	**Algorithm**	(PC) = (PC) + 2 (byte) = (byte) – 1 if (byte) ≠ 0 then (PC) = (PC) + rel

Operation	DJNZ (PC) ← (PC) + 2 (byte) ← (byte) – 1 if byte ≠ 0 then (PC) ← (PC) + (rel)	– This instruction decrements the specified register or memory location by 1 and branches to the address indicated by the second operand if the resulting value is nonzero. – The destination is calculated by adding the signed relative displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the next instruction.
		– The original value of 00H will underflow to FFH if decremented. – It supports two addressing modes : register addressing mode and direct addressing mode.
Example	DJNZ R5, SUB	– Let R5 = 50 H the instruction DJNZ R5, SUB will cause jump to the instruction at label SUB and R5 = 4FH.

4.7.18 NOP

Mnemonic	NOP	**Function**	No operation.
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	(PC) = (PC) + 1

Operation	NOP (PC) ← (PC) + 1	– No operation is performed. Only the PC is affected. The contents of PC are incremented by 1. – It is mainly used for inserting delay in programs.

4.8 Instruction Comparison

4.8.1 Comparison of AJMP, SJMP and LJMP Instruction

➔ **(MU - Dec. 14, May 16, Dec. 16)**

Q. 4.8.1 Compare AJMP, SJMP, LJMP instructions of 8051. *(Ref. Sec. 4.8.1)*

Dec. 14, May 16, Dec. 16. 5 Marks

Sr. No.	Feature	AJMP	SJMP	LJMP
1.	Instruction size	2 bytes	2 bytes	3 bytes
2.	Address field size	11 bits	8 bits	16 bits
3.	Address type	Only lower 11 bits of address are given which are put into the lower 11 bits of the program counter, keeping the upper 5 bits unchanged	The 8-bit value given is in 2's complement form, hence it is just added to the current value of the program counter	The entire 16-bit address given is put into the program counter

Sr. No.	Feature	AJMP	SJMP	LJMP
4.	Branching range	Within the current 2K page, where the 64K memory is divided into 32 pages of 2K each	128 bytes behind or 127 bytes ahead from the current value of program counter	Anywhere in the entire 64K memory
5.	Example	AJMP 004CH	SJMP 05H	LJMP 08C8H

4.9 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : Addressing Modes**

Q. 1 What are the Addressing Modes of 8051 microcontroller ? Explain with example in each addressing mode. **(Refer section 4.2)** **(M-15, D-15, D-16, M-17, D-17, 10 Marks)**

Q. 2 Explain the different addressing modes of 8051. **(Refer section 4.2)** **(8 Marks)**

Q. 3 How will an instruction distinguish between internal and external memory reference ? **(Refer section 4.2)** **(5 Marks)**

Q. 4 Describe the instructions of 8051,MOVX @ DPTR, A with one example. **(Refer section 4.3.4)** **(M-15, 2 Marks)**

Q. 5 Describe the following instruction : XCHD A, @R1. **(Refer section 4.3.8)** **(2 Marks)**

☞ **Syllabus Topic: Arithmetic Operations**

Q. 6 Describe the instruction of 8051, JNC and MUL with one example. **(Refer Sections 4.7.12 and 4.4.7)** **(D-17, 5 Marks)**

Q. 7 Describe the instruction of 8051, SWAP A with one example. **(Refer section 4.5.10)** **(M-15, 2 Marks)**

☞ **Syllabus Topic: Logical Operations**

Q. 8 Explain the following instructions.

(a) RETI (b) AJMP

(c) SJMP (d) DJNZ **(Refer section 4.7)** **(8 Marks)**

Q. 9 Explain the following instruction : RETI. **(Refer section 4.7.4)** **(2 Marks)**

Q. 10 Explain the following instruction : AJMP. **(Refer section 4.7.5)** **(2 Marks)**

Q. 11 Explain the following instruction : DJNZ.. **(Refer section 4.7.17)** **(2 Marks)**

Q. 12 Compare AJMP, SJMP, LJMP instructions of 8051. **(Refer section 4.8.1)** **(D-14, M-16, D-16, 5 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 5

Programming with 8051

5.1 Introduction - Programming

- The 8051 is capable of performing a few operations at very high speed. 8051 instruction set indicates that :
  1. The ALU in combination with various registers can be controlled by binary operational codes to perform arithmetic operations,
  2. Microcontroller can transfer data inside the CPU,
  3. Microcontroller can perform logical and arithmetic manipulations,
  4. Microcontroller can make decisions based on manipulation result and
  5. Microcontroller can move data into and out of the microcontroller.
- The microcontroller cannot, by itself, perform simple calculations or data transfer. It must be told in clear terms, in complete detail, just what to do and in what order. The physical components, connection and logic circuits that take part in decoding and execution of operation codes constitute the computer hardware.
- The lists of specific instructions, selected from those allowed by the microcontroller manufacturer and organized to control operations constitute computer **software**.

In this chapter we will study the programming techniques and programming. The programming is divided into two parts. Simple programs and after that a concept of looping is introduced for programs.

5.2 Programming Steps

In programming technique five different points have been given, we follow those

Step 1 : Define the problem to be solved
Step 2 : Solution plan
Step 3 : Flowchart
Step 4 : Program
Step 5 : Check the result

☞ Step 1 : Define the problem to be solved

The problem for which you are preparing the program, the different terms must be clearly mentioned such as :

(i) What are the inputs to your program ?

In the case of 8051, the possible conditions that arise are

(a) The direct data is available

(b) The data is stored in register R1 – R7 of the selected register bank.

(c) The data is stored in memory location.

(d) The data is stored in memory location whose address is pointed by register R1 – R7 of the selected register bank.

The inputs must be clearly specified in the definition.

(ii) What is the operation you are expecting. The operation you want to perform on input data must be clear, ex : $y = a \times b$, $x = a + b + c$, etc.

(iii) Where you want the output ?

You have specified the inputs, operation, but where the result is to be stored or it is to be displayed must be specified clearly.

☞ Step 2 : Solution plan

To perform the operation specified in definition, the plan to solve the problem should be prepared it includes the following points

(i) How are you taking input data.

(ii) Which method you are using to solve the problem. What are the steps in this method.

(iii) How will you output the result.

By considering the above 3 points we prepare the plan of action.

☞ Step 3 : Flowchart

- The flowchart is a pictorial representation of various actions and computations that are taken to perform any task. A flowchart is similar to a block diagram, representing the structure of the program.
- Flowcharts break large, complex programs into smaller logical units and make it easier, to make changes and corrections.
- Generally, flowchart is used for 2 things
  (1) To assist and clarify the thinking process,
  (2) To communicate the program logic to others.
- With the program structure clearly defined in blocks, the flowchart can be converted directly into a set of instructions, using any programming language. Symbols commonly used in flowcharting are as shown in Fig. 5.2.1.

☞ Step 4 : Program

Go on putting the instructions instead of flowchart blocks.

☞ **Step 5 : Check the result**

Now find the codes for instructions, feed in 8085 system and execute the program. It will give you the result. If the result is correct the program you have prepared is correct. But if the result is not correct then you have to debug your program to find the error.

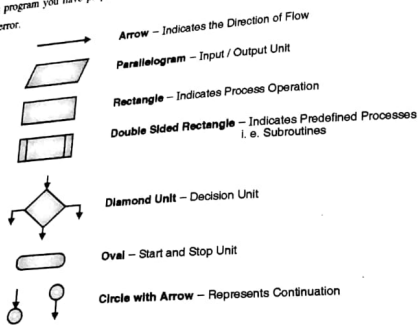


Fig. 5.2.1 Flowchart symbols

5.3 Assembly Language Programs

Program 5.3.1 : To store 8-bit data in registers.

>> Program Statement

Write program to load Data 05 H into accumulator and then transfer the same data to the B register.

>> Explanation

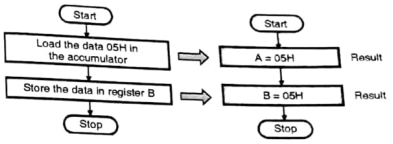
- We have immediate data 05 H. To load this data into the accumulator we will use the *MOV A, #05 H* instruction.
- To transfer the data to B register we will use the instruction *MOV B, A*.

>> Algorithm

Step I : Load the accumulator with immediate data *05 H*

Step II : Copy the contents of accumulator to the register B.

>> Flowchart : *Refer Flowchart 5.3.1.*

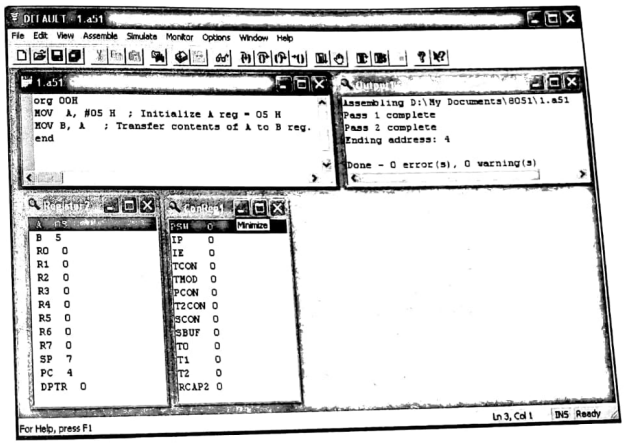


Flowchart 5.3.1

>> Program

Instruction	Comment	Operation
ORG 00H		
MOV A, #05 H	Initialize A reg = 05 H	A=05H (Result)
MOV B, A	Transfer contents of A to B reg.	B=05H (Result)
END	End Program	

>> Output



Program 5.3.2 : *To store 8-bit data in registers.*

>> Program Statement

Write program to load Data 05 H into accumulator and then transfer the same data to the register R0 and register R1 of the register bank 1.

>> Explanation

- We have immediate data 05 H. To load this data into the accumulator we will use the MOV A, #05H instruction.
- To transfer the data to register R0 of the selected register bank 1. We will use the instruction MOV R0, A
- To transfer the data to register R1 of the selected register bank 1. We will use the instruction MOV R1, A

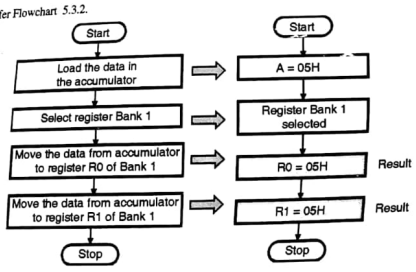
>> Algorithm

Step I : Load the accumulator with immediate data 05 H

Step II : Copy the contents of accumulator to the register R0 of the selected register bank 1.

Step III : Copy the contents of accumulator to the register R1 of the selected register bank 1.

>> Flowchart : Refer Flowchart 5.3.2.



Flowchart 5.3.2

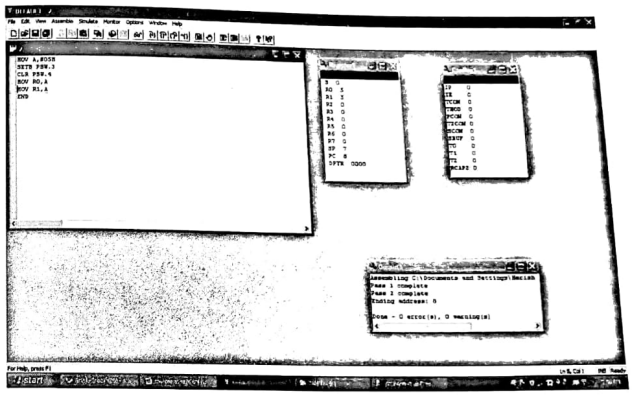
Method 1

Instruction	Comment	Operation
MOV A, #05H	Initialize A reg = 05 H	**A=05H (Result)**
SETB PSW.3		
CLR PSW.4	Clear 4th bit of the PSW to select bank 1.SETB	Select register bank 1
MOV R0, A	Transfer contents of A to register R0 of the selected register bank.	**R0=05 H (Result)**
MOV R1, A	Transfer contents of A to register R1 of the selected register bank.	**R1=05 H (Result)**
END	End Program	

Method 2

Instruction	Comment	Operation
MOV A, # 05H	Initialize A reg = 05 H	**A = 05H (Result)**
CLR PSW.4	Clear 4th bit of the PSW to select bank 1.	Select register bank 1
SETB PSW.3	Set 3rd bit of PSW	**R0 = 05 H , R1 = 05 H (Result)**
MOV R1, A		
MOV R0, A		
END	End Program	

>> Output



Program 5.3.3 : To find the 1's complement of the number

>> Program Statement

Write a program to find the 1's complement of the number. Assume that the number 20H is stored in register R3 of the register bank 0.

>> Explanation

- One's complement of number means to invert each bit of that number. So our task is to complement each bit of number.

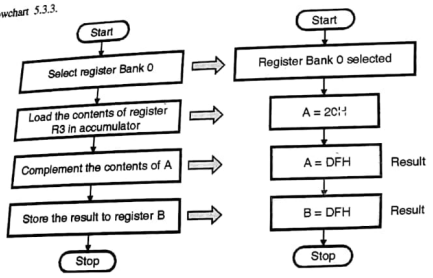
– We will first load the number whose 1's complement is to be found in the accumulator.
– Using the CPL instruction we will complement the accumulator. This is 1's complement of the number.
– Store the result in register B.

➢➢ Algorithm

Step I : Select register bank 0
Step II : Get the number in accumulator
Step III : Complement the number
Step IV : Store the result.

➢➢ Flowchart :

Refer Flowchart 5.3.3.

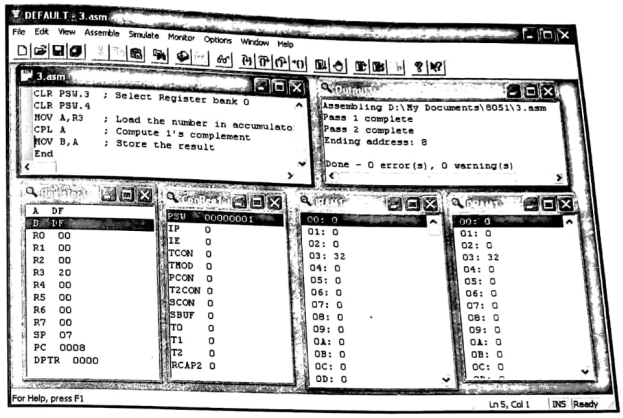


Flowchart 5.3.3

➢➢ Program

Instruction	Comment	Operation
CLR PSW.3 CLR PSW.4	Select Register bank 0	Register bank 0 selected
MOV A , R3	Load the number in accumulator	A= 20 H (0010 0000)
CPL A	Compute 1's complement	A= DF H (1101 1111) (Result)
MOV B,A	Store the result	B= DF H (Result)
END	End Program	

➢➢ Output



Program 5.3.4 : To find the 2's complement of the number

➢➢ Program Statement

Write a program to find the 2's complement of the number. Assume that the number is stored in register R3 of the register bank 0.

➢➢ Explanation

– Two's complement of number means add 1 to the 1's complement of that number.
– One's complement of number means to invert each bit of that number. So our task is to complement each bit of number.
– We will first load the number whose 1's complement is to be found in the accumulator.
– Using the CPL instruction we will complement the accumulator. This is 1's complement of the number. Now add 1 to this number so that 2's complement is computed.
– Store the result in register R3 of register bank 0.

➢➢ Algorithm

Step I : Select register bank 0.

Step II : Get the number in accumulator.

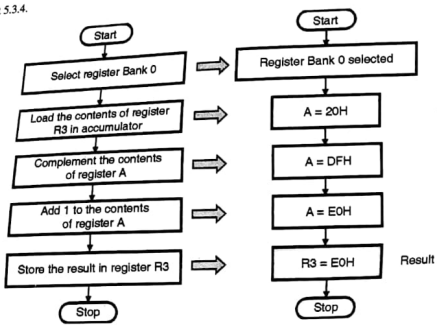
Step III : Complement the number.

Step IV : Add 1 to compute 2's complement.

Step V : Store the result.

➢➢ Flowchart :

Refer Flowchart 5.3.4.

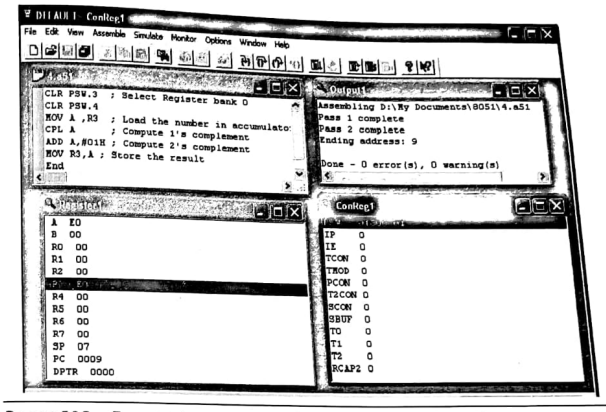


Flowchart 5.3.4

➢➢ Program

Instruction	Comment	Operation
CLR PSW.3	Select Register bank 0	Register bank 0 selected
CLR PSW.4		
MOV A , R3	Load the number in accumulator	A= 20 H (0010 0000)
CPL A	Compute 1's complement	A= DF H (1101 1111)
ADD A , #01H	Compute 2's complement	**A= E0 H (Result)**
MOV R3, A	Store the result	**R3= E0 H (Result)**
END	End Program	

➢➢ Output



Program 5.3.5 : To convert given 8 bit binary number to 3 digit Unpacked BCD number.

➢➢ Program Statement

Write a program to convert given binary number to 3 digit Unpacked BCD number. Load the 8 bit number in accumulator. Store the hundred's digit of the result in register R2 , ten's digit in the register R1 and the one's digit in the register R0 of the selected register bank.

➢➢ Explanation

- We are given an 8 bit binary number. Load this number in the accumulator.
- Load 64 H i.e. the hexadecimal equivalent of decimal 100 in the B register. Now divide the number by 100. The result of division will be stored in the A and B registers. The quotient is stored in the register A and the remainder in register B. Store the quotient in register R2.
- Now load the remainder obtained after division in accumulator. Now divide the number by 10. Load the quotient in register R1 and remainder in register R0.

➢➢ Algorithm

Step I : Start

Step II : Get the number in A register.

Step III : Load the register B with 100 decimal i.e. 64H.

Step IV : Divide the number by 100.

Step V : Store the quotient in register R2.

Step VI : Load the remainder in register A.

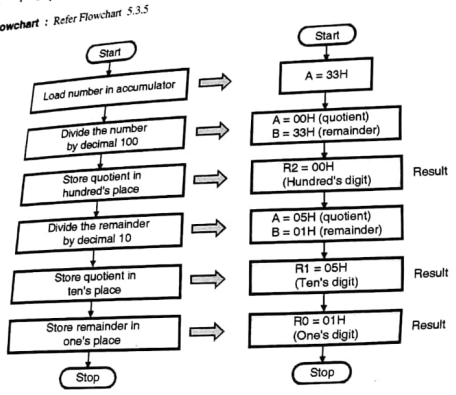
Step VII : Divide the number by 10.

Step VIII : Store the quotient in register R1.

Step IX : Store the remainder in register R0.

Step X : Stop

➢➢ **Flowchart** : Refer Flowchart 5.3.5



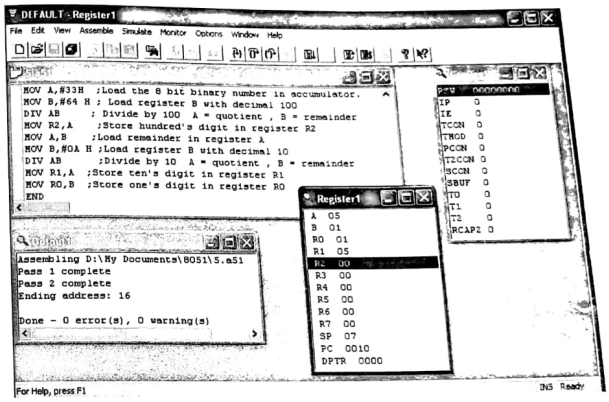
Flowchart 5.3.5

➢➢ **Program**

Instruction	Comment	Operation
MOV A,#33H	Load the 8 bit binary number in accumulator.	A = 33H
MOV B,#64H	Load register B with decimal 100	B=64H
DIV AB	Divide by 100, A = quotient, B = remainder	A= 00H , B=33H
MOV R2,A	Store hundred's digit in register R2	R2=00H (Result) Hundred's digit
MOV A, B	Load remainder in register A	A= 33H
MOV B,#0AH	Load register B with decimal 10	B= 0AH

Instruction	Comment	Operation
DIV AB	Divide by 10, A = quotient, B = remainder	A= 05H , B=01H
MOV R1,A	Store ten's digit in register R1	R1= 05 H (Result) Ten's digit
MOV R0 , B	Store one's digit in register R0	R0= 01 H (Result) One's digit
END	End Program	

➢➢ **Output**



Program 5.3.6 : Program to unpack the packed BCD number.

➢➢ **Program statement**

- A two digit BCD number is stored in the register A. Write a program in the ALP of 8051 to unpack this BCD number. Draw flowchart.
- Store the MSB digit in register R1 and LSB digit in register R0 of the register bank 3.

➢➢ **Explanation**

- A digit BCD number is available in register A. We have to unpack this BCD number i.e. we have to separate the BCD digits.
- e.g : If the number = 92 H then in unpack form the two digits will 02H and 09H. i.e. we have to mask the lower nibble, first and rotate four times to the right to get the MSB digit or use the SWAP instruction.

– *Then to get the LSB digit mask the upper nibble.*

– *Masking lower nibble means ANDing the number with 0F0 to get MSB.*

➤➤ **Algorithm**

Step I : *Load number into register A.*

Step II : *Mask the lower nibble.*

Step III : *Rotate 4 times left to make MSB digit = LSB.*

Step IV : *Store the digit in register R1.*

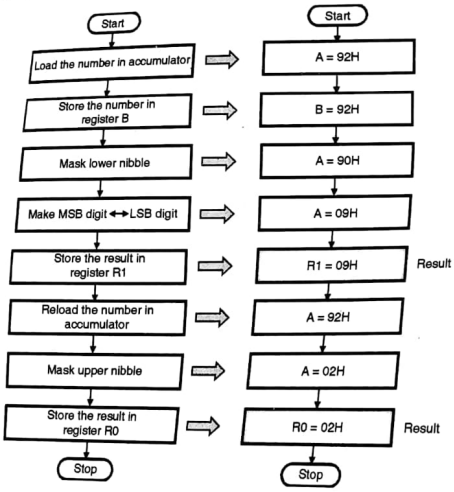
Step V : *Load number in register A.*

Step VI : *Mask upper nibble.*

Step VII : *Store the digit in register R0.*

Step VIII : *Stop.*

➤➤ ***Flowchart :*** *Refer Flowchart 5.3.6.*

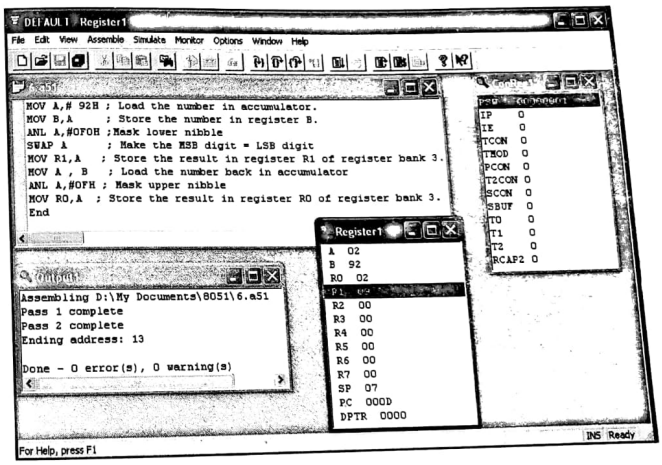


Flowchart 5.3.6

➤➤ **Program**

Instruction	Comment	Operation
MOV A ,#92H	Load the number in accumulator.	A = 92 H
MOV B , A	Store the number in register B.	B = 92 H
ANL A, #0F0H	Mask lower nibble	A = 90 H
SWAP A	Make the MSB digit ↔ LSB digit	A= 09 H
MOV R1 ,A	Store the result in register R1 of register bank 3	**R1= 09 H (MSB digit of result)**
MOV A , B	Load the number back in accumulator	A = 92 H
ANL A, #0FH	Mask upper nibble	A = 02 H
MOV R0, A	Store the result in register R0 of register bank 3.	**R0 = 02 H (LSB digit of result)**
END	End Program	

➤➤ **Output**



Program 5.3.7 : *To add Block of data assuming the sum to be 8-bit.*

➢➢ **Program statement**

Write a program to add ten bytes in internal RAM. Assume that the starting location of the Block is 40 H. Assume sum to be 8 bit. Store the result in register R0 of bank 1.

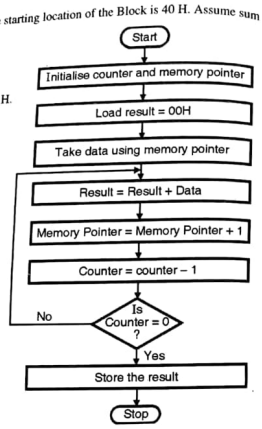
➢➢ **Explanation**

- *Consider that a block of 10 bytes is present at source location i.e. 40 H.*
- *We have to add these 10 bytes.*
- *We will initialize this as count in the R0 register.*
- *The register R1 will act as pointer to point the block.*
- *Using ADD instruction add the contents, byte by byte of the block.*
- *Increment R1 to point to next element.*
- *Decrement the counter and continue till all the contents are added.*
- *Result is stored in A. This result is stored in register R0 of bank 1.*

For example :

Block Data : *01 02 03 04 05 06 07 08 09 0A*

Result : *01 + 02 + 03 + 04 + 05 + 06 + 07 + 08 + 09 + 0A = 37 H*



Flowchart 5.3.7

➢➢ **Algorithm**

Step I : Initialise R1 as pointer with source address.

Step II : Initialise R0 register with count.

Step III : Add data, byte by byte.

Step IV : Increment pointer.

Step V : Decrement counter.

Step VI : Check for count , if not zero go to step III else go to step VII.

Step VII : Store the result of addition.

Step VIII : Stop.

➢➢ **Flowchart :** Refer Flowchart 5.3.7.

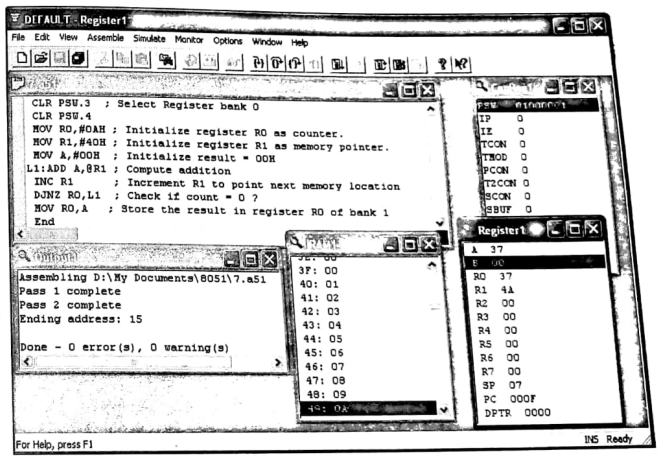
➢➢ **Program**

Label	Instruction	Comment	Operation
	CLR PSW.3	Select Register bank 0	Register Bank 0 selected
	CLR PSW.4		
	MOV R0 , #0AH	Initialize register R0 as counter.	R0 = 0A H

Label	Instruction	Comment	Operation
	MOV R1 , #40H	Initialize register R1 as memory pointer.	R1 = 40 H
	MOV A , #00H	Initialize result = 0	A = 00 H
L1:	ADD A , @R1	Compute addition	A = A + @R1
	INC R1	Increment R1 to point next memory location	R1 = R1 + 1
	DJNZ R0, L1	Check if count = 0 ?	Is R0 = 0 , if not continue executing loop L1
	MOV R0 , A	Store the result in register R0 of bank 1	R0 = 37 H (Result)
	END	End Program	

➢➢ **Result : 37 H**

➢➢ **Output**



Program 5.3.8 : To add Block of data assuming the sum to be 16 bit.

>> Program statement

Write a program to add ten bytes in internal RAM. Assume that the starting location of the Block is 40 H. Assume sum to be 16 bit. Store the result in registers R2 and R3 of register bank 0 with the LSB stored in register R2 and MSB in register R3.

>> Explanation

- Consider that a block of 10 bytes is present at source location i.e. 40 H.
- We have to add these 10 bytes.
- We will initialize this as count in the R0 register.
- The register R1 will act as pointer to point the block.
- Using ADD instruction add the contents, byte by byte of the block.
- Check for carry. If carry is there store the carry in register R3.
- Increment R1 to point to next element.
- Decrement the counter and continue till all the contents are added.
- Result is stored in A. This result is stored in register R2 of bank 0.

For example : Block Data : 51 02 03 04 05 06 07 08 09 0A

Result : 51 + 46 + 22 + 35 + 45 + 66 + 57 + 8A + D0 + 0A = 354 H

Hence R3 = 03 H and R2 = 54 H

>> Algorithm

Step I : Initialize R1 as pointer with source address.

Step II : Initialise R0 register with count.

Step III : Add data, byte by byte.

Step IV : Check for carry. If carry is present increment register R3.

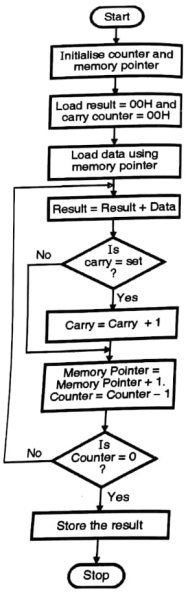
Step V : Increment pointer.

Step VI : Decrement counter.

Step VII : Check for count , if not zero goto step III else goto step VIII.

Step VIII : Store the result of addition.

Step IX : Stop.



Flowchart 5.3.8

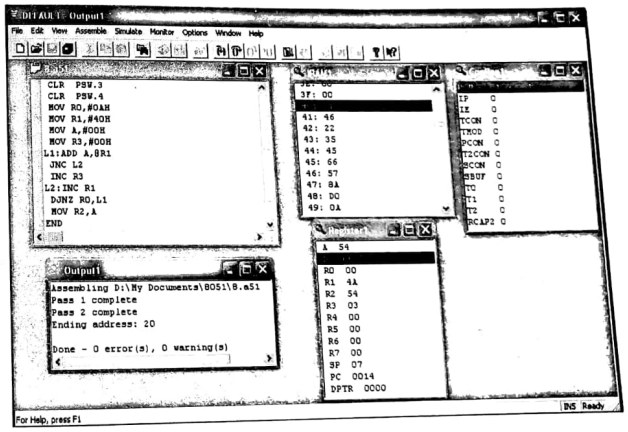
>> Flowchart : Refer Flowchart 5.3.8.

>> Program

Label	Instruction	Comment	Operation
	CLR PSW.3 CLR PSW.4	Select Register bank 0	Register bank 0 selected

Label	Instruction	Comment	Operation
	MOV R0 , #0AH	Initialize register R0 as counter.	R0 = 0AH
	MOV R1 , #40H	Initialize register R1 as memory pointer.	R1 = 40H
	MOV A , #00H	Clear accumulator	A = 00H
	MOV R3 , #00H	Clear register R3	R3 = 00H
L1:	ADDA , @R1	Compute addition	A = A+ @R1
	JNC L2	Check for carry	If CY = 1 go to L2
	INC R3	Increment register R3 if carry is present	If CY = 1 then R3 = R3+1
			R3 = 03H (Result)
L2 :	INC R1	Increment R1 to point next memory location	R1 = R1+ 1
	DJNZ R0, L1	Check if count = 0 ?	If R0 ≠ 0 then go to L1 else continue
	MOV R2, A	Store the result in register R2 of bank 0	**R2 = 54H (Result)**
	END	End Program	

>> Output



Program 5.3.9 : *Count the number of 1's and 0's in a number.*

➤➤ **Program statement**

– *Write a program in the ALP of 8051 to count the number of 1's and 0's in a byte stored in external memory location 3000 H. Draw flowchart.*

➤➤ **Explanation**

– *We will load the byte in A register from the external memory location.*

– *Initialize the counter 1 = 8; to count number of 0's.*

– *Initialize counter 2 = 0; to count the number of 1's.*

– *We will rotate the number in A along with carry by 1 bit to the right. If there is a carry we will increment counter 2 and decrement counter 1. This process will continue till all the bits are checked.*

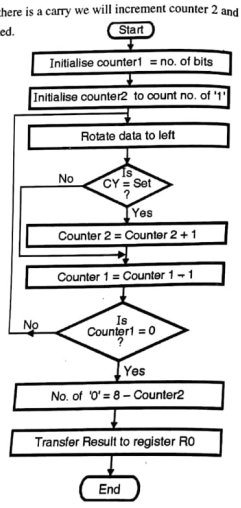
– *The counter 2 will indicate the number of 1's present in the word. The result of counter 2 is stored in R0 register.*

– *The number of zero's is found out by subtracting the number of one's from 8.*

e.g. : AX = 52 0101 0010

no. of 1's = 3

no. of 0's = 5



Flowchart 5.3.9

➤➤ **Algorithm**

Step I : Load the number in A register from the external memory.

Step II : Initialize counter 1 = 08.
Initialize counter 2 = 00

Step III : Rotate contents of register so that LSB will go in carry.

Step IV : Check if carry = 1. If not go to step VII.

Step V : Increment counter 2 i.e. increment R0.

Step VI : Decrement counter 1.

Step VII : Check if counter 1 = 0. If not go to step III.

Step VIII : Find the number of 0's = 8 – number of 1's in R0.

Step IX : Stop.

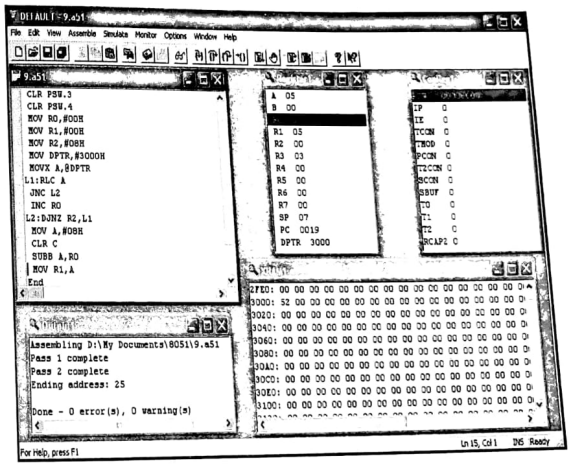
➤➤ **Flowchart :** Refer Flowchart 5.3.9.

➤➤ **Program**

Label	Instruction	Comment	Operation
	CLR PSW.3	Select Register bank 0	
	CLR PSW.4		
	MOV R0 , #00H	Initialize register R0 as 1's counter.	R0 = 00H

Label	Instruction	Comment	Operation
	MOV R1 , #00H	Initialize register R1 as 0's counter.	R1 = 00H
	MOV R2 , #08H	Initialize register R2 as counter for number of bits.	R2 = 08H
	MOV DPTR , #3000H	Load DPTR with the address of data	DPTR = 3000H
	MOVX A , @DPTR	Load accumulator with data	A = 52H
L1:	RLC A		
	JNC L2	Check for carry to find whether bit is one or zero.	Loop for counting number of 1's
	INC R0	Increment 1's counter	
L2 :	DJNZ R2 , L1	Decrement counter	R0 = 3 (Result)
	MOV A , #08H	Initialize A = 08H to find number of zero's	If R2 ≠ 0 then R2= R2 – 1
	CLR C	Clear carry	A= 08H
	SUBB A, R0	number of zero's = 8 – number of one's	CY = 1
	MOV R1 , A	Store the number of one's in register R1.	A = A – R0 = 08 – 03 = 05H
	END		R1 = 05H (Result)

➤➤ **Output**



Program 5.3.10 : Count the number of positive numbers in an array of numbers.

➢➢ Program statement

- Write a program in the ALP of 8051 to count the number of positive numbers. Assume that the length of array is stored in register R0 of bank 0. Assume that the numbers are stored from external memory location 3000 H.
- Draw flowchart. Store the result in register R1 of register bank 0.

➢➢ Explanation

- We will load the byte in A register from the external memory location.
- Assume that counter 1 = 8; i.e. count number of elements in the array.
- Initialize counter 2 = 0; to count the number of positive numbers.
- We will rotate the number in A along with carry by 1 bit to the right. If there is no carry, we will increment counter 2 i.e. number is positive.
- Decrement counter 1. This process will continue till all the bits are checked i.e. counter 1 becomes 0.
- The counter 2 will indicate the number of positive numbers present in the byte. The result of counter 2 is stored in R1 register.

➢➢ Algorithm

Step I : Initialize bank 0.

Step II : Counter 1 = R0 i.e. numbers in array.
Initialize counter 2 = 00 i.e. count for number of positive numbers.

Step III : Load the number in A register from the external memory.

Step IV : Rotate contents of register so that LSB will go in carry.

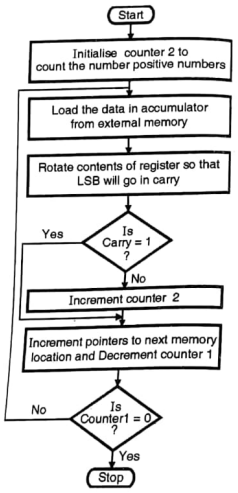
Step V : Check if carry = 1. If carry go to step VII.

Step VI : Increment counter 2 i.e. increment R1.

Step VII : Increment counter to next memory location and Decrement counter 1.

Step VIII : Check if counter 1 = 0. If not go to step III.

Step IX : Stop.



Flowchart 5.3.10

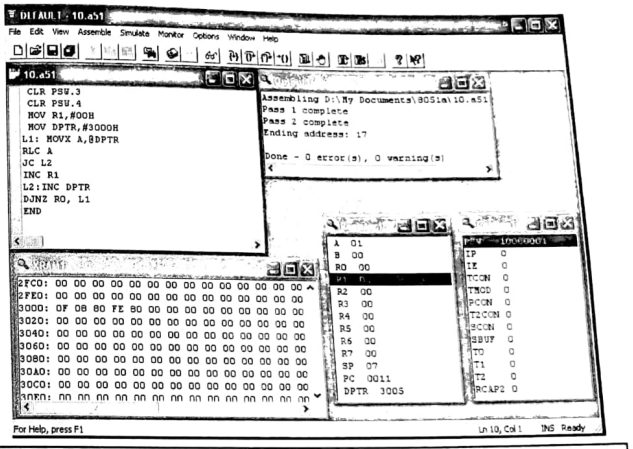
➢➢ Flowchart : Refer Flowchart 5.3.10.

➢➢ Program

Label	Instruction	Comment
	CLR PSW.3	Select Register bank 0
	CLR PSW.4	
	MOV R1, #00H	Initialize register R1 as positive number's counter.

Label	Instruction	Comment
	MOV DPTR, #3000H	Load DPTR with the address of data
L1:	MOVX A , @DPTR	Load accumulator with data
	RLC A	Rotate accumulator left through carry
	JC L2	Check for carry to find whether bit is one or zero
	INC R1	Increment positive number's counter
L2 :	INC DPTR	Increment pointer to point next memory location
	DJNZ R0, L1	Decrement counter
	END	End Program

➢➢ Output



Note : When executing this program initialize R0 to number of elements.

Program 5.3.11 : To search a byte in an array.

➢➢ Program statement

- Write a program in the ALP of 8051 to search a byte in an array. Assume that the length of array is stored in register R0 of bank 0.
- Assume that the numbers are stored from external memory location 3000 H. Draw flowchart. The byte to be searched is stored in the register R1 of register bank 0.

➢➢ Explanation

- DPTR is initialized as a memory pointer.
- The length of array is stored in register R0 , register R0 behaves as counter.
- We will load the byte in A register from the external memory location.
- Compare if the byte in A register matches with the byte to be searched. If the byte matches then store the address of the location where the byte is found.
- Otherwise increment the memory pointer and compare byte to be searched with byte from the next memory location.
- Decrement count in R0. This process will continue till match is found.

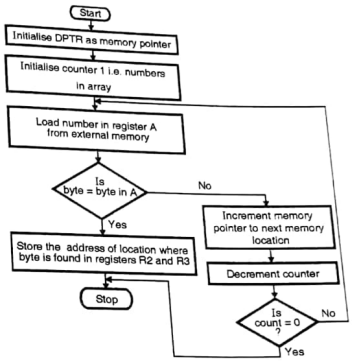
➢➢ Algorithm

Step I : DPTR is initialized as a memory pointer.
Step II : Initialize counter 1 = R0 i.e. numbers in array.
Step III : Load the number in A register from the external memory.
Step IV : Compare if the byte in A register matches with the byte to be searched. If match is not found go to step VII.
Step V : Store the address of location where byte is found in registers R2 and R3.
Step VI : Stop
Step VII : Increment memory pointer i.e. increment DPTR.
Step VIII : Decrement counter 1.
Step IX : Check if counter 1 = 0. If not go to step III.
Step X : Stop.

➢➢ Program

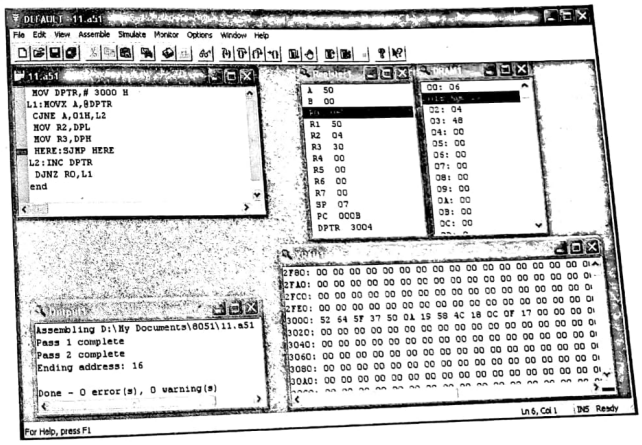
Label	Instruction	Comment	Operation
	MOV DPTR,#3000 H	Load DPTR with the address of data	DPTR = 3000 H
L1:	MOVX A, @DPTR	Load accumulator with data	Loop for searching byte . 01 H is the direct address of register R1 in DRAM where byte to be searched is stored.
	CJNE A, 01 H, L2	Search byte. 01 H is the direct address of register R1 where byte to be searched is stored. Byte in R1 = 50 H.	
	MOV R2, DPL	Store LSB of address of matched location in R2.	**R2 = 04 H (Result LSB of address)**
	MOV R3, DPH	Store MSB of address of matched location in R3.	**R3 = 30 H (Result MSB of address)**
HERE:	SJMP HERE	Stop	Stop
L2 :	INC DPTR	Increment pointer to point next memory location.	DPTR = DPTR + 1
	DJNZ R0, L1	Decrement counter	R0 = R0 – 1 , if R0 ≠ 0 go to L1
	End	Stop	

➢➢ Flowchart : Refer Flowchart 5.3.11.



Flowchart 5.3.11

➢➢ Output



Note : When executing this program initialize R0 to number of elements.

Program 5.3.12 : Program to find maximum/largest number in the array.

➢➢ Program statement

- Given an array of N numbers. Write a program in ALP of 8051 to find the maximum number amongst the N numbers.
- Assume the length of array is stored register R0 of the register bank 0. Assume that the array begins from memory location 3000 H. Store the Maximum or the largest number in the register R1 of register bank 0.

➢➢ Explanation

- We have an array of 10 numbers. So we initialize the counter with 10. Also we initialize a pointer to point these numbers.
- Compare first number with initial maximum number i.e. zero. If number > maximum number, save number otherwise increment pointer to compare next number.
- Decrement counter, compare till all the numbers are compared.
- Store the maximum number in register R1.

➢➢ Algorithm

Step I : Initialize memory pointer.

Step II : Load data pointed by memory pointer i.e. OP1 and store it in register B.

Step III : Decrement counter.

Step IV : Check if count = 0. If yes goto step X.

Step V : Increment memory pointer to next memory location.

Step VI : Load data pointed by pointer i.e. OP2.

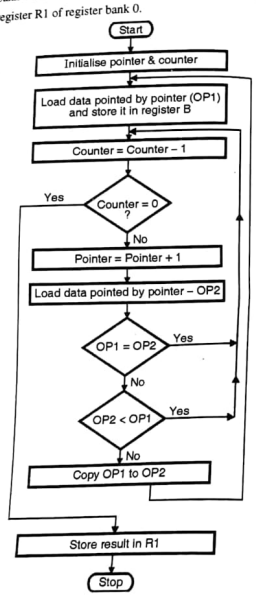
Step VII : Compare the two numbers i.e. OP1 and OP2. If OP1 = OP2 ; goto step III.

Step VIII : If OP2 < OP1 ; goto step III

Step IX : If OP2 > OP1 ; goto step II

Step X : Store the result in register R1.

Step XI : Stop



Flowchart 5.3.12

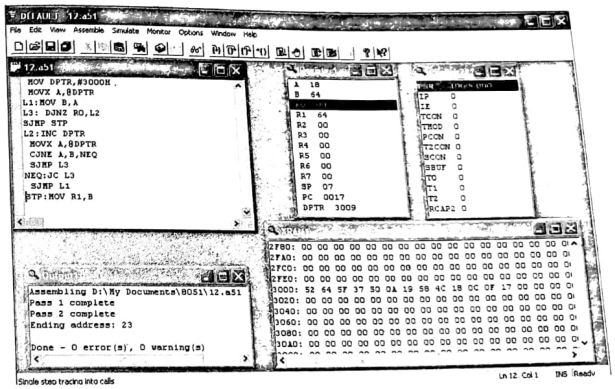
➢➢ Flowchart : Refer Flowchart 5.3.12.

➢➢ Program

Label	Instruction	Comment	Operation
	MOV DPTR, #3000H	Load DPTR with the address of data	DPTR = 3000H
	MOVX A, @DPTR	Load accumulator with data	A = @DPTR

Label	Instruction	Comment	Operation
L1 :	MOV B, A	Store the number in register B	B = A
L3 :	DJNZ R0, L2	Decrement counter	If R0 ≠ 0 then R0 = R0 – 1 and go to L2
	SJMP STP	Stop if counter = 0	Stop
L2:	INC DPTR	Increment data pointer to point next memory location	DPTR = DPTR + 1
	MOVX A, @DPTR	Load accumulator with data	A = @DPTR
	CJNE A, B, NEQ	Find the maximum number	**B = 64H (Result)**
	SJMP L3	If A = B, continue	
NEQ:	JC L3	If A < B, continue	
	SJMP L1	If A > B, exchange the contents.	
STP :	MOV R1 , B	Store the largest number in register R1	**R1 = 64H (Result)**

➢➢ Output



Note : When executing this program initialize R0 to number of elements.

Program 5.3.13 : Write a assembly language program for 8051 to find largest number from a data block of ten bytes that pres... 20 H to 29H. Store the result in memory location ... May 15,10 Marks

➢➢ **Program**

Label	Instruction	Comment	Operation
	MOV R2,#09	Load R2 with the number of comparisons i.e. 9	R2=09H
	MOV R0, #20H	Load R0 with the address of data	R0 = 20H
	MOVX A, @R0	Load accumulator with data	A = @R0
L1 :	MOV B, A	Store the number in register B	B = A
L3 :	DJNZ R2, L2	Decrement counter	R2 = R2 – 1 If R2 ≠ 0 then and go to L2
	SJMP STP	Stop if counter = 0	Stop
L2:	INC R0	Increment data pointer to point next memory location	R0 = R0 + 1
	MOVX A, @R0	Load accumulator with data	A = @R0
	CJNE A, B, NEQ	Find the maximum number	
	SJMP L3	If A = B, continue	
NEQ:	JC L3	If A < B, continue	
	SJMP L1	If A > B, exchange the contents.	
STP :	MOV 2AH , B	Store the largest number in memory location 2AH	

Program 5.3.14 : Program to find the least /smallest number in the array.

➢➢ ***Program statement***

- Given an array of N numbers. Write a program in ALP of 8051 to find the minimum number amongst the N numbers.
- Assume the length of array is stored register R0 of the register bank 0. Assume that the array begins from memory location 3000 H.
- Store the Smallest or the Least number in the register R1 of register bank 0.

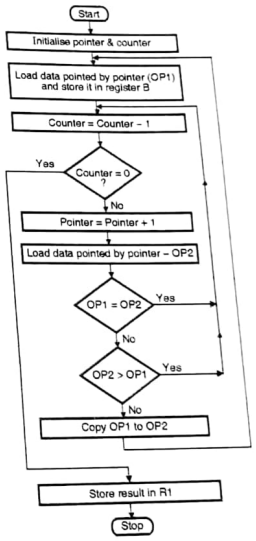
➢➢ **Explanation**

- We have an array of 10 numbers. So we initialize the counter with 10. Also we initialize a pointer to point these numbers.
- Compare first number with initial minimum number i.e. zero. If number < minimum number, save number otherwise increment pointer to compare next number.
- Decrement counter, compare till all the numbers are compared.
- Store the smallest number in register R1.

➢➢ **Algorithm**

Step I : Initialize memory pointer.
Step II : Load data pointed by memory pointer i.e. OP1 and store it in register B.
Step III : Decrement counter.
Step IV : Check if count = 0. If yes goto step X.
Step V : Increment memory pointer to next memory location.
Step VI : Load data pointed by pointer i.e. OP2.
Step VII : Compare the two number i.e. OP1 and OP2. If OP1 = OP2 , goto step III.
Step VIII : If OP2 > OP1 ; goto step III
Step IX : If OP2 < OP1 ; goto step II
Step X : Store the result in register R1.
Step XI : Stop

➢➢ **Flowchart :** Refer Flowchart 5.3.14.

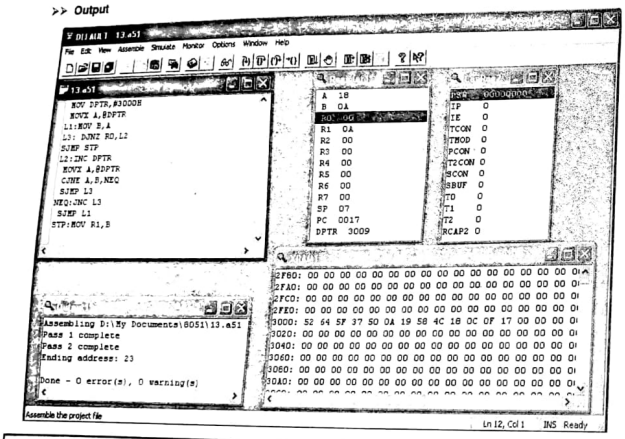


Flowchart 5.3.14

>> **Program**

Label	Instruction	Comment	Operation
	MOV DPTR, #3000H	Load DPTR with the address of data	DPTR = 3000H
	MOVX A, @DPTR	Load accumulator with data	A = @ DPTR
	MOV B, A	Store the number in register B	B = A
L1 :	DJNZ R0, L2	Decrement counter	If R0 ≠ 0 then R0 = R0 – 1 and go to L2
L3 :	SJMP STP	Stop if counter = 0	stop
L2 :	INC DPTR	Increment data pointer to point next memory location	DPTR = DPTR + 1
	MOVX A, @DPTR	Load accumulator with data	A = @DPTR
	CJNE A, B, NEQ	Find the minimum number	B = 0AH (Result)
	SJMP L3	If A = B, continue	
NEQ :	JNC L3	If A > B, continue	
	SJMP L1	If A < B, exchange the contents.	
STP :	MOV R1, B	Store the smallest number in register R1.	R1 = 0AH (Result)

>> **Output**



Note : When executing this program initialize R0 to number of elements.

Program 5.3.15 : Subtract two 8 bit numbers.

>> **Program statement**

– Assuming two numbers are available in A and B registers, write a program in assembly language of 8051 to subtract two 8 bit numbers.

>> **Explanation**

– Consider that a byte of data is present in the A register and second byte of data is present in the B register.
– We have to subtract the byte in B from the byte in A. Using sub instruction subtract the contents of two registers.
– Result will be stored in the A register.

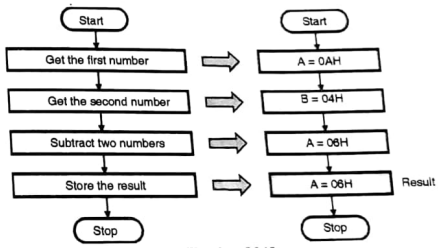
For example :

$$\begin{array}{lll} A = 0A\ H & 0A\ H & (10)_{10} \\ B = 04\ H & -\ 04\ H & (4)_{10} \\ & 06H & \end{array}$$

>> **Algorithm**

Step I : Get the first number in A register.
Step II : Get the second number in B register.
Step III : Subtract the two numbers.
Step IV : Stop

>> **Flowchart :** Refer Flowchart 5.3.15.

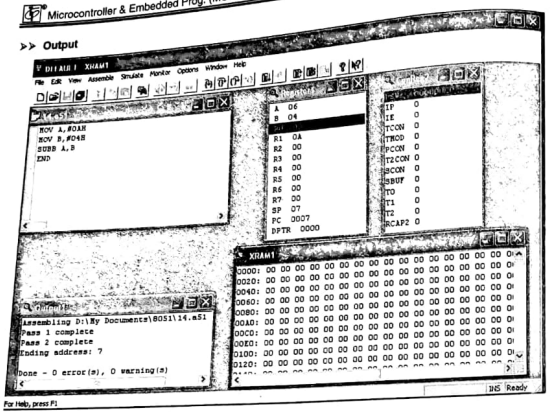


Flowchart 5.3.15

>> **Program**

Instruction	Comment	Operation
MOV A, #0AH	Load the first number in register A.	A= 0AH
MOV B, #04H	Load the second number in register B.	B= 04H
SUBB A, B	Compute subtraction. Result is stored in accumulator	**A= 06H (Result)**
END	End program	End

➤➤ **Output**



Program 5.3.16 : ADD two 8 bit numbers.

➤➤ **Program statement**

- Assuming two numbers are available in A and B registers, write a program in assembly language of 8051 to add two 8 bit numbers.

➤➤ **Explanation**

- Consider that a byte of data is present in the A register and second byte of data is present in the B register.
- We have to add the byte in B from the byte in A. Using ADD instruction add the contents of two registers.
- Result will be stored in the A register.

For example :

$$\begin{array}{lll} A = 06\,H & 06\,H & (06)_{10} \\ B = 04\,H & + \quad 04\,H & (04)_{10} \\ & 0AH & (10)_{10} \end{array}$$

➤➤ **Algorithm**

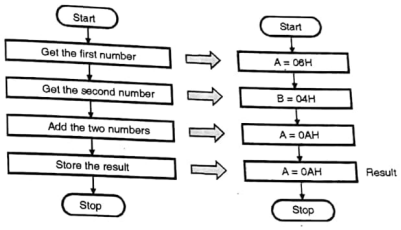
Step I : Get the first number in A register.

Step II : Get the second number in B register.

Step III : Add the two numbers.

Step IV : Stop.

➤➤ **Flowchart :** Refer Flowchart 5.3.16.

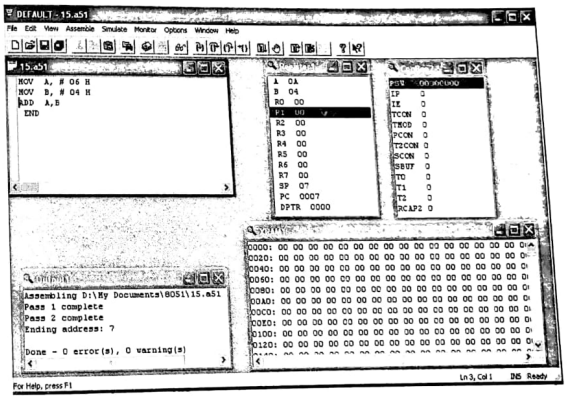


Flowchart 5.3.16

➤➤ **Program**

Instruction	Comment	Operation
MOV A, #06H	Load the first number in register A.	A = 06H
MOV B, #04H	Load the second number in register B.	B = 04H
ADD A, B	Compute addition. Result is stored in accumulator	A = 0AH (Result)
END	End program	End

➤➤ **Output**



Program 5.3.17 : Program to mask lower nibble.

➢➢ Program statement

Write an assembly language program to mask the lower nibble of an 8 bit number. Assume the 8 bit number is in the A register. Store the result in register B

➢➢ Explanation

- Let the 8 bit number be in the A register.
- We have to mask the lower nibble, i.e. we have to separate the lower nibble.
- In the result only MSB number should remain.

For example

e.g. : A = 5B H. 0101 1011

Logically AND with F0 H 1111 0000

0101 0000 = 50 H.

Result = 50 H.

- Store the result.

➢➢ Algorithm

Step I : Load the number in A.

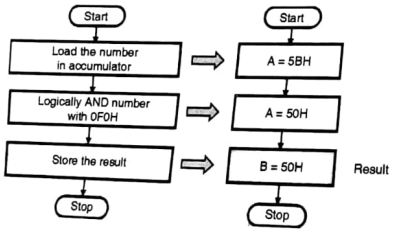
Step II : Mask lower nibble i.e. ANL 0F0 H.

Step III : Store result in register B.

Step IV : Stop.

➢➢ Flowchart :

Refer Flowchart 5.3.17.

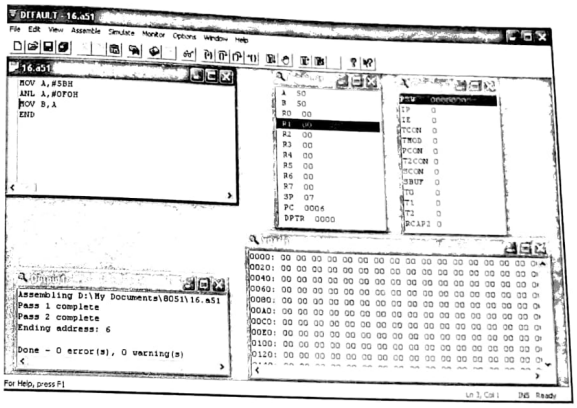


Flowchart 5.3.17

➢➢ Program

Instruction	Comment	Operation
MOV A, #5BH	Load number in register A	A = 5BH
ANL A, #0F0H	Logically AND the number	A= 50H
MOV B, A	Store the result.	B= 50H (Result)
END	End Program	End

➢➢ Output



Program 5.3.18 : Program to mask upper nibble.

➢➢ Program statement

Write an assembly language program to mask the upper nibble of an 8 bit number. Assume the 8 bit number is in the A register. Store the result in register B.

➢➢ Explanation

- Let the 8 bit number be in the A register.
- We have to mask the upper nibble i.e. we have to separate the upper nibble.
- In the result only LSB number should be present.

e.g. : A = 67H. 0110 0111

Logically AND with 0FH 0000 1111

0000 0111 = 07H

Result = 07 H, 60 is masked.

Store the result.

➢➢ Algorithm

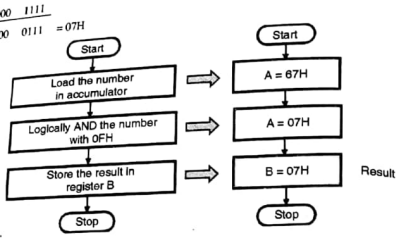
Step I : Load the number in A.

Step II : Mask the upper nibble.

Step III : Store the result in register B.

Step IV : Stop.

➢➢ Flowchart : Refer Flowchart 5.3.18.

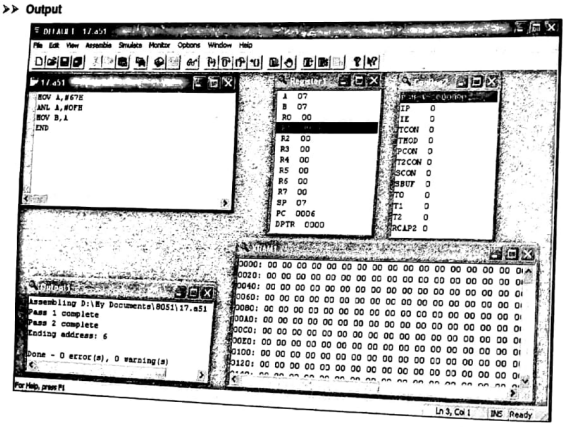


Flowchart 5.3.18

➢➢ Program

Instruction	Comment	Operation
MOV A, #67H	Load the number in accumulator	A = 67H
ANL A, #0FH	Logically AND the number	A = 07H
MOV B, A	Store the result.	B = 07H (Result)
END	End Program	

➢➢ Output



Program 5.3.19 : Program to sort the numbers in ascending order. **MU - May 16,10 Marks**

➢➢ Program statement

- Write a program in assembly language of 8051 to sort the given N numbers from a block in ascending order. Assume that the memory block begins at 3000 H.

➢➢ Explanation

- Consider that a block of N words is present. Now we have to arrange these N words in ascending order, Let N = 20 for example. We will use register R0 as pointer to point the block of N words.
- Initially in the first iteration we compare first number with the second number. If first number < second number, don't interchange the contents, otherwise if first number > second number swap the contents.
- In the next iteration we go on comparing the first number with third number. If first number < third number, don't interchange the contents. If first number > third number then swapping will be done.
- Since the first two numbers are in ascending order the third number will go to first place, first number in second place and second number will come in third place in the second iteration only if first number > third number.
- In the next iteration first number is compared with fourth number. So on comparisons are done till all N numbers are arranged in ascending order. This method requires approximately n comparisons.

➢➢ Algorithm

Step I : Initialize the number of elements counter 1.

Step II : Initialize the number of comparisons counter 2.

Step III : Compare the elements. If first element < second element goto step V.

Step IV : Swap the elements.

Step V : Decrement the comparison counter.

Step VI : Is counter 2 = 0 ? if yes goto step VIII else goto step II.

Step VII : Insert the number in proper position

Step VIII : Increment the number of elements counter.

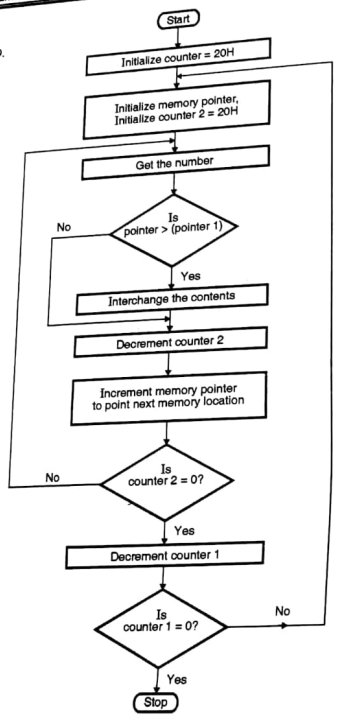
Step IX : Is counter 1 = 0 ? If yes, goto step XI else goto step II

Step X : Store the result.

Step XI : Stop.

➤➤ Flowchart :

Refer Flowchart 5.3.19.



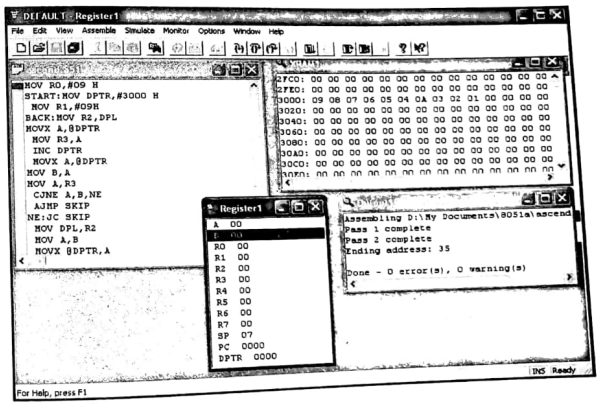
Flowchart 5.3.19

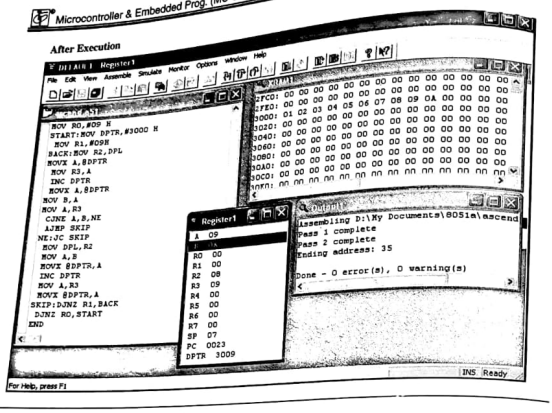
➤➤ Program

Label	Instruction	Comment
	MOV R0, #20H	Initialize counter 1
START:	MOV DPTR, #3000H	Initialize memory pointer
	MOV R1, #20H	Initialize counter 2

BACK:	MOV R2, DPL	Save the address of lower byte
	MOVX A, @DPTR	Get the number in accumulator
	MOV R3, A	Store the number in R3
	INC DPTR	Increment memory pointer
	MOVX A, @DPTR	Get the next number in accumulator
	MOV B,A	
	MOV A,R3	
	CJNE A, B, NE	Compare number with next number
	AJMP SKIP	If less, don't interchange
NE:	JC SKIP	If equal, don't interchange
	MOV DPL, R2	
	MOV A,B	
	MOVX @DPTR, A	Otherwise swap the contents
	INC DPTR	Increment pointer to next memory location
	MOV A, R3	
	MOVX @DPTR, A	
SKIP :	DJNZ R1, BACK	If R1≠ 0 then goto BACK
	DJNZ R0, START	If R0≠ 0 then goto START
	END	

➤➤ Output Before Execution





Program 5.3.20 : Multiply two 8 bit numbers.

➢➢ Program statement

Multiply two 8 bit numbers stored in external memory locations 3000H and 3001H. Store the result in memory locations 3020 H and 3021H.

➢➢ Explanation

- Consider that a byte is present at the memory location 3000H and second byte is present at memory location 3001H.
- We have to multiply the bytes present at the above two memory locations.
- We will multiply the numbers using MUL instruction. As MUL instruction operates only on the A and B register we will load the two numbers from memory locations 3000H and 3001H to the A and the B registers. The result of multiplication is stored in the A and B registers. LSB digit is stored in the accumulator while the MSB digit is stored in the B register.
- Result is stored at memory locations 3020 H and 3021 H with LSB stored at memory location 3020 H and MSB stored at memory location 3021 H.

For example : 3000H = 09 H, 3001H = 02 H

For example :

$$\begin{array}{rl} A = 09\ H & 09\ H \\ B = 02\ H & \times \quad 02\ H \\ \hline & 0012\ H \end{array}$$

Result = 0012 H

3020 H = 12 H

3021 H = 00 H

➢➢ Algorithm

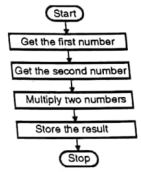
Step I : Get the first number.

Step II : Get the second number.

Step III : Multiply the two numbers.

Step IV : Store the result.

Step V : Stop.



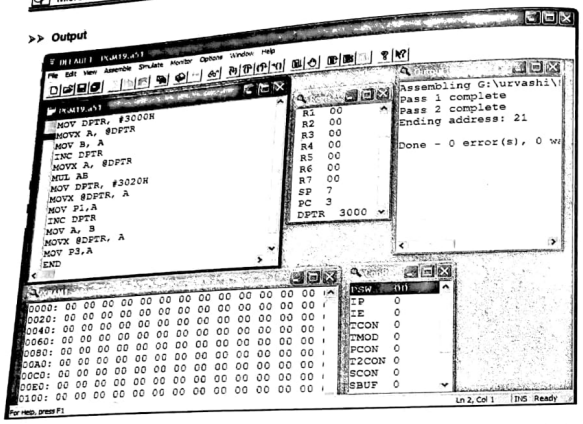
Flowchart 5.3.20

➢➢ Flowchart : Refer Flowchart 5.3.20.

➢➢ Program

Instruction	Comment	Operation
MOV DPTR, #3000H	Initialize DPTR as memory pointer	DPTR = 3000H
MOVX A, @DPTR	Get the first number in register A	A = 09H
MOV B, A	B = first number	B = 09H
INC DPTR	Increment memory pointer to point the second number	DPTR = 3001H
MOVX A, @DPTR	Get the second number in register A	A = 02H
MUL AB	Compute multiplication. A= LSB digit B = MSB digit after multiplication.	**A = 12H , B = 00H (Result)**
MOV DPTR, #3020H	Initialize DPTR as memory pointer for storing result	DPTR = 3020H
MOVX @DPTR, A	Store the LSB digit obtained at location 3020H.	**3020H = 12H** **(LSB digit of result)**
MOV P1, A	Send result on Port 1.	
INC DPTR	Increment memory pointer.	DPTR = 3021H
MOV A, B	Store the MSB digit obtained in register A.	A = 00H
MOVX @DPTR, A	Store the MSB digit obtained at location 3021H.	**3021H = 00H** **(MSB digit of result)**
MOV P3, A	Send result on Port 3.	
END		

>> **Output**



Program 5.3.21 : Write assembly language program for 8051 to multiply two 8 bit numbers stored in external memory locations 4000 H and 4001 H. Send the result on PORT 1 and PORT 3.

MU - Dec. 14, 10 Marks

>> **Program**

Instruction	Comment	Operation
MOV DPTR, #4000H	Initialize DPTR as memory pointer	DPTR = 4000H
MOVX A, @DPTR	Get the first number in register A	
MOV B, A	B = first number	
INC DPTR	Increment memory pointer to point the second number	DPTR = 4001H
MOVX A, @DPTR	Get the second number in register A	
MUL AB	Compute multiplication. A= LSB digit B = MSB digit after multiplication.	
MOV P1, A	Send result on Port 1.	
MOV A, B	Store the MSB digit obtained in register A.	
MOV P3, A	Send result on Port 3.	
END		

Program 5.3.22 : Divide two 8 bit numbers.

>> **Program statement**

Divide two 8 bit numbers stored in external memory locations 3000 H and 3001 H. Store the result in memory locations 3020 H and 3021 H. The quotient is to be stored at memory location 3020 H and remainder at memory location 3021 H.

>> **Explanation**

- Consider that a byte is present at the memory location 3000 H and second byte is present at memory location 3001 H.
- We have to divide the bytes present at the above two memory locations.
- We will divide the numbers using DIV instruction. As DIV instruction operates only on the A and B register we will load the two numbers from memory locations 3000 H and 3001 H to the A and the B registers. The quotient obtained after the division is stored in the A register and the remainder is stored in the B register.
- Result is stored at memory locations 3020H and 3021 H with quotient stored at memory location 3020 H and remainder stored at memory location 3021H

For example : 3000 H = 0F H, 3001 H = 08H

A = 0F H (i.e. 15) 08) 000F (01

↓

B = 08 H − 08 Quotient

Remainder 07

3020 H = 01 H

3021 H = 07 H

>> **Algorithm**

Step I : Get the first number.

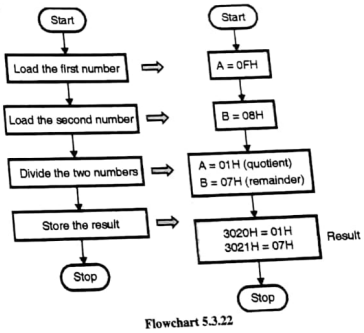
Step II : Get the second number.

Step III : Divide the two numbers.

Step IV : Store the result.

Step V : Stop.

>> **Flowchart :** Refer Flowchart 5.3.22.



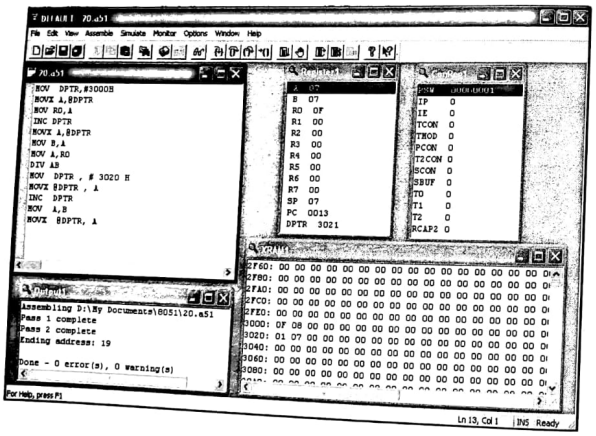
Flowchart 5.3.22

>> **Program**

Instruction	Comment	Operation
MOV DPTR,#3000H	Initialize DPTR as memory pointer	DPTR = 3000H
MOVX A, @DPTR	Get the first number in register A	A= 0FH

Instruction	Comment	Operation
MOV R0, A	R0 =first number	R0 = 0FH
INC DPTR	Increment memory pointer to point the second number	DPTR = 3001H
MOVX A, @DPTR	Get the second number in register A	A = 08H
MOV B , A	Load the divisor in register B	B= 08H
MOV A, R0	Load the dividend in register A	A = 0FH
DIV AB	Compute division. A= quotient , B = remainder.	A = 01H , B=07H (Result)
MOV DPTR , #3020H	Initialize DPTR as memory pointer for storing result	DPTR = 3020 H
MOVX @DPTR , A	Store the quotient obtained at location 3020H.	3020H = 01H (Result)
INC DPTR	Increment memory pointer.	DPTR = 3021H
MOV A, B	Store the remainder obtained in register A.	A = 07H
MOVX @DPTR, A	Store the remainder obtained at location 3021H.	3021H = 07H (Result)
End	End program	

➤➤ **Output**



Program 5.3.23 : Multiply two 8 bit numbers using successive addition method.

➤➤ **Program statement**

– Assuming that MUL instruction is not available in the instruction set of 8051, write a program in assembly language of 8051 to simulate the MUL instruction. Assuming that two digits are available in A and B registers. Use successive addition method.

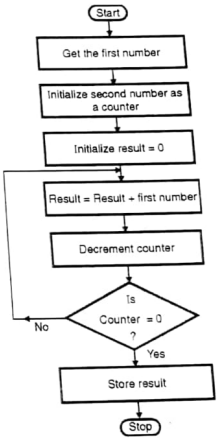
➤➤ **Explanation**

– Consider that a byte is present in the A register and second byte is present in the B register.
– We have to multiply the byte in A with the byte in B.
– We will multiply the numbers using successive addition method.
– In successive addition method, one number is accepted and other number is taken as a counter. The first number is added with itself, till the counter decrements to zero.
– Result is stored in the registers A and B.

For example : A = 12 H, B = 10 H

Result = 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H

Result = 00B4 H



Flowchart 5.3.23

➤➤ **Algorithm**

Step I : Get the first number.
Step II : Get the second number as counter.
Step III : Initialize result = 0.
Step IV : Result = Result + First number.
Step V : Decrement counter
Step VI : If count ≠ 0, go to step V.
Step VII : Store the result.
Step VIII : Stop.

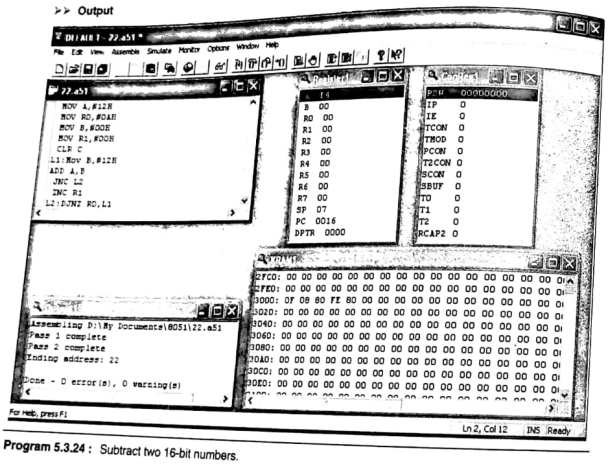
➤➤ **Flowchart :** Refer Flowchart 5.3.23.

➤➤ **Program**

Label	Instruction	Comment
	MOV A , #12H	Get the first number
	MOV R0 , #0AH	Initialize second number as counter.
	MOV B , #00H	Initialize the LSB of result =0
	MOV R1 , #00H	Initialize the MSB of result =0

Label	Instruction	Comment
	CLR C	Clear carry
L1:	MOV B, #12H	
	ADD A, B	Add the numbers.
	JNC L2	check for carry
	INC R1	If carry is there then increment R1
L2:	DJNZ R0, L1	Decrement count and continue till count = 0.
	MOV B, R1	Store the MSB result of multiplication in register B.
	END	END PROGRAM

➢➢ **Output**



Program 5.3.24 : Subtract two 16-bit numbers.

➢➢ **Program Statement**

Write a program to subtract two 16 bit numbers. Let the two numbers be 7856 H and 1234 H. Store the result of subtraction in the DPTR register.

➢➢ **Explanation**

- We are given two 16 bit numbers : 1234 H and 7856 H
- We need to subtract the two LSBs and the two MSBs separately as 8051 is an 8 bit microcontroller.
- We will store the first number in the DPTR and the second number in the A and B registers with register A = LSB of number and register B = MSB of number.
- Then we will subtract the two LSBs i.e. number in register DPL and number in register A. Store the result of LSB subtraction in DPL register.
- Then we will add the two MSBs i.e. number in register DPH and number in register B
- Store the result of MSB subtraction in DPH register.

➢➢ **Algorithm**

Step I : Start

Step II : Get the first 16 bit number in DPTR.

Step III : Get the LSB of the second number in register A.

Step IV : Get the MSB of the second number in register B.

Step V : Subtract the LSB of two numbers.

Step VI : Store the result of LSB subtraction in register DPL.

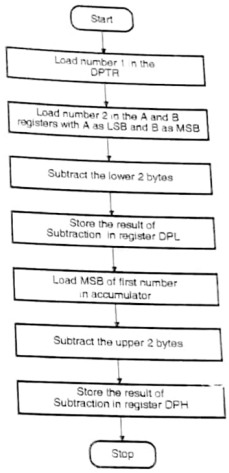
Step VII : Load the MSB of the first number in accumulator.

Step VIII : Subtract the MSB of two numbers.

Step IX : Store the result of MSB subtraction in register DPH.

Step X : Stop

➢➢ **Flowchart** : Refer Flowchart 5.3.24.



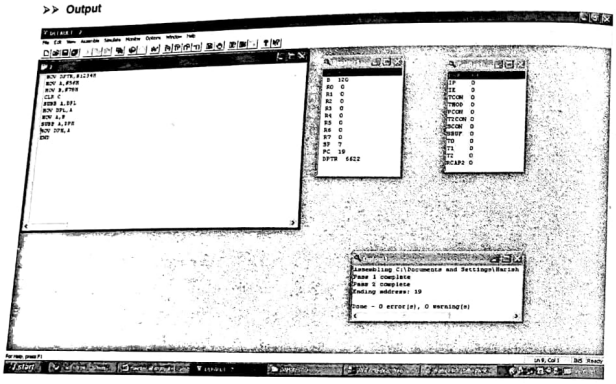
Flowchart 5.3.24

➢➢ **Program**

Instruction	Comment	Operation
MOV DPTR, #1234H	Load the first number in DPTR	DPTR = 1234H
MOV A, #56H	Load the LSB of second number in register A.	A=56H
MOV B, #78H	Load the MSB of second number in register B.	B=78H
CLR C		
SUBB A, DPL	Subtract the two LSBs.	A = 56 – 34 = 12H
MOV DPL, A	Store the result of LSB subtraction in DPL register	**DPL=12H (Result)**

Instruction	Comment	Operation
MOV A, B	Load the MSB of second number in accumulator	A=78H
SUBB A, DPH	Subtract the two MSBs.	A=78-12=66H
MOV DPH, A	Store the result of MSB subtraction in DPH register	DPH = 66H (Result)
END	END PROGRAM	

➢➢ **Output**



Program 5.3.25 : Program for Binary-Gray conversion.

➢➢ **Program statement**

- Write a program in the assembly language of 8051 to convert a given binary number into its Gray code equivalent. Draw flowchart. Store the Gray equivalent in register A.

➢➢ **Explanation**

- For Binary-Gray conversion.
  - (i) Record the MSB as it is.
  - (ii) Add this bit to the next position, recording the sum and neglecting carry if any.
  - (iii) Record successive sums until completed.

- e.g. to convert 0AH i.e. 1010 binary to gray.

Given binary no :- 1 0 1 0

Gray code :- 1 1 1 1 LSB = (0F)H

- For our program the logic that we will be using for Binary-Gray conversion is that first we will add the number with itself.
- Then we will X-OR this added number with the original number. Then we will shift this X-ORed number by 1 bit position to the right. This gives the equivalent gray code. Display this Gray code.
- e.g. number = 0A.

```
    0 A H
+   0 A H
  -------
   14 H
```

XOR added number	0 0 0 1 0 1 0 0	(14 H)
with original number	0 0 0 0 1 0 1 0	(0A H)
	0 0 0 1 1 1 1 0	
Shift by 1 bit to right	0 0 0 0 1 1 1 1	(0F H)

- (0FH) is the gray equivalent of 0A H.

➢➢ **Algorithm**

Step I : Get the number whose gray code equivalent is be found.

Step II : Add number with itself.

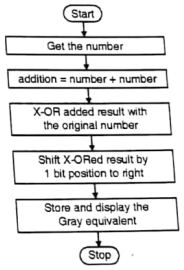
Step III : XOR this added result with the original number.

Step IV : Shift the XOR ed number by 1 bit position to the right to get the Gray equivalent.

Step V : Store the result.

Step VI : Stop.

➢➢ **Flowchart :** Refer Flowchart 5.3.25.



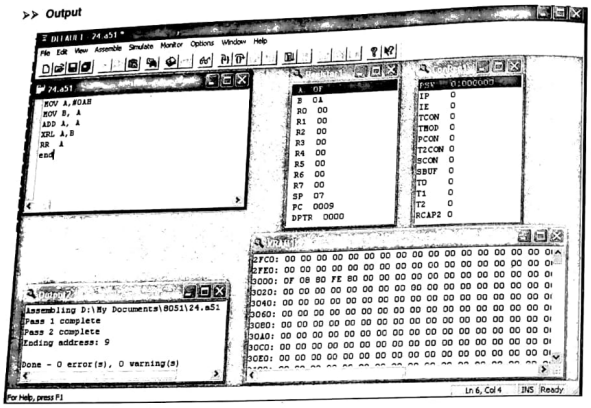
Flowchart 5.3.25

➢➢ **Program**

Instruction	Comment	Operation
MOV A,#0AH	Load number in register A	A=0AH
MOV B, A	Get the number in register B also.	B= 0AH
ADD A, A	Add contents of A with itself	A = 14H , PSW =40H

Instruction	Comment	Operation
XRL A, B	XOR the added contents with number itself	A= 1EH , PSW = 40H
RR A	Roll by 1 bit to right to get gray equivalent	A = 0FH (Result)
END	END PROGRAM	

➤➤ **Output**



Program 5.3.26 : Transfer a block of N bytes from source to destination (Overlapped block transfer)

➤➤ **Program statement**

Write an ALP to move a block of N bytes of data from source to destination. Assume that the length of block is stored in register R2 of register bank 0. The source block starts from memory location 20 H and the destination block begins from memory location 25 H.

➤➤ **Explanation**

- Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
- The number of bytes N (Let N=10 i.e. 0A H) is stored in register R2 of bank 0.
- We will have to initialize this as count.

- For pointing the source address we will use the R0 register and for the destination address we will use the R1 register.
- Transfer the data byte by byte from source to destination block.
- Initialise R2 register with the count.

➤➤ **Algorithm**

Step I : Initialise R2 register with the count.

Step II : Initialise R0 and R1 with the source and destination address of the last element in the block.

Step III : Transfer the data block byte by byte to destination.

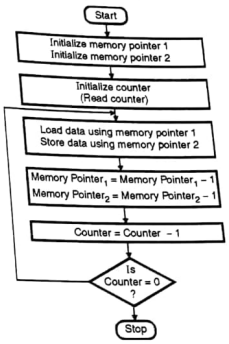
Step IV : Decrement Count

Step V : Decrement source and destination memory pointer.

Step VI : Check for count in R2, if not zero goto step III else goto step VII.

Step VII : Stop.

➤➤ **Flowchart :** Refer Flowchart 5.3.26.



Flowchart 5.3.26

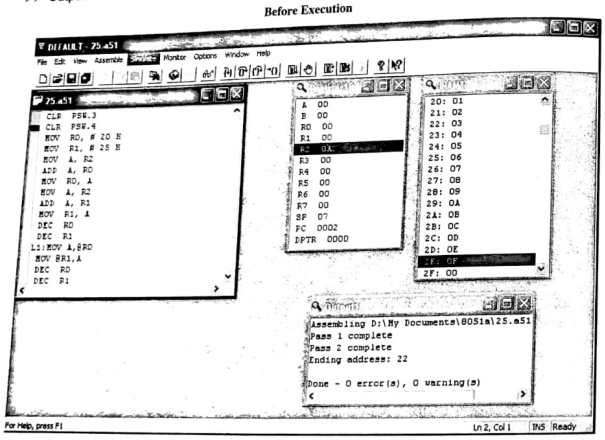
➤➤ **Program**

Label	Instruction	Comment	Operation
	CLR PSW.3 CLR PSW.4	Select Register bank 0	Register Bank 0 selected
	MOV R0, #20H	Initialize register R0 as with source address.	R0 = 20H
	MOV R1, #25H	Initialize register R1 as with destination address.	R1 = 25H
	MOV A, R2	load accumulator with the size of block	A = 0AH
	ADD A, R0		A = 2AH
	MOV R0, A	R0 contains address of the number after the last element in the source block.	R0 = 2AH
	MOV A, R2	load accumulator with the size of block	A = 0AH
	ADD A, R1		A = 2FH
	MOV R1, A	R1 contains address of the number after the last element in the destination block.	R1 = 2FH
	DEC R0	R0 contains address of the last element of source	R0 = 29H

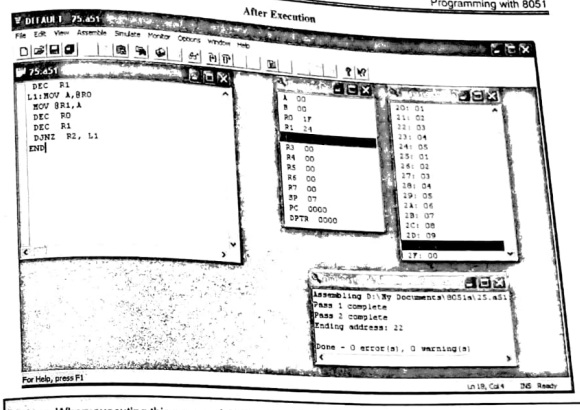
Label	Instruction	Comment	Operation
	DEC R1	R1 contains address of the last element of destination	R1 = 2EH
L1 :	MOV A, @R0	Load accumulator with number from source block.	Loop for transferring byte by byte from source to destination . e.g. data at memory location 20H i.e. 01H is transferred to memory location 25H
	MOV @R1,A	Store the data in desired memory location	
	DEC R0	Decrement source memory pointer	
	DEC R1	Decrement destination memory pointer	
	DJNZ R2, L1	Check if count = 0 ?	
	END	End Program	End

>> **Output**

Before Execution



After Execution



Note : When executing this program initialize R2, to the number of elements.

Program 5.3.27 : Program to find Average of block of N bytes.

>> **Program statement**

– Write an ALP to find the average of block of N bytes.

>> **Explanation**

– Consider that a block of data of N bytes is present at the source location. Let the number of bytes N = 10 for example.

– We have to initialize this as count in the R0 register. Our task is to find out the average of these 10 bytes. i.e. to find out average of 10 numbers.

$$\text{Average} = \frac{\text{sum of N bytes}}{N}.$$

– Assume that the source address is in the R1 register.

– Initially, we will add the N bytes, byte by byte using ADD instruction.

– Once the addition is compute, result of addition will be stored in register A.

– Now, we will divide this sum by number of bytes N to find the average.

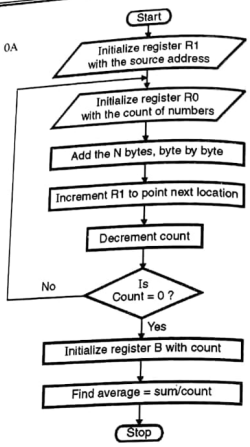
– The result of division i.e. quotient is present in the A register and remainder in the B register.

For example

Block Data : 01 02 03 04 05 06 07 08 09 0A

Sum : 01 + 02 + 03 + 04 + 05 + 06 + 07 + 08 + 09 + 0AH = 37 H

average = $\frac{37\ H}{0AH} = \frac{(55)_{10}}{(10)_{10}}$ = 5 H



Flowchart 5.3.27

➤➤ **Algorithm**

Step I : Initialise R1 with source address.

Step II : Initialise R0 register with the count.

Step III : Add the N bytes, byte by byte.

Step IV : Increment pointer R1.

Step V : Decrement count.

Step VI : Check for count in R0, if not zero goto step II.

Step VII : Initialise register B with count of numbers.

Step VIII : Find average = $\frac{sum}{count}$

Step IX : Stop.

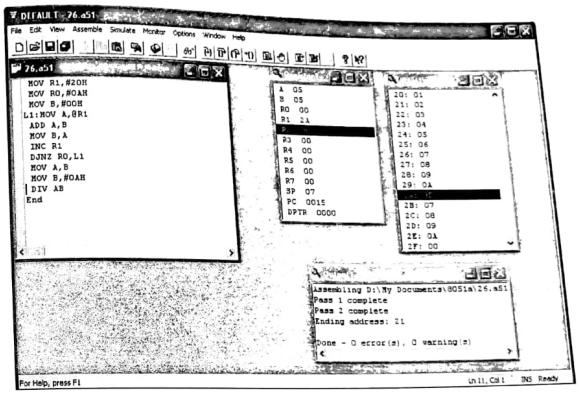
➤➤ **Flowchart :** Refer Flowchart 5.3.27.

➤➤ **Program**

Label	Instruction	Comment	Operation
	MOV R1, #20H	Initialize register R1 as with source address.	R1=20H
	MOV R0, #0AH	Initialize register R0 with count of numbers.	R0 = 0AH
	MOV B, #00H	Initialize result in register B.	B =00H
L1:	MOV A, @R1	Load the number in accumulator.	A = @R1
	ADD A, B	Add the data byte by byte	A = A + B
	MOV B, A	Store the result of addition in register B.	B = A
	INC R1	Increment source memory pointer to next location	R1 = R1 +1
	DJNZ R0, L1	Decrement counter and continue till count =0	IF R0 ≠ 0 THEN R0=R0 – 1 and continue executing L1

Label	Instruction	Comment	Operation
	MOV A, B	Store the result of addition in register A.	A = B
	MOV B, #0AH	Initialize register B with count of numbers.	B = 0AH
	DIV AB	average = sum/count A = quotient, B = remainder	A = 05H , B= 05H (Result)
	End	End program	

➤➤ **Output**



Program 5.3.28 : To reverse the contents of block of N bytes and transfer them from source to destination.

➤➤ **Program Statement**

– Write a program in the ALP of 8051 to reverse the contents of a block of N bytes and transfer the reverse contents to destination block. Assume that the length of block is stored in register R2 of register bank 0. The source block starts from memory location 20 H and the destination block begins from memory location 25 H.

➤➤ **Explanation**

– Consider that a block of data of N bytes is present at source location.

– Now this block of N bytes is to be moved from source location to a destination location in reverse manner. i.e. the contents of the last memory location i.e. the N^{th} memory location will be transferred to the first source location.

– The number of bytes N are stored in register R2 of bank 0.

– We will have to initialize this as count.

– For pointing the source address we will use the R0 register and for the destination address we will use the R1 register.

– Transfer the data byte by byte from source to destination block.

>> Algorithm

Step I : Initialise R2 register with the count.

Step II : Initialise R0 with the source address of the last element in the block and R1 with the start of destination block.

Step III : Transfer the data block byte by byte to destination.

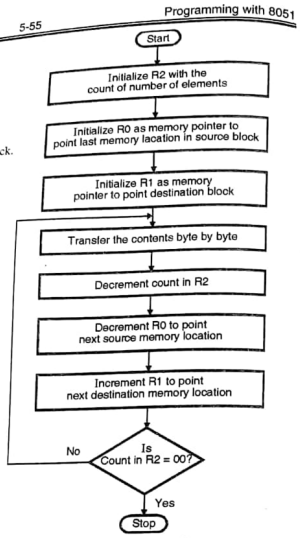
Step IV : Decrement Count

Step V : Decrement source memory pointer and increment destination memory pointer.

Step VI : Check for count in R2, if not zero goto step III else goto step VII.

Step VII : Stop.

>> Flowchart : Refer Flowchart 5.3.28.



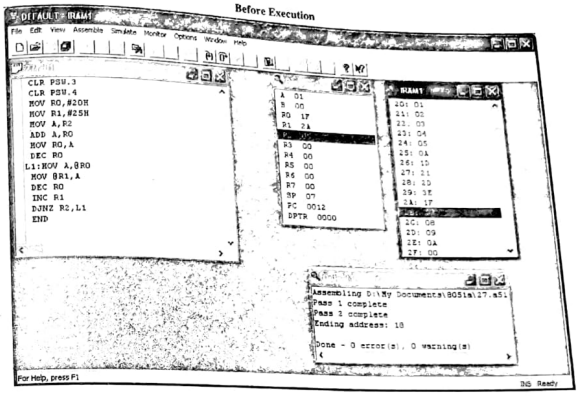
Flowchart 5.3.28

>> Program

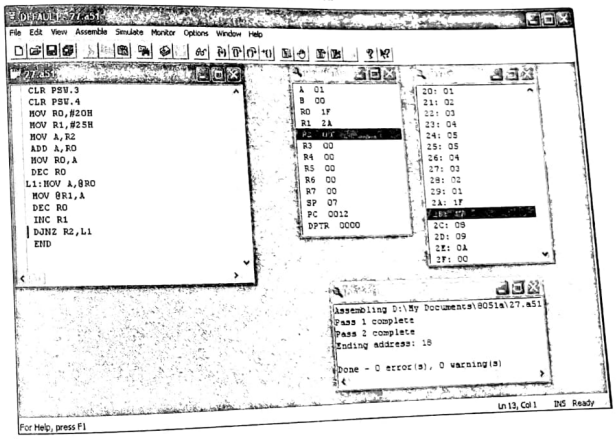
Label	Instruction	Comment
	CLR PSW.3	Select Register bank 0
	CLR PSW.4	
	MOV R0, #20H	Initialize register R0 as with source address.
	MOV R1, #25H	Initialize register R1 as with destination address.
	MOV A, R2	load accumulator with the size of block
	ADD A, R0	
	MOV R0, A	R0 contains address of the number after the last element in the source block .
	DEC R0	R0 contains address of the last element of source
L1 :	MOV A, @R0	Load accumulator with number from source block.
	MOV @R1, A	Store the data in desired memory location
	DEC R0	Decrement source memory pointer
	INC R1	Increment destination memory pointer
	DJNZ R2, L1	Check if count = 0 ?
	END	

>> Output

Before Execution



After Execution



Note : *Initialize R2, to the number of elements before executing.*

Program 5.3.29 : *Add two 16 bit BCD numbers.*

➢➢ **Program statement**

– *Add two 4 digit BCD numbers. Store the result after addition in the DPTR register. Ignore carry after 16 bit.*

➢➢ **Explanation**

– *Consider that two words are given. We have to add these two words. We will load one word in the DPTR register and we will load the LSB of the second number in the A register and the MSB of the second number in the B register.*

– *Using add instruction, add the contents, of the lower two bits i.e. add A and DPL.*

– *The result of this addition is stored in the A register. Store this result in the DPL register.*

– *DAA instruction is then used to convert the result to valid BCD.*

– *Now, add the contents of MSB along with carry if generated in LSB addition.*

– *The result of MSB addition is stored in the A register. Adjust this result to valid BCD number. The final result is available in the DPH register.*

eg. : DPH = 36 H DPL = 29 H

B = 47 H A = 38 H

Step I :

```
    2  9
+   3  8
-------
    6  1   and auxiliary carry = 1
```

As AC = 1, add 6 to make result valid.

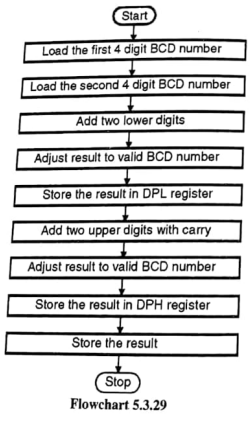
61 + 6 = 67

Step II : 36 + 47 + 0 (Carry of LSB) = 7D.

Lower nibble of addition is greater than 9, so add 6.

```
    7  D
+   0  6
-------
    8  3
```

Final result : 8367 BCD.



Flowchart 5.3.29

➢➢ **Algorithm**

Step I : Load the first number into DPTR register.

Step II : Load the second number into A and B registers.

Step III : Add two lower digits.

Step IV : Adjust result to valid BCD number.

Step V : Store the result in DPL.

Step VI : Add the two upper digits with carry.

Step VII : Adjust result to valid BCD number.

Step VIII : Store the result in DPH.

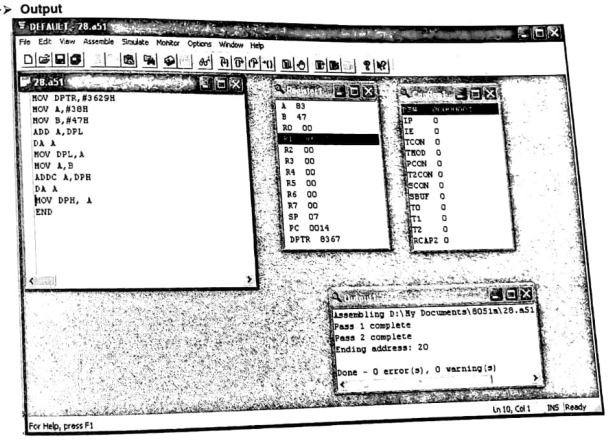
Step IX : Stop.

➢➢ **Flowchart :** Refer Flowchart 5.3.29.

➢➢ **Program**

Instruction	Comment	Operation
MOV DPTR , #3629H	Load the first number in DPTR	DPTR = 3629H
MOV A , #38H	Load the LSB of second number in register A.	A = 38H
MOV B, #47H	Load the MSB of second number in register B.	B = 47H
ADD A, DPL	Add the two LSBs .	A = 61H , PSW = 01000001 (INVALID BCD)
DA A	Adjust Result to Valid BCD	A = 61 + 06 = 67
MOV DPL, A	Store the result of LSB addition in DPL register .	**DPL = 67 (LSB Result)**
MOV A , B	Load the MSB of second number in accumulator	A = 47H
ADDC A, DPH	Add the two MSBs .	A = 7DH
DA A	Adjust Result to Valid BCD	A = 7D +06 = 83
MOV DPH, A	Store the result of MSB addition in DPH register.	**DPH = 83 (MSB Result)**
END	End Program	

➢➢ **Output**



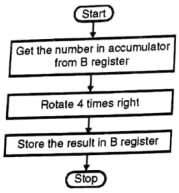
Program 5.3.30 : Program to shift data within 8 bit register.

➢➢ **Program statement**

Write a program to shift an eight bit data by four bits to the right . Assume that the data is present is in register B.

➢➢ **Explanation**

- Get the number in the accumulator from register B.
- Rotate the number 4 times to the right
- Store the shifted number back in register B



Flowchart 5.3.30

➢➢ **Algorithm**

Step I : Get the number in the accumulator from register B .

Step II : Rotate the number 4 times to the right .

Step III : Store the result in register B.

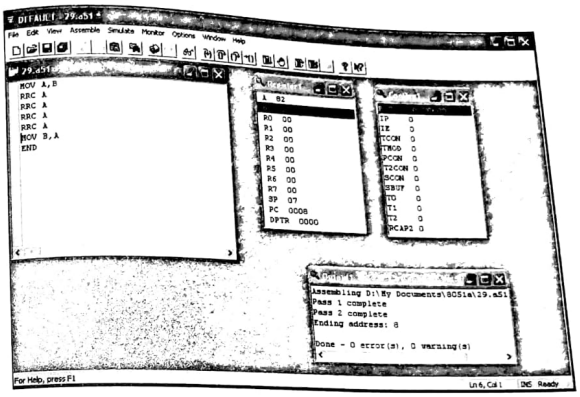
Step IV : Stop

➢➢ **Flowchart :** Refer Flowchart 5.3.30.

➢➢ **Program**

Instruction	Comment	Operation
MOV A,B	Load the number in register A	A= 24H = 00100100 B
RRC A	Rotate the number right	A = 12H = 00010010 B, CY=0
RRC A	Rotate the number right	A = 09H = 00001001 B, CY=0
RRC A	Rotate the number right	A = 04H = 00000100 B, CY=1
RRC A	Rotate the number right	A = 82H = 10000010 B, CY=0
MOV B,A	Store the shifted number	B = 82H (Result)
END	End Program	

➢➢ **Output**



Program 5.3.31 : Program to shift data within 16 bit register.

➢➢ **Program statement**

Write a program to shift an 16 bit data by 1 bits to the right. Assume that the data is present is in register DPTR is 2040 H .

➢➢ **Explanation**

- Get the number in the accumulator from register DPL.
- Rotate the number 1 time to the right
- Store the shifted number back in register DPL
- Get the number in the accumulator from register DPH.
- Rotate the number 1 time to the right
- Store the shifted number back in register DPH

➢➢ **Algorithm**

Step I : Get the number in accumulator from register DPL.

Step II : Rotate the number 1 time to the right.

Step III : Store the result in register DPL.

Step IV : Get the number in accumulator from register DPH.

Step V : Rotate the number 1 times to the right .

Step VI : Store the result in register DPH.

Step VII : Get the number in accumulator from DPL.

Step VIII : Store carry flag in MSB of accumulator

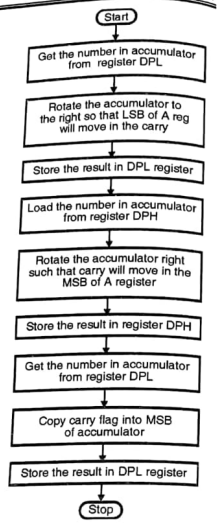
Step IX : Store the result in register DPL.

Step X : Stop

➢➢ **Flowchart :** Refer Flowchart 5.3.31.

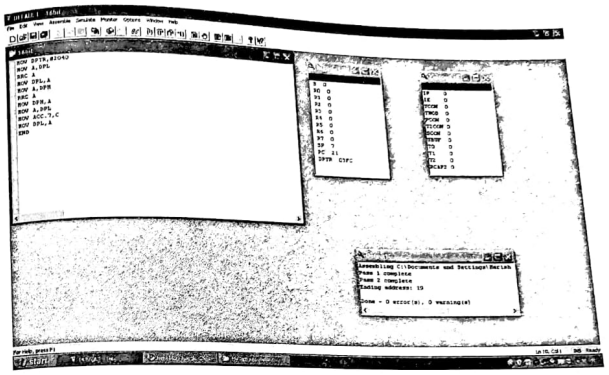
➢➢ **Program**

Instruction	Comment	Operation
MOV DPTR, #2040H		
MOV A,DPL	Load the number in register A	A = 40H
RRC A	Rotate the number 1 time to the right	A= 20H
MOV DPL, A	Store the shifted number	DPL = 20H (Result)
MOV A,DPH	Load the number in register A	A=20H
RRC A	Rotate the number 1 time to the right	A=10H
MOV DPH, A	Store the shifted number	DPH = 10H (Result)
MOV A, DPL	Load the number in register A	A = 20H
MOV ACC.7, C	Copy carry flag in MSB of A	A = 20H
MOV DPL, A	Store the final result	DPL = 20H (Result)
End	End Program	



Flowchart 5.3.31

➢➢ **Output**



Program 5.3.32 : To find and count the number of negative numbers from an array of signed numbers.

MU - Dec. 16.10 Marks

➢➢ **Program Statement**

Write a program in the assembly language of 8051 to find and count the negative numbers from an array of signed numbers. Assume that the array begins at memory location 20 H. Store the count of negative numbers in register B and negative numbers in an array that begins from memory location 30 H.

➢➢ **Explanation**

- An array consisting of negative and positive i.e. signed number is given to us.
- From this array we have to find and count the number of negative numbers.
- We know that if the MSB of a number is 1, then it is a negative number.
- Inorder to check the MSB, we will AND each number with 80H (i.e. 1000 0000). If the MSB is 1, we will increment the count and transfer the negative number to another array arr1. If the MSB is 0, we will check for the next number.
- Repeat this process till all the numbers are scanned.
- Store the count of negative numbers and then the negative numbers.

e.g. Let the numbers be 83H, 12H, F0H, 0BH, 89H.

The number of negative numbers = 3.

The negative number are 83H, F0H, 89H. They are stored at memory locations 30H, 31H, 32H.

➤➤ **Algorithm**

Step I : Load count in R2 register.

Step II : Initialize R0 with the start of array.

Step III : Initialize R1 with the start of array of negative numbers.

Step IV : Load the number in A register.

Step V : AND A with 80H.

Step VI : Is MSB = 1 ? If not, go to step IX.

Step VII : Load that negative number in A and store it to destination array pointed by register R1.

Step VIII : Increment R1 to next location.

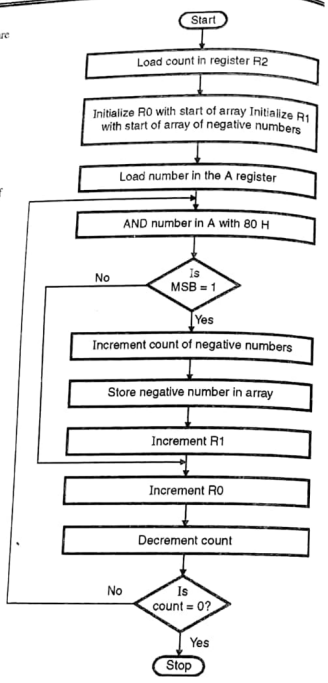
Step IX : Increment R0 to next location.

Step X : Decrement count.

Step XI : Is count = 0 ? If not, go to step V.

Step XII : Stop.

➤➤ **Flowchart :** Refer Flowchart 5.3.32.



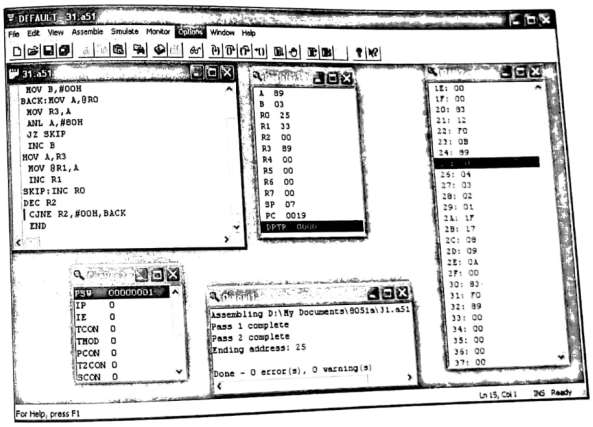
Flowchart 5.3.32

➤➤ **Program**

Label	Instruction	Comment
	MOV R0, #20H	Initialize pointer to start of array
	MOV R2, #05H	initialize count
	MOV R1, #30H	initialize pointer for storing the negative numbers.

Label	Instruction	Comment
	MOV B, #00H	initialize count of number of negative numbers.
BACK:	MOV A, @R0	Get the number
	MOV R3, A	
	ANL A, 80H	Check for MSB
	JZ SKIP	
	INC B	If MSB = 1 then increment negative number count
	MOV A, R3	
	MOV @R1, A	
	INC R1	Increment R1 to point next memory location
SKIP:	INC R0	Increment pointer
	DEC R2	Decrement count
	CJNE R2, #00H,BACK	If count ≠ 0 repeat
	END	End Program

➤➤ **Output**



Program 5.3.33 : Program for checking the parity of number is Odd or even.

➢➢ Program statement given a number which is 8 bit. Write a program in ALP to find the parity of this number. If parity is even display 00 in the result and if parity is Odd display 01 in the result. Draw flowchart.

➢➢ Explanation

- We have a number given. Initially, we will count the number of 1's in that number. For this we will rotate the contents of number bit by bit to right, alongwith carry.
- If carry = 1 then increment the count for number of 1's. In this way we will count the number of 1's.
- Then AND the number of 1's with 01 H. If number is Odd, ZF = 1.
- For even number ZF = 0.
- Store register A = 00 if number is even, otherwise store A = 01 H if the number is odd.
- Store the result.
- For e.g. if the number is 09 H (0000 1001 B) i.e. number has 2 ones. This means that this number has even parity.

➢➢ Algorithm

Step I : Initialize counter = 8 for number of bits and count = 0.

Step II : Get the number

Step III : Rotate the number by 1 bit to right alongwith carry.

Step IV : Check if carry = 1 ? If not goto step VI.

Step V : Increment count for number of 1's.

Step VI : Decrement counter.

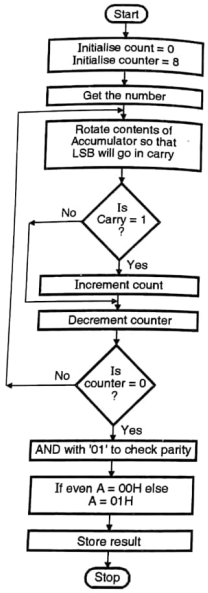
Step VII : Check if count = 0 ? If not, goto step III.

Step VIII : AND with 01 H to check for parity.

Step IX : Store result.

Step X : Stop.

➢➢ Flowchart : Refer Flowchart 5.3.33.

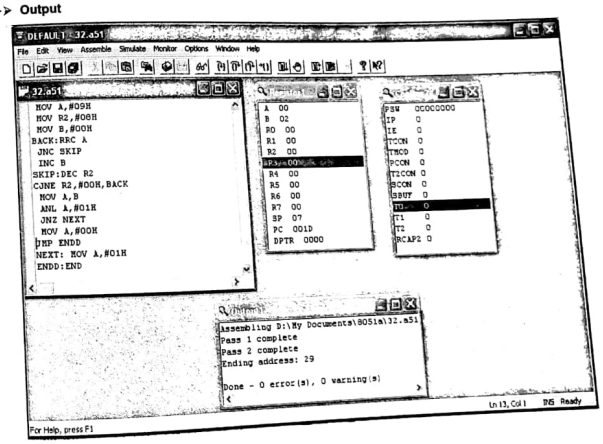


Flowchart 5.3.33

➢➢ Program

Label	Instruction	Comment
	MOV A, #09H	Load the number whose parity is to be found in register A
	MOV R2, #08H	initialize counter
	MOV B, #00H	initialize counter for counting of number of 1's.
BACK:	RRC A	Rotate accumulator right through carry
	JNC SKIP	Check If Carry = 1
	INC B	Increment the count of number of number of 1's
SKIP:	DEC R2	Decrement count
	CJNE R2,#00H,BACK	If count ≠ 0 repeat
	MOV A,B	Store the count of number of 1's in register A
	ANL A,#01H	Check if number of 1's is even or odd to find parity
	JNZ NEXT	If number of 1's odd then jump to NEXT
	MOV A, #00H	Store 00 H in register A if number has even parity
	JMP ENDD	
NEXT:	MOV A, #01H	Store 01 H in register A if number has odd parity
ENDD:	END	

➢➢ Output



Program 5.3.34 : *Program to Subtract two 3 × 3 matrices.*

>> Program statement

- Write a program in ALP of 8051 for subtraction of two 3 × 3 matrices.
- The matrices are stored in the form of lists (row wise) in memory locations 20 H and 30 H. Store the result of addition in third list in external memory location 1200 H.
- Draw flow chart.

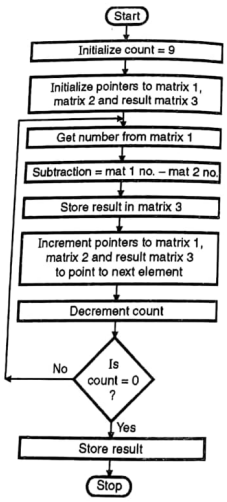
>> Explanation

- We have two matrices mat 1 and mat 2 present. Now our task is to subtract these two matrices and display the result. As it is a 3 × 3 matrix, number of elements present is 9.
- We will initialise counter R2 with 9 as 9 subtractions are to be made. Also we will use three pointers R0 for matrix 1, R1 for matrix 2 and DPTR for result matrix.
- Elements 1 of matrix 1 and matrix 2 are subtracted.
- The result of subtraction is stored in A register and then in the result matrix .
- The counter is decremented.
- Pointers R0 , R1 , DPTR are incremented to point next element in the array .
- The next elements from two matrices is subtracted and result is stored in result matrix.
- The process continues till all the elements are subtracted.

>> Algorithm

Step I : Initialize register R2 counter = 9.
Step II : Initialize pointer R0 to matrix 1.
Step III : Initialize pointer R1 to matrix 2.
Step IV : Initialize pointer DPTR to result matrix 3.
Step V : Get the number from matrix 1.
Step VI : Subtract number from matrix 1 with matrix 2 number.
Step VII : Save the result in result matrix 3.
Step VIII : Increment R0, R1, DPTR to point to next element.
Step IX : Decrement count.
Step X : Check if count = 0, if not goto step V.
Step XI : Store the result.
Step XII : Stop.

>> Flowchart : Refer Flowchart 5.3.34.

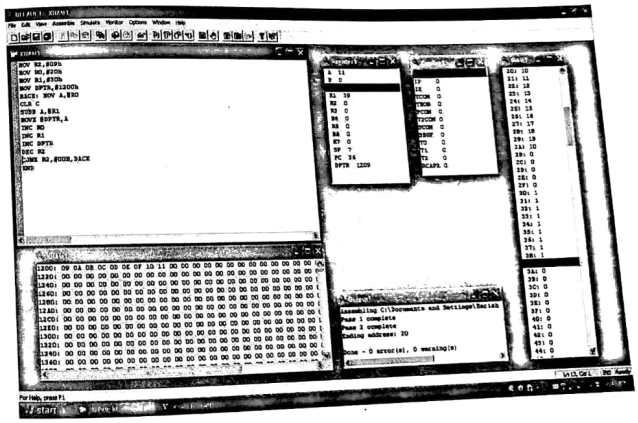


Flowchart 5.3.34

>> Program

Label	Instruction	Comment
	MOV R2 ,#09H	initialize counter
	MOV R0, #20H	initialize pointer to matrix1
	MOV R1, #30H	initialize pointer to matrix2
	MOV DPTR, #1200H	initialize pointer to result matrix3
BACK :	MOV A ,@R0	take no from matrix 1
	CLR C	
	SUBB A, @R1	subtraction =mat1-mat2
	MOVX @DPTR , A	move the result in matrix3
	INC R0	get next element from matrix1
	INC R1	get next element from matrix2
	INC DPTR	Point to next element
	DEC R2	Decrement R2
	CJNE R2,#00H , BACK	continue till all elements are added
	END	

>> Output



Program 5.3.35 : To find the square of a number.

>> Program Statement

Write a program in the assembly language of 8051 to find the square of a number. Display the result in its equivalent decimal form.

>> Explanation

- To find the square of a number we multiply the number with itself.
- The result is stored in register A and B registers in the Hex form.
- Store the result.

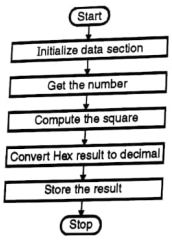
>> Algorithm

Step I : Get the number.

Step II : Compute the square of number. Result is stored in the A and B registers in Hex form.

Step III : Store the result.

>> Flowchart : Refer Flowchart 5.3.35.

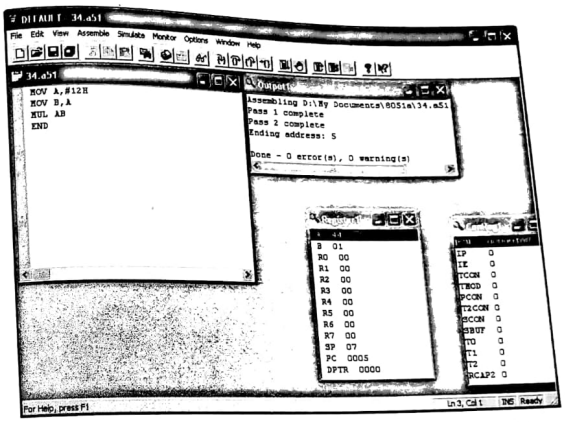


Flowchart 5.3.35

>> Program

Instruction	Comment	Operation
MOV A , #12H	Load number in register A i.e. decimal 12	A=12H
MOV B,A	Load the number in register B	B=12H
MUL AB	Compute the square. Result LSB in register A and MSB in B	A=44H , B= 01 (Result)
END	End Program	

>> Output



Program 5.3.36 : To find Fibonacci series of N given terms.

>> Program Statement

- Write a program in the assembly language of 8051 to compute the Fibonacci series of N terms.

>> Explanation

- The Fibonacci series is

  0 1 1 2 3 5 8 13 21....

- The first two terms are 0 1, the third term is computed as 0 + 1 = 1, fourth term 1 + 1 = 2, fifth term = 1 + 2 = 3. i.e. n^{th} term = $(n-2)^{th}$ term + $(n-1)^{th}$ term.
- We will find the series say for 10 terms i.e. N = 10.
- We will initialize count = 10. An array is initialized, so that the computed terms will be stored in it. R0 is initialized to start of array.
- The first term i.e. 0 is stored at start. R0 is incremented and next term is stored i.e. 1. Then addition of contents of two locations i.e. 0 and 1 is done.
- The next term is computed and result is stored at third location. Process is repeated till all 10 terms are computed.
- Store the result.

– The result in Hex will be

1 1 2 3 5 8 0D 15 22.

>> Algorithm

Step I : Initialize the counter = 10 i.e. 0AH.

Step II : Initialize R0 to starting address of the array.

Step III : Store the first term at location where R0 is pointing.

Step IV : Increment R0 to point next location.

Step V : Store next term '1' at location where R0 is pointing.

Step VI : Increment R0 to next location.

Step VII : next term = [R0 – 2] + [R0 – 1]

Step VIII : Store the result to location that R0 indicates.

Step IX : Increment R0 to point next location.

Step X : Decrement counter.

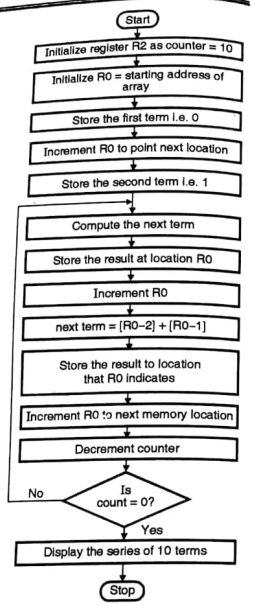
Step XI : Check if count = 0, if not go to step VI.

Step XII : Display the series of 10 terms.

Step XIII : Stop.

>> Flowchart :

Refer Flowchart 5.3.36.



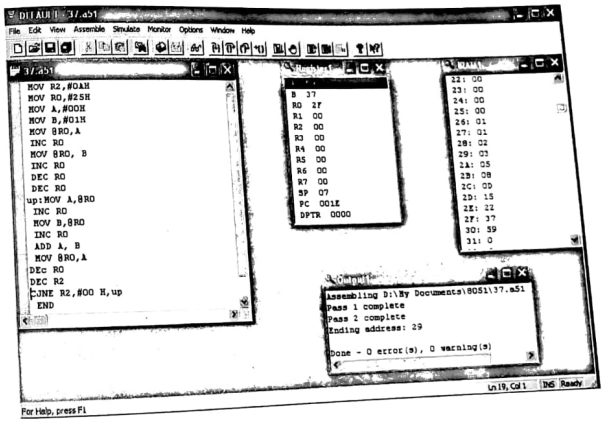
Flowchart 5.3.36

>> Program

Label	Instruction	Comment
	MOV R2 , #0AH	initialize counter
	MOV R0 , #25H	initialize R0 to start of series
	MOV A, #00H	initialize A=0 i.e. first term of series
	MOV B , #01H	initialize B=1 i.e. next term of series
	MOV @ R0, A	store the first term of series
	INC R0	increment R0 to next location
	MOV @R0, B	store next term in series

Label	Instruction	Comment
	INC R0	increment R0 to next location
	DEC R0	decrement R0 to previous location
	DEC R0	decrement R0 to previous location
up :	MOV A , @R0	Store term in A
	INC R0	increment R0 to next location
	MOV B , @R0	store next term in series
	INC R0	increment R0 to next location
	ADD A, B	compute the next term in series
	MOV @R0, A	store the computed term
	DEC R0	decrement R0 to previous location
	DEC R2	decrement R2
	CJNE R2, # 00 H, up	repeat till all terms are calculated
	END	End program

>> Output



Program 5.3.37 : Packed BCD number is stored at RAM location 30 H. Write assembly language for 8051 to convert this number to ASCII number and store it in 32 H and 33 H.

>> Explanation

- We have a packed BCD number stored at RAM location 30 H. We will unpack the number, as the two unpacked digits will give two ASCII numbers.
- After unpacking digits, we will add 30 H to the unpacked digits, to convert it to its ASCII equivalent.
- Store the digits at memory locations 32 H and 33 H.

>> Algorithm

Step I : Get the number stored at RAM location 30 H in accumulator.

Step II : Unpack the number.

Step III : Add 30 H to each digit to convert it to its ASCII equivalent.

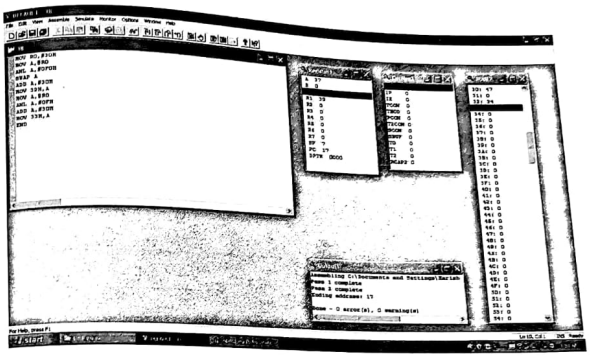
Step IV : Store the numbers at locations 32 H and 33 H.

Step V : Stop.

>> Program

Instruction	Comment	Operation
MOV R0, #30H	Get number stored at memory location 30H in accumulator by using R0 as pointer.	R0 = 30H
MOV A, @R0		A = 47H
ANL A, #0F0H	mask lower nibble.	A = 40H
SWAP A	make MSB digit = LSB digit	A = 04H
ADD A, #30H	convert number to ASCII equivalent	A = 34H
MOV 32H, A	store the digit at location 32 H	32H = 34H (Result)
MOV A, @R0	get the number back in accumulator	A = 47H
ANL A, #0FH	mask upper nibble	A = 07H
ADD A, #30H	convert number to ASCII equivalent	A = 37H
MOV 33H, A	store digit at location 33 H	33H = 37H (Result)
END		

>> Output



Program 5.3.38 : The number A4H is placed somewhere in the external RAM between 0500H and 0600H. Find the address of the number and store it in R4(LSB) and R5(MSB). Write 8051 code to do above operation.

Soln. :

>> Program

Label	Instruction	Comment	Operation
	MOV R0, #0FF	Load R0 as counter for the size of array	R0=FFH
	MOV DPTR, #0500H	Load DPTR with the address of data	DPTR = 0500H
L1:	MOVX A, @DPTR	Load accumulator with data	Loop for searching byte A4H.
	CJNE A, #A4H, L2	Search byte A4 H.	
	MOV R4, DPL	Store LSB of address of matched location in R4.	R4 will have result LSB of address
	MOV R5, DPH	Store MSB of address of matched location in R5.	R5 will have Result MSB of address
HERE:	SJMP HERE	Stop	Stop

Label	Instruction	Comment	Operation
L2 :	INC DPTR	Increment pointer to point next memory location.	DPTR = DPTR + 1
	DJNZ R0 , L1	Decrement counter	R0 = R0 – 1 , if R0 ≠ 0 go to L1
	End	Stop	

Program 5.3.39 : Write instruction sequence in 8051 to reverse the bits in accumulator. Bit 7 and bit 0 are swapped, bit 6 and bit 1 are swapped etc.

Soln. :

Label	Instruction	Comment
	MOV A, #23H	
	CLR C	Clear C
	MOV C, ACC.7	
	MOV F0H, C	Move carry flag to bit 0 position of register B.
	MOV C, ACC.0	Move bit 0 of accumulator into carry
	MOV ACC.7, C	Store bit 0 at bit 7
	MOV C, F0H	
	MOV ACC.0, C	Store bit 7 at bit 0
	MOV C, ACC.6	
	MOV F1H, C	Move carry flag to bit 1 position of register B.
	MOV C, ACC.1	
	MOV ACC.6, C	Store bit 1 at bit 6
	MOV C, F1H	
	MOV ACC.1, C	Store bit 6 at bit 1
	MOV C, ACC.5	
	MOV F2H, C	Move carry flag to bit 2 position of register B
	MOV C, ACC.2	
	MOV ACC.5,C	Store bit 2 at bit 5
	MOV C, F2H	
	MOV ACC.2 C	Store bit 5 at bit 2
	MOV C, ACC.4	
	MOV F3H, C	Move carry flag at bit 3 position of register B
	MOV C, ACC.3	
	MOV ACC.4, C	Store bit 3 at bit 4
	MOV C, F3H	
	MOV ACC.3, C	Store bit 4 at bit 3.

Syllabus Topic : Development Tools

5.4 Assembly Language Program Development Tools

Development tools for microcontroller

The basic development tools used for microcontroller based systems are :

(i) Software development tools
(ii) Hardware development tools

5.4.1 Software Development Tools

The software development tools used for microcontroller based systems are :

1. Assembler

- Each assembly level instruction has a mnemonic. For example in the instruction MOV A, #40H. MOV represents the mnemonic.
- An assembler is a program which translates the assembly language mnemonics into corresponding binary codes.
- The assembler reads the source file more than once.

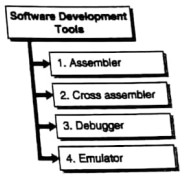


Fig. 5.4.1 : Software Development Tools

Assembler operation

- The assembler first reads the source file of program.
- Then it determines the displacement of data items, offsets of labels etc. and puts this information into a symbol table.
- Then it produces the binary codes for each assembly language instruction and detects syntax errors if any. Then it inserts the offsets etc. calculated earlier.

File Generation in assembler

- An assembler generates two files namely the **object file** and the **assembler list file.**
- The object file is given extension .OBJ whereas the assembler list file is given extension .LST.
- **Object file :** It contains the binary codes of the program instructions and the information about the addresses of instructions.
- **List file :** It contains the assembly language statements, the binary codes for each instruction and the offset of each instruction.
- Any typing or syntax errors are indicated in the assembly listing if we take a print out of .LST file.

Error detection and correction

- The assembler is capable of only finding the syntax errors.
- To check if our program is working, we have to test and run the program.
- The errors indicated by the assembler should be edited using the editor.

- This edit-assemble loop should be executed till all the errors are corrected.

➔ **2. Cross assembler**

- The special feature of the cross assembler is that it is not written in the same language that is used by the microcontroller that executes the machine code that is generated by the assembler.
- The cross assembler is usually written in a high level language like FORTRAN, C, PASCAL etc that makes it machine independent.
- An assembler that runs on one type of computer and assembles the source code for a different target computer is called as a **cross assembler.**
- For example,
  (i) An assembler that runs on an Intel x86 machine and generates object code for Motorola's 68HC05.
  (ii) 8086 assembler may be written in C and then the assembler may be executed on some other machine like Motorola 6800.

➔ **3. Debugger**

- A software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.
- If a program is directly accessible from the microcomputer and does not need any external hardware, then we can use a debugger to run and test the program.
- Debugger is basically a program which permits the user to load object code program into the system memory, execute the program and debug it.
- The debugger also permits the change in register contents, memory locations and rerun the program.
- With the help of the debugger, we can stop the program execution after each instruction so that we can check or alter the memory and register contents.
- In other words we can put breakpoints in the program and execute the program from one breakpoint to the other.
- It is possible to examine the register and memory contents after partial execution of program between the breakpoints.
- We can use the debugger to check and correct the program till all the errors are corrected.
- For most IBM PC type computers the basic debugger comes by default.
- They make the debugging easier and allow the user to see the contents of registers and memory locations as the program is executed.

➔ **4. Emulator**

- The emulator is used to test and debug the hardware and software of an external system such as the Microcontroller based system.
- Emulator is a combination of hardware and software.
- An emulator consists of a multi-wire cable that connects the host system to the external system.

- Through this cable the software of the emulator allows the user to download the object code program into RAM in the external system being developed.
- Like the debugger the emulator also allows the user to load the programs to be tested, run the, programs check and modify the contents of various registers and memory locations and also insert the breakpoints.
- As each instruction in the assembly language program is executed, the emulator as if takes "snapshot" of the register contents, activities on the address and data buses and the state of flag register.
- The emulator stores this data as "trace data".
- It is possible to take out the print out of the trace data so as to analyse the results produced in the program on the step by step basis.

5.4.2 Hardware Development Tools

It is a difficult task to develop a microprocessor based system. In absence of developing tool, the task is time consuming and hectic. Three tools are used for the development of a microprocessor / microcontroller based system. They are in-circuit emulator, logic analyzer and simulator. Let us study them one by one.

5.4.2.1 An In Circuit Emulator

- An **in-circuit emulator** (ICE) is a hardware device used during the development of **microprocessor based / microcontroller based systems.** Virtually all such systems have a hardware element and a software element, which are separate but highly interdependent.
- The ICE allows the software element to be run and tested on the actual hardware on which it is to run, but still allows programmer conveniences such as source-level **debugging** and single-stepping, etc.

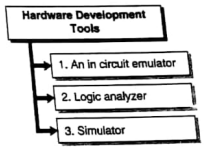


Fig. 5.4.2 : Hardware Development Tools

- Without an ICE, the development of microprocessor / microcontroller based systems can be extremely difficult, since if something does not function correctly, it is often very hard to tell what went wrong without some sort of monitoring system to oversee it.
- Most ICEs consist of an adaptor unit that sits between the host computer and the system to be tested. A large header and cable assembly connects this unit to where the actual **CPU or microcontroller** mounts within the system to be tested.
- The unit emulates the CPU, such that from the system's point of view, it has a real processor fitted. From the host computer's point of view, the system under test is under full control, allowing the developer to debug and test code directly.
- The emulator is used to test RAM, I/O ports and control functions of the development system by replacing the microprocessor IC on the board by the in-circuit emulator. E.g. with the help of the in-circuit emulator a user could stop in between during the execution of a program for examining the contents of memory locations and registers. This helps the user to see the intermediate results obtained at a particular condition, by which the user can come to know why the hardware is not performing upto the expectations.

5.4.2.2 Logic Analyzer

Suppose we have to analyze a memory chip which is soldered on a PC board. In order to analyze the memory chip the analog analyzers are incapable because the number of signals to be observed are very larger in number and they are digital in nature i.e. we should have a device which is capable of displaying a number of signals, which is compatible with the TTL and CMOS logic levels. It should be able to sense and display the address bus, data bus, memory read, memory write, I/O read, I/O write etc. signals. All these tasks can be achieved by a Logic Analyzer. The normal oscilloscope deals with time domain, spectrum analyzer with frequency domain and the logic analyzer with digital domain.

5.4.2.2(A) Features

1. *The logic analyzer displays signals from many inputs at a time. The number of inputs may be 16, 32, 48, 64 or even more.*
2. *The logic analyzer shows the display in sequence of instructions that have occurred.*
3. *It responds to the logic levels i.e. it only displays logic 0 or 1 state of input signal.*
4. *It takes input samples and stores the samples in its own memory.*
5. *It has capability of displaying the words that occurred before and after the trigger.*
6. *The logic analyzer displays data in four different ways. They are :*
   - (i) *Timing diagram method*
   - (ii) *Logic state method*
   - (iii) *Hexadecimal method*
   - (iv) *Map method*
7. *It is used to identify a malfunctioning system, with the help of its map display. In this method the map of system is compared with map of a good system.*
8. *The logic analyzer has a programmable trigger facility, which allows the user to take input sample at any instant.*

The logic analyzers are of two types :

(a) **Logic timing analyzer :** *In this analyzer, the data is sampled as the clock signal is generated and at regular intervals. The data sampled is stored in the memory and this stored information is displayed. It is preferred for troubleshooting of the problems related to the computer hardware. It is an asynchronous measurement method.*

(b) **Logic state analyzer :** *In this analyzer, the data is sample when the clock signals are synchronized with the measured device. The data sampled is stored in the memory and this stored information is displaying in the binary or hexadecimal format. This method is preferred for troubleshooting of software problems. It is a synchronous measurement method.*

The logic analyzer is basically a multichannel oscilloscope. Fig. 5.4.3 shows the block diagram of a logic analyzer.

- *The probes connect the logical analyzer to system which is under test. The probes operate as voltage divides, the lowest possible slew rate can be selected by dividing the input signal. This helps the device to capture high speed signals.*

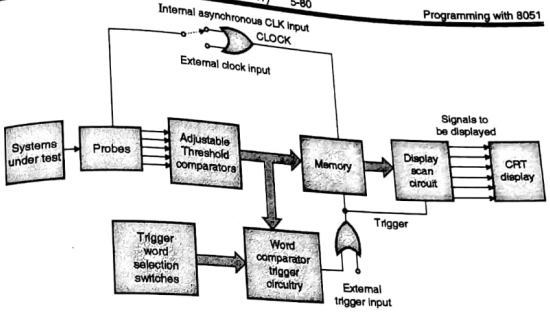


Fig. 5.4.3 : Block diagram of logic analyzer

- The different logic families i.e. TTL, CMOS, NMOS etc. have different threshold voltages and hence adjustable threshold comparators are used. Each signal is connected to each line of the logic analyzer. The reference signal of each comparator is set to a voltage which is equal to the logic threshold voltage of the logic family under test.
- The logic analyzer memory consists of a RAM. The clock signals i.e. internal or external clock input is connected to the memory. On receiving clock signal, the logic analyzer samples the data present on input signals. These samples are stored in the memory. For each input channel the analyzer can store from 256 to 1024 samples.
- When the memory receives a trigger signal then the samples are stored in it and displayed on the CRT display. This trigger signal may be provided externally or it may be provided from the word recognizer circuitry.
- We can set a binary word using switches or through keyboard in the word recognizer circuit. The word recognizer circuit compares this word with the binary input word. When the two words match it sends a trigger signal to the memory. When the memory receives a trigger signal, it sends the samples to a CRT display. There are three types of displays, depending on the trigger signal.

(i) Pre-trigger display
(ii) Post-trigger display
(iii) Center trigger display

(i) In **pretrigger mode**, the memory acts as a loop. The samples before the trigger event are captured which represent the time before event. It is useful to find the causes of malfunctioning of a circuit. It is also called as negative time capturing.

(ii) In the **post trigger mode**, the memory displays the samples which are captured after the occurrence of trigger signal.

(iii) In the **center trigger mode** half of samples are taken before the trigger signal and the other half samples are taken after the trigger signal.

5.4.2.2(B) Display Methods

The logic analyzer allows the user to display the samples it stores in the memory in four ways.

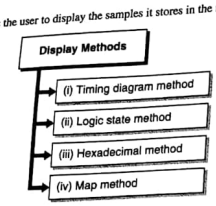


Fig. 5.4.4 : Display methods

➔ **(i) Timing diagram method**

In this method a particular portion could be zoomed in or zoomed out on the timing diagram. Such a display is best suitable for finding out glitches and displaying long data sequences.

Fig. 5.4.5 shows this method.

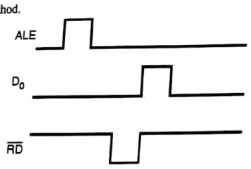


Fig. 5.4.5 : Timing diagram method

➔ **(ii) Logic state method**

In this method the sampled data is displayed in logic state format of 1's and 0's. This format is very easy to read.

Fig. 5.4.6 shows display of data sampled using logic state method.

A_0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	1
A_1	0	1	0	0	0	1	1	0	0	1	1	1	0	0	0	0
A_2	1	0	1	0	1	0	0	1	1	0	1	1	1	1	0	1
A_3	0	1	0	1	1	0	1	1	0	0	0	1	1	0	1	1
A_4	1	0	1	1	0	1	0	0	1	1	0	1	0	0	1	1
A_5	0	1	0	1	0	1	1	0	0	1	0	0	1	0	0	1
A_6	0	0	1	0	1	0	0	1	1	0	1	1	1	1	1	1

Fig. 5.4.6

➔ **(iii) Hexadecimal method**

This method is used whenever it is essential to improve the readability of sampled data. The sampled data is displayed in hexadecimal format. Fig. 5.4.7 shows this. Whenever the contents of address and data bus are to be observed, this method is used.

$A_0 - A_7$	20	50	7F	30	31	2E	3B	5A
$D_0 - D_7$	54	76	8E	2B	1A	3C	4D	7E

Fig. 5.4.7

➔ **(iv) Map method**

In this method the data sampled is sampled in the form of a map. This method is most difficult method in regards to reading the data, but it is very useful inorder to identify a malfunctioning system. This is done by comparing its map with that of a good system.

5.4.2.2(C) Applications of a Logic Analyzer

- They are used for the troubleshooting and analysis of complex digital systems.
- As it has the facility of asynchronous internal clocking, the activities in the system which is under the test can be seen on real time basis.
- It can be used to observe up to 64 signals at a time, while the oscilloscope can be used to observe 4 channels at a time.
- It has compatibility with RS 232 and IEEE 488 and can be attached to printer, for taking hard copy of the signals.
- It is able to detect and trigger on signal glitches. A glitch is a signal which makes a transition through the threshold voltage two or more times between the successive clock samples. Glitches are unwanted signals. They can cause malfunctions in the system.

5.4.2.3 Simulator

- A **Simulator** is often used to execute a program that has to run on some inconvenient type of computer. For example, simulators are usually used to debug a microprogram.
- Since the operation of the computer is simulated, all of the information about the computer's operation is directly available to the programmer, and the speed and execution of the simulation can be varied as per the user's wish.
- Simulators may also be used to interpret **fault trees**, or test logic designs before they are constructed.
- Many video games are also simulators, that are implemented inexpensively.
- Simulator is a software package that functions like a hardware without acquiring hardware. The 8051 microcontroller simulator gives the user an 8051 environment on the PC
- It performs the functions of the 8051 microcontroller / 8085 microprocessor without using it.
- It also performs a simulation of different peripherals that are used with the microcontroller / microprocessor in any application
- It provides facilities that help the user to find logical errors, facilities to help the user learn about the initialization of different peripherals and get an insight of the microcontroller functioning.

- It does acts as a direct replacement of the costly 8051 microcontroller kit.
- It shows all internal registers, entire memory and peripherals on the monitor.
- It supports breakpoint and single stepping facilities that help the user to debug their programs.

5.4.2.3(A) Computer Configuration Required to Run the Simulator

The 8051 simulator runs on the PC. The requirement for PC is

(a) RAM of 512 KB.

(b) DOS Compatibility

(c) IBM Mono, CGA, EGA, VGA or Compatible Monitor.

(d) Two Disk Drives.

5.4.2.3(B) Features

- Easy to operate.
- Allows Simulation of peripherals.
- Allows Simulation of Interrupts.
- Provides continuous display of the system register values.
- The Program can be easily modified.
- The core can be viewed and the corresponding instruction makes it easy to understand the logic of the program while the program executes.
- It is a powerful debugging tool with different high level debugging facilities.
- It saves the development time.
- It allows checking of Software before the hardware is available to the user.
- It has the ability for the user to construct screens that show various parts of the 8051 system. Each screen is made up of separate windows that display internal CPU registers, code and Data Memory areas.
- The contents of any location in the code ROM and Internal RAM (including SFR's) and External Memory can be changed as the program executes.
- The I/O ports may be simulated by changing the value of the port SFR's.
- Interrupts are simulated by striking the function on the keyboard.

For executing a simulation, the screen set file is loaded into the simulator first. This file is followed by a program that is written in the object code format. The program is then executed using these simulator commands.

1. Reset the Program Counter to 0000H.
2. Single Step the Program
3. Free run the program.
4. Free run until breakpoint is reached
5. Stop Free run

5.4.2.3(C) Modes of Simulator

The simulator has three modes of operation. They are :

1. **Idle mode :** This is a non executing mode. This mode allows the user to set the configuration i.e. loading the program file, saving required memory contents on the disk, change default directory or drive etc.
2. **Execute Mode :** This mode is a continuous execution mode .In this mode the program that the user wishes is continuously executed.
3. **Single Step Mode :** In this Mode the user program is executed step by step When the user presses the key F2 an instruction is executed and the PC points to the next instruction.

Syllabus topic: Assembler Directives

5.5 Assembler Directives

→ (MU - Dec. 15, May 17)

Q. 5.5.1 Explain in brief Assembler Directives with respect to 8051 assembler. (Ref Sec. 5.5) **Dec. 15. 3 Marks**

Q. 5.5.2 Write note on Assembler directives (Ref. Sec. 5.5) **May 17. 5 Marks**

- The assembler directives are the statements that direct the assembler to do something. The speciality of the statements is that they are effective only during the assembly of a program but they do not generate any code that is machine executable.
- The following are widely used assembler directives for 8051 :

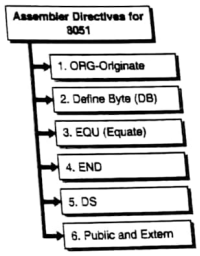


Fig. 5.5.1

→ **1. ORG-Originate**

- The ORG directive allows us to set the beginning address.
- The number after the ORG can be in hex or decimal. If the number is decimal, the assembler will convert it to hex.
- Its format is ORG expression.

E.g. : ORG 00 H ; start program at location 0.

→ **2. Define Byte (DB)**

- This directive defines byte type variable.
- When DB used to define data, the numbers can be in decimal, binary, hex or ASCII formats.
- For the binary number B is used after the number. Similarly, an H is used after hexadecimal number.
- Irrespective of the type of byte, the assembler will convert the byte to hex.
- To indicate ASCII numbers, the characters are placed in quotation marks ('character').
- DB is the only directive that can be used to define ASCII strings greater than two characters.

– The format of DB directive is,

```
          name    DB      initial value
e.g.:     DAT1    DB      24          ; Decimal
          DAT2    DB      01101111B   ; Binary
          DAT3    DB      "235"       ; ASCII
          DAT4    DB      10H         ; Hexadecimal
```

– DB is also used to allocate memory in byte-sized chunks.

➔ **3. EQU (Equate)**

– It is used to give a name to some value or symbol in the program.

– EQU is used to define a constant without occupying a memory location.

– Each time when the assembler finds that name in the program, it replaces that name with the value assigned to that variable.

– Its format is

[name] EQU initial value.

e.g. FACT EQU 04H

– This statement is written at the beginning of the program and whenever FACT appears in an instruction or another directive, the assembler substitutes the value 4.

– The advantage of using EQU in this manner is that if FACT is used several times in a program and the value has to be changed, all that has to be changed is the EQU statement and reassemble the program.

– The assembler with automatically put the new value each time it finds the name FACT.

➔ **4. END**

– This directive is placed at the end of the source and it acts the last statement of a program.

– The END directive terminate the entire program.

– The assembler will neglect any statement after an END directive.

– The format of the END directive is as follows : **END**.

➔ **5. DS**

This directive declares a 24 byte stack in data segment that starts at internal RAM location 08H.

➔ **6. Public and Extern**

– The public and extern directive are used where the source files start. The directive public declares the variables defined in a specific file that can be used in the other source files.

– The directive extern declares the variables that are used in the present source file but are defined in some other source file.

5.6 Exam Pack (University and Review Question)

Q. 1 Write assembly language program for 8051 to find the 2's complement of the number. **(Refer program 5.3.4)** **(10 Marks)**

Q. 2 Write assembly language program for 8051 to unpack the packed BCD number. **(Refer program 5.3.6)** **(10 Marks)**

Q. 3 Write assembly language program for 8051 to count the number of 1's and 0's in a number. **(Refer program 5.3.9)** **(10 Marks)**

Q. 4 Write a assembly language program for 8051 to find largest number from a data block of ten bytes that present in internal memory locations 20 H to 29H. Store the result in memory location 2A H. **(Refer program 5.3.13)** **(M-15, 10 Marks)**

Q. 5 Program to sort the numbers in ascending order. **(Refer program 5.3.19)** **(M-16, 10 Marks)**

Q. 6 Write assembly language program for 8051 to multiply two 8 bit numbers stored in external memory locations 4000 H and 4001 H. Send the result on PORT 1 and PORT 3. **(Refer program 5.3.21)** **(D-14, 10 Marks)**

Q. 7 Write assembly language program for 8051 to multiply two 8 bit numbers using successive addition method. **(Refer program 5.3.23)** **(10 Marks)**

Q. 8 To find and count the number of negative numbers from an array of signed numbers. **(Refer program 5.3.32)** **(D-16, 10 Marks)**

Q. 9 Write assembly language program for 8051 to Subtract two 3 × 3 matrices. **(Refer program 5.3.34)** **(10 Marks)**

☞ **Syllabus topic: Assembler Directives**

Q. 10 Explain in brief Assembler Directives with respect to 8051 assembler. **(Refer section 5.5)** **(D-15, 3 Marks)**

Q. 11 Write note on Assembler directives. **(Refer section 5.5)** **(M-17, 5 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 6

Programming Input / Output, Timers and Interrupt Service Routines

Syllabus Topic : Programming Based on I/O Parallel and Serial Ports, Timers

6.1 Programming the 8051 Timer and I/O Ports

We have studied the details of 8051 timer / counter earlier in the chapter 3. In, this section we will study its programming.

Program 6.1.1 : Indicate the mode in which the timer will be operated after the execution of the following instructions :

(i) MOV TMOD, #20H

(ii) MOV TMOD, 02H

Solution :

Initially we will convert the hex values to binary.

(i) TMOD = 0010 0000. Hence mode 2 of timer 1 is selected.

(ii) TMOD = 0000 0010. Hence mode 2 of timer 0 is selected.

Program 6.1.2 : Estimate the timer's clock frequency and its period, for the 8051 based system that has clock frequency of 16 MHz.

Solution :

We know that the frequency for the timer is $\frac{1}{12}^{th}$ of the crystal frequency.

$$\therefore \text{ Clock frequency } = \frac{1}{12} \times 16 = 1.333 \text{ MHz}$$

$$\text{Clock period } = \frac{1}{f} = \frac{1}{1.333} \text{ MHz} = 0.75 \ \mu s$$

6.1.1 Programming the Timer in Mode 1

Following are the steps to program the timer in mode 1.

Step I : Load the TMOD register.

Step II : Load the TL and TH registers with initial count values.

Step III : Start timer.

Step IV : Observe the timer flag (TF). Exit from the loop when the TP flag is set.

Step V : Stop timer.

Step VI : Clear the timer for next round.

Step VII : Go back to step II.

Program 6.1.3 : Assuming that XTAL = 11.0592 MHz, write a program to generate a square wave of 2 KHz frequency on pin 1.5.

Solution :

The period of square wave is $\frac{1}{2 \text{ kHz}} = 500 \ \mu s$

Let us assume that duty cycle of square wave is 50%. Hence, the square wave will be high for 250 μs and it will be low for 250 μs.

XTAL = 11.0592 MHz i.e. the counter will count up every 1.085 μs.

$\frac{250 \ \mu s}{1.085 \ \mu s} = 230$, 1.085 μs intervals will make a 500 μs / 2 KHz pulse.

The values that should be loaded into TH and TL registers are $65536 - 230 = (65306)_{10} = FF1A\ H$

$\therefore$ TL = 1 AH and TH = FF H

>> Program

Label	Instruction	Comments
	MOV TMOD,#10H	Load TMOD register in timer 1, mode 1
L1 :	MOV TL1,#1AH	Load TL
	MOV TH1,#0FFH	Load TH
	SETB TR1	Start timer 1
L2 :	JNB TF1, L2	Remain until timer rolls over
	CLR TR1	Stop timer 1
	CPL P1.5	
	CLR TF1	Clear timer 1 flag
	SJMP L1	Reload timer as mode 1 is not auto reload

Program 6.1.4 : Write a program to generate square wave with 50% duty cycle from P.1.5 (use timer 0) (Total time period = 30.38 μs) (clock frequency 11.0592 MHz)

Solution :

The period of square wave is 30.38 μs. The duty cycle of the square wave is 50%. Hence, the square wave will be high for 15.19 μs. XTAL = 11.0592 MHz i.e. counter will count up every 1.085 μs.

$\therefore \frac{15.19 \ \mu s}{1.085 \ \mu s} = 14$, 1.085 μs will make a 30.38 μs pulse.

The values that should be loaded into TH and TL registers are

$65536 - 14 = (65522)_{10}$ = FFF2 H.

$\therefore$ TL = F2H and TH = FFH

>> Program

Label	Instruction	Comments
	MOV TMOD,#01	Load TMOD register in timer 0, mode 1
L1 :	MOV TL0,#1AH	Load TL0
	MOV TH0,#0FFH	Load TH0
	SETB TR0	Start Timer 0
L2 :	JNB TF0, L2	Remain until timer rolls over
	CLR TR0	Stop Timer 0
	CPL P1.5	
	CLR TF0	Clear timer 0 flag
	SJMP L1	Reload timer as mode 0 is not auto reload

Program 6.1.5 : *Write a program to generate square wave of 500 Hz on P1.0 using Timer 1 of 8051. Oscillator frequency is 12 MHz.*

Solution :

Frequency = 500 Hz

$\therefore$ 1 pulse = $\frac{1}{500}$

= 2 ms

$\therefore$ 1 ms is the ON time and 1 ms is the off time.

Hence a delay of 1 msec is required

$\therefore$ count = $\frac{1 \text{ msec}}{1 \mu\text{sec}}$

= 1000

$\therefore$ count = 65536 – 1000

= $(64536)_{10}$

$\therefore$ count = FC18H

$\therefore$ TL1 = 18 H and TH1 = FCH

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	by pass interrupt service routine
	ORG 001BH	
	CLR TR1	
	CPL P1.0	Complement P1.0 bit
	MOV TL1, #18H	
	MOV TH1, #0FCH	
	SETB TR1	
	RETI	
	ORG 1000H	
main :	MOV TMOD,#10H	Timer 1, mode 1
	MOV TL1,#18H	Load count
	MOV TH1,#0FCH	
	MOV IE,88H	Enable Timer 1 interrupt
	SETB TR1	Start Timer 1
here :	SJMP here	

Program 6.1.6 : *Write an assembly language to generate square wave of 2 KHz at pin P1.1 using 8051. Assume 8051 operating frequency 12 MHz.* **MU - May 16. 10 Marks**

Solution :

Frequency = 2 KHz.

$\therefore$ 1 clock pulse = $\frac{1}{2\text{KHz}}$ = 500 µsec.

$\therefore$ 250 µsec is 'ON' time' and 250 µsec s 'OFF' time

Hence, a delay of 250 µsec required

$\therefore$ count = $\frac{250 \mu\text{sec}}{1 \mu\text{sec}} = 250$

Using mode 2, the timer will have to be initialized to 256 – 250 = 6

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	by pass interrupt service routine
	ORG 000BH	Interrupt vector for Timer 0
	CPL P1.1	Complement P1.1 bit
	RETI	Return from ISR
	ORG 1000H	Start main program after interrupt vector table

Label	Instruction	Comments
main :	MOV TMOD,#02H	Initialize timer 0 is mode 2
	MOV TH0,#06	Load timer count
	MOV TL0,#06	
	MOV IE,#82H	Enable Timer 0 interrupt
	SETB TR0	Start Timer 0
here :	SJMP here	Wait it timer rolls off

Program 6.1.7 : Program timer 1 of 8051 to generate 1 KHz square wave on P2.1 using mode 1. Assume Crystal frequency to 12 MHz.

Solution :

Frequency = 1 KHz

$\therefore$ 1 pulse = $\frac{1}{1\text{ KHz}}$ = 1 msec

$\therefore$ 500 µsec 'ON' time and 500 µsec 'OFF' line.

Crystal frequency = 12 MHz

$\therefore$ 1 clock pulse = 1 µsec.

$\therefore$ Count = $\frac{500\ \mu\text{sec}}{1\ \mu\text{sec}} = 500$

Counter initial value = 65536 – 500 ($\because$ Mode 1)

= $(65036)_{10} = (FE0C)_{16}$

TMOD

Gate	C/$\overline{T}$	M1	M0	Gate	C/$\overline{T}$	M1	M0
0	0	0	0	0	0	0	1

0 1

IE

EA	-	-	ES	ET1	EX1	ET0	EX0
1	0	0	0	0	0	1	0

8 2

>> Program

Label	Instruction	Comment
	org 0000H	
	LJMP main	

Label	Instruction	Comment
	org 000BH	
	CPL P2.1	
	CLR TR0	
	MOV TL0, #0CH	Reinitialize the counter
	MOV TH0, #0FEH	
	SETB TR0	
	RETI	
	org 0100H	
main :	MOV IE, #82 H	Initialize timer 0 Interrupt
	MOV TMOD, #01H	Timer 0 in Mode 1
	MOV TL0, #0CH	Initialize calculated count
	MOV TH0, #0FEH	
	SETB TR0	Start timer 0
here :	SJMP here	

Program 6.1.8 : Write assembly language program to generate a rectangular waveform of frequency 1 KHz and 50% duty cycle at pin P1.7 using 8051. Assume 8051 operating frequency 12 MHz.

MU - Dec. 17. 10 Marks

Solution :

Frequency = 1 KHz

$\therefore$ 1 pulse = $\frac{1}{1\text{ KHz}}$ = 1 msec

$\therefore$ 500 µsec 'ON' time and 500 µsec 'OFF' line.

Crystal frequency = 12 MHz

$\therefore$ 1 clock pulse = 1 µsec.

$\therefore$ Count = $\frac{500\ \mu\text{sec}}{1\ \mu\text{sec}} = 500$

Counter initial value = 65536 – 500 ($\because$ Mode 1)

= $(65036)_{10} = (FE0C)_{16}$

TMOD

Gate	C/$\overline{T}$	M1	M0	Gate	C/$\overline{T}$	M1	M0
0	0	0	0	0	0	0	1

0 1

IE

EA	-	-	ES	ET1	EX1	ET0	EX0
1	0	0	0	0	0	1	0

8 2

>> Program

Label	Instruction	Comment
	org 0000H	
	LJMP main	
	org 000BH	
	CPL P1.7	
	CLR TR0	
	MOV TL0, #0CH	Reinitialize the counter
	MOV TH0, #0FEH	
	SETB TR0	
	RETI	
	org 0100H	
main :	MOV IE, #82 H	Initialize timer 0 Interrupt
	MOV TMOD, #01H	Timer 0 in Mode 1
	MOV TL0, #0CH	Initialize calculated count
	MOV TH0, #0FEH	
	SETB TR0	Start timer 0
here :	SJMP here	

Program 6.1.9 : Assume an oscillator running at 12 MHz controls 8051 microcontroller. Write a program to generate square wave of 2 KHz on P1.0 using Timer 0 in mode 1. **MU-May 17, 10 Marks**

Solution :

The period of square wave is $\frac{1}{2\text{ kHz}} = 500\ \mu s$

Let us assume that duty cycle of square wave is 50%. Hence, the square wave will be high for 250 μs and it will be low for 250 μs.

XTAL = 11.0592 MHz i.e. the counter will count up every 1.085 μs.

$\frac{250\ \mu s}{1.085\ \mu s} = 230$, 1.085 μs intervals will make a 500 μs / 2 KHz pulse.

The values that should be loaded into TH and TL registers are $65536 - 230 = (65306)_{10} = FF1A\ H$

$\therefore$ TL = 1AH and TH = FF H

>> Program

Label	Instruction	Comments
	MOV TMOD,#01H	Load TMOD register in timer 0, mode 1
L1 :	MOV TL1,#1AH	Load TL
	MOV TH1,#0FFH	Load TH
	SETB TR1	Start timer 1
L2 :	JNB TF1, L2	Remain until timer rolls over
	CLR TR1	Stop timer 1
	CPL P1.0	
	CLR TF1	Clear timer 1 flag
	SJMP L1	Reload timer as mode 1 is not auto reload

Program 6.1.10 : Write a assembly language program to generate a rectangular waveform of frequency 1 KHz and 30% duty cycle at pin P1.0 using 8051. Assume 8051 is operating at frequency 12 MHz. **MU - May 15, 10 Marks**

Solution :

Crystal frequency = 12 MHz

$\therefore$ Time for 1 Machine cycle = $\frac{12}{12\text{ MHz}} = 1\ \mu$ sec.

$\therefore$ Square period = 1 m sec. ($\because$ 1 kHz)

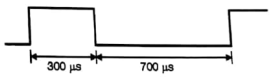


Fig. P. 6.1.10

We will use timer 1 in mode 1 count required

(a) for 'ON' period : $\frac{300\ \mu\text{sec.}}{1\ \mu\text{sec.}} = 300$

$\therefore$ Count = $65536 - 300 = (65236)_{10}$

$= (FED4)_{16}$

(b) for OFF period = $\frac{700\ \mu\text{sec.}}{1\ \mu\text{sec.}} = 700$

$\therefore$ Count = $65536 - 700 = (64836)_{10} = (FD44)_{16}$

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	org 001BH	
	CPL P1.0	Toggle output pin swap the counts as calculated above.
	CLR TR1	
	MOV TL1, R1	
	MOV R3, 01	
	MOV R1, 00	
	MOV TH1, R2	
	MOV 00, R4	
	MOV R4, 02	
	MOV R2, 00	
	SETS TR1	
	CLR TF1	
	RETI	
	org 0100 H	
main :	SETB P1.0	
	MOV IE, #88H	
	MOV TM0D, #10H	
	MOV R1, # 044H	
	MOV R2, # 0FDH	
	MOV R3, # 0D4H	
	MOV R4, # 0FEH	
	MOV TH1, R4	
	MOV TL1, R3	
	SETB TR1	
here :	SJMP here	
	end	

6.1.2 Mode 0 Programming

The programming of timer in mode 0 is same as the programming of mode 1. In mode 0 the timer is a 13 bit timer, so the maximum possible count ranges in values between 0000H to 1FFFH. Hence, when timer reaches to 1FFFH, it rolls over to 0000H and TF is set.

6.1.3 Programming the Timer in Mode 2

Following are the steps to program the timer in mode 2.

Step I : Load the TMOD register.

Step II : Load the TH register with initial count.

Step III : Start the timer.

Step IV : Observe the TF flag. Exit from loop when the TF flag is set.

Step V : Clear the TF flag.

Step VI : Goto step IV, as mode 2 is auto reload.

Program 6.1.11 : Compute the frequency of the square wave generated on P1.5 in the following program.

Label	Instruction
	MOV TMOD, #20H
	MOV TH1, #4H
	SET TR1
L1 :	JNB TF1, L1
BACK :	CPL P1.5
	CLR TF1
	SJMP BACK

Solution :

Since TH1 is 8 bit in mode 2

$\therefore$ 256 – 4 = 252 cycles

$252 \times 1.085\ \mu s = 272.33\ \mu$sec the square wave will remain high and for 272.33 µs the square wave will remain low.

$\therefore$ Period T = $2 \times 272.33 = 544.67\ \mu s$

Frequency = $\frac{1}{T} = 1.8359$ KHz.

Program 6.1.12 : Write an assembly language program for 8051 such that LED connected to port P1.0 will flash at a rate 0.5 sec rate when line P2.3 goes high use timer 0 for generating delay.

Solution : Let XTAL = 12 MHz.

$\therefore$ Timer clock frequency = $\frac{12MHz}{12} = 1$ MHz

$\therefore$ T = 1 µs.

Hence we can get a maximum delay of $65536 \times 1\ \mu s = 65.536$ ms.

To get a delay of 0.5 sec we will program timer 0 to give a delay of 50 ms. We will execute the delay for 10 times so that we will get a delay of 0 .5 sec.

To obtain a delay of 50 ms, the values that should be loaded into TH and TL registers are :

$(65536 - 50000)_{10} = (3CB0)_H$

$\therefore$ TL = B0H and TH = 3CH

>> Program

Delay routine

Label	Instruction	Comments
DELAY :	MOV R0,#0AH	Initialize counter to 10
L1 :	MOV TL0,#0B0	Load TL
	MOV TH0,#03CH	Load TH
	SETB TR0	Start timer 0
L2 :	JNB TF0, L2	Remain until timer rolls over
	CLR TR0	Stop timer 0
	CLR TF0	Clear timer 0 flag
	DJNZ R0, L1	Decrement R0 and if R0 ≠ 0 repeat
	RET	

Main program

Label	Instruction	Comment
	MOV TMOD,#01	Timer 0, Mode 1
	MOV P2,#0FFH	Let P2 = input port
AL1 :	JNB P2.3, AL1	Wait till P2.3 = 0
HERE :	CPL P1.0	Toggle P1.0
	ACALL DELAY	Call Delay
	SJMP HERE	Repeat

Syllabus Topic : Programming Based on Counters

6.2 Counter and Pulse Width Measurement (PWM)

- 8051's Timer/Counter can also be used as a counter when $C/\overline{T}$ = '1'. We will see this feature of 8051 in this section. This feature can also be used to measure the frequency by measuring the number of pulses in 1 second.

- Another feature of 8051's timer/counter is that it can be used to measure the width of a pulse. This can be implemented by programming the timer/counter in timer mode. The pulse to be measured is to be applied on the $\overline{INT0}$ or $\overline{INT1}$ pins for timer 0 and timer1 respectively. The GATE bit of the TMOD register is to be made '0'. The count in the timer registers indicate the number of machine cycles for which the pulse was at logic '1'. This when multiplied with the time period of 1 machine cycle gives the time period for which the pulse was at logic '1'.

Program 6.2.1 : Write an assembly program for counting the pulses on P3.5 pin (T1) and display the hex count on P0 (LSB) P1(MSB).

Solution :

Label	Instruction
	Org 0000H
	LJMP main
	Org 1000H
main :	
	MOV P0,#00H
	MOV P1,#00H
	MOV TMOD,#50H
	MOV TL0,#00H
	MOV TH0,#00H
	SETB P3.5
	SETB TR0
wait :	MOV P0,TL0
	MOV P1,TH1
	SJMP wait

Syllabus Topic : Programming Based on ISR

6.3 Interrupt Service Routines

- Whenever more than one I/O devices are connected to a microprocessor based system, any one of the I/O devices may ask service at any time. There are two methods in which the microprocessor can service these I/O devices. One method is to use the **polling routine**, while the other method employs **interrupt**.
- In the polling routine the microprocessor checks whether any of the I/O devices is requesting for service.
- The polling routine is a simple program that keeps a check for the occurrences of interrupt.
- For e.g. : Let us assume that our polling routine is servicing I/O ports 1, 2, 3,.....8. The polling routine will check the status of the I/O ports in a proper sequence.

- The polling routine will first transfer the status of the I/O port 1 to the accumulator. It then checks the contents of accumulator to determine if the service request bit is set. If the bit is set then I/O port 1 service routine is called, otherwise the polling routine will move forward to check if port 2 is requesting service. On completion of the service to port1, the polling routine will test port2. The process is repeated till all the 8 ports are tested and all the I/O ports those are demanding service are processed. On completion of the polling routine, the microprocessor will resume with the execution of the program.
- The polling routine has priorities assigned to the different I/O devices. Once the routine begins port1 will always be checked first, then port2 and so on.
- Another way that allows the microprocessor stop with the execution of the program and give service to the I/O devices is **interrupt**. It is an external asynchronous input that informs the microprocessor to complete the instruction that it is currently executing and fetch a new routine in order to offer service to the I/O device. Once the I/O device is serviced, the microprocessor will continue with the execution of its normal program.
- Interrupt Service Routines (ISRs) can be written for the peripherals of 8051 like timer/counter, serial port and also for hardware interrupts $\overline{INT0}$ and $\overline{INT1}$. The ISRs are to be written at the specified vector address for each peripheral as given in the Table 6.3.1.

Table 6.3.1

Sr. No.	Interrupt Name	Vector Address
1.	External Intr 0 ($\overline{INT0}$)	0003H
2.	Timer 0 Intr (T0)	000BH
3.	External Intr 1 ($\overline{INT1}$)	0013H
4.	Timer 1 Intr (T1)	001BH
5.	Serial Intr (RI/TI)	0023H

Program 6.3.1 : Write an assembly program to generate a square wave of 10 KHz with timer 0 on Port pin.

Solution :

Frequency = 10 KHz.

$\therefore$ 1 clock pulse $= \frac{1}{10\ KHz} = 100\ \mu sec.$

$\therefore$ 50 µsec is 'ON' time' and 50 µsec is 'OFF' time

Hence, a delay of 50 µsec required

$\therefore$ count $= \frac{50\ \mu sec}{1 \mu sec}$ (Assuming 12 MHz crystal)

$= 50.$

Using mode 2, the timer will have to be initialized to 256 – 50 = 206

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	by pass interrupt service routine
	ORG 000BH	Interrupt vector for Timer 0
	CPL P0.0	Complement P1.0 bit
	RETI	Return from ISR
	ORG 1000H	Start main program after interrupt vector table
main :	MOV TMOD,#02H	Initialize timer 0 is mode 2
	MOV TH0,#206	Load timer count
	MOV TL0,#206	
	MOV IE,#82H	Enable Timer 0 interrupt
	SETB TR0	Start Timer 0
here :	SJMP here	Wait it timer rolls off

Program 6.3.2 : Assume crystal frequency of 12 MHz to 8051 microcontroller, write assembly language program to generate square wave of 1 KHz on port bit P1.0 using timer 1 interrupt.

Solution :

Frequency = 1 KHz

$\therefore$ 1 pulse $= \frac{1}{1\ KHz} = 1$ msec

$\therefore$ 500 µsec is "ON time" and 500 µsec is the "off" time

Hence, a delay of 500 µsec is required.

$\therefore$ count $= \frac{500\ \mu sec}{1\ \mu sec}$ (as crystal frequency = 12 MHz)

$\therefore$ count = 500

$\therefore$ count = 65536 – 500

$= (65036)_{10} = FE0CH$

$\therefore$ TL1 = 0CH and TH1 = FEH

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	By pass interrupt service routine

Label	Instruction	Comments
	ORG 001BH	
	CLR TR1	
	CPL P1.0	Complement P1.0 bit
	MOV TL1, #0CH	
	MOV TH1, #0FEH	
	SETB TR1	
	RETI	
	ORG 1000H	
main :	MOV TMOD,#10H	Initialize timer 1 as mode 1
	MOV TL1,#0CH	Load timer count
	MOV TH1,#0FEH	
	MOV IE,88H	Enable Timer 1 interrupt
	SETB TR1	Start Timer 1
here :	SJMP here	

Program 6.3.3 : Write a program to generate frequencies of 2 KHz and 10 KHz on pins P0.0 and P0.1 respectively. Assume crystal frequency = 12MHz.

Solution :

Timer clock frequency $= \frac{12 \times 10^6}{12} = 1$ MHz

For 2 KHz

On period is 0.25 msec and off period is 0.25 msec.

$$\therefore \quad TH0 = 256 - (0.25 \times 10^{-3} / 1 \times 10^{-6})$$

$$TH0 = (6)_{10} = 6H$$

For 10 KHz

On period is 0.05 msec and off period is 0.05 msec.

$$\therefore \quad TH1 = 256 - (0.05 \times 10^{-3} / 1 \times 10^{-6})$$

$$TH1 = (206)_{10} = CEH$$

Program

Label	Instruction	Comments
	ORG 0000H	Jump over interrupt service routines
	LJMP main	
	ORG 000BH	ISR for Timer 0 interrupt
	CPL P0.0	Complement bit P0.0
	RETI	Return from ISR
	ORG 001BH	ISR for Timer 1 interrupt complement bit return from ISR
	CPL P0.1	
	RETI	
main :	ORG 0030H	
	MOV TMOD,#22H	Initialize both timers in mode 2
	MOV IE,#8AH	Enable Timer 0 and Timer 1 interrupts
	MOV TH0,#06H	Count value for 2 KHz wave
	MOV TH0,#CEH	Count value for 10 KHz wave
	SETB TR0	Start Timer 0
	SETB TR1	Start Timer 1
here :	SJMP here	Wait till either timer rolls off
	END	

Program 6.3.4 : Write an assembly program to switch 'on' or 'off ' a LED connected on P1.3 when external interrupt INT0 is activated.

Solution :

Label	Instruction	Comments
	ORG 0000H	Bypass interrupt service routine
	LJMP MAIN	
	ORG 0003H	Interrupt vector for Interrupt 0
	SETB P1.3	Turn on LED
	MOV R0,#200	Wait for sometime

Label	Instruction	Comments
L1 :	DJNZ R0, L1	
	RETI	Return to main program
	ORG 0030H	
MAIN :	MOV IE,#81H	Enable INT0
L2:	SJMP L2	
	END	End program

6.4 Design Problems

Design 6.4.1 : Design a 8051 based system to generate a square wave of 10 KHz and write the corresponding program in assembly.

Solution :

Part (a) **Design** : Let us generate this square wave on the P1.5 pin. The circuit interface for this system is shown in the Fig. P. 6.4.1.

Part (b) **Program :** Let us use timer 1 for this application. We need a square wave of 10 KHz.

$\therefore$ time period of 1 pulse $= \frac{1}{f} = \frac{1}{10\ kHz} = 100\ \mu sec.$

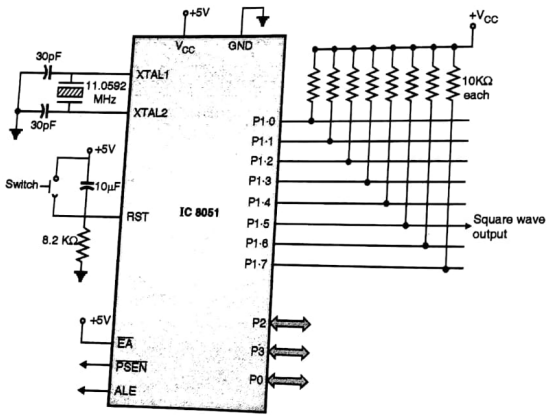


Fig. P. 6.4.1 : Interface diagram

The output should be logic '1' for 50 μ sec and should be logic '0' for 50μ sec. Hence we need a delay of 50 μ sec after which we will toggle the pin and repeat this every 50 μ sec.

For a delay of 50 μ sec, we need to calculate the number of machine cycles required. Since we have connected a crystal of 11.0592 MHz, each

$$\text{clock pulse period} = \frac{1}{11.0592\ MHz} = 90.42\ nsec$$

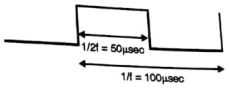


Fig. P. 6.4.1(a) : Waveform to be generated

$\therefore$ 1 machine cycle time, that comprises of 12 clock pulse = 90.42 nsec × 12 = 1.085 μsec

$\therefore$ For a delay of 50 μ sec the number of machine cycles to be counted $= \frac{50\ \mu\ sec}{1.085\ \mu\ sec} = 46.082 \cong 46$

Since we need a continuous waveform, we will program the timer in Mode 2 which is Auto reload mode. The counter should be initialized to 256 (FFH) – 46 = $(210)_{10} = (D2)_{16}$

Note : We initialize it to $(D2)_{16}$, because it is an up counter. It starts counting from $(D2)_{16}$ till $(FF)_{16}$ and then to $(00)_{16}$. This causes an overflow and TF1 to be set to '1'. Hence generates an interrupt. Thus it counts $(FF)_{16} - (D2)_{16} = (46)_{10}$ machine cycles before generating the interrupt.

➢➢ Algorithm

(A) Main Program

Step I : Set Port1.5 pin to logic '1'.

Step II : Enable the timer 1 interrupt and global interrupt in the Interrupt Enable register.

Step III : Initialize TMOD register to be timer mode 2 for the timer 1

Step IV : Load the count calculated above i.e. $(D2)_{16}$ in TL1 and TH1 registers.

Step V : Set TR1 bit to run timer 1.

Step VI : Do nothing i.e. interrupt will take care

(B) Timer 1 ISR

Step I : Toggle P1.5 pin

Step II : Clear Timer 1 overflow flag i.e. TF1

➢➢ Registers value

(1) Interrupt Enable (IE) Register

→ To enable global interrupt and Timer 1 interrupt.

D7	D6	D5	D4	D3	D2	D1	D0
EA	-	-	ES	ET1	EX1	ET0	EX0
1	0	0	0	1	0	0	0
			8			8	

$\therefore$ IE = 0x88

(2) Timer Mode (TMOD) Register

→ To initialze Timer / Counter 1 as timer

→ To intitialize Timer 1 in mode 2

TIMER 1 TIMER 0

D7	D6	D5	D4	D3	D2	D1	D0
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
0	0	1	0	0	0	0	0
		2			0		

$\therefore$ TMOD =0x20

(3) The calculated count of 0xD2 should be initialized in TL1 and TH1

$\therefore$ TL1 = 0xD2

and TH1 = 0xD2

Note : The count is loaded into TH1 also, because it works as reload register.

>> Assembly Program

Label	Instruction	Comments
	org 0000H	
	LJMP Start	
	org 001BH	ISR for timer 1
	CLR TF1	Clear Timer 1 overflow flag
	CPL P1.5	Toggle P1.5 pin
	RETI	
	org 1000H	
Start:	SETB P1.5	Set P1.5 pin to logic '1'
	MOV IE,#88H	Enable Timer 1 and Global interrupts
	MOV TMOD,#20H	Timer 1 programmed as timer and in mode 2
	MOV TH1, #0D2H	Load the reload register as D2H
	MOV TL1, #0D2H	Load the counter register as D2H
	SETB TR1	Set Timer 1 in Run mode
here:	SJMP here	Do Nothing loop. Wait for interrupt.
	End	

>> Output

The output of the above program is as shown in Fig. P. 6.4.1(b).

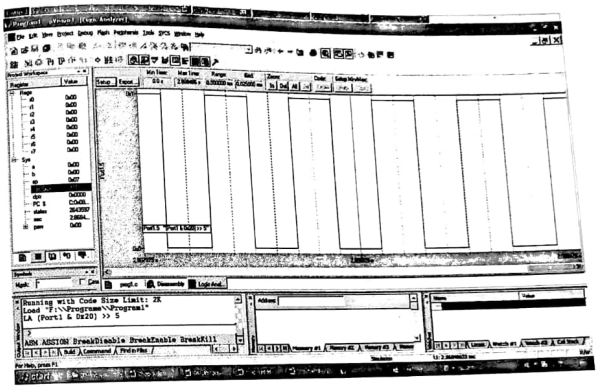


Fig. P. 6.4.1(b) : Output

6.5 Solved Examples

Example 6.5.1 : If the crystal frequency of 8051 is 12 MHz, write its assembly language program to do the following.

(a) generate a delay of 2 ms

Solution :

(a) Generate a delay of 2ms.

Crystal frequency = 12 MHz

$$\therefore \quad T = \frac{12}{12 \times 10^6} s = 1\ \mu s$$

Let us determine the count to get a delay of 1 ms

$\therefore \frac{1\ ms}{1\ \mu s} = 1000$ clocks

$\therefore$ Count = 65536 – 1000

$\therefore$ Count = 64536

$\therefore$ Count = FC18 H

To get a delay of 2 msec we have to repeat the delay of timer 0, 2 times.

>> Program

Label	Instruction	Comments
	MOV TMOD,#01H	Timer 1, mode 0
	MOV R0,#2	Count for running delay 2 times
BACK :	MOV TL0,18H	Load count in TL0
	MOV TH0,FCH	Load count in TH0
	SETB TR0	Start Timer 0
AGAIN :	JNB TF0, AGAIN	Stay until timer rolls over
	CLR TR0	Stop Timer 0
	CLR TF0	Clear Timer flag
	DJNZ R0, BACK	If R0 ≠ 0, reload timer
	END	

Example 6.5.2 : Write an assembly language program for 8051 to generate a delay of 1msec using 12MHz crystal.

Solution :

Crystal frequency = 12 MHz.

$\therefore$ Time for 1 machine cycle = $\frac{12}{12\ MHz} = 1\ \mu sec.$

$\therefore$ Delay period = 1 m sec.

Hence a delay of 1 msec is required.

$\therefore$ We will require mode 2; and execute a delay of (lets say) 250 µsec four times.

Suppose we use timer 1, TMOD = 20 H

TH1 = TL1 = $(256)_{10} - (250)_{10} = (6)_{10} = (6)_{16}$

The ISR should just toggle a port pin of 8051.

Label	Instruction
Delay :	MOV TMOD, #20H
	MOV TH1, #06H
	MOV TL1, #06H
	SETB TR1
here :	JNB TF1, here
	RETI

Example 6.5.3 : Write a following programs (Use 8051 µc)

(i) Create a square wave of 50% duty cycle on bit 0 of port 1.

(ii) Create a square wave of 66% duty cycle on bit 3 of port 1.

Solution :

(i) Create a square wave of 50% duty cycle on bit 0 of port 1.

50% duty cycle indicates that the on time and off time is same. Hence, we will toggle bit P1.0 with time delay in each state.

Label	Instruction	Comments
L1 :	SETB P1.0	Make P1.0 = 1
	ACALL DELAY	Wait for sometime
	CLR P1.0	Make P1.0 = 0
	ACALL DELAY	Wait for sometime
	SJMP L1	Keep doing it.

(ii) Create square wave of 66% duty cycle on bit 3 of Port 1.

Let crystal frequency = 12 MHz.

$\therefore$ Time for 1 machine cycle = $\frac{12}{12\ MHz} = 1\ \mu sec.$

Let square wave period = 1 msec

Hence the delay required.

(i) For "ON" period

$\frac{660\ \mu sec}{1\ \mu sec} = 660$

$\therefore$ count = 65536 – 660 = $(64876)_{10}$

$\therefore$ count = FD6C H

(ii) For "OFF" period

$\frac{340\ \mu sec}{1\ \mu sec} = 340$

$\therefore$ count = 65536 – 340 = $(65196)_{10}$

$\therefore$ count = FEAC H

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	
	ORG 001BH	
	CPL P1.3	Toggle output pin
	CLR TR1	
	MOV TL1, R1	Swap the contents as calculated above
	MOV 00,R3	
	MOV R3,01	
	MOV R1,00	
	MOV TH1,R2	
	MOV 00,R4	
	MOV R2,00	
	SETB TR1	
	CLR TF1	
	RETI	
Main :	ORG 0100H	
	SETB P1.3	
	MOV IE,#88H	
	MOV TMOD,#10H	
	MOV R1,#ACH	
	MOV R2,#FEH	
	MOV R3,#6CH	
	MOV R4,#FDH	
	MOV TH1,R4	
	MOV TL1,R3	
	SETB TR1	
Here:	SJMP here	
	End	

Example 6.5.4 : Write a program to generate a square wave of frequency 1KHz and 75% duty cycle at pin P1.0 using 8051 microcontroller. Assume microcontroller is operating at 6MHz.

Solution :

Crystal frequency = 6 MHz.

$\therefore$ Time for 1 machine cycle $= \frac{12}{6\ MHz} = 0.5\ \mu sec.$

$\therefore$ Square wave period = 1 msec. ($\because$ 1 KHz)

Fig. P. 6.5.4

Hence the delay required (Using timer 1 in Mode 1)

(i) For 'ON' period

$$\frac{750\ \mu sec}{0.5\ \mu sec} = 1500$$

$\therefore$ Count = 65536 – 1500

$= (64036)_{10} = (FA24)_{16}$

(ii) For 'OFF' period

$$\frac{250\ \mu sec}{0.5\ \mu sec} = 500$$

$\therefore$ Count = 65536 – 500

$= (65036)_{10} = (FE0C)_{16}$

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	Org 001BH	
	CPL P1.5	Toggle the output pin
	CLR TR1	
	MOV TL1, R1	Swap the counts as calculated above
	MOV 00, R3	
	MOV R3, 01	
	MOV R1, 00	

Label	Instruction	Comments
	MOV TH1, R2	
	MOV 00, R4	
	MOV R4, 02	
	MOV R2, 00	
	SETB TR1	
	CLR TF1	
	RETI	
	org 0100 H	
main :	SETB P1.5	
	MOV IE, #88H	
	MOV TMOD, #10H	
	MOV R1, #0CH	
	MOV R2, #0FEH	
	MOV R3, #24H	
	MOV R4, #0FAH	
	MOV TH1, R4	
	MOV TL1, R3	
	SETB TR1	
here :	SJMP here	
	end	

Example 6.5.5 : Design 8031 based on intrusion warning system using interrupts that sounds a 400 Hz tone for 1 second (using a loudspeaker connected to P1.7) whenever a door sensor connected to $\overline{INT0}$ makes a high to low transition.

Solution :

Label	Instruction	Comment
	LS BIT P1.7	Loudspeaker
	ORG 0000H	
	LJMP MAIN	
	ORG 0003H	$\overline{INT0}$
	SETB P3.2	High to low transition
	CLR P3.2	
	SETB P1.7	Turn loudspeaker on for 1s
	RETI	

Example 6.5.6 : Assuming crystal frequency = 11.0592 MHz, write as Assembly Language Program for 8051 to generate square wave of 50 Hz at port pin P2.3. Draw circuit diagram to implement the same. MU - Dec. 15 10 Marks

Solution :

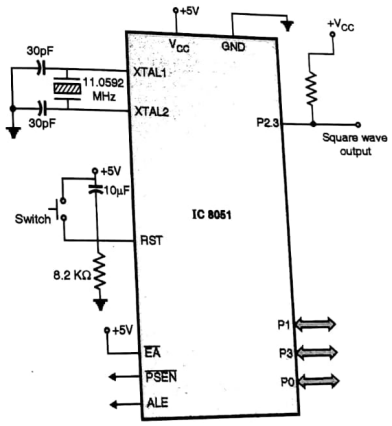


Fig. P. 6.5.6

Crystal frequency = 11.0592 MHz

Time for 1 machine cycle = $\frac{12}{11.0192 \text{ MHz}} = 1.085 \mu \text{ sec}$

Square period = $\frac{1}{50 \text{ Hz}} = 0.02 \text{ sec} = 20 \mu \text{ sec}$

$\therefore$ On period = OFF period = 10 μ sec

We will use timer 1 in mode 1.

Count required = $\frac{10 \mu \text{ sec}}{1.085 \mu \text{ sec}} = 9216.6 \cong 9217$

$\therefore$ Count = 65536 – 9217 = $(56319)_{10}$

= $(DBFF)_{16}$

Label	Instruction	Comment
	org 0000H	
	LJMP main	
	org 001BH	
	CPL P2.3	Toggle output pin
	CLR TR1	
	MOV TL1, R1	Reload the count
	MOV TH1, R2	
	SETB TR1	
	CLR TF1	
	RETI	
	org 0100 H	
main :	SETB P2.3	
	MOV IE, #88H	
	MOV TMOD, #10H	
	MOV R1, #0FFH	
	MOV R2, #0DBH	
	MOV TL1, R1	
	MOV TH1, R2	
	SETB TR1	
here :	SJMP here	
	end	

6.6 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : Programming Based on I/O Parallel and Serial Ports, Timer**

Q. 1 Write an assembly language to generate square wave of 2 KHz at pin P1.I using 8051. Assume 8051 operating frequency 12 MHz. **(Refer Program 6.1.6)** **(M-16, 10 Marks)**

Q. 2 Write assembly language program to generate a rectangular waveform of frequency 1 KHz and 50% duty cycle at pin P1.7 using 8051. Assume 8051 operating frequency 12 MHz. **(Refer Program 6.1.8)** **(D-17, 10 Marks)**

Q. 3 Assume an oscillator running at 12 MHz controls 8051 microcontroller. Write a program to generate square wave of 2 KHz on P1.0 using Timer 0 in mode 1. **(Refer Program 6.1.9)** **(M-17, 10 Marks)**

Q. 4 Write a assembly language program to generate a rectangular waveform of frequency 1 KHz and 30% duty cycle at pin P1.0 using 8051. Assume 8051 is operating at frequency 12 MHz. **(Refer Program 6.1.10)** **(M-15, 10 Marks)**

Q. 5 Write a program using interrupts to get data serially and send it to P2 while at the same time Timer 0 is generating square wave of 500 Hz. **(Refer Program 6.1.5)** **(10 Marks)**

☞ **Syllabus Topic : Programming Based on Counters**

Q. 6 Write short note on counter. **(Refer Section 6.2)** **(2 Marks)**

☞ **Syllabus Topic : Programming Based on ISR**

Q. 7 Assuming crystal frequency = 11.0592 MHz, write as Assembly Language Program for 8051 to generate square wave of 50 Hz at port pin P2.3. Draw circuit diagram to implement the same. **(Refer Example 6.5.6)** **(D-15, 10 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 7

Asynchronous Serial Data Communication

7.1 Introduction

Q. 7.1.1 Distinguish between serial and parallel data transfer. (Ref. Sec. 7.1) **(5 Marks)**

The word, *communication* specifies, data transfer between two points.

Sr. No.	Parallel data transfer	Serial data transfer
1.	8 bits of data is transferred at a time.	One bit of data is transferred at a time.
2.	9 lines required to be connected between 2 points.	Only 2 lines required to be connected.
3.	Data transfer is fast.	Data transfer is slow.
4.	More advantageous over small distances only.	More advantageous over long distances.

- In these two methods the cost of connecting two distant points, is the main factor. So though the parallel data transfer is faster, it is preferred for small distances only. But for long distances, serial data transfer is preferred.
- In serial data transfer the 8 bits of data is converted into serial 8 bits; using shift register (parallel in serial out mode). These serial bits are transferred on single line using serial I/O data transfer. To transfer 8 bits of data, it will require 8 clock pulses. On the other side exactly opposite process is done. These serial 8 bits are accepted and converted to parallel form to get 8 bits of data. This process is shown in Fig. 7.1.1.

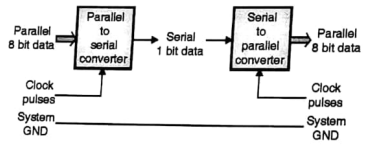


Fig. 7.1.1 : Serial I/O

7.2 Types of Communication Systems

Q. 7.2.1 Explain data communication formats in serial communication. (Ref. Sec. 7.2) **(5 Marks)**

The communication systems are classified on the basis of transmission :

(1) Simplex (2) Duplex

(1) Simplex

- The simplex is one way transmission.
- The connection exists such that data transfer takes place only in one direction.
- There is no possibility of data transfer in the other direction.
- System A is transmitter and system B is receiver only.

(2) Duplex

The duplex is two way transmission. It is further divided in 2 groups :

(a) Half duplex (b) Full duplex.

(a) Half duplex

- It is a connection between two terminals such that, data may travel in both the directions, but transmission activated in one direction at a time.
- This indicates that the line has to turn around after communication is complete in one direction.

(b) Full duplex

It is a connection between two terminals such that, data may travel in both the directions simultaneously. So it will contain one way transmission or two way transmission at a time.

7.3 Serial Transmission Formats

Q. 7.3.1 Differentiate between Synchronous and Asynchronous data transfer. (Ref. Sec. 7.3) **(5 Marks)**

In serial communication, two formats of data transfer are used. These two formats are :

(1) Asynchronous (2) Synchronous.

7.3.1 Asynchronous Data Transfer

- It is a '*character*' oriented data transfer. In this one data byte is transferred serially at a time. When data is not transferred output line stays HIGH.
- The start of data is indicated by a low start bit. The start bit is used to synchronise transmitter and receiver. After the start bit, data bits are transferred serially $D_0, D_1 \ldots D_7$ (least significant bit first).
- The data bits are followed by a one or more high stop bits. After the stop bits same logical level is maintained on output line i.e. marking state for next data byte.
- At the receiver end the receiver will check for marking state. When it detects a low on data line for 1 bit time it is treated as valid start bit.
- After start bit it will start accepting data bits D_0 to D_7. At the end it will check stop bits and go back to initial marking state. This process is repeated for next data byte.

- The asynchronous data format consists of a start bit, data bits and stop bits (also called as frame) is as shown in Fig. 7.3.1.
- The clock to both transmitter and receiver are separate. But the operating frequency for both is maintained same.
- The bit period (baud rate) is pre-decided to maintain synchronization between the transmitter and receiver.

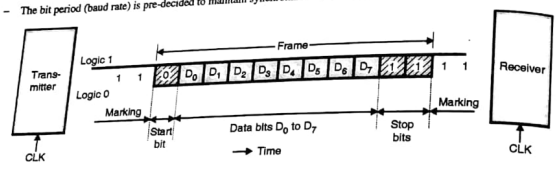


Fig. 7.3.1 : Asynchronous data format

The Table 7.3.1 gives the list of asynchronous data format standards.

Table 7.3.1

Specification	Typical values
Data bits/character	5, 6, 7 or 8
Stop bits	1, 1½ or 2
Parity bit	Odd or even parity
Baud rates	75, 100, 150, 300, 600, 1200, 2400, 4800, 9600, 19200.

7.3.2 Synchronous Data Transfer

- It is a ***'block or packet of data'*** oriented data transfer. The block(s) of data bytes are transferred serially.
- Packet of data is of size *'n'*. The data bytes in a block are transferred one after the other. As shown in Fig. 7.3.2. D_0 to D_7 data bits of $data_1$, $data_2$, ... $data_n$ is transferred serially.
- Before sending data, special characters are transferred by transmitter to achieve synchronization between transmitter and receiver.

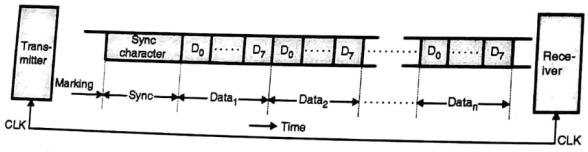


Fig. 7.3.2 : Synchronous data format

- This special character is called as ***Sync character***.
- Normally the output line of transmitter is HIGH i.e. marking state.
- To start transmission, the sync character bits are sent by transmitter followed by data bits.
- At receiver end, the receiver will go on checking input line.
- The checking operation is comparing ***previous 8 bits*** of data received with receivers sync character. When match occurs i.e. both the bit patterns are exactly equal then it is considered that the receiver is in synchronization with transmitter.
- When the receiver is synchronized, it will go on accepting data bits in same sequence i.e. D_0 to D_7 and so on.
- The term, ***previous 8 bits*** is used for sync character.
- It specifies previously accepted 7 bits and present bit will be 8th bit.
- The sync character may be, any digital bit pattern which is used for transmitter and receiver. Refer Fig. 7.3.2.
- The clock to both transmitter and receiver is same. If there is little difference between clock synchronization the synchronous method will accept wrong data bits so the same clock is used.

7.3.3 Comparison of Asynchronous and Synchronous Format

Sr. No.	Asynchronous data transfer	Synchronous data transfer
1.	It is used to transfer one character at a time.	It is used to transfer a block of characters at a time.
2.	Used for data transfer rates ≤ 20 k bits / second.	Used for high data transfer rates ≥ 20 k bits / second.
3.	Sync characters are not transmitted along with characters.	Sync character are transmitted along with the group of characters.
4.	Start bit and stop bit for each character is present which forms a frame.	No start and stop bit is used.
5.	Two separate clock inputs can be used for transmitter and receiver.	One clock is used for both transmitter and receiver.
6.	No synchronization is required hence hardware and software implementation is possible.	Since synchronization is involved, this can be implemented by using hardware only.
7.	Speed is less, because overheads are more per byte (start bit, stop bit(s) etc.)	Speed is high because overheads are spread over a frame or packet of data bytes.

7.3.4 Baud Rate

In serial communication the rate at which data bits are transmitted generates a term Baud rate. The Baud rate is defined as bits / second or the changes in voltage levels / second.

$$\text{Baud} = \frac{\text{Bits transmitted}}{\text{Second}}$$

Typical Baud rates are 110, 300, 600, 1200, 2400, 4800 and 9600.

A teletype typically uses 110 Baud.

$$\text{Transmission rate} = 110 \text{ Baud} = 110 \text{ bits / sec.}$$

$$\text{Time for one bit} = \frac{1}{110} = 9.1 \text{ ms.}$$

7.4 RS 232 Standard

Q. 7.4.1 How is data transmitted and received using RS232 ? (Ref. Sec. 7.4) **(5 Marks)**

- In the early 1960s, a standards committee, today known as the Electronic Industries Association (EIA), developed a common interface standard for data communications equipment.
- At that time, data communications was thought to mean digital data exchange between a centrally located mainframe computer and a remote computer terminal, or possibly between two terminals without a computer involved.
- These devices were linked by telephone voice lines, and consequently required a modem at each end for signal translation.
- While simple in concept, the many opportunities for data error that occur when transmitting data through an analog channel require a relatively complex design.
- It was thought that a standard was needed first to ensure reliable communication and second to enable the interconnection of equipment produced by different manufacturers, thereby fostering the benefits of mass production and competition. From these ideas, the RS 232 standard was born. It specified signal voltages, signal timing, signal function, a protocol for information exchange and mechanical connectors.

7.4.1 Signals used in RS 232

Table 7.4.1 : Pin Description of RS-232 DB-9 Connector

Pin	Description
1	Data carrier detect ($\overline{DCD}$)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready ($\overline{DTR}$)
5	Signal ground (GND)
6	Data set ready ($\overline{DSR}$)
7	Request to send ($\overline{RTS}$)
8	Clear to send ($\overline{CTS}$)
9	Ring indicator (RI)

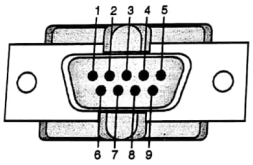


Fig. 7.4.1 : RS-232 DB-9 Connector

- It is a bus standard used for transmitting and receiving Serial Data.
- The format specifies handshake as well as communication signals, and gives hardware specifications.

Hardware specifications

(1) Voltage levels +3 V ... +25 V for logic 1 and –3 V ... –25 V for logic 0.

(2) Transmission line should be a Twisted pair wire with capacitance between 300 ... 2500 pF.

(3) Max. Length of the line = 50 ft.

(4) The circuit should be single end, bi-polar, voltage un-terminated.

Signal specifications

The Data Terminal Equipment (DTE) wants to send data through the Data Communication Equipment (DCE), to a Remote MODEM. The signals used for this purpose are as follows :

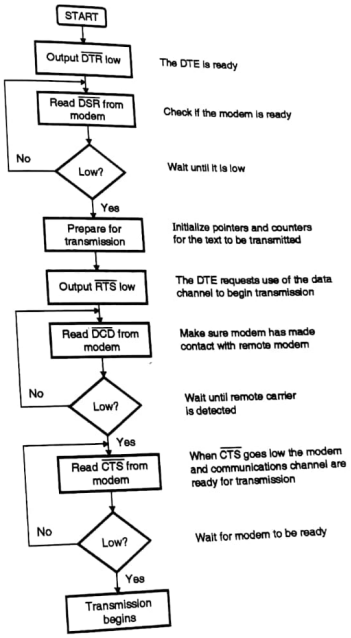


Fig. 7.4.2 : Flowchart for RS-232 connection

(1) $\overline{\text{DTR}}$ (Data Terminal Ready)

- The DTE alerts the DCE for data communication by sending this signal.
- In response to this signal, the DCE prepares itself for the communication.

(2) $\overline{\text{DSR}}$ (Data Set Ready)

The DCE sends this signal to the DTE to inform the DTE that it is ready for communication.

(3) $\overline{\text{RTS}}$ (Request To Send)

- Once DTE receives the DSR, it prepares itself for the data transfer.
- It initializes the Memory pointer and the Byte Counters.
- It then sends the $\overline{\text{RTS}}$ signal to the DCE, to inform that it is ready for the transfer.

(4) $\overline{\text{DCD}}$ (Data Carrier Detect)

- In response to the $\overline{\text{RTS}}$, DCE tries to make contact with the Remote MODEM.
- If the Carrier is detected it informs the DTE about it through the $\overline{\text{DCD}}$ signal.

(5) $\overline{\text{CTS}}$ (Clear To Send)

- Once a stable contact is established with the Remote MODEM, the DCE sends the $\overline{\text{CTS}}$ to DTE.
- Due to this, the transmission begins.

7.5 8051 Connection to RS 232

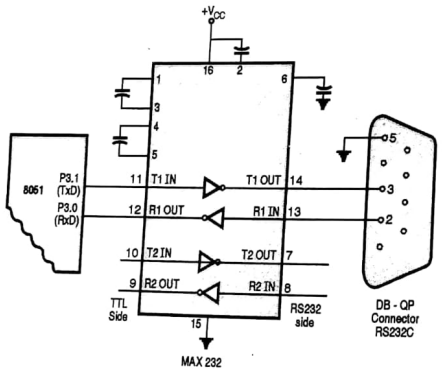
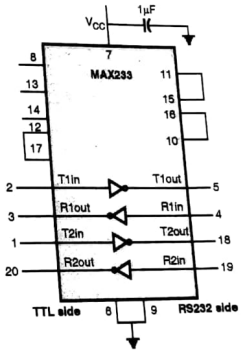
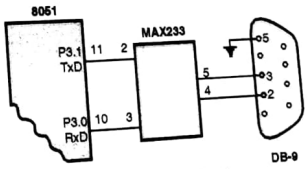


Fig. 7.5.1 : 8051 connection to RS 232

- We know that the RS 232 is not compatible with the TTL logic levels and hence line drivers and receivers are used to interface RS 232 and the TTL devices. The MAX 232 converts the voltage levels from RS 232 to TTL voltage levels and vice versa.
- Fig. 7.5.1 shows the connection of 8051 to the RS 232.
- The 8051 has two pins RxD (P3.0) and TxD (P3.1) for the reception and the transmission of serial data.
- The pins P3.0 and P3.1 are TTL compatible and require line driver and receiver to make them TTL compatible.
- The MAX 232 has two sets of line drivers and receivers for transmitting and receiving data. The line drivers that are used for transmitting serial data are T1 and T2 while R1 and R2 are used for line receivers.
- Also MAX 232 requires four capacitors that range from 1 to 22 μF.
- MAX233 performs the same job as MAX232 but eliminates the need for capacitors. Notice that MAX233 and MAX232 are not pin compatible. The standard interface diagram for MAX233 and its connection with 8051 is shown in Fig. 7.5.2(a) and 7.5.2(b) respectively.



(a) Interface diagram of MAX233



(b) Its connection with 8051.

Fig. 7.5.2

7.6 Serial Communication Programming

7.6.1 Programming the 8051 to Transfer Data Serially

➔ (MU - May 16, Dec. 17)

Q. 7.6.1 8051 microcontroller with XTAL frequency = 11.0592 MHz, find the TH1 value needed to have the following baud rates (i) 9600 (ii) 2400 (iii) 1200. (Ref. Sec. 7.6.1) **May 16. 5 Marks**

Q. 7.6.2 8051 microcontroller with XTAL frequency = 11.0592 MHz. Find the TH value needed to have the following baud rates of 9600. (Ref. Sec. 7.6.1) **Dec. 17. 5 Marks**

Inorder to transfer data bytes serially, the following steps must be considered.

Step I : The TMOD register is loaded with the value 20 H. This will indicate the use of timer 1 in mode 2 to set the baud rate.

Assume that XTAL = 11.0592 MHz. Now we have to set the baud rate to (a) 9600 (b) 4800 (c) 2400 (d) 1200. For setting the baud rate we need to get the value that is to be loaded into the TH 1.

The machine cycle frequency of 8051 = $\frac{11.0592}{12}$ MHz = 921.6 KHz

The frequency provided by UART = $\frac{921.6}{32}$ KHz = 28,800 Hz.

(a) To get baud rate 9600, $\because \frac{28,800}{3} = 9600$

where TH1 = FFH – 3 = FD H.

(b) To get baud rate 4800, $\because \frac{28,800}{6} = 4800$

where TH1 = FFH – 6 (decimal) = FA H.

(c) To get baud rate 2400, $\because \frac{28,800}{12} = 2400$

where TH1 = FFH – 12 (decimal) = F4 H

(d) To get baud rate 1200, $\because \frac{28,000}{24} = 1200$

where TH1 = FFH – 24 (decimal) = E8 H

Summarizing

Baud Rate	TH 1 (Decimal)	TH 1 (Hex.)
9600	– 3	FD
4800	– 6	FA
2400	– 12	F4
1200	– 24	E8

Step II : Load TH1 with one of the values given is above table to set the baud rate for serial data transfer. (These values of baud rates are assuming XTAL = 11.0592 MHz.)

Step III : Load the SCON register with 50 H. This indicates serial mode 1.

Step IV : TR1 is set to 1 to start timer 1.

Step V : Clear TI bit.

Step VI : The character byte that is to be serially transmitted is written into the SBUF register.

Step VII : To check whether the character byte has been completely transmitted observe the TI bit.

Step VIII : To transfer next character byte, go to step V.

The Table 7.6.1 shows the commonly used baud rates and the way in which they can be obtained from Timer 1, the oscillator frequency is specified.

Table 7.6.1

Baud Rate (bits per second)	f_{osc}	SMOD	Timer		
			$C/\overline{T}$	Mode	Reload value
Mode 0 maximum = 1 Mbps	12 MHz	×	×	×	×
Mode 1,3 maximum = 62.5 Kbps	12 MHz	1	0	2	FF H
Mode 2 maximum = 375 Kbps	12 MHz	1	×	×	×
19.2 Kbps	11.059 MHz	1	0	2	FD H
9.6 Kbps	11.059 MHz	0	0	2	FD H
4.8 Kbps	11.059 MHz	0	0	2	FA H
2.4 Kbps	11.059 MHz	0	0	2	F4 H
1.2 Kbps	11.059 MHz	0	0	2	E8H
137.5 bps	11.986 MHz	0	0	2	1DH
110 bps	6 MHz	0	0	2	72H
110 bps	12 MHz	0	0	1	FEEBH

Program 7.6.1 : Write a program for 8051 to transfer letter "B" serially at 1200 baud, continuously.

Solution :

Label	Instruction	Comments
	MOV TMOD,#20H	Timer 1, mode 2
	MOV TH1,#E8H	1200 baud rate
	MOV SCON,#50H	8 bit, 1 stop, REN enabled
	SETB TR1	Start timer 1
L2 :	MOV SBUF,#"B"	Letter B to be transferred
L1 :	JNB TI, L1	Wait for last bit
	CLR TI	Clear TI for next character
	SJMP L2	Keep sending B

Program 7.6.2 : Write a program for 8051 to transfer the message "HI" serially at 4800 baud, 8 bit data, 1 stop bit, continuously.

Solution :

Label	Instruction	Comments
	MOV TMOD,#20H	timer 1, mode 2
	MOV TH1,#FAH	4800 baud rate
	MOV SCON,#50H	8 bit, 1 stop, REN enabled.
	SETB TR1	start timer 1
L1 :	MOV A,#"H"	transfer "H"
L3:	JNB TI, L3	wait for transmission to complete
	CLR TI	
	MOV A,#"I"	transfer "I"
L4:	JNB TI, L4	wait for transmission to complete
	CLR TI	
	SJMP L1	repeat the transmission
	RET	

7.6.2 Programming the 8051 to Receive Data Serially

Q. 7.6.3 Give steps for programming 8051 for serial data transfer. (Ref. Sec. 7.6.2) **(5 Marks)**

Inorder to receive the data bytes serially, the following steps must be considered

Step I : The TMOD register is loaded with the value 20 H. This indicates the use of timer 1 in mode 2 to set the baud rate.

Step II : Load TH 1 with value to set the baud rate for serial data transfer.

Step III : Load the SCON register with 50 H. This indicates serial mode 1.

Step IV : TR1 is set to 1 to start timer 1.

Step V : Clear RI bit.

Step VI : To check whether the entire character has been received, the RI bit is observed.

Step VII : If RI is set, then the byte is received in SBUF register. Save this character byte.

Step VIII : To receive next character, go to step V.

Program 7.6.3 : Write an 8051 assembly language program to receive bytes serially with baud rate of 2400, 8 bit data and 1 stop bit. Simultaneously send the received character bytes to port 2.

Solution :

Label	Instruction	Comments
	MOV TMOD,#20H	timer 1 mode 2.
	MOV TH 1,#F4H	2400 baud rate
	MOV SCON,#50H	8 bit, 1 stop bit, REN enabled.
	SETB TR1	start timer 1
L1 :	JNB RI, L1	save the received character
	MOV P2, A	send character to port 2
	CLR RI	Get ready to receive next character
	SJMP L1	Goto receive next character.

Program 7.6.4 : Write an assembly program to take data from ports 0 and 1, one after the other and transfer data serially continuously.

Solution : Goto receive next character.

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	org 1000H	
main :	MOV TMOD,#20H	Timer 1 in Mode 2 for Baud rate
	MOV TH1,#0FDH	Timer 1 count for 9600 baud/sec
	MOV SCON,#50H	Mode 1 for serial communication
	MOV P0,#0FFH	P0 as input port
	MOV P1,#0FFH	P1 as input port
	SETB TR1	Run timer 1
here :	MOV SBUF, P0	Send data from P0 to serial port
WAIT :	JNB TI, WAIT	Wait for transmission to complete
	CLR TI	
	MOV SBUF, P1	Send data from P1 to serial port
WAIT1 :	JNB TI, WAIT1	
	CLR TI	
	SJMP here	

Program 7.6.5 Write an assembly program to receive the data in serial form and send it out to port 2 in parallel form continuously.

Solution :

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	org 1000H	
main:	MOV TMOD,#20H	Timer 1 in Mode 2 for Baud rate
	MOV TH1,#0FDH	Timer 1 count for 9600 baud/sec
	MOV SCON,#50H	Mode 1 for serial communication
	SETB TR1	Run timer 1
here:	JNB RI, here	
	MOV P2, SBUF	
	CLR RI	
	SJMP here	

Program 7.6.6 : Port 0 of 8051 is used to monitor certain parameters in a system. If the parameters reading on port 0 is 5AH, the system is working fine. Otherwise there is some problem. Write an assembly program to send information "OK" or "DANGER" according to the parameters on P0.

Solution :

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	org 0100H	
	dB "Danger"	
	org 0110H	
	dB "OK"	
	org 1000H	
main :	MOV P0,#0FFH	P0 mode as input port

Label	Instruction	Comments
	MOV TMOD,#20H	Timer 1 in Mode 2 for Baud rate
	MOV TH1,#0FDH	Timer 1 count for 9600 baud/sec
	MOV SCON,#50H	Mode 1 for serial communication
	SETB TR1	Run timer
repeat :	MOV A, P0	
	MOV R1,#02H	
	MOV DPTR,#0110H	
	SJMP over	
danger :	MOV R1,#06H	
	MOV DPTR,#0100H	
over :	ACALL send-data	
	SJMP repeat	
	org 2000H	
send-data :	MOV A,#00H	
	MOVC A, @A+DPTR	
	MOV SBUF, A	
wait :	JNB TI, wait	
	CLR TI	
	DJNZ R1, send-data	
	RET	

Program 7.6.7 : Write an assembly program to take the data from port 0 and transfer it serially continuously use serial interrupt and write the ISR.

Solution :

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	org 0023H	
	JNB TI, done	
	MOV SBUF, P0	
	CLR TI	

Label	Instruction	Comments
done :	RETI	
	org 1000H	
main :	MOV P0,#0FFH	
	MOV TMOD,#20H	
	MOV TH1,#0E8H	
	MOV SCON,#50H	
	MOV IE,#90H	
	SETB TR1	
	MOV SBUF, P0	
here :	SJMP here	

Program 7.6.8 : *Write an assembly program to send YES using serial port of 8051. Use interrupt.*

Solution :

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	org 0100H	
	dB "YES"	
	org 0023H	
	JNB TI, done	
	MOV A, R3	
	MOV A, @A+DPTR	
	MOV SBUF, A	
	INC R3	
	CJNE R3, #03H, done	
	MOV R3, #00H	
done :	CLR TI	
	RETI	
	org 1000H	
main :	MOV TMOD, #20H	
	MOV TH1, #0FDH	
	MOV SCON, #50H	

Label	Instruction	Comments
	MOV IE, #90H	
	SETB TR1	
	MOV R3, #00H	
	MOV DPTR, #0100H	
	MOV A, #00H	
	MOV A, @A+DPTR	
	MOV SBUF, A	
	INC R3	
here :	SJMP here	

7.7 Solved Examples

Example 7.7.1 : If 8051 is working at 12 MHz crystal frequency, write the program to transmit characters at 2400 baud rate. The delay between sending the two characters is 1 ms and assume that characters to be sent are stored in external RAM from 0100H to 01FFH. Give flowchart for this program.

Solution :

For a delay of 1 msec. We will use timer 0, and timer 1 for baud rate.

Crystal frequency = 12 MHz

$\therefore$ Time for 1 machine cycle = $\frac{12}{12\text{MHz}} = 1\ \mu\text{sec}$.

Delay required = 1 msec.

$\therefore$ Machine cycles required = $\frac{1\ \text{msec}}{1\ \mu\text{sec}} = 1000$

Using timer 0, in mode 1 we will initialize the counter to $(65536)_{10} - (1000)_{10} = (64536)_{10} = (FC18)_{16}$

For baud rate of 2400,

$$\text{Baud rate} = \frac{2^{\text{SMOD}}}{(32)_{10}} \times \frac{\text{Crystal frequency}}{(12)_{10} \times (256 - \text{TH1})}$$

$$\therefore \quad \text{TH1} = (242.97)_{10} \cong (243)_{10} = (F3)_{16}$$

(Assuming SMOD = '0')

Program

Label	Instruction
	org 0000H
	LJMP main
	org 000BH

Label	Instruction
	CLR TR0
	DJNZ R0, next
	SJMP done1
next :	MOV TL0, #18H
	MOV TH0, #0FCH
	INC DPTR
	MOVX A, @DPTR
	MOV SBUF, A
	SETB TR0
done1 :	RETI
	org 0023H
	JNB TI, over
	SETB TR0
over :	RETI
	org 0100H
main :	CLR SMOD
	MOV TMOD, #21H
	MOV IE, #92H
	MOV TH1, #0F3H
	MOV TH0, #0FCH
	MOV TL0, #18H
	MOV SCON, #20H
	SETB TR1
	MOV R0, #0FFH
	MOV DPTR, #0100H
	MOVX A, @DPTR
	MOV SBUF, A
here :	SJMP here
	end

Example 7.7.2 : Write assembly language program for 8051 microcontroller to transfer message "ARM7" serially at baud rate of 2400 in mode 1. **MU - Dec. 17. 10 Marks**

Solution :

For a delay of 1 msec. We will use timer 0, and timer 1 for baud rate.

Crystal frequency = 12 MHz

$\therefore$ Time for 1 machine cycle = $\frac{12}{12MHz} = 1\ \mu sec.$

Delay required = 1 msec.

$\therefore$ Machine cycles required = $\frac{1\ msec}{1\ \mu sec} = 1000$

Using timer 0, in mode 1 we will initialize the counter to $(65536)_{10} - (1000)_{10} = (64536)_{10} = (FC18)_{16}$

For baud rate of 2400,

$$\text{Baud rate} = \frac{2^{SMOD}}{(32)_{10}} \times \frac{\text{Crystal frequency}}{(12)_{10} \times (256 - TH1)}$$

$\therefore$ TH1 = $(242.97)_{10} \cong (243)_{10} = (F3)_{16}$

(Assuming SMOD = '0')

Program

Label	Instruction
	org 0000H
	LJMP main
	org 000BH
	CLR TR0
	DJNZ R0, next
	SJMP done1
next :	MOV TL0, #18H
	MOV TH0, #0FCH
	INC DPTR
	CJNE DPL,#54H,ovr
	MOV DPL,#50H
ovr:	MOV A,#00H
	MOVC A, @A+DPTR
	MOV SBUF, A
	SETB TR0

Label	Instruction
done1 :	RETI
	org 0023H
	JNB TI, over
	SETB TR0
over :	RETI
	org 0050H
	"ARM7"
	org 0100H
main :	CLR SMOD
	MOV TMOD, #21H
	MOV IE, #92H
	MOV TH1, #0F3H
	MOV TH0, #0FCH
	MOV TL0, #18H
	MOV SCON, #50H
	SETB TR1
	MOV R0, #0FFH
	MOV DPTR, #01050H
	MOV A,#00H
	MOVC A, @A+DPTR
	MOV SBUF, A
here :	SJMP here
	end

Example 7.7.3 : Write a program to send string of twenty characters on TxD line of 8051 microcontroller. A string is stored from internal data memory location 30H onwards. Microcontroller operating on 11.0592MHz, choose baud rate at 9600.

Solution :

Baud rate = 9600 bps

$$\therefore \quad 9600 = \frac{2^{SMOD}}{(32)_{10}} \times \frac{\text{Crystal frequency}}{(12)_{10} \times (256 - TH1)}$$

$$\therefore \quad TH1 = (253)_{10} = (FD)_{16}$$

(Assuming SMOD = '0')

Label	Instruction
	org 0000H
	LJMP main
	org 0023H
	CLR TI
	INC R0
	CJNE R0, #44H, over
	RETI
over :	MOV SBUF, @R0
	RETI
	org 0100H
main :	MOV TH1, #0FDH
	MOV TMOD, #20H
	MOV IE, #90H
	SETB TR1
	MOV R0, #30H
	MOV SBUF, @R0
here :	SJMP here
	End

Example 7.7.4 : Write a 8051 program to send 'N' or 'H' to the serial port depending on the baud rate chosen as normal or high speed. A switch is connected to port 2.0 to decide the baud rate as
SW = 0, 28800 baud rate "Normal" speed
SW = 1, 56K baud rate "High" speed
Assume XTAL= 11.0592MHz for both cases.

Solution :

Normal speed, baud rate = 28800 bps

$$\therefore \quad 28{,}800 = \frac{2^{SMOD}}{(32)_{10}} \times \frac{\text{Crystal frequency}}{(12)_{10} \times (256 - TH1)}$$

(Assume SMOD = '0')

$\therefore$ $TH1 = (255)_{10} = (FF)_{16}$

and for baud rate of 56000 bps. We will make SMOD = 1, to get double baud rate.

Label	Instruction
	org 0000H
	LJMP main
	org 0100H
main :	MOV TMOD, #20H
	SETB P2.0
	MOV TH1, #0FFH
repeat :	JNB P2.0, normal
	SETB SMOD
	MOV SBUF, 'H'
	SJMP over
normal :	CLR SMOD
	MOV SBUF, 'N'
over :	JNB TI, over
	SJMP repeat

Example 7.7.5 : Write a program to transfer message "PASS" using serial communication of 8051 at 4800 baud rate. Oscillator frequency is 11.0592 MHz.

Solution :

Baud rate = 4800 bps

$$\therefore \quad 4800 = \frac{2^{SMOD}}{(32)_{10}} \times \frac{\text{Crystal frequency}}{(12)_{10} \times (256 - TH1)}$$

$\therefore$ $TH1 = (250)_{10} = (FA)_{16}$

Label	Instruction	Comments
	MOV TMOD,#20H	Timer 1, mode 2
	MOV TH1, #0FDH	9600 baud rate
	MOV SCON,#50H	
	SETB TR1	Start Timer 1
L1 :	MOV A,#"P"	Transfer "P"

Label	Instruction	Comments
L2 :	JNB TI, L2	Wait for transmission to complete
	CLR TI	
	MOV A, #"A"	Transfer "A"
L3 :	JNB TI, L3	Wait for transmission to complete
	CLR TI	
	MOV A, #"S"	Transfer "S"
L4 :	JNB TI, L4	Wait for transmission to complete
	CLR TI	
	MOV A, #"S"	Transfer "S"
L5 :	JNB TI, L5	Wait for transmission to complete
	CLR TI	
here:	SJMP here	
	RET	

Example 7.7.6 : Write a program to transmit "HAPPY" serially on Tx pin of 8051 microcontroller with a baud rate of 9600. Assume crystal frequency of 11.0592 MHz.

Solution :

Label	Instruction	Comments
	MOV TMOD, #20H	Timer 1, mode 2
	MOV TH1, #FDH	9600 baud rate
	MOV SCON, #50H	8 bits, 1 stop bit, REN enabled
	SETB TR1	Start Timer 1
L1 :	MOV A, # 'H'	Transfer 'H'
	ACALL TRANS	
	MOV A, # 'A'	Transfer 'A'
	ACALL TRANS	
	MOV A, # 'P'	Transfer 'P'
	ACALL TRANS	
	MOV A, # 'P'	Transfer 'P'

Label	Instruction	Comments
	ACALL TRANS	
	MOV A, # 'Y'	Transfer 'Y'
	ACALL TRANS	
	SJMP L1	Load SBUF
L2 :	JNB TI, L2	wait for last bit to transfer
	CLR TI	get ready for next character byte
	RET	

Example 7.7.7 : Write assembly language program for 8051 to transfer message "ENGINEER" serially at the baud rate of 4800 in mode 1. MU - Dec. 17, 10 Marks

Solution :

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	org 0100H	
	dB "ENGINEER"	
	org 1000H	
main :	MOV TMOD,#20H	Timer 1 in Mode 2 for Baud rate
	MOV TH1,#0FAH	Timer 1 count for 4800 baud/sec
	MOV SCON,#50H	Mode 1 for serial communication
	SETB TR1	Run timer 1
	MOV R1,#08H	
	MOV DPTR,#0100H	
send-data :	MOV A,#00H	
	MOVC A, @A+DPTR	
	MOV SBUF, A	
wait :	JNB TI, wait	
	CLR TI	
	INC DPTR	
	DJNZ R1, send-data	
	RET	

7.8 Exam Pack (University and Review Question)

Q. 1 Distinguish between serial and parallel data transfer. **(Refer Section 7.1)** **(5 Marks)**

Q. 2 Explain data communication formats in serial communication. **(Refer Section 7.2)** **(5 Marks)**

Q. 3 Differentiate between Synchronous and Asynchronous data transfer. **(Refer Section 7.3)** **(5 Marks)**

Q. 4 How is data transmitted and received using RS232 ? **(Refer Section 7.4)** **(5 Marks)**

Q. 5 8051 microcontroller with XTAL frequency = 11.0592 MHz, find the TH1 value needed to have the following baud rates (i) 9600 (ii) 2400 (iii) 1200. **(Refer Section 7.6.1)** **(M-16, 5 Marks)**

Q. 6 8051 microcontroller with XTAL frequency = 11.0592 MHz. Find the TH value needed to have the following baud rates of 9600. **(Refer Section 7.6.1)** **(D-17, 5 Marks)**

Q. 7 Give steps for programming 8051 for serial data transfer. **(Refer Section 7.6.2)** **(5 Marks)**

Q. 8 Write assembly language program for 8051 microcontroller to transfer message "ARM7" serially at baud rate of 2400 in mode 1. **(Refer Example 7.7.2)** **(D-17, 10 Marks)**

Q. 9 Write assembly language program for 8051 to transfer message "ENGINEER" serially at the baud rate of 4800 in mode 1. **(Refer Example 7.7.7)** **(D-14, 10 Marks)**

Chapter Ends...

□□□

CHAPTER 8

Interfacing Memory to 8051

8.1 8051 Interfacing to External Memory

Pin configuration

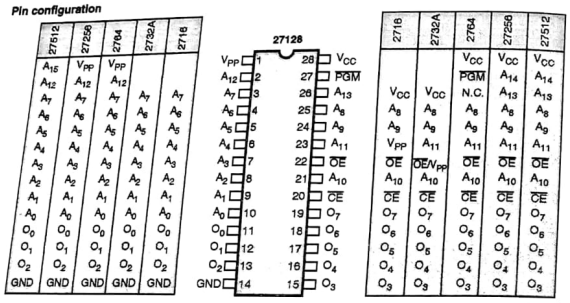
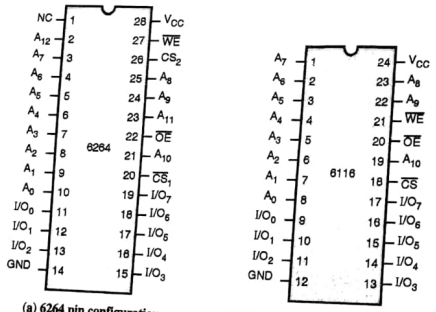


Fig. 8.1.1 : EPROM series Pin configuration



(a) 6264 pin configuration (b) 6116 pin configuration

Fig. 8.1.2

8.2 Memory Organisation

Q. 8.2.1 Write short note on : Memory Organisation (Ref. Sec. 8.2) **(4 Marks)**

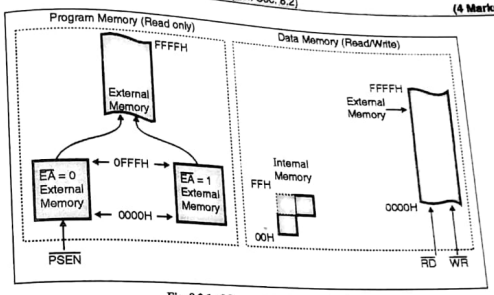


Fig. 8.2.1 : Memory structure of 8051

- Fig. 8.2.1 shows the basic memory structure for microcontroller 8051. It can access upto 64 KB of program memory and 64 KB of data memory. It has 4 KB of internal program memory and 256 bytes of internal data memory.
- Each memory has a separate addressing mechanism. It also has different control signals and different functions. Each memory will use the same address and data bus, but with different control signals.

8.2.1 Program Memory

Q. 8.2.2 Explain the external program memory interface of 8051 (Ref. Sec. 8.2.1) **(6 Marks)**

- The microcontroller 8051 has 64 KB external and 64 KB internal program memory. Fig. 8.2.2 shows the memory map of program memory.

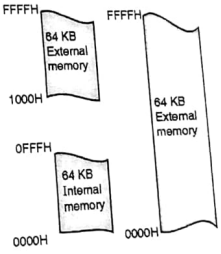


Fig. 8.2.2 : 8051 Program memory

- If the $\overline{EA}$ pin is connected to V_{cc} then the program fetches the addresses from 0000H through 0FFFH are directed to the internal memory and the addresses from 1000H through FFFFH through the external ROM
- If the $\overline{EA}$ pin is grounded then all the addresses from 0000H through FFFFH are fetched by the 64 KB external ROM/EPROM. The $\overline{PSEN}$ signal activates the output enable signal for ROM/EPROM.

Fig. 8.2.3 shows how 8051 accesses the external program memory.

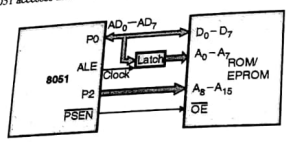


Fig. 8.2.3 : Accessing external program memory

- Port 0 is used as multiplexed address / data bus ($AD_0 - AD_7$) which is demultiplexed using a latch to give the lower order address bus $A_0 - A_7$ and data bus.
- Port 2 is used to provide higher order address bus. Fig. 8.2.4 shows the timing diagram for external program memory read cycle.

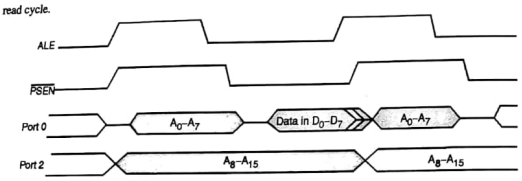


Fig. 8.2.4 : Timing waveforms for external program memory read cycle

8.2.2 Data Memory

Q. 8.2.3 Explain the external data memory interface of 8051. (Ref. Sec. 8.2.2) **(6 Marks)**

- 8051 microcontroller can address upto 64 KB of external data memory and 256 bytes of internal data memory.
- Fig. 8.2.5 shows a memory map of the 8051 data memory.
- The instruction "MOVX" is used to access the external data memory. The internal data memory space is divided into three blocks lower 128 bytes (00H to 7FH), upper 128 bytes (80H to FFH) and the SFRs. The upper 128 bytes and SFRs occupy the same block of address space, although both are separate.

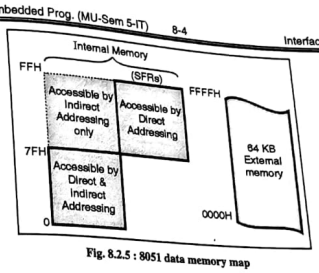


Fig. 8.2.5 : 8051 data memory map

- Fig. 8.2.6 shows data is accessed from external data memory.
- The multiplexed address / data bus is demultiplexed using a latch to generate lower order address ($A_0 - A_7$) and data bus ($D_0 - D_7$).
- Port 2 generates higher order address ($A_8 - A_{15}$).
- The $\overline{RD}$ and $\overline{WR}$ signals will according select memory read and memory write operation respectively. Fig. 8.2.7 shows the timing diagram for external data memory read cycle and Fig. 8.2.8 shows the timing diagram for external data memory write cycle.

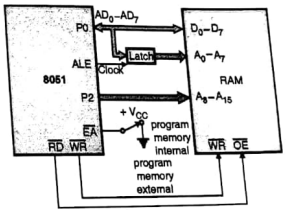


Fig. 8.2.6 : Accessing external data memory

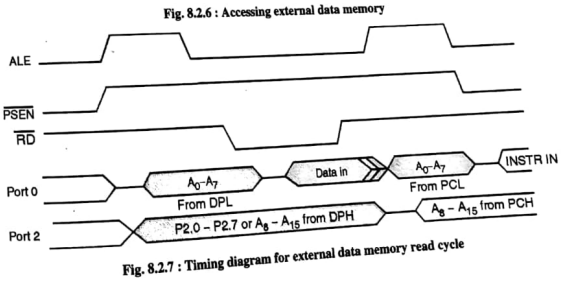


Fig. 8.2.7 : Timing diagram for external data memory read cycle

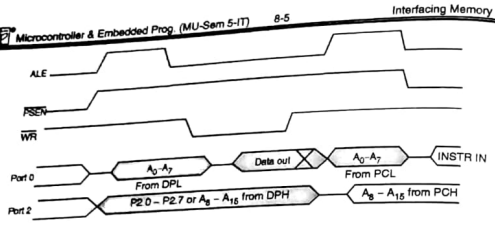


Fig. 8.2.8 : Timing diagram for external data memory write cycle

8.2.3 Interfacing 8051 to External Data ROM

Q. 8.2.4 Draw and explain interfacing 8051 to External Data ROM. (Ref. Sec. 8.2.3) **(8 Marks)**

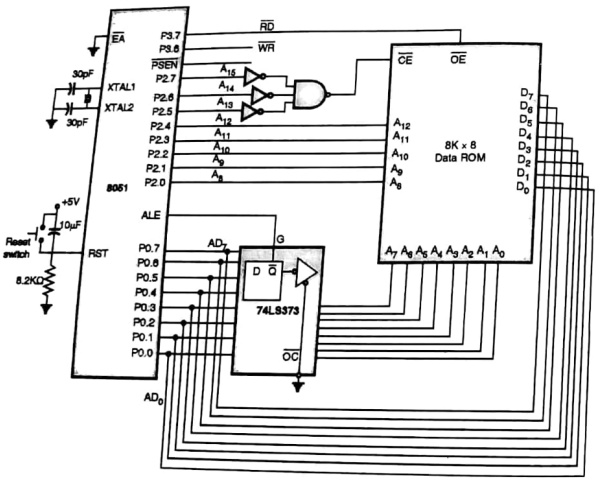


Fig. 8.2.9 : 8051 Connection to External Data ROM

- The 8051 has 64 KB data memory space and 64 KB program memory space. Program memory space is accessed using the program counter (PC) to locate and fetch the instructions.
- The data memory space is accessed using DPTR register and an instruction MOVX where X indicates external.
- To connect 8051 to external data ROM containing we use $\overline{RD}$ (pin 3.7) as shown in Fig. 8.2.9
- In external program ROM, $\overline{PSEN}$ signal was used for fetching the code. In external data ROM $\overline{RD}$ signal is used for the fetching the data.
- MOVX instruction is used to access the external data memory space.

8.3 Generation of Address, Data and Control Bus

Micro-controller follows Intel standard. It provides multiplexed address and data bus. For demultiplexing, use 74LS373 latch. Refer Figs. 8.3.1(a) and (b).

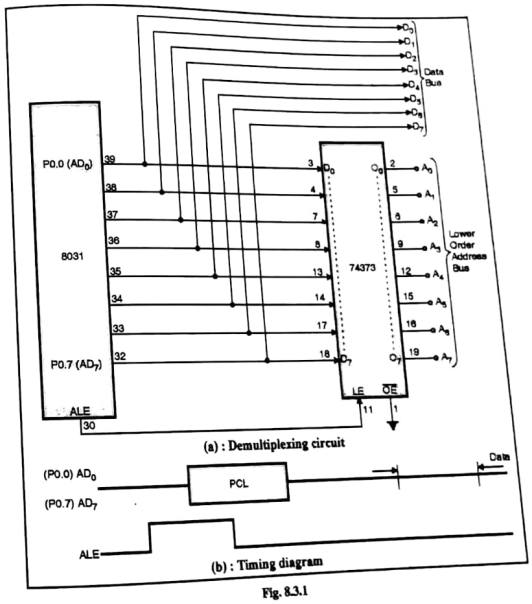


Fig. 8.3.1

- *Initially ALE is HIGH, therefore 74373 is enabled.*
- *When ALE is HIGH, on P0 we get address i.e. PCL (Program counter lower byte). Therefore output of latch is nothing but $A_0 - A_7$, as LE = 1.*
- *After sometime ALE = 0, therefore Contents will be latched. Now eventhough input disappears, output $A_0 - A_7$ will be intact till next ALE cycle.*
- *At this stage we are ready with data bus $D_0 - D_7$ and lower address bus . Higher order address bus we get from port 2.*
- *Finally in control bus, the signals will be*

(i) $\overline{RD}$ (ii) $\overline{WR}$

(iii) $\overline{PSEN}$ (iv) Reset

(v) ALE

8.4 Interfacing Examples

Example 8.4.1 : Design a 8051 based microcontroller system with following specifications.

(i) 8051 CPU working at 12 MHz (ii) 32 KB program memory

(iii) 32 KB data memory (iv) 8255 PPI

Discuss the design.

Solution :

Step 1 : Total EPROM required = 32 KB

Chip (Device) size available = 8 KB (i.e. IC 2764)

$\therefore$ No. of chips required = $\frac{32\ KB}{8\ KB} = 4$

Chip 1 : Starting address = 0000 H

Chip size = 8KB = 1FFFH

$\therefore$ Ending address = 1FFFH.

Chip 2 : Starting address = 2000H

Chip size = 8 KB = 1FFFH

$\therefore$ Ending address = 3FFFH.

Chip 3 : Starting address = 4000H

Chip size = 8 KB = 1FFFH

$\therefore$ Ending address = 5FFFH

Chip 4 : Starting address = 6000H

Chip size = 8 KB = 1FFFH

$\therefore$ Ending address = 7FFFH

Step 2 : Total RAM required = 16 KB

Chip size = 8 KB (i.e. IC 6264)

$\therefore$ Number of chip required = $\frac{16\ KB}{8KB} = 2$

Chip 1 : Starting address = 0000H

Chip size = 8 KB = 1FFFH

$\therefore$ Ending address = 1FFFH

Chip 2 : Starting address = 2000H

Chip size = 8 KB = 1FFFH

$\therefore$ Ending address = 3FFFH

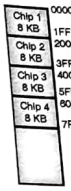


Fig. P. 8.4.1: ROM Organization

Step 3 : Memory map for EPROM.

		A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
Chip 1	Starting address = 0000H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_0$	Ending address = 1FFFH	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
Chip 2	Starting address = 2000H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_1$	Ending address = 3FFFH	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Chip 3	Starting address = 4000H	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_2$	Ending address = 5FFFH	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
Chip 4	Starting address = 6000H	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_3$	Ending address = 7FFFH	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Decoding logic : Chip size = $8KB = 2^{13}$

Thus for decoding logic neglect lower 13 address lines (A_0 to A_{12}) and consider the remaining 3 address lines ($A_{13} - A_{15}$) as shown, in the memory map table above. And accordingly use the decoder select lines for example $\overline{y}_0$ for 000, $\overline{y}_1$ for 001 and so on; as marked by rectangles in the above memory map.

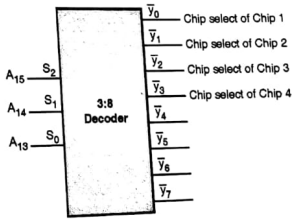


Fig. P. 8.4.1(a) : Decoder for ROM

Step 4 : Memory map for RAM.

		A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
Chip 1	Starting address = 0000H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_0$	Ending address = 1FFFH	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
Chip 2	Starting address = 2000H	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_1$	Ending address = 3FFFH	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Decoding logic : Chip size = $8KB = 2^{13}$

Thus for decoding logic neglect lower 13 address lines (A_0 to A_{12}) and consider the remaining 3 address lines (A_{13} – A_{15}) as shows, in the memory map table above. And according use the decoder select lines for example $\overline{y}_0$ for 000, $\overline{y}_1$ for 001 ; as marked by circles in the above memory map.

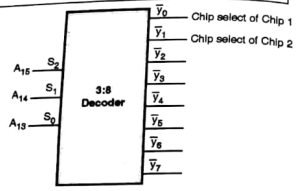
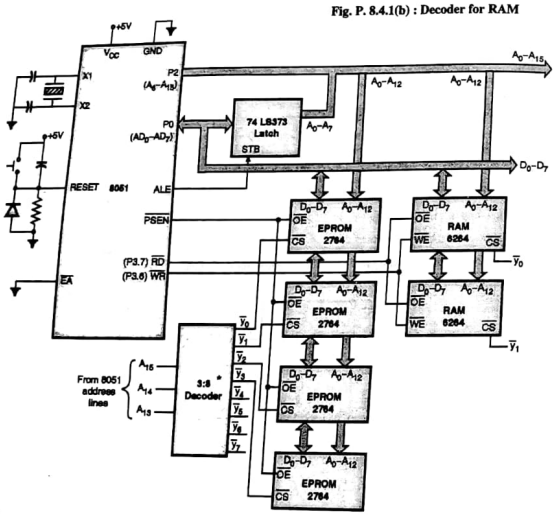


Fig. P. 8.4.1(b) : Decoder for RAM



*Common decoder in this case is possible as the number of lines used for decoding are same

Fig. P. 8.4.1(c) : Final design implementation

Example 8.4.2 : With neat schematic only, show the interface of 8051 will following memory and I/O sub systems.

(i) 32 KB SRAM (16 KB device)

(ii) 32 KB EPROM (16 KB device)

(iii) 8 LEDs with drivers

Solution :

Step 1 : Total EPROM required = 32 KB

Chip size available = 16 KB (i.e. IC 27128)

∴ Number of chips required = $\frac{32\ KB}{16\ KB} = 2$

Chip 1 : Starting address = 0000H

Chip size = 16KB = 3FFFH

∴ Ending address = 3FFFH.

Chip 2 : Starting address = 4000H

Chip size = 16 KB = 3FFFH

∴ Ending address = 7FFFH.

Step 2 : Total RAM required = 32 KB Chip size available = 16 KB (i.e. IC 62128)

∴ Number of chip required = $\frac{32\ KB}{16KB} = 2$

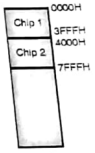


Fig. P. 8.4.2 : ROM organization

Chip 1 : Starting address = 0000H

Chip size = 16 KB = 3FFFH

∴ Ending address = 3FFFH

Chip 2 : Starting address = 4000H

Chip size = 16 KB = 3FFFH

∴ Ending address = 7FFFH

Step 3 and 4 : Memory map in EPROM.

(In this case memory map for RAM is also same.)

		A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
Chip 1	Starting address = 0000H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_0$	Ending address = 3FFFH	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
Chip 2	Starting address = 4000H	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_1$	Ending address = 7FFFH	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

Decoding logic

Chip size = $16KB = 2^{14}$

Thus for decoding logic neglect lower 14 address lines (A_0 to A_{13}) and consider the remaining 2 address lines (A_{14} ~ A_{15}) as shown, in the memory map table above. And accordingly use the decoder's select lines for example $\overline{y}_0$ for 00, $\overline{y}_1$ for 01; as marked by rectangles in the above memory map.

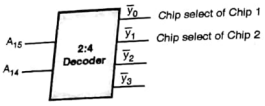


Fig. P. 8.4.2(a) : Decoder for ROM / RAM

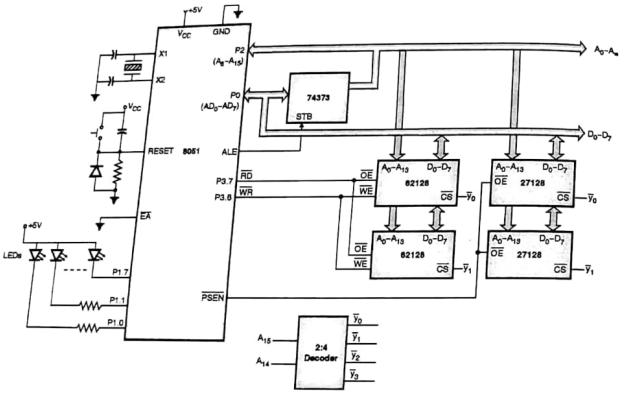


Fig. P. 8.4.2(b) : Final design implementation

Example 8.4.3 : Design a 8051 based computer having,

(a) CPU working at 6MHz.

(b) Firmware of 8KB using 4KB devices.

(c) Data memory of 16 KB using 8KB devices

(d) ADC 0809 (8 channel, 8 bit ADC)

Discuss the design and give proper memory maps.

Solution :

Step 1 : Total EPROM required = 8 KB

Chip size available = 4 KB (i.e. IC 2732)

$\therefore$ Number of chips required = $\frac{8\text{ KB}}{4\text{ KB}} = 2$

Chip 1 : Starting address = 0000H

Chip size = 4KB = 0FFFH

$\therefore$ Ending address = 0FFFH.

Chip 2 : Starting address = 1000H

Chip size = 4 KB = 0FFFH

$\therefore$ Ending address = 1FFFH.

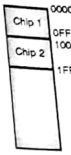


Fig. P. 8.4.3(a) : ROM organization

Step 2 : Total RAM required = 16 KB

Chip size available = 8 KB (i.e. IC 6264)

$\therefore$ Number of chips required = $\frac{16\text{ KB}}{8\text{ KB}} = 2$

Chip 1 : Starting address = 0000H

Chip size = 8 KB = 1FFFH

$\therefore$ Ending address = 1FFFH

Chip 2 : Starting address = 2000H

Chip size = 8 KB = 1FFFH

$\therefore$ Ending address = 3FFFH

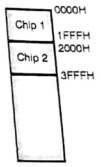


Fig. P. 8.4.3(b) : RAM organization

Step 3 : Memory map for EPROM,

		A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
Chip 1	Starting address = 0000H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_0$	Ending address = 0FFFH	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
Chip 2	Starting address = 1000H	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
$\overline{y}_1$	Ending address = 1FFFH	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

Decoding logic :

Chip size = $4KB = 2^{12}$

Thus we neglect 12 address lines (A_0 to A_{11}) and use the remaining 4 address lines (A_{12} to A_{15}) for decoding logic as shown in the table above.

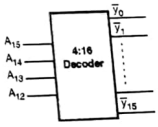


Fig. P. 8.4.3(c) : Decoder for ROM

Step 4 : Memory map for RAM.

		A_{15} A_{14} A_{13}	A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
Chip 1	Starting address = 0000H	0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0
$\overline{y}_0$	Ending address = 1FFFH	0 0 0	1 1 1 1 1 1 1 1 1 1 1 1 1
Chip 2	Starting address = 2000H	0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0
$\overline{y}_1$	Ending address = 3FFFH	0 0 1	1 1 1 1 1 1 1 1 1 1 1 1 1

Decoding logic

Chip size $= 8KB = 2^{13}$

Thus we neglect 12 address lines (A_0 to A_{12}) and we the remaining 3 address lines (A_{13} to A_{15}) for decoding logic as shown in the table above.

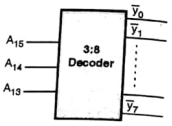


Fig. P. 8.4.3(d) : Decoder for RAM

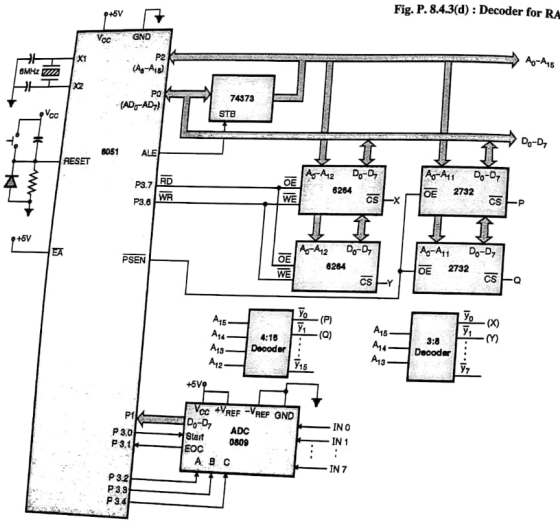


Fig. P. 8.4.3(e) : Final design implementation

Example 8.4.4 : Design 8751 (EPROM version of 8051) based data monitoring system with following specifications :

(i) Microcontroller operating at 6 MHz.
(ii) 4 KB EPROM for firmware support
(iii) Minimum 80 B SRAM
(iv) ADC 0809 (8-bit, 8 channel successive approximate type ADC)
(v) Serial port interface
(vi) RESET facility should be provided.

Solution :

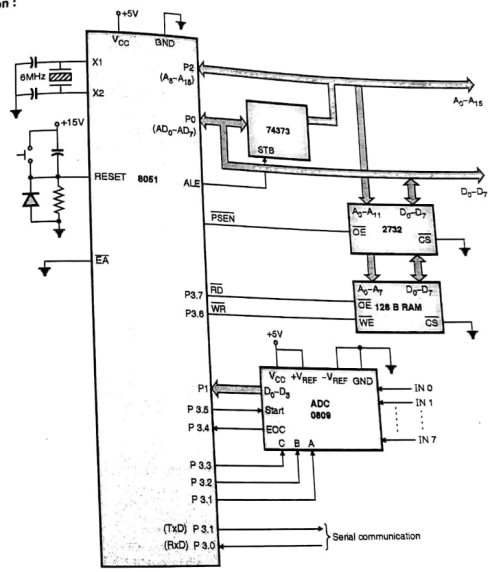


Fig. P. 8.4.4(a) : Final design implementation

Step 1 : Total EPROM required = 4 KB

Chip size available = 4 KB (assume) ∴ Number of chips = 1

Chip 1 : Starting address = 0000H

Chip size = 4KB = 0FFFH ∴ Ending address = 0FFFH.

Example 8.4.5 : Explain how to interface external 16 K × 8 Data RAM and program RAM with 8051 microcontroller. Draw the interface diagram.

Solution :

Step 1 : Total RAM required (Data) = 16 KB

Chip size available = 16 KB (Assumption)

∴ Number of chips required = 1

Chip 1 : Starting address = 0000H

Chip size = 16 KB ⇒ 3FFFH

∴ Ending address = 3FFFH

Step 2 : Total RAM required (Program) = 16 KB

Chip size available = 16 KB (Assumption)

∴ No. of chips required = 1

Chip 1 : Starting address = 0000H

Chip size = 16 KB ⇒ 3FFFH

∴ Ending address = 3FFFH

Step 3 : Memory Map

Case (a) Data Memory `

		A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
RAM	SA = 0000H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Chip 1	EA = 3FFFH	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Case (b) Program Memory

In this case the memory map will be same as above.

Step 4 : Final implementation

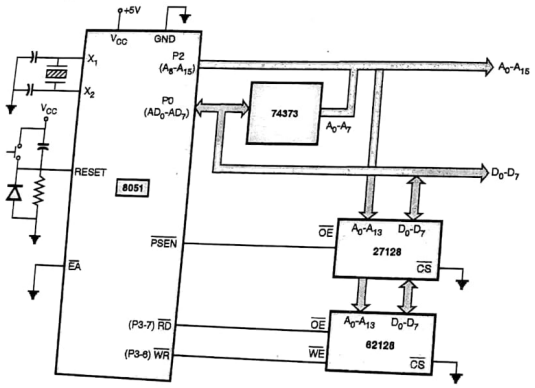


Fig. P. 8.4.5

8.5 Comparison of Different Microprocessors

Processor	Date	Performance (MIPS) (instr. set)	Registers (no-Size bits) (tot regs)	Data/Addr bus PC bus CPU max I/O	Clock (Hz) Bus Power (v/W)	Transistors (10⁶) (mm/la yers) (mm²)	Max. Mem Cap Physical/Virtual (L1/L2 Cache)	No. pins (pack)	Notes
4004 4040	1971 1972	0.006 (45)	16x4b GP	4/12 bits (? B/s)	0.108 MHz (-15w/30mW)	0.0023 (10mm pMOS) (12mm²)	2Kb code-ROM/ 640b data-RAM	16-DIP	**This was the first microprocessor chip.** **4004-CPU:** It had 16x4b-regs, 4b-Acc, 12b-PC, 3 level-stack, 4b-BCD-ALU and 32x4b I/O ports. **4040-CPU:** It was a 24-pin IC. It had 14 instructions, 8 level-stack, 8K-ROM & INT. *Applications (BCD arith): calculators & embedded control, eg SIM4; first microcomputer (1972), elevators, NASA Pioneer.*
8008	1972	.06(48)	7x8b GP	8/14 bits (? B/s)	0.2-0.8 MHz (5,-9v)	0.002-0.0035 (10mm pMOS)	16Kb	18-DIP	***First 8-bit CPU chip. It had 8 level-stack, 2 clocks, 2 voltages, first vect-interrupt, requires 20-40 interfacing chips, 8-in/24-output 8b/ ports.***
8080 8080A	1974 1976	0.29(78)	10x8/16b	8/16 bits (? B/s)	2-3 MHz (- 5,5,12w/1.5W)	0.0045, (6mm pMOS/nMOS)	0.006 64Kb	40-DIP	*It had 2 clock, 3 volt, (6b addr, ext stack, multi interrupt, 256 I/O port, 6 support-chips. It was used in first Home Computer.*
8085	1976	0.37(80)	10x8/16b	8/16 bits (? B/s)	3-6 MHz (5v/1.5W)	0.0065 (3mm nMOS)	64Kb	40-DIP	It was 8080 compatible, first single-clock, vectored int, first single-5v, first on-chip serial I/O, clock generator, bus controller Min system of 3 IC's: CPU, RAM/IO, ROM. Good support-chips.
8086 8088	1978 1979	0.33-0.75 (95/300)	14x16b	16,8/20 bits (2-3 MB/s) (2-10Mb/s)	5-10 MHz (5-10 (5v/2.5W)	0.029 (3mm inHMOS) (33mm²)	1Mb	40 DIP	**First 16-bit x86 chip.** Inst cycle: <1/s ms. It had CISC arch, parallel EU & BIU, 1-stg pipeline with 6byte prefetch-queue, 64K I/O 8/16b ports, 256 intr. 64K segmented memory, Integer full arithmetic. String operations. Low-cost 8088 (16b CPU, 8b data bus, 4b p queue).

Processor	Date	Performance (MIPS) (instr. set)	Registers Size (no-bits) (tot regs)	Data/Addr bus PC bus speed CPU max I/O	Clock (Hz) Bus (Hz) Power (vW)	Transistors (10⁶) (mm/a yers) (mm²)	Max Mem Cap Physical/Virtual (L1/L2 Cache)	No. pins (pack)	Notes
80186 80188	1982 1990	1(105)	14x16b	16,8/20 bits (? B/s)	6-25 MHz (2.7-5v/1.5W)	0.055 (3.1mm nMOS)	1Mb	68-LCC 80-PGA	On-chip integration: PIC, 3 timers, 2 DMA, clock, serial, I/O ports, support logic (replace up to 20 chips). +10 instr. Fault tolerance protection.
80286	1982	0.9-2.6 (110)	15x16b	16/24 bits (4-12 MB/s) (10-38 MB/s)	6-20 MHz (6-12 MHz) (5v/2-3.3w)	0.134 (1.5mm HMOS) (47 mm²)	16Mb/1Gb	68-LCC 68-PGA	First "real" processor. Inst cycle: <¼ ms. It had real & protected mode, On-chip MMU, virtual mem (1Gb), 4+ rings & multitasking. 1-stg pipeline 6byte prefetch-queue.
80386 DX, SX, SL	1985 1988 1990 1991	2.5-11(120+)	18x32b	32/32 bits 16/24 bits (32-50 MB/s) (60-150 MB/s)	16-33 MHz (8-16 Mhz) (3.5w/2.5W)	0.275-0.855 (1.5,1mm CHMOS) (43mm²)	16Mb/256Gb 4Gb/64Tb 32-256K L2	132-PGA	**First 32-bit x86 chip.** Most instr exec 2 cycles (4-stg pipeline 16byte prefetch-queue). Extended 32b registers. First mem cache: TLB buffer & L2. Real, protected & v86 mode. Built-in "true" multitasking & 4K-page/64K-seg virtual mem.).
80486 DX, SX, DX2,DX4	1989- 1994	16-70	20x32b 8x80b FPU	32/32 bits (50-132 MB/s) (90-200 MB/s)	16-100 MHz (16,25,33Mhz) (3.3,5v/5W)	1.2-1.6 (1..8,.6mm/4 CHMOS) (81,345mm²)	4Gb/64Tb 8,16K L1 256K L2	168-PGA176-TQFP208-SQFP	First on-chip L1 cache & math coprocessor. Most inst exec 1 cycle (5-stg pipeline).
Pentium MMX Mobile Overdrive	1993- 1998	100-250 iSpec95 2-7	20x32b GP 8x80 FPU 8x64 MMX	64/32 bits 264 MB/s) (450-500 Mb/s)	60-266 MHz (50,60,66Mhz) (2.8-5v/3-16W)	3.1,4.5 (.8-.25mm/4 BICMOS) (90-296mm²)	4Gb/64Tb 16K L1 256-512K L2	273-PGA296-PGA320-TCP	Quasi-superscalar architecture: 2 exec. 5-stg pipeline (2 inst/cycle). Branch prediction. CPUID. On-chip cache (8K-data, 8K-code) & math unit (2-int, 1-FPU 8-stg pipeline). MMX (Matrix Math eXt, Multi-Media eXt,): Additional 57 instr.

8.6 Comparison of Different Microcontroller

Q. 8.6.1 Compare the various microcontroller device. (Ref. Sec. 8.6) **(6 Marks)**

Device	Flash (Kbytes)	RAM (Bytes)	F.max (MHz)	Vcc (V)	I/O Pins	UART	16-bit Timers	Analog Comparator	Program memory lock	Interrupt sources	Packages
AT89C51	4	128	24	2.7-6.0	32	1	2	-	3 level	6	PDIP 40
AT89C52	8	256	24	2.7-6.0	32	1	3	-	3 level	8	PDIP 40
AT89C1051	1	64	24	2.7-6.0	15	1	1	-	2 level	6	PDIP 20
AT89C2051	2	128	24	2.7-6.0	15	1	2	1	2 level	6	PDIP 20
AT89C2052	2	128	24	2.7-6.0	15	1	2	1	2 level	6	PDIP 20

8.7 Exam Pack (Review Questions)

Q. 1 Write short note on : Memory Organization. **(Refer Section 8.2)** **(4 Marks)**

Q. 2 Explain the external program memory interface of 8051. **(Refer Section 8.2.1)** **(6 Marks)**

Q. 3 Explain the external data memory interface of 8051. **(Refer Section 8.2.2)** **(6 Marks)**

Q. 4 Draw and explain Interfacing 8051 to External Data ROM. **(Refer Section 8.2.3)** **(8 Marks)**

Q. 5 Compare the various microcontroller device. **(Refer Section 8.6)** **(6 Marks)**

Chapter Ends...

□□□

CHAPTER 9

Interfacing Hex Keyboard and LCD Display

9.1 Keyboard

Q. 9.1.1 List the most likely effect if the keyboard program does not accomplish the following :

(i) Debounce keys when pressed down

(ii) Check for valid key code

(iii) Wait for all keys up before ending keyboard routine

(iv) Debounce keys when released (Ref. Secs. 9.1 and 9.2) **(10 Marks)**

- It is a human oriented input peripheral. It is used to input data or program into the microcomputer.
- It consists of push button type switches. When a key is pressed, the microcontroller identifies key depression and then performs appropriate operation.

9.1.1 Key Switch Mechanism

- The aim of this mechanism is to generate and transmit a code each time a key is pressed.
- The mechanism should send one and only proper code, when the key is pressed. Fig. 9.1.1 shows the general operation of a keyboard.

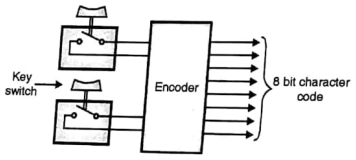


Fig. 9.1.1 : General operation of a keyboard

- The input keyboard is composed of a set of labelled push button switches. Each switch makes electrical contact when pressed.
- The nature of the contact should be reliable, have long life and feel right.

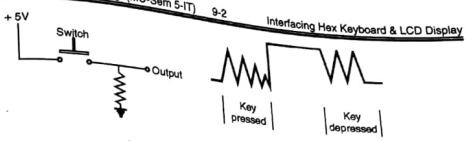


Fig. 9.1.2 : Bouncing of key switch

- In case of a push button key, the metal contact bounces few times, hence the voltage across the switch fluctuates and generates spikes in the signal.
- Therefore, it is necessary to debounce the mechanical switches.
- The key debouncing is done through hardware and software. Fig. 9.1.2 shows the bouncing of key switch.

9.1.2 Hardware Key Debouncing

Q. 9.1.2 What is key debounce ? How it is achieved ? (Ref. Sec. 9.1.2) **(4 Marks)**

- It is implemented by using flip-flop or latch. Fig. 9.1.3 shows a circuit diagram of hardware key debouncing.

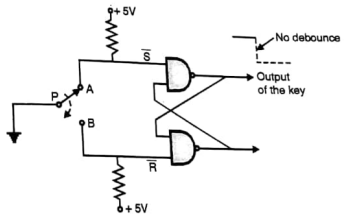


Fig. 9.1.3 : Hardware key debouncing

- When the switch is connected to A, the output of the latch goes high. When the key makes contact with B, the output changes from logic 1 to logic 0.
- The wiper bounces many times on contact B, but the output does not fluctuate between logic 1 and logic 0. When the wiper is not connected either to A or B, the output of the latch remains constant.

9.1.3 Software Key Debouncing

- In the software technique the microcontroller waits for 20 ms before it accepts the key as an input. If after 20 ms the key is pressed the key is accepted by microcontroller.
- The process of software key debouncing is as shown in Fig. 9.1.4.

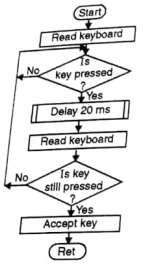


Fig. 9.1.4 : Software key debouncing

9.2 Keyboard Interface Circuit

Q. 9.2.1 List the most likely effect if the keyboard program does not accomplish the following :

(i) Debounce keys when pressed down

(ii) Check for valid key code

(iii) Wait for all keys up before ending keyboard routine

(iv) Debounce keys when released (Ref. Secs. 9.1 and 9.2) **(10 Marks)**

- The keyboard is interfaced with microcontroller through input ports.
- The keyboard consists of mechanical switches. These switches are arranged in non-matrix or matrix form.

9.2.1 Non-matrix Type Keyboard

- In non-matrix type keyboard, the key closure is identified by reading the port data, but it requires many port lines.
- The number of I/O lines is equal to number of keys. Fig. 9.2.1 shows the interfacing of octal non-matrix type keyboard.

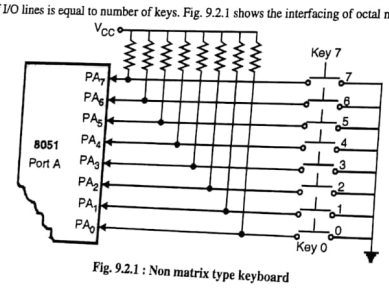


Fig. 9.2.1 : Non matrix type keyboard

To identify the key value the following three functions should be performed :

(1) Identifying a key closure.

(2) Debouncing the key.

(3) Encoding the key to an appropriate code like hexadecimal.

- The above three functions can be performed through hardware as well as software. As an example we will see hardware technique for identification of key closure. The interfacing is as shown in Fig. 9.2.2.

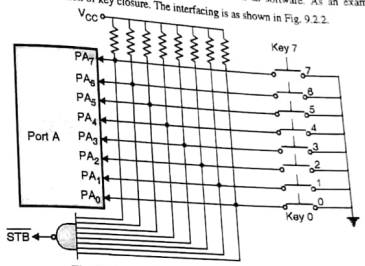


Fig. 9.2.2 : Hardware technique of identification

- When all keys are open, the output of NAND gate ($\overline{STB}$) goes low. When one of the keys is pressed, the output of NAND gate ($\overline{STB}$) goes high. The $\overline{STB}$ is used to identify that the key is pressed. This $\overline{STB}$ signal can be used to interrupt the microcontroller.

Syllabus Topic : KBD Matrix

9.2.2 Matrix Keyboard Interface

Q. 9.2.2 Write a short note on following : Interfacing Hex Keyboard with 8051. (Ref. Sec. 9.2.2) **(5 Marks)**

- In a simple keyboard interface one input line is required to interface one key and this increases the number of keys.
- When a large number of keys are to be interfaced, this technique is not useful. Matrix method is used in such cases, so that the number of connections are reduced.
- Fig. 9.2.3 shows 16 keys arranged in 4 rows and 4 columns. No connections is there, when the keys are open.
- If a key is pressed then there is connection between corresponding rows and columns. Such a matrix requires eight lines to complete the connections.

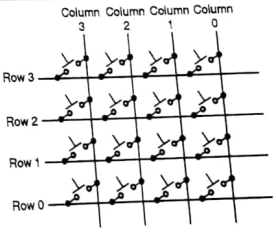


Fig. 9.2.3 : Matrix keyboard

– *If non matrix type connection is used then 16 lines will be required. So using method reduces the number of connections.*

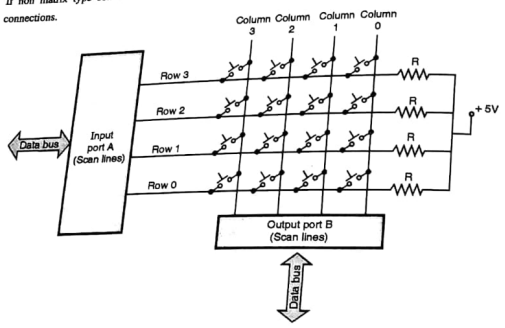


Fig. 9.2.4 : Matrix keyboard connections

– *Fig. 9.2.4 shows the interfacing of a matrix keyboard, it requires two ports : an input port and an output port. The columns are referred to as scan lines and rows are referred to as return lines.*

– *When a key is pressed, the corresponding row and column are connected i.e. they are shorted. If the output line of a row is high, then it makes the line of a column high and vice versa.*

– *The key is recognized by data which is sent on the output port and the input code that is received from the input port. The steps required to identify the pressed key are,*

(i) To identify if any key is pressed or not.

(a) All the column lines are made zero by sending low on all the output lines. i.e. all the keys in the keyboard matrix are activated.

(b) Read the status of rows i.e. return lines. If the status of all lines is logic high, the key is not pressed. Otherwise if the status of all lines is logic low, the key is pressed.

(ii) Debouncing the key. (Using software debouncing as explained earlier)

(iii) Identifying the pressed key.

(a) Activate the keys from one column by making one column line zero.

(b) Read the status of return lines. The zero on any return line indicates that key is pressed.

(c) Activate the keys from next column and repeat steps (b) and (c) for all the columns.

Example 9.2.1 : Interface a simple keyboard to microcontroller 8051.

Solution :

– Fig. P. 9.2.1 shows how a simple keyboard is interfaced to microcontroller 8051.

– As shown in Fig. P. 9.2.1 eight keys are connected to port 1 pins. Each port pin gives the status of key that is connected to that pin.

– If a pin shows logic 1 then the key is open otherwise the key is closed.

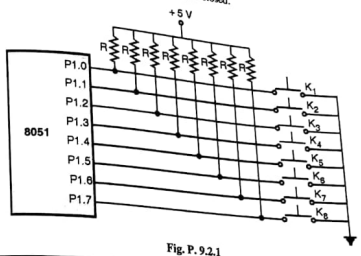


Fig. P. 9.2.1

Example 9.2.2 : Interface a 4 × 4 matrix keyboard to the microcontroller 8051.

Solution : Fig. P. 9.2.2 shows how a matrix keyboard is connected to the microcontroller 8051. The 4 × 4 matrix keyboard is connected to the Port 1 of microcontroller. The P1.4 - P1.7 lines of Port 1 are used as scan lines while the P1.0 to P1.3 lines of Port 1 are used as return lines.

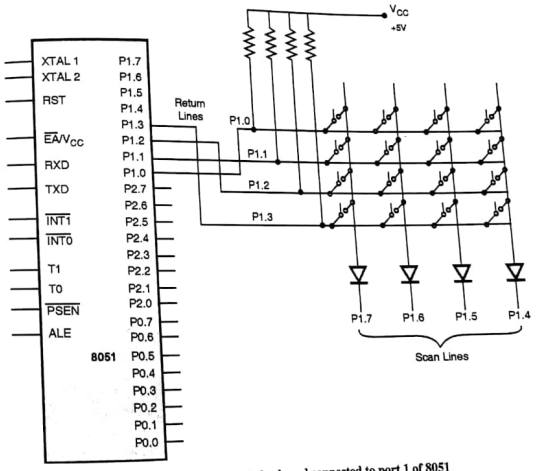


Fig. P. 9.2.2 : 4 × 4 matrix keyboard connected to port 1 of 8051

9.3 Display

- It is a human oriented output peripheral. It is used to display result or operand. One may use CRT, LED or LCD displays.
- A CRT is used to display large amount of data. LED and LCD displays are used to display small amount of data. The commonly used LED displays are numeric displays.

9.3.1 LED Displays

To drive a LED, there are two methods.

☞ **(Method 1)**

Connect the cathode of LED to ground. Connect the anode of LED to port pin of 8051, through a resistor as shown in Fig. 9.3.1.

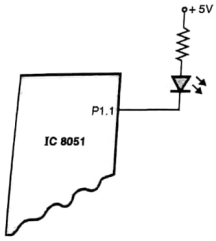


Fig. 9.3.1 : LED driven by 8051 port pin (Method 1)

- This method requires 8051 to source a huge amount of current required by the LED i.e. around 20 mA. But 8051 is not capable of sourcing a current more than 2 mA. This will make the LED glow very dim.

☞ **(Method 2)**

- Connect the anode of LED to V_{CC} through resistor. Connect the cathode of LED to the port pin of 8051 as shown in the Fig. 9.3.2.
- This method requires 8051 to sink a huge current required by LED i.e. 20 mA. 8051 can sink huge currents and hence it makes LED glow brighter. Hence we will always use method 2, to interface LED.

Note : Source current is current to be sourced i.e. given or provided Sink current is the current to be sinked i.e. connected to ground or given a path to ground. Every microcontroller has a better current sinking capability than its current sourcing capability.

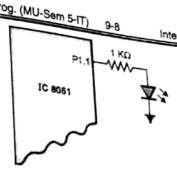


Fig. 9.3.2 : LED driven by 8051 port pin (Method 2)

Example 9.3.1 : Design a 8051 based system to blink a LED at a frequency of 1Hz. Also write the corresponding assembly program.

Solution :

Part a) Design

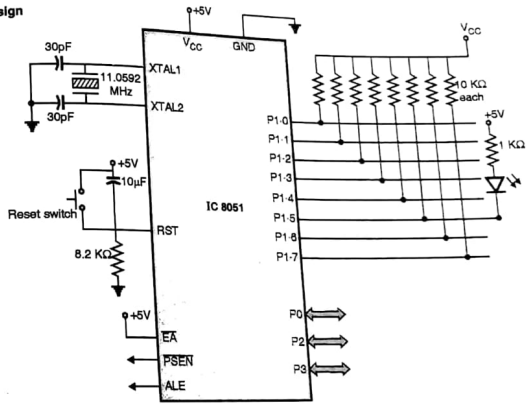


Fig. P. 9.3.1 : Interface diagram

Part b) Program

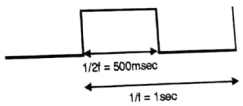


Fig. P. 9.3.1(a) : Required waveform

We need a delay of 500 msec as seen from the Fig. P. 9.3.1(a). The output should be logic '1' for 500 msec and should to logic '0' for 500 msec.

Crystal frequency = 11.0592 MHz

$\therefore$ 1 machine cycle = 1.085 µsec.

$\therefore$ for delay of 500 msec, no of machine cycles = $\frac{500\ msec}{1.085\ \mu sec} \cong (460829)_{10} = (7081D)_{16}$

But the 16 bit timer of 8051 can have a maximum of 16 bit count i.e. $(FFFF)_{16}$

Hence we will design a system with 50 msec delay and execute it 10 times to get a delay of 500 msec.

For delay of 50 msec no of machine cycle = $\frac{50\ msec}{1.085\ \mu sec} \cong (46083)_{16} = (B403)_{16}$

$\therefore$ Counter is to be initialized as $(FFFF)_{16} - (B403)_{16} = (4BFC)_{16}$

>> Algorithm

(A) Main Program

Step I : Initialize the count to 10, for calling 50msec delay 10 times and hence getting a 500msec delay.
Step II : Set P1.5 pin to logic 1
Step III : Enable timer 0 and global interrupts
Step IV : Initialize TMOD for timer 0 in timer mode 1
Step V : Load TL0 and TH0 with the count calculated
Step VI : Set TR0 bit to run timer 0
Step VII : Do nothing. Wait for interrupt.

(B) Timer 0 ISR

Step I : If count has become 0, toggle P1.5 pin and reinitialize count to 10.
Step II : Stop timer 0 (i.e. TR0 = 0) and reload timer 0 with count calculated.
Step III : Set TR0 to restart timer and clear timer 0 overflow flag.
Step IV : Decrement the count

>> Registers value

(1) Interrupt Enable (IE) Register

→ To enable global interrupt and Timer 1 interrupt.

D7	D6	D5	D4	D3	D2	D1	D0
EA	-	-	ES	ET1	EX1	ET0	EX0
1	0	0	0	0	0	1	0
			8			2	

$\therefore$ IE = 0x82

(2) Timer Mode (TMOD) Register

→ To initialize Timer / Counter 0 as timer
→ To initialize Timer 0 in mode 1

Timer 1 Timer 0

D7	D6	D5	D4	D3	D2	D1	D0
GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0
0	0	0	0	0	0	0	1
		0				1	

$\therefore$ TMOD = 0x01

(3) Count value = $(4BFC)_{16}$

$\therefore$ TL0 = 0xFC and TH0 = 0x4B

>> Assembly Program

Label	Instruction	Comments
	org 0000H	
	LJMP Start	
	org 000BH	ISR for timer 0.
	MOV A,R5	
	CJNE A,#00,Next	If 10 times 50msec delays over toggle P1.5 pin and reinitialize count to 10.
	CPL P1.5	
	MOV R5,#0AH	
Next :	CLR TR0	Clear Timer 0 run bit.
	MOV TL0,#0FCH	Load the count for a delay of 50msec in TL0 and TH0.
	MOV TH0,#4BH	
	SETB TR0	Set the Timer 0 in run mode.
	CLR TF0	Clear Timer 0 overflow flag.
	DEC R5	Decrement the variable count after every 50msec.
	RETI	
	org 1000H	
Start:	MOV R5,#0AH	Initialise the variable count to 10.

Label	Instruction	Comments
	SETB P1.5	Set P1.5 pin to logic 1.
	MOV IE,#82H	Enable Timer 0 interrupt and global interrupt.
	MOV TMOD,#01H	Initialize timer 0 as timer and in mode 1.
	MOV TH0, 4BH	Load the count for 50msec in the TL1 and TH1 registers.
	MOV TL0, 0FCH	
	SETB TR0	Set Timer 0 in run mode.
here:	SJMP here	Do nothing loop. Wait for interrupts.
	End	

➤➤ **Output**

The Fig. P. 9.3.1(b) shows the output of the above program.

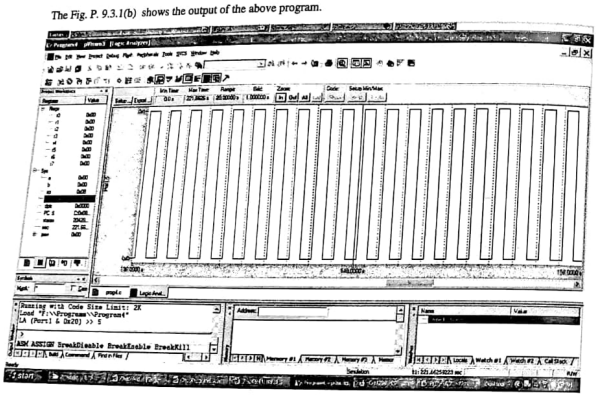


Fig. P. 9.3.1(b) : Output

Example 9.3.2 : Port 1 of 8051 is to be connected to two on-off switches and two LEDs. It is required to sense the status of the switches and indicate it through the LEDs.

Write a program to achieve this task and give the essential interfacing details.

Solution :

P1.0 and P1.1 lines are used to connect the LEDs, while the P1.2 and P1.3 lines of Port 1 are used to connect the switches.

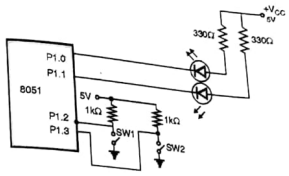


Fig. P. 9.3.2

The status of the switches is sensed by the instruction BIT TEST instruction. The LEDs are driven by the instruction BIT SET.

Program

Label	Instruction	Comments
	MOV P1,#0CH	Initialize P 1.2 and P 1.3 lines as input and switch off the LEDS.
BACK:	JNB P1.2,L1	If bit is set, glow LED1
	CLR P1.0	
	SJMP over	
L1:	SETB P1.0	
over:	JNB 1.3,L2	If bit is set, glow LED2
	CLR P1.1	
	SJMP BACK	Keep polling
L2:	SETB P1.1	
	SJMP BACK	Keep polling

9.3.2 Seven Segment Display (SSD)

– As shown in the Fig. 9.3.3 the seven segment display uses seven LEDs to make any digit. If all the LEDs are on, it shows the digit 8. There are two types SSDs available.

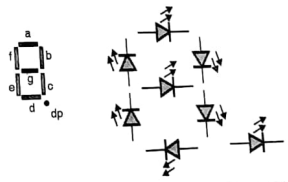


Fig. 9.3.3 : Structure of Seven Segment Display (SSD)

(i) Common cathode i.e. the cathode of all the LEDs are given as a common pin. In this case the anode is connected to the port pins. As already discussed in the section for LEDs, this method requires port pins to source large current. But 8051 cannot source current beyond 2 mA.

(ii) Common anode i.e. the anode of all the LEDs are given as a common pin. In this case the cathode is connected to the port pins. Hence port pins have to sink current of 20 mA.

- Hence we use common anode SSD, and connect it to 8051 as shown in Fig. 9.3.4.

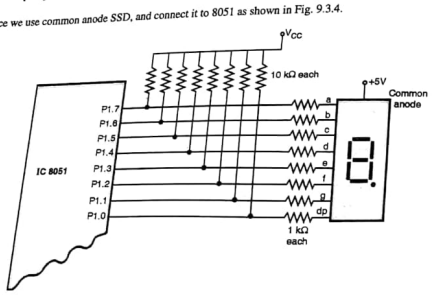


Fig. 9.3.4 : Interfacing common anode SSD to 8051

- The 'dp' stands for decimal point. The Table 9.3.1 illustrates the binary and hexadecimal data given to the pins of SSD, to display different digits.
- A pin should be given logic '0', to switch on the corresponding LED.

Table 9.3.1 : Code for 0 to F given to Common Anode SSD

Digit	a	b	c	d	e	f	g	dp	Hexadecimal form
0	0	0	0	0	0	0	1	1	0x03
1	1	0	0	1	1	1	1	1	0x9F
2	0	0	1	0	0	1	0	1	0x25
3	0	0	0	0	1	1	0	1	0x0D
4	1	0	0	1	1	0	0	1	0x99
5	0	1	0	0	1	0	0	1	0x49
6	0	1	0	0	0	0	0	1	0x41
7	0	0	0	1	1	1	1	1	0x1F
8	0	0	0	0	0	0	0	1	0x01
9	0	0	0	0	1	0	0	1	0x09
A	0	0	0	1	0	0	0	1	0x11

Digit	a	b	c	d	e	f	g	dp	Hexadecimal form
B	1	1	0	0	0	0	0	1	0xC1
C	0	1	1	0	0	0	1	1	0x63
D	1	0	0	0	0	1	0	1	0x85
E	0	1	1	0	0	0	0	1	0x61
F	0	1	1	1	0	0	0	1	0x71

9.3.3 Interfacing Multiple Seven Segment Displays

- To drive multiple seven segment displays, it is difficult to spare separate port for each SSD. For example to interface 4 SSDs we will require 4 ports i.e. all the ports of 8051 will be used.
- In this case, no other device can be interfaced to 8051. Hence we multiplex the SSDs, as shown in the Fig. 9.3.5.
- Now to control these SSDs we utilize one of the characteristic of human eye i.e. persistence of vision. We will give the data for the first SSD (i.e. thousands place) and enable its CA (Common Anode) pin, keeping the CA pin of remaining SSDs disabled.

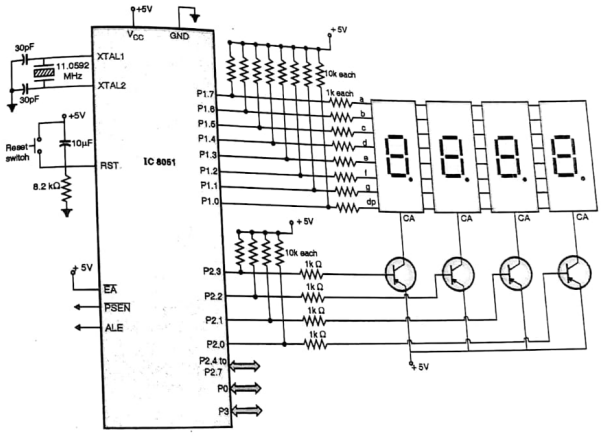


Fig. 9.3.5 : Interfacing multiple SSDs

- A PNP transistor is connected to each of the CA pin. Hence, when we provide a 0V (i.e. logic '0') to the base of the transistor it will be 'ON' and the corresponding SSD will be selected.
- For e.g. when we give P2 = 0x07, the first SSD is enabled, while others are disabled.

... Similarly we send 0 to the data on P1 pins for second SSD and enable its CA pin, keeping the CA pin of others disabled. This can be achieved by giving 0x0B on P3. Similarly for third and fourth SSD.

This entire procedure will be repeated continuously after every 1/50th of a second (i.e. every 20 msec), so that human eye feels all the SSDs to be displaying continuously.

Example 9.3.3 Write a program to display message on an 8 bit 7 segment LED display that is interfaced through Port 1 and Port 3 of 8051.

Solution

Fig. P. 9.3.3 shows a multiplexed 8 digit 7 segment LED display connected in 8051 system using Port 1 and Port 3. Both the ports are used as output ports. Transistors are used to drive LED segments.

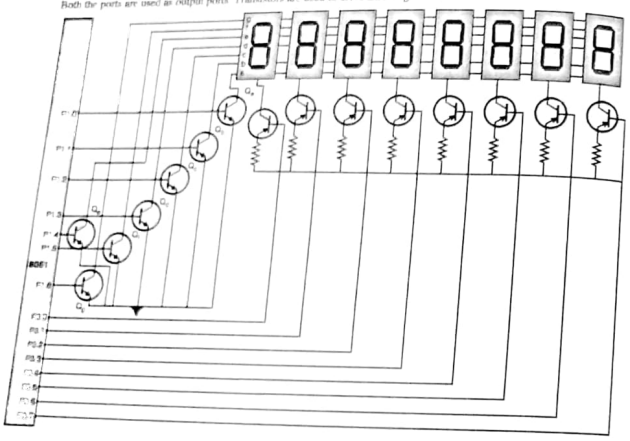


Fig. P. 9.3.3

Subroutine to display message

Label	Instruction	Comments
	MOV R0, #08 H	Initialize counter for 8 digits
	MOV R1, #7FH	Load the select pattern
	MOV DPTR, #3000H	Load starting address of message to be displayed
L1 :	MOV P3, R1	Select digit

Label	Instruction	Comments
	MOVX A, @DPTR	Get send
	MOV P1, A	Send data to Port A for display
	LCALL DELAY	Wait some time
	MOV A, R1	Next select pattern
	RR A	
	MOV R1, A	
	INC DPTR	Increment message pointer
	DJNZ R0, L1	
	RET	

Syllabus Topic : Interfacing LCD

9.3.4 Interfacing Liquid Crystal Display (LCD) to 8051

Q. 9.3.1 Explain various commands associated with LCD module. (Ref. sec. 9.3.4) **(5 Marks)**

Q. 9.3.2 Explain the interfacing of LCD module to 8051. (Ref. sec. 9.3.4) **(8 Marks)**

- LCD displays are widely used because of its low current consumption as compared to SSD. Also that LCD can be used to display any character as it uses a 5 × 7 dot matrix to display.
- For e.g. to display '1' of LCD as shown in Fig. 9.3.6.
- An LCD allows the user to output a specific message making the application more user friendly and attractive.
- LCDs are invaluable for displaying status messages and information while a program is being debug.

Fig. 9.3.6 : Displaying 1 on LCD display of a 5 × 7 dot matrix

- The LCDs generally use a common controller chip, Hitachi 44780 and common connector interface. Due to these factors, the alphanumeric LCDs range in size from 8 characters to 80 characters.
- All the characters are interchangeable without any hardware or software changes. They are arranged in 40 by 2 or 20 by 4 or 10 by 2 or 20 by 1 or 20 by 2.
- The first figure represents the number of characters in each line and second figure represents the number of lines the display has.

- *A typical 16 by 2 (i.e. 16 characters and 2 such lines) LCD looks as shown in Fig. 9.3.7*

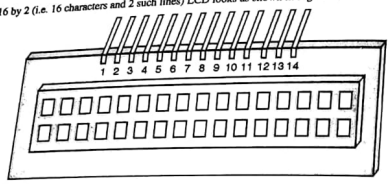


Fig. 9.3.7 : Structures of 16 × 2 LCD

- *Some LCD have their pins on left or on bottom. The functions of the pins of LCD are listed in the Table 9.3.2.*

Table 9.3.2 : Pin description of LSD

Pins	Symbol	Functions
1	V_{ss}	Ground
2	V_{dd} (or V_{cc})	+ 5 V supply
3	V_o	Contract voltage
4	RS	Should be '0' for instruction and '1' for data.
5	$R/\overline{W}$	Should be '0' to write and '1' to read
6	E	Enable display logic
7	D0	Data bus bit 0
8	D1	Data bus bit 1
9	D2	Data bus bit 2
10	D3	Data bus bit 3
11	D4	Data bus bit 4
12	D5	Data bus bit 5
13	D6	Data bus bit 6
14	D7	Data bus bit 7 (Also used as busy pin)

- There are two registers of LCD viz. Instruction command code register and data register.
- The RS pin is used to select one of these register.
- $R/\overline{W}$ pin is used to read or write to LCD.
- The enable pin E, is used to latch the data into the data or command register. When data is supplied, a high-to-low (negative edge) is required for LCD to latch the data.

- The list of commands that can be given to the LCD are as listed in the Table 9.3.3.

Table 9.3.3 : LCD commands

Hex command	Function
0x01	Clear display
0x02	Return cursor to home
0x04	Decrement cursor (i.e. shift cursor left)
0x06	Increment cursor (i.e. shift cursor right)
0x05	Shift display right
0x07	Shift display left
0x08	Display off, cursor off
0x0A	Display off, cursor on
0x0C	Display on, cursor off
0x0E	Display on, cursor on
0x0F	Display on, cursor on and blinking
0x10	Move cursor one position left
0x14	Move cursor one position right
0x18	Shift entire display left
0x1C	Shift entire display right
0x80	Move cursor to beginning of 1st line
0xC0	Move cursor to beginning of 2nd line
0x38	Initialize 2 line display of 5 × 7 matrix

9.3.5 Initialization of LCD

- The following algorithm is required to initialize and write data to LCD.
- Wait 1 second after power up for display to stabilize.
- Initialize the LCD by giving the instruction 0x38 to the command subroutine.
- Wait for 5 msec.
- Issue the command 0x0F to command subroutine for display on, cursor on and cursor blinking.
- Wait for 5 msec.
- Issue the command 0x01 for clearing display to command subroutine.
- Wait for 5 msec.
- Issue the command 0x06 for making LCD in increment mode i.e. cursor should increment after every character is written to command subroutine.

- Wait for 5 msec.
- Issue the command 0x80 (to command subroutine), to position the cursor at 1st line 1st character.
- Issue the data character one by one giving their ASCII values using data subroutine.

☞ **Command subroutine**

- Give the instruction to the port connected to data bus of the LCD.
- Make RS = '0', to indicate instruction.
- Make R/$\overline{W}$ = '0', to indicate write.
- Make E = '1'
- Wait for 120 µsec. } To give a high-to-low pulse on E pin so as
- Make E = '0' to latch the command
- Return.

☞ **Data subroutine**

- Check if LCD is ready by calling ready subroutine.
- Give the data to the port connected to the data bus of the LCD.
- Make RS = '1', to indicate data
- Make R/$\overline{W}$ = '0', to indicate write
- Make E = '1'
- Wait for 120 µsec. } To give a high-to-low pulse on E pin so as
- Make E = '0' to latch the data

Return

☞ **Ready subroutine**

- Make the busy pin (i.e. data bus bit 7) = '1', to program the corresponding port pin of 8051 as input port.
- Make RS = '0' to indicate instruction.
- Make R/$\overline{W}$ = '1', to indicate read.
- Make E = 0
- Make E = 1
- Check if busy pin = '0'. If it is '1', indicates LCD is busy, hence again make E = '0', then E = '1' and check busy pin. Repeat this until busy pin = '0'.
- Return.

Example 9.3.4 : Write a program to display "Engineer" on LCD on size 8 × 1. **(10 Marks)**

Solution :

Label	Instruction	Comments
	org 0000H	
	LJMP START	
	org 0050H	
	dB "Engineer"	data
	org 0100H	
COMMAND :	MOV P2, A	Function to write command on LCD
	CLR P1.0	LCD data lines are connected on Port 2 (assumption)
	CLR P1.1	RS = 0 (assuming RS pin of LCD is connected on P1.0)
	SETB P1.2	RW = 0 (assuming RW pin of LCD is connected on P1.1)
	CLR P1.2	High to low pulse on EN pin (assuming EN pin of LCD is connected on P1.2)
	RET	
	org 0200H	
DISPLAY :	MOV P2, A	function to write data on LCD
	SETB P1.0	
	CLR P1.1	
	SETB P1.2	
	CLR P1.2	
	RET	
	org 0500H	
START :	MOV A, #38H	
	LCALL COMMAND	
	MOV A, #0FH	
	LCALL COMMAND	
	MOV A, #01H	
	LCALL COMMAND	
	MOV A, #06H	
	LCALL COMMAND	
	MOV A, #80H	
	LCALL COMMAND	

Label	Instruction	Comments
	MOV DPTR, #0050H	
	MOV R3, #08H	
next :	MOV A, #00H	
	MOVC A, @A+DPTR	
	LCALL DISPLAY	
	DJNZ R3, next	
here :	SJMP here	

Example 9.3.5 : Interface 16 × 2 LCD display to 8051 microcontroller and display a single character "H" on it. **(10 Marks)**

Solution :

Fig. P. 9.3.5 shows the interfacing of a 16 character × 2 line LCD module with the microcontroller 8051. The data lines are connected to Port 1 of 8051. The control lines RS, R/$\overline{W}$ and E are driven by P 3.2, P 3.3 and P 3.4. The voltage at V_{EE} pin is adjusted by potentiometer to adjust contrast of LCD.

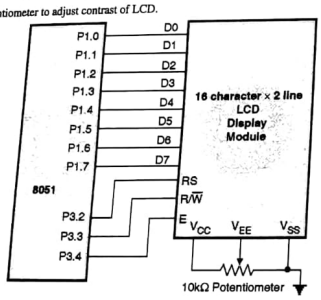


Fig. P. 9.3.5 : Interfacing 16 × 2 LCD to 8051

Let us write assembly language program to display message "H".

>> Program

Label	Instruction	Comments
	MOV 81 H, #30H	Initialize stack pointer
	MOV A, #3CH	Command code for 5 × 10 dots, DL = 8 bits, N = 2 lines
	LCALL COMMAND	
	MOV A, #0EH	Command for setting display cursor on

Label	Instruction	Comments
	LCALL COMMAND	
	MOV A, #01H	Command for clearing display
	LCALL COMMAND	
	MOV A, #06H	Shift cursor right
	LCALL COMMAND	
	MOV A, #C0H	Cursor line 2, position 0
	LCALL COMMAND	
	MOV A, #'H'	Display Letter **H**
	LCALL DISPLAY	
HERE :	SJMP HERE	Loop here after displaying message

Command Routine

Instruction	Comments
LCALL READY	Check if LCD is ready
MOV P1, A	Issue command code
CLR P3.2	Make RS = 0 to issue command
CLR P3.3	Make R/$\overline{W}$ = 0 to enable writing
SETB P3.4	Make E = 1
CLR P3.4	Make E = 0
RET	Return

Display Routine

Instruction	Comments
LCALL READY	Check if LCD is ready
MOV P1, A	Give data
SETB P3.2	RS = 1 to get data
CLR P3.3	R/$\overline{W}$ = 0 to enable writing
SETB P3.4	E = 1
CLR P3.4	E = 0
RET	Return

Ready Routine

Label	Instruction	Comments
	CLR P3.4	Disable display
	CLR P3.2	RS = 0 inorder to access command register
	MOV P1, #0FFH	Configure P1 as input port
	SETB P3.3	R/$\overline{W}$ = 1 to enable writing
L1 :	SETB P3.4	Make E = 1
	JB P1.7, L1	Check D7 bit. If 1, LCD is busy wait till it becomes 0.
	CLR P3.4	Make E = 0 to disable display
	RET	

Example 9.3.6 : *With neat labelled diagram explain interfacing of LCD with 8051. Use Port 0 for data bus and Port 1 for control bus.* **(10 Marks)**

Solution :

Fig. P. 9.3.6 shows the interfacing of a 16 character × 2 line LCD module with the microcontroller 8051. The data lines are connected to Port 0 of 8051. The control lines RS, R/$\overline{W}$ and E are driven by P 1.2, P 1.3 and P 1.4. The voltage at V_{EE} pin is adjusted by potentiometer to adjust contrast of LCD.

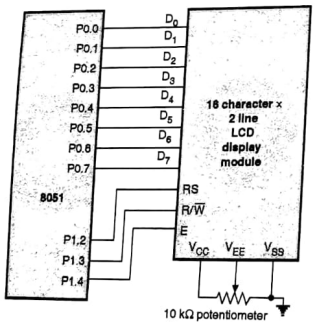


Fig. P. 9.3.6 : Interfacing 16 × 2 LCD to 8051

9.4 Design Examples

Example 9.4.1 : Design a 8051 based system to interface LCD and 4 keys. Write an assembly program to display the last key pressed.

Solution :

Part (a) : Design

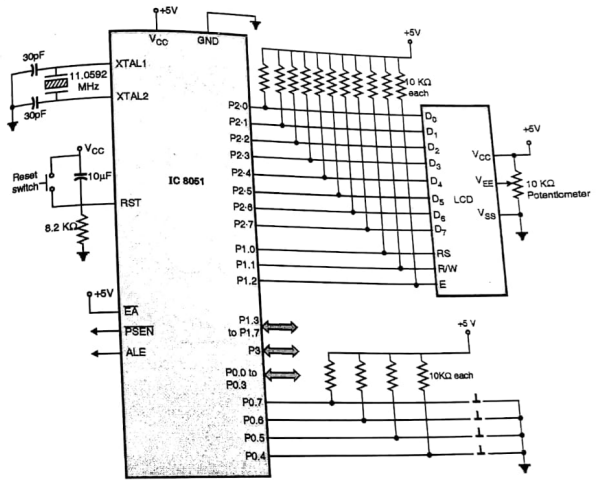


Fig. P. 9.4.1 : Interface diagram

Part (b) : Program

For delay of 20 msec (for debouncing), number of machine cycle

$$= \frac{20\text{msec}}{1.085\mu\text{sec}} \cong (18432)_{10}$$

$$= (4800)_{16}$$

$\therefore$ counter is to be initialized $(FFFF)_{16} - (4800)_{16} = (B7FF)_{16}$

➤➤ **Algorithm**

(A) Main Program

Step I : Initialize the LCD.

Step II : Wait for some software delay after power up for display to stabilize.

Step III : Initialize the LCD by giving the instruction 0x38 to the command subroutine.

Step IV : Wait for small software delay.

Step V : Issue the command 0x0E to command subroutine for display on, cursor on.

Step VI : Wait for small software delay.

Step VII : Issue the command 0x01 for clearing display to command subroutine.

Step VIII : Wait for small software delay.

Step IX : Issue the command 0x06 for making LCD in increment mode i.e. cursor should increment after every character is written to command subroutine.

Step X : Wait for small software delay.

Step XI : Issue the command 0x80 (to command subroutine), to position the cursor at 1st line 1st character.

Step XII : Initialize a port 2 as input port for keys by writing 0xF0 on it.

Step XIII : Initialise the output port for keyboard to all 0's

Step XIV : Initialise TMOD to 0x01 i.e. in mode 1. Timer 0 will be used as key debouncing. Disable all interrupts

Step XV : Take the data from input port for keys into a variable say key

Step XVI : Check if any key is pressed by comparing key with 0xF0. If no key is pressed, go back to step XV, else goto step XVII

Step XVII : Perform key debouncing. This is done by initialising the timer 0 for 20 msec delay, making TR0=1 and waiting for TF0 to become 1. This is use of timer as a delay but without interrupt method. Once TR0 is '1', check for any key pressed again as in step XV and XVI. If the key is still pressed it is a valid key press else it was a bounce.

Step XVIII : Check which key is pressed by comparing the input with the different combinations possible i.e. 0xE0, 0xD0, 0xB0 or 0x70. In each of these combinations only one bit is '0' and the others are '1', based on whichever key is pressed.

Step XIX : According to the key pressed pass the corresponding digit to the data subroutine to display that digit.

(B) Command subroutine

Step I : Give the instruction to the port connected to data bus of the LCD.

Step II : Make RS = '0', to indicate instruction.

Step III : Make $R/\overline{W}$ = '0', to indicate write.

Step IV : Make E = '1'
Step V : Wait for 120 µsec.
Step VI : Make E = '0'
} To give a high-to-low pulse on E pin so as to latch the command

Step VII : Return.

(C) Data subroutine

Step I : Check if LCD is ready by calling ready subroutine.

Step II : Give the data to the port connected to the data bus of the LCD.

Step III : Make RS = '1', to indicate data

Step IV : Make $R/\overline{W}$ = '0', to indicate write

Step V : Make E = '1'
Step VI : Wait for 120 µsec.
Step VII : Make E = '0'
} To give a high-to-low pulse on E pin so as to latch the data

Step VIII : Return

(D) Ready subroutine

Step I : Make the busy pin (i.e. data bus bit 7) = '1', to program the corresponding port pin of 8051 as input port.

Step II : Make RS = '0' to indicate instruction.

Step III : Make $R/\overline{W}$ = '1', to indicate read.

Step IV : Make E = 0

Step V : Make E = 1

Step VI : Check if busy pin = '0'. If it is '1', indicates LCD is busy, hence again make E = '0', then E = '1' and check busy pin. Repeat this until busy pin = '0'.

Step VII : Return.

➤➤ **Registers value**

(1) Interrupt Enable (IE) Register

→ To disable all interrupts.

D7	D6	D5	D4	D3	D2	D1	D0
EA	-	-	ES	ET1	EX1	ET0	EX0
0	0	0	0	0	0	0	0
			0				0

∴ IE = 0x00

(2) Timer Mode (TMOD) Register

→ To initialize Timer / Counter 0 as timer

→ To initialize Timer 0 in mode 1

	TIMER 1				TIMER 0			
D7	D6	D5	D4	D3	D2	D1	D0
GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0
0	0	0	0	0	0	0	1
		0				1	

$\therefore$ TMOD = 0x01

(3) The count for 20 msec as calculated should be initialized in TL1 and TH1

$\therefore$ TL1 = 0xFF

and TH1 = 0xB7

>> Assembly Program

Label	Instruction	Comments
	ORG 0000H	
	LJMP Start	
	ORG 0100H	Function to write the command to LCD
COMMAND:	MOV P2,R3	Write the command on the Port 2 so as to issue it to LCD.
	CLR P1.0	RS=0, Indicates instruction.
	CLR P1.1	RW=0, Indicates Write.
	SETB P1.2	A high to low pulse on EN pin to latch the command
	LCALL DELAY	
	CLR P1.2	
	LCALL DELAY	Wait for some time, software delay
	RET	
	ORG 0200H	
DISPLAY:		Function to write data to LCD
	LCALL READY	Check if the LCD is ready by calling the ready function.
	MOV P2,R3	Write the data on the Port 2 so that it is given to the LCD.
	SETB P1.0	RS=1, Indicates data.
	CLR P1.1	RW=0, Indicates Write.
	SETB P1.2	A high to low pulse on EN pin to latch the command
	LCALL DELAY	
	CLR P1.2	
	LCALL DELAY	Wait for some time, software delay

Label	Instruction	Comments
	RET	
	ORG 0300H	
READY:	SETB P2.7	Function to check if LCD is busy or ready.
	CLR P1.0	Making the P2.7 pin as input pin by writing a '1' on it.
	SETB P1.1	RS=0, Indicates instruction and not data
WAIT:	CLR P1.2	RW=1, Indicates read and not write
	LCALL DELAY	A low to high going pulse on en pin.
	SETB P1.2	
	JB P2.7,WAIT	Wait till the LCD is busy.
	RET	
	ORG 0400H	
DELAY:	MOV R5,#1CH	A subroutine to implement small software delay
REP:	DJNZ R5,REP	
	RET	
	ORG 1000H	
Start:	MOV R4,#0FFH	Wait for some time for LCD to stabilize when power-on.
AGAIN1:	MOV R5,#0FFH	
AGAIN:	DJNZ R5,AGAIN	
	DJNZ R4,AGAIN1	
	MOV R3,#38H	Issue the command to initialize 16 × 2 LCD.
	LCALL COMMAND	
	MOV R3,#0FH	Issue the command for display on, cursor on and cursor blinking.
	LCALL COMMAND	
	MOV R3,#01H	Issue the command to clear display.
	LCALL COMMAND	
	MOV R3,#06H	Issue the command to increment cursor position on every character written.
	LCALL COMMAND	
	MOV R3,#80H	Issue the command to position the cursor on the first position on line 1.
	LCALL COMMAND	
	MOV TMOD,#01H	Initialize timer 0 as timer in mode 1.
	MOV IE,#00H	Disable all interrupts
	MOV P0,#0F0H	Make Port 0 higher bits as input lines
HERE:		Repeat again and again loop.

Label	Instruction	Comments
	MOV A,P0	Take the input from input port i.e. port 0 and mask the lower four bits.
	ANL A,#0F0H	
	CJNE A,#0F0H,OVER	If any key is pressed the input will not be 0xF0. Since if no switch is
	SJMP HERE	pressed all input lines will be connected to Vcc, i.e. logic '1' through the resistor.
OVER:		While if a switch is pressed, the corresponding line will be connected to ground i.e. logic '0'.
	MOV TL0,#0FFH	Debounce the key. Initialise timer 0 registers for 20 msec delay
	MOV TH0,#0B7H	
	SETB TR0	Set TR0 bit to make timer 0 in run mode.
WAIT1:	JNB TF0, WAIT1	Wait for timer 0 overflow flag to overflow i.e. 20 msec delay
CLR TR0;		Clear timer 0 run bit once overflow takes place
	MOV A,P0	Again check for any key pressed.
	ANL A,#0F0H	
	CJNE A,#0F0H,OVER1	If the key is still pressed, it is a valid key press. If key is not found pressed
	SJMP HERE	now, it means it was a key bounce.
OVER1:	MOV R3,#80H	Initialise the LCD to 1st line 1st character to display the key pressed.
	LCALL COMMAND	
	CJNE A,#0E0H, A2	Check which key is pressed
	MOV R3,#'1'	In case if 1st key is pressed, pass the ASCII data '1' to LCD.
	LCALL DISPLAY	
	SJMP HERE	
A2:	CJNE A,#0D0H, A3	In case if 2nd key is pressed, pass the ASCII data '2' to LCD.
	MOV R3,#'2'	
	LCALL DISPLAY	
	SJMP HERE	
A3:	CJNE A,#0B0H, A4	In case if 3rd key is pressed, pass the ASCII data '3' to LCD.
	MOV R3,#'3'	
	LCALL DISPLAY	
	SJMP HERE	
A4:	CJNE A,#70H, A5	In case if 4th key is pressed, pass the ASCII data '4' to LCD.

Label	Instruction	Comments
	MOV R3,#'4'	
	LCALL DISPLAY	
	SJMP HERE	
A5:	SJMP HERE	
	END	

Example 9.4.2 : Design a 8051 based system to interface LCD and 4 × 4 matrix keyboard. Write an Assembly program to display the last key pressed.

Solution :

Part (a) : Design

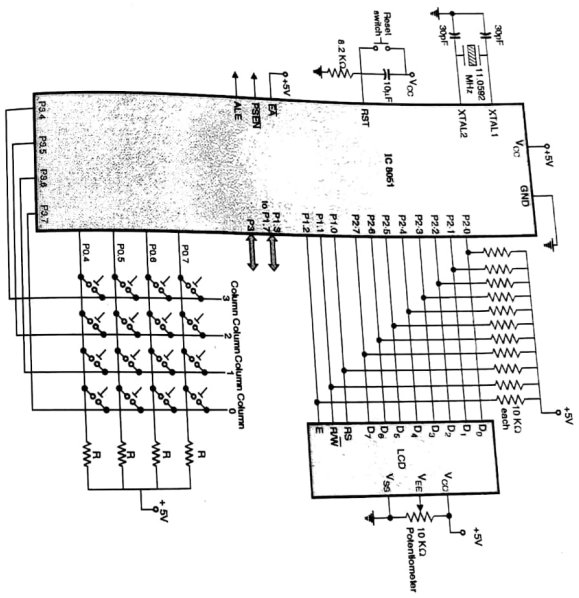


Fig. P. 9.4.2 : Interface diagram

Part (b) : Program

>> Algorithm

(A) Main Program

Step I : Initialize the LCD.

Step II : Wait for some software delay after power up for display to stabilize.

Step III : Initialize the LCD by giving the instruction 0x38 to the command subroutine.

Step IV : Wait for small software delay.

Step V : Issue the command 0x0E to command subroutine for display on, cursor on.

Step VI : Wait for small software delay.

Step VII : Issue the command 0x01 for clearing display to command subroutine.

Step VIII : Wait for small software delay.

Step IX : Issue the command 0x06 for making LCD in increment mode i.e. cursor should increment after every character is written to command subroutine.

Step X : Wait for small software delay.

Step XI : Issue the command 0x80 (to command subroutine), to position the cursor at 1st line 1st character.

Step XII : Initialize a port 2 as a input port for keys by writing 0xF0 on it.

Step XIII : Initialise the output port for keyboard to all 0's

Step XIV : Initialise TMOD to 0x01 i.e. in mode 1. Timer 0 will be used as key debouncing. Disable all interrupts

Step XV : Take the data from input port for keys into a variable say key

Step XVI : Check if any key is pressed by comparing key with 0xF0. If no key is pressed, go back to step XV, else goto step XVII

Step XVII : Perform key debouncing. This is done by initialising the timer 0 for 20msec delay, making TR0=1 and waiting for TF0 to become 1. This is use of timer as a delay but without interrupt method. Once TR0 is '1', check for any key pressed again as in step XV and XVI. If the key is still pressed it is a valid key press else it was a bounce. If bounce goto step XIX.

Step XVIII: Enable one row at a time and check which key is pressed by comparing the input with the different combinations possible i.e. 0xE0, 0xD0, 0xB0 or 0x70. In each of these combinations only one bit is '0' and the others are '1', based on whichever key is pressed.

StepXIX : Repeat the above procedure for all the rows.

Step XX : According to the key pressed pass the corresponding digit to the data subroutine to display that digit.

(B) Command subroutine

Step I : Give the instruction to the port connected to data bus of the LCD.

Step II : Make RS = '0', to indicate instruction.

Step III : Make R/$\overline{W}$ = '0', to indicate write.

Step IV : Make E = '1' } To give a high-to-low
Step V : Wait for 120 µsec. } Pulse on E pin so as
Step VI : Make E = '0' to latch the command

Step VII : Return.

(C) Data subroutine

Step I : Check if LCD is ready by calling ready subroutine.

Step II : Give the data to the port connected to the data bus of the LCD.

Step III : Make RS = '1', to indicate data

Step IV : Make R/$\overline{W}$ = '0', to indicate write

Step V : Make E = '1' } To give a high-to-low
Step VI : Wait for 120 µsec. } Pulse on E pin so as
Step VII : Make E = '0' to latch the data

Step VIII : Return

(D) Ready subroutine

Step I : Make the busy pin (i.e. data bus bit 7) = '1', to program the corresponding port pin of 8051 as input port.

Step II : Make RS = '0' to indicate instruction.

Step III : Make R/$\overline{W}$ = '1', to indicate read.

Step IV : Make E = 0

Step V : Make E = 1

Step VI : Check if busy pin = '0'. If it is '1', indicates LCD is busy, hence again make E = '0', then E = '1' and check busy pin. Repeat this until busy pin = '0'.

Step VII : Return.

>> Registers value

1) Interrupt Enable (IE) Register

→ To disable all interrupts.

D7	D6	D5	D4	D3	D2	D1	D0
EA	-	-	ES	ET1	EX1	ET0	EX0
0	0	0	0	0	0	0	0
			0		0		

∴ IE = 0x00

(2) Timer Mode (TMOD) Register

→ To initialize Timer 0 as timer in mode 1

TIMER 1 TIMER 0

D7	D6	D5	D4	D3	D2	D1	D0
GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0
0	0	0	0	0	0	0	1

0 1

$\therefore$ TMOD = 0x01

(3) Timer 0 will be used for checking of key debounce and hence a delay of 20msec will be required

$\therefore$ TL0 = 0xFF and TH0 = 0xB7

>> Assembly Program

Label	Instruction	Comments
	ORG 0000H	
	LJMP Start	
	ORG 0100H	Function to write the command to LCD
COMMAND:	MOV P2,R3	Write the command on the Port 2 so as to issue it to LCD.
	CLR P1.0	RS=0, Indicates instruction.
	CLR P1.1	RW=0, Indicates Write.
	SETB P1.2	A high to low pulse on en pin to latch the command
	LCALL DELAY	
	CLR P1.2	
	LCALL DELAY	Wait for some time, software delay
	RET	
	ORG 0200H	
DISPLAY:		Function to write data to LCD
	LCALL READY	Check if the LCD is ready by calling the ready function.
	MOV P2,R3	Write the data on the Port 2 so that it is given to the LCD.
	SETB P1.0	RS=1, Indicates data.
	CLR P1.1	RW=0, Indicates Write.
	SETB P1.2	A high to low pulse on en pin to latch the command

Label	Instruction	Comments
	LCALL DELAY	
	CLR P1.2	
	LCALL DELAY	Wait for some time, software delay
	RET	
	ORG 0300H	Function to check if LCD is busy or ready.
READY:	SETB P2.7	Making the P2.7 pin as input pin by writing a '1' on it.
	CLR P1.0	RS=0, Indicates instruction and not data
	SETB P1.1	RW=1, Indicates read and not write
WAIT:	CLR P1.2	A low to high going pulse on en pin.
	LCALL DELAY	
	SETB P1.2	
	JB P2.7,WAIT	Wait till the LCD is busy.
	RET	
	ORG 0400H	
DELAY:		A subroutine to implement small software delay
	MOV R5,#1CH	
REP:	DJNZ R5,REP	
	RET	
	ORG 1000H	
Start:	MOV R4,#0FFH	Wait for some time for LCD to stabilize when power-on.
AGAIN1:	MOV R5,#0FFH	
AGAIN:	DJNZ R5,AGAIN	
	DJNZ R4,AGAIN1	
	MOV R3,#38H	Issue the command to initialize 16×2 LCD.
	LCALL COMMAND	
	MOV R3,#0FH	Issue the command for display on, cursor on and cursor blinking.
	LCALL COMMAND	
	MOV R3,#01H	Issue the command to clear display.
	LCALL COMMAND	
	MOV R3,#06H	Issue the command to increment cursor position on every character written.
	LCALL COMMAND	

Label	Instruction	Comments
	MOV R3,#80H	Issue the command to position the cursor on the first position on line 1.
	LCALL COMMAND	
	MOV TMOD,#01H	Initialize timer 0 as timer in mode 1.
	MOV IE,#00H	Disable all interrupts
	MOV P0,#0F0H	Make Port 0 higher bits as input lines
HERE:		Repeat again and again loop.
	MOV P3,#00H	Make output port P3. All lines as '0'.
	MOV A,P0	Take the input from input port i.e. port 0 and mask the lower four bits.
	ANL A,#0F0H	
	CJNE A,#0F0H,OVER	If any key is pressed the input will not be 0xF0. Since if no switch is pressed all input lines will be connected to V_{cc}, i.e. logic '1' through the resistor. While if a switch is pressed, the corresponding line will be connected to ground i.e. logic '0'.
	SJMP HERE	
OVER:	MOV TL0,#0FFH	Debounce the key. Initialise timer 0 registers for 20 msec delay
	MOV TH0,#0B7H	
	SETB TR0	Set TR0 bit to make timer 0 in run mode.
WAIT1:	JNB TF0, WAIT1	Wait for timer 0 overflow flag to overflow i.e. 20 msec delay
CLR TR0;		Clear timer 0 run bit once overflow takes place
	MOV A,P0	Again check for any key pressed.
	ANL A,#0F0H	
	CJNE A,#0F0H,OVER1	If the key is still pressed, it is a valid key press.
	SJMP HERE	If key is not found pressed now, it means it was a key bounce.
OVER1:	MOV R3,#80H	Initialise the LCD to 1st line 1st character to display the key pressed.
	LCALL COMMAND	
	MOV R6,#00H	Counter for row number
	MOV R7,#00H	Counter for column number
	MOV R5,#0F0H	Register to select the row one by one
NEXTROW:	MOV A,R5	Select the row one by one
	RR A	
	MOV R5,A	

Label	Instruction	Comments
	MOV P3,R5	
	MOV A,P0	Check the key if pressed
	ANL A,#0F0H	
	CJNE A,#0F0H,DONE	
	INC R6	If not this row then increment row number and repeat check for the next row.
	SJMP NEXTROW	
DONE:		If key is found pressed it comes here
	RLC A	Check the column number and accordingly increment R7
	JNC FINISH	
	INC R7	
	SJMP DONE	
FINISH:	MOV B,#04H	Calculate key number pressed as row number multiplied by 4 and added by column number
	MOV A,R6	
	MUL AB	
	ADD A,R7	
	ADD A,#30H	Convert it to ASCII by adding 30H and if it is greater than 9, also add 7 for the characters A to F.
	MOV R3,A	
	CLR C	
	SUBB A,#3AH	
	JC DECIMAL	
	MOV A,R3	
	ADD A,#07H	
	MOV R3,A	
DECIMAL:	LCALL DISPLAY	And display the key pressed on the LCD
	SJMP HERE	
	END	

9.5 Solved Examples

Example 9.5.1 : *A 8051 based temperature controller controls (on/off) a air conditioner so that room temperature is within the range 22°C to 25°C. Assume that signals "high" (to indicate temperature is >25°C) and "low" (to indicate temperature is <22°C) are available. The air conditioner requires a signal "control" to turn on and turn off. The system has a display which continuously displays "01" or "00" depending on whether air conditioner is on or off. Design the above system and write a program the air conditioner and display the message accordingly.* **(20 Marks)**

Solution :

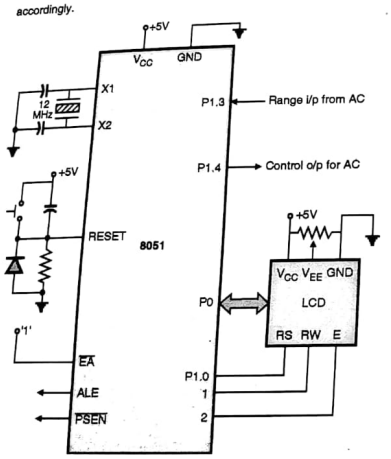


Fig. P. 9.5.1

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	org 0100H	
command :	MOV P0, A	
	CLR P1.0	

Label	Instruction	Comments
	CLR P1.1	
	SETB P1.2	
	MOV R0, #25H	
here1 :	DJNZ R0, here1	
	CLR P1.2	
	RET	
	org 0200H	
data1 :	MOV P0, A	
	SETB P1.0	
	CLR P1.1	
	SETB P1.2	
	MOV R0, #25H	
here2 :	DJNZ R0, here2	
	CLR P1.2	
	RET	
	org 0300H	
main :	SETB P1.3	Set P1.3 as input pin
	CLR P1.4	Commands to initialize LCD
	CALL command	
	MOV A, #0FH	
	CALL command	
	MOV A, #01H	
	CALL command	
	MOV A, # 06H	
	CALL command	
repeat :	MOV A, #80H	Command to set cursor
	CALL command	Position on character 1 of line 1
	JNB P1.4, over	Check range input

Label	Instruction	Comments
	SETB P1.3	If temp > 25°C, switch
	MOV A, #30H	'on' the control output and
	CALL data1	display '01' on LCD
	MOV A, #31H	
	CALL data1.	
	SJMP repeat	
over :	CLR P1.3	If temp < 22°C, switch
	MOV A, #30H	'off' the control output and display '00' on LCD
	CALL data1	
	MOV A, #30H	
	CALL data1	
	SJMP repeat	
	end	

Example 9.5.2 : How timer/counter of 8051 is used as counter? What is the maximum frequency that can be counted by 8051? It is required to count arrival of 10 pulses from a device mounted at the entrance of a room and give visual indication after 10 pulses are counted. Show how can you do this using timer/counter of 8051. **(12 Marks)**

Solution :

Label	Instruction
	org 0000H
	LJMP main
	org 000BH
	SETB P1.0
	RETI
	org 0100H
main :	MOV TMOD, #06H
	MOV IE, #82H
	MOV TL0, #0F5H
	SETB TR0
here :	SJMP here

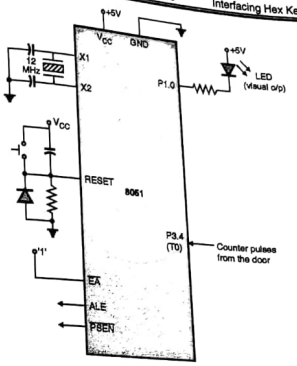


Fig. P. 9.5.2

Syllabus Topic : 8255 PPI

9.6 8255

Q. 9.6.1 Write short on 8255 PPI. (Ref. Sec. 9.6) **(4 Marks)**

- The 8255 is a programmable peripheral interface i.e. PPI 8255.
- It is a general purpose programmable parallel I/O device.
- It contains 3 I/O ports which can be programmed in different modes.
- To program the function to all three I/O ports it contains a register called as ***control register***. The control register defines the function of each I/O port and in which mode they should operate.
- 8255 is a general purpose in nature and provides many facilities for connecting different devices. So it is used frequently in different applications.

9.7 Features of 8255

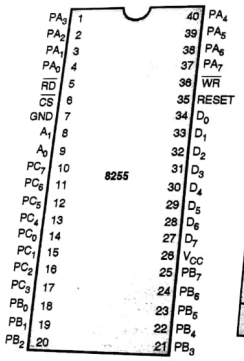
Q. 9.7.1 Write features of 8255. (Ref. Sec. 9.7) **(5 Marks)**

- It is a programmable parallel I/O device.
- It contains 24 programmable I/O pins arranged as 2-8 bit ports and 2-4 bit ports.
- It has 3, 8 bit ports : Port A, Port B and Port C, which are arranged in two groups of 12 pins.

- *Fully compatible with Intel microprocessor families.*
- *TTL compatible.*
- *Direct bit set/reset capability is available for port C.*
- *Improved DC driving capability.*
- *It can operate in 3 modes :*
  - *(a) Mode 0 - Simple I/O*
  - *(b) Mode 1 - Strobed I/O*
  - *(c) Mode 2 - Strobed bi-directional I/O*

9.8 Pin Configuration of 8255

The pin configuration of 8255 programmable peripheral interface is as shown in Fig. 9.8.1.



PA_0–PA_7	I/O	Port A pins
PB_0–PB_7	I/O	Port B pins
PC_0–PC_7	I/O	Port C pins
D_0–D_7	I/O	Data pins
RESET	I	Reset input
$\overline{RD}$	I	Read input
$\overline{WR}$	I	Write input
A_0–A_1	I	Address pins
$\overline{CS}$	I	Chip select
V_{CC}–GND	I	+5 V supply ground

Fig. 9.8.1 : Pin configuration

Symbol **Name and function**

(1) $D_0 - D_7$: **Data bus :** These are 8 bit bi-directional data bus lines, connected to system data bus for data transfer between CPU and 8255.

(2) $\overline{CS}$: **Chip select :**

This is an active LOW input signal used to select 8255 IC.

If $\overline{CS}$ = 0 then 8255 will get selected and take part in data transfer from/to CPU, otherwise 8255 will be in inactive state.

Symbol **Name and function**

(3) $\overline{RD}$: **Read :** This is an active LOW input signal used in co-ordination with other signals to send data to CPU through data lines.

(4) $\overline{WR}$: **Write :** This is an active LOW input signal used in co-ordination with other signals to send data to 8255.

(5) $A_0 - A_1$: **Address lines :** These are input, active HIGH address lines used to distinguish different ports of 8255 such as port A, port B, port C, control register. These lines are internally decoded by 8255 to select ports as follows :

A_1	A_0	Selected port
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control register

(6) RESET : **Reset :** This is an active HIGH input signal used to reset 8255. When 8255 is reset it clears control word register and all ports are set to input mode.

(7) $PA_0 - PA_7$: **Port A pins 0 to 7 :** These are 8 bit bi-directional I/O pins used to send data to peripheral or to read data from peripheral. The contents are transferred to/from Port A.

(8) $PB_0 - PB_7$: **Port B pins 0 to 7 :** These are 8 bit bi-directional I/O pins used same as $PA_0 - PA_7$.

(9) $PC_0 - PC_7$: **Port C pins 0 to 7 :** These are 8 bit bi-directional I/O pins. These lines are divided in 2 sections i.e. PC_0 to PC_3 and PC_4 to PC_7. These two sections can be individually used to transfer 4 bits of data from two separate port C sections i.e. upper port C and lower Port C.

Table 9.8.1 : Port and register select signals summary

A_1	A_0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	Operations
					Input (Read) Operation
0	0	0	1	0	Port A to data bus
0	1	0	1	0	Port B to data bus
1	0	0	1	0	Port C to data bus
					Output (Write) Operation
0	0	1	0	0	Data bus to port A
0	1	1	0	0	Data bus to port B
1	0	1	0	0	Data bus to port C
1	1	1	0	0	Data bus to control register

A_1	A_0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	Operations
					Disable Function
×	×	×	×	1	Data bus tri-stated
1	1	0	1	0	Illegal condition
×	×	1	1	0	Data bus tri-stated

9.9 8255 Functional Block Diagram

The block diagram of 8255 is as shown in Fig. 9.9.1.

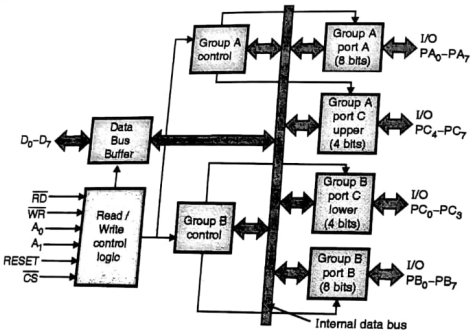


Fig. 9.9.1 : Block diagram of 8255

It contains following blocks

(1) Data bus buffer	(2) Read/Write control logic
(3) Group A and group B control	(4) Port A and port B
(5) Port C	

➔ **1. Data Bus Buffer**

- The 8 bit bi-directional, tristate data bus buffer is used to interface 8255 internal data bus with system data bus.
- The direction of data buffer is decided by read and write control signals.
- When read is activated, it transmits data to the system data bus.
- When write is activated, it receives data from system data bus.

➔ **2. Read / Write Control Logic**

- This block accepts inputs from system control bus and address bus and performs operations as shown in Table 16.3.1.
- The control signals are $\overline{RD}$ and $\overline{WR}$ and address signals used are A_0 and A_1 and $\overline{CS}$.
- The signals $\overline{RD}$ and $\overline{WR}$ are connected to $\overline{IOR}$, $\overline{IOW}$ or $\overline{MEMR}$, $\overline{MEMW}$.
- A_0 and A_1 of 8051 are directly connected to address lines A_0 and A_1 of 8255.
- $\overline{CS}$ is connected to address chip select decoder.
- The 8255 operation / selection is enabled/disabled by $\overline{CS}$ signal.

➔ **3. Group A and Group B Control**

- The 8255 I/O ports are divided into 2 sections Group A (GA) and Group B (GB).
- Group A consists of port A and port C upper.
- Group B consists of port B and port C lower.
- Each group is programmed through software.
- The GA and GB control block receives commands from the R/W control logic to accept bit pattern from CPU.
- GA control will control GA ports and GB control will control GB ports.
- The bit pattern given by CPU consists of information (i) To control the operation of GA and GB (ii) The mode in which they should be operated.

➔ **4. Port A and Port B**

- The port A and port B consists of 8 bit bi-directional data output latch/ buffer and 8 bit data input buffer.
- The function of port A and B is decided by control bit pattern available in GA and GB control.
- The function of ports A and B are also dependent on mode of operation.

➔ **5. Port C**

- The port C consists of 8 bit bi-directional data output latch/buffer and 8 bit data input buffer.
- It is divided into 2 sections, port C upper PC_U and port C lower PC_L These two sections can be programmed and used separately as a 4 bit I/O ports.
- The port C function is dependent on mode of operation.
- It can be used as : (i) simple I/O (ii) handshake signals (iii) status signal inputs.
- For handshake signals and status signals it is used in co-ordination with port A and port B.
- The direct bit set/reset capability is provided by port C only.

9.10 8255 Operating Modes

The 8255 IC provides one control word register.

- It is selected when $A_0 = 1$, $A_1 = 1$, $\overline{CS} = 0$ and $\overline{WR} = 0$.

- The read operation is not allowed for control register.
- The bit pattern loaded in control word register specify an I/O function for each port and the mode of operation in which the ports are to be used.
- There are 2 different control word formats which specify 2 basic modes :
  (1) BSR - Bit set reset mode
  (2) I/O mode.
- The two basic modes are selected by D_7 bit of control register. When $D_7 = 1$ it is a I/O mode and when $D_7 = 0$; it is a BSR mode.

9.10.1 BSR Mode

- The BSR mode is a port C bit set/reset mode.
- The individual bit of port C can be set or reset by writing control word in the control register.
- The control word format of BSR mode is as shown in Fig. 9.10.1.

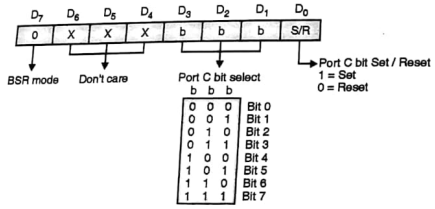


Fig. 9.10.1 : BSR control word format

- The pin of port C is selected using bit select bits [b b b] and set or reset is decided by bit S/R.
- The BSR mode affects only one bit of port C at a time.
- The bit set using BSR mode remains set unless and until you change the bit. So to set any bit of port C, bit pattern is loaded in control register.
- If a BSR mode is selected it will not affect I/O mode.

Example 9.10.1 : Write a set of instructions to perform the following :

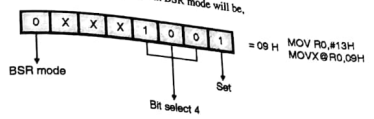
(1) Set bit 4 of port C.

(2) Reset bit 4 of port C.

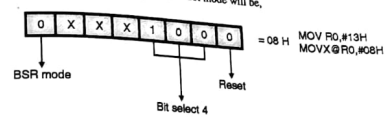
Assume the address of PA = 10 H, PB = 11 H, PC = 12 H and Control reg. = 13 H.

Solution :

(1) To set bit 4 of port C the bit pattern required in BSR mode will be,



(2) To reset bit 4 of port C the bit pattern required in BSR mode will be,



9.10.2 I/O Modes

There are three I/O modes of operation :

(1) Mode 0 - Basic I/O

(2) Mode 1 - Strobed I/O

(3) Mode 2 - Bi-directional I/O

The I/O modes are programmed using control register. The control word format of I/O modes is as shown in Fig. 9.10.2.

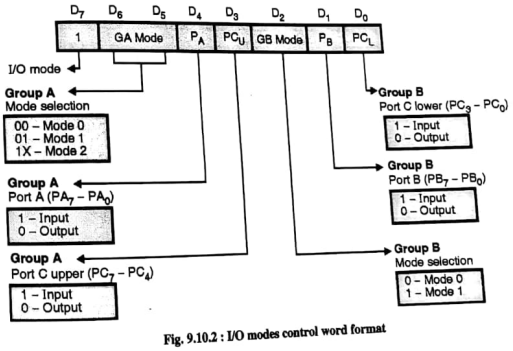


Fig. 9.10.2 : I/O modes control word format

Function of each bit is as follows :

(1) D_7 : When the bit $D_7 = 1$ then I/O mode is selected, if $D_7 = 0$ then BSR mode is selected. The function of bits D_0 to D_6 is dependent on mode (I/O mode or BSR mode).

(2) D_6 and D_5 : In I/O mode the bits D_6 and D_5 specifies the different I/O modes for group A i.e. Mode 0, Mode 1 and Mode 2 for port A and port C upper.

(3) D_4 and D_3 : In I/O mode the bits D_4 and D_3 selects the port function for group A. If these bits = 1 the respective port specified is used as input port. But if bit = 0, the port is used as output port.

(4) D_2 : In I/O mode the bit D_2 specifies the different I/O modes for group B i.e. Mode 0 and Mode 1 for port B and port C lower.

(5) D_1 and D_0 : In I/O mode the bits D_1 and D_0 selects the port function for group B. If these bits = 1 the respective port specified is used as input port. But if bit = 0, the port is used as output port.

From the above explanation you can observe that all the 3 modes i.e. Mode 0, Mode 1 and Mode 2 are only for group A ports, but for group B only 2 modes i.e. Mode 0 and Mode 1 are provided.

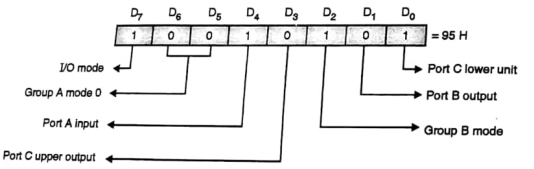
When 8255 is reset, it will clear control word register contents and all the ports are set to input mode. The ports of 8255 can be programmed for other modes by sending appropriate bit pattern to control register.

Example 9.10.2 : Write a set of instructions to perform the following :

(1) Initialise port A as input, port B as output, port C upper as output and port C lower as input.

(2) Use Mode 0 for group A and Mode 1 for group B.

Solution : The control word format required will be,



The initialization instructions are :

```
MOV R0, #13H
MOVX @R0, #95H
```

9.11 I/O Operating Modes

- The 8255 can operate in basic 3 I/O modes; mode 0, mode 1 and mode 2. Now we will see details of 8255 modes.

9.11.1 Mode 0 (Simple Input / Output Mode)

- In Mode 0, all ports i.e. port A, port B, port C provide simple input or output operation separately.
- The data is simply read from a port or it is simply written to a port. In Mode 0 there is no restriction between function of ports.
- The port C, 2 sections port C upper and port C lower can be individually programmed as 4 bit ports.

Features of Mode 0

- Two 8 bits ports P_A and P_B and two 4 bit ports PC_L and PC_U.
- All the ports can be separately programmed as input or output.
- If the port is programmed as output, the outputs are latched.
- If the port is programmed as input, the inputs are buffered.
- As 4 ports are to be used, 16 different I/O configurations are possible.
- No facility for interrupt driven I/O.

Mode 0 - Input mode

- The 8255 is initialized in input mode by using control register.
- When CPU wants to read data from an input port, the CPU will first send address of port on address lines so $\overline{CS}$, A_0 and A_1 will select the appropriate port. After selecting a port CPU will send a control signal $\overline{RD}$ to read data from external peripheral through port and data bus.
- The timing waveform for Mode 0 input mode is as shown in Fig. 9.11.1.

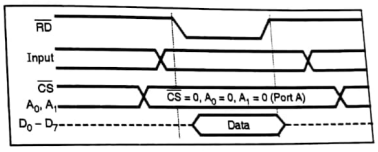


Fig. 9.11.1 : Timing waveforms for mode 0 input mode

Mode 0 - Output mode

- The 8255 is initialized in output mode by using control register.
- When CPU wants to send data to an output port, the CPU will first send address of port on address lines so $\overline{CS}$, A_0 and A_1 will select the appropriate port. After selecting a port CPU will send data and control signal $\overline{WR}$ to write data to port through data bus.
- As the port is in output mode the contents will get latched in the port.

– The timing waveform for Mode 0 output mode is as shown in Fig. 9.11.2.

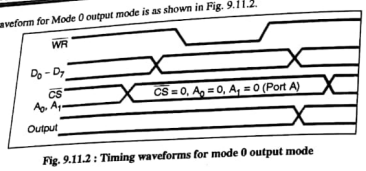


Fig. 9.11.2 : Timing waveforms for mode 0 output mode

9.11.2 Mode 1 (Strobed I/O)

- In mode 1, group A and/or group B can be used.
- Each group consists of 8 bit latched input or output port and 3 bits of port C.
- When port A is used in Mode 1, port C upper bits are used to control the port A.
- When port B is used in Mode 1, port C lower bits are used to control the port B.
- When P_A is used as input in Mode 1 the bits PC_3, PC4 and PC_5 are used for control.
- When P_A is used as output in Mode 1, the bits PC_3, PC_6 and PC_7 are used for control.
- When P_B is used in Mode 1 the bits PC_0, PC_1 and PC_2 are used for control in both input and output mode.
- In all, if both P_A and P_B are programmed in Mode 1, it uses 6 port C bits to control and remaining 2 bits are not used for Mode 1.
- The 2 unused bits can be used as simple I/O. The function of 2 unused bits is decided by bit D_3 of the control word. If $D_3 = 1$ the bits are used as input pins and if $D_3 = 0$, the bits are used as output pins.
- The Mode 1 provides facility to transfer data to/from an external peripheral to 8255 using handshake signals.
- When an external device wants to send data to 8051 through 8255, it will send data to port and inform 8255 about it.
- The data is available in 8255 and next data should not be sent by external device so 8255 will generate a signal to indicate data present. In addition to this 8255 will inform 8051 about availability of data.
- When 8051 gets this signal, 8051 will read data from 8255 port. Now the 8255 is ready to accept next data byte so it will remove the signal of data present. If next data byte is available in peripheral device, then it will repeat the above process.
- The above function of handshake signals can also be performed using Mode 0 but in that case 8051 has to read the status of peripheral, check the status and then take decision on that. This process will waste valuable time of processor.

Features of Mode 1

- Two 8 bit ports A and B function as I/O ports.
- Each port uses 3 lines from port C as handshake signals.
- The remaining lines of port C can be used as simple I/O.
- The input and output data are latched.
- Port C handshake signals can be used to interrupt the CPU i.e. interrupt logic is supported by 8255 to transfer data from 8255 to CPU.

Mode 1 - Input Mode

- In Mode 1 input mode port A uses PC_3, PC_4, PC_5 and port B uses PC_0, PC_1, PC_2 signals as handshake signals.
- The port A, port B and handshake signals are interfaced with peripheral as shown in Fig. 9.11.3. The aim of interfacing is to transfer data from peripheral to CPU through 8255. The Fig. 9.11.3 shows interfacing I/P port to PA, similarly it can be done with PB.
- The different handshake signals used are $\overline{STB}$, IBF and INTR. The internal organisation of these signals is as shown in Fig. 9.11.4.

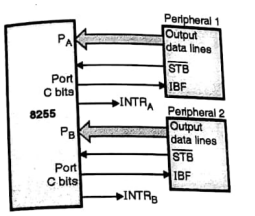


Fig. 9.11.3 : Mode 1 : Input mode interfacing

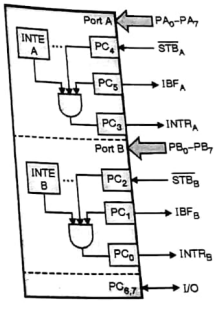


Fig. 9.11.4 : P_A, P_B and P_C in mode 1 input mode

(1) $\overline{STB}$ (Strobe Input)

- This is active low input signal for 8255.
- If IBF signal is not present this signal is generated by peripheral to indicate that it has transmitted data or written data to input port.
- When 8255 gets data byte with $\overline{STB}$ signal the input buffer contains a data byte so 8255 will generate IBF signal.

(2) IBF (Input buffer full)

- This is an active high output signal generated by 8255.
- This signal is generated by 8255 in response to $\overline{STB}$ signal to give acknowledge to peripheral device.
- The input buffer full signal indicates that a data byte is present in input port latch.
- This signal is checked by peripheral device to take decision, for sending next byte of data.
- If IBF signal is active the peripheral will not send next data. IBF is reset when CPU reads input port.

(3) INTR (Interrupt request)

- This is an active high output signal given by 8255.
- If interrupt driven I/O is used the INTR signal is used to interrupt CPU.
- The INTR signal is conditioned by $\overline{STB}$, IBF and INTE signals. If $\overline{STB}$ = 1, IBF = 1 and INTE = 1 then INTR = 1, for all other combinations INTR = 0. The INTR signal is used to indicate CPU to read input port. The falling edge of $\overline{RD}$ will reset INTR.

(4) INTE (Interrupt enable)

- This is an internal flip-flop used to enable or disable interrupt signal.
- If INTE flip-flop is set the interrupt will be generated depending on $\overline{STB}$ and IBF signals.
- The two flip-flops $INTE_A$ and $INTE_B$ are used for group A and group B separately.
- These INTE flip-flops are set/reset by using BSR mode only. The $INTE_A$ is set/reset through PC_4 bit and $INTE_B$ is set/reset through PC_2 bit.
- If interrupt driven I/O is used for data transfer then INTE bit must be set, which will be used to generate INTR signal.
- The INTR logical equations will be,

$$INTR_A = INTE_A \cdot \overline{STB}_A \cdot IBF_A$$

$$INTR_B = INTE_B \cdot \overline{STB}_B \cdot IBF_B$$

- The timing diagram of control signals in Mode 1 input mode is as shown in Fig. 9.11.5.

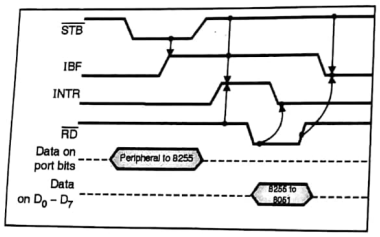


Fig. 9.11.5 : Timing diagram of mode 1 input mode

(I) Interrupt driven I/P : In this case the INTR signal is connected as interrupt input to 8051. The sequence of events will be as follows :

- The data is transferred by peripheral to 8255 and $\overline{STB}$ signal is made low.
- When $\overline{STB}$ signal is received by 8255. 8255 generates a signal IBF.
- When IBF is generated the condition of $\overline{STB}$, IBF and INTE is satisfied to generate INTR signal.

- In response to INTR signal, 8051 reads data from 8255 input port.
- The read operation will reset IBF signal and INTR signal.
- When the peripheral wants to send next data byte it will check IBF signal. If it is low it will repeat the above steps to transfer next data byte.

(II) Status driven I/P

- In Mode 1 of 8255 the port C is used as status word. In this method the INTR signal of 8255 is not used to interrupt 8051. INTE bit will be reset. The service to 8255 is given by polling the status register. The sequence of events will be as follows :
- CPU will read port C of 8255.
- It will check IBF signal if it is high then a data is read from port.
- If IBF is low the CPU will go on reading and checking the signal IBF.
- The port C is used as status word and its definitions will be as shown in Fig. 9.11.6.

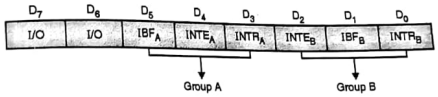


Fig. 9.11.6 : Port C definitions in mode 1 input mode

Mode 1 - Output Mode

- In Mode 1 output mode port A uses PC_3, PC_6 and PC_7 and port B uses PC_0, PC_1, PC_2 signals as handshake signals.

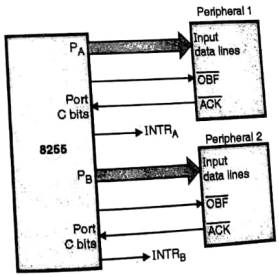


Fig. 9.11.7 : Mode 1 : Output mode interfacing

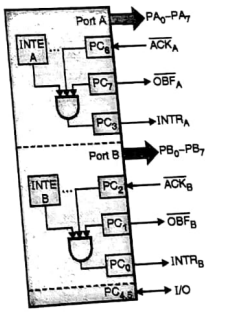


Fig. 9.11.8 : P_A, P_B and P_C in mode 1 output mode

- The port A, port B and handshake signals are interfaced with peripheral as shown in Fig. 9.11.7. The aim of interfacing is to transfer data from CPU to peripheral through 8255.
- The different handshake signals used are $\overline{OBF}$, $\overline{ACK}$ and INTR the internal organisation of these signals is as shown in Fig. 9.11.8.
- The function of these handshake signals is as follows :

(1) $\overline{OBF}$ (Output buffer full)

- This is an active low output signal generated by 8255.
- When CPU writes data to output port, the data is available in output port. It is indicated by giving $\overline{OBF}$ signal. This signal is used to indicate, that new data is ready to be read. In response to $\overline{OBF}$ signal peripheral reads data byte from output port and acknowledges it by $\overline{ACK}$ signal. So after receiving $\overline{ACK}$ signal, the 8255 removes $\overline{OBF}$ signal.

(2) $\overline{ACK}$ (Acknowledge)

- This is an active low input signal for 8255.
- The peripheral reads the data byte and gives acknowledge to 8255. The acknowledge signal indicates that new data byte can be loaded in output buffer.

(3) INTR (Interrupt request)

- This is an active high output signal given by 8255.
- The INTR signal is used to interrupt CPU, if interrupt driven I/O system is to be used.
- The INTR signal is conditioned by $\overline{OBF}$, $\overline{ACK}$ and INTE signals. If $\overline{OBF}$ = 1, $\overline{ACK}$ = 1 and INTE = 1 then INTR = 1, for all other combinations INTR = 0.
- The INTR = 1 condition specifies that output buffer is empty, peripheral is not reading data and interrupt system is enabled. In response to INTR signal CPU writes data to 8255 output port.
- The above steps for data transfer will be repeated.
- When a data is written to output port it makes $\overline{OBF}$ signal LOW and resets INTR signal. Again when peripheral reads data, INTR signal is set.

(4) INTE (Interrupt enable)

- This is an internal flip-flop used to enable or disable interrupt signal.
- If INTE flip-flop is set, the interrupt will be generated depending on $\overline{OBF}$ and $\overline{ACK}$ signals.
- The two flip-flops $INTE_A$ and $INTE_B$ are used for group A and group B separately.
- These INTE flip-flops are set/reset by using BSR mode only.
- The $INTE_A$ is set/reset through PC_6 bit and $INTE_B$ is set/reset through PC_2 bit.
- If interrupt driven I/O is used, for data transfer, then INTE bit must be set, which will be used to **generate** INTR signal.

- The INTR logical equations will be,

$$INTR_A = INTEA \cdot \overline{ACK}_A \cdot \overline{OBF}_A; \quad INTR_B = INTE_B \cdot \overline{ACK}_B \cdot \overline{OBF}_B$$

- The timing diagram of control signals in Mode 1 output mode is as shown in Fig. 9.11.9.

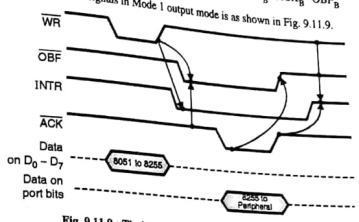


Fig. 9.11.9 : Timing diagram of mode 1 output mode

(I) Interrupt driven output

In this case, the INTR signal is connected as interrupt input to 8051. The sequence of events will be as follows :

- The data is transferred by CPU to 8255 output port, by using $\overline{WR}$ signal.
- The $\overline{OBF}$ will go LOW to indicate data is available in output port.
- When peripheral detects $\overline{OBF}$ signal, it reads data from output port and acknowledge it by giving $\overline{ACK}$ signal.
- The acknowledge from peripheral will make $\overline{OBF}$ signal HIGH and condition of IBF, ACK and INTE is satisfied to generate INTR signal.
- In response to INTR, CPU writes next data to output port and above steps for data transfer will get repeated.

(II) Status driven output

In Mode 1 of 8255, the port C is used as status word. In this method, the INTR signal of 8255 is not used to interrupt 8051. INTE bit will be reset. The service to 8255 is given by polling the status register. The sequence of events will be as follows :

- CPU will read port C of 8255.
- It will check $\overline{OBF}$ signal if it is high then data is transferred to output port.
- If $\overline{OBF}$ signal is low the CPU will not write data to output port and will go on reading and checking the $\overline{OBF}$ signal.
- The port C is used as status word and its definitions will be as shown in Fig. 9.11.10.

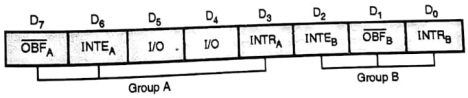


Fig. 9.11.10 : Port C definitions in mode 1 output mode

9.11.3 Mode 2 - Strobed Bi-directional I/O

- In this mode group A is used as input and output i.e. for transmitting and receiving data from peripheral through 8255 as shown in Fig. 9.11.11.
- The transfer of data is achieved by port C handshake signals. The group B can be in Mode 0 or Mode 1.
- The bi-directional data is transferred through port A so it consists of input and output latch.
- The Mode 2 is combination of Mode 1 input and output both at a time to port A.
- The interrupt signals of input and output mode are combined to generate common interrupt signal to CPU. The internal organization of these signals is as shown in Fig. 9.11.12.

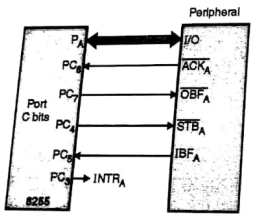


Fig. 9.11.11 : Mode 2 interfacing

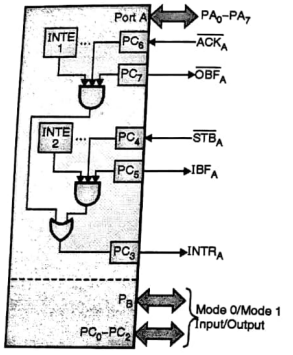


Fig. 9.11.12 : P_A, P_B and P_C in mode 2

- The different handshake signals used are $\overline{OBF_A}$, $\overline{ACK_A}$, $\overline{STB_A}$, IBF_A and $INTR_A$.2 handshake signals are used for output operation, 2 are used for input operation and one is common to both.

Output operation

$\overline{OBF}$ (Output buffer full) : This is an active low output signal generated by 8255. When CPU writes data to output port 8255 will enable $\overline{OBF}$ signal to indicate peripheral that data is available in output buffer.

$\overline{ACK}$ (Acknowledge) : This is an active low input signal for 8255. When the peripheral detects $\overline{OBF}$ signal, it reads data from 8255 port and makes $\overline{ACK}$ = 0 and. The ACK signal is used to acknowledge 8255 that data is read from port so 8255 will remove $\overline{OBF}$ signal to indicate output buffer is empty.

Input operation

$\overline{STB}$ (strobe) : This is an active low input signal. When the peripheral writes data to input buffer, it generates a signal $\overline{STB}$ to indicate 8255 that it has written data.

IBF (Input buffer full)

When data is available in input buffer 8255 will enable IBF signal to indicate that data is available in input buffer.

INTR (Interrupt request)

- This is an output signal given by 8255 to request CPU service.
- The INTR is generated in two different conditions input and output.

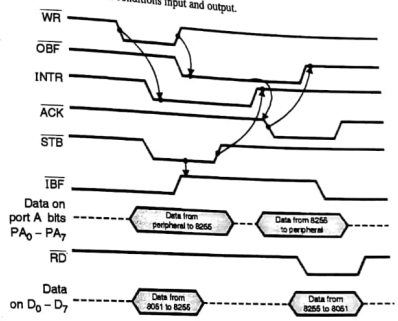


Fig. 9.11.13 : Timing diagram of mode 2

- The interrupt is generated for input mode when IBF = 1, $\overline{STB}$ = 1 and $INTE_1$ = 1 and for output mode when $\overline{OBF}$ = 1, $\overline{ACK}$ = 1 and $INTE_2$ = 1. The $INTE_1$ and $INTE_2$ are set/reset using BSR mode, port C bits used are PC_6 and PC_4 respectively.
- The logical equation will be,

$$INTR_A = INTE_1 \cdot \overline{ACK_A} \cdot \overline{OBF_A} + INTE_2 \cdot \overline{STB_A} \cdot IBF_A$$

The timing diagram of Mode 2 bi-directional data transfer for data transfer from peripheral to CPU and CPU to peripheral are as shown in Fig. 9.11.13.

The Mode 2 also supports both modes of data transfer i.e. Interrupt drive I/O and status driven I/O. The port C is used as status word and its definitions are as follows :

$\overline{OBF_A}$	$INTE_1$	IBF_A	$INTE_2$	$INTR_A$	X	X	X

Q. 1 If PC_7 is set for input mode and P_A is in mode 1 input. Does BSR operation affect the PC_7 bit?

Ans. : When port C pin is used in input mode it does not get affected because of BSR mode.

Q. 2 If P_A and P_B are in mode 0 and P_C lower is being used as output. Does BSR operation affect the PC_1 bit ?

Ans. : When port C pin is being used in output mode and is not being used in handshaking mode the port C bit can change because of BSR operation.

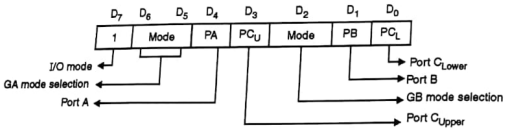
Q. 3 Consider port A and port B has been configured as Mode 1 input. Which bit of port C can be changed by the instruction OUT P_C.

Ans. : When port A and port B are used in handshaking mode they use port C bits as handshake signals. So only bits PC_6 and PC_7 remains unused. If these bits are programmed in output mode they will get changed.

Now let's summarize, three modes of 8255.

Example 9.11.1 : What will be the control word to initialise 8255 in following modes :
port A - Mode 0 - Input , port B - Mode 0 - Output,
port C_{upper} - Mode 0 - Output, port C_{lower} - Mode 0 - Input.

Solution : The control word format is as follows :



PA - Mode 0 - Input : $D_6 = 0$, $D_5 = 0$ and $D_4 = 1$

PC_{upper} - Mode 0 - Output : $D_3 = 0$

PB - Mode 0 - Output : $D_2 = 0$ and $D_1 = 0$

PC_{lower} - Mode 0 - Input : $D_0 = 1$

1	0	0	1	0	0	0	1

= 91 H

9.12 I/O Expansion using 8255 or Interfacing 8255 with 8051

- We know that for interfacing the external memory to 8051, port 0 and port 2 are used as multiplexed address/data bus and higher order address bus.

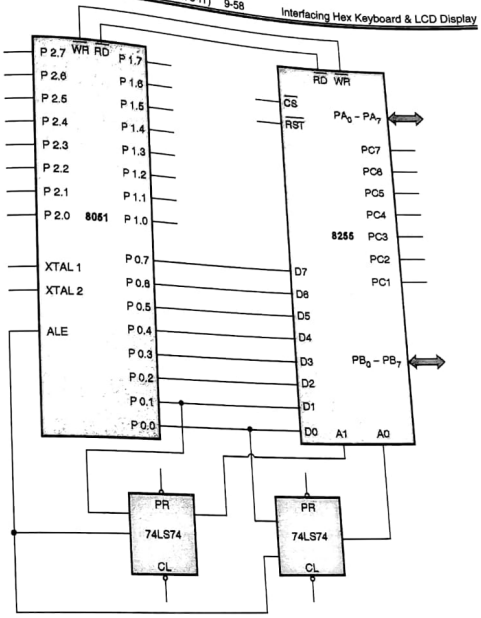


Fig. 9.12.1 : I/O expansion using 8255

- If the circuit requires interrupts or I/Os then port 1 is only available for this purpose. Hence, in such situations it becomes essential to increase the I/O handling capacity. The I/O expansion is possible with the help of 8255 PPI.
- Fig. 9.12.1 shows the expanded I/O ports using 8255.Data bus of 8255 is connected to Port 0.

9.13 Typical MCS 51 based System

A typical MCS 51 based system includes a port expander, an 8 KB EPROM and an 8 KB RAM. Fig. 9.13.1 shows a typical MCS 51 based system.

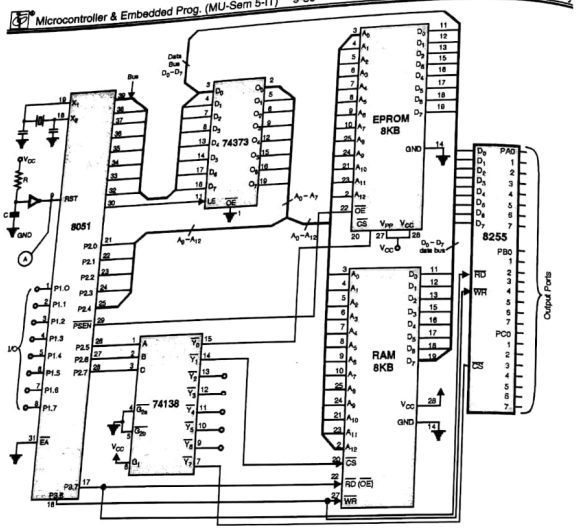


Fig. 9.13.1 : Typical MCS 51 based system

9.14 8255 Interfacing

Program 9.14.1 : Stepper Motor Interfacing using 8255

Solution :

Stepper motor is motor useful for moving things in small increments. The step sizes range from 0.9° to 30°. A stepper motor is stepped from one position to the next by changing the current through the fields in the motor.

Steps for interfacing

Step I : The microcontroller is interfaced with 8255 in the I/O mapped I/O. The port A is used to give steps to the stepper motor. i.e. port A acts as an output port.

Step II : The 8255 provides very less current. Hence, it is not able to drive stepper motor coils. For driving the stepper motor eve require a driver like ULN 200300 or then we can use transistorized buffer circuits like the Darlington pair.

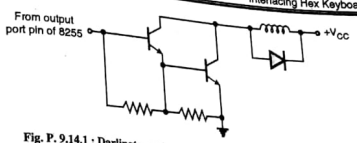


Fig. P. 9.14.1 : Darlington pair transistors to drive stepper motor coils

Step III : The interfacing of stepper motor is shown in Fig. P. 9.14.1(a).

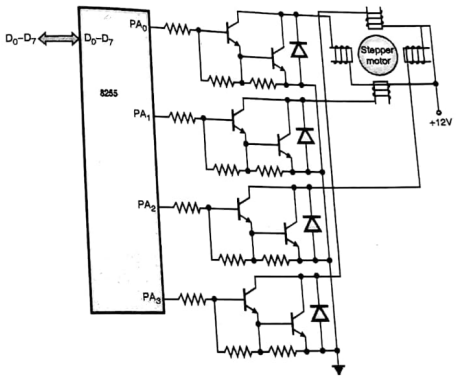


Fig. P. 9.14.1(a) : Stepper motor interface

Step IV : There are two possible step modes.

Mode 1 : Full stepping

In this case the motor moves by 1.8°, to do this 2 bits are changed at a time, bit patterns are as follows :

Table P. 9.14.1(a)

Step	A	B	C	D	
1	1	0	1	0	= 0AH
2	1	0	0	1	= 09H
3	0	1	0	1	= 05H
4	0	1	1	0	= 06H

Reverse ↓ / ↑ Forward

Mode 2 : Half stepping

In this case the motor moves by 0.9°, to do this 1 bit is changed at a time, the bit patterns will be as follows :

Table P. 9.14.1(b)

Step	A	B	C	D	
1	1	0	1	0	= 0A
2	1	0	0	0	= 08
3	1	0	0	1	= 09
4	0	0	0	1	= 01
5	0	1	0	1	= 05
6	0	1	0	0	= 04
7	0	1	1	0	= 06
8	0	0	1	0	= 02
1	1	0	1	0	= 0A

Reverse

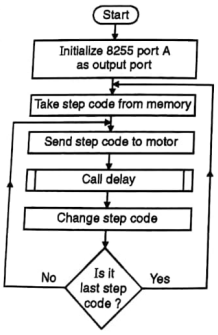
Forward

Step V : The data is stored in memory. Depending on the sequence in which the data is stored the motor will move in forward or reverse direction.

Step VI : Flowchart for full stepping mode

Step VII : Program for full stepping mode.

Label	Instruction	Comment
	.MODEL SMALL	
	.DATA	
	PortA EQU 8000 H	
	Code EQU 30 H	
	.CODE	
	MOV A,#80H	Make port A as output port.
	MOV DPTR,#PortA	Load port number of port A in DPTR
	MOV @DPTR,A	
	MOV R2,64H	Set repetition count to 100
L2 :	MOV R0,#Code	Initialize R0 to code sequence
	MOV A,@R0	A = code
	MOV B,04H	Load the sequence count
L1 :	MOV @DPTR,A	Output excitation code on port A pin
	CALL DELAY	Wait
	INC R0	Increment pointer to point next code
	DEC B	Decrement sequence count
	JNZ L1	
	DEC R2	Decrement repetition count
	JNZ L2	
	END	



Flowchart 9.14.1

Step VIII : Flowchart for half stepping mode is same as that for full stepping mode. The motor moves by 0.9.

Step IX : Program for half stepping mode.

Label	Instruction	Comment
	.MODEL SMALL	
	.Data	
	PortA EQU 8000 H	
	Code1 EQU 30 H	
	.Code	
	MOV A,#80H	Initialize port A of 8255 as output port
	MOV DPTR,#Port A	Load port A number in DPTR
	MOV @DPTR,A	
	MOV R2,#64H	Load R2 with repetition count
L2 :	MOV R0,#code1	R0 = point code sequence
	MOV A,@R0	A = code
	MOV B,#08 H	Load count of sequence in B
L1 :	MOV @DPTR,A	Output excitation code on port A
	CALL DELAY	Call delay
	INC R0	R0 = next code
	DEC B	Decrement sequence count
	JNZ L1	
	DEC R2	Decrement repetition count
	JNZ L2	
	END	

Program 9.14.2 : LCD Interfacing using 8255

Solution :

Fig. P. 9.14.2 shows the interfacing of LCD using 8255.

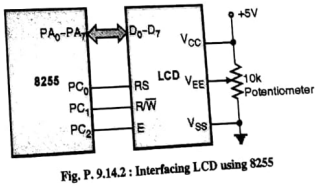


Fig. P. 9.14.2 : Interfacing LCD using 8255

Program 9.14.3 : *Interfacing ADC with 8255*

Solution :

>> Program Statement

Write 8051 ALP to convert an analog signal in the range of 0V to 5V to its corresponding digital signal using successive approximation ADC and dual slope ADC. Find the resolution in both the ADC's and compare the result.

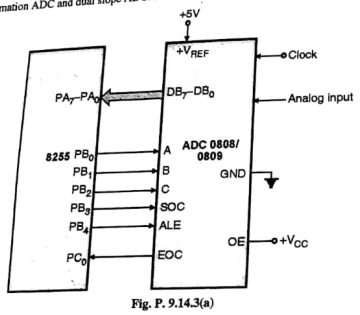


Fig. P. 9.14.3(a)

>> Explanation

- ADC-0809 is an 8 bit successive approximation ADC. This chip has 8 channels along with multiplexer. The channel select has address lines A, B, C. We will use channel 0 as input. Thus, address lines A, B, C will be grounded for channel 0.
- The ALE pin is connected to the clock input.
- At the time of power on the valid channel address is latched at the rising edge of the ALE signal. ADC 0809 has an SOC (start of conversion) pin. A positive going pulse of short duration, is applied to this pin. This pin starts the A/D conversion process.
- The OE should always be high, when data is to be read. After the conversion, EOC is given through PC_7 indicating end of conversion.
- The port A and C are defined in the input mode, whereas port B of 8255 is configured in output mode. The data is read through port A of 8255.
- Positive (d.c.) and negative (d.c.) or (a.c.) voltage is applied as the analog input at channel 0. Hence decoupling capacitors are used to maintain minimum noise level.
- Internal oscillator can be enabled only when A/D conversion is to be done. The oscillator oscillates till the INT.SOC enables pin PB_2 of the 8255.

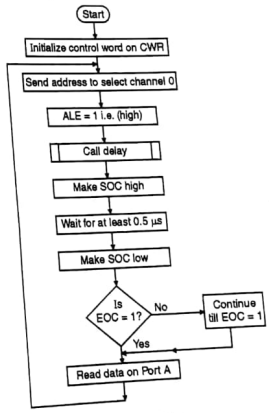
>> Algorithm

Step I : Initialize the data section.
Step II : Initialize A with control word.
Step III : Output contents of A i.e. control word on control register.
Step IV : Load A = 00H.
Step V : Output contents of A on port B, to select channel 0.
Step VI : Send the ALE high.
Step VII : Call some delay i.e. wait for atleast 2.5 µs.
Step VIII : Make SOC high.
Step IX : Wait for atleast 0.5 µs.
Step X : Make SOC low.
Step XI : Check for EOC.
Step XII : If not wait.
Step XIII : Read the digital input available on port A.
Step XIV : Go to step IV.

>> Control word

I/O	Mode	A	P_A	P_{cu}	Mode B	P_B	P_{CL}	
1	0	0	1	1	0	0	1	= 99H

>> Flowchart : Refer Flowchart 9.14.3(a).



Flowchart 9.14.3(a)

>> **Program**

Label	Instruction	Comments
	Port A EQU 0000 H	
	Port B EQU 0002 H	
	Port C EQU 0004 H	
	CWR EQU 0006 H	
	MOV A, #99H	AL=Control word.
	MOV DPTR, #CWR	
	MOV @DPTR, A	Output control word on control word register.
L1 :	MOV A, #00H	A = 00 to select channel 0.
	MOV DPTR, #Port B	
	MOV @DPTR, A	Send address to select channel 0.
	MOV A, #08H	
	MOV @DPTR, A	Latch the given address by sending ALE high.
	Call Delay	Wait for 2.5 μs.
	MOV A, #18H	Make SOC high.
	MOV @DPTR, A	
	NOP	Wait for atleast 0.5 μs.
	MOV A, #08H	
	MOV @DPTR, A	Make SOC low.
	MOV DPTR, #Port C	
CHECK :	MOV A, @ DPTR	Check for EOC.
	ANL A, 01H	
	JZ CHECK	
	MOV DPTR, #Port A	
	MOV A, @DPTR	
	JMP L1	

The observation table will include.

Analog input (V)	Digital equivalent
0	
1	
2	
3	
4	
5	

$\text{Output} = \frac{255 \times \text{voltage}}{5V}$

>> **Using Dual Slope ADC**

IC 7109 is used. It is a 12-bit dual slope A/D converter.

It has RUN / $\overline{\text{HOLD}}$ input and STATUS output, which monitors and control conversion timing. It can operate with upto 30 conversions per second. Fig. P. 9.14.3(b) shows the interfacing diagram.

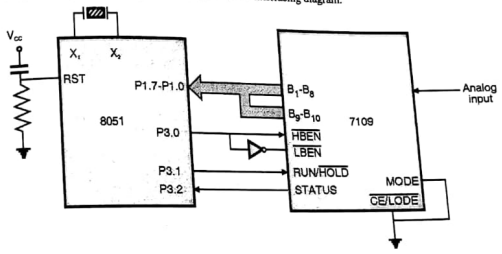


Fig. P. 9.14.3(b) : Interfacing 12 bit ADC 7109 with 8051

>> **Algorithm**

Step I : Initialize the stack pointer .

Step II : Make all port lines as input lines

Step III : Make $\overline{\text{LBEN}}$ enabled.

Step IV : Make Run / $\overline{\text{Hold}}$ signal high.

Step V : Configure P3.2 bit as an input bit

Step VI : Check for status. Is it 0. If not wait till status = 0.

Step VII : Read low order byte on Port 1.

Step VIII : Make $\overline{\text{HBEN}}$ enabled.

Step IX : Read higher nibble.

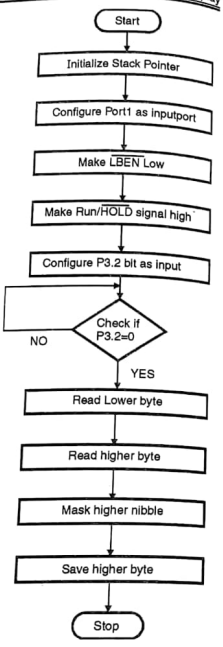
Step X : Save higher nibble

Step XI : Stop.

>> **Flowchart :** Refer Flowchart 9.14.3(b).

>> **Program**

Label	Instruction	Comments
	Port A EQU 0000 H	
	Port B EQU 0001 H	
	Port C EQU 0002 H	
	CWR EQU 0003 H	
	MOV SP, #80 H	Initialize Stack pointer
	MOV P1, #0FF H	Make all port lines as input lines
	SETB P3.0	Make $\overline{LBEN}$ enabled
	SETB P3.1	Make Run /$\overline{Hold}$ signal high.
	SETB P3.2	Configure P3.2 bit as an input bit
BACK:	JB P3.2, BACK	Check for status. Is it 0. If not wait till status = 0.
	MOV R0, P1	Read low order byte on Port1.
	CLR P3.0	Make $\overline{HBEN}$ enabled
	MOV A, P1	Read higher nibble .
	ANL A, #0FH	Mask higher nibble.
	MOV R1, A	Save higher nibble
	END	



Flowchart 9.14.3(b)

After execution of program 12 bit digital data is available on R0 and R1 Registers.

The resolution for both ADC's.

ADC 0809 – 8 bit

ADC 7109 – 12 bit

Program 9.14.4 : Write a C program to toggle all ports of 8255. Assume the address of Port A = 8000 H, Port B = 8001 H, Port C = 8002 H

Solution :

```
#include<reg51.h>
#include<absacc.h>
void main ()
{
   unsigned char a, i;
   a = 0x00;
   while (1)
   {
     XBYTE[0x8000] = a;
     XBYTE[0x8001] = a;
     XBYTE[0x8002] = a;
     for (i=0; i<200 ; i++);          // software delay
     a = ~ a;                         // toggle ports
   }
}
```

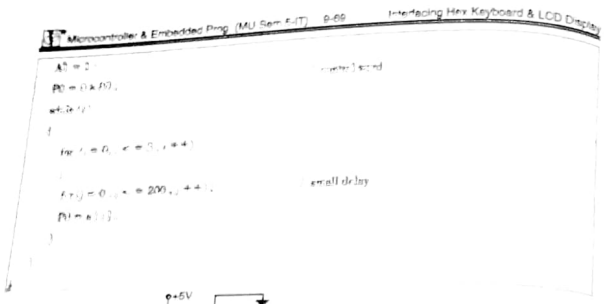
Program 9.14.5 : Interface a 4 pole Stepper motor with 8051 through 8255. Write an assembly language program to rotate it. Explain the control word used.

Solution :

1	GA mode		PA	PCU	GB mode		PB	PCL
1	0	0	0	0	0		0	0

0×8 0

```
# include <reg 51.h>
sbit RD = P3.0 ;
sbit WR = P3.1 ;
sbit A0 = P3.2 ;
sbit A1 = P3.3 ;
void main ( )
{
   char i,j,a [ ] = { 0×0A, 0×09, 0×05, 0×06 } ;
   RD = 1 ;
   WR = 0 ;
   A0 = 1 ;
```



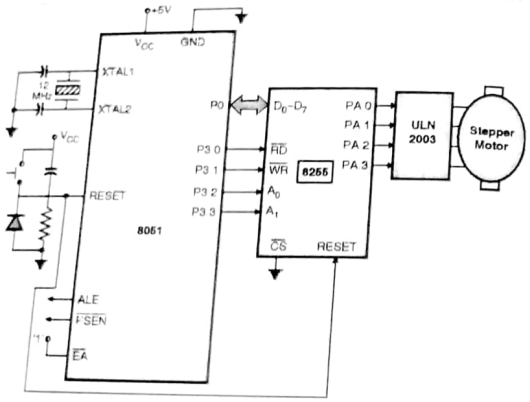


Fig. P. 9.14.5 : Control word of 8255

9.15 Exam Pack (Review Questions)

Q. 1 List the most likely effect if the keyboard program does not accomplish the following :

(i) Debounce keys when pressed down

(ii) Check for valid key code

(iii) Wait for all keys up before ending keyboard routine

(iv) Debounce keys when released **(Refer Sections 9.1 and 9.2)** **(10 Marks)**

Q. 2 What is key debounce ? How it is achieved ? **(Refer Section 9.1.2)** **(4 Marks)**

☞ **Syllabus Topic : KBD Matrix**

Q. 3 Write a short note on following : Interfacing Hex Keyboard with 8051 **(Refer Section 9.2.2)** **(5 Marks)**

☞ **Syllabus Topic : Interfacing LCD**

Q. 4 Explain various commands associated with LCD module. **(Refer Section 9.3.4)** **(5 Marks)**

Q. 5 Explain the interfacing of LCD module to 8051 **(Refer Section 9.3.4)** **(8 Marks)**

Q. 6 Interface 16 × 2 LCD display to 8051 microcontroller and display a single character 'H' on it. **(Refer Example 9.3.5)** **(10 Marks)**

Q. 7 With neat labelled diagram explain interfacing of LCD with 8051. Use Port 0 for data bus and Port 1 for control bus. **(Refer Example 9.3.6)** **(10 Marks)**

Q. 8 A 8051 based temperature controller controls (on/off) a air conditioner so that room temperature is within the range 22°C to 25°C. Assume that signals "high" (to indicate temperature is >25°C) and "low" (to indicate temperature is <22°C) are available. The air conditioner requires a signal "control" to turn on and turn off. The system has a display which continuously displays "01" or "00" depending on whether air conditioner is on or off. Design the above system and write a program the air conditioner and display the message accordingly. **(Refer Example 9.5.1)** **(20 Marks)**

Q. 9 How timer/counter of 8051 is used as counter? What is the maximum frequency that can be counted by 8051? It is required to count arrival of 10 pulses from a device mounted at the entrance of a room and give visual indication after 10 pulses are counted. Show how can you do this using timer/counter of 8051. **(Refer Example 9.5.2)** **(12 Marks)**

☞ **Syllabus Topic : 8255 PPI**

Q. 10 Write short on 8255 PPI. **(Refer Section 9.6)** **(4 Marks)**

Q. 11 Write features of 8255. **(Refer Section 9.7)** **(5 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 10

Interfacing ADC-DAC and Stepper Motor

10.1 Interfacing DAC, ADC and Sensors

- Most of the physical quantities such as temperature, pressure, displacement, vibrations etc. are available in analog form. These quantities are represented accurately in analog form but it is difficult to process, store or transmit the analog signal because noise easily introduces error.
- Hence to reduce these errors it is always better to express these physical quantities in the digital form.
- The digital representation of a signal makes storage possible, processing simpler and transmission easier.
- Therefore A to D conversion is necessary. Now once the processing, transmission etc. is done the signal should be brought back to its analog form, for which the D to A conversion is essential.
- Both ADC and DAC circuits are called as data converters and they are available in the IC form.
- Fig. 10.1.1 shows a A/D and D/A converter application.

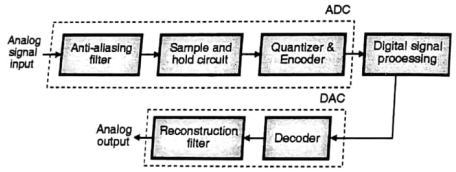


Fig. 10.1.1 : A/D and D/A converter application

As shown in the Fig. 10.1.1 the A/D conversion involves band limiting the signal, sampling the signal, quantizing it and encoding it into a suitable digital format before transmission. Once the signal is transmitted, it is received and converted back to analog form by the decoder and the reconstruction filter.

10.2 D/A Converters

DAC 0808

The DAC 0808 is an 8-bit current output monolithic DAC manufactured by the National semiconductor corporation. It is a 16 - pin IC available in dual in line DIP plastic package. The analog output is available in the form of current I_o. That means I_o is proportional to the 8-bit digital input. The important features of DAC 0808 are as follows.

Features of DAC 0808

Fast settling time	150 nsec. typically
Power supply voltage range	± 4.5 mW at ± 5V
Low power consumption.	33 mW at ± 5V.
High speed multiplying input slew rate	8 mA / μ sec.

Interfaces directly with TTL, DTL and CMOS logic levels.

Pin configuration and functional block diagram

- The pin configuration and functional block diagram of DAC 0808 are as shown in Figs. 10.2.1(a) and (b) respectively.
- The internal block diagram shows that DAC 0808 consists of R-2R ladder along with current switches and reference current amplifier.
- A_1 to A_8 are the 8-digital input lines with A_1 as the most significant bit and A_8 as the least significant bit.
- The analog output is available in the form of current I_o, therefore we need to use an external current to voltage converter if the analog output in the form of voltage is required.
- DAC 0808 requires a dual polarity (±) supply voltage, typically ± 15V, for its operation. The reference voltage can be either positive or negative.
- An external reference voltage should be applied to either V_{REF} (+) or V_{REF} (–) depending on the polarity of reference voltage.

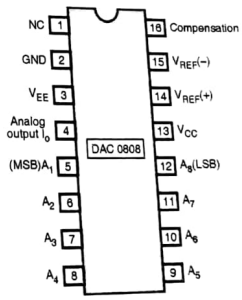


Fig. 10.2.1(a) : Pin configuration of DAC 0808

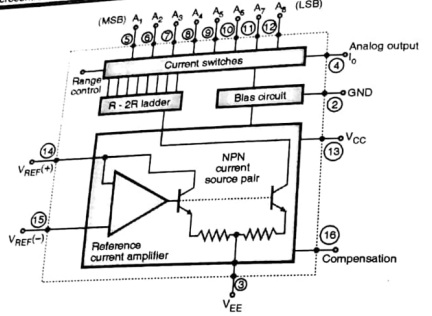


Fig. 10.2.1(b) : Functional block diagram of DAC 0808

10.3 A/D Converters

Q. 10.3.1 List the features of ADC 0808. (Ref. Sec. 10.3) **(3 Marks)**

ADC 0808/0809

- A large number of ADC ICs have been produced by the manufacturing companies like National semiconductors, Motorola, Intersil etc. to meet various demands such as speed of response resolution, compatibility and ease of interfacing with microprocessors etc.
- The National semiconductor produces ADC 0809 which is an 8-bit ADC whereas Intersil produces IC7109. ICL 7109 which is a 12-bit ADC. A serial 8-bit ADC is available i.e. MAX1112. This ADC gives the digital data out in serial form i.e. 1 bit at a time. Let us discuss the ADC 0808/0809 in detail.

Principle of A to D conversion in ADC 0808/0809

- The ADC 0809 operates on the successive approximation technique of A to D conversion.
- It is a CMOS device with 8-analog inputs, an 8 channel multiplexer and microprocessor compatible control logic.
- As the number of bits $n = 8$, it includes a 256 resistor voltage divider, a group of analog switches and a Successive Approximation Register (SAR).
- As there are 8-analog channels, we can connect upto 8 analog inputs to this IC.
- However due to the use of a multiplexer, at a time only one analog input will be converted into an equivalent 8-bit digital output. The analog input channels can be selected using the three address lines A, B and C.

Features of ADC 0808/0809

- Inbuilt 8 analog channels with multiplexer.
- Zero or Full scale adjustment is not required.
- 0 to 5 V input voltage range with a single polarity 5 V supply
- Output is TTL compatible.
- High speed.
- Low conversion time (100 μs).
- High accuracy.
- 8-bit resolution.
- Low power consumption (less than 15 mW).
- Easy to interface with all microprocessors.
- Minimum temperature dependence

Pin Configuration of ADC 0808/0809

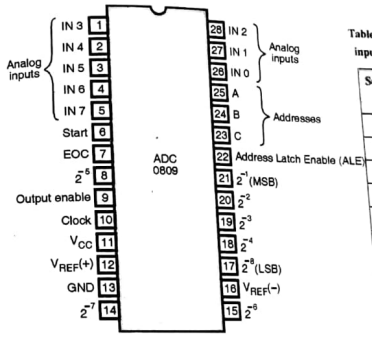


Fig. 10.3.1 : Pin configuration of IC ADC 0808/0809

Table 10.3.1 : Selection of one of the analog inputs using the address lines A, B and C

Selected analog channel	Address		
	C	B	A
IN 0	0	0	0
IN 1	0	0	1
IN 2	0	1	0
IN 3	0	1	1
IN 4	1	0	0
IN 5	1	0	1
IN 6	1	1	0
IN 7	1	1	1

Description

(i) Analog Inputs (IN 0 to IN 7)

Pin numbers 1 to 5 and 26 to 28, designated as IN 0 to IN 7 are the eight analog inputs of this IC. We can connect signals coming from eight different transducers to these inputs. Each one of these inputs will be converted to an 8-bit equivalent digital (binary word). However these inputs are converted into digital form one by one and not all at a time. Hence, one of these eight inputs should be selected for conversion. This selection is done by means of the address pins A, B and C.

(ii) Address Pins A, B, C (Pin 23, 24, 25)

These pins will decide or select one out of the eight analog inputs, for conversion into digital form. For example if CBA = 010 then the "IN 2" is selected and the analog signal at this input is converted to equivalent digital form.

(iii) Reference Voltage [V_{REF} (+) and V_{REF} (–)]

Depending on the desired polarity of the reference voltage, we can connect a positive or negative reference voltage externally to these pins. (2^{-1} to 2^{-8})

(iv) ALE and Output Enable

As shown in the functional block diagram of ADC 0808/0809 (Fig. 10.3.2), the address latch enable (ALE) input is useful in enabling the address latch which stores the address on lines A, B and C. The output enable pin, when activated will make the digital output available on the output pins.

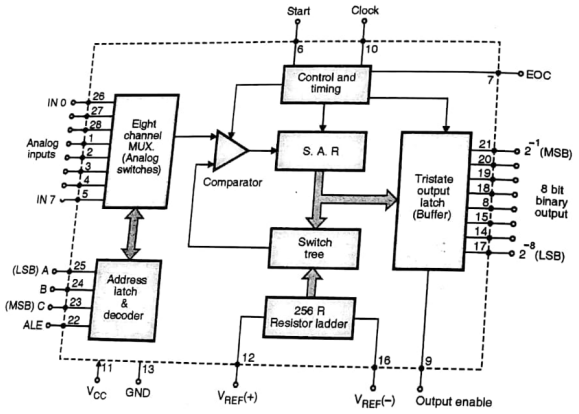


Fig. 10.3.2 : Functional block diagram of IC ADC 0808/0809v) Start and EOC

(v) We have to enable the start input to begin the A to D conversion. A pulse is to be given on start pin to start conversion. The end of conversion is indicated by EOC (End Of Conversion) output.

(vi) Digital Outputs [2^{-1} to 2^{-8}]

The digital output is available to these pins. 2^{-1} represents the MSB and 2^{-8} represents the LSB of digital output.

Syllabus Topic : Interfacing ADC, DAC

10.4 Interfacing ADC and DAC to MCS-51 Family

Q. 10.4.1 Draw and explain the typical interfacing circuit for DAC 0808. (Ref. Sec. 10.4) **(6 Marks)**

Q. 10.4.2 Give a complete scheme to interface an 8-bit ADC to microcontroller 8051. (Ref. Sec. 10.4) **(6 Marks)**

- The interfacing diagram of ADC and DAC with 8051 is as shown in Fig. 10.4.1.

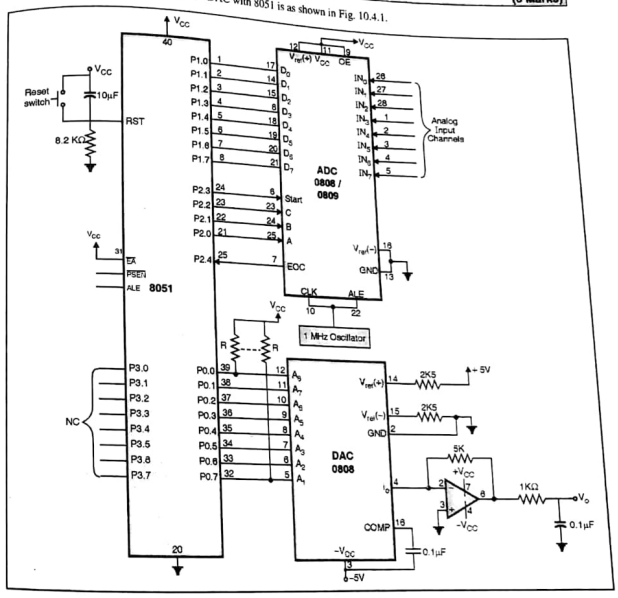


Fig. 10.4.1 : Interfacing ADC and DAC to 8051

- ALE pulse occurs twice in a machine cycle hence its frequency is approximately 2MHz
- ADC requires a clock pulse in the range of 600kHz to 1.2 MHz

- Hence, this 1MHz oscillator can be taken from ALE output of the 8051 by dividing it with 2
- Dividing the ALE pulse frequency by 2 can be done using a D Flip-Flop i.e. IC74LS74
- Also as shown in the Fig. 10.4.1 the pulse may also be taken from some external source.

10.5 ADC 0804

Q. 10.5.1 Draw and explain the interfacing of ADC 0804 to 8051 (Ref. Sec. 10.5) **(8 Marks)**

The ADC 0804, ADC 0801, ADC 0802, ADC 0803 and ADC 0805 are CMOS 8 bit ADCs manufactured by the National Semiconductor corporation. It is a 20-pin IC available in dual is line DTP package. It uses successive approximation technique. The important features of ADC 0804 are as follows :

10.5.1 Features of ADC 0804

(1) Resolution 8 bit.
(2) Conversion time 100 μs.
(3) No zero adjustment required.
(4) On chip clock generator.
(5) Easy interface to all microprocessors.
(6) Operates on single 5 V supply.
(7) The logic inputs and outputs mat MOS and TTL voltage level specifications.

10.5.2 Pin Configuration and Functional Block Diagram

- The pin configuration and functional block diagram of ADC 0804 are as shown in Fig. 10.5.1 and Fig. 10.5.2 respectively.
- As shown in Fig. 10.5.2 the ADC contains a circuit equivalent of the 256 R network.
- The analog switches are sequenced by the successive approximation logic to match the analog difference input voltage $[V_{IN}(+) - V_{IN}(-)]$ to a corresponding tap on the R network.
- The most significant bit is tested first and after 8 comparisons an 8 bit binary code is transferred to the output latch and then an interrupt is asserted.

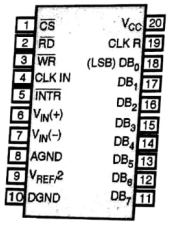


Fig. 10.5.1 : Pin diagram of ADC 0804

10.5.3 Interfacing ADC 0804 to MCS-51 Family

Q. 10.5.2 Interface with 8051 microcontroller Analog to Digital converter ADC 0804. Draw the interface diagram and Read, Write pulses input waveform. (Ref. Sec. 10.5.3) **(10 Marks)**

Fig. 10.5.2 shows the interfacing of ADC 0804 to 8051.

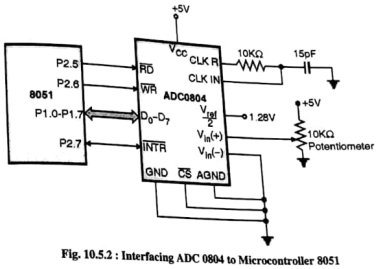


Fig. 10.5.2 : Interfacing ADC 0804 to Microcontroller 8051

10.6 Temperature Sensor LM35

- Temperature is the most-measured process variable in industrial automation. Most commonly, a temperature sensor is used to convert temperature value to an electrical value. Temperature Sensors are the key to read temperatures correctly and to control temperature in industrials applications.
- The LM34 are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Fahrenheit temperature.
- The LM35 are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Celsius (Centigrade) temperature.

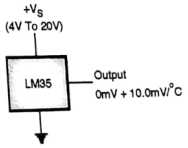


Fig. 10.6.1 : Circuit diagram for the LM35 basic temperature sensor (+2° C to +150° C)

- The LM34/LM35 thus has an advantage over linear temperature sensors calibrated in degrees Kelvin, as the user is not required to subtract a large constant voltage from its output to obtain convenient Fahrenheit scaling.

- LM35 does not require any external calibration or trimming to provide typical accuracies of ±1/4°C at room temperature and ±3/4°C over a full –55 to +150°C temperature range.
- The LM35 is rated to operate over a -55°C to +150°C temperature range.
- As shown in the Fig. 10.6.1 the connection for LM35 is very simple. Also the output scale is linear with a change of 10mV/°C.

10.7 Stepper Motor

Q. 10.7.1 Write short note on : Stepper motor interfacing with 8051. **(7 Marks)**
(Ref. Sec. 10.7)

- A stepper motor is a device that translates electrical pulses to mechanical movement. It is used in applications like robotics, dot matrix printers, disk drives for position control.
- Stepper motors have a permanent magnet rotor surrounded by a stator.
- Generally the stepper motors have four stator windings that are paired with a common center tap as shown in Fig. 10.7.1.
- Such a stepper motor is called as four phase or unipolar stepper motor.

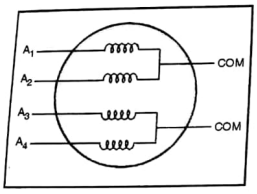


Fig. 10.7.1 : Stator windings configuration

- With the help of center tap the current direction in each of the two coils can be changed when a winding is grounded. This results in a polarity change of stator.
- The stator is responsible for the direction of rotation by the stator poles. The stator poles are determined by the current sent through the wire coils. As the direction of current is changed, the polarity is also changed causing the reverse motion of the stepper motor.
- The stepper motor has 6 leads, 4 leads representing four stator windings and 2 leads for 2 commons for the center tapped leads. On application of power to the stator the rotor rotates.
- There are many sequences for rotation. Each sequence has a different degree of precision.
- Table 10.7.1 shows a 2 phase 4 step stepping sequence.

Table 10.7.1 : 4 step sequence (Full stepping sequence)

Step	Winding A_1	Winding A_2	Windlog A_3	Winding A_4
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1

Counter clockwise

Clockwise

- Although we can start with any sequence, we must continue in proper order. e.g. if we start with step 2 then the sequence of steps is 3,4,1 etc.
- Although we can start with any sequence, we must continue is proper order. e.g. if we start with step 2 then the sequence of steps is 3,4,1 etc.

10.7.1 Step Angle

- Step angle is defined as the minimum degree of rotation associated with a single step.

Table 10.7.2 gives some step angles.

Table 10.7.2 : Stepper motor step angles

Step angle	Steps per revolution
0.72	500
1.8	200
2.0	180
2.5	144
5.0	72
7.5	48
15	24

The total number of steps required to rotate one complete rotation of 360° is called as **steps per revolution.**

$$\text{Steps per second} = \frac{\text{rpm} \times \text{Steps per revolution}}{60}$$

Syllabus Topic : Interfacing Stepper Motor

10.7.2 8051 Interfacing to Stepper Motor

Example 10.7.1 : Draw 8051 connection to stepper motor and code a program to continuously rotate it.

Solution :

Fig P. 10.7.1 shows 8051 connection interfacing to stepper motor. The four leads of the stator winding are controlled by port 1 bits P1.0 – P1.3. The 8051 does not have sufficient current to drive the stepper motor windings. Hence, a driver like ULN 2003 is required to energize the stator.

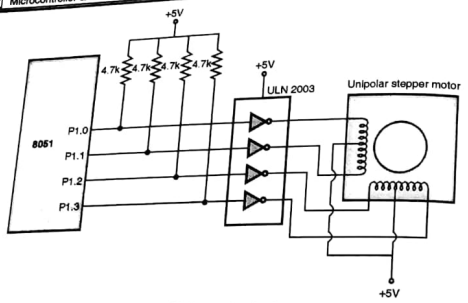


Fig. P. 10.7.1 : 8051 connection to stepper motor

>> Program

Label	Instruction	Comments
	MOV A, #66 H	Load step sequence
L1 :	MOV P1, A	Give sequence to motor
	RR A	Rotate right clockwise
	ACALL DELAY	Wait for sometime
	SJMP L1	Continue doing
DELAY :	MOV R2, #100	
L3 :	MOV R3, #255	
L2 :	DJNZ R3, L2	
	DJNZ R2, L3	
	RET	

10.7.3 Four Step Sequence and 8 Step Sequence

- Table 10.7.1 shows 4 step sequence. As after switching four steps the same two windings will be 'on'.
- After completing four steps, the rotor moves only one tooth pitch. Hence, in a stepper motor with 180 steps per revolution, the rotor has 45 teeth as $4 \times 45 = 180$ steps are required to complete one revolution.
- The minimum step angle in function of number of teeth on rotor.
- The four step sequence is also called as **full stepping** sequence.
- To allow finer resolutions all stepper motors use an 8 step switching sequence. It is also called as **half stepping.**
- Each step is the half of the normal step angle. Table 10.7.3 shows a half stepping sequence.

Table 10.7.3 : Half stepping sequence

Step	Winding A_1	Winding A_2	Winding A_3	Winding A_4	
1	1	0	0	1	Counter clockwise
2	1	0	0	0	
3	1	1	0	0	
4	0	1	0	0	
5	0	1	1	0	
6	0	0	1	0	
7	0	0	1	1	
8	0	0	0	1	Clockwise

10.7.4 Using Transistors as Drivers

Fig. 10.7.2 shows an interface to unipolar stepper motor using transistors.

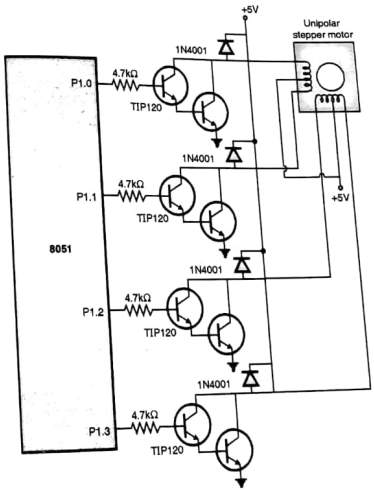


Fig. 10.7.2 : 8051 interfacing using transistor drivers for stepper motor

- Diodes are used to reduce the back EMF spike created when the coils are energized and deenergized in the same manner as the electromechanical relays.
- TIP 120 Darlington transistors can be used to supply higher current to the motors.

10.7.5 Controlling Stepper Motor Via Optoisolator

- The optoisolators are widely used to isolate the stepper motors EMF voltage and keep it from damaging the microcontroller system.
- Fig. 10.7.3 shows stepper motor control using optoisolator

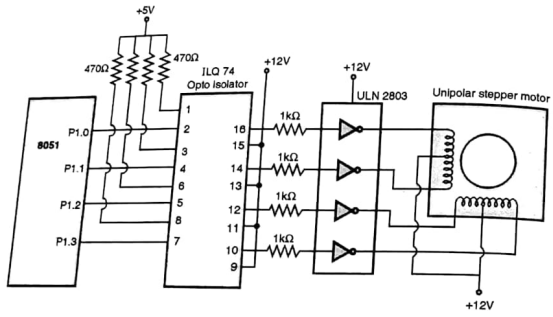


Fig. 10.7.3 : Controlling stepper motor with opto-isolator

Example 10.7.2 : Write an assembly program to rotate a motor 117° in clockwise direction. The motor has a step angle of 1.8°.

Solution :

$$1 \text{ step} = 1.8°$$

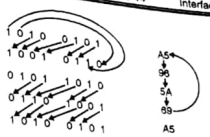
$$x \text{ steps} = 117°$$

$$x = \frac{117}{1.8} = 65$$

$\therefore$ 65 steps are to be given to stepper motor.

The sequence to be given is 0x05, 0x06, 0x04, 0x0A and 0x09. If we load 0xA5 in Accumulator and rotate it left twice, it gives the next code in LSBs. If we repeat it we get the third code and so on.

Hence we will rotate the data left twice, to generate the next step code in the above manner.



Label	Instruction
	org 0000H
	LJMP main
	org 1100H
delay:	
	MOV R4,#250
here :	DJNZ R4,here
	RET
	org 1000H
main :	MOV R0, #65
	MOV A, #0A5H
next :	MOV P0,A
	ACALL delay
	RL A
	RL A
	DJNZ R0, next
	END

Example 10.7.3 : Write an assembly program to rotate the stepper motor clockwise if P1.0 = '1' and anticlockwise if P1.0 = '0'.

Solution :

Label	Instruction
	org 0000H
	LJMP main
	org 1100H
delay:	
	MOV R4,#250

Label	Instruction
here :	DJNZ R4,here
	RET
	org 1000H
main :	MOV A,#0A5H
cw :	JNB P1.0, acw
	MOV P0,A
	ACALL delay
	RL A
	RL A
	SJMP cw
acw :	JB P1.0,cw
	MOV P0,A
	ACALL delay
	RR A
	RR A
	END

Example 10.7.4 : Write an 8051 C program to rotate the stepper motor clockwise if clockwise if P1.0 = '1' and anticlockwise if P1.0 = '0'

Solution :

```
# include < reg 51.h >
sbit sw = P1.0;
unsigned char i, dat = 0xA5;
void main()
{
  sw = 1;
  while (1)
  {
    p0 = dat;
    for (i= 0; i < 200; i++)
    if (sw = = 1)
    dat << 2;
    else
    dat >> 2;
  }
}
```


Example 10.7.5 : Write a program to rotate the stepper motor continuously in clockwise direction using half step and step sequence. Use the sequence values saved in program ROM locations.

Solution : We assume the sequence values are stored in ROM locations starting from 0100 H.

Label	Instruction	Comments
	ORG 0000H	
START :	MOV R0, #08	
	MOV DPTR, #0100H	
L1 :	CLR A	
	MOVC A, @A + DPTR	
	MOV P1, A	
	ACALL DELAY	
	INC DPTR	
	DJNZ R0, L1	
	SJMP START	
	ORG 0100 H	Code for clockwise rotation
	DB 09H,08H,0CH,04H,06H,02H,03H,02H	
	END	

Example 10.7.6 : Write assembly language program to control conveyer belt using stepper motor and 8051 controller. Belt moves continuously at rate of 1 step/sec but stops for 5 sec when external interrupt occurs and then continues to move.

Solution :

Label	Instruction	Comments
	MOV IE, #81 H	Enable external interrupt 0
L2 :	MOV DPTR, #2000 H	Initialize pointer to excitation code table.
	MOV R1, #04 H	Initialize counter to excitation code sequence
L1 :	MOVX A, @DPTR	Get the excitation code
	MOV P1, A	Send excitation code
	MOV A, #14 H	Count = 20 for delay
	LCALL DELAY	Wait for 1 sec
	INC DPTR	
	DJNZ R1, L1	
	SJMP L2	Repeat
	ORG 2000H	
	DB 03H,06H,09H,0CH	Code sequence for clockwise rotation
	END	

Assume external interrupt $\overline{INT0}$ is used.

Instruction	Comments
ORG 0003 H	
MOV A, #64H	Initialize count = 100
ACALL DELAY	Call delay routine
RETI	Return to main program

Delay routine

Label	Instruction	Comments
DELAY :	MOV TMOD, #01H	Timer 0, mode 1 (16 bit mode)
	MOV R0, A	
L4 :	MOV TL0, #B0H	Low byte
	MOV TH0, #3CH	Higher byte
	SETB TR0	Start timer 0
REPE :	JNB TF0, REPE	Check timer 0 flag till it rolls over
	CLR TR0	
	CLR TF0	
	DJNZ R0, L4	
	RET	

The timer 0 gives a delay of 50 ms

∴ For delay of 1 sec = 50 ms × 20

Hence we load 20 ie. 14H as count

In order to get a delay of 5 sec = 50 ms × 100, we load 100 i.e. 64H as count.

10.7.6 Wave Drive 4 Step Sequence

In addition to the 4 and 8 step sequences there is a sequence called as wave drive 4 step sequence as shown Table 10.7.4. The 8 step sequence is combination of wave drive and 4 step sequence.

Table 10.7.4

Step	Winding A_1	Winding A_2	Winding A_3	Winding A_4
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

Counter clockwise ↑ ↓ Clockwise

Example 10.7.7 : Write a program to rotate the stepper motor clockwise using the wave drive 4 step sequence. Use the sequence values saved in program ROM locations.

Solution : We assume that the sequence values are saved in ROM locations starting from 0300 H.

Label	Instruction	Comments
	ORG 0000H	
START :	MOV R0, #04 H	Initialize counter to excitation code sequence
	MOV DPTR, #0300H	
L1 :	CLR A	
	MOVC A, @A+DPTR	
	MOV P1, A	
	ACALL DELAY	
	INC DPTR	
	DJNZ R0, L1	
	SJMP START	
	ORG 0300H	
	DB 8,4,2,1	Code sequence for clockwise rotation
	END	

Example 10.7.8 : Interface stepper motor to 8051. Draw interfacing diagram with driver circuit. Write assembly language program to rotate motor 45° forward and 30° reverse.

Solution : Fig. P. 10.7.8 shows the interfacing diagram.

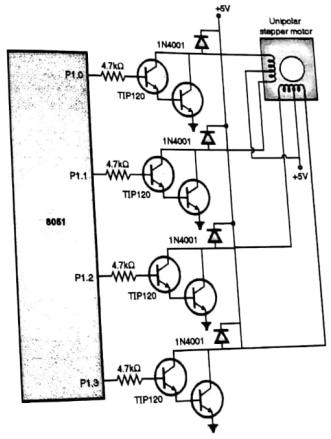


Fig. P. 10.7.8 : 8051 interfacing to stepper motor using transistor drivers

Let us assume 1 step = 2°

$\therefore$ For forward rotation of 45°

$$\text{Number of steps} = \frac{45}{2} \approx 25 \text{ steps}$$

The sequence for forward rotation is 0x05, 0x06, 0x0A, 0x09 assuming clockwise rotation of motor.

For reverse rotation of 30°

$$\text{Number of steps} = \frac{30}{2} = 15 \text{ steps}$$

15 steps are to be given to the motor. The sequence to be given to the motor is 0x09, 0x0A, 0x06, 0x05.

>> Program

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	
	ORG 1100H	
DELAY :	MOV R4, #250	
here :	DJNZ R4, here	
	RET	
	ORG 1000H	
main :	MOV R0, #25	Count for rotating 25 steps in forward direction
	MOV A, #0A5H	
	MOV P0, A	
next :	ACALL DELAY	
	RL A	forward rotation
	RL A	
	DJNZ R0, next	
	MOV R1, #15	Count for rotating 30° reverse i.e. 15 steps in reverse direction
	MOV A, #69H	
back :	MOV P0, A	
	ACALL DELAY	
	RR A	reverse rotation
	RR A	
	DJNZ R1, back	

Example 10.7.9 : Interface stepper motor to 8051. Write an ALP to rotate the stepper motor in clockwise direction for 90°. Use full step mode.

Solution :

Fig. P. 10.7.9 shows the interfacing diagram.

Let us assume that motor has a step angle of 2°.

$$1 \text{ step} = 2^\circ$$

$$x \text{ steps} = 90^\circ$$

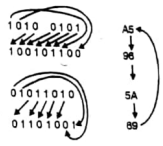
$$\therefore \quad x = \frac{90}{2} = 45$$

$\therefore$ 45 steps are to be given to the motor.

The sequence to be given is 0x05, 0x06, 0x0A, 0x09. If we load 0xA5 in accumulator and rotate it left twice, it gives the next code in LSBs. If we repeat rotation we get the third code and so on.

So, we will rotate the data left twice, to generate the next step code.

Label	Instruction	Comment
	ORG 0000H	
	LJMP main	
	ORG 1100H	
DELAY :	MOV R4, #250	
here :	DJNZ R4, here	
	RET	
	ORG 1000 H	
main :	MOV R0, #45	count for number of steps
	MOV A, #0A5H	
next :	MOV P0,A	
	ACALL DELAY	
	RL A	
	RL A	
	DJNZ R0, next	
	END	



Note : If we want to move motor in reverse direction then we rotate it by two times to right to next code.

Refer Fig. P. 10.7.9.

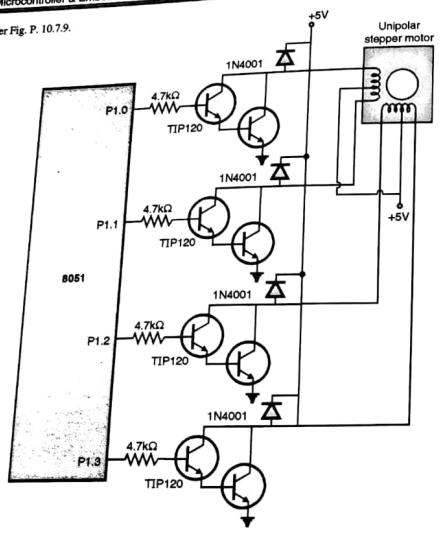


Fig. P. 10.7.9 : Using transistor drivers for stepper motor

Example 10.7.10 : Interface stepper motor to 8051/8951. Draw interfacing diagram showing drive circuit assume winding current of 1.0 Amp for each. Write assembly language program to rotate a motor 50 steps forward and 50 steps backward. Assume wave drive for stepper motor.

Solution :

Fig. P. 10.7.10 shows interfacing diagram.

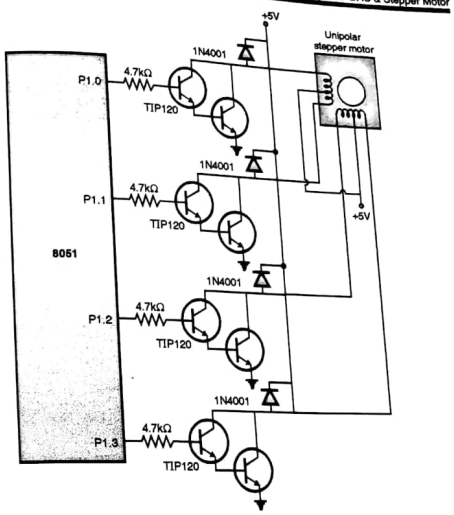


Fig. P. 10.7.10 : Using transistor drivers for stepper motor

The sequence for forward rotation using wave drive is 0x08, 0x04, 0x02, 0x01. We load 0x08 in accumulator and rotate it by one bit to the right we get the next LSB code for forward rotation. The sequence for reverse rotation is 0x01, 0x02, 0x04, 0x08. If we load 0x11 in accumulator and rotate it by one bit to the left we get the next LSB code for reverse rotation.

Label	Instruction	Comment
	ORG 0000H	
	LJMP main	
	ORG 1100H	
DELAY :	MOV R4, #250	
here :	DJNZ R4, here	

Label	Instruction	Comment
	RET	
	ORG 1000H	
main :	MOV R0, #50	Count for 50 steps for forward rotation
	MOV A, #88H	
next :	MOV P0, A	
	ACALL DELAY	
	RR A	
	DJNZ R0, next	
	MOV A, #11H	
	MOV R1, #50	Count for 50 steps for reverse rotation
L1 :	MOV P0, A	
	ACALL DELAY	
	RL A	
	DJNZ R1, L1	
	END	

Example 10.7.11 : Design a 8051 based system to interface stepper motor. Write the corresponding assembly language program to control the stepper motor using keyboard of the PC

- 'c' key to START motor in clockwise direction.
- ' t ' key STOP the motor.
- 'a' key to START motor in anticlockwise direction.
- ' f ' key for increasing the speed (fast).
- 's' key for decreasing the speed (slow).

Solution :

Part (a) : Design

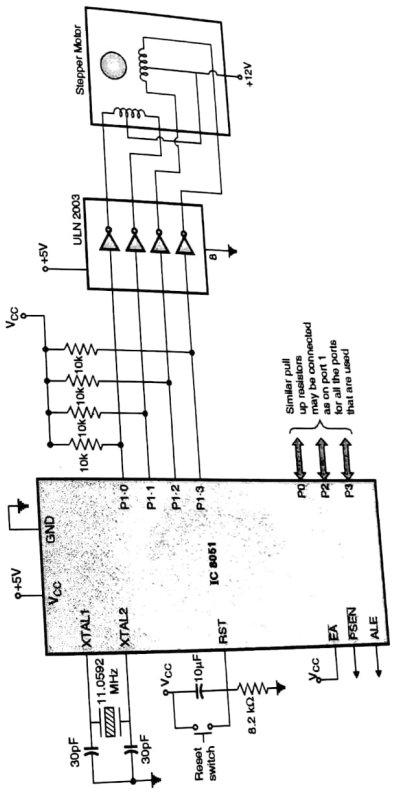


Fig. P. 10.7.11 : Interface diagram

Part (b) Program

We will implement the following state diagram (as shown in Fig. P. 10.7.11) to decide the next state of the system i.e. clockwise rotation, anticlockwise rotation or stop the motor.

>> Algorithm

(A) Main Program

Step I : Initialize serial port in mode 1 and enable reception.
Step II : Initialize TH1 for the required baud rate.
Step III : Enable serial and global interrupts.
Step IV : Initialize timer 1 in mode 2.
Step V : Switch on the timer 1 run bit.
Step VI : Implementation of state diagram, so initial state is 0.
Step VII : If state is 0, check if key is 1 or 2 and change state accordingly to key pressed and transmitted from PC.
Step VIII : If state is 1 and If stop key is pressed change state to 0. Call subroutine for clockwise motion.
Step IX : If state is 2 and If stop key is pressed change state to 0. Call subroutine for anticlockwise motion.
Step X : Goto to step VII

(B) ISR of serial interrupt

Step I : If interrupt is because of TI then clear TI and return
Step II : If interrupt is because of RI then copy the data into a temp register and change the value of a variable (say key) to indicate the next state of the state diagram system. If speed is to be increased or decreased, decrease or increase the delay respectively.
Step III : Clear RI flag

(C) Subroutine for Clockwise rotation

Step I : Give the data in the sequence for forward motion.
Step II : Give a delay after every data.

(D) Subroutine for Anticlockwise rotation

Step I : Give the data in the sequence for reverse motion.
Step II : Give a delay after every data.

>> Registers value

(1) Interrupt Enable (IE) Register

→ To enable global and serial interrupts.

D7	D6	D5	D4	D3	D2	D1	D0
EA	-	-	ES	ET1	EX1	ET0	EX0
1	0	0	1	0	0	0	0

9 0

∴ IE = 0x90

(2) Timer Mode (TMOD) Register

→ To initialize Timer 1 in mode 2 as timer

TIMER 1 TIMER 0

D7	D6	D5	D4	D3	D2	D1	D0
GATE	$\overline{C/T}$	M1	M0	GATE	$\overline{C/T}$	M1	M0
0	0	1	0	0	0	0	0

2 0

∴ TMOD = 0x20

(3) Serial Control (SCON) Register

→ To enable reception and initializing serial communication in mode 1

D7	D6	D5	D4	D3	D2	D1	D0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI
0	1	0	1	0	0	0	0

5 0

∴ SCON = 0x50

(4) Timer 1 registers are to be initialized to 0xFD, so that a baud rate of 9600 can be achieved

∴ TH1 = 0xFd

>> C Program

```
#include<reg51.h>
unsigned char name, key, state;                    //Variables
unsigned int i=0, j=0, delay=500;
unsigned char excit[4]={0x09, 0x0A, 0x06, 0x05};   //Sequence of steps for forward motion
unsigned char excitr[4]={0x05, 0x06, 0x0A, 0x09};  //Sequence of steps for reverse motion
void serialISR(void) interrupt 4                   //Serial port ISR
{
   if (TI) TI=0;                                   //If interrupt is because of transmission return & clear TI
   if(RI)
   {
      name=SBUF;
      RI=0;                                        //Clear RI
```

```
        switch(name)                    //If interrupt is because of reception change the value of key
        {                               // according to the data received
          case 'c': key=1;              //c indicates Clockwise motion, make key=1
          break;
          case 'a': key=2;              //a indicates Anticlockwise motion, make key=2
          break;
          case 't': key=3;              //t indicates stop, make key=3
          break;
          case 's': delay=delay+50;     //s indicates slow, increase delay
          break;
          case 'f': delay=delay-50;     //f indicates fast, decrease delay
          break;
          default: break;
        }
      }
    }
  void clockwise(void)                  //Subroutine for clockwise movement
  {
    for(i=0;i<=3 && (key !=3);i++)      //Give the data in the sequence for forward motion
    {                                   //with the delay according to the speed
    P1=excit[i];
    for(j=0;j<delay;j++);
    }
  }
  void anticlockwise(void)              //Give the data in the sequence for reverse motion
    {                                   //with the delay according to the speed
    for(i=0;i<=3 && (key !=3);i++)
    {
    P1=excitr[i];
    for(j=0;j<delay;j++);
    }
}
void main()
{
  SCON=0x50;                            //Initialize serial port in mode 1 and enable reception
```



```
TH1=0xFD;                   //Initialize TH1 for baud rate of 9600
EA=1;                       //Enable global and serial interrupt
ES=1;
TMOD=0X20;                  //Enable timer 1 in mode 2 for baud rate generation
TR1=1;                      //Switch on timer 1 Run bit
state=0;                    //Implementation of state diagram, so initial state is 0.
while(1)
{
switch(state)
{
  case 0:                   //If state is 0, check if key is 1 or 2 and change state
  if(key==1) state=1;       //accordingly to key pressed and transmitted from PC.
  if(key==2) state=2;
  break;
  case 1:
  if(key==3) state=0;       //If stop key is pressed goto state 0.
  clockwise();              //Call subroutine for clockwise motion.
  break;
  case 2:
  if(key==3) state=0;       //If stop key is pressed goto state 0.
  anticlockwise();          //Call subroutine for anticlockwise motion.
}
}
}
```

>> Assembly Program

Label	Instruction	Comment
	ORG 0000H	
	LJMP START	
	ORG 0023H	Serial port ISR
	JNB RI,NXT	If interrupt is because of reception
	CLR RI	Clear RI
	MOV A,SBUF	change the value of key i.e. R1 according to the data received
	CJNE A,#'c',A1	c indicates Clockwise motion, make key=1
	MOV R1,#01H	
	SJMP NXT	

Label	Instruction	Comment
A1 :	CJNE A,#'a',A2	a indicates Anticlockwise motion, make key=2
	MOV R1,#02H	
	SJMP NXT	
A2 :	CJNE A,#'t',A3	t indicates stop, make key=3
	MOV R1,#03H	
	SJMP NXT	
A3 :	CJNE A,#'s',A4	s indicates slow, increase delay in R3 and R4 together
	MOV R2,A	
	MOV A,R3	
	ADD A,#50	
	MOV R3,A	
	MOV A,R2	
	JNC A5	
	INC R4	
A5 :	SJMP NXT	
A4 :	CJNE A,#'f',NXT	f indicates fast, decrease delay in R3 and R4 together
	MOV R2,A	
	MOV A,R3	
	CLR C	
	SUBB A,#50	
	MOV R3,A	
	MOV A,R2	
	JNC NXT	
	DEC R4	
NXT :	JNB TI,NXT1	If interrupt is because of transmission return & clear TI
	CLR TI	
NXT1 :	RETI	
	ORG 0100H	Subroutine for clockwise movement
CLOCKWISE :	MOV R0,#30H	Give the data in the sequence for forward motion with the delay according to the speed
STEP :	MOV P1,@R0	
	MOV 06,R3	
	MOV 07,R4	

Label	Instruction	Comment
AGAIN1 :	MOV 04,R7	
AGAIN :	DJNZ R4,AGAIN	
	DJNZ R3,AGAIN1	
	MOV 03,R6	
	MOV 04,R7	
	INC R0	
	CJNE R0,#34H,STEP	
	RET	
	ORG 0200H	
ANTICLOCKWISE:	MOV R0,#40H	Give the data in the sequence for reverse motion with the delay according to the speed
STEP1 :	MOV P1,@R0	
	MOV 06,R3	
	MOV 07,R4	
AGAIN3 :	MOV 04,R7	
AGAIN2 :	DJNZ R4,AGAIN2	
	DJNZ R3,AGAIN3	
	MOV 03,R6	
	MOV 04,R7	
	INC R0	
	CJNE R0,#44H,STEP1	
	RET	
	ORG 1000H	
START :	MOV R3,#100	Initialize counter for delay
	MOV R4,#50	
	MOV R0,#30H	Store the sequence in RAM for clockwise movement
	MOV @R0,#09H	
	INC R0	
	MOV @R0,#0AH	
	INC R0	
	MOV @R0,#06H	
	INC R0	
	MOV @R0,#05H	
	MOV R0,#40H	Store the sequence in RAM for anticlockwise movement
	MOV @R0,#05H	

Label	Instruction	Comment
	INC R0	
	MOV @R0,#06H	
	INC R0	
	MOV @R0,#0AH	
	INC R0	
	MOV @R0,#09H	
	MOV SCON,#50H	Initialize serial port in mode 1 and enable reception
	MOV TH1,#0FDH	Initialize TH1 for baud rate of 9600
	MOV IE,#90H	Enable global and serial interrupt
	MOV TMOD,#20H	Enable timer 1 in mode 2 for baud rate generation
	SETB TR1	Switch on timer 1 Run bit
	MOV R5,#00H	Implementation of state diagram, so initial state (Register R5) is 0.
HERE :	CJNE R5,#00H,B1	If state is 0, check if key is 1 or 2 and change state accordingly to key pressed and transmitted from PC.
	CJNE R1,#01H,B3	
	MOV R5,#01	
B3 :	CJNE R1,#02H,B1	
	MOV R5,#02	
	SJMP HERE	
B1 :	CJNE R5,#01H,B2	
	CJNE R1,#03H,B4	If stop key is pressed goto state 0.
	MOV R5,#00	
	SJMP HERE	
B4 :	LCALL CLOCKWISE	Call subroutine for clockwise motion.
	SJMP HERE	
B2 :	CJNE R5,#02H,HERE	
	CJNE R1,#03H,B5	If stop key is pressed goto state 0.
	MOV R5,#00	
	SJMP HERE	
B5 :	LCALL ANTICLOCKWISE	Call subroutine for anticlockwise motion.
	SJMP HERE	
	END	

10.8 Design Examples

Example 10.8.1 : Design a 8051 based system to interface DAC. Write the corresponding Assembly program to generate triangular wave. **(14 Marks)**

Solution :

Part (a) : Design

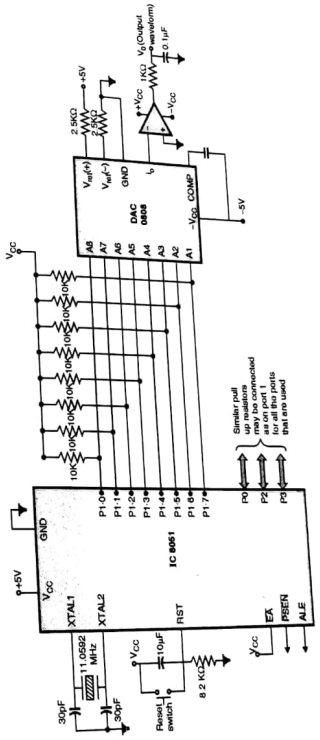


Fig. P. 10.8.1 : Interface diagram

Part (b) : Program

>> Algorithm

Main Program

Step I : Initialize port 1 to all 0s

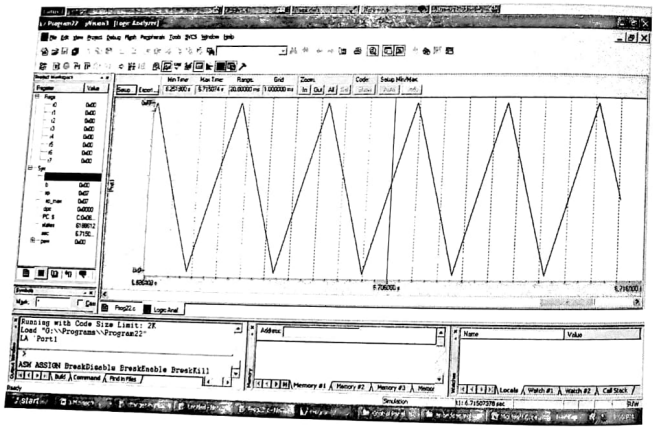
Step II : Increment the data in P1 till it reaches the maximum i.e. FFH

Step III : Once it reaches the maximum, decrement the data in P1 till it reaches minimum i.e. 00H

>> Assembly Program

```
ORG 0000H
LJMP START
ORG 0100H
START:
MOV A,#00H          //Initialise Port 1 to all 0's
HERE:
MOV P1,A
INC A
CJNE A,#0FFH,HERE   //Increment the data in port 1 till it reaches the maximum i.e. FFH
NXT:
MOV P1,A
DEC A
CJNE A,#00H,NXT     //Once it reaches maximum, decrement the data in P1 till it reaches
SJMP HERE           // minimum i.e. 00H
END
```

>> Output



Example 10.8.2 : Design a 8051 based system to interface DAC. Write the corresponding Assembly program to generate Sinusoidal wave. **(14 Marks)**

Solution :

Part (a) : Design

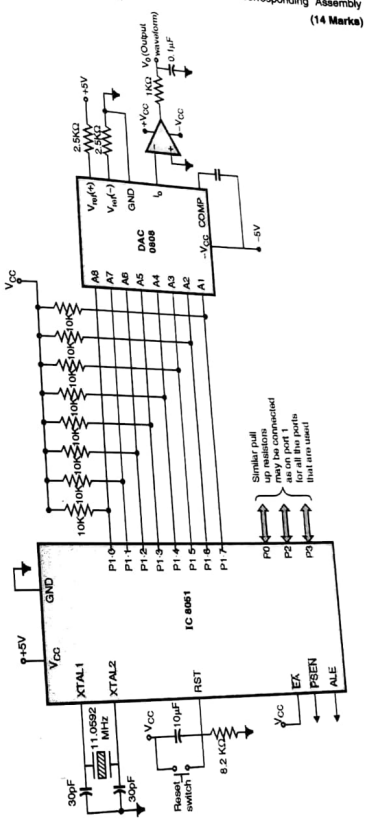


Fig. P. 10.8.2 : Interface diagram

Part (b) : Program

We need to calculate the values for sinusoidal waveform between 00H and FFH. This is done by the following table. Here we have divided the entire cycle of 360° into 48 parts each of incremental 7.5°. Hence the angles are 0°, 7.5 °, 15°, 22.5°....

We cannot have negative voltage in the output of our microcontroller. Hence for Sin 0, we need the output to be in centre i.e. 2.5V, treating it as x-axis. Hence we add 2.5V to every calculation as seen in the fourth row of the table below. Then we find the corresponding value for each voltage by multiplying it with the maximum count and dividing it by maximum voltage as given in the fifth column of the table.

Note: If you still reduce the step size from 7.5°, you may get a better waveform. For the program in assembly, we have taken the angles at an interval of 30°. This gives less accurate output.

Calculation of array elements

Sr. No.	Angle in degrees (θ)	sin (θ)	Count = 2.5+(2.5 * sin θ)	count*256/5
0	0	0	2.5	128
1	7.5	0.130578428	2.826446071	145
2	15	0.258920827	3.147302068	161
3	22.5	0.382829457	3.457073643	177
4	30	0.500182502	3.750456255	192
5	37.5	0.608970405	4.022426013	206
6	45	0.707330278	4.268325695	219
7	52.5	0.793577803	4.483944508	230
8	60	0.866236075	4.665590188	239
9	67.5	0.924060891	4.810152227	246
10	75	0.966062056	4.915155141	252
11	82.5	0.991520342	4.978800856	255
12	90	0.9999998	4.9999995	256
13	97.5	0.991355227	4.978388068	255
14	105	0.965734654	4.914336634	252
15	112.5	0.923576807	4.808942018	246
16	120	0.8656036	4.664008999	239
17	127.5	0.792807767	4.482019417	229
18	135	0.706435867	4.266089666	218
19	142.5	0.607966935	4.019917336	206
20	150	0.499087156	3.74771789	192
21	157.5	0.381660992	3.45415248	177
22	165	0.257699252	3.144248131	161

Sr. No.	Angle in degrees (θ)	sin (θ)	Count = 2.5+(2.5 * sin θ)	count*256/5
23	172.5	0.129324662	2.823311654	145
24	180	0.001264489	2.496838778	128
25	187.5	0.131831986	2.170420034	111
26	195	0.260141988	1.849645029	95
27	202.5	-0.38399731	1.540006725	79
28	210	0.501277049	1.246807379	64
29	217.5	0.609972902	0.975067745	50
30	225	0.708223559	0.729441104	37
31	232.5	0.794346571	0.514133573	26
32	240	0.866867165	0.332832087	17
33	247.5	0.924543497	0.188641258	10
34	255	0.966387914	0.084030214	4
35	262.5	0.991683872	0.02079032	1
36	270	0.999998201	4.9999	256
37	277.5	0.991188527	0.022028682	1
38	285	0.965405707	0.086485732	4
39	292.5	0.923091247	0.192271883	10
40	300	-0.86496974	0.337575649	17
41	307.5	0.792036463	0.519908843	27
42	315	0.705540326	0.736149186	38
43	322.5	0.606962492	0.98259377	50
44	330	0.497991012	1.25502247	64
45	337.5	0.380491917	1.548770208	79
46	345	0.256477265	1.858806837	95
47	352.5	0.128070688	2.17982328	112
48	360	0.002528976	2.50632244	128

Note : You need not take so many samples, instead you can take angle at an interval of 30° and have 12 values (For Exam)

➢➢ Algorithm

(A) Main Program

Step I : Initialize data according to the above table

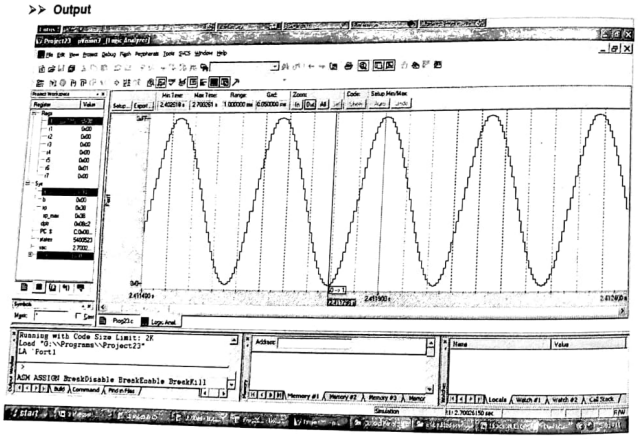
Step II : Issue the data one by one from the array.

Step III : Repeat the above step continuously to get a continuous waveform.

>> **Assembly Program**

Label	Instruction	Comments
	org 0000H	
	org 0050H	
	array dB 128, 145, 161, 177, 192, 206, 219, 230, 239, 246, 252, 255, 256, 255, 252, 246, 239, 229, 218, 206, 192, 177, 161, 145, 128, 111, 95, 79, 64, 50, 37, 26, 17, 10, 4, 1, 0, 1, 4, 10, 17, 27, 38, 50, 64, 79, 95, 112, 128	
	org 0100H	
START:	MOV DPTR, #0050H	Initialize pointer
	MOV R0, #31H	Initialize counter
back:	MOV A, #00H	
	MOVC A, @A+DPTR	get a data
	MOV P1, A	place the data on Port 1
	INC DPTR	Increment pointer
	DJNZ R0, back	Decrement counter and return if not zero
here:	SJMP here	
	END	

>> **Output**



Example 10.8.3 : Write an assembly language program for square wave generation. **(14 Marks)**

Solution :

A square wave has only two amplitudes, a minimum say 0 V (00H) and a maximum of 10 V (FFH). To generate square wave we have to output 00H and then FFH to Port 1 of 8051. The Port 1 is connected as input to DAC 0808. According to the frequency requirement delay is provided between the outputs. Fig. P. 10.8.3 shows the interfacing diagram for a 1 KHz square assuming 50% duty cycle, the time period.

$$T = \frac{1}{f} = 1 \text{ ms and } T_{ON} = T_{OFF} = 0.5 \text{ ms}$$

>> **Program**

```
# include <reg51.h>
main ()
{
   unsigned int i;
   while (1)
   {
      P1 = 00;                    // output low
      for (i = 0; i<1275; i++ )   // 1 ms delay
      P1 = 0×FF;                  // output high
      for (i = 0; i<1275; i++)    // 1 ms delay
   }                              // end of while
}
```

>> **Assembly Program**

Label	Instruction	Comments
AGAIN :	MOV A, #00H	
	MOV P1, A	Output low
	ACALL DELAY	0.5 ms delay
	MOV A, #0FFH	
	MOV P1, A	Output high
	ACALL DELAY	
	SJMP AGAIN	
DELAY :	MOV R0, #0FFH	
HERE :	DJNZ R0, HERE	
	RET	
	END	

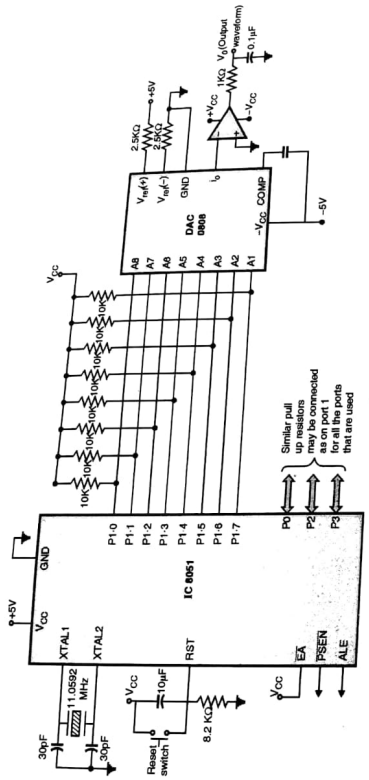


Fig. P. 10.8.3

Example 10.8.4 : Design 8751 (EPROM version of 8051) system to send a value of an analog signal to the desktop computer through serial port. The analog signal is in the range of 0V to 5.12V. se ADC0809 (8 bit, successive approximation type of ADC). Explain the design with the help of flowchart.

(10 Marks)

Solution :

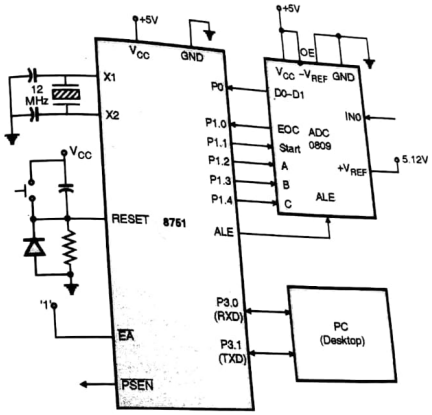


Fig. P. 10.8.4

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	
	ORG 0100H	
main:	SETB P1.0	
	MOV TMOD, #20H	
	MOV TH1, #0FDH	
	SETB TR1	
	MOV SCON, #20H	
	CLR P1.2	

Label	Instruction	Comments
	CLR P1.3	
	CLR P1.4	
here:	SETB P1.1	
wait1:	JB P1.0, wait1	
wait2:	JNB P1.0, wait2	
	MOV SBUF, P0	
	MOV R7, #100	
wait3:	DJNZ R7, wait3	
	SJMP here	
	END	

10.9 Exam Pack (Review Questions)

Q. 1 List the features of ADC 0808. **(Refer section 10.3)** **(3 Marks)**

☞ **Syllabus Topic : Interfacing ADC, DAC**

Q. 2. Draw and explain the typical interfacing circuit for DAC 0808. **(Refer section 10.4)** **(6 Marks)**

Q. 3. Give a complete scheme to interface an 8-bit ADC to microcontroller 8051. **(Refer section 10.4)** **(6 Marks)**

Q. 4 Draw and explain the interfacing of ADC 0804 to 8051 **(Refer section 10.5)** **(8 Marks)**

Q. 5 Interface with 8051 microcontroller Analog to Digital converter ADC 0804. Draw the interface diagram and Read, Write pulses input waveform. **(Refer section 10.5.3)** **(10 Marks)**

☞ **Syllabus Topic : Interfacing Stepper Motor**

Q. 6 Write an assembly language program for square wave generation. **(Refer Example 10.8.3)** **(14 Marks)**

Q. 7 Suggest the scheme for generating square wave and triangular wave of frequency 1KHz using 8051 microcontroller. Assume microcontroller is operating on 6MHz frequency, write a program for each of them. **(Refer Examples. 10.8.1 and 10.8.3)** **(14 Marks)**

Q. 8 Design 8751 (EPROM version of 8051) system to send a value of an analog signal to the desktop computer through serial port. The analog signal is in the range of 0V to 5.12V. Use ADC0809 (8 bit, successive approximation type of ADC). Explain the design with the help of flowchart. **(Refer Examples. 10.8.4)** **(10 Marks)**

Q. 9 Write short note on : Stepper motor interfacing with 8051. **(Refer section 10.7)** **(7 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 11

Introduction to ARM Processor

11.1 The Acorn RISC Machine

- The first ARM processor was developed at Acorn Computers Limited, England, between 1983 and 1985.
- At that time ARM stood for Acorn RISC Machine, until the formation of Advanced RISC Machines Limited when it was renamed Advanced RISC Machine in 1990.
- The most 32-bit microcontrollers in the embedded industry are ARM microcontrollers, and it has a stake of 75% in this industry.
- With a common architecture, ARM has a very wide range of processor cores and delivers high performance together with low power consumption as well as system cost.
- The ARM processor are of a wide range with solutions for :
  - (1) Open platforms for wireless, consumer and imaging applications, that use complicate operating system.
  - (2) Mass storage, automotive, industrial and networking with real time embedded applications.
  - (3) Smart cards and SIMs requiring high security.

11.1.1 RISC Properties

A RISC system must satisfy the following properties :

1. Single-cycle execution of all (or at least 80 percent) instructions.
2. Single-word standard length of all instructions.
3. Small number of instructions (<= 128).
4. Small number of instruction formats (<= 4).
5. Small number of addressing modes (<= 4).
6. Memory access possible by load and store instructions only.
7. All operations, except load and store, are register to register i.e. within the CPU.
8. It must have a hardwired control unit.
9. It must also have a relatively large (at least 32) general-purpose CPU register file.

11.1.2 Register Window

1. Since there are a huge number of registers in a RISC processor, it can be useful in saving the latency period during procedure call. Parameter passing in is possible by the **register window**. This policy also allows reasonable HLL support in RISC designs.
2. The register file is subdivided into groups of registers, called register **windows**.
3. A certain group of 'i' registers, suppose R_0 to $R_{(i-1)}$, are designated as **global registers**. The global registers are accessible to all procedures running on the system at all the times.
4. On the other hand, each procedure is assigned a separate window within the register file, which is not accessible to other procedures.
5. The window base (first register within the window) is pointed to by a field called **current window pointer (CWP)** located in the CPU's **status register (SR).**
6. If the currently running procedure is assigned the register window J, hence taking up registers K, K+1, . . . , K+W-1 (where W is the number of registers per window), the CWP contains the value J, hence pointing to the base of window J. If the next procedure to execute takes up window J+1, the value in the CWP field will be incremented accordingly to point to J+1.

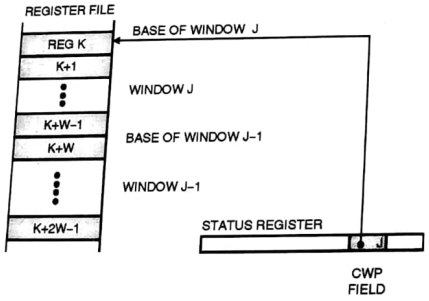


Fig. 11.1.1 : Simple non-overlapping register window

7. Register windowing can work more efficiently for parameter passing between calling and called procedures by partial overlapping of the windows. The last N registers of window J will be the first N registers of window J+1.
8. If the procedure that is using window 'J', calls another procedure, then the new procedure will be assigned the next window i.e. 'J+1', and the former procedure can pass N parameters to the called procedure by placing their values into registers (K+W-N) to (K+W-1). Since the two windows are overlapping, the same registers will be available to the called procedure without actually moving the data.

11.1.3 Miscellaneous Features of RISC Systems

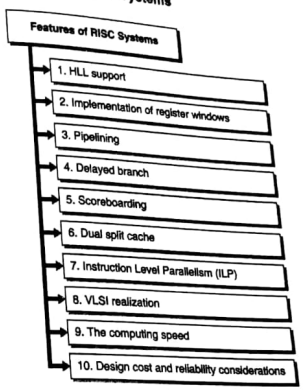


Fig. 11.1.2 : Features of RISC Systems

➔ **1. HLL support**

(a) The support for High Level Language (HLL) features is mandatory in the design of any computing system.

(b) The procedure call-return and parameters passing is the most time-consuming operation in typical HLL programs.

(c) HLL support is provided in RISC machines by supporting efficiently the handling of local variables, constants, and procedure calls, while leaving less frequent HLL operations to instructions sequences and subroutines.

(d) One of the mechanisms supporting the handling of procedures, and their parameter passing in particular, is the feature of the **register window**.

➔ **2. Implementation of register windows**

(a) A CISC processor's control unit takes up a large percentage of the chip area, leaving very little space for other subsystems and basically not permitting a large register file, needed for an efficient implementation of windowing.

(b) A RISC processor's control unit makes up a must smaller percentage of the chip area, yielding the necessary space for a large register file.

(c) And hence implementation of register windowing (that requires huge number of registers) is possible in RISC processors

3. Pipelining

(a) Pipelining was used on various CISC systems even before the RISC approach became popular.

(b) But, a streamlined RISC can handle pipelines more efficiently.

4. Delayed branch

(a) The problem occurs in a system where instructions are prefetched, right after a branch.

(b) If the branch is conditional, and the condition is not satisfied, then the next instruction, which was prefetched, is executed, and since no branch is to be performed, no time is lost.

(c) But, if the branch condition is satisfied, or the branch is unconditional, the next prefetched instruction is to be flushed and other instruction pointed to by the branch address is to be fetched in its place. The time required to prefetch the flushed instruction is wasted.

(d) Such waste of time is solved by using the **delayed branch approach**.

(e) In this approach, the instructions are reshuffled such that the operation does not change the result.

(f) A successful branch is assumed and the execution of the branch is delayed until the already prefetched instructions are executed. Hence, no time is lost and there is no change in the intended program operation.

(g) But the compiler has to take care that the instructions followed by the branch instructions are to be executed irrespective to the branch to be taken or not.

5. Scoreboarding

(a) Another problem in instruction pipelines is that of data dependency.

(b) The data in some register put by instruction 1 may be required by instruction 2; and before the value in the register is available, the instruction 2 may be ready for execution yielding a possibly incorrect result.

(c) A method used to solve this problem is called as **scoreboarding**.

(d) A special CPU control register i.e. the **scoreboard register,** is required for this purpose.

(e) If there are 32 registers, a scoreboard register 32-bit long will be required; each of its bit represents one of the 32 CPU registers.

(f) If register 'i' is involved as a destination in the execution of instruction 1, bit 'i' in the scoreboard register is set; and as long as bit 'i' is set, any subsequent instruction in the pipeline will be prevented from using Ri in any way until bit 'i' is cleared.

(g) This instruction will now be executed as soon as the execution of the instruction, which caused bit 'i' to be set, is completed.

6. Dual split cache

Another feature, implemented in not only a CISC but also RISC systems, is separated data and code caches, or split cache.

7. Instruction Level Parallelism (ILP)

Superscalar and superpipelined designs are also mostly implemented in a RISC design.

8. VLSI realization

(a) The chip area, dedicated to the realization of the control unit, is considerably less. Therefore, on a RISC VLSI chip, there is more area available for other features (cache, FPU, part of the main memory, memory management unit, I/O ports, etc).

(b) As a result of the considerable reduction of the control area, a large number of CPU registers can fit on-chip.

(c) By reducing the control area on the VLSI chip and filling the area by numerous identical registers, the **regularization factor** (which is defined as, ratio of chip area utilized by other features to tha chip space required by control unit) of the chip increases. The higher the regularization factor, the lower is the VLSI design cost.

9. The computing speed

(a) A simpler and smaller control unit in RISC requires fewer gates. This results in shorter propagation paths for control unit signals, decreasing the delay time for control signals and hence yielding a faster operation.

(b) A significantly reduced number of instructions, formats, and addressing modes results in a simpler and smaller decoding system, resulting in faster decoding operation.

(c) A hardwire-controlled system can hence be implemented, with a reduced control unit that will in general be faster than a microprogramed control unit.

(d) A relatively large CPU register file also reduces CPU-memory traffic to fetch and store data operands.

(e) A large register set can also be used to store parameters to be passed from a calling to a called procedure, to store the information of a process that was interrupted by another.

10. Design cost and reliability considerations

(a) It takes a shorter time to complete the design of a RISC control unit, because of the smaller instruction set, fixed instruction length and less instruction formats. Thus contributing to the reduction in the overall design cost.

(b) A simpler and smaller control unit will obviously have a reduced number of design errors and, therefore, a higher reliability.

Syllabus Topic : Architecture Inheritance

11.1.4 Architecture Inheritance

(MU - Dec. 14, May 15, Dec. 15, May 16, Dec. 17)

Q.11.1.1 Describe the principal features of the ARM architecture. (Ref. Sec. 11.1.4) **Dec. 14, Dec.15, May 16, 6 Marks**

Q.11.1.2 Describe the features of ARM that makes it suitable for embedded system. (Ref. Sec. 11.1.4) **May 15, May 16, Dec. 17, 6 Marks**

The ARM architecture incorporated a number of **features** inherited from RISC design like :

(1) A load-store architecture

- A load-store architecture means that ALU instructions cannot access memory, they can operate only on registers.

- Data transfer between memory and processor is possible only by Load and Store instructions.
  - LOAD instruction is used to load the data from memory into the processor's register.
  - STORE instruction is used to store data in memory from the processor's register.
  - The data in the registers can be processed by the ALU.

☞ **(2) Fixed-length 32-bit instructions**

- All the instructions are 32-bit in length.
- This fixed length instructions make the design of the control unit simpler.
- Fixed length instructions also make it possible to use Hardwired control unit, which makes the operation of control unit faster.

☞ **(3) 3-address instruction formats**

- This is also referred to as 3-opeand instructions
- The two source operands are different then the destination operand for e.g. in the add instruction given below

ADD d, s_1, s_2 ;

Operation of above instruction: $d := s1 + s2$

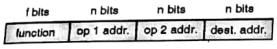


Fig. 11.1.2 : A 3-address instruction format.

Another feature of ARM is

☞ **(4) Simplicity**

- Since the ARM has simple hardware and a strong instruction, it results in a better code density compared to general RISC.
- Although ARM is said to be a RISC processor, it has not implemented certain standard features of RISC processor. Here are some of them:
  (1) Register Windowing
  (2) Delayed Branching
  (3) Single cycle execution of all instructions.

11.2 ARM Family Core Architecture

- The ARM core is an engine within the system. It is responsible for fetching the instructions from the memory and executing them.
- The size of ARM core is very small; hence many additional components are added to the same chip.
- We will study the basic structure of the ARM core, as selection of an ARM core is the most critical decision while designing a system.

- Fig. 11.2.1 shows the ARM core data flow model. It supports Von-Neumann architecture.

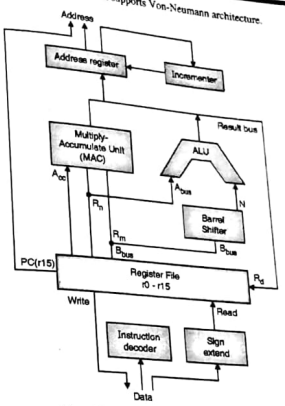


Fig. 11.2.1 : ARM core data flow model

Now, let us see the different functional units :

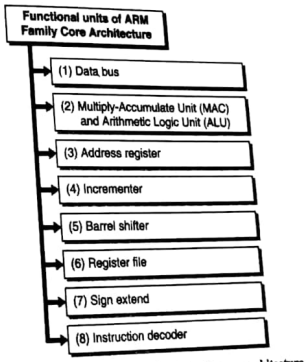


Fig. 11.2.2 : Functional units of ARM family core architecture

➔ **(1) Data bus**

The data enters the ARM core through the data bus . The same data bus is used by all the instructions as the ARM core uses Von-Neumann architecture. There are two buses A_{bus} and B_{bus}. The data can be an instruction opcode.

➔ **(2) Multiply-Accumulate Unit (MAC) and Arithmetic Logic Unit (ALU)**

- Generally the ARM instructions are of three operands. In order to store the two input operands registers R_m and R_n are used.
- The ALU or MAC reads the two operand values from R_m and R_n registers using the A_{bus} and B_{bus}.
- Then the desired operation is done and the result is stored using the result bus or C_{bus} in the destination register R_d.

➔ **(3) Address register**

It is responsible for storing the address generated by the load and store instructions and placing it on the address bus.

➔ **(4) Incrementer**

- The incrementer is responsible for upgrading the address register contents before the core reads the next register value from memory location or writes the next register value to memory location.
- In case of exception / interrupt the execution of instructions is changed.

➔ **(5) Barrel shifter**

- The R_m register contents can be preprocessed in the barrel shifter before applying them to the ALU.
- A wide range of shift operations are possible using the barrel shifter and the ALU.

➔ **(6) Register file**

The register file is a bank of sixteen 32-bit registers. The registers are used for storing the data.

➔ **(7) Sign extend**

Some of the instructions require signed values. Whenever the processor reads an 8 or 16 bit signed number from the memory, the sign extend hardware converts the numbers to 32 bit values and places the number in the register file.

➔ **(8) Instruction decoder**

- The instruction decoder decodes the instruction opcode read from the memory and then the instruction is executed.
- The ARM cores are of three types :

  (1) Application Core (2) Embedded Core (3) Secure Core
- Since the architecture of ARM based processor takes very less space and performs a lot of operations it is fit to be used in embedded systems

11.3 Versions and Variants

- In the 1980s Apple Computer started working with a company called as Acorn. This work was such crucial that Acorn spun off the design team in 1990 into a new company called Advanced RISC Machines (ARM)
- Later, it became ARM Limited and it was floated on the London Stock Exchange and NASDAQ in 1998.
- The ARM was developed for the first time with its samples in 1985 and was called as **ARM1**, and the first "real" production systems was called as **ARM2** developed in 1986.
- This ARM2 had a 32-bit data bus and 26 bit address bus with 32 bit registers.
- There were 16 such 32-bit registers with one of them serving as a program counter. Some of the bits of program counter were used as control and status flags.
- This ARM2, that used 32000 transistors, was almost the most simple and very useful 32-bit microprocessor in the world.
- The ARM2 was succeeded by **ARM3**, that had a cache memory of 4KB and hence improving the performance.
- Later ARM6 was developed and then the most successful implementation of ARM i.e. the ARM7TDMI which is present in hundreds of millions of cellular phones. Freescale (spun off of Motorola in 2004), IBM, Texas Instruments, Nintendo, Philips, VLSI, Atmel, Sharp, Samsung and STMicroelectronics have also licensed the basic ARM design for various uses.
- The ARM chip has become one of the most used CPU designs in the world, found in everything from hard drives, to mobile phones, to routers, to calculators, to children's toys. Today it accounts for over 75% of all 32-bit embedded CPU's.
- The ARM implementation has five versions of the instructions set that are defined till date. The versions are represented by version numbers 1 to 5.
- In order to specify collections of additional instructions included the instruction set, some of the versions are qualified with variant letters.
- M variant denotes addition of four instructions and T variant denotes additions of entire thumb instruction set.

The five versions are as follows :

☞ **(1) Version 1**

- This version was implemented on ARM 1 processor. It has 26 bit address space.
- It supports branch instructions including a branch and link instruction designed for subroutine calls.
- It supports a software interrupts instruction, for use in making operating system calls.
- It supports byte, word and multi-word load-store instructions.
- It supports the data processing instructions also.
- Version 1 nowadays has become obsolete.

☞ **(2) Version 2**

- This version is a updated version of version 1. It also has 26 bit address space.
- It supports coprocessor supported instructions.

- It supports multiply and multiply-accumulate instructions.
- It includes load and store instructions called SWP and SWPB in a variant called version 2.
- It supports two more banked registers in fast interrupt mode.
- Version 2 is also obsolete.

☞ **(3) Version 3**

- This version has 32 bit address space.
- In this version with a view to retain the contents of CPSR register in the case of an exception two new registers were added. The program status information which was stored in register R15 is now stored in the CPSR (Current Program Status Register) and the SPSR (Saved Program Status Register).
- In version 3 two new instructions (MRS and MSR) were added to access the CPSR and SPSR registers.
- It supports two new processor modes. The new processor modes help to use Data Abort, undefined instruction exceptions in the code.

☞ **(4) Version 4**

- The version 4 is an extension of version 3.
- It includes additional features like :
  - (i) load / store halfword instructions.
  - (ii) instruction transfer to T state in T variants.
  - (iii) new privileged processor mode using the user mode register.
- Version 4 made a clear distinction of the instructions that should cause the exception to be taken.
- Backwards-compatibility support for 26 bit architecture ceased to be obligatory in version 4.

☞ **(5) Version 5**

- The version 5 is an upward extension of version 4.
- It includes additional instructions that improve the efficiency of ARM interworking in T variants.
- Version 5 allows same code generation techniques to be used for T and non T variants.
- It supports an additional software breakpoint instruction.
- Version 5 also supports a count leading zeros (CLZ) instruction that allows efficient integer divide and interrupt prioritization.
- Version 5 tightens the definition of how flags are set by multiply instructions.
- It supports co-processor instructions.

The variants are as follows :

☞ **(1) The Thumb Instruction Set (T variants)**

- It is a subset of the ARM instruction set. The thumb instruction set has 16 bit instructions. i.e. the Thumb instructions are half the size of ARM instructions.
- The Thumb instruction set has greater code density than the ARM instruction set.

- The drawbacks of the Thumb instruction set are that it requires more instructions to complete the same task. Hence the ARM instruction set is best for maximizing the performance.
- The Thumb instruction set does not include instructions required for exception handling.
- Due to these drawbacks the Thumb instruction set is used in conjunction with a suitable version of the ARM instruction set.
- The Thumb instruction set is not valid prior to version 4 of the ARM architecture. It is signified by letter **T**.

☞ **(2) Thumb instruction set versions**

- There are two versions of the Thumb instruction set i.e. version 1 and version 2.
- The version 1 of the Thumb instruction set is used in T variants of the ARM architecture version 4 and version 2 is used in T variants of the ARM architecture version 5.
- The Thumb instruction set version 2 modifies the instructions to improve the efficiency of ARM interworking.
- The Thumb instruction set tightens the definition of how flags are set by multiply instructions.
- The Thumb instruction set includes the breakpoint instruction.
- These modifications are related to the changes between ARM architecture versions 4 and 5.

☞ **(3) M variants (Long Multiply Instructions)**

- The ARM instruction set includes four additional multiply instructions for producing 64 bit multiply and multiply and accumulate results.
- These instructions imply the existence of a multiplier that is larger than the minimum. Its presence is denoted by variant letter **M**.
- The long multiply instructions were first included in the variant of version 3 and exists in all after version 3.

☞ **(4) E variants (Enhanced DSP Instructions)**

- The E variants of the ARM instructions set consist of a number of ARM instructions that enhance the performance of an ARM processor for digital signal processing applications.
- The **E** variants support load, store, co-processor register transfer instructions that act on double word data.
- They include a cache preload instruction (PLD).
- They include several new multiply and multiply-accumulate instructions acting on 16 bit data.
- They were first included in the ARM architecture version 5T. They are not valid in non-T or non-M variants of the ARM architecture.

11.3.1 Terms Related to ARM Instruction Set

- Given below are some terms used in the ARM instruction set :
- {cond}: Condition field :
- We know that an ARM instruction is 32 bit long. The 4 MSBs of the instructions are called as the **condition field**. The condition field is of 4 bits, hence there are 16 conditions as listed in Table 11.3.1.

Fig. 11.3.1 shows the condition field.

Fig. 11.3.1 : ARM condition field

Table 11.3.1

Condition field	{cond}	
Suffix	**Description**	**Condition flag state**
EQ	Equal	Z = 1
NE	Not Equal	Z = 0
CS	Unsigned higher or same	C = 1
CC	Unsigned lower	C = 0
MI	Negative	N = 1
PL	Positive 0 or zero	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned higher	C = 0 and Z = 0
LS	Unsigned lower or same	C =1 and Z = 1
GE	Signed greater than or equal	N = 1 and V = 1 or N = 0 and V = 0.
LT	Signed less than	N = 1 and V = 0 or N = 0 and V = 1.
GT	Signed greater than	Z = 0 and either N = 1 and V = 1 or N = 0 and V= 1.
LE	Signed less than or equal	Z = 1 or N = 1 and V = 0 or N = 0 and V = 1.
AL	Always (unconditional)	
NV	Reserved	

Syllabus Topic : Detected Study of Programmer's Model

11.4 The ARM Programmers Model

➔ **(MU - May 16)**

Q. 11.4.1 Explain detailed programmer's model of ARM 7. (Ref. Sec. 11.4) **MU - May 16, 10 Marks**

- The ARM processor can be operated in seven different modes. The current mode of the processor decides the availability of the registers to the programmer.

- Fig. 11.4.1 shows the programming model of the ARM processor. The ARM processor has 37 thirty two bit registers.
- The registers include of
  - (i) a dedicated program counter.
  - (ii) a dedicated current program status register.
  - (iii) a dedicated saved program status registers.
  - (iv) 30 general purpose registers.
- However these registers are arranged into several banks, with the accessible bank being governed by the processor mode.
- Each processor mode can access a particular set of R0-R12 registers, R13 register (stack pointer) and link register (R14), register R15 (program counter) and CPSR (current program status register). Privileged modes can access SPSR (saved program status register).

Fig. 11.4.1 shows the programming model of the ARM processor.

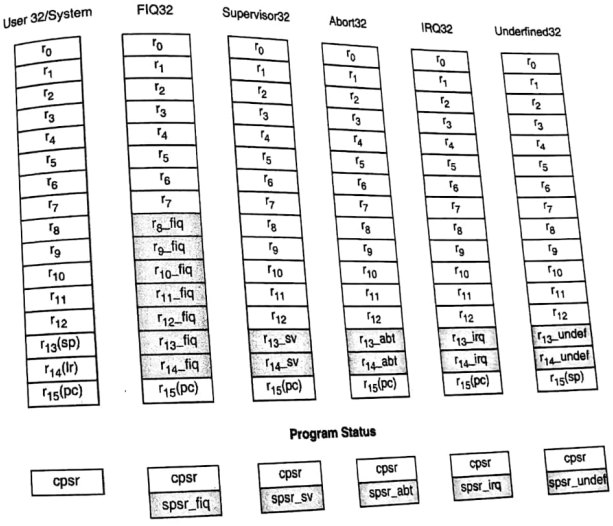


Fig. 11.4.1 : ARM programming model

(1) General purpose registers

- The registers R_0 to R_{12} are used as general purpose registers.
- They can be used for holding the data or addresses.
- Depending on the purpose, the registers R_{13} to R_{15} can also be used as general purpose registers.

(2) Stack Pointer (SP-R13)

- Register R13 is a stack pointer as shown in Fig. 11.4.1.
- It stores the top of the stack in the current processor mode.

(3) Link Register (LR-R14)

- Register R14 is a link register. It is used for storing the return address in case of subroutine calls.

(4) Program Counter (PC-R15)

- Register R15 is the program counter.
- It stores the address of the next instruction to be fetched from the memory by the processor.
- It can be used in place of registers R0 to R14. Hence it is considered as one of the general purpose registers.
- Its use is sometimes restricted depending on the type of instruction.
- Generally it is used as a pointer to the instruction which is two instructions after the instruction being executed.
- We know that all ARM instructions are 32 bit long. i.e. they are aligned on word boundary. Hence the PC value is stored in bits [31:2] with bits [1:0] equal to zero.

(5) Registers R0-R7

- These registers are called as **unbanked registers**. They are general purpose registers and can be used wherever general purpose registers can be used.
- Each register is a 32 bit physical register in all the processor modes. The registers R0-R7 do not have any specific use.

(6) Registers R8-R14

- These registers are called as **banked registers.** This is because the physical size of each of these registers depends on the current processor mode.
- Whenever a general purpose register is used, any of these registers can be accessed.
- Out of 37 registers, only 20 registers which are shown shaded in Fig. 11.4.1 are used as banked registers.
- Fig. 11.4.1 also indicates the mode of the bank registers. Banked registers of a particular mode are denoted by r number_mode e.g. abort mode has banked registers r13_abt, r14_abt and spsr_abt.

Note : Register R8 to R12 are subject to restrictions on the use of bank registers in systems where FIQ mode is never used. Hence, only one physical version of the register is always in use.

- Registers R8 to R12 do not have any specific applications. However, for interrupts that are simple enough to be processed using these registers, the existence of separate FIQ mode versions of register allows fast interrupt processing.
- Each of registers R8 to R12 have two banked physical registers such that one register is used in all processor modes except FIQ mode and the other is used in FIQ mode.
- Registers R13 and R14 have six banked physical registers, such that one register is used in user and system mode while the each of remaining five registers is used in one of the five exception modes.
- Wherever it is essential to be specific about the version used, use the names of form : r13_mode e.g. r13_svc (supervisor mode).
- Register R13 is Stack Pointer (SP). In the ARM instruction set there are no specific instructions or functionality that use register R13. However in the Thumb instruction set there are such instructions.
- Each exception mode has its own banked version of r13, which should be initialized to point to a stack dedicated to that exception mode. The exception handler stores the values of other registers on the stack at the start. On return by reloading the values back to the registers will ensure that the exception handler will not corrupt the state of program which was in execution at the time when exception occurred.
- Register R14 is a link register. It can be used as a general purpose registers. It has two special functions. They are :
  - (a) In each mode, the modes own version of register R14 is used to hold the subroutine return address. The return task is achieved by loading R14 back to the program counter. This task can be obtained by methods given below :
    - (i) Execute either of these instructions :

      ```
      MOV  R15, R14
      ```

      or

      ```
      MOV  PC, LR
      ```

      OR

      ```
      BX  LR
      ```

    - (ii) While entering the subroutine, store the contents of register R14 to the stacking using instruction

      ```
      STMFD SP!, {<registers>, LR}
      ```

      and to return use instruction

      ```
      LDMFD SP! {<registers>, PC}.
      ```

  - (b) Incase of an exception, the exception mode's version of R14 is set to the exception return address. The exception return is done processed in the same way as the subroutine return, but using different instructions to ensure full restoration of the program state.

11.5 Program Status Registers

→ (MU - Dec. 14, May 16)

Q. 11.5.1 Explain CPSR of ARM 7 processor. (Ref. Sec. 11.5.1) MU - Dec. 14, May 16, 10 Marks

- The ARM architecture supports two program status registers. They are the **current program status register (CPSR)** and the **saved program status register (SPSR)**.
- The CPSR is accessible in all the processor modes while the SPSR is accessible in privileged modes.
- The CPSR contains condition code flags, interrupt disable bits, current processor mode and other control and status information.
- Each exception mode contains a saved program status register (SPSR). It is responsible for holding the value of CPSR incase the exception occurs. Fig. 11.5.1 shows the format of the CPSR and SPSR registers.

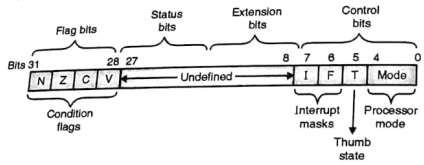


Fig. 11.5.1 : CPSR and SPSR register format

Now let us see description of each bit.

(1) Control Bits (Bits 0-7)

- Bits (0-7) are the control bits of CPSR and SPSR register. These bits change incase an exception occurs.
- The control bits can be altered by the software only when the processor is in privilege mode.

(2) Bits 0-4 (Processor Mode)

- These bits determine the processor mode. The modes are listed in Table 11.5.1.

Table 11.5.1 : Processor modes

Processor mode	Bits				
	4	3	2	1	0
User	1	0	0	0	0
Fast Interrupt Request (FIQ)	1	0	0	0	1
Interrupt Request (IRQ)	1	0	0	1	0
Supervisor	1	0	0	1	1
Abort	1	0	1	1	1
Undefined	1	1	0	1	1
System	1	1	1	1	1

(3) Bit 5 (Thumb Bit) (Architecture version 4T only)

- The Thumb bit determines the state of the ARM core. The state of core determines the instruction set that is being executed.
- There are 3 instruction sets : ARM, Thumb and Jazelle.
- When the T bit is 1, the processor is in the Thumb state and executes the Thumb instructions.
- Some ARM processor have additional bits to determine the state of the processor. e.g. J bit. The J bit is present in flags field is Jazelle enabled processors.
- In such processors the J and T bits decide the state of the processor.
- When both the T and J bits are zero, the processor is in the ARM state and executes ARM instructions. When the T bit is zero and J bit is 1, the processor is in Jazelle state and executes Jazelle instructions.
- To change the state of processor specialized branch instructions are used.

Comparison of ARM and Thumb instruction set

Sr. No.	Parameter	ARM instruction set	Thumb instruction set
1.	CPSR bit T	T = 0	T = 1
2.	Instruction size	32 bit	16 bit
3.	Core instructions	58	30
4.	Conditional execution	For most of the instructions	For branch instructions only.
5.	Program status register	Read-Write in privileged mode	No direct access.
6.	Register access	All 15 general purpose registers + PC	8 general purpose registers + PC + 7 high registers.
7.	Data processing instructions	Access to Barrel shifter and ALU	Separate Barrel shifter and ALU instructions.
8.	Versions	Supported by all versions	Supported to all versions after version 4
9.	Exception handling instructions	Supported	Not supported
10.	Instructions required to do a task	Less	More than ARM instructions

(4) Jazelle instruction set

- The Jazelle instructions are 8 bit in size. These instructions are designed with the view to increase the speed of execution of Java bytecodes.
- The Jazelle instructions are a hybrid mixture of software and hardware. Over 60% of the Java bytecodes are implemented in hardware while remaining are implemented in software.
- The Jazelle technology and a modified version of Java virtual machine are required to execute Java bytecodes.

- It is a closed instruction set. It is not openly available. To use Jazelle, a licensed software from ARM Limited and Sun Microsystems is required.

☞ **(5) Bits 6, 7 (Interrupt Masks)**

- The ARM core has two interrupts. They are fast interrupt request (FIQ) and interrupt request (IRQ).
- These interrupts are maskable. Bit 6 (F) controls the FIQ and Bit 7 (I) controls the IRQ. For their mask enable/disable
- When bit is set i.e. F = 1 or I = 1 the respective interrupt is masked or disabled. When bit is clear, the interrupt is unmasked or available.

☞ **(6) Condition code flags**

- The flags are updated by the operations performed by the ALU. The flags in the CPSR can be tested by the instructions to determine whether the instruction is to be executed or not.
- The flags can be updated by execution of a comparison instruction or by execution of arithmetic, logical or move instruction where the destination register is not R15.

☞ **(7) Bit 27 (Saturation Flag : Q)**

- It is present on the DSP extension of ARM processor core.
- This bit is set if an overflow or saturation occurs. The hardware can only set this flag. In order to clear this bit we need to write to CPSR directly.
- In case of SPSR this bit is used to hold and restore the CPSR Q flag incase of exceptions.

☞ **(8) Bit 28 (Overflow Flag : V)**

- This flag is set if an overflow occurs while addition or subtraction.
- For operations other than addition/subtraction overflow is unchanged.

☞ **(9) Bit 29 (Carry Flag : C)**

This flag is set if one of the following conditions exist.

(i) For addition alongwith the comparison instruction CMP, if the addition produced a carry otherwise flag is clear.

(ii) For subtraction alongwith the comparison instruction CMN, if subtraction produced a borrow, then the carry flag is cleared otherwise it is set.

(iii) For operations not involved with addition/subtraction but include a shift operation, the carry flag is set to the last bit shifted out of the value by the shifter. For operations that do not include a shift, carry flag remains unaltered.

☞ **(10) Bit 30 (Zero flag : Z)**

If the result of comparison is zero i.e. equal numbers are present then the zero flag is set to 1. Z flag is otherwise cleared.

☞ **(11) Bit 31 (Negative Flag : N)**

- The negative flag is set if the result is negative and regarded as two's complement of a signed integer.
- Otherwise the flag is cleared if the result is positive or zero.

- Other methods to modify the N, Z, C and V flags are listed below

(i) Execution of flag setting variants arithmetic and logical instructions such that the destination register is R15. The variants also copy the SPSR to CPSR. They are mainly used while returning from exceptions.

(ii) Execution of MRC instruction such that the destination register is R15

(iii) Execution of MSR instruction, as part of its function of writing a new value to CPSR or SPSR

(iv) Execution of some variants of LDM instruction.

☞ **(12) Reserved bits**

- These bits are reserved for further expansion in future
- The user should write his program in such manner that the bits are unmodified. If the user fails to do this it may lead to side-effects on future versions of the architecture.

11.6 Barrel Shifter

→ **(MU - Dec. 17)**

Q. 11.6.1 Explain in detail the barrel shifter of ARM 7 processor. (Ref. Sec. 11.6) **(5 Marks)**

Q. 11.6.2 Give details of Barrel Shifter of ARM 7 processor and the various operations carried out by the same. (Ref. Sec. 11.6) **Dec. 17, 10 Marks**

- The ARM processor does not support actual shift instructions. But instead it supports a **Barrel shifter**. The Barrel shifter is responsible for carrying the shifts as a part of other instructions
- The barrel shifter can be used to perform left shift operation by required amount, i.e. it multiplies by the powers of 2.

e.g. : LSL #4 (multiply by 16).

Fig. 11.6.1 : Logical shift left operation

- The barrel shifter can also to used to perform logical and arithmetic right.
- In logical shift right the barrel shifter shifts the bits to the right by specified amount, i.e. it divides by power of 2.

e.g. LSR #3 (divide by 8).

Fig. 11.6.2 : Logical shift right

- In arithmetic shift right the barrel shifter shifts the bits to the right by specified amount and preserves the sign bit, for 2's complement operations.

e.g. ASR #5 (divide by 32).

Fig. 11.6.3 : Arithmetic shift right

- The barrel shifter can also be used for performing rotations.
- In Rotate right (ROR) the barrel shifter performs operation similar to arithmetic shift right but wraps the bits around as they leave the LSB and appear as MSB. The last bit rotated is also used as carry out along with LSB.

e.g. ROR #2.

Fig. 11.6.4 : Rotate right

- For Rotate Right through carry or Rotate Right Extended (RRX) the barrel shifter uses the CPSR C flag as 33rd bit. In this operation the bits are rotated through the carry. The carry propagates to LSB and MSB to carry.

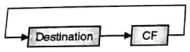


Fig. 11.6.5 : Rotate right through carry

- Using the barrel shifter alongwith ALU a wide range of addresses can be calculated.
- The contents of register Rm are preprocessed in the barrel shifter before applying them as input to the ALU. The register Rm can be a register, shifted register or immediate value.

11.7 Data Types

- The data types supported by the ARM processors are

  Byte : 8 bits.

  Half word : 16 bits. (halfwords should be aligned to two-byte boundaries)

  Word : 32 bit (words must be aligned to four-byte boundaries).
- The ARM instructions are one word instructions and Thumb instructions are half word instructions.
- All data operations are done on word quantities.
- All the three data types are supported in the ARM architecture version 4 and above. Prior to ARM architecture version 4 only bytes and words were supported.

11.8 Nomenclature

- The ARM processor uses a nomenclature that describes the processor implementations.
- The word "ARM" is followed by letters and numbers indicating features of the processor as given below :

  ARM {x}{y}{z}{T}{D} {M}{I}{E}{J} {F}{S}.

  Such that

  x : ARM family.

  y : Memory management/protection unit.

  z : Cache.

  T : Thumb instruction set.

D : Debug via JTAG interface.

M : Long multiply instructions.

I : Embedded ICE macrocell.

E : Enhanced instructions (DSP applications).

J : Jazelle.

F : Vector Floating point unit.

S : Synthesizable version.

e.g. : ARM7TDMI

- It implies that processor core is based on ARM7 family.
- It supports the Thumb instruction set. It supports debug via JTAG interface.
- It supports the long multiply instructions.
- It supports hardware breakpoints and watchpoints via the Embedded ICE macrocell.
- The nomenclature does not consist any information regarding architecture revision.
- A group of processors of processor family have the same hardware characteristics e.g. : ARM920T, ARM940T share ARM 9 family characteristics.
- The Thumb instruction set is supported for all versions after version 4. The variant T is automatic for ARM v6 and above.
- The core supports debug via JTAG interface. JTAG interface is described by IEEE 1149.1 standard Test Access Port and boundary scan architecture. It is used to send and receive debug information between processor core and the test equipment. The variant D is automatic for ARM v5 and above.
- The ARM core supports long multiply instructions for version 3 and above.
- Embedded ICE macrocell is the debug hardware that is built in processor for setting breakpoints and watchpoints.
- Enhanced DSP instructions are included in version 5 and above of ARM architecture. Variant E is included for all versions of ARM v 6 and above.
- The ARM core supports Jazelle Java acceleration architecture.
- It also supports vector floating point architecture.
- Synthesizable version implies that the processor core is supplied as source code that can be compiled into form easily used by EDA tools.

11.9 Processor Modes

Q. 11.9.1 List and explain processor modes of ARM 7 processor. (Ref. Sec. 11.9) **(5 Marks)**

- The ARM processor supports **seven** operating modes. The processor mode decides availability of the registers to the programmer and also access rights to the CPSR.
- The processor modes are classified as :

  (i) privileged mode (ii) non-privileged mode.

- Privileged mode supports full read and write access to the CPSR. The privileged modes supported by processor core are :
  (i) Abort (ii) Fast interrupt request
  (iii) Interrupt request (iv) Supervisor
  (v) Undefined (vi) System.
- Non-privileged mode allows only read access to the control field in the CPSR. However it allows read-write access to the condition flags. Non-privileged mode supported by processor core is :

☞ **(1) User**

(i) User mode

- It is a non privileged mode. Most of the tasks are executed in the user mode.
- Memory access is restricted in user mode. User cannot read directly from hardware device.

(ii) Fast Interrupt Mode (FIQ mode)

- This mode is entered whenever a high priority of interrupt is raised.
- FIQ mode is used to handle the peripherals that issue fast interrupts.
- This mode is privileged mode. The devices causing FIQs are floppy disc handling data, serial port etc.

(iii) Interrupt Mode (IRQ)

- This mode is a privileged mode.
- It is entered when a low priority interrupt is raised. e.g. keyboard, hard disk, floppy disc etc.

☞ **Difference between FIQ and IRQ**

- An IRQ may be interrupted by an FIQ but an FIQ cannot be interrupted by IRQ.
- With FIQ we have to do processing fast. For achieving this, the processor has shadow registers.
- FIQs cannot call software interrupts (SWIs).
- If it becomes essential for an FIQ routine to re-enable interrupts, it will take longer time than it would take by an IRQ.
- FIQ should disable interrupts.

(iv) Supervisor mode (SVC)

- The ARM processor enters this mode on rest. This mode can also be entered if a software interrupt (SWI) instruction is executed.
- Supervisor mode has additional privileges that allow greater control of the system.
- An operating system kernel operates in this mode.
- In order to read from a I/O Module, programmer has to enter the supervisor mode.

(v) Abort mode

The abort mode is entered whenever an attempt access memory fails.

(vi) Undefined mode (Undef)

This mode is use to handle undefined instructions.

(vii) System mode

- This mode is a privileged mode. It is a special version of the user mode.
- It allows full read-write access to the CPSR.
- It is present in ARM architecture version 4 and above.

Summary of ARM processor modes is given in Table 11.9.1 :

Table 11.9.1 : ARM processor modes

Mode	Denotation	Privileged	Enter this mode in case of Exception	Mode select Bits in CPSR				
				B4	B3	B2	B1	B0
User mode	usr	No	No	1	0	0	0	0
Fast Interrupt request	fiq	Yes	Yes	1	0	0	0	1
Interrupt request	irq	Yes	Yes	1	0	0	1	0
Supervisor mode	svc	Yes	Yes	1	0	0	1	1
Abort	abt	Yes	Yes	1	0	1	1	1
Undefined	undef	Yes	Yes	1	1	0	1	1
System	sys	Yes	No	1	1	1	1	1

Syllabus Topic : Pipelining

11.10 Pipeline

➔ **(MU - May 15, Dec. 15, May 16, Dec. 16, May 17)**

Q. 11.10.1 Explain in detail ARM 7 pipelining. (Ref. Sec. 11.10)

May 15, Dec. 15, May 16, Dec. 16, May 17, 10 Marks

- The process of fetching the next instruction while the current instruction is being executed is called as **"pipelining"**.
- Pipelining is supported by the processor to increase the speed of program execution. It also increases throughput.
- Several operations take place simultaneously, rather than serially in pipelining system.
- The pipeline has three stages fetch, decode and execute as shown in Fig. 11.10.1.

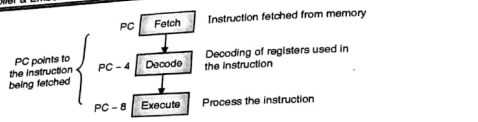
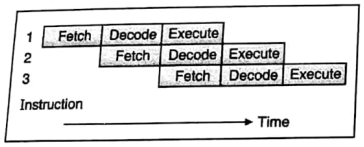


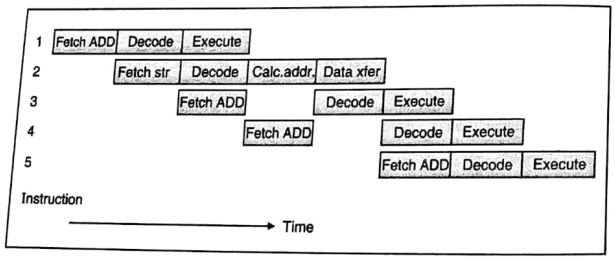
Fig. 11.10.1 : Three stage pipeline

- *The three stages used in the pipeline are :*
  - (i) **Fetch :** *In this stage the ARM processor fetches the instruction from the memory.*
  - (ii) **Decode :** *In this stage recognizes the instruction that is to be executed.*
  - (iii) **Execute :** *In this stage the processor processes the instruction and writes the result back to desired register.*
- *If these three stages of execution are overlapped, we will achieve higher speed of execution. Such pipeline exists in version 7 of ARM processor.*
- *Once the pipeline is filled, each instruction requires one cycle to complete execution.*

Fig. 11.10.2 shows a three stage pipelined instruction execution.



(a) Single cycle instruction execution for a 3-stage pipeline in ARM



(b) Multi cycle instruction execution for a 3-stage pipeline in ARM

Fig. 11.10.2 : Three stage pipelined instruction execution

- As shown in Fig. 11.10.2(a) in first cycle, the processor fetches instruction 1 from the memory. In the second cycle the processor fetches instruction 2 from the memory and decodes instruction 1. In the third cycle the processor fetches instruction 3 from memory, decodes instruction 2 and executes instruction 1. In the fourth cycle the processor fetches instruction 4, decodes instruction 3 and executes instruction 2. The pipeline thus executes an instruction in three cycles i.e. it delivers a throughput equal to one instruction per cycle.
- In case of a multi-cycle instruction as shown in Fig. 11.10.2(b), instruction 2 (i.e. STR of the store instruction) requires 4 clock cycles and hence the pipeline stalls for one clock pulse. The first instruction completes execution in the third clock pulse, while the second instruction instead of completing execution in fourth clock pulse completes the same in fifth clock pulse. Thereafter every instruction completes execution in one clock pulse as seen in the figure.
- The amount of work done at each stage can be reduced by increasing the number of stages in the pipeline. To improve the performance, the processor then can be operated at higher operating frequency.
- As more number of cycles are required to fill the pipeline, the system latency also increases. The data dependency between the stages can also be increased as the stages of pipeline increase. So the instructions need to be schedule while writing code to decrease data dependency.

The organization of an ARM processor with three stage pipeline consists of the following:

1. Register bank: This includes various registers as seen in the programmers model of ARM.
2. Barrel Shifter: We have also seen this barrel shifter that is used to do various operations as discussed earlier in this chapter.
3. The ALU: This unit performs the various arithmetic and logical operations.
4. Address register and incrementer: The register stores the address and the incrementer increments the same so as to point to the next instruction.
5. Data register: It is used as a buffer to store the data when written to the memory or read from the memory.
6. Instruction Decoder: As the name says, it decodes the instruction and issues the control signals accordingly. Hence, the Instruction decoder is associated with the control logic that issues the control signals.

- Fig 11.10.3 shows the implementation of the three stage pipelining in ARM. In the figure, the flow of the instruction shows the three stages of pipelining. The instruction is first fetched by the PC (at the top) giving address to the address register and is incremented using the incrementer, so that the PC points to the next instruction. The instruction is fetched through the data bus (at the bottom) and is given to the Instruction Decode and Control unit. This unit decodes the instruction, which is the second stage of the pipeline and then the instruction is executed by the ALU, multiplier and barrel shifter using the registers from the register bank.

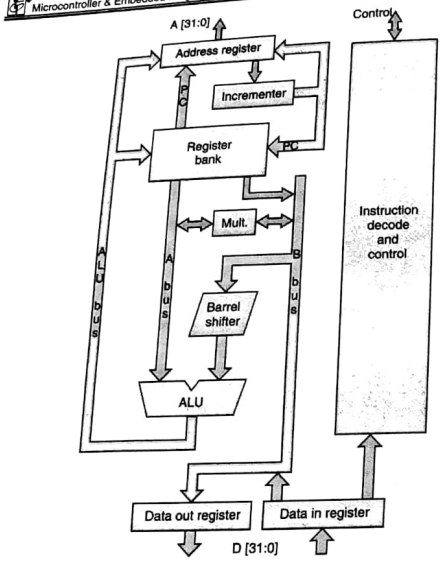


Fig. 11.10.3 : Organization of 3-stage pipelining in ARM

☞ ARM9 pipeline

- The pipeline for ARM9 processor is a five stage pipeline as shown in Fig. 11.10.4.
- The five stages of pipeline are fetch, decode and register read, execute shift and ALU, memory access and multiply, write register.
- The instruction throughput increased by 13% as compared to ARM7 three stage pipeline.
- The core frequency of ARM9 is greater than the ARM7 processor.
- In case of 5-stage pipelining, ARM processor also uses data forwarding technology. This enables the data to be forwarded internally from one stage to next and hence reduce the chances of pipeline stalling.

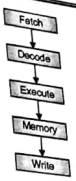


Fig. 11.10.4 : Five stage pipeline to be forwarded internally from one stage to next and of ARM9 TDMI processor

- Fig 11.10.5 shows the organization of the 5-stage pipeline organization in ARM9TDMI.
- In this case the fetching, decoding and execution are done in the same manner as in the 3-stage pipeline, by the respective bocks as seen in the Fig 11.10.5.

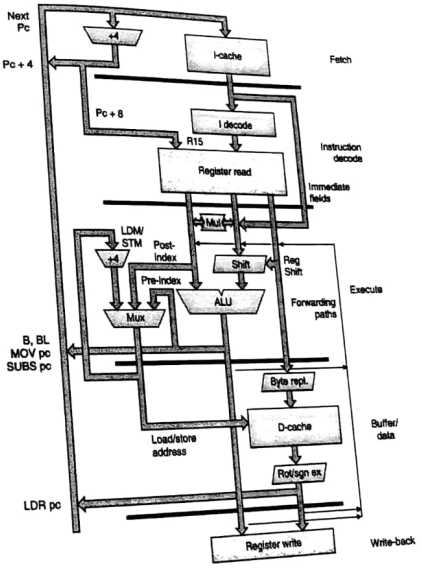


Fig. 11.10.5 : Organization of 5-stage pipeline in ARM

- The "I-cache" as shown in the figure is instruction cache and provides the instruction to the decode unit and hence is involved in the fetch operation as shown in the figure.
- The "I decode" decodes the instructions and forwards to the execute stage after allocating the register read operation if any.
- In the stage three i.e. the execute stage, the ALU performs the operation to be carried out according to the instruction.
- The stage four i.e. the Memory access is used to perform memory read or write in case of the load or store instructions respectively. This is done with the help of the "D-Cache" i.e. Data cache.
- The stage five i.e. the Write stage, is used to perform the write operation i.e. to update the register for the result to be written.

☞ ARM10 pipeline

- The pipeline of ARM10 processor has six stages fetch, issue, decode and read register, execute shift and ALU, memory access and multiply, write register as shown in Fig. 11.10.6 :
- The instruction throughput increases by around 34% more than ARM7 processor. The system latency also increases.

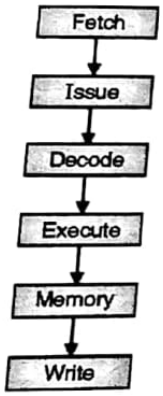


Fig. 11.10.6 : Six stage pipeline of ARM10 processor

Reset

When Reset pin is activated, the current execution is stopped by the processor. After reset the processor begins execution at an address 0x00000000 or 0xFFFF0000 is the supervisor mode with interrupts disabled.

Undefined instructions

- Whenever an attempt is made to execute an undefined instruction, an undefined instruction exception occurs.
- If a coprocessor instruction is executed, the processor waits for any external coprocessor to acknowledge that it can execute the instruction. Incase no external processor acknowledges, an undefined instruction exception occurs.

Software interrupt

Whenever a software interrupt instruction is executed a software interrupt exception is caused. The processor enters the supervisor mode inorder to request a particular supervisor function.

Prefetch abort vector

- This exception is generated if the processor tries to execute an invalid instruction. If the instruction is not executed then no prefetch abort exception occurs.
- This exception can also be generated as a result of executing a breakpoint instruction in ARM architecture version 5 and above.

Data abort

A memory abort is signalled by the memory system. Activating an abort in response to data access (i.e. load or store) marks the data as invalid.

Interrupt request (IRQ)

- This exception is generated by asserting the IRQ input on the processor. When the I bit in CPSR is set, this interrupt is disabled.
- It has a lower priority than the FIQ.
- If the I bit in CPSR is clear then the processor checks for an IRQ at instruction boundaries.

Fast interrupt request (FIQ)

- If the FIQ input is asserted, an FIQ exception is generated.
- FIQ supports data transfer or channel process and has sufficient private registers to remove the need for register saving in such applications minimizing the overhead of context switching.
- Fast interrupts are disabled if the F bit in CPSR is set. If the bit is clear then the processor checks for an FIQ at instruction boundaries.

High vectors

- Some of the ARM implementations allow the exception vector locations to be shifted from their normal address 0x00000000-0x0000001C to address 0xFFFF0000-0xFFFF001C. These locations are called as **high vectors**.

Exception priorities

Table 11.11.2 shows the exception priorities

Table 11.11.2

Priority	Exception
Highest 1	Reset
2	Data Abort
3	FIQ
4	IRQ
5	Prefetch Abort
Lowest 6	Undefined instruction, SWI

- The priority of a data abort exception is higher than FIQ. The priority of SWI and undefined instruction is same. But, both of them cannot take place at the same time.

Syllabus Topic : Addressing Modes

11.12 Memory Access and Addressing Modes

→ **(MU - Dec. 14, May 16, May 17)**

Q. 11.12.1 Explain addressing modes of ARM 7 processor. (Ref. Sec. 11.12) **MU - Dec. 14, May 16, May 17, 10 Marks**

Q. 11.12.2 List and explain addressing modes of ARM 7 processor. (Ref. Sec. 11.12) **(5 Marks)**

As already discussed Load and Store instructions are used to access memory. The different addressing modes to access memory and data processing instructions are shown in Fig. 11.12.1.

Addressing modes refer to the different methods of addressing (selecting) the operand. In ARM instruction set, the addressing modes are broadly divided into two categories viz for data processing operands and for memory access operands. Each of these categories have different methods as shown in Fig. 11.12.1.

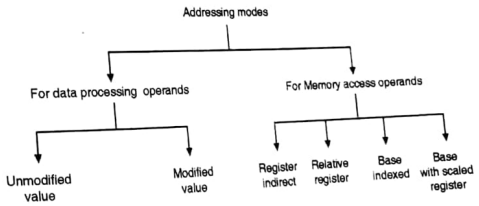


Fig. 11.12.1 : ARM addressing modes

Each of the memory addressing modes accept for register indirect can also be used with pre-index and post index as explained in the subsequent subsections.

11.12.1 Addressing Modes for Data Processing Operands (i. e. op1)

These are two method for addressing these operands :

➔ **1. Unmodified value**

In this addressing mode, the register or a value is given unmodified i.e. without any shift or rotation.

e. g. (i) MOV R0, # 1234 H

This instruction will move the immediate constant value $(1234)_{16}$ into register R0.

(ii) MOV R0, R1

This instruction will move the value in the register R1 into the register R0

➔ **2. Modified value**

In this addressing mode, the given value or register is shifted or rotated. These are different shift and rotate operations possible as listed below with examples.

(i) Logical shift left

This will take the value of a register and shift the value towards most significant bits, by n bits.

e.g. MOV R0, R1, LSL # 2

After the execution of this instruction R0 will become the value of R1 shifted by 2 bits.

(ii) Logical shift right

This will take the value of a register and shift the value towards right by n bits.

e.g. MOV R0, R1, LSR R2

After the execution of this instruction R0 will have the value of R1 shifted right by R2 times. R1 and R2 are not altered.

(iii) Arithmetic shift right

This is similar to logical shift right, except that the MSB is retained as well as shifted for arithmetic shift operation .

e.g. MOV R0, R1, ASR #2

After the execution of this instruction R0 will have the value of R1 Arithmetic shifted right by 2 bits.

(iv) Rotate right

This will take the value of a register and rotate it right by n bits

e. g. MOV R0, R1, ROR R2

After the execution of this instruction R0 will have the value of R1 rotated right for R2 times.

(v) Rotate right extended

This is similar to Rotate right by one bit, with the carry flag moved into the MSB, i.e. it is similar to rotate right through carry

e. g. MOV R0, R1 RRX

After the execution of this instruction R0 will have the value of register R1 rotated right through carry by 1 bit

11.12.2 Addressing Modes for Memory Access Operands

As already discussed load and store instructions are used to access memory. The different memory access addressing modes are :

(i) Register indirect addressing mode
(ii) Relative register indirect addressing mode
(iii) Base indexed indirect addressing mode
(iv) Base with scale register addressing mode.

Each of these addressing modes have offset addressing, Pre-index addressing and post-index addressing as explained in the examples for each addressing mode

➔ **(i) Register indirect addressing mode**

- In this addressing mode, a register is used to give the address of the memory location to be accessed.

  e. g. LDR R0, [R1]

- This instruction will load the register R0 with the 32-bit word at the memory address held in the register R1

➔ **(ii) Relative register indirect addressing mode**

- In this addressing mode the memory address is generated by an immediate value added to a register. Pre index and post index are supported in this addressing mode.

  e. g. (a) LDR R0, [R1, #4]

- This instruction will load the register R0 with the word at the memory address calculated by adding the constant value 4 to the memory address contained in the R1 register

  e.g. (b) LDR R0, [R1, #4]!

- This is a pre-index addressing. This instruction is same as that in e. g. (a) this instruction also places the new address in R1 i.e. R1 ← R1 + 4.

  e.g. (c) LDR, [R1], #4

- This is post-index addressing. This instruction will load register R0 with the word at memory address given in register R1. It will then calculate the new address by adding 4 to R1 and place this new address in R1

➔ **(iii) Base indexed indirect addressing mode**

- In this addressing mode the memory address is generated by adding the values of two registers. Pre-index and post-index are supported also in this addressing mode.

e. g. (a) LDR R0, [R1, R2]

- This instruction will load the register R0 with the word at memory address calculated by adding register R1 to register R2.

e.g. (b) LDR R0, [R1, R2]!

- This is pre-index addressing. This instruction is same as that in e.g. (a). this instruction also places the new address in R1 i. e. R1 ← R1 + R2.

e.g. (c) LDR R0, [R1], R2

- This is a post-index addressing. This instruction will load register R0 with the word at memory address given in register R1. It will then calculate the new address by adding the value in register R2 to register R1 and Place this new address in R1.

➔ **(iv) Base with scaled register addressing mode**

- In this addressing mode the memory address is generated by a register value added to another register shifted left. Pre-index and post-index are supported in this addressing mode.

e.g. (a) LDR R0, [R1, R2, LSL #2]

- This instruction will load the register R0 with the word at the memory address calculated by adding register R1with register R2 shifted left by 2 bits.

e.g. (b) LDR R0,[R1, R2, LSL #2]!

- This is a pre-indexed addressing. This instruction will load the register R0 with the word at the memory address calculated by adding register R1 with register R2 shifted left by 2 bits. The new address is placed in register R1.

i. e. R1 ← R1 + R2 << 2.

e.g. (c) LDR R0, [R1], R2, LSL #2.

This is a post-indexed addressing. This instruction will load the register R0 with the word at memory address contained in register R1. It will then calculate the new address by adding register R1 with register R2 shifted left by two bits. The new address is placed in register R1.

Syllabus Topic : ARM Development Tools

11.13 ARM Development Tools

➔ **(MU - May 15)**

Q. 11.13.1 Describe the flow of ARM development tools for embedded system design. (Ref. Sec. 11.13) **May 15, 10 Marks**

There are various tools available for the development of the ARM based programming. We can have an ARM development board and the programs written for the kit can be downloaded from the computer itself. The programming can be done using either assembly language programming or high level language programming like C. In case of programming is done by assembly language programming, we need an assembler. In case the programming is done using C language, we need compiler. The Fig. 11.13.1 shows the different ways of programming the kit and the tools required for the same. In the figure the sharp edged rectangles indicate the tools while the round edged rectangles indicate a file or a set of files like library.

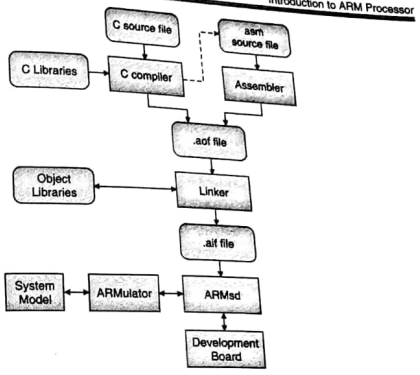


Fig. 11.13.1 : ARM cross-development toolkit

Hence as seen in the Fig. 11.13.1, the C source file is first given to a compiler which makes the use of the C library. The compiler converts the program to a .aof file (ARM Object File). If the assembly file (.asm file) is the source then the assembler converts the same to the .aof file. The .aof file is then given to the linker, whose job is to link all the branches and the function call operations to the right address. There can be various .aof file that can be linked together and hence a mixed language programming. The linker generates a .aif file (ARM Image Format file). The .aif file is then given to the ARMsd (ARM symbolic Debugger). The ARMsd can load, run and debug the program either on an emulator like ARMulator (an software emulator model integrated with the system model) or on the development board.

In the following sub-sections we will discuss these tools used for ARM system development.

11.13.1 ARM Assembler

The source to the assembler is the assembly file i.e. the near machine language instructions. It includes the ARM and the THUMB instructions. The assembler as discussed earlier gives a of file.

11.13.2 Compiler

The C compiler is mostly used for writing programs. The advantage of writing programs in C is that it is a very simple programming language and the complicated tasks can be easily implemented. The only disadvantage is that there are certain operations that are not possible in C and for that we need to use the assembly programming language.

- 'C' generates an object code that is extremely fast and compact, but it is not as fast as the object code generated by a good programmer using assembly language.

- It is true that the time needed to write a program in assembly language is much more than the time taken in Higher Level Languages like C.
- However, there are special cases where a function is coded in assembly language to reduce execution time. E.g. : The Floating Point math package must be coded in assembly language as it is used frequently and its execution speed will have great effect on the overall speed of the program that uses it.
- There are also occasions when some hardware devices need exact timing then it is necessary to write assembly level programs to meet such strict timing restrictions.
- In addition, certain instructions cannot be executed in Higher Level Languages like C. E.g. : C does not have an instruction for performing bit-wise rotation operation.
- Thus in spite of C being very powerful, routines must be written in assembly language to :
  - Increase the speed and efficiency of the routine.
  - Perform Machine specific functions not available in Microsoft C or in Turbo C.
  - Use third party routines.

☞ **Combining C and assembly**

There are 2 ways of combining C and Assembly language.

☞ **Method 1**

- Some compilers have built-in inline assembler that are used to include assembly language routines in the C-program.
- These assembly language routines are compiled along with the 'C' language routines rather and then linked together using the linker in the C Compiler.

☞ **Method 2**

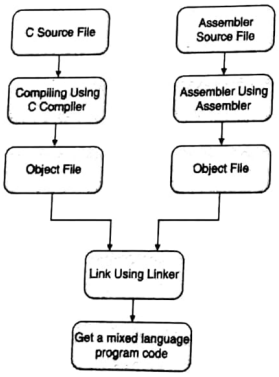


Fig. 11.13.2

- Some compilers do not have an in-line assembler. In such cases we need to write separate C routines and assembly routines.
- They are to be separately compiled or assembled. The C subroutines are to be compiled to give the object file; while the assembly files are tp be assembled to give the object file.
- The multiple object files generated in this manner are to be linked together to give a single executable file.
- This process is shown in the flowchart in Fig. 11.13.2.
- Fig. 11.13.2 shows Compile, Assemble, and link processes using C compiler, Assembler and LINKER.

11.13.3 Linker

The linker as discussed can link one or more .a of files or the object files. Hence the input to the Linker is one or multiple .aof files and its output is a .aif file. It uses the object libraries to perform its task. If the object files are compiled in such a manner that the full debug information is to be generated then the full symbolic debug tables will be generated when linking the source code.

11.13.4 ARMsd

It is a symbolic debugger. The ARMsd as discussed earlier can be used to emulate the ARM based programs written in C or in assembly languages. It can be either interfaced with the ARM development tool kit or with the ARMulator. The ARMulator will give a proper timing related idea and works as an emulator for ARM processor. The ARMsd can also be used to interface the code with the development toolkit and verify the working of the program written.

11.14 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : Architecture Inheritance**

Q. 1 Describe the principal features of the ARM architecture. **(Refer Section 11.1.4)** **(D-14, D-15, M-16, 6 Marks)**

Q. 2 Describe the features of ARM that makes it suitable for embedded system. **(Refer Section 11.1.4)** **(M-15, M- 16, D- 17, 6 Marks)**

☞ **Syllabus Topic : Detailed Study of Programmer's Model**

Q. 3 Explain detailed programmer's model of ARM 7. **(Refer Section11.4)** **(M- 16, 10 Marks)**

Q. 4 Explain CPSR of ARM 7 processor. **(Refer Section 11.5.1)** **(D-14, M- 16, 10 Marks)**

Q. 5 Explain in detail the barrel shifter of ARM 7 processor. **(Refer Section11.6)** **(5 Marks)**

Q. 6 Give details of Barrel Shifter of ARM 7 processor and the various operations carried out by the same. **(Refer Section11.6)** **(D- 17, 10 Marks**

Q. 7 List and explain processor modes of ARM 7 processor. **(Refer Section11.9)** **(5 Marks)**

☞ **Syllabus Topic : Pipelining**

Q. 8 Explain in detail ARM 7 pipelining. **(Refer Section 11.10)** **(M-15, D-15, M-16, D-16, M-17, 10 Marks)**

☞ **Syllabus Topic : Brief Introduction to Exceptions and Interrupts Handling**

Q. 9 Explain how Exceptions and Interrupts are handled in ARM7. **(Refer Section 11.11)** **(D-15, 10 Marks)**

☞ **Syllabus Topic : Addressing Modes**

Q. 10 Explain addressing modes of ARM 7 processor. **(Refer Section 11.12)** **(D-14, M-16, M-17, 10 Marks)**

Q. 11 List and explain addressing modes of ARM 7 processor. **(Refer Section11.12)** **(5 Marks)**

☞ **Syllabus Topic : ARM Development Tools**

Q. 12 Describe the flow of ARM development tools for embedded system design. **(Refer Section 11.13)** **(M-15, 10 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 12

ARM Processor Programming

12.1 Memory Access and Addressing Modes

→ (MU - Dec. 15)

Q. 12.1.1 Write short note on : Addressing Modes of ARM 7 Processor. (Ref. Sec. 12.1) Dec. 15, 7 Marks

As already discussed in previous chapters that only Load and Store instructions are used to access memory. The different addressing modes to access memory and data processing instructions are shown in Fig. 12.1.1

Addressing modes refer to the different methods of addressing (selecting) the operand. In ARM instruction set, the addressing modes are broadly divided into two categories viz for data processing operands and for memory access operands. Each of these categories have different methods as shown in Fig. 12.1.1.

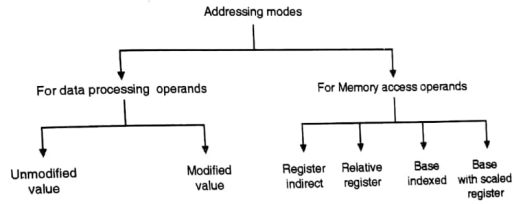


Fig. 12.1.1 : ARM addressing modes

Each of the memory addressing modes accept for register indirect can also be used with pre-index and post index as explained in the subsequent subsections.

12.1.1 Addressing Modes for Data Processing Operands (i. e. op1)

These are two methods for addressing these operands :

1. **Unmodified value :** In this addressing mode, the register or a value is given unmodified i.e. without any shift or rotation.

   e. g. (i) MOV R0, # 1234 H

   This instruction will move the immediate constant value $(1234)_{10}$ into register R0.

(ii) MOV R0, R1

This instruction will move the value in the register R1 into the register R0

2. Modified value

In this addressing mode, the given value or register is shifted or rotated. These are different shift and rotate operations possible as listed below with examples.

(i) Logical shift left

This will take the value of a register and shift the value towards most significant bits, by n bits.

e.g. MOV R0, R1, LSL # 2

After the execution of this instruction R0 will become the value of R1 shifted by 2 bits.

(ii) Logical shift right

This will take the value of a register and shift the value towards right by n bits.

e.g. MOV R0, R1, LSR R2

After the execution of this instruction R0 will have the value of R1 shifted right by R2 times. R1 and R2 are not altered.

(iii) Arithmetic shift right

This is similar to logical shift right, except that the MSB is retained as well as shifted for arithmetic shift operation .

e.g. MOV R0, R1, ASR #2

After the execution of this instruction R0 will have the value of R1 Arithmetic shifted right by 2 bits.

(iv) Rotate right

This will take the value of a register and rotate it right by n bits

e. g. MOV R0, R1, ROR R2

After the execution of this instruction R0 will have the value of R1 rotated right for R2 times.

(v) Rotate right extended

This is similar to Rotate right by one bit, with the carry flag moved into the MSB, i.e. it is similar to rotate right through carry

e. g. MOV R0, R1 RRX

After the execution of this instruction R0 will have the value of register R1 rotated right through carry by 1 bit

12.1.2 Addressing Modes for Memory Access Operands

As already discussed load and store instructions are used to access memory. The different memory access addressing modes are :

(i) Register indirect addressing mode
(ii) Relative register indirect addressing mode
(iii) Base indexed indirect addressing mode
(iv) Base with scale register addressing mode.

Each of these addressing modes have offset addressing, Pre-index addressing and post-index addressing as explained in the examples for each addressing mode

➔ **(i) Register indirect addressing mode**

In this addressing mode, a register is used to give the address of the memory location to be accessed.

e. g. LDR R0, [R1]

This instruction will load the register R0 with the 32-bit word at the memory address held in the register R1

➔ **(ii) Relative register indirect addressing mode**

In this addressing mode the memory address is generated by an immediate value added to a register. Pre index and post index are supported in this addressing mode.

e. g. (a) LDR R0, [R1, #4]

This instruction will load the register R0 with the word at the memory address calculated by adding the constant value 4 to the memory address contained in the R1 register

e.g. (b) LDR R0, [R1, #4]!

This is a pre-index addressing. This instruction is same as that in e. g. (a) this instruction also places the new address in R1 i.e. R1 ← R1 + 4.

e.g. (c) LDR, [R1], #4

This is post-index addressing. This instruction will load register R0 with the word at memory address given in register R1. It will then calculate the new address by adding 4 to R1 and place this new address in R1

➔ **(iii) Base indexed indirect addressing mode**

In this addressing mode the memory address is generated by adding the values of two registers. Pre-index and post-index are supported also in this addressing mode.

e. g. (a) LDR R0, [R1, R2]

This instruction will load the register R0 with the word at memory address calculated by adding register R1 to register R2.

e.g. (b) LDR R0, [R1, R2]!

This is pre-index addressing. This instruction is same as that in e.g. (a). this instruction also places the new address in R1 i. e. R1 ← R1 + R2.

e.g. (c) LDR R0, [R1], R2

This is a post-index addressing. This instruction will load register R0 with the word at memory address given in register R1. It will then calculate the new address by adding the value in register R2 to register R1 and Place this new address in R1.

➔ **(iv) Base with scaled register addressing mode**

In this addressing mode the memory address is generated by a register value added to another register shifted left. Pre-index and post-index are supported in this addressing mode.

e.g. (a) LDR R0, [R1, R2, LSL #2]

This instruction will load the register R0 with the word at the memory address calculated by adding register R1with register R2 shifted left by 2 bits.

e.g. (b) LDR R0,[R1, R2, LSL #2]!

This is a pre-indexed addressing. This instruction will load the register R0 with the word at the memory address calculated by adding register R1 with register R2 shifted left by 2 bits. The new address is placed in register R1.

i. e. R1 ← R1 + R2 << 2.

e.g. (c) LDR R0, [R1], R2, LSL #2.

This is a post-indexed addressing. This instruction will load the register R0 with the word at memory address contained in register R1. It will then calculate the new address by adding register R1 with register R2 shifted left by two bits. The new address is placed in register R1.

Syllabus Topic : Instruction Set

12.2 ARM Instruction Set

Q. 12.2.1 List the instruction set of ARM processor. (Ref. Sec. 12.2) **(2 Marks)**

The instruction set of ARM processor is divided into the following categories :

(1) Control Flow instructions i.e. Branch instructions and Branch with link instructions.
(2) Data Processing Instructions.
(3) Multiply Instructions.
(4) Load and Store Instructions or the Data transfer Instructions
(5) Program Status Register Access Instructions.
(6) Swap/Semaphore Instructions.
(7) Counting Leading Zeros Instruction.
(8) Exception/Software Interrupt generating Instructions.
(9) Multiple Register Transfer Instructions.
(10) Co-processor Instructions.

Syllabus Topic : Control Flow

12.2.1 Branch and Branch with Link Instructions (Control Flow Instructions)

Q. 12.2.2 Explain the following instructions with suitable examples w.r.t ARM processor : BL. (Ref. Sec. 12.2.1) **(2 Marks)**

Q. 12.2.3 Explain the following instructions with suitable examples w.r.t. ARM processor : BLX. (Ref. Sec. 12.2.1) May 17, 2 Marks

Q. 12.2.4 Explain the following instructions with suitable examples w.r.t ARM processor BX. (Ref. Sec. 12.2.1) Dec. 16, 2 Marks

(1) B. Branch

Mnemonic	B {< cond > } < target address > or B { < cond >} < label >	**Algorithm**	R15 = target address
Description	– The ARM instruction allows a conditional branch forward or backward upto 32 MB. – Upon entering a branch instruction the ARM processor will jump to the target address given in the instruction, and it will resume execution from there.		
Example : BCC label BNE label	– This instruction will branch to label if carry flag is clear. – This instruction will branch to label if zero flag is clear		

Note : The actual value stored in the branch instructions is an offset from the current value of R15, rather than an absolute address.

The various conditions possible in the branch instruction are listed in Table 12.2.1.

Table 12.2.1 : Conditions of branch instructions in ARM

Sr. No.	Branch Instruction		Branch on condition
	Mnemonic	Meaning	
1.	B	Unconditional	Always take this branch
2.	BAL	Always	
3.	BEQ	Equal	Comparison Equal or Zero result
4.	BNE	Not Equal	Comparison not equal or non-zero result
5.	BPL	Plus	Result is non-negative (positive or zero)
6.	BMI	Minus	Result is negative
7.	BCC	Carry clear	Carry is not generated i.e. comparison gave result as lower
8.	BLO	Lower	
9.	BCS	Carry Set	Carry is generated i.e. comparison gave result is higher or same
10.	BHS	Higher or Same	
11.	BVC	Overflow clear	No overflow occurred
12.	BVS	Overflow set	Overflow occurred
13.	BGT	Greater Than	Signed comparison gave result as greater than
14.	BGE	Greater or Equal	Signed comparison gave result as greater than or equal
15.	BLT	Less Than	Signed comparison gave result as less than

Sr. No.	Branch Instruction		Branch on condition
	Mnemonic	Meaning	
16.	BLE	Less or Equal	Signed comparison gave result as less than or equal
17.	BHI	Higher	Unsigned comparison gave result as higher
18.	BLS	Lower or same	Unsigned comparison gave result as lower or same

(2) BL : Branch with Link

Mnemonic	BL { < cond > } label.	Algorithm	R14 = R15, R15 = address of label.
Description	– This instruction loads the contents of register R15 to the LR (R14) register just before the branch i.e. it loads the return address for the branch in register R14. – This allows us to initiate subroutine calls. – The register R14 can be reloaded to register R15, to return to the main program execution after the execution of the branch.		
Example : BL func	– subroutine call to function.		

(3) BLX : Branch with Link and Exchange

Mnemonic	BLX < label >	Algorithm	R14 = R15, T = 1
Description	– This instruction is used to call a Thumb subroutine from the ARM instruction set at an address that is specified in the instruction. – This instruction is unconditional. It stores the address of the instruction following the branch instruction in the link register R14. – The instruction sets the T flag. The execution of the Thumb instruction begins at the address specified.		
Example : BLX T_Func	– Thumb subroutine call to function.		

(4) BX : Branch and Exchange Instruction

Mnemonic	BX {< cond > } < R_m >.	Algorithm	R15 = R_m
Description	– This instruction branches the program control to the address specified in register R_m, with an optional switch to Thumb execution. – The register R_m holds the value of branch address. – Bit ϕ of the register R_m is 0 in order to select a target ARM instruction or 1 to select a target Thumb instruction.		
Example : BX func	– This instruction will transfer the program control to target ARM instruction or target Thumb instruction depending on the status of Bit [0] of R_m register.		

Syllabus Topic : Data Processing

12.2.2 Data Processing Instructions

Q. 12.2.5 Explain data processing instruction in brief. (Ref. Sec. 12.2.2) **(6 Marks)**

The ARM instruction set has 16 data processing instructions. The data processing instructions include following instructions.

(i) Arithmetic instructions.
(ii) Comparison and test instructions.
(iii) Logical instructions.
(iv) Data movement between registers.

Table 12.2.2 lists the data processing instructions.

Table 12.2.2

Mnemonic	Operation	Action
ADD	Add	R_d : = R_n + <operand2>
ADC	Add with carry	R_d : = R_n + <operand2> + carry
SUB	Subtract	R_d : = R_n – <operand2>
SBC	Subtract with carry	R_d : = R_n – <operand2> – NOT (carry)
RSB	Reverse Subtract	R_d : = <operand2 > – R_n
RSC	Reverse Subtract with carry	R_d : = <operand2> – R_n – NOT (carry)
CMP	Compare	CPSR flags : = R_n – <operand2>
CMN	Compare Negated	CPSR flags : = R_n + <operand2>
TST	Test	CPSR flags : = R_n AND <operand2>
TEQ	Test Equivalence.	CPSR flags : = R_n EOR <operand2>
AND	Logical AND	R_d : = R_n AND <Operand2>
EOR	Logical Exclusive OR	R_d : = R_n EOR <operand2>
ORR	Logical OR	R_d : = R_n OR <operand2>
BIC	Bit Clear (Logical AND)	R_d : = R_n AND NOT <operand2>
MOV	Move	R_d : = <operand2>
MVN	Move Not	R_d : = NOT <operand2>

– The data processing instructions operate on registers only. They cannot be operated on memory.

– They perform a specific operation on one or more operands. The first operand is always register Rn and second operand is sent to the ALU through the barrel shifter.

– The compare and test instructions update the condition flags only. While the other data processing instructions stores the register into a register and update the condition flags.

– The operand2 is an immediate number or a register.

12.2.2.1 Arithmetic Instructions

Q. 12.2.6 Explain the following instructions with suitable examples w.r.t ARM processor : SBC. **(2 Marks)**
(Ref. Sec. 12.2.2.1)

Q. 12.2.7 Explain the following instructions with suitable examples w.r.t ARM processor : RSB. **(2 Marks)**
(Ref. Sec. 12.2.2.1)

Q. 12.2.8 Explain the following instructions with suitable examples w.r.t ARM processor : RSC. **(2 Marks)**
(Ref. Sec. 12.2.2.1)

The arithmetic instructions are listed in the Table 12.2.3.

Table 12.2.3 : Arithmetic instructions

Mnemonic	Operation	Action
ADD	Add	$R_d := R_n$ + <oprnd 2>
ADC	Add with carry	$R_d := R_n$ + <oprnd 2> + carry
SUB	Subtract	$R_d := R_n$ – <oprnd 2>
SBC	Subtract with carry	$R_d := R_n$ – <oprnd 2> – NOT (carry)
RSB	Reverse subtraction	$R_d :=$ <oprnd 2> – R_n
RSC	Reverse subtraction with carry	$R_d :=$ <oprnd 2> – R_n – NOT (carry)

Let us study these instructions one by one.

(1) ADD : Addition

Mnemonic	ADD {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	$R_d = R_n$ + <oprnd 2>, if condition is true, condition is optional
Description	$R_d \leftarrow R_n$ + <oprnd 2> – This instruction will add the two operands. The result is placed in the destination register R_d. – Operand2 can be a register, shifted register or an immediate number.		
Example : (i) ADD R0,R1,R2 R0 ← R1 + R2 (ii) ADD R0, R1, #256 R0 ← R1 + 256 (iii)ADD R0, R2, R3, LSL #1 R0 ← R2 + (R3 << 1)	– This instruction will add the contents of registers R1 and R2 and place the result in the destination register R0. – This instruction will add the contents of register R1 with the immediate value 256 and store the result in register R0. – This instruction will shift the contents of register R3 by 1 bit position to the left. Then it will add the contents of registers R2 and R3 and store the result in register R0.		

(2) ADC : Addition with Carry

Mnemonic	ADC {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	$R_d = R_n$ + <oprnd 2> + carry if the condition is true
Description	– $R_d \leftarrow R_n$ + <oprnd 2> + carry. – This instruction adds the two operands, placing the result in the destination register. It uses a carry bit, inorder to add numbers greater than 32 bits.		

Example :
Addition of 64 bit numbers.

64 bit result	;	registers R2 and R3.
64 bit first	;	registers R0 and R1.
64 bit second	;	registers R2 and R3.
ADDS R2, R2, R0	;	Add lower 32 bits and save status of carry.
ADC R3, R3, R1	;	Add upper 32 bits and carry of LSB addition.

Note : While adding numbers greater than 32 bits, suffix S is added, so that the status of carry flag is updated.

(3) SUB : Subtraction

Mnemonic	SUB {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	$R_d = R_n$ – <oprnd 2>, if condition is true
Description	– $R_d \leftarrow R_n$ – <oprnd 2> – This instruction will subtract the operand2 from register R_n, placing the result in the destination register. – The operand2 can be a register, shifted register or an immediate number.		
Example : SUB R0, R1, R2 R0 ← R1 – R2.	– This instruction will subtract the contents of register R2 from the contents of register R1, placing the result in register R0.		

(4) SBC : Subtract with Carry

Mnemonic	SBC {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	$R_d = R_n$ – <oprnd> – ! carry, if the condition is true.
Description	– $R_d \leftarrow R_n$ – <oprnd> – ! carry. – This will subtract the two operands, placing the result in the destination register. – The carry bit is used to represent "borrow" inorder to subtract numbers greater than 32 bits.		

Note : SUB and SBC generate the carry flag in a inverted way i.e. if a borrow is required then the carry flag is not set. Thus, this instruction requires a NOT carry flag. The carry flag is automatically inverted during the instruction.

(5) RSB : Reverse Subtract

Mnemonic	RSB {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	R_d = <oprnd 2> – R_n if condition is true.
Description	– $R_d \leftarrow$ <oprnd 2> – R_n – This instruction will perform reverse subtraction i.e. the contents of register R_n are subtracted from operand2 and the result is stored in the destination register. – Operand2 can be a register, shifted register or an immediate number. – The subtraction may be performed on signed or unsigned numbers.		

(6) RSC : Reverse Subtraction with Carry

Mnemonic	RSC {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	R_d = <oprnd 2> – R_n – ! carry if condition is true
Description	– This instruction will subtract the two operands, placing the result in destination register. – It is same like SBC instruction, except that subtraction is carried in reverse order.		

12.2.2.2 Comparison and Test Instructions

➔ **(MU - Dec. 16, May 17)**

Q. 12.2.9 Explain the following instructions with suitable examples w.r.t ARM processor : TEQ
(Ref. Sec. 12.2.2.2) **Dec. 16, 2 Marks**

Q. 12.2.10 Explain the following instructions with suitable examples w.r.t. ARM processor : CMN
(Ref. Sec. 12.2.2.2) **May 17, 2 Marks**

The comparison and test instructions are listed in Table 12.2.4.

Table 12.2.4 : Comparison and test instructions

Mnemonic	Operation	Action
CMN	Compare Negative	CPSR flags = R_n + <oprnd 2>
CMP	Compare	CPSR flags = R_n – <oprnd 2>
TEQ	Test Equivalence	CPSR flags = R_n EOR <oprnd 2>
TST	Test bits	CPSR flags = R_n AND <oprnd 2>

Let us see these instructions

(1) CMN : Compare Negative

Mnemonic	CMN {<cond>} R_n, <oprnd 2>	Algorithm	CPSR flags are updated on the operation R_n + <oprnd 2> i.e. R_n – <– oprnd 2>, if the condition is true
Description	– This instruction compares the contents of the register R_n with the logical NOT of operand i.e. against small negative values. – It performs a subtraction, but the result of comparison is not stored anywhere. The flags are updated. – The instruction does not affect the other registers.		

Example : CMN R0, R2	– This instruction will compare the contents of register R0 with R2 and update flags according to the result.

(2) CMP : Compare

Mnemonic	CMP {<cond>} R_n, <oprnd 2>	Algorithm	CPSR flags are updated on the operation R_n – <oprnd 2>, if the condition is true.
Description	– This instruction compares the contents of register R_n with the operand 2, updating the status flags. – It performs a subtraction, but the result is not stored anywhere. The flags are updated.		
Example: CMP R0, R1.	– This instruction will perform subtraction, R0 – R1, and update flags according to the result.		

(3) TEQ : Test Equivalence

Mnemonic	TEQ {<cond>} R_n, <oprnd 2>	Algorithm	CPSR flags = R_n EOR <oprnd 2>, if the condition is true
Description	– This instruction performs logical EOR (exclusive OR) operation on the two operands. The result of the operation is not stored anywhere. The result of operation is reflected in the status flags. – This instruction is used with the P suffix to alter the flags in R15 (in 26 bit mode). – It provides a way to see if the bits in both the operands are same or not without affecting the carry flag. – Operand2 can be a register, shifted register or immediate value.		
Example : TEQ R2, #3	– This instruction will logically perform EOR operation on register R2 and immediate number 3. The result will be updated in the status flags.		

(4) TST : Test bits

Mnemonic	TST {<cond>} Rn, <oprnd 2>	Algorithm	CPSR flags = R_n AND <oprnd 2>, if condition is true.
Description	– This instruction performs a logical AND operation on the two operands. The result updates the status flags. – This instruction is used to see if a particular bit is set. – After testing the zero flag will be set upon a match, otherwise it is cleared.		
Example : TST R0, R1, SL#3	– This instruction will logically AND R0 and left shifted contents of R1 by 3 bit positions and update the flags in CPSR according to the result.		

12.2.2.3 Logical Instructions

➔ **(MU - Dec. 16)**

Q. 12.2.11 Explain the following instructions with suitable examples w.r.t ARM processor : BIC.
(Ref. Sec. 12.2.2.3) **Dec. 16, 2 Marks**

Table 12.2.5 lists the logical instructions.

Table 12.2.5

Mnemonic	Operation	Action
AND	Logical AND	$R_d = R_n$ AND <oprnd 2>
ORR	Logical (inclusive) OR	$R_d = R_n$ OR <oprnd 2>
EOR	Logical Exclusive OR	$R_d = R_n$ EOR <oprnd 2>
BIC	Bit clear (Logical NAND)	$R_d = R_n$ AND NOT <oprnd 2>

Let us study these instructions one by one.

(1) AND : Logical AND

Mnemonic	AND {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	$R_d = R_n$ AND <oprnd 2>, if condition is true.
Description	– $R_d \leftarrow R_n$ AND <oprnd 2> – This instruction will logically AND the two operands and place the result in the destination register specified in the instruction. – This instruction is useful for masking the bits the user wants. – The operand2 can be a register, shifted register or an immediate number.		
Example : AND R0, R0, #3	– This instruction will logically AND the contents of register R0 with immediate data, 3 and store the result in register R0.		

AND Table

R_n	Operand2	Results
0	0	0
0	1	0
1	0	0
1	1	1

(2) BIC: Bit Clear

Mnemonic	BIC {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	$R_d = R_n$ AND NOT <oprnd 2>, if condition is true
Description	– $R_d \leftarrow R_n$ AND ! <oprnd 2>. – This instruction will logically AND the contents of register R_n and complemented contents of operand2. The result of operation will be stored in the destination register. – Operand2 is a 32 bit mask. If a bit is set in the mask, it will be cleared. Unset mask bits indicate bits to be left as it is. – This instruction provides a way to clear bits within a word.		
Example : BIC R0, R1, #%1011	– This instruction will clear the bits zero, one and three in register R1 and leave the remaining bits as it is. The result will be stored in register R0.		

(3) EOR : Logical Exclusive OR

Mnemonic	EOR {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	$R_d = R_n$ EOR <oprnd 2>, if condition is true
Description	– $R_d \leftarrow R_n$ EOR <oprnd 2>. – This instruction will perform a logical EOR operation between register R_n and operand2, the result will be stored in the destination register. – Operand2 can be a register, shifted register or an immediate value. – This instruction is useful for inverting some bits.		
Example : EOR R0, R0, #3.	– This instruction will logically EOR the register R0 with 3 i.e. it will inverts bits zero and one in register R0 and store the result in register R0.		

EOR Table

R_n	Operand2	Result
0	0	0
0	1	1
1	0	1
1	1	1

(4) ORR : Logical (Inclusive) OR

Mnemonic	ORR {<cond>} {S} R_d, R_n, <oprnd 2>	Algorithm	$R_d = R_n$ OR <oprnd 2>, if condition is true
Description	$R_d \leftarrow R_n$ OR <oprnd 2>. – This instruction will perform logical OR operation between register R_n and operand2 and the result will be stored in destination register R_d. – Operand2 can be a register, shifted register or an immediate value. – This instruction is useful for setting certain bit to be set.		
Example : ORR R0, R1, R2, LSL #3.	– This instruction will logically OR the register R1 with the left shifted contents of register R2 by three locations. The result of operation is stored in the register R0.		

OR Table

R_n	Operand2	Result
0	0	0
0	1	1
1	0	1
1	1	1

12.2.2.4 Data Movement between Registers

➔ (MU - May 17)

Q. 12.2.12 Explain the following instructions with suitable examples w.r.t. ARM processor : MVN. **May 17, 2 Marks**
(Ref. Sec. 12.2.2.4)

Table 12.2.6 lists the instructions use to transfer data between registers.

Table 12.2.6

Mnemonic	Operation	Action
MOV	Move	R_d = <operand2>
MVN	Move Not	R_d = ! <operand2>

(1) MOV : Move

Mnemonic	MOV {<cond>} R_d, <oprnd 2>	Algorithm	R_d = <oprnd 2>, if condition is true
Description	– $R_d \leftarrow$ <oprnd 2>. – This instruction loads value from operand2 to the destination register. – Operand2 may be a register, shifted register or an immediate value. – For the effect of NOP instruction the same register can be specified or shifted.		
Example : (i) MOV R0, R0 R0 ←R0 (ii)MOV R0, R1 R0 ← R1	– This instruction will load the contents of R0 from R0. – This instruction will load the contents of register R1 to register R0.		

(2) MVN : Move Negative

Mnemonic	MVN {<cond>} R_d, <oprnd 2>	Algorithm	R_d = ! <oprnd 2>, if condition is true
Description	– R_d = ! <oprnd 2>. – This instruction will load the inverted value of operand2 (i.e. negative number or 1's complement of number to the destination register.) – Operand2 can be a register, shifted register or an immediate number.		
Example : MVN R0, #4	– This instruction will load the register R0 with one's complement of 4.		

12.2.3 Counting Leading Zeros Instruction

Mnemonic	CLZ {<cond>} R_d, R_n
Description	– The versions 5 and above of ARM processor support this instruction. – It returns the number of 0 bits at the most significant end of its operand before the first '1' is encountered. – It is used to locate the highest priority bit in a bit mask. – It is used to determine how many bits the operand should be shifted left so that it can be normalized such that MSB is 1.

12.2.4 Multiplication Instructions

The multiplication instructions are different from the normal arithmetic instructions. This is because there are restrictions on the operands. The restrictions are :

(i) All the operands and the destination should be registers.

(ii) For the operand2 immediate values or shifted registers cannot be used. Operand2 must be a simple register.

(iii) The destination register and register R_m must both be different.

(iv) Register R15 cannot be used as the destination register.

There are two types of multiplication instructions. They are :

(I) Normal multiply instructions

– These instructions produce 32 bit results. The normal multiply instructions are listed in Table 12.2.7.

Table 12.2.7

Mnemonic	Operation	Action
MLA	Multiply and Accumulated	$R_d = (R_m * R_s) + R_n$
MUL	Multiply	$R_d = R_m * R_s$

– Both these instructions can optionally set the Negative (N) and Zero (Z) condition flags.

– There is no distinction between signed and unsigned variants.

– Register $\mathbf{R_m}$ specifies the register that contains the first operand to be multiplied. Register $\mathbf{R_s}$ specifies the second operand. Register R_n contains the value that is added to the product of R_s and R_m.

(a) MUL : Multiply

Mnemonic	MUL {<cond>} {S}, R_d, R_m, R_s,	Algorithm	$R_d = R_m * R_s$, if condition is true
Description	– $R_d \leftarrow R_m * R_s$. – This instruction multiplies the 32 bit values in the two registers and stores the result in destination register. – If the operands are signed, then the result is also signed.		

Example : (i) MUL R3, R2, R1. (ii) MULS R3, R2, R1. R3 ← R2 * R1.	– This instruction will multiply the contents of registers R1 and R2 and store the result in destination register R3. – It will perform the same operation as example 1 and in addition to that it will set the N and Z flags.

(b) MLA : Multiply and Accumulate

Mnemonic	MLA {<cond>} {S} R_d, R_m, R_s, R_n.	**Algorithm**	$R_d = R_m * R_s + R_n$, if condition is true
Description	– $R_d \leftarrow R_m * R_s + R_n$. – This instruction multiplies the values of two registers, adds the value of third register, truncates the result to 32 bit and stores the result in the destination register. – This instruction is useful for running totals.		
Example : MLA R4, R3, R7, R8. R4 = R3*R7 + R8.	– This instruction will multiply the two registers R3 and R7, add value of R8 to result of multiplication. The total result is truncated to 32 bits and stored in the destination register R4.		

(ii) Long Multiply Instructions

- These instructions produce 64 bit results. They can optionally set the N(negative) and Z (zero) condition code flags.
- There are 4 Long Multiply Instructions.
- Two of the variants multiply the values of two registers and store the 64 bit result in third and fourth registers. One of the variant is signed (SMULL) and the other variant is unsigned (UMULL). The signed variants produce a different result in MSB 32 bits if one or both of the operands is/are negative.
- The other two variants multiply the two registers, add the 64 bit value from third and fourth registers and store the 64 bit result back in those registers. One of variant is signed (SMLAL) and other is unsigned (UMLAL). These variants perform long multiply and accumulate.
- Table 12.2.8 lists the Long Multiply Instructions.

Table 12.2.8

Mnemonic	Operation	Action
SMLAL	Signed multiply and accumulate long	[RdHi, RdLo] = [RdHi, RdLo] + (R_m *R_s)
SMULL	Singed multiply long	[RdHi, RdLo] = R_m *R_s
UMULL	Unsigned multiply long	[RdHi, RdLo] = R_m *R_s
UMLAL	Unsigned multiply and accumulate long	[RdHi, RdLo] = [RdHi, RdLo] + (R_m *R_s)

nemonic : Instruction {<cond>} {S}, RdLo, RdHi, R_m, R_s.

Example	(i) SMULL R3, R4, R2, R1. R3 ← Bits 0 to 31 of R2 × R3. R4 ← Bits 32 to 63 of R2 × R3.	This instruction will multiply the lower 32 bits of registers R2 and R3 and store the LSB result in R3. Then it will multiply the upper 32 bits of registers R2 and R3 and store the MSB result in register R4.

12.2.5 Status Register Access Instructions

- For transferring the contents of program status register to a general purpose register or vice versa, two instructions are used. The CPSR and SPSR can be accessed.
- The instructions in this category are :

(i) MRS {<cond>}R_d, CPSR : Move data from CPSR to register R_d.

Mnemonic	MRS {<cond>} R_d, CPSR.	**Algorithm**	R_d = CPSR, if condition is true
Description	– $R_d \leftarrow$ CPSR – This instruction will transfer the contents of CPSR to destination register R_d i.e. it will read the CPSR.		
Example : MRS R0, CPSR.	– This instruction will copy the contents of CPSR in register R0.		

(ii) MRS {<cond>} Rd, SPSR : Move data from SPSR to register

Mnemonic	MRS {<cond>} R_d, SPSR.	**Algorithm**	R_d = SPSR, if condition is true
Description	– $R_d \leftarrow$ SPSR. – This instruction will transfer the contents of SPSR to destination register R_d. i.e. it will read the SPSR.		
Example : MRS R1, SPSR.	– This instruction will transfer the content of SPSR to register R1.		

(iii) MSR {<cond>} CPSR_<fields>, #<immediate> : Move immediate data to CPSR flags

Mnemonic	MSR {<cond>} CPSR_ <fields>, # <immediate>	**Algorithm**	CPSR = # 32 bit immediate data, if condition is true
Description	– CPSR ← # 32 bit immediate data. – This instruction will copy the 32 bit immediate data to the CPSR flags.		
Example : MSR CPSR_f, # FE002134H.	– This instruction will set the field mask bit in CPSR and copy 32 bit FE002134 H in the CPSR.		

Note : Fields is a sequence of one or more of the following

- c : It sets the control field mask bit (bit 16)
- x : It sets the extension field mask bit (bit 17).
- s : It sets the status field mask bit (bit 18).
- f : It sets the flags field mask bit (bit 19).

(iv) MSR {<cond>} CPSR_<fields>, R_m : Move data from register to CPSR

Mnemonic	MSR {<cond>} CPSR_<fields>, R_m	Algorithm	CPSR = R_m, if condition is true
Description	– CPSR ← R_m. – This instruction will transfer the contents of register R_m to CPSR flags. R_m is any general purpose register.		

Example

```
MRS R0, CPSR        ; Read CPSR
BIC R0, R0, # 0x1F  ; Clear mode bits
ORR R0, R0, # 0x11  ; Set mode bits to FIQ mode
MSR CPSR_C, R0      ; Update the control field mask bit in the CPSR. Processors will now be in FIQ mode
```

(v) MSR {<cond>} SPSR_<fields>, # <immediate> : Move immediate data to SPSR flags

Mnemonic	MSR {<cond>} SPSR_<fields>, # <immediate>	Algorithm	SPSR = 32 bit immediate data, if condition is true
Description	– SPSR ← 32 bit immediate data. – This instruction will copy the 32 bit immediate data to register SPSR.		

(vi) MSR {<cond>} SPSR_<fields>, R_m : Move data from register to SPSR flags

Mnemonic	MSR {<cond>} SPSR_<fields>, R_m	Algorithm	SPSR = R_m, if condition is true
Description	– SPSR ← R_m. – This instruction will transfer data from register R_m to SPSR flags. Register R_m is any general purpose register.		

12.2.6 Semaphore/Swap Instructions

➔ **(MU - May 17, Dec. 17)**

Q.12.2.13 Explain the following instructions with suitable examples w.r.t. ARM processor : SWP. (Ref. Sec. 12.2.6) **May 17. 2 Marks**

Q. 12.2.14 What is semaphore ? Explain the use of semaphore with respect to embedded operating systems. (Ref. Sec. 12.2.6) **Dec. 17. 5 Marks**

The Table 12.2.9 list the semaphore/swap instructions.

Table 12.2.9

Mnemonic	Operation
SWP	Swap Word
SWPB	Swap Byte

- These instructions are used for process synchronization
- They generate an automatic load and store operation, allowing a memory semaphore to be loaded and altered without interruption.
- Both the instructions support a single addressing mode.
- The swap instructions do not provide a compare and conditional write facility
- Inorder to specify the value to be stored and the destination where data is to loaded, separate registers must be used. If same registers are used for loading and storing then value is exchanged with value in memory

Mnemonic	SWP {<cond>} R_d, R_m, [R_n] : for swap word SWPB {<cond>} R_d, R_m, [R_n] : for swap byte.
Example : (i) SWP R1, R2, [R3] (ii) SWPB R2, R1, [R4]	– This instruction will load register R1 with the word addressed by register R3, and store register R2 at R3. – This instruction will load register R2 with the byte addressed by R4 and store bits 0 to 7 of R1 at R4.

12.2.7 Exception - Generating Instructions (Software Interrupt)

➔ **(MU - Dec. 16)**

Q. 12.2.15 Explain the following instructions with suitable examples w.r.t ARM processor : BKPT. (Ref. Sec. 12.2.7) **Dec. 16. 2 Marks**

The ARM instruction set supports two instructions whose main purpose is to cause a processor exception to take place. They are :

(i) SWI (Software interrupt)
(ii) BKPT (Breakpoint)

➔ **(i) SWI Software interrupt**

Mnemonic	SWI {<cond>} <imm_24>
Description	– This instruction causes a software interrupt process exception. – Using this mechanism user can make calls to privileged operating system code.

Note : <imm_24> is a 24 bit immediate value that is put into bits [23 : 0] of the instruction. The 24 bit value is ignored by the processors. It can be used by the operating system to determine which operating system service is being requested.

➔ **(ii) BKPT : Breakpoint**

Mnemonic	BKPT <imm>
Description	– This instruction is used for inserting software breakpoints. It causes a prefetch abort exception to occur. – It is used in the version 5 and above of ARM processor. – A debug monitor program that had been installed on prefetch abort vector can handle the exception caused.

Note : <imm> is a 16 bit immediate value. The 12 MSB bits are placed in bits 19-8 of the instruction and 4 LSB bits are placed in bits 3-0 of the instruction. This value is used by debugger to store information about the breakpoint.

12.2.8 Coprocessor Instructions

➔ **(MU - Dec. 16, May 17)**

Q. 12.2.16 Explain the following instructions with suitable examples w.r.t ARM processor : STC. *(Ref. Sec. 12.2.8)* **Dec. 16, 2 Marks**

Q. 12.2.17 Explain the following instructions with suitable examples w.r.t. ARM processor : LDC. *(Ref. Sec. 12.2.8)* **May 17, 2 Marks**

– For communication with the coprocessors the ARM processors, instruction set supports different instructions. They are :

(i) CDP : Coprocessor data operation
(ii) LDC : Load coprocessor register
(iii) MCR : Move to coprocessor from ARM register.
(iv) MRC : Move to ARM register from coprocessor.
(v) STC : Store coprocessor register.

– When the ARM processor executes a coprocessor instruction, it will offer it to any coprocessor that is attached to the system.
– All the coprocessor instructions can be conditionally executed.
– The coprocessor instructions allow the ARM processors to generate addresses for the coprocessor Load and Store instructions.
– They allow the ARM processor to initiate a coprocessor data processing operation.
– The coprocessor instructions support data transfer to/from ARM processors register to coprocessor registers.

(i) CDP : Coprocessor Data Operation

Mnemonic	– CDP {<cond>} <coproc>, <opcode_1> <CR_d>, <CR_n>, <CR_m>, <opcode_2> – CDP <coproc>, <opcode_1>, <CR_d>, <CR_n>, <CR_m>, <opcode_2>. Where, – <coproc> : Specifies the name of the coprocessor and causes the corresponding coprocessor number to be placed. The coprocessor names are p0, p1, p2, p3, ..., p15. – <opcode_1> and <opcode_2> : – Specifies the coprocessor operation that is to be done.
	– <CR_d> : Specifies coprocessor destination register for instruction. – <CR_n> : Specifies the coprocessor register that contains the first operand for the instruction. – <CR_m> : Specifies the coprocessor register containing the second operand.
Description	– It instructs the coprocessor to do some processing.
Example : CDP p4, 3, C12, C13, C3, 4	– This instruction indicates that it is a coprocessor 4 data operation such that opcode_1 = 3, opcode_2 = 4, destination register is C12 and source register are C13 and C3.

(ii) MCR : Move to Coprocessor from ARM register

Mnemonic	MCR <coproc> <opcode_1>, <R_d>, <CR_n>, <CR_m> <opcode_2> Where, – R_d : ARM destination register. – If R15 is used as destination register then the result is unpredictable. – CR_n : Destination coprocessor register. – CRm : Addition destination or source coprocessor register.
Description	– This instruction transfers data from an ARM register to a coprocessor register.
Example : MCR p3, 1, R8, C7, C5, 4.	– This instruction will transfer data from ARM register R8 to coprocessor 3 such that opcode_1 = 1, opcode_2 = 4, coprocessor destination registers are C7 and C5.

Note : Other fields have same meaning as CDP instruction.

(iii) MRC : Move to ARM register from coprocessor register

Mnemonic	– MRC <coproc>, <opcode_1>, <R_d>, <CR_n>, <CR_m>, <opcode_2> where fields have same meaning like MCR instruction.
Description	– This instruction will transfer data from a coprocessor register to an ARM register
Example : MRC p2, 5, R2, C0, C5, 3.	– This instruction will transfer data from coprocessor 2 to ARM register R2 such that opcode_1 = 5, opcode_2 = 3, Source registers are C0 and C5.

(iv) LDC : Load coprocessor register

Mnemonic	– LDC {<cond>} {L} <coproc>, <CR_d>, <address> – LDCL <coproc>, <CR_d>, <address>. Where, – L sets N bit (bit 21) in instruction to 1 indicating a long load instruction. If L bit is omitted, then the N bit is zero and instruction is a short load instruction.
Description	– This instruction loads data from memory to coprocessor register. If no coprocessor is present for execution of instruction, an undefined instruction exception is generated. – The data is loaded form the memory from the sequence of consecutive memory addresses calculated by the addressing mode. – The addressing mode can be one of the following : (i) immediate offset : [<R_n>, # + / – <offset_8> * 4] (ii) immediate preindexed : [<R_n>, # + / – <offset_8> * 4] ! (iii) immediate postindexed : <R_n>, # + / – <offset_8> * 4 (iv) unindexed : [<R_n>], <option>.
Example : LDC p2, CR0, [R3]	– This instruction will load the data from memory to coprocessor 2. ARM register R3 consists the address to load in register CR0

(v) STC : Store coprocessor register

Mnemonic	– STC {<cond>} {L} <coproc>, <CR_d>, <address> – STCL <coproc>, <CR_d>, <address> – Fields have same meaning as in LDC instruction.
Description	– This instruction stores data from a coprocessor register to memory. If no coprocessor is present for the execution of instruction, an undefined instruction exception is generated. – The data is loaded to the memory from the sequence of consecutive memory addresses calculated by the addressing mode. – The addressing mode can be one of the following : (i) immediate offset : [<Rn>, # + / – <offset_8> * 4] (ii) immediate pre-indexed : [<Rn>, # + / – <offset_8> * 4] ! (iii) immediate post indexed : [<Rn>] # + / – <offset_8> * 4. (iv) unindexed : [<Rn>, <option>]
Example : STC P2, CR8, [R3], # – 8	– This instruction will store data from coprocessor 8 to memory. The ARM register R3 holds the address after the transfer – R3 = R3 – 16. The data is stored from coprocessor register 8.

Syllabus Topic : Data Transfer

12.2.9 Load and Store Instructions or Data Transfer Instructions

– The ARM architecture supports two types of instructions which load the value of register from or to memory. They are :

(i) to load or store a 32 bit word or an unsigned byte.

(ii) to load or store a 16 bit unsigned halfword or load and sign extend a 16 bit halfword or a byte. These instruction are available in ARM architecture version 4 and higher versions.

12.2.9.1 Load and Store Word or Unsigned Byte Instruction

Mnemonic	– LDR / STR {<cond>} {B} {T} R_d, <address> Where, B : Distinguishes between an unsigned byte (B = = 1) and a word (B = = 0) access. T : Load with user mode privilege.
Description	– The load instructions load a single value from memory and store it on a general purpose register. – The store instructions read a value from the general purpose register and store it to the memory.
Example : (i) LDR R1, [R0] : (ii) LDR R12, [R14, # – 3] : (iii) STR R2, [R1, # 0x100] :	– This instruction will load value from address in register [R0] to ARM register R1. – This instruction will load value from address [R14] – 3 and store it on general purpose register R12. – This instruction will store the value in ARM register R2 to address in [R1] + 0x100.

12.2.9.2 Load and Store Halfword and Load Signed Byte

Mnemonic	– LDR / STR {<cond>} H / SH / SB. R_d, <address> Where H : Distinguishes between a halfword (H = = 1) and a signed byte (H = = 0) access. S : Distinguishes between a signed (S = = 1) and an unsigned (S = = 0) halfword access. **Note :** (i) If the L bit is zero and S bit is one, the instruction is unpredictable. (ii) If the S and H bit are both zero, then the instruction encodes Multiply or SWAP instruction.

Description

- The load instructions load data from memory and store it to an ARM register.
- The store instructions read data from an ARM register and store it to memory.

Example :

- Table 10.2.10 lists the instructions in this group.

Table 12.2.10

Mnemonic	Operation
LDRB	Load Byte
LDRBT	Load byte with user mode privilege.
LDRH	Load unsigned halfword.
LDRSB	Load signed byte
LDRSH	Load signed halfword
LDRT	Load word with user mode privilege
STRB	Store byte
STRBT	Store byte with user mode privilege.
STRH	Store halfword
STRT	Store word with user mode privilege.

(i) LDRB R4, [R8]	– This instruction will load byte from address in R8 to ARM register R4. The upper 3 bytes will be zero.
(ii) LDRH R1, [R0]	– This instruction will load the halfword from address in R0 to ARM register R1. The upper two bytes in register R1 will be zero.
(iii) LDRSB R3, [R4, #3]	– This instruction will load signed byte from address R8 + 3 to register R3.
(iv) LDRSH R4, [R8]	– This instruction will load the signed halfword from address in R8 to register R4.
(v) STRB R5, [R4, # – 2] !	– This instruction will store the byte form ARM register R5 to address R4 – 2 and then R4 = R4 – 2.
(vi) STRH R2, [R7, – R3]	– This instruction will store halfword from ARM register R2 to address R7 – R3

12.2.10 Multiple Register Transfer Instruction

➔ **(MU - Dec. 17)**

Q. 12.2.18 Explain multiple register load and store instructions of ARM7 processor. (Ref. Sec. 12.2.10) **Dec. 17, 10 Marks**

- The multiple register transfer instructions load or store a subset or all of the general purpose registers to or from the memory.
- The main use of these instructions is to dump registers that need to be preserved onto the stack.

Mnemonic

- LDM {<cond>} <address> R_n {!}, <R list> {^}
- STM {<cond>} <address> Rn {!}, <Rlist> {^}

R_n : express for evaluating valid register number.

<Rlist> : is a list of registers and register changes enclosed in s{ } e.g. {R1, R2-R7, R10}

{!} : if present requests write back (W = 1), otherwise W = 0.

{^} : if present set S bit to load the CPSR along with the PC or force transfer of user bank when in privileged mode.

12.3 Condition Execution

A very special feature of ARM instruction set is that almost all the instructions have a condition associated with it. The instruction will be executed only if the condition associated with is true. This concept is called as condition execution. This is not in many processors. The major advantage of this condition execution is that it dramatically reduces the branches and hence the pipeline stalls. Every instruction has the first four bits dedicated for condition check. Since there are 4-bits, we can have a maximum of 16 conditions to be checked. The Table 12.3.1 lists these conditions that can be checked by the ARM instructions.

Table 12.3.1 : ARM condition codes

Op code [31:28]	Mnemonic extension	Interpretation	Status flag stage for execution
0000	EQ	Equal/Equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/ Unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/Negative	N set
0101	PL	Plus/Positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HL	Unsigned higher	C set and z clear
1001	LS	Unsigned lower or same	C clear and Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always (need not to use !)	Any
1111	NV	Never (do not use !)	None

If nothing is mentioned in the instruction, then the "Always" case is used. The last case i.e. "Never" should never be used, else the instruction will never be executed. All other case, check a particular flag or status bit to decide, whether the instruction should be executed or not, based on the condition given. The conditions or the status flags can be checked to be set or clear. The different status bits that can be checked are Zero flag, Carry flag, Negative flag, Overflow flag and their combination as shown in Table 12.3.1

For example if the first case i.e. 'EQ' is used with ADD instruction, then the instruction will be written as ADDEQ in this case the ADD operation will be performed only if the condition 'EQ' is true i.e. zero flag is set to '1.'

12.4 The Thumb Programmer's Model and Instruction Set

12.4.1 Introduction to Thumb Instruction Set

The ARM7TDMI processor has two sets of instruction sets:

i) standard 32-bit ARM instructions, studied until this section

ii) 16-bit Thumb instruction set, to be studied in this section.

Thus the processor is said to operate in two states viz.

i) the ARM state to execute the 32-bit, word aligned ARM instructions

ii) the Thumb state to execute the 16-bit half-word aligned Thumb instructions.

The Thumb instruction set is available from the ARM versions above v4T. The functionality of these Thumb instructions is a subset of the functionality of the ARM instructions. In other words, the Thumb instructions do some of the operations that ARM instruction set can perform. But there are certain differences that will be seen in this section. An important thing to note is that the processor can execute the Thumb instructions when in Thumb state, while ARM instructions when in ARM state. The processor in ARM state cannot execute the Thumb instructions of 16-bit and also when the processor is in Thumb state it cannot execute the ARM instructions of 32-bit. There are instructions available in each set to switch the processor state. The default state of ARM processor when switched on is the ARM state. In Thumb instruction application, we have

1. Two operand (or address) instructions i.e. one of source is destination.
2. Conditional execution is not executed
3. Inline barrel shifter cannot be used
4. The size of the constants is limited
5. The lower part of registers are only used.

It should be noted that Thumb in itself is not a complete architecture of the processor, it needs the main state of processor operation. Also all the ARM development tools support the Thumb instruction set.

12.4.2 Switching between ARM State and Thumb State

Thumb applications are normally the 16-bit applications of embedded systems. The embedded system can be such designed that during the 32-bit ARM state the processor uses the on-chip 32-bit memory, while the processor uses the 16-bit off-chip memory when in the Thumb state. The Thumb state should be for large non-critical sub-routines.

The instruction *BX* is used for switching between the two states. The *BX* instruction stands for Branch and Exchange, which is discussed earlier in this chapter. This instruction sets the T bit in the program status register, if the bottom bit of the specified register was set and hence enters into the Thumb state. In this state the program counter switches address given by the remainder part of the register specified in the instruction. Thus this causes change in the value of the program counter i.e. a branching occurs and hence the flushing of the pipeline has to be done.

Switching back to the ARM state from the Thumb state can be done by the processor using the same instruction i.e. the *BX*. Also the processor returns automatically to the ARM state if the exception occurs in the Thumb state. When the processor returns from the Thumb state to the ARM state the previous value of the SPSR is copied into the CPSR register.

12.4.3 Thumb Programmer's Model

The Thumb programmers model includes the following:

1. The registers R0 to R7 (only lower part)
2. Few instructions require R8 to R15, some of which also have a special function.
   (i) R13 works as a stack pointer (ii) R14 works as a link register
   (iii) r15 works as a program counter.

Fig 12.4.1 shows the programmers model of the Thumb state of the ARM processor.

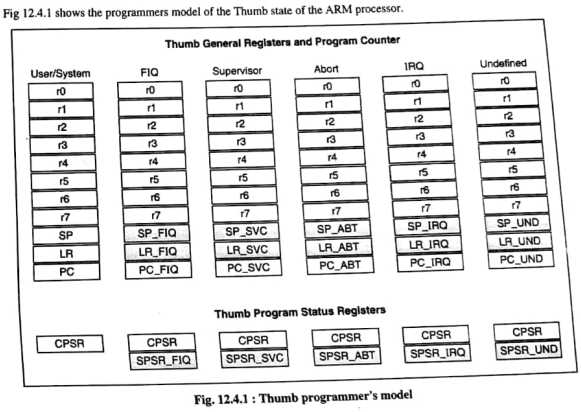


Fig. 12.4.1 : Thumb programmer's model

12.4.4 Thumb Instructions

The Thumb instructions as discussed earlier are similar to that of the ARM instruction set with 16-bit registers and the Thumb instruction set is a subset of ARM instruction set. The Table 12.4.1 shows some of the differences of ARM state and the Thumb state of the processor with respect to the instruction set

Table 12.4.1 : ARM and thumb instruction set differences

Sr. No.	ARM state	Thumb state
1.	All the instructions are 3-operand or 3-address format	Most of the instructions are 2-address or 2-operand format

Sr. No.	ARM state	Thumb state
2.	Many of the ARM instructions are executed conditionally	Many of the instructions in the Thumb mode are executed unconditionally
3.	The instruction formats of the ARM state do not have too many variations	The instruction formats of the Thumb state have a lot of variations

The following are some of the similarities of ARM state and the Thumb state of the processor.

(i) Both use the Load-Store architecture to access memory

(ii) They support 8-bit, 16-bit and 32-bit data

12.4.4.1 Branching Instructions of Thumb

The various branching instructions in the Thumb instruction set are as given below

1. Short conditional branch

The format of the short conditional branch is shown in the Fig. 12.4.2. It has 8-bit offset address that is added to the current value of Program counter if the condition is correct. It is used for short jumps like in loops of if-else condition implementation.

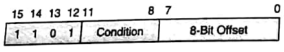


Fig 12.4.2 : Short conditional branch instruction format

2. Medium range unconditional branch

This format has a 11-bit offset address as shown in the Fig. 12.4.3. This value is added to the current value of program counter if the condition is correct. It is normally used for slightly longer jumps to implement instructions like *GOTO* of high level languages.

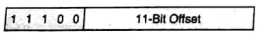


Fig. 12.4.3 : Medium range unconditional branch instruction format

3. Long range subroutine calls

In this case the instruction is used for branch and link operation. The 'H' bit controls operation for this instruction. If the 'H' bit is '0', then the Link Register gets a new value which is equal to the current value of PC added with the offset given in the instruction shifted 12 bits left. If the 'H' bit is '1', then the program counter register gets a value of Link Register added with the offset given in the instruction shifted one bit left; and the Link register gets the older value of the Program counter added with 3.

If H = '0', LR = PC + (sign extended offset shifted left 12 times)

If H ='1', PC = LR+ (offset shifted left one time)

LR = (Old value of PC) + 3

Fig 12.4.4 shows the format of the BL instruction.

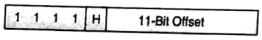


Fig. 12.4.4 : Long range subroutine call instruction format

4. *BLX* instruction is same as that of the ARM instruction set. This is studied earlier in this chapter.

12.4.4.2 Thumb Software Interrupt Instruction

The Fig. 12.4.5 shows the instruction format for the Thumb software interrupt.

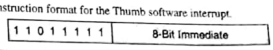


Fig. 12.4.5 : Thumb software interrupt instruction format

Whenever a software interrupt instruction is given in the Thumb mode, the following events occur:

1. The address of the next instruction to be executed is stored in the Link register i.e. r14_svc
2. The value of CPSR is copied into the SPSR.
3. The interrupt is disabled by clearing the IRQ bit and the processor enters into the supervisor mode.
4. The program counter is forced to the address 08H

Thus the software interrupt instruction executes the interrupt.

12.4.4.3 Thumb Data Processing Instructions

The Thumb instruction set for the data processing has both 3-operand and 2-operand instructions. The mnemonics of the instructions for data processing in the ARM state and the Thumb state are same. The operand options are either two address (operand) instructions of 3-address (operand) instructions

The data processing instructions of the Thumb instruction set has complex formats that cover most of the operations required by the compiler. The various formats of these data processing instructions are shown in the Fig. 12.4.6.

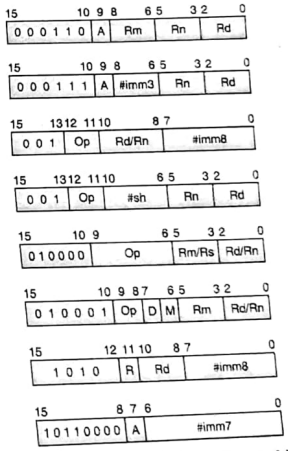


Fig. 12.4.6 : Data processing instruction formats of thumb

The examples for some of these instruction formats can be as given below in the same sequence.

1. The first format is a three address format i.e. has three operands. It is a register addressing mode i.e. all the operands are in the registers. Example of such an instruction is: ADD r3,r4,r5.
2. The second format is an immediate addressing mode example and has one immediate operand. This is also The example of such an instruction is: ADD r2,r3,#34

The remaining instructions are the two address format instructions.

The example of instruction of the format 6 is: ADD r3,r4 Similarly the example for the 7th instruction format is : ADD r3,#34

12.4.4.4 Properties / Features of Thumb Instructions

The features of Thumb instructions set with respect to ARM instructions set in terms of performance is as given below:

1. The Thumb instructions require only 70% of the memory space compared to that of the ARM instructions. Hence the power requirement for accessing the memory is also lesser by 30% in case of the Thumb instructions compare to the ARM instructions.
2. The Thumb code requires 40% more instructions to perform the same task as that of the ARM instructions.
3. Since ARM instruction set has 32-bit memory, it is 40% faster compared to the Thumb instructions to access memory.
4. Since Thumb instruction set has 16-bit memory, it is 45% faster compared to the ARM instructions for code execution.

Syllabus Topic : Writing Simple Assembly Language Programming

12.5 ARM Assembly Language Programming

- The assembly language instructions have a definite structure.
- The assembly language statement is divided into various fields. These fields are separated by spaces and tabs.
- The general format of an assembler instruction is as follows :

Label	Mnemonic	Operands	Comment
(Optional) used to identify the instruction	Instruction name and type	The data which is being operated upon	(Optional)

Label <space> opcode <space> operands <space> ; comment

- The comments allow user to annotate each line of source code in the program i.e. make it simple for user to understand the program.
- The following instruction illustrates the various fields clearly.

e.g.

```
up :   MOV   PC, LR   ;   Load data from R14 to R15
```

Label — Mnemonic — operands — comment

AREA, ENTRY and END Assembler Directives :

- Assembly language consists of a number of statements that help us to control the manner in which a source program assembles and lists. Such statements are called as **directives.**

Let us see the directives.

(i) **AREA** : This directive specifies the chunks of code or data that are manipulated by the linker.

One of more areas can exist in an application. A single CODE area is minimum required inorder to produce an application.

(ii) **ENTRY** : This directive marks the first instruction to be executed within an application. An application can consist a single entry point.

(iii)**END** : This directive marks the end of the module.

Programs

Program 12.5.1 : To transfer block of data from one address to other address.

Solution : We will transfer data from BLOCK 1 to BLOCK 2.

>> **Program**

Label	Instructions	Comments
	AREA BLOCKCOPY, CODE, READONLY	Declare code area.
	SWI_Write CEQU & 0	Output character in R0.
	SWI_Exit EQU & 11	End program.
	ENTRY	Code entry point.
START :	ADR R1, BLOCK 1	Load address of BLOCK 1 in register R1 R1.
	ADR R2, BLOCK 2	Load address of BLOCK 2 in register R2 R2.
	ADR R3, T1END	Load the address of END of BLOCK 1 in register R3
LOOP1 :	LDR R0, [R1], #4	Obtain 1st word of BLOCK 1.
	STR R0, [R2], #4	Store the 1st word of block 1 to BLOCK 2.
	CMP R1, R3	Is storing task finished ?
	BLT LOOP1	If not, continue else copy data.
	ADR R1, BLOCK 2	Load address of block 2 in register R1.
LOOP2 :	CMP R0, #0	
	SWI NE	Check for end of text string if not print and loop back
	SWI_Write C	
	BNE LOOP2	
	SWI	

Label	Instructions	Comments
	SWI_Exit	End.
BLOCK 1	"This is the correct string !", & 0a, & 0d, 0.	
B1END		
	ALIGN	Check word alignment.
BLOCK2	This is the wrong string !", 0	
	END.	

Note : SWI 0x00 : Write the character in bottom 8 bits of R0 to the display.
SWI 0x11 : exits the program.

Program 12.5.2 : Hex to ASCII conversion.

Solution :

Label	Instructions	Comments
	AREA Hex_ASCII, Code, Read only	Declare code area.
	SWI_Write CEQU & 0	Output character in R0.
	SWI_Exit EQU & 11	End program.
	ENTRY	
LDR :	R1, DATA	Load data in R1.
	BL Hex_ASCII	Call subroutine Hex_ASCII.
	SW SWI_Exit	End.
	DATA DCD & 87654321	
	Hex_ASCII	
	MOV R2, #8	Load R2 with iteration count.
LOOP :	MOV R0, R1, LSR #28	Move the upper 4 bits of R1 to R0.
	CMP R0, #9	Compare R0 with 9
	ADDGT R0, R0, #'A' – 10	If R0 > 9, add 37 H.
	ADDLE R0, R0, #0	If R0 < 9, add 30 H.
	SWI SWI_Write C	Print value in R0.
	MOV R1, R1, LSL #4	Shift R4 by four bits to left to display next nibble.
	SUBS R2, R2, #1	decrement 2
	BNE LOOP	If not zero, continue.
	MOV PC, LR	
	END	End.

Program 12.5.3 : To unpack bits 8-15 of register R0 and place the result in R1 register.

Solution :

Instructions	Comments
Area Unpack, Code, Readonly	Declare code area.
SWI_WriteC EQU & 0	Output character in R0.
SWI_Exit EQU & 11	End program
ENTRY	.
AND R1, R0, #00F0 H	Mask bits 8-15 of register R0 and store result in register R1.
SWI SWI_Exit	End.
END	

Program 12.5.4 : To divide a 32 bit number by an 8 bit number.

Solution :

Instructions	Comments
Area Unpack, Code, Readonly	Declare code area.
SWI_WriteC EQU & 0	Output character in R0.
SWI_Exit EQU & 11	End program
ENTRY	.
MOV R1, #23567890 H	Load 32 bit number in register R1. register R1.
MOV R2, #8	Load 8 bit number in register R2.
MOV R1, R1, LSR #3	divide 32 bit number by 8 bit number and store the result in register R1.
SWI SWI_Exit	End
END	

12.6 ARM Instruction Execution

12.6.1 5 Stage ARM9 Core Architecture

The ARM9 architecture is slightly different than the ARM architecture seen in the previous chapter. Fig 12.6.1 shows the 5-stage pipeline organization of the ARM9 processor.

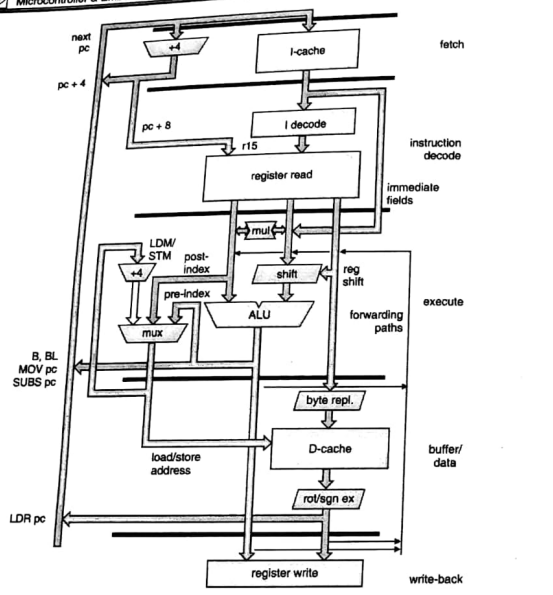


Fig. 12.6.1 : 5 stage pipeline of ARM9

12.6.2 Understanding the ARM Instruction Execution

The data and address flow for the execution of instructions in ARM9 processor is discussed in this module. The following common steps are taken for execution of all the instructions in ARM processor.

1. The Program counter issues the address that is given to the "increment" circuit. The increment circuit increments the value of program counter and gives the incremented value back to the program counter as well as to the address register from where it is issued on the address bus.
2. The instruction is then read from the memory and is stored into the "i.pipe".

The flow of the address and the instruction is shown by dark arrows in Fig. 12.6.2.

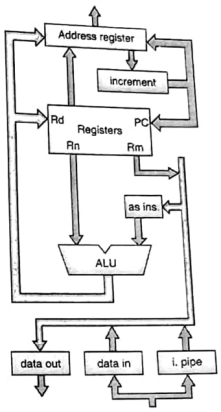


Fig. 12.6.2 : Instruction access in ARM9 processor

The further execution of the instruction depends on the operation to be performed. For a data transfer instruction, simply the data will be transferred from one place to another.

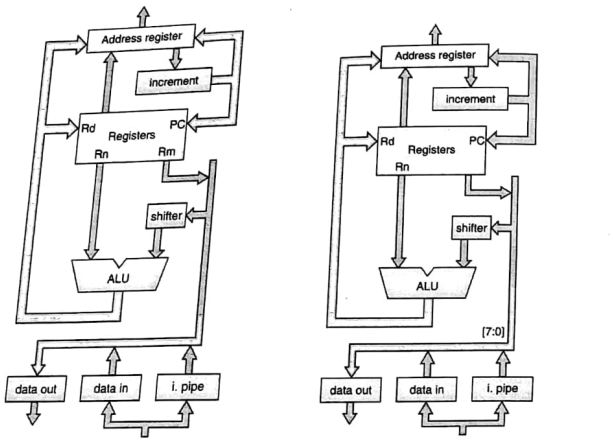
12.6.3 Execution of Data Processing Instruction in ARM

For a data processing instruction the ALU will perform the execution of the instruction as required. The initial two steps will be same as discussed earlier i.e. the Program counter issuing the address and fetching the instruction in i.pipe. Then the registers are read and the ALU performs the operation as specified in the instruction. The source operands are the R_m and the R_n registers. The result is then given back to the destination register i.e. R_d. The steps are mentioned below:

1. The Program counter issues the address that is given to the "increment" circuit. The increment circuit increments the value of program counter and gives the incremented value back to the program counter as well as to the address register from where it is issued on the address bus.
2. The instruction is then read from the memory and is stored into the "i.pipe".
3. The source operands are given to the ALU, with one operand as register R_n. The second operand can be either a register i.e. R_m or its value shifted / rotated as specified in the instruction. Fig 12.6.3 shows the flow with dark arrows, wherein the operation to be performed by the shifter and the ALU depends as specified in the instruction. There are two cases possible i.e. either the second operand is in the register or is an immediate data. These are shown in the Fig. 12.6.3 (a) and Fig. 12.6.3 (b)

4. Finally the result is stored in the destination register i.e. R_d

The flow of the address and the instruction is shown by dark arrows in Fig. 12.6.3. The two cases as discussed are based on the second operand being register or immediate data.



(a) Register to register operation (b) Register and immediate operation

Fig. 12.6.3 : Data processing instruction execution

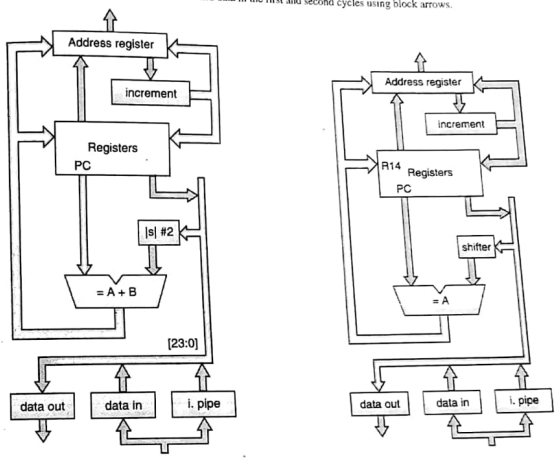
12.6.4 Execution of the Branch Instructions in ARM Processor

In this case again the initial process of fetching the instruction into the i.pipe is same as discussed in the earlier subsection. Thereafter the 24-bit immediate value after being shifted left twice is added to the current value of program counter. This value is given to the address register and the new instruction is fetched from there. Also the return address value from the program counter is given to the link register, if required. This value given to the link address is also adjusted to point to the next instruction. Thus it is a three cycle operation. The three cycles are required for the following operations:

1. The target address is computed in the first cycle. As discussed, this is done by adding the 24-bit immediate value left shifted twice with the value of PC.

2. In the second cycle the address result issued in the first cycle is fetched from and the instruction is filled in the i.pipe. Also in this cycle the return address is copied in the link register, if required.
3. In the third cycle a slight adjustment is done in the link address by decrementing its value with 4

Fig. 12.6.4 shows the flow of the address and data in the first and second cycles using block arrows.



(a) First cycle (b) second cycle

Fig. 12.6.4 : Branch instruction execution

12.7 Exam Pack (University and Review Questions)

Q. 1 Write short note on : Addressing Modes of ARM 7 Processor. **(Refer section 12.1)** **(D-15, 7 Marks)**

☞ **Syllabus Topic : Instruction Set**

Q. 2 List the instruction set of ARM processor. **(Refer section 12.2)** **(2 Marks)**

☞ **Syllabus Topic : Control Flow**

Q. 3 Explain the following instructions with suitable examples w.r.t ARM processor : BL. **(Refer section 12.2.1)** **(2 Marks)**

Q. 4 Explain the following instructions with suitable examples w.r.t. ARM processor : BLX. **(Refer section 12.2.1)** **(M-17, 2 Marks)**

Q. 5 Explain the following instructions with suitable examples w.r.t ARM processor BX. **(Refer section 12.2.1)** **(D-16, 2 Marks)**

☞ **Syllabus Topic : Data Processing**

Q. 6 Explain data processing instruction in brief. **(Refer section 12.2.2)** **(6 Marks)**

Q. 7 Explain the following instructions with suitable examples w.r.t ARM processor : SBC. **(Refer section 12.2.2.1)** **(2 Marks)**

Q. 8 Explain the following instructions with suitable examples w.r.t ARM processor : RSB. **(Refer section 12.2.2.1)** **(2 Marks)**

Q. 9 Explain the following instructions with suitable examples w.r.t ARM processor : RSC. **(Refer section. 12.2.2.1)** **(2 Marks)**

Q. 10 Explain the following instructions with suitable examples w.r.t ARM processor : TEQ. **(Refer section 12.2.2.2)** **(D-16, 2 Marks)**

Q. 11 Explain the following instructions with suitable examples w.r.t. ARM processor : CMN. **(Refer section 12.2.2.2)** **(M-17, 2 Marks)**

Q. 12 Explain the following instructions with suitable examples w.r.t ARM processor : BIC. **(Refer section 12.2.2.3)** **(D-16, 2 Marks)**

Q. 13 Explain the following instructions with suitable examples w.r.t. ARM processor : MVN. **(Refer section 12.2.2.4)** **(M-17, 2 Marks)**

Q.14 Explain the following instructions with suitable examples w.r.t. ARM processor : SWP. **(Refer section 12.2.6)** **(M-17, 2 Marks)**

Q. 15 What is semaphore ? Explain the use of semaphore with respect to embedded operating systems. **(Refer section 12.2.6)** **(D-17, 5 Marks)**

Q. 16 Explain the following instructions with suitable examples w.r.t ARM processor : BKPT. **(Refer section 12.2.7)** **(D-16, 2 Marks)**

Q. 17 Explain the following instructions with suitable examples w.r.t ARM processor : STC. **(Refer section 12.2.8)** **(D-16, 2 Marks)**

Q. 18 Explain the following instructions with suitable examples w.r.t. ARM processor : LDC. **(Refer section 12.2.8)** **(M-17, 2 Marks)**

☞ **Syllabus Topic : Data Transfer**

Q. 19 Explain multiple register load and store instructions of ARM7 processor. **(Refer section 12.2.10)** **(D-17, 10 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 13

Embedded / Real Time Operating System

Syllabus Topic : Basics of RTOS : Real-time Concepts, Hard Real Time and Soft Real Time

13.1 Introduction to RTOS

→ **(MU - Dec. 15, Dec. 16)**

Q. 13.1.1	Explain in brief Real Time Operating Systems. (Ref. Sec. 13.1)	**Dec. 15, 4 Marks**
Q. 13.1.2	Explain real time operating systems. (Ref. Sec. 13.1)	**Dec. 16, 3 Marks**

- The embedded software consists of the operating system and the applications. The operating system to be used in most of the cases is the Real time Operating System (RTOS).
- RTOS has two terms i.e. "Real time" and "Operating System". The term real time refers to the events with deadlines i.e. whenever an event occurs, there is a deadline for the event to be serviced and the event should be serviced before its deadline. The term operating system (OS) refers to the basic functions of the OS that it has to perform scheduling, synchronization, multitasking, memory management, inter-task communication etc.
- There are two types of RTOS based on the deadline of the tasks. They are called as Hard RTOS and Soft RTOS.
- The hard real-time system is required to complete the critical task within a guaranteed amount of time. Every process is submitted along with the time in which the task must be completed.
- Thus the scheduler does the following two operations:
  1. The first is allowing the process, guaranteeing that the process will be completed on time.
  2. The second is that rejection of the process i.e. resource reservation. In this case it is required that the scheduler knows the exact length of time each type of process that the operating system has to perform.
- Examples of where a hard RTOS would be used are where safety critical systems are required like the space shuttles of NASA.
- Soft real-time operating systems on the other hand are less restrictive by nature. A soft real-time system requires that the critical processes receive priority over less critical ones.
- Examples of applications where a soft RTOS may be used are multimedia devices, graphic devices or most commonly found appliances in our homes.

Table 13.1.1 : Comparison of general purpose computer and embedded system

Sr. No.	General purpose computer	Embedded System
1.	It is designed using a microprocessor as the main processing unit.	It is mostly designed using a microcontroller as the main processing unit.
2.	It contains large memory semiconductor memories like cache and RAM. It also contains secondary storage like hard disks etc.	It uses semiconductor memories, but normally doesn't require secondary memories like hard disks, CD etc. It sometimes has a special memory called as flash memory.
3.	It is designed such that it can cater to multiple tasks as per requirement.	It is designed such that it can cater to a particular predefined task.
4.	It is mostly costlier compared to the embedded systems.	It is mostly cheaper compared to the embedded systems.
5.	It is mostly costlier compared to the embedded systems.	It is mostly cheaper compared to the embedded systems.
6.	The Operating system and other software for the general purpose computers, are normally complicated and occupy more memory space.	The operating system (mostly RTOS i.e. Real time operating system) and other software occupy less memory space.

Syllabus Topic : Differences between General Purpose OS and RTOS

13.1.1 RTOS Vs GPOS

- Operating systems are the interface between the hardware of the system and the applications running on the system. In computers we use General Purpose Operating Systems (GPOS) while in embedded systems we use Real Time Operating Systems (RTOS).
- Following are the some key functional differences that keep RTOSes apart from General Purpose Operating Systems GPOS.
  1. The reliability of RTOS is more than GPOS in embedded application.
  2. RTOS has the ability to scale as per the needs of application.
  3. In embedded systems, the performance of RTOS is better
  4. RTOS requires less memory
  5. In embedded systems the scheduling policies are as per the real time requirement
- When the general purpose system such as desktop computer is turned on, general purpose operating system takes the control of desktop machine and then allows the user to run his/her application. In desktop computer the user application program separately compile and link from the general purpose operating system. In an embedded system, usually embedded software links with RTOS. At the boot up time, embedded software get the control first and it then starts the RTOS.

- Many RTOSes do not protect themselves as carefully from embedded software as do the general purpose operating systems. For example, most general purpose operating system check any pointer pass into a system validity, for many RTOSes skip this step in the interest of better performance.
- RTOSes typically include services that need the embedded system. Most RTOSes allows to configure them extensively before link to the embedded software application. Thus RTOSes are scalable.

Syllabus Topic : Advantages and Disadvantages of RTOS

13.1.2 Advantages and Drawbacks of RTOS

- A real time system requirement define the requirements of it's underlying RTOS. Some of key characteristics of an RTOS are :

1. Reliability	2. Predictability
3. Performance	4. Compactness
5. Scalability	

➜ **1. Reliability**

- Embedded systems must be reliable. Embedded system should be operated for long time without human intervention. The degree of reliability depends on application.
- For example, a digital solar powered calculator might reset itself, if it does not get the enough light, still everyone wants to use such calculator. On the other hand, telecom switch needs to remain functional even for no usage of it.
- Only the reliability of RTOS does not decide the reliability of system. Reliability of system depends upon the hardware, RTOS and application.

➜ **2. Predictability**

- The RTOSs written for the embedded system must be predictable i.e. the time required for servicing the task or interrupt should be predictable.
- The response time, the extra penalties required for the task must also be predicatble.

➜ **3. Performance**

- Performance of RTOS affects the functioning of real time system. Performance of real time embedded system depends upon the processing power of CPU and capability of RTOS to meet all the dead lines in the system.
- Sometimes embedded software developers measure RTOS performance on a call-by-call basis. Benchmarks program are written to measure time required to execute the system functions. This step can be useful in the analysis of design. However true performances testing is achieved only when the complete system response is measured.

➔ **4. Compactness**

- Application design and cost constraints determine the compactness of the embedded system. For example, a cell phone should be small, portable and low cost. These design requirements limit the system memory, which in turn limits the size of the application software and RTOS.
- In such constrained embedded systems, the hardware is limited due to size and costs, the RTOS clearly must be small and efficient. To meet total system requirements, designers must consider both the static and dynamic memory consumption of RTOS and application that will run on it.

➔ **5. Scalability :**

- RTOSes can be used in a wide variety of embedded systems, they must be flexible to scale up or down to meet application - specific requirements. Depending on how much functionality is required, an RTOS should have capability to add or delete modular components, including file systems and protocol stocks.
- If RTOS is not scable, then embedded system developer has to buy or build the required documents.

Syllabus Topic : RTOS Issues

13.1.2.1 RTOS Issues

- Embedded systems have to handle many problems/situation beyond those found in general purpose computer system. Embedded system often have several things to do simultaneously. They must respond to external events. They must handle with all unusual conditions without human intervention.

☞ **Unpredictability**

- Embedded systems are continuously interacting with real world through sensor and input/output devices which are connected to the input port of the systems. Any changes happening in the real world are captured by the sensors or input devices in real time.
- The real world event may be periodic one or an unpredicted one. If the event is unpredicted one such that the system should be designed in such a way that it should be scheduled to capture the events without missing them.

Syllabus Topic : Basic Architecture of an RTOS

13.2 Architecture of Kernel

➔ **(MU - Dec. 15, Dec. 17)**

Q. 13.2.1 List the Kernel objects and explain functions of each of the objects. (Ref. Secs. 13.2, 13.3, 13.4, 13.5, 13.7 and 13.8) **Dec. 15, 10 Marks**

Q. 13.2.2 Explain the architecture of the kernel of a Real Time Operating Systems. (Ref. Sec. 13.2) **(5 Marks)**

Q. 13.2.3 List functions of Kernel. Also explain different types of kernel. (Ref. Sec. 13.2) **Dec. 17, 10 Marks**

- Kernel is an interface between the application software and the hardware i.e. memory, I/O devices and CPU. Kernel is the heart of the operating system. Hence, the above discussed functions like memory management, I/O device management, interrupt handling, time management etc. of the OS are actually performed by the kernel in the OS.
- The following is a list of various kernel objects:
  1. Tasks,
  2. Task Scheduler,
  3. Interrupt Service Routines,
  4. Semaphores,
  5. Mutex,
  6. Mailbox,
  7. Message Queue,
  8. Pipe,
  9. Event Register,
  10. Signals,
  11. Timers, etc
- Fig. 13.2.1 shows the different objects of a kernel.

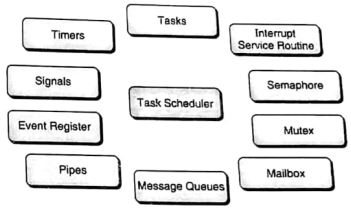


Fig. 13.2.1 : Objects of a kernel

- Each of these objects of the kernel will be discussed in details in the subsequent sections.

13.3 Task and Task Scheduler

➔ **(MU - Dec. 14, Dec. 15)**

Q. 13.3.1 What do you mean by Task and Task state related to embedded operating systems and also discuss about Task Control Block (TCB) and its data. (Ref. Secs. 13.3, 13.3.1 and 13.3.1.1) **Dec. 14, 10 Marks**

Q. 13.3.2 List the Kernel objects and explain functions of each of the objects. (Ref. Secs. 13.2, 13.3, 13.4, 13.5, 13.7 and 13.8) **Dec. 15, 10 Marks**

- There are many tasks in an embedded system and they are mainly the application software tasks and the OS tasks. Most of the tasks of the embedded system are in an infinite loop i.e. they reoccur after a given time.
- Each task object should have the following :
  1. Name of the task
  2. Unique ID of the task
  3. Priority of the task

4. Stack and
5. Task Control Block (TCB) that contains various information related to the task.

- The application software tasks have their priority and the tasks for the OS or the kernel also have their own tasks with priority. The different tasks of an OS or the kernel are:
  1. Startup task to be executed in the beginning of OS start
  2. Log Task that stores the log of different system messages
  3. Exception handling task to handle the exceptions
  4. Idle task which is of lowest priority and runs when no other task requires a service or no other task needs to run. Thus when no task is to be executed, the CPU executes this task and hence ensuring that the CPU is never idle.
- Task scheduling is very important as the CPU can execute only one task at any given instant. Thus one of the tasks can be accessing the CPU at any given time. The task scheduling should ensure that one task does not get a lot of CPU time, if other tasks are waiting for the CPU.
- Task scheduling refers to assigning priorities to different tasks and a mechanism that decides which task will get the CPU at any given time.
- The task scheduler besides doing the CPU time sharing also has to manage the sharing of system resources like memory, I/O devices, CPU registers etc. The task scheduler also has to protect the data of different tasks from other tasks.
- These resources that are to be shared by different tasks are called as shared resources. To manage these shared resources for avoiding data corruption we need objects like semaphores.
- A **re-entrant function** is a function that can be used by more than one task without corrupting the data of these tasks.
- If a function corrupts the data of different tasks when it is called by different tasks it is called as **non-re-entrant function**.
- Thus when executing a non-re-entrant function the code should not be interrupted and hence it is called as a critical section. For example if a function accesses a global variable, it will be considered as non-re-entrant function and hence while executing this function the task will be said to be in critical section. During the execution of the critical section, the interrupts are also to be disabled so that the data doesn't get corrupted.
- When a data is to be passed from one task to another we require inter-task communication. For inter task communication there are various objects available like message queues, pipes, event registers, mailboxes, signals etc.

13.3.1 Task States

➔ **(MU - Dec. 14)**

Q. 13.3.3 What do you mean by Task and Task state related to embedded operating systems and also discuss about Task Control Block (TCB) and its data. (Ref. Secs. 13.3, 13.3.1 and 13.3.1.1)

Dec. 14, 10 Marks

- A task consists of executable program or code controlled by the OS. For example displaying the data on the LCD display will be done by a small code or program that can be termed as one task.
- The task as discussed earlier is associated with a task object that also stores the state of the task. The various states of a task are running state, waiting state and ready to run state.

- The task switching state diagram is shown in Fig. 13.3.1.

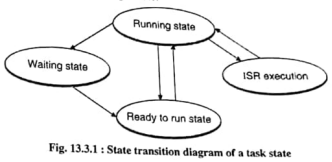


Fig. 13.3.1 : State transition diagram of a task state

- The task state transition is required to handle multiple tasks in the system. Thus if there are various tasks, whenever a task is to be executed, it will be in the "ready to run" state.
- According to the scheduling method used, whenever it is the turn for the said task, it will enter into the "running state" i.e. the task will utilize the resource CPU.
- During the execution of the task, if it requires to execute the ISR, the same will be executed and the task state will return to the "running state".
- In case if the task is to wait for some shared resource the task will be sent to the "waiting state" and another task will be in running state. The waiting task will be released back to the "ready to run" state whenever the required shared resource is made available.
- Once the time of the task for "running state" is over and there is another task in "ready to run state", the earlier task will be switched to "ready to run state" and the next task based on priority will be switched to the "running state".
- A task that is currently being executed by the CPU is said to be in running state. At any given instant, only one of the many tasks can be in this state.
- A task that is waiting for a shared resource or an event to occur, is said to be in waiting state. There can be multiple tasks in this waiting state at any given time.
- The various functions required to manage a task in the kernel are

1. Create a task	2. Delete a task
3. Suspend a task	4. Resume a task
5. Change the priority of a task	6. Query a task

- A task that is ready to be executed i.e. task that is neither waiting for a resource or event nor is in the running state as there is some other task that is being executed by the processor, then such a task is said to be in ready to run state. There can be multiple tasks in this state, and one of these tasks will enter the running state based on the priority mechanism that will be discussed later.

13.3.1.1 Task Control Block (TCB)

➔ **(MU - Dec. 14)**

Q. 13.3.4 What do you mean by Task and Task state related to embedded operating systems and also discuss about Task Control Block (TCB) and its data. (Ref. Secs. 13.3, 13.3.1 and 13.3.1.1)

Dec. 14, 10 Marks

When a new task is created, the Kernel also creates and maintains an associated Task Control Block (TCB). TCBs are system data structures that the Kernel uses to keep task specific information. TCBs contains all the information everything a Kernel needs to know about a particular task. When a task is running, its context is frequently updating.

This dynamic context is maintained in the TCB. When the task is not running, it's context is saved within TCB to be restored the next time when the task runs. When the Kernel's scheduler evaluates that it needs to stop running task 1 and start running task 2, it takes the following action.

1. The Kernel saves task 1's context information in it's TCB.
2. It loads task 2's context information from it's TCB, which becomes the current thread of execution.
3. The context of task 1 is saved while task 2 executes, but if the schedular needs to run task 1 again, task 1 continues from where it left off just before context switch by restoring it's TCB.

13.3.2 Context Switching

- Whenever another task (say task 2)is to be executed the task which is already in execution (say task 1) has to be shifted to "ready to run state". In this case the context of this task 1 has to be stored somewhere, so that next time when it is to be again executed, the context can be retrieved.
- Each task will have its own stack to store the context i.e. the local variables, function parameters, CPU register values and the return address. This stack size should be optimum so as to store all the required data and still not waste memory space. This stack will be used when task switching or when an interrupt occurs.

Syllabus Topic : Scheduling Systems

13.3.3 Task Scheduling

→ **(MU - Dec. 15, Dec. 17)**

Q. 13.3.5 List task scheduling algorithms. Explain operation of each of the algorithms. (Ref. Sec. 13.3.3) **Dec. 15, 10 Marks**

Q. 13.3.6 Explain any 3 scheduling algorithms. (Ref. Sec. 13.3.3)

Q. 13.3.7 Illustrate scheduling algorithms of tasks in real time systems. (Ref. Sec. 13.3.3) **Dec. 17, 10 Marks**

- The process of deciding which task will utilise the CPU time is called as task scheduling.
- The scheduling of the tasks may be on the basis of their priorities.
- The priority assignment mechanism for the tasks can be either static or dynamic. In case of static priority assignment, the priority for a task is assigned as soon as the task is created, thereafter the priority cannot be changed. In case of dynamic priority assignment the priorities of the tasks can be changed during the runtime.
- Rate Monotonic Analysis (RMA) is a good method to be used for assigning priorities. RMA assumes the following :

1. The task with highest priority will run first i.e. in case a lower priority task (say task 1) is in running state and a higher priority task (say task 2)is in ready to run state, then task 2 will be given the CPU time and the task 1 will have to go to ready to run state (this is called as pre-emptive scheduling)
2. The tasks are periodic i.e. the tasks reoccur are regular intervals.

3. None of the tasks share resources i.e. no task synchronisation is required.

- In RMA the priorities are assigned on the basis of the frequency of the task's frequency i.e. its reoccurrence time. For example if a task "i" requires CPU time of say "T_i" and the time of reoccurrence is "E_i", then $\frac{E_i}{T_i} \times 100$ indicates the percentage CPU time required for that particular task.
- "Schedulability test" in RMA will give an indication of the total CPU time utilised, and is given by

$$\sum \frac{E_i}{T_i} \leq U(n) = n(2^{1/n} - 1)$$

- Here, n is the number of tasks and U(n) is called as the utilization factor.
- There are various scheduling algorithms followed:

1. First In First Out (FIFO)
2. Round robin
3. Round robin with priority
4. Shortest job first
5. Non-pre-emptive scheduling
6. Pre-emptive scheduling

13.3.3.1 First In First Out (FIFO) Algorithm

- In this algorithm, simply the task that came in first in the ready to run state will go out first into the running state. The task that goes out of the running state goes to the ready to run state.
- Fig. 13.3.2 shows the movement of the tasks in the ready to run and running state.

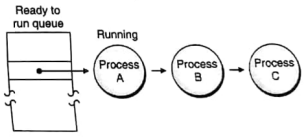


Fig. 13.3.2 : FIFO scheduling

- The advantage of this scheduling is that it is very simple to implement.
- The disadvantage of this scheduling is that we cannot have any priority mechanisms. In real time examples, each task has a different priority and it is to be implemented, but using FIFO we cannot implement priority based scheduling

13.3.3.2 Round Robin Scheduling

- In this case each task gets its turn after all the other tasks are given their time slots. Thus it can be said that it is a time slicing wherein the time for each time slot is same and is given to the tasks one-by-one.

- The CPU time utilization for three tasks according to round robin is shown in Fig. 13.3.3. In this example, it is assumed that the time slots are of 5 milliseconds (5ms) each.

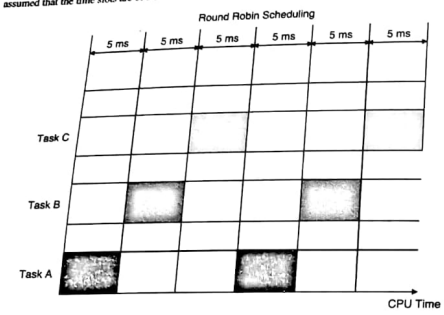


Fig. 13.3.3 : Round robin scheduling

- The switching of the tasks occurs in the following cases :
  1. Current task has completed its work before the completion of its time slice.
  2. Current task has no work to be done.
  3. Current task has completed the time slice allocated to it.
- The advantage of this algorithm is also that it is very easy to implement.
- The disadvantage is again that all the tasks are considered at same level.
- This scheduling method can be used for small devices like microwave ovens, digital multimeters etc.

13.3.3.3 Round Robin Scheduling with Priority

- It is a modified version of round robin scheduling mechanism.
- In this case the tasks are given priorities based on their significance and deadlines. Thus tasks with higher priority can interrupt the processor and utilize the CPU time.
- If multiple tasks have same priority then round robin scheduling will be used for them. But whenever a higher priority task occurs, it will be executed first. The CPU will suspend the task it was executing and execute the higher priority task.
- For example, bar code scanner can use this scheduling mechanism. This method can be used in soft real-time systems.

13.3.3.4 Shortest Job First (SJF) Scheduling

- In this case, the task with the shortest execution time is executed first. This ensures less number of tasks in the ready to run state. Thus it can be said that the priority is higher for a task with lesser execution time and the priority of the task is lesser for the task with higher execution time.
- The disadvantage of this scheduling method is that, if there are too many short execution time tasks, then the task with more execution time will never be executed.
- The advantage is that the implementation of this scheduling method is also simpler as just the execution time of each of the tasks in ready to run state are to be compared to decide which task will be executed by the processor.

13.3.3.5 Non-pre-emptive Scheduling

Q. 13.3.8 Explain with examples Non- pre-emptive Scheduling. (Ref. Sec. 13.3.3.5) **(5 Marks)**

- This scheduling mechanism can be implemented in any of the previously seen scheduling mechanisms that have the concept of priority.
- As the name says, in this case if a task (say task 1)is in running state and another task (say task 2) with higher priority enters into the ready to run state, the earlier task (i.e. task 1) continues the execution until its time slice and the higher priority task (i.e. task 2) has to wait for its turn. The Fig. 13.3.4 shows an example of non-pre-emptive scheduling.

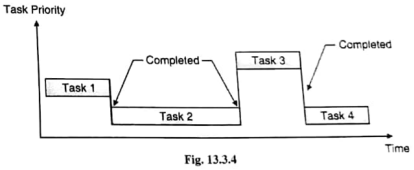


Fig. 13.3.4

- Let us take an example to understand this. The table given below shows the process name, their arrival time and execution time.

Process	Arrival Time	Execution Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Fig. 13.3.5 : Non-pre-emptive SJF scheduling

- The figure SJF non-preemptive scheduling of this example is shown in the Fig 13.3.4. Since process P1 came first it started first and the process p2 arrived at 2.0 sec, it was not given the CPU time although it is of higher priority based on its less execution time. Similar is the case with process P3. But, at time 7 sec, when P1 completes execution, P2, P3 and P4 are all in ready to run state, hence they are executed based on their priorities i.e. execution time.
- In this case, average waiting time = (0 + 6 + 3 + 7)/4 = 4 The waiting time for task P1 is 0 for P2 it is 6 for P3 it is 3 and for P4 it is 7.

13.3.3.6 Pre-emptive Scheduling

Q. 13.3.9 Explain with examples pre-emptive Scheduling. (Ref. Sec. 13.3.3.6) **(5 Marks)**

- This scheduling mechanism can be implemented in any of the previously seen scheduling mechanisms that have the concept of priority.
- As the name says, in this case if a task (say task 1)is in running state and another task (say task 2) with higher priority enters into the ready to run state, the earlier task (i.e. task 1) has to release the CPU and the later task (i.e. task 2) will get the execution.
- Thus the higher priority task will get the CPU as soon as it enters into the ready to run state, and this higher priority task will enter into the running state. The Fig. 13.3.6 shows an example of pre-emptive scheduling.

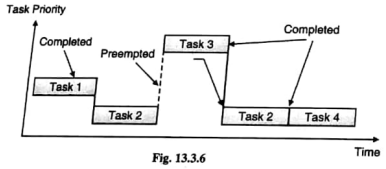


Fig. 13.3.6

- Let us take an example to understand this. The table given below shows the process name, their arrival time and execution time.

Process	Arrival Time	Execution Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

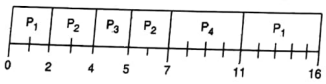


Fig. 13.3.7 : Pre-emptive SJF scheduling

- The figure SJF preemptive scheduling of this example is shown in the Fig. 13.3.7. Since process P1 came first, it started first and the process p2 arrived at 2.0 sec, it was given the CPU time since it is of higher priority based on its less execution time. Similar is the case with process P3, when it arrived at 4 seconds. Hence the lowest priority process P1 completes its execution at the end.
- In this case, average waiting time = (9 + 1 + 0 +2)/4 = 3

Syllabus Topic : Performance Matrix in Scheduling Models

13.3.3.7 Performance Matrix in Scheduling Models

Q. 13.3.10 Explain performance matrix in scheduling models. (Ref. Sec. 13.3.3.7) **(5 Marks)**

As we already discussed, multitasking involves task switching where each task compete for CPU time for its execution. Selecting the task which is to be executed at given point of time is known a task (process) scheduling. Task scheduling provides the decision power to multitasking. Algorithm are developed using scheduling policies.

☞ **Criterion for selecting scheduling algorithm are as follows :**

1. **CPU utilization :** The scheduling algorithm should always utilize CPU efficiently.
2. **Throughput :** Throughput indicates the number of tasks executed per unit of time. The throughput for a good scheduler should be high.
3. **Turnaround time :** It is amount of time taken by a task for its completion. It includes time required to perform various activity like the time spent by task to access memory, time spent in ready queue, time spent for completing input / output operation, and time spent in execution.
4. **Waiting time :** It is amount of time spent by task in the 'Ready' queue to get the CPU time for execution.
5. **Response time :** It is the time elapsed between creation of a task and the first response.

Syllabus Topic : Interrupt Management in RTOS environment

13.4 Interrupt Management in RTOS environment

➔ **(MU - Dec. 15)**

Q. 13.4.1 List the Kernel objects and explain functions of each of the objects. (Ref. Secs. 13.2, 13.3, 13.4, 13.5, 13.7 and 13.8) **(Dec. 15, 10 Marks)**

Interrupt is a very important term in embedded system and RTOS. Some important definitions given below will clear a lot of concepts related to interrupt.

1. **Interrupt :** It is a mechanism by which an I/O device (Hardware interrupt) or an instruction (Software interrupt) can suspend the normal execution of the processor and get itself serviced.
2. **Interrupt service routine (ISR) :** A small program or a routine that when executed services the corresponding interrupting source is called as an ISR.
3. **Interrupt latency :** It is defined as the time between the occurrence of an interrupt and the beginning of its servicing.
4. **Interrupt response time :** It is defined as the time between the occurrence of an interrupt and the completion of the interrupt servicing i.e. the completion of the execution of ISR. In case of a system analysis the worst case response time must be considered.

5. **Interrupt recovery time :** The time between the completion of the ISR execution and the returning of the earlier task execution is termed as the interrupt recovery time. This is the time required for executing the return instruction of the ISR and context recovery of the interrupted task from the stack. In fact, this time will also include the kernel time to check if any higher priority task is in ready to run state, for pre-emptive scheduling. Hence for a pre- emptive scheduling, the interrupt recovery time is a summation of time to check if a higher priority task is in ready to run state + time to restore the context of the highest priority task + time to execute the return instruction of the ISR.

Fig. 13.4.1 shows the different times w.r.t interrupts.

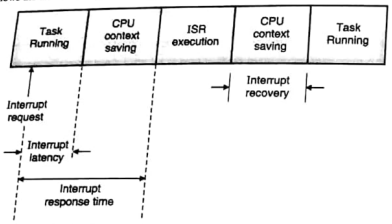


Fig. 13.4.1 : Interrupt latency, response and recovery time

13.5 Semaphores

→ **(MU - Dec. 14, May 15, Dec. 15, Dec. 16, Dec. 17)**

Q. 13.5.1 What is semaphore ? Explain the use of semaphore with respect to embedded operating systems. (Ref. Sec. 13.5) **Dec. 14, Dec. 16, Dec. 17, 10 Marks**

Q. 13.5.2 Explain how semaphores can be used to solve shared data problem. (Ref. Sec. 13.5) **May 15, 8 Marks**

Q. 13.5.3 List the Kernel objects and explain functions of each of the objects. (Ref. Secs. 13.2, 13.3, 13.4, 13.5, 13.7 and 13.8) **Dec. 15, 10 Marks**

- A semaphore is a mechanism for controlling concurrent access to a shared resource.
- Instead of using operations to read and write a shared variable, a semaphore encapsulates the necessary shared data, and allows access only by a restricted set of operations.
- Because a semaphore has the power to suspend and wake processes, semaphores and similar higher-level constructs require the co-operation of the operating system and/or programming language run-time system.

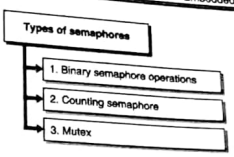


Fig. 13.5.1 : Types of semaphores

- There are two operations on a semaphore S. Worker processes can wait() or signal() a semaphore. For historical reasons, the wait and signal operations are sometimes abbreviated as P and V respectively. This will be discussed in the next subsection.
- Note that with semaphores, worker processes do not execute a potentially wasteful busy-waiting loop.

There are various types of semaphores: Binary semaphore, Counting semaphore, MUTEX.

13.5.1 Binary Semaphore Operations

→ **(MU - May 17)**

Q. 13.5.4 Explain semaphores and mutex in RTOS. (Ref. Secs. 13.5.1 and 13.5.3) **May 17, 5 Marks**

The operation of a binary semaphore can be understood by the Fig. 13.5.2.

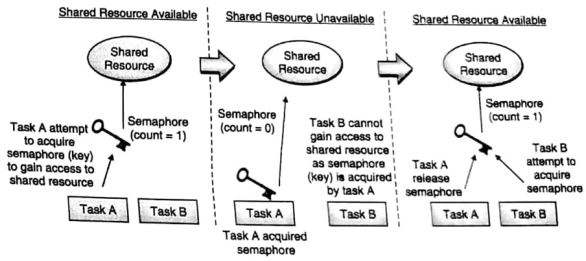


Fig. 13.5.2 : Binary semaphore

- As shown in the Fig. 13.5.2, initially the key for the shared resource is available. i.e. count = '1'. Task 'A' attempts to acquire this and gets the same. Now task 'B' needs the same shared resource, but finds the key is not available i.e. count = '0' and hence doesn't get the resource. Once task 'A' completes its requirement of the shared resource, it releases the shared resource by releasing the key and then task 'B' gets the same.

- To perform the above process, we need two operations described below :
  1. **Wait()** : a process performs a wait operation to tell the semaphore that it wants exclusive access to the shared resource. If the semaphore is empty, then the semaphore enters the full state and allows the process to continue its execution immediately. If the semaphore is full, then the semaphore suspends the process (and remembers that the process is suspended).
  2. **Signal()** : a process performs a signal operation to inform the semaphore that it is finished using the shared resource. If there are processes suspended on the semaphore, the semaphore wakes up one of them. If there are no processes suspended on the semaphore, the semaphore goes into the empty state.
- The various functions required to work with semaphore are :
  1. Create a semaphore
  2. Delete a semaphore
  3. Acquire a semaphore
  4. Release a semaphore
  5. Query a semaphore

13.5.2 Counting Semaphore

Q. 13.5.5 Write a short note on "Counting Semaphore". (Ref. Sec. 13.5.2) **(5 Marks)**

- When multiple copies of a resource are available, multiple task can access the same simultaneously. In such cases we use a counting semaphore. The count value depends on the number of resources available at any given time.
- Let us take some examples to understand this concept :
  1. **Car Parking lot :** We have multiple car parking possible in a mall. The counting semaphore can be used over here, that keeps a track of number of parking lots free. Every time a car enters the lot the count decrements and the count increments every time a car leaves the parking lot. If the count is zero, new car doesn't get the access to the parking lot.
  2. **Pool of printers :** Let us say that we have multiple printers in a system. The counting semaphore can be used over here, that keeps a track of number of printers free. Every time a task acquires the printer the count decrements and the count increments every time a task releases the printer. If the count is zero, new task doesn't get the access to any printer.
- Fig 13.5.3 shows the behaviour of the counting semaphore for a shared resource.
- As shown in the Fig. 13.5.3, there is a count maintained, which is two in this case and hence there are two keys available. Thus a maximum of two tasks can access the shared resource at any given time.

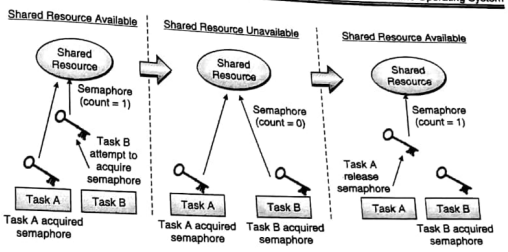


Fig. 13.5.3 : Counting semaphore

13.5.3 Mutex

→ **(MU - May 17)**

Q. 13.5.6 Explain semaphores and mutex in RTOS. (Ref. Secs. 13.5.1 and 13.5.3) **May 17, 5 Marks**

- Mutex is a short form for Mutual exclusion. It is used for resource and task synchronization. It is a special binary semaphore that can be in a locked or unlocked state.
- A task acquires or locks a mutex whenever required, uses the resource, and finally unlocks or releases the same. In this case the owner of the semaphore will be a particular task and this task can acquire the same resource for multiple times. Another task will not be able to acquire the resource if it is not the owner of the mutex.
- Fig. 13.5.4 shows the example of a mutex.

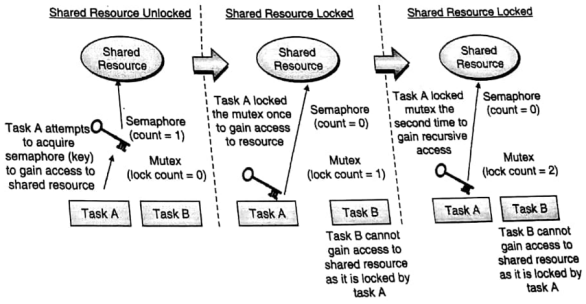


Fig. 13.5.4 : Mutex

– In the Fig. 13.5.4, the task 'A' has multiple lock counts and the semaphore once acquired by task 'A' cannot be acquired by any other task.
– If the owner acquires or locks the mutex for 'x' times then the task has to also release it for 'x' times. A task that owns the mutex cannot be deleted unless and until it releases all the mutexes.
– In case of mutex we can have a problem of priority inversion that can be manage by some special protocols discussed later in the next section.
– Mutex can be achieved by the following mechanisms :

1. Disabling the interrupts
2. Disabling the scheduler
3. Test-and-set operations
4. Using semaphore

➜ **1. Disabling the interrupts :**

– In this case when the interrupt s are disabled, tasks that are in waiting state will not be able to reach to the ready to run state.
– Hence the critical section of the task in running state will be completed and then the interrupts will be enabled. Thus mutual exclusion will be achieved.

➜ **2. Disabling the scheduler :**

– In this case the scheduler is disabled, so it will not be able to schedule any other task to running state. Thus the critical part of the code will be executed with mutual exclusion and then the scheduler will be enabled again.
– But this can cause a major problem if some problem occurs with the task that disables the scheduler. In such case the scheduler will never be enabled and hence the system may behave abruptly.

➜ **3. Test-and-set operations :**

– The test and set operations can be implemented by testing a particular global variable to find whether the shared resource is available or not.
– For example there may be a global variable which will be initially '0', and whenever uses a resource the corresponding global variable will be set to '1' by that task.
– Thus if another task tests for the availability of that resource by checking the global variable, it will get the information that the task is used by some other task and hence is not available. The procedure for accessing the shared resource can be as given below:

(i) Disable the interrupts
(ii) Set the global variable corresponding to the resource to be acquired.
(iii) Use the shared resource
(iv) Reset or clear the global variable corresponding to the resource used.
(v) Enable the interrupts.

➜ **4. Using semaphore :**

– Semaphores can also be used as discussed in the previous sub-sections.
– The function calls required to work with mutex are :

1. Create a mutex	2. Delete a mutex
3. Acquire a mutex	4. Release a mutex
5. Query a mutex	6. Wait on a mutex

13.6 Priority Inversion Problems

➜ **(MU - Dec. 14, May 16)**

Q. 13.6.1 Explain priority inversion problem in embedded system. How does it is resolved ? (Ref. Sec. 13.6) **Dec. 14, May 16. 8 Marks**

– If there is a shared resource and the resource is acquired by a lower priority task, then the higher priority task may have to wait for the same, and in case if a medium priority task is being executed, then this is called as priority inversion problem i.e. the medium priority task is being executed and the higher priority task has to wait.
– Fig. 13.6.1 shows a case of priority inversion

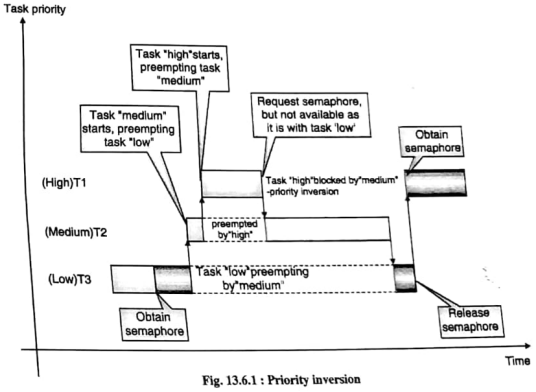


Fig. 13.6.1 : Priority inversion

– The priority inversion problem can be solved in two ways:

1. **Priority Inheritance :** In this case the lower priority task is given the priority of the highest task requiring a particular resource and hence not allowing a medium level priority task to preempt. Fig. 13.6.2 shows the concept

of priority inheritance. The figure explains the stepwise operations that will take place for priority inheritance. The procedure for the same is as given below:

(i) Task 3 is put into running state

(ii) Task 3 acquires the mutex

(iii) Task 3 starts executing the critical section that requires the mutex

(iv) Task 1 preempts task 3

(v) Task 1 starts its execution

(vi) Task 1 tries to acquire the mutex and hence task 3 is to be executed with the priority same as that of Task 1, so that the medium priority task 2 cannot preempt the task 3.

(vii) Task 3 completes the critical task

(viii) Task 3 releases the mutex it gains back its original priority

(ix) Task 1 completes its critical section and releases the mutex.

(x) Task 1 completes its operation and next priority task i.e. task 2 i.e. medium priority task gets into running state

(xi) Task 2 completes its execution

(xii) Task 3 now gets into running state and gets the CPU time.

2. **Priority ceiling :** It is same as that of the priority inheritance, only that the lower priority task with semaphore is given the highest priority in the system instead of the priority of the higher task requiring semaphore as in priority inheritance.

– Thus the priority inversion problem can be solved by one of the two methods discussed above.

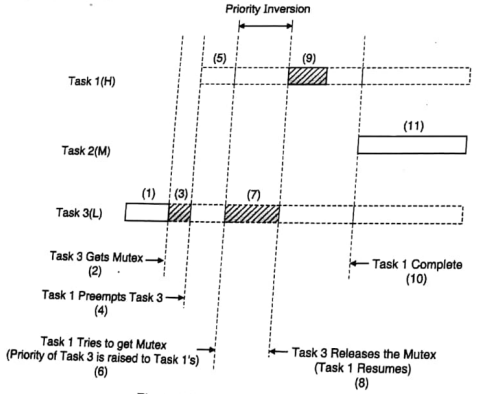


Fig. 13.6.2 : Priority inheritance

13.6.1 Deadlock

– Deadlock occurs when two tasks are waiting for a resource held by each other

– To avoid the deadlock, we can set a time limit for the resource to be held by a particular task. Hence the semaphore and the resource will be automatically released after the predefined time and once the resource is available the task in waiting state will come into the ready to run state. Thus solving the problem of deadlock.

Syllabus Topic : Inter-process Communication

13.7 Inter Task or Inter Process Communication

→ **(MU - Dec. 15, May 17)**

Q. 13.7.1 List the Kernel objects and explain functions of each of the objects. (Ref. Secs. 13.2, 13.3, 13.4, 13.5, 13.7 and 13.8) **Dec. 15, 10 Marks**

Q. 13.7.2 Explain the various methods to implement interprocess communication. (Ref. Sec. 13.7) **May 17, 10 Marks**

Q. 13.7.3 Explain any three Inter-Task or Inter-process communication methods. (Ref. Sec. 13.7) **(5 Marks)**

– There are many cases wherein there is a requirement of communication between different tasks or processes. There are various methods to implement the same, like:

1. Mailboxes	2. Message Queues
3. Event registers	4. Pipes
5. Signals etc	

– We will discuss these in the subsequent subsections.

13.7.1 Mailboxes

– Mailbox is similar to the postal mailbox. The mails of a particular task can be checked only by that task.

– Each task has its own mailbox. Other tasks, processes or an ISR can post messages for the task in its corresponding mailbox.

– Fig. 13.7.1 shows the mailbox implementation.

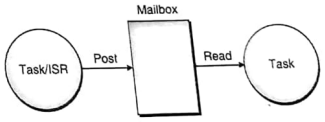


Fig. 13.7.1 : Mailbox implementation

– The various functions required to implement the mailbox are as listed below:

1. Create a mailbox	2. Delete a mailbox
3. Post a message in the mailbox	4. Read a message from the mailbox
5. Query a mailbox	

13.7.2 Message Queues

– Message queue is as good as an array of mailboxes i.e. the mailbox can store only one message at a given time, while the message queue can store many message simultaneously.

– The application of a message queue can be like the taking of input from keyboard and displaying the output or taking input from sensors and transmitting the data over a network, etc. Thus it is said that the message queue are mainly used for producer consumer applications i.e. a task produces data and another consumes it.

– Fig. 13.7.2 shows the implementation of a message queue.

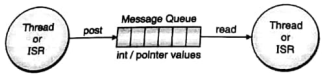


Fig. 13.7.2 : Message queue implementation

– The various functions required for implementation of message queue are :

1. Create a message queue	2. Delete a message queue
3. Post a message in message queue	4. Post a message in the beginning of the queue
5. Broadcast a message	6. Read a message from message queue
7. Query a queue to give its info	8. Show queue waiting information
9. Flush the queue	

– Message queue can be implemented as unidirectional in case of producer consumer application or it may also be two directional for an application if required. It can also be used to broadcast a message to all the tasks.

13.7.3 Event Registers

➔ **(MU - May 15)**

Q. 13.7.4 What is Event register ? Explain the use of Event function with respect to embedded operating systems. (Ref. Sec. 13.7.3) **May 15, 10 Marks**

– Event register can be used to keep a track of the events that have occurred. Each bit in the task register can be kept for a particular event. Hence the task can keep a track of which events have occurred and accordingly the different operations to be performed.

– The different functions required to implement event register are as given below :

1. Create an event register	2. Delete an event register
3. Set an event flag	4. Clear an event flag
5. Query an event flag.	

– These functions will be required to implement the event register in the OS. These functions are described below :

➔ **1. Create Event register**

A memory location can be used for this purpose. Most of the microcontrollers have bit accessible memory locations. These are very useful for implementing i.e. being used as event registers. The single bit location can be used to indicate whether the event has occured or not. If the event occurs, the bit can be made '1' else it will remain '0'

➔ **2. Delete an event register**

This function will be used to delete the event register created. The register once deleted can be used for other purposes. It is very important in an embedded system to save memory or have optimum memory usage. Hence as soon as the use of a particular event register is over, this function should be called to delete the register created.

➔ **3. Set an Event flag**

Whenever an event occurs, this function must be called, which will set the event register flag indicating that the event has occurred.

➔ **4. Clear an event flag**

Whenever the event flag is set to '1', the corresponding action will have to be taken. Once the corresponding action is taken, it is necessary that the event flag is cleared, so that the flag can again be set when the event occurs for the next time. Hence this function must be called to clear the event flag as soon as the action corresponding to the event starts.

➔ **5. Query an Event flag**

This function is used for checking the state of the event flag. If the flag is set to '1' or '0' can be indicated to the caller of the function by this function.

13.7.4 Pipes

– Pipes are also much like queues. Pipes are used to send messages from one task to another. The operation is demonstrated in Fig. 13.7.3(a).

– Thus pipes are used for inter-task communication i.e. from one task to another or from an ISR to a task.

– A task can write into the pipe of another task and if required for other way communication there will be another pipe required. The other direction pipe may also be used to send acknowledgements to the sender task. This is shown in the Fig. 13.7.3(b).

(a) Simple pipe implementation **(b) Bidirectional pipe implementation**

Fig 13.7.3 : Pipes implementation

– The various functions required to implement a pipe are as listed below:

1. Create a pipe	2. Delete a pipe
3. Write in the pipe	4. Read from the pipe
5. Close the pipe	

13.7.5 Signals

- Signals are simple indications from one task to another task to indicate the occurrence of an event. These indications may be used to trigger some task, or indicate the availability of a resource etc.
- The various functions required to implement the Signals are listed below:

1. Create or Install a signal handler	2. Delete or Remove the installed signal handler
3. Send a signal from one task to another task	4. Ignore the received signal
5. Block the signal to be delivered	6. Unblock an already blocked signal

13.8 Timers

➔ (MU - Dec. 15)

Q. 13.8.1 List the Kernel objects and explain functions of each of the objects. (Ref. Secs. 13.2, 13.3, 13.4, 13.5, 13.7 and 13.8) **Dec. 15, 10 Marks**

- These are special tools required in embedded systems to find out the time elapsed for a particular event. There are various evens for which the kernel has to keep a track of time elapsed like :
  1. To execute periodic tasks i.e. after a fixed time the task is to be executed repeatedly. In such case the timer will keep the track of the time.
  2. A task may be waiting in the waiting state for a long time. After a specific time of the task being in waiting state a proper action has to be taken. The timer will keep the track of this specific time elapsed or not.
  3. A task may also be in waiting state for an event to occur, and if the event doesn't occur for a long time a proper action has to be take,. The timer will keep the track of this specific time elapsed or not.
  4. As discussed earlier to avoid deadlock the semaphore is to be released after a predefined time.
- The various functions required to manage the timer operations are as given below :

1. Get current time	2. Set the time to a particular value
3. Set for a time delay	4. Reset the timer.

Syllabus Topic : Memory Management

13.9 Memory Management

➔ (MU - May 15)

Q. 13.9.1 How RTOS manages the memory ? Give the memory management strategy of RTOS in embedded system. (Ref. Sec. 13.9) **May 15, 10 Marks**

- As discussed earlier among the different tasks of the kernel, memory management is also an important task to be performed.

- Memory management involves allocating memory blocks to various tasks such that these memory blocks should not be overlapping each other and still if required there should be mechanisms to help inter-task communication as discussed in section 13.7
- The various functions required for the memory management are as listed below:
  1. Create a memory block
  2. Free the created memory block
  3. Get the data from the memory
  4. Store the data in a memory block
  5. Query the memory block
- These functions that are required for memory management are discussed below :

➔ **1. Create a memory block**

- This function as the name says will create a memory block.
- Whenever a new task is initiated, memory is required for storing the data for that task. This memory block required for the task is created by the use of this function

➔ **2. Free the created memory block**

- Once the task is completed i.e. it goes to dead state, the memory block must be freed.
- The memory utilization in an embedded system is critical, hence it is very important that the memory must be made free as soon as possible so that the other tasks can use it.

➔ **3. Get the data from the memory block**

- This function as the name says is used to get the data from the memory block.
- A proper offset address will be required to be passed to the function so that it will return the data from that location

➔ **4. Store the data in the memory block**

- This function as says will store the data into the memory location specified.
- The parameters required to pass to this function will be the data and the address of the location where the data is to be stored.

➔ **5. Query the memory block**

- This function is used to check the status of the memory block.
- It will give information like how much memory locations are available, the task for which the block is allotted (if any), whether the memory block is free or is in use, etc.

Syllabus Topic : File Systems

13.9.1 Introduction to File System

Q. 13.9.2 Explain file systems. (Ref. Sec. 13.9.1) **(4 Marks)**

- File system is essential in any computer systems for organizing data. In Linux, data is organized in files and directories in a hierarchical structure. Top of the structure is root.
- Linux has a single global hierarchy of files and directories. This hierarchy can be composed of many file systems.
- File systems are mounted on specific locations in directory hierarchy.
- This specific location is called the mount point. When mounted properly, the directory content reflects the content of storage device.
- Embedded system developers have to make their choice for choosing the file system for their device.
- This requires the study available file system for embedded Linux and their tradeoffs.

13.9.2 Linux File System Concepts

Q. 13.9.3 Explain Linux File System. (Ref. Sec. 13.9.2) **(4 Marks)**

- Linux file systems rely on logical division of the physical device. This logical division is called partitions. Linux file system exists in these partitions.
- In embedded system, these partitions are created on flash drive.
- Data is organized in these partitions according to specification of partition types.
- One partition can be logically viewed as a single entire device where the specific file system exists.
- Fig. 13.9.1 shows an abstract view the partition in a flash drive its file system.

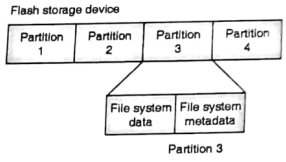


Fig. 13.9.1 : Flash device partitions and file system

- Installed file system uses the partition to store actual data and partitions related information as well.
- Partitions can be managed and manipulated by a Linux utility called **fdisk.**
- This utility is aware of several partition types like Linux, FAT32, Linux swap.

Syllabus Topic : I/O Systems

13.10 I/O System or Device Drivers Basics

13.10.1 Introduction to Device Drivers

Q. 13.10.1 Explain device drivers and its types. (Ref. Sec. 13.10.1) **(5 Marks)**

- Device drivers are programs designed and implemented to control particular type of device connected to a computer system.
- Device drivers provide a software interface for operating system and other programs to deal with complex hardware operations.
- **For example :** A user program will just ***create*** a file and issue a ***write*** command. It does not specify the hardware sectors and its size where the data needs to be stored. Detailed operations are taken care by device driver.
- Devices drivers are basically kernel components which can be loaded as module. These modules take care of data transfer between OS and devices.
- Device drivers operations are characterized by the type of devices connected to the system. Devices have following types :

1. Block device
2. Character device
3. Network device

➜ **1. Block Device**

- Block devices can be addressed in chunks of device-specified data called blocks (randomly accessible). Examples are hard drives, Blu-ray disks, flash devices.
- Each devices is accessed by a special file called a ***block device node*** and mounted as filesystem.

➜ **2. Character Device**

- Character devices are operated as continuous flow of bytes. Generally data is not addressable and can be accessed as stream of bytes.
- Example of such devices is: Keyboard, printers etc. These types of devices are accessed by a special file called ***character device node.***

➜ **3. Network Device**

- Like character and block devices, network devices are also represented by a special device file in Linux. These files are created by the Linux when it finds and initialize a new network adapter or network controller.
- These devices are network device operations are controlled by networking subsystem (socket implementation).

13.10.2 An Architectural Overview of Device Drivers

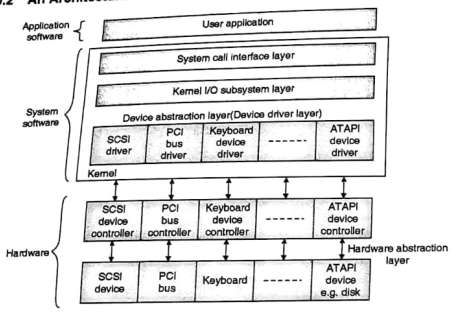


Fig. 13.10.1 : Device driver and Kernel IO structure

- Fig. 13.10.1 shows an abstract overview of device driver and kernel IO subsystem.
- Device driver modules are part of kernel space.
- The set of interfaces and unique structuring techniques enables the operating system to treat all the connected devices in standard manner.
- The Fig. 13.10.2 shows a set of layers between user program and the actual hardware.
- Making a simple print statement in your program causes a communication starting from system call to actual device via device driver and device controller.
- The flow the communication includes hardware and software interfaces.

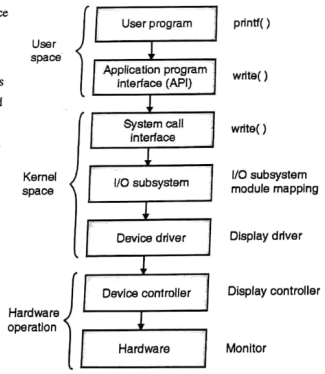


Fig. 13.10.2 : Communication flow for printf statement

User program -> API -> System call -> IO Subsystem -> Device Driver ->Device Controller -> Hardware

- User application interacts with kernel using system call interface. Most application programmers use some convenient API set for their programming.

- **For example,** C programmers uses standard C library for many operations including the simplest printf() statement. This printf function is implemented in user space library and makes the actual system call.
- System call like write() is implemented within kernel. It invokes the IO subsystem of the kernel that is able to map actual driver needed to fulfil the IO operation.
- Kernel modules like IO subsystem interact with device drivers for device operation like read, write etc.
- Kernel has a device abstraction layer that represents each device as block device or character device. These details are encapsulated within the specific device drivers.
- Finally device driver communicates with the device controller to complete the required operation. Controller and device driver use interrupts for communication and synchronizing their operations.
- A typical interaction of device driver is shown Fig. 13.10.3. A driver operation can be invoked or accessed in the following scenarios :

1. **Boot time :** Once kernel loaded, kernels boot system (or the initialization system and scripts) invokes the device driver for first time to check device availability. It also initializes the hardware by invoking initialization functions on device drivers.
2. **Application requesting IO operations :** User application requests IO operation by making appropriate system calls. System calls like read() and write() are translated into driver specific operations.
3. **Hardware interrupts :** Interrupts generated by hardware is received by kernel and appropriate interrupt specific driver code is loaded.
4. **Bus reset operation :** In some scenarios, drivers are called by kernel to reinitialize its state. This happens when the connecting bus is reset. PCI bus driver is discussed in Section 13.10.6.

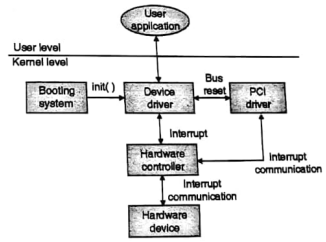


Fig. 13.10.3 : Device Driver Interaction

Syllabus Topic : Selecting a Real Time Operating System and RTOS Comparative Study

13.11 Off-the-shelf Operating Systems : Selecting a Real Time Operating System and RTOS Comparative Study

- There are many operating systems available in the market. Instead of developing your own operating system with all the features discussed in the earlier sections of this chapter, it is better to take an operating system off-the shelf and use it.
- The selection of the operating system should be such that it is best suited for your application.
- There are various types of operating systems available off-the-shelf. Some of these are :

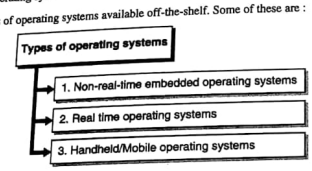


Fig. 13.11.1 : Types of operating system

➜ **1. Non-real-time embedded operating systems**

- These are basic kernel operating systems.
- These OS can be used for embedded applications where deadlines are not important.

➜ **2. Real time operating systems**

- These operating systems provide the special functions required to implement the deadline based scheduling.
- These are used in embedded applications where deadlines are very important.

➜ **3. Handheld/Mobile Operating systems**

- These are special operating systems that are used for mobile or handheld devices like smart phones.
- We will discuss each of these in the subsequent sections.
- There are certain common things in all the OS, and they are listed below :

1. **Integrated Development Environment (IDE) :** All the OS developers provide some tools for programming i.e. editor, compiler, debugger etc.
2. **POSIX compatible :** POSIX stands for "Portable Operating System Interface". It is a set of standards defined by the IEEE and issued by ANSI and ISO. The main goal of POSIX is to ease the cross-platform software development by establishing a standard for operating system vendors to follow. Thus the programmer will have to write a program and it will work on all POSIX-compliant systems. There are many different POSIX releases, but the most important ones are POSIX.1 and POSIX.2, which define system calls and command-line interface,

respectively. POSIX.4 defines the standards for real-time OS. POSIX.13 defines the real time application environment profiles. Each profile lists the services that are necessary for a particular application environment.

3. **Support for TCP/IP :** This is a particular protocol required for internet connection.
4. **Device Drivers :** A number of device drivers are included in each OS

- There are certain uncommon things or differences in various OS. These are listed below :

1. **Task Priority mechanism :** This is the major thing in which most of the OS differ from each other. The method used to assign priorities and to handle them is different in different OS.
2. **Priority Inheritance :** The priority inheritance may or may not be given to the programmers for various OS
3. **Scheduling Algorithm :** The method of deciding of which task will utilise the CPU at any given time, varies from OS to OS.
4. **Maximum number of tasks :** The maximum number of tasks that the OS can handle also varies from OS to OS.
5. **Support for POSIX :** Certain OS may either not support the POSIX standards or may partially support the same or may support different versions of the POSIX.
6. **License :** The terms and conditions for the License of the different OS are also different.

13.12 Embedded Operating Systems (Non-real-time)

- As discussed earlier, these are the operating systems, that are such that they support all the features of the OS. The only thing is that the deadlines to be met is not very strict in these cases.
- Some of the examples of these type of Embedded OS are as listed below :
  1. Windows XP Embedded
  2. Embedded Linux
  3. Embedded NT
- Microsoft Windows NT version 4.0 Embedded was a reintroduction of a Microsoft desktop operating system for the embedded market.
- Windows XP Embedded builds on the momentum of Windows NT Embedded with improved features and better performance.
- Looking in from the outside, the Microsoft embedded strategy is to offer two types of embedded operating systems: Microsoft® Windows® CE for small battery powered devices and Windows XP Embedded for high-end embedded applications.
- Each version of Windows CE has added more capability, allowing it to move from only a handheld operating system to a real-time embedded operating system (OS) used in multiple applications.
- Microsoft Windows NT® Embedded was introduced shortly after Windows CE 2.1 as a feature-rich embedded operating system.
- The Table 13.12.1 shows the comparison of the Embedded NT and Windows XP Embedded w.r.t. the core features supported.

Table 13.12.1 : Windows NT embedded vs windows XP embedded

Feature	Windows NT Embedded 4.0	Windows XP Embedded
Device Driver Rollback		X
Device Driver Signing of third-party drivers		X
Enhanced Registry performance with large amount of key caching		X
Parallel processing of drivers and registry data at boot		X
Dynamic file layout modifications designed to optimize boot load		X
Optimized disk defragmentation to support faster boot time		X
Memory pooling allowing for prioritization of memory tasks, creating shorter boot time		X
Simple Boot Flag Support		X
I/O Throttling		X

- Many consumer electronics equipments like TV, automation, instruments, routers, etc. use the Linux kernel based operating system.
- The main features of embedded Linux are:
  1. Open source, hence no royalty
  2. POSIX support
  3. Many expert people available with knowledge of Linux are easily available.
  4. Also many software support resources w.r.t. the Embedded Linux is available.

13.13 Real Time Operating System (RTOS)

- As discussed earlier the main difference of the non-real-time embedded OS and Real-time Embedded OS is that the RTOS has a deadline for each task to be followed.
- Some of the RTOS available off-the shelf are as listed below :

1. **QNX Neutrino :** QNX Neutrino is a popular real-time operating system software systems limited. The QNX Neutrino RTOS is a full-featured and robust OS that scales down to meet the constrained resource requirements of real time embedded systems. Its true microkernel design and its modular architecture enable customers to create highly optimized and reliable systems with low total cost of ownership. It offers the embedded industry's only field-proven, clean strategy for migrating from single-core to multi-core processing.
2. **VxWorks :** As the first RTOS with 32-bit and 64-bit processing, multi-core and multi-OS support, and diverse connectivity options, VxWorks provides you with the functionality and support you require to stay competitive. And as your platform plans evolve to take advantage of next-generation processor capability, we continue to stay ahead of the technology curve, continually expanding VxWorks' proficiency to extract maximum performance

from the new multi-core landscape. VxWorks' unique combination of high speed and scalability with trusted safety and security capabilities will keep your platforms running at the leading edge of the latest embedded technology.

3. **MicroC/OS-II :** MicroC/OS is a scalable OS i.e. only those functions of the OS that are used in the program written are copied into the system. Thus those functions that are required must be specified in the configuration file. It uses a pre-emptive scheduler. It has various service functions that are required for managing the tasks in OS like, creating a task, deleting a task etc.
4. **RTLinux :** RTLinux has all the functions such that the time to execute a function is directly related to the size of input to the function. The RTLinux is also a pre-emptive scheduler and also supports spin-lock. It has many device drivers already available with it. It uses POSIX threads and processes.

13.14 Handheld Operating Systems

- As discussed earlier, these are OS that are designed especially for the handheld devices like smart phones, etc.
- Some of the Handheld OS available off-the-shelf are as listed below :

1. **Palm OS :** It is one of the most popular handheld OS. It is used by various companies manufacturing handheld devices like Sony, Handspring, etc. It is mainly designed for the touchscreen -based graphical user interface devices. The key features of the Palm OS include simple, single task environment, handwriting recognition tool, sound playback capability, expansion memory support, etc.
2. **Symbian OS :** It has a very wide market. It is used in many mobile phones. The main users include Ericsson, Kenwood, Motorola, Nokia, Panasonic, Sanyo, Siemens, Sony etc. This OS has a support for IR communication called as IrDA as well as for bluetooth. The applications can be developed in various languages like C++, Java, etc.
3. **Windows CE :** Pocket PC hardware works using Microsoft Windows CE has the OS. There are various x86 based pocket computers that use Windows CE. Some of the companies that have their handheld computers working on Windows CE are Casio, Philips, Sharp, NP, NEC, etc. The tools like VC++, VB++ can be used to design the applications for the Windows CE based system.
4. **Windows CE.NET :** It is a successor of Windows CE and is a real time OS. It has gain significance in devices like CD players, digital cameras, DVD players, etc.

Syllabus Topic : POSIX Standards

13.15 POSIX Standards

Q. 13.15.1 Explain POSIX standards in details. (Ref. Sec. 13.15.1) **(5 Marks)**

POSIX stands for Portable Operating System Interface for Unix. It is an IEEE standard for the operating system to comply with. Those OS that comply with the basic POSIX are supposed to offer the following operations :

1. **Process Creation and Control :** It should be able to create a process and control the same. Thus if a request for the creation of a process is given, the same should be created. The control of the same i.e. execution, suspend and termination should be handled.

[illegible] should be supported.

2. **Signals :** The event or signal as a tool to have inter process [illegible] should be supported.
3. **Pipes :** Another tool for inter task (process) communication i.e. the pipe should also be implemented.
4. **Segmentation or other memory violations :** There should be proper care taken so as to [illegible] the [illegible] of a process/task accessing [illegible] or segments of other process/task.
5. **Illegal instructions :** Care must be taken to handle the illegal instructions.
6. **Bus Errors :** The errors on the bus during transfer of data should be properly handled.
7. **Timer :** The tick timer for synchronization of the various events should be implemented.
8. **Directory operations :** There should be a proper directory maintained for the files stored in the memory.
9. **Floating Point Exceptions :** In case of an error of a floating point operation, the same should be notified by and interrupt or a signal.
10. **Standard C library functions :** These functions should be supported.
11. **Process Triggers :** The triggers for the various operations on the process should also be implemented.
12. **I/O port interface & control :** The support for interfacing the I/O ports and the control for same should also be supported for some basic devices like LCD display, Matrix keyboard etc.

13.16 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : Basics of RTOS : Real-time Concepts, Hard Real Time and Soft Real Time**

Q. 1 Explain in brief Real Time Operating Systems. **(Refer section 13.1)** **(D-15, 4 Marks)**

Q. 2 Explain real time operating systems. **(Refer section 13.1)** **(D-16, 2.5 Marks)**

☞ **Syllabus Topic : Basic Architecture of an RTOS**

Q. 3 List the Kernel objects and explain functions of each of the objects.
(Refer sections 13.2, 13.3, 13.4, 13.5, 13.7 and 13.8) **(D-15, 10 Marks)**

Q. 4 Explain the architecture of the kernel of a Real Time Operating Systems.
(Refer section 13.2) **(5 Marks)**

Q. 5 List functions of Kernel. Also explain different types of kernel. **(Refer section 13.2)** **(D-17, 10 Marks)**

Q. 6 What do you mean by Task and Task state related to embedded operating systems and also discuss about Task Control Block (TCB) and its data.
(Refer sections 13.3, 13.3.1 and 13.3.1.1) **(D-14, 10 Marks)**

☞ **Syllabus Topic : Scheduling Systems**

Q. 7 List task scheduling algorithms. Explain operation of each of the algorithms.
(Refer section 13.3.3) **(D-15, 10 Marks)**

Q. 8 Explain any 3 scheduling algorithms. **(Refer section 13.3.3)** **(5 Marks)**

Q. 9 Illustrate scheduling algorithms of tasks in real time systems. **(Refer section 13.3.3)** **(D-17, 10 Marks**

Q. 10 Explain with examples Non pre-emptive Scheduling. **(Refer section 13.3.3.5)**

Q. 11 Explain with examples pre-emptive Scheduling. **(Refer section 13.3.3.6)** **(5 Marks)**

☞ **Syllabus Topic : Performance Matrix in Scheduling Models**

Q. 12 Explain performance matrix in scheduling models. **(Refer Section 13.3.3.7)** **(5 Marks)**

☞ **Syllabus Topic : Interrupt Management in RTOS environment**

Q. 13 List the Kernel objects and explain functions of each of the objects.
(Refer sections 13.2, 13.3, 13.4, 13.5, 13.7 and 13.8) **(D-15, 10 Marks)**

Q. 14 What is semaphore ? Explain the use of semaphore with respect to embedded operating systems.
(Refer section 13.5) **(D-14, D-16, D-17, 10 Marks)**

Q. 15 Explain how semaphores can be used to solve shared data problem.
(Refer section 13.5) **(M-15, 8 Marks)**

Q. 16 Explain semaphores and mutex in RTOS. **(Refer sections 13.5.1 and 13.5.3)** **(M-17, 5 Marks)**

Q. 17 Write a short note on "Counting Semaphore". **(Refer section 13.5.2)** **(5 Marks)**

Q. 18 Explain priority inversion problem in embedded system. How does it is resolved ?
(Refer section 13.6) **(D-14, 8 Marks)**

☞ **Syllabus Topic : Inter-process Communication**

Q. 19 List the Kernel objects and explain functions of each of the objects.
(Refer sections 13.2, 13.3, 13.4, 13.5, 13.7 and 13.8) **(D-15, 10 Marks)**

Q. 20 Explain the various methods to implement interprocess communication.
(Refer section 13.7) **(M-17, 10 Marks)**

Q. 21 Explain any three Inter-Task or Inter-process communication methods.
(Refer section 13.7) **(5 Marks)**

Q. 22 What is Event register ? Explain the use of Event function with respect to embedded operating systems.
(Refer section 13.7.3) **(M-15, 10 Marks)**

☞ **Syllabus Topic : Memory Management**

Q. 23 How RTOS manages the memory ? Give the memory management strategy of RTOS in embedded system. **(Refer section 13.9)** **(M-15, 10 Marks)**

☞ **Syllabus Topic : File Systems**

Q. 24 Explain file systems. **(Refer Section 13.9.1)** **(4 Marks)**

Q. 25 Explain Linux File System. **(Refer Section 13.9.2)** **(4 Marks)**

☞ **Syllabus Topic : I/O Systems**

Q. 26 Explain device drivers and its types. **(Refer Section 13.10.1)** **(5 Marks)**

☞ **Syllabus Topic : POSIX Standards**

Q. 27 Explain POSIX standards in details. **(Refer Section 13.15.1)** **(5 Marks)**

Chapter Ends...

❑❑❑

CHAPTER 14

Introduction to Embedded Target Boards

Syllabus Topic : Introduction to Arduino, Raspberry Pi, ARM Cortex, Intel Galileo etc. , Open-source prototyping platform

14.1 Open-Source Prototyping Platform : Introduction to Arduino, Raspberry PI, ARM Cortex, Intel Galileo etc.

- *The main components required to make an embedded system are Hardware, RTOS and the application program. In the previous chapters we have seen all three of these : 8051 as hardware, RTOS as well as making application programs.*
- *Today, many ready-to-use hardware boards are available that make the life of a embedded system designer quite easy.*
- *He needs to directly port the RTOS into the desired board and start writing application program for the specific application. There are various boards available in the market, but we will discuss the top four in this section*

14.1.1 Raspberry PI

Q. 14.1.1 *Compare various models of Raspberry Pi.* **(Ref. Sec. 14.1.1)** **(5 Marks)**

- *The most popular boards available in the market is Raspberry Pi. It has many high end application uses. It is mainly used as a web server. It is compatible with Android and can also be used to write and test the applications written for smartphones.*

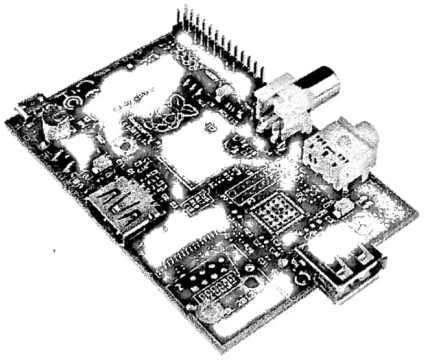


Fig. 14.1.1 : Raspberry Pi

- There are various models available and are also more expensive compared to the others, but the features provided are really worth its cost.
- It includes a Broadcom chip with 700MHz CPU based on the old ARM11 processor design. The PCB can also handle 1080p multimedia and comes with at least one HDMI port. A comparison chart of various models of Raspberry pi is given in Table 14.1.1 and Fig. 14.1.1 shows an image of a Raspberry Pi.

Table 14.1.1 : Comparison of Raspberry Pi models

Raspberry Pi	Model A+	Model B	Model B+	2, Model B
Quick summary	Cheapest, smallest single board computer.	The original Raspberry Pi.	More USB and GPIO than the B. Ideal choice for schools	Newest, most advanced Raspberry Pi.
Chip	Broadcom BCM2835			Broadcom BCM2836
Processor	ARMv6 single core			ARMv7 quad core
Processor Speed	700 MHz			900 MHz
Voltage and Power Draw	600mA @ 5V			650mA @ 5V
GPU	Dual Core VideoCore IV Multimedia Co-Processor			
Size	65x56mm	85x56mm		
Memory	256 MB SDRAM @ 400 MHz	512 MB SDRAM @ 400 MHz		1 GB SDRAM @ 400 MHz
Storage	Micro SD Card (not included)	SD Card (not included)	Micro SD Card (included)	Micro SD Card (not included)
GPIO	40	26	40	
USB 2.0	1	2	4	
Ethernet	None	10/100mb Ethernet RJ45 Jack		
Audio	Multi-Channel HD Audio over HDMI, Analog Stereo from 3.5mm Headphone Jack			

14.1.2 Arduino Board

Q. 14.1.2 *Compare various models of Arduino. (Ref. Sec. 14.1.2)* **(5 Marks)**

- *The Arduino board is the largest open-source platform in the world. Not only is it easy to understand, but its affordability makes it a viable choice to all skill levels.*
- *The size of the Arduino board paired with the great support that Arduino offers make it an easy pick for our top 4 open source boards list.*
- *Typically, an Arduino board consists of a Microcontroller ATmega328, operating voltage varies between models, a variety of Analog input pins, digital I/O pins, Flash memory, SRAM (varies/KB), EEPROM (varies/KB), Clock speed which changes depending on model. The various models are discussed in Table 14.1.2 and an image of Arduino is shown in Fig. 14.1.2*

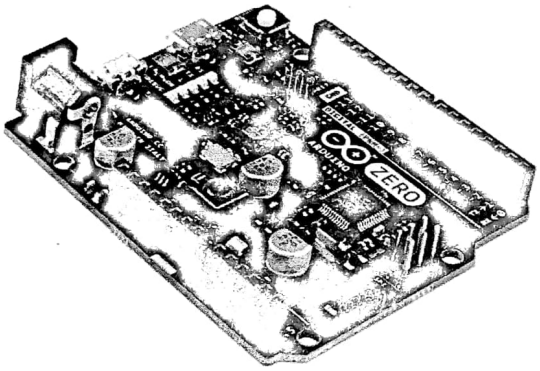


Fig. 14.1.2 : Arduino

Table 14.1.2 : Comparison of Arduino models

Arduino Board	Family	SRAM	MEMORY FLASH	EEPROM	Clock	UART	PWM	Digital	Analog	VCC	Vin Range	USB-Serial
Duemilanove (328)	ATmega328	2k	32K	1kB	16 MHz	1	6	14	6	5V	7-12V	Atmega16U2
Uno	ATmega328	2k	32k	1kB	16 MHz	1	6	14	6	5V	7-12V	Atmega16U2
Arduino Mega2560	ATmega2560	8k	256k	1kB	16 MHz	4	14	54	16	5V	7-18V	Atmega16U2
Arduino Mega ADK	ATmega2560	8k	256k	1kB	16 MHz	4	14	50	16	5V	7-18V	Atmega16U2
Arduino Ethernet	ATmega328	2k	32k	1kB	16 MHz	1	4	9	6	5V	5-16V	N/A
Arduino BT	ATmega328	2k	32k	1kB	16 MHz	1	6	14	6	5.5V	1.2V-5.5V	Bluegga WT11
Arduino Pro Mini 328 5V	ATmega328	2k	32k	1kB	16 MHz	1	6	14	6	5V	5-12V	N/A
Arduino Nano 3.0	ATmega328	2k	32k	1kB	16 MHz	1	6	14	8	5V	7-12V	FTDI FT232RL
Arduino Mini	ATmega328	2k	32k	1kB	16 MHz	1	6	14	8	5V	7V-9V	N/A
Arduino Pro 3..3V	ATmega328	2k	32k	1kB	8 MHz	1	6	14	6	3.3V	3.35-12V	N/A
Arduino Pro 5V	ATmega328P	2k	32k	1kB	16 MHz	1	6	14	6	5V	5-12V	N/A
Arduino Fio	ATmega328P	2k	32k	1kB	8 MHz	1	6	14	8	3.3V	3.35-12V	N/A
LilyPad Simple Board	ATmega168P	1k	16k	512kB	8 MHz	1	6	9	4	2.7-5.55V		N/A
LilyPad 328 Main Board	ATmega328P	2k	32k	1kB	8 MHz	1	6	14	6	2.7-5.5V		N/A

14.1.3 BeagleBoard

Q. 14.1.3 List the features of Beagleboard (Ref. Sec. 14.1.3) **(5 Marks)**

- BeagleBone is an open hardware platform. It is single board computer manufactured by Texas Instrument. The board combines the open source philosophy of hardware and software at a single platform. The hardware architecture is kept open and capable of running the Open source Linux. It is used widely used among college students, developers and hobbyist to develop and test embedded system projects.

- Beaglebone Black is one of the latest ARM processor board from Beagleboard family. This is small credit card sized board and a multi featured embedded computer system in itself. It hosts a low cost Sitara Cortex A8 ARM processor which is manufactured by Texas Instrument.
- It comes with default installation of Debian Embedded Linux Distribution. One can also install other operating systems like Ubuntu and Android. It has built in networking capability, HDMI support and SD card slot.
- Operating system can also be directly booted from SD card. In many ways, it is similar to previous BeagleBone boards but has few noticeable enhancements. Few differences between BeagleBone and BeagleBone Black are shown in Table 14.1.3.

Table 14.1.3 : BeagleBone Black Features

	Feature	
Processor	Sitara AM3358BZCZ100 1 GHz, 2000 MIPS	
Graphics engine	SGX530 3D, 20M Polygons/S	
SDRAM Memory	512MB DDR3L 800MHz	
Onboard Flash	4GB, 8bit Embedded MMC	
PMIC	TPS65217C PMIC regulator and one additional LDO	
Debug Support	Optional Onboard 20-pin CTIJTAG. Serial Header	
Power Source	miniUSB USB or DC Jack	5VDC External Via Expansion
PCB	3.4″ × 2.1″	6 layers
Indicators	1-Power, 2-Ehternet, 4-User Controllable LEDs	
HS USB2.0 Client Port	Access to USB0, Client mode via miniUSB	
HS USB2.0 Host Port	Access to USB1. Type A Socket, 500 mA LS/FS/HS	
Serial Port	UART0 access via 6 pin 3.3V TTL Header. Header is populated	
Ethernet	10/100, RJ45	
SD/MMC Connector	microSD, 3.3V	
User input	Reset Button Boot Button Power Button	
Video out	16b HDMI, 1280 × 1024(MAX) 1024 × 768, 1280 × 720, 1440 × 900, 1920 × 1080@ 24Hz w/EDID support	

	Feature
Audio	Via HDMI interface, stereo
Expansion connectors	Power 5V, 3.3V, VDD_ADC (1.8V) 3.3V I/O on all signals McASP0, SPI1, I2C, GPIO (69 max), LCD, GPMC, MMC1, MMC2, 7 AIN (1.8V MAX), 4 Timers, 4 Serial Ports, CAN0 EHRPWM(0.2), XDMA Interrupt, Power button, Expansion Board ID (UP to 4 can be stacked)
Weight	1.4 oz (39.68 grams)

BeagleBone Black Picture

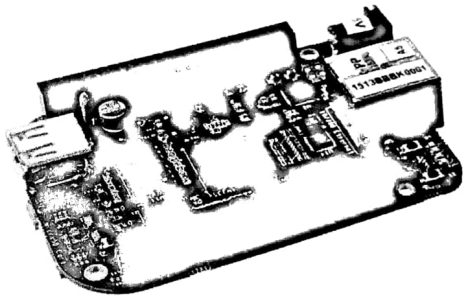


Fig. 14.1.3 : Rev A5A board

14.1.4 Intel : the Edison and the Galileo

- Last but not certainly least, are Intel's Open Source Hardware. Intel currently offers two different type of Open Source Boards: the Edison and the Galileo.
- The Edison Development Platform is designed to help entry level inventors and entrepreneurs mass produce wearable computing merchandise, while the Galileo board is for the advanced user and is used primarily for commercial products.

– The Fig. 14.1.4 shows an image of Intel Edison

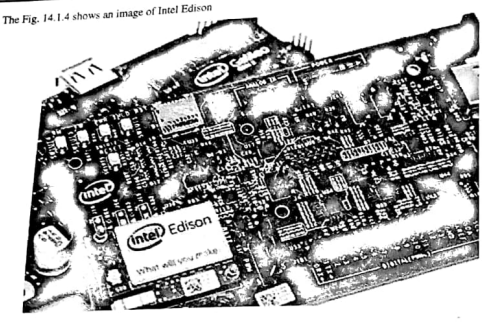


Fig. 14.1.4 : Intel Edison

Intel's boards combines the best of both worlds the inclusion of both a Linux and an Arduino emulator. The Arduino emulator makes the design process run a lot smoother—being able to run the code on an emulator is always a good idea for debugging, especially on a Linux-based operating system since there aren't many restrictions on what you can do. This family of boards also offers the wide range of projects which can be used at a beginner or expert level.

Syllabus Topic : Basic Arduino Programming

14.2 Basic Arduino Programming

Q. 14.2.1 Give the structure of an Arduino program and explain the same with an example. **(5 Marks)**
(Ref. Sec.14.2)

The basic structure of an Arduino program is as shown below:

```
void setup()
{
   statements;
}
void loop()
{
   statements;
}
```


– The programming is quite similar to C programming. The function setup() describes the tasks to be done initially i.e. to initialize the system.

– The statements in the setup() function will be executed only once in the beginning of the program execution.

– The loop() function describes the tasks to be repeatedly performed. This function will be executed continuously after the setup() function is executed once.

– For example, if a pin is to generate a square wave, then the initialization of the pin as output pin will be done in setup() function while the pin given logic high and low alternatively will be done in the loop() function, as shown below:

```
void setup()
{
   pinMode(pin,output);          //This sets the pin as output
}
void loop()
{
   digitalWrite(pin,HIGH);       //This makes the pin to be set to '1'
   delay(1000);                  //This gives a delay of 1000msecs
   digitalWrite(pin,LOW);        //This makes the pin to be reset to '0'
   delay(1000);                  //This gives a delay of 1000msecs
}
```

14.2.1 Functions

– A function is a block of code that has a name and a block of statements that are executed when the function is called. The functions void setup() and void loop() have already been discussed and other built-in functions will be discussed later.

– Custom functions can be written to perform repetitive tasks and reduce clutter in a program. Functions are declared by first declaring the function type.

– This is the type of value to be returned by the function such as 'int' for an integer type function. If no value is to be returned the function type would be void. After type, declare the name given to the function and in parenthesis any parameters being passed to the function.

```
type functionName (parameters)
{
   Statements;
}
```

– The following integer type function delayVal() is used to set a delay value in a program by reading the value of a potentiometer.

– It first declares a local variable v, sets v to the value of the potentiometer which gives a number between 0-1023, then divides that value by 4 for a final value between 0-255, and finally returns that value back to the main program.

```
int delayVal( )
{
  int v;                          // create temporary variable 'v'
  v = analogRead (pot);           //read potentiometer value
  v /= 4;                         // converts 0-1023 to 0-255
  return v;                       // return final value
}
```

14.2.2 Data Types

Q. 14.2.2 List and explain different data types used in Arduino programming. (Ref.Sec.14.2.2) **(5 Marks)**

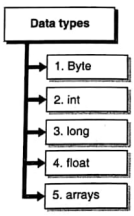


Fig. 14.2.1 : Data types

➜ **Byte**

Byte stores an 8-bit numerical value without decimal points. They have a range of 0-255.

```
byte someVariable = 180;      // declares 'someVariable' as a byte type
```

➜ **int**

Integers are the primary datatype for storage of numbers without decimal points and store a 16-bit value with a range of 32,767 to – 32,768.

```
int someVariable = 1500;      // declares 'someVariable' as an integer type
```

Note : Integer variables will roll over if forced past their maximum or minimum values by an assignment or comparison. For example, if x = 32767 and a subsequent statement adds 1 to x, x = x + 1 or x++, x will then rollover and equal – 32,768.

➜ **long**

Extended size datatype for long integers, without decimal points, stored in a 32-bit value with a range of 2,147, 483.647 to – 2, 147, 483, 648.

```
long someVariable – 90000;          // declares 'someVariable' as a long type
```

➜ **float**

A datatype for floating-point numbers, or numbers that have a decimal point. Floating-point numbers have greater resolution than integers and are stored as a 32-bit value with a range of 3.4028235E+38 to – 3.4028235E+38.

```
float someVariable = 3.14;          // declares 'someVariable' as a floating-point type
```


➜ **arrays**

- An array is a collection of values that are accessed with an index number. Any value in the array may be called upon by calling the name of the array and the index number of the value.
- Arrays are zero indexed, with the first value in the array beginning at index number 0. An array needs to be declared and optionally assigned values before they can be used.
- int myArray [ ] = {value0, value1, value2...}
- Likewise it is possible to declare an array by declaring the array type and size and later assign values to an index position :

```
int myArray [ 5 ];          // declares integer array w/6 positions
myArray [ 3 ] = 10;         // assigns the 4th index the value 10
```

- To retrieve a value from an array, assign a variable to the array and index position :

```
x = myArray [ 3 ];          // x now equals 10
```

- Arrays are often used in for loops, where the increment counter is also used as the index position for each array value. The following example uses an array to flicker an LED.
- Using a for loop, the counter begins at 0, writes the value contained at index position 0 in the array flicker [], in this case 180, to the PWM pin 10, pauses for 200ms, then moves to the next index position.

```
                                    // LED on pin 10
byte flicker [ ] = {180, 30, 255, 200, 10, 90, 150, 60};
                                    // above array of 8 different values
```

Syllabus Topic : Extended Arduino Libraries

14.2.3 Extended Arduino Libraries

Q. 14.2.3 List and explain any 5 extended libraries of Arduino (Ref.Sec.14.2.3) **(10 Marks)**

☞ **pinMode (pin, mode)**

- Used in void setup ( ) to configure a specified pin to behave either as an INPUT or an OUTPUT.

```
pinMode (pin, OUTPUT);          // sets 'pin' to output
```

- Arduino digital pins default to inputs, so they don't need to be explicitly declared as inputs with pinMode( ). Pins configured as INPUT are said to be in a high-impedance state.
- There are also convenient 20 kΩ pullup resistors built into the Atmega chip that can be accessed from software. These built-in pullup resistors are accessed in the following manner:

```
pinMode (pin, INPUT) ;          // set 'pin' to input
digitalWrite (pin, HIGH);       // turn on pullup resistors
```

- Pullup resistors would normally be used for connecting inputs like switches. Notice in the above example it does not convert pin to an output, it is merely a method for activating the internal pull-ups.
- Pins configured as OUTPUT are said to be in a law-impedance state and can provide 40 mA (milliamps) of current to other devices/circuits. This is enough current to brightly light up an LED (don't forgot the series resistor), but not enough current to run most relays, solenoids, or motors.
- Short circuits on Arduino pins and excessive current can damage or destroy the output pin, or damage the entire Atmega chip. It is often a good idea to connect an OUTPUT pin to an external device in series with a 470 Ω or 1 kΩ resistor.

☞ digitalRead(pin)

Reads the value from a specified digital pin with the result either HIGH or LOW.

The pin can be specified as either a variable or constant (0-13).

```
value = digitalRead(Pin);          // sets 'value' equal to
                                   // the input pin
```

☞ digitalWrite(pin, value)

Outputs either logic level HIGH or LOW at (turns on or off) a specified digital pin. The pin can be specified as either a variable or constant (0-13).

```
digitalWrite(pin, HIGH);           // sets 'pin' to high
```

The following example reads a pushbutton connected to a digital input and turns on an LED connected to a digital output when the button has been pressed:

```
int led = 13;                      // connect LED to pin 13
int pin = 7;                       // connect pushbutton to pin 7
int value = 0;                     // variable to store the read value
void setup ( )
{
   pinMode (led, OUTPUT);          // sets pin 13 as output
   pinMode (pin, INPUT);           // sets pin 7 as input
}
void loop ( )
{
   value = digitalRead (pin);      // sets 'value' equal to the input pin
digitalWrite (led, value);         // sets 'led' to the button's value
}
```


☞ analogRead(pin)

Reads the value from a specified analog pin with a 10-bit resolution. This function only works on the analog in pins (0-5). The resulting integer-values range from 0 to 1023.

```
value = analogRead (pin);          // sets 'value' equal to 'pin'
```

☞ analogWrite(pin, value)

- Writes a pseudo-analog value using hardware enabled Pulse Width Modulation (PWM) to an output pin marked PWM. On newer Arduinos with the Atmega 168 chip, this function works on pins 3, 5, 6, 9, 10 and 11.
- Older Arduinos with an Atmega8 only support pins 9, 10 and 11. The value can be specified as a variable or constant with a value from 0-255.

```
analogWrite(pin, value);           // writes 'value' to analog 'pin'
```

- A value of 0 generates a steady 0 volts output at the specified pin; a value of 255 generates a steady 5 volts output at the specified pin. For values in between 0 and 255, the pin rapidly alternates between 0 and 5 volts – the higher the value, the more often the pin is HIGH (5 volts).
- For example, a value of 64 will be 0 volts three-quarters of the time, and 5 volts one quarter of the time; a value of 128 will be at 0 half the time and 255 half the time; and a value of 192 will be 0 volts one quarter of the time and 5 volts three-quarters of the time.
- Because this is a hardware function, the pin will generate a steady wave after a call to analogWrite in the background until the next call to analogWrite (or a call to digitalRead or digitalWrite on the same pin).

☞ delay(ms)

Pauses your program for the amount of time as specified in milliseconds, where 1000 equals 1 second.

```
delay (1000) ;                     // waits for one second
```

☞ millis()

Returns the number of millseconds since the Arduino board began running the current program as an unsigned long value.

```
value = millis ( );                // sets 'value' equal to millis ( )
```

☞ min (x, y)

Calculates the minimum of two numbers of any data type and returns the smaller number.

```
value = min (value, 100);          // sets 'value' to the smaller of
                                   // 'value' or 100, ensuring that
                                   // it never gets above 100.
```

☞ **max(x, y)**

Calculates the maximum of two numbers of any data type and returns the larger number.

```
value = max (value, 100);        // sets 'value' to the larger of 'value' or 100, ensuring that it is at least 100.
```

☞ **randomSeed(seed)**

Sets a value, or seed, as the starting point for the random( ) function.

randomSeed (value); // sets 'value' as the random seed

Because the Arduino is unable to create a truly random number, randomSeed allows you to place a variable, constant, or other function into the random function, which helps to generate more random "random" numbers. There are a variety of different seeds, or functions, that can be used in this function including millis( ) or even analogRead( ) to read electrical noise through an analog pin.

☞ **random(max)**

random(min, max)

The random function allows you to return pseudo-random numbers within a range specified by min and max values.

```
value = random (100, 200);        // sets 'value' to a random number between 100-200
```

☞ **Serial.begin(rate)**

This function opens serial port and sets the baud rate for serial data transmission as specified in the arguments.

```
void setup( )
{
Serial.begin (9600);              // opens serial port sets data rate to 9600 bps
}
```

Note : When using serial communication, digital pins 0 (RX) and 1 (TX) cannot be used at the same time.

☞ **Serial.println(data)**

Prints data to the serial port, followed by an automatic carriage return and line feed. This command takes the same form as Serial.print( ), but is easier for reading data on the Serial Monitor.

```
Serial.println (analogValue);        // sends the value of 'analogValue'
```

Syllabus Topic : Arduino-based Internet Communication

14.2.4 Arduino-based Internet Communication

Q. 14.2.4 Explain how to implement "Arduino based Internet communication". (Ref.Sec.14.2.4) (8 Marks)

To connect Arduino to Web and have internet communication, the following steps are to be done.

Step 1 : Get the login ID, password and IP address of the router connected to internet.

Step 2 : Plug in the Ethernet cable from Router into the Arduino and the USB port of Arduino connected to PC.

Step 3 : Find the IP address of your Arduino and then find a page in the router called "port Forwarding". From here you will have to find the option to "add custom service" or something similar. To setup a new custom service, enter the LAN IP address of your Arduino, choose TCP/UDP for protocol, choose an open port, 8081 - use this number for both the starting port and ending port, and a name for the new service. Once created, your new custom service (port forwarding for your Arduino) it will show up in the port forwarding list.

Step 4 : Save the code given below in your Arduino. This is a code to switch on and off an LED connected on Arduino GPIO pin 2

```
#include <SPI.h>
#include <Ethernet.h>
// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(10, 0, 0, 20);      // IP address, may need to change depending on network
EthernetServer server(80);       // create a server at port 80

String HTTP_req;                 // stores the HTTP request
boolean LED_status = 0;          // state of LED, off by default

void setup()
{
  Ethernet.begin(mac, ip);       // initialize Ethernet device
  server.begin();                // start to listen for clients
  Serial.begin(9600);            // for diagnostics
  pinMode(2, OUTPUT);            // LED on pin 2
}

void loop()
{
  EthernetClient client = server.available();   // try to get client

  if (client) {
    boolean currentLineIsBlank = true;
    while (client.connected()) {
      if (client.available()) {
        char c = client.read();
```

```
        HTTP_req += c;
        if (c == '\n' && currentLineIsBlank) {
          client.println("HTTP/1.1 200 OK");
          client.println("Content-Type: text/html");
          client.println("Connection: close");
          client.println();
          // send web page
          client.println("<!DOCTYPE html>");
          client.println("<html>");
          client.println("<head>");
          client.println("<title>Arduino LED Control</title>");
          client.println("</head>");
          client.println("<body>");
          client.println("<h1>LED</h1>");
          client.println("<p>Click to switch LED on and off.</p>");
          client.println("<form method=\"get\">");
          ProcessCheckbox(client);
          client.println("</form>");
          client.println("</body>");
          client.println("</html>");
          Serial.print(HTTP_req);
          HTTP_req = "";   // finished with request, empty string
          break;
        }
        if (c == '\n') {
          currentLineIsBlank = true;
        }
        else if (c != '\r') {
          // a text character was received from client
          currentLineIsBlank = false;
        }
      }
    }
}
```



```
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  }
}
// switch LED and send back HTML for LED checkbox
void ProcessCheckbox(EthernetClient cl)
{
  if (HTTP_req.indexOf("LED2=2") > -1) {  // see if checkbox was clicked
    // the checkbox was clicked, toggle the LED
    if (LED_status) {
      LED_status = 0;
    }
    else {
      LED_status = 1;
    }
  }

  if (LED_status) {    // switch LED on
    digitalWrite(2, HIGH);
    // checkbox is checked
    cl.println("<input type=\"checkbox\" name=\"LED2\" value=\"2\" \
    onclick=\"submit();\" checked>LED2");
  }
  else {             // switch LED off
    digitalWrite(2, LOW);
    // checkbox is unchecked
    cl.println("<input type=\"checkbox\" name=\"LED2\" value=\"2\" \
    onclick=\"submit();\">LED2");
  }
}
```

Step 5 : In a Web browser navigate to "http://xx.xxx.xx.xx:8081/" , connect the hardware and check the working

Syllabus Topic : Raspberry PI

14.3 Raspberry PI

Q. 14.3.1 Write a short note on Raspberry Pi and various interfaces to it (Ref.Sec.14.3) **(8 Marks)**

Raspberry Pi as discussed earlier has a lot of functionality. Infact, it can be said that the small card sized Raspberry Pi is an entire computer in itself.

14.3.1 Connecting Video

- You need to connect a display before you can start using your Raspberry Pi. The Raspberry Pi has an HDMI port which you can connect directly to a monitor or TV with an HDMI cable. This is the easiest solution; some modern monitors and TVs have HDMI ports, some do not, but there are other options. HDMI Video port found on the bottom of the Pi (see Fig. 14.3.1) gives a better-quality and picture.
- All models of Raspberry Pi have a composite out port for connecting to analog devices, but the type of connector varies depending on the model. The original Raspberry Pi used an RCA connector, and a standard RCA composite video lead will work. Others models (Raspberry Pi B+ and later) combine the audio out and composite out on to the same 3.5mm jackplug.

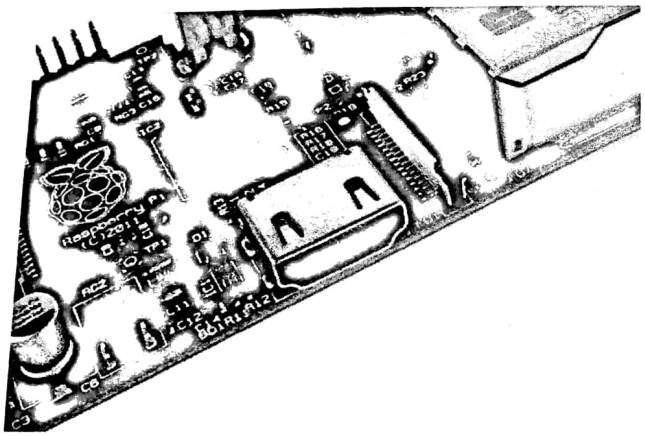


Fig. 14.3.1 : Backside of a Raspberry Pi showing HDMI port

14.3.2 Connecting Audio

- If you're using the Raspberry Pi's HDMI port, the same will be used for audio output also. But in case, if you want the audio output, you can take it from the audio connector.
- If you are using any other video communication port (like VGA or RCA, etc), then you need to connect a separate cable from the audio jack to the speakers.

14.3.3 Connecting a Keyboard and Mouse

- After the output devices like video and audio are connected, we need to interface the input devices to the Pi.
- The keyboard and mouse need to be USB based ports. They can be used to give the input to the Pi. If required to connect multiple USB devices, you can use a USB hub to increase the number of USB ports.

14.3.4 SD Card

- The Raspberry Pi works with any SD card, but with some guidelines.
- The card should be minimum 4 to 8 GB. It should be full sized in model B, while the other can work with micro SD card.

14.3.5 Connecting External Storage

- While the Raspberry Pi uses an SD card for its main storage device—known as a boot device—you may find that you run into space limitations.
- USB Mass Storage (UMS) devices, these can be physical hard drives, solid-state drives (SSDs) or even portable pocket-sized flash drives can be used to increase the storage capacity.

14.3.6 Connecting to the Network

- An Ethernet cable can be used to connect your Pi to the network. This facility is available in models BB+/2/3 only.
- The Pi by itself doesn't have a wireless modem, but an external USB wireless dongle can be connected in case of a wireless network is to be connected.

14.3.7 Connecting Power

- The Raspberry Pi is powered by the small micro-USB connector found on the lower left side of the circuit board The Pi is powered by a USB Micro power supply (like most standard mobile phone chargers).
- A good-quality power supply that can supply at least 2A at 5V for the Model 3B, or 700mA at 5V for the earlier, lower powered models will be required. Low current (~700mA) power supplies will work for basic usage, but are likely to cause the Pi to reboot if it draws too much power

Syllabus Topic : ARM Cortex Processors

14.4 ARM Cortex Processors

Q. 14.4.1 Explain ARM Cortex Processors. (Ref. Sec. 14.4) **(5 Marks)**

- Cortex R processors are used in the embedded applications where mixed signal processing is possible. There are various places where this mixed signal processing is required like various household appliances (refrigerator control, freezers, washing machine, etc), consumer products, medical instrumentation etc.
- The various processors in the series of Cortex-M are M0, M0+, M3, M4 and M7.
- The Cortex-M0 has the lowest area and cost the Cortex-M0+ is the lowest power consuming and hence most energy efficient. The ARM Cortex-M0 processor is suitable for applications like touchscreen controller and power management. The ARM Cortex-M0+ processor is suitable for applications like bluetooth smart transceiver, etc.
- ARM Cortex-M3 is performance efficiency and has large set of features for connectivity. It is very efficient in terms of various protocols it supports for connectivity. The applications for which ARM Cortex-M3 processor is most suitable are WiFi transceiver, activity tracker etc.
- ARM Cortex-M4 is a processor with DSP features and also supports SIMD and floating point operations. The ARM Cortex-M4 processor is most suitable for smart metering or other meter applications
- ARM Cortex-M7 is a processor with maximum digital signal computing performance. It has a flexible memory organization. It has cache memory, error coding control etc. It supports single as well as double precision of floating point standards for the data. The ARM Cortex-M7 processors are most suitable for applications like high end audio headset, automotive electronics etc.

14.5 Comparison of ARM-v7-A (CortexA8), ARM-v7-R (CortexR4), ARM-v7-M (Cortex-M3)

Q. 14.5.1 Compare various ARM cortex series. (Ref. Sec.14.5) **(4 Marks)**

- There are various Cortex series of ARM processor based on the special features provided in the processor. The Cortex-A series of microcontrollers are for applications that require a lot of computation and controlling. The Cortex-R are especially meant for the real time based applications.
- The Cortes-M processors are meant for the special category of microcontrollers with low power consumption and hence are also called as the future microcontrollers. They have a wide range of features that are improved over the previous versions. Each of the microcontrollers have some or the other improvement over its predecessor.

14.5.1 Cortex-A Series

- The ARM cortex-A series processors are a series of application processors used in a wide range of devices that host a rich OS.
- The applications of the Cortex A series ranges from low cost handset to the smart phones. They are used in mobile computing, digital TV and set-top boxes, networking, printers etc.

- Cortex-A processors are used in embedded systems that have high-computational requirements and run on rich operating systems. They are also used to deliver interactive media and graphics experience from the latest technological mobile devices that need to have internet access such as handsets and ultra-portable SmartBooks to automotive information and entertainment systems and next generation digital TV systems. Cortex-A processors are also geared towards providing the full internet experience, but are used in a wide variety of applications.

The following is a list of application areas of Cortex A processors:

1.	Smartbook	2.	Tablet	3.	E-reader		
4.	Thin client	5.	Smart Phones	6.	Feature phones		
7.	Set top box	8.	Digital TV	9.	Blu Ray player		
10.	Gaming consoles	11.	Navigation	12.	Printers	13.	Routers, etc

14.5.2 Cortex-R Series

The ARM Cortex-R series is a real time processor series that have high performance and computing solutions for applications that require high reliability, availability, fault tolerance and maintainability. Thus the applications that require ARM Cortex-R series processors are mainly such that they require the following attributes:

1. **High Performance :** The processors are fast in processing as well as have high clock frequency.
2. **Real-time :** The processors meet the real time deadline constraints in all cases.
3. **Safe :** The processors are reliable and dependable
4. **Cost effective :** It requires low power and physical area and hence are cost effective.

The following is a list of application areas of Cortex R processors:

1.	Vehicle airbag	2.	Vehicle braking
3.	HDD (Hard Disk Drive) Controller	4.	3G, 4G mobiles
5.	Smart phones	6.	Modems
7.	Medical and industrial applications	8.	Networking
9.	Digital camera	10.	Printers, etc

14.5.3 Cortex-M Series

The Cortex-M processors are meant mainly for mixed signal processing. The ARM Cortex M series processors are very high end processors designed to cater the needs of the future upcoming embedded system applications. They are designed for mixed signal processing applications such as smart metering, human interface devices, consumer instruments, medical instruments, etc.

The main advantages of ARM Cortex-M series are given below :

1. **Energy efficiency :** These processors have low energy costs and longer battery life
2. **It requires smaller code :** Since the instruction density is higher, the code is smaller i.e. occupies less memory. Hence less cost of the memory.
3. **Ease of use :** Since there are lot of tools available, the designing and development is much easy.
4. **High Performance :** It delivers rich performance at low power.

Syllabus Topic : Intel Galileo Boards

14.6 Intel Galileo Boards

Q. 14.6.1 Write a short note on Intel Galileo board. (Ref. Sec.14.6) (4 Marks)

- The Intel® Galileo Board provides a programmable control PCB for the maker community, students, and professional developers. It is based on the Intel® Quark™ SoC X1000 Application Processor, a 32-bit Intel Pentium-class system on a chip.
- The Intel® Galileo Board is the first board based on Intel® architecture designed to be hardware and software pin-compatible with Arduino shields designed for the Uno R3.
- It is also software-compatible with the Arduino* Software Development Environment, making usability and development a snap. In addition to Arduino hardware and software compatibility, the Intel® Galileo Board has several industry-standard I/O ports and features to expand native usage and capabilities beyond the Arduino shield ecosystem.
- Fig. 14.6.1 shows the key components of Intel Galileo board.

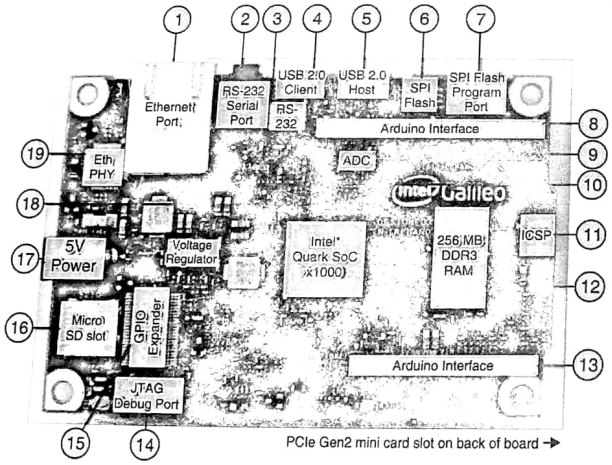


Fig. 14.6.1 : Key components of Intel Galileo

The table 14.6.1 gives the description of Intel Galileo components.

Table 14.6.1 : Description of Intel Galileo components

Number	Component	Description
1.	Ethernet Port	10/100 Ethernet connector
2.	RS-232 Serial Port	3-pin 3.5mm jack (not audio)
3.	RS-232	RS-232 transceiver
4.	USB 2.0 Client	USB Client connector (Micro-USB Type B) : a fully compliant USB 2.0 Device controller, typically used for programming.
5.	USB 2.0 Host	USB 2.0 Host connector (Micro-USB Type AB) : supports up to 128 USB end point devices.
6.	SPI Flash	8 MByte Legacy SPI Flash to store the firmware (or bootloader) and the latest sketch.
7.	SPI Flash Program Port	7-pin header for Serial Peripheral Interface (SPI) programming. Defaults to 4 MHz to support Arduino Uno shields, Programmable up to 25 MHz. **Note :** The board has a native SPI controller, however, it will act as a master and not as an SPI slave. Therefore, it cannot be a SPI slave to another SPI master. It can act, however, as a slave device via the USB Client connector.
8.	Shield Interface	Complies with Arduino Uno Revision 3 shield pinout.
9.	ADC	Analog to Digital converter.
10.	Intel® Quark™ SoC X1000	400 MHz 32-bit Intel® Pentium instruction set architecture (ISA) – compatible processor – 16 Kbyte L1 cache – 512 Kbytes of on-die embedded SRAM. – Simple to program: Single thread, single core, constant speed – ACPI compatible CPU sleep states supported – An integrated Real Time Clock (RTC), with an optional 3V "coin cell" battery for operation between turn on cycles.
11.	ICSP	6-pin In-Circuit Serial Programming (ICSP) header, located appropriately to plug into existing shields. These pins support SPI communication using the SPI library.
12.	256 MB DDR3 RAM	256 MByte DRAM, enabled by the firmware by default.
13.	Arduino Interface	Complies with Arduino Uno Revision 3 pinout.
14.	JTAG Debug Port	10-pin standard JTAG header for debugging
15.	GPIO Expander	GPIO pulse with modulation provided by a single I^2C I/O expander.

Number	Component	Description
16.	Micro SD slot	(Optional) Supports micro SD card up to 32 GBytes
17.	5V Power	The board is powered via an AC-to-DC adapter, connected by plugging a 2.1 mm center-positive plug into the board's power jack. The recommended output rating of the power adapte is 5V at up to 3A.
18.	Voltage Regulator	Generates 3.3 volt supply. Maximum current draw to the shield is 800 mA.
19.	Eth PHY	Ethernet Physical layer transceiver.

14.6.1 Arduino Connector Pinout Details

The Intel® Galileo Board is designed to support shields that operate at either 3.3V or 5V. The core operating voltage of Intel® Galileo Board is 3.3V; however, a jumper on the board enables voltage translation to 5V at the I/O pins.

The Intel® Galileo Board complies with the Arduino Uno Revision 3 pinout as follows :

- **14 digital input/output pins (IO2-IO13, TX, RX) :** Each of the 14 digital pins on Galileo can be used as an input or output, using pinMode(), digitalWrite(), and digitalRead() functions.
- The pins operate at 3.3 volts or 5 volts. Each pin can source a max of 10 mA or sink a maximum of 25 mA and has an internal pull-up resistor (disconnected by default) of 5.6 k to 10 kOhms.
- **6 digital pins can be used as Pulse Width Modulation (PWM) outputs :** The RX and TX pins control the programmable speed UART port
- **SCL and SDA pins control the I2C* bus. TWI :** A4 or SDA pin and A5 or SCL pin. TWI communication is supported via the Arduino Wire library. AREF is unused. Providing an external reference voltage for the analog inputs is not supported.
- **6 analog input pins (A0-A5) :** Each one of the 6 analog input pins provides 12 bits of resolution (that is, 4096 different values). By default, they measure from ground to 5 volts.
- **7 power pins :** IOREF: The IOREF pin allows an attached shield with the proper configuration to adapt to the voltage provided by the board. The IOREF pin voltage is controlled by a jumper on the board, i.e., a selection jumper on the board is used to select between 3.3 V and 5 V shield operation.
- RESET button/pin: Bring this line LOW to reset the sketch. Typically used to add a reset button to shields that block the one on the board.
- 3.3V output pin: A 3.3 Volt supply generated by the on-board regulator. Maximum current draw to the shield is 800 mA.
- 5V output pin: This pins outputs 5 V from the external source or the USB connector. Maximum current draw to the shield is 800 mA.
- GND (2 pins): Ground pins.
- VIN: The input voltage to the Intel® Galileo Board when it is using an external power source (as opposed to 5 Volts from the regulated power supply connected at the power jack).

14.6.2 Programming Intel Galileo

Q. 14.6.2 Explain the steps for programming Intel Galileo. (Ref. Sec.14.6.2) **(3 Marks)**

Use the Arduino Software Development Environment to create programs, called sketches, for the Intel® Galileo Board. To run a sketch on the board :

1. Connect a power supply.
2. Connect the board's USB client port to a computer.
3. Upload the sketch using the IDE interface.

The sketch runs on the Intel® Galileo Board and communicates with the Linux* kernel in the board's firmware using the Arduino I/O adapter. For complete details on programming your board.

When the board boots up, two scenarios are possible : If a sketch is present in persistent storage, it is executed. If no sketch present, the board waits for upload commands from the IDE.

If a sketch is executing, you can upload from the IDE without having to press the reset button on the board. The sketch is stopped; the IDE waits for the upload state, and then starts the newly uploaded sketch. Pressing the reset button on the board restarts a sketch if it is executing and resets any attached shields.

Syllabus Topic : Sensors and Interfacing : Temperature, Pressure, Humidity

14.7 Sensors and Interfacing : Temperature, Pressure, Humidity

We have seen the basics of Arduino programming. Let us discuss in detail two mini projects for and Arduino based system.

14.7.1 Temperature and Humidity sensor DHT11 interfacing with Arduino

Q. 14.7.1 Draw and explain the interfacing of a temperature and humidity sensor with Arduino. Also write a program for the same. (Ref. Sec.14.7.1) **(10 Marks)**

- Humidity and temperature are common parameters to measure environmental conditions. In this Arduino based project we are going to measure ambient temperature and humidity and display it on a 16x2 LCD screen.
- A combined temperature and himidity sensor DHT11 is used with Arduino uno to develop this Celsius scale thermometer and percentage scale humidity measurement project.
- The block diagram of the system can be as shown in Fig. 14.7.1.

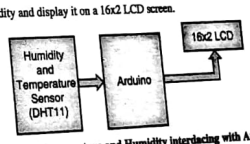


Fig. 14.7.1 : Temperature and Humidity interfacing with Arduino

- Here in this project we have used a sensor module namely DHT11. This module features a humidity and temperature complex with a calibrated digital signal output means DHT11 sensor module is a combined module for sensing humidity and temperature which gives a calibrated digital output signal.
- DHT11 gives us very precise value of humidity and temperature and ensures high reliability and long term stability. This sensor has a resistive type humidity measurement component and NTC type temperature measurement component

with an 8-bit microcontroller inbuilt which has a fast response and cost effective and available in 4-pin single row package.

- DHT11 module works on serial communication i.e. single wire communication. This module sends data in form of pulse train of specific time period. Before sending data to arduino it needs some initialize command with a time delay. And the whole process time is about 4ms. A complete data transmission is of 40-bit and data format.
- The circuit connection for the same can be as shown in Fig. 14.7.2.

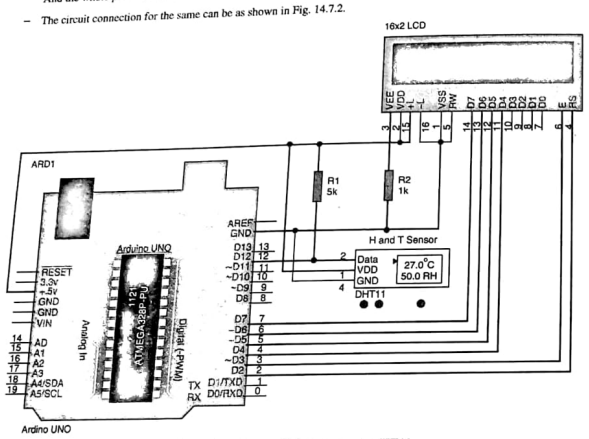


Fig. 14.7.2 : Interfacing of ARduino with DHT11

The following code can be used for the measurement and display of temperature and humidity.

- First we need to include the header file ('#include<LiquidCrystal.h>'), which has functions written in it, that enables the user to interface an LCD to UNO in 4 bit mode. With this header file we need not have to send data to LCD bit by bit, this will all be taken care of and we don't have to write a program for sending data or a command to LCD bit by bit.
- Second we need to tell the board which type of LCD we are using here. Since we have so many different types of LCD (like 20*4, 16*2, 16*1 etc.). In here we are going to interface a 16*2 LCD to the UNO so we get 'lcd.begin(16,2);'. For 16*1 we get 'lcd.begin(16,1);'.
- In this instruction we are going to tell the board where we connected the pins, The pins which are connected are to be represented in order as "RS, En, D4, D5, D6, D7". These pins are to be represented correctly. Since we connected RS to PIN0 and so on as show in circuit diagram, We represent the pin number to board as "Liquid Crystal lcd (0,1,8,9,10,11);".


```
#include<dht.h>                    // Including library for dht
#include<LiquidCrystal.h>          //Including library for LCD display
LiquidCrystal lcd(2, 3, 4, 5, 6, 7); //Port connection for LCD display
#define dht_dpin 12                //Digital data pin for DHT11
dht DHT;
byte degree[8] =
      {
       0b00011,
       0b00011,
       0b00000,
       0b00000,
       0b00000,
       0b00000,
       0b00000,
       0b00000
      };
void setup()
{
 lcd.begin(16, 2);                 //Initializing LCD
 lcd.createChar(1, degree);
 lcd.clear();                      //Clear LCD display
 lcd.print("  Humidity  ");        //Display Humidity on LCD display
 lcd.setCursor(0,1);               //Position the cursor
 lcd.print("  Measurement ");      //Display the word measurement
 delay(2000);                      //Delay
 lcd.clear();                      //Clear display
}
void loop()
{
 DHT.read11(dht_dpin);             //Take the reading from DHT11
 lcd.setCursor(0,0);               //Position the cursor
 lcd.print("Humidity: ");          //Display the word Humidity
 lcd.print(DHT.humidity);          //Display humidity read from DHT11 on LCD
 lcd.print(" %");                  //Display the symbol %
 lcd.setCursor(0,1);               //Position the cursor
 lcd.print("Temperature:  ");      //Display the word Temperature
 lcd.print(DHT.temperature);       // Display temperature read from DHT11 on LCD
 lcd.print("Cel");                 //Display the unit Cel
 delay(500);                       //Delay before next reading in loop
}
```

14.7.2 Pressure sensor BMP180 Interfacing with Arduino

Q. 14.7.2 Draw and explain the interfacing of a pressure sensor with Arduino. Also write a program for the same. (Ref. Sec.14.7.2) **(10 Marks)**

The sensor BMP180 is used to measure temperature as well as pressure. We will see in this section how to measure pressure using this sensor. The circuit connection for interfacing BMP180 with Arduino is shown in Fig. 14.7.3.

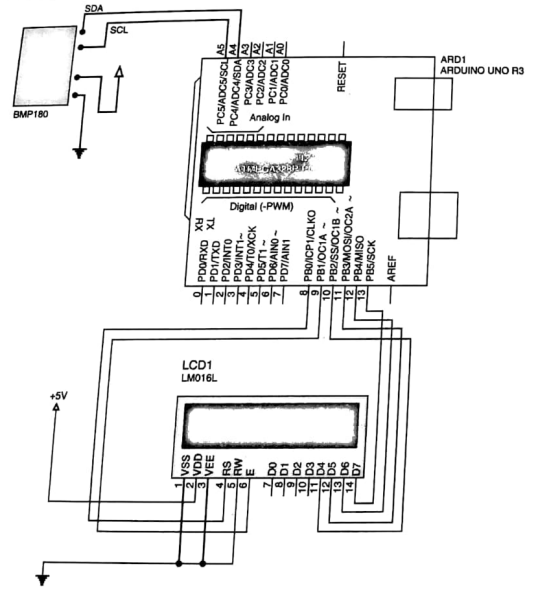


Fig. 14.7.3 : Interfacing Pressure sensor BMP180 with Arduino

The interfacing of LCD is in the same manner as in the previous section. Since this is a SPI-capable sensor, we can use hardware or 'software' SPI. To make wiring identical on all Arduinos, we'll begin with 'software' SPI.

The following pins should be used:

- Connect Vin to the power supply, 3V or 5V is fine. Use the same voltage that the microcontroller logic is based off of. For most Arduinos, that is 5V
- Connect GND to common power/data ground
- Connect the SCK pin to Digital #13 but any pin can be used later.

The code for displaying pressure using BMP180 is as given below.

```
#include <Adafruit_BMP085.h>          //Library functions for Sensor BMP180
#include <Wire.h>                     //Library Functions for SPI communication
#include <LiquidCrystal.h>            //Library functions for LCD display
LiquidCrystal lcd(8, 9, 10, 11, 12, 13);    //Pins used for LCD

char PRESSURESHOW[4];                 // initializing a character of size 4 for showing the  result

void setup() {
        lcd.begin(16, 2);             //Initialize LCD
        lcd.clear();                  //clear display
        Serial.begin(9600);           //Initialize SPI port baud rate
        if (!bmp.begin())
        {
                Serial.println("ERROR");    //if there is an error in communication
                while (1) {}
        }
}

void loop()
{
        lcd.print("Pressure= ");       // Display the word pressure on LCD
        String PRESSUREVALUE = String(bmp.readPressure());
        // convert the reading to a char array
        PRESSUREVALUE.toCharArray(PRESSURESHOW, 4);
        lcd.print(PRESSURESHOW);      //Display the read value of pressure
        lcd.print("hPa ");            //Display the units for pressure
        lcd.setCursor(0, 0);          // set the cursor to column 0, line1
        delay(1000);                  //Small delay before beginning with next reading
}
```

14.8 Exam Pack (Review Questions)

☞ **Syllabus Topic : Introduction to Arduino, Raspberry PI, ARM Cortex, Intel Galileo etc. , Open-source prototyping platform**

Q. 1 Compare various models of Raspberry Pi. **(Refer Section 14.1.1)** **(5 Marks)**

Q. 2 Compare various models of Arduino. **(Refer Section 14.1.2)** **(5 Marks)**

Q. 3 List the features of Beagleboard. **(Refer Section 14.1.3)** **(5 Marks)**

☞ **Syllabus Topic : Basic Arduino Programming**

Q. 4 Give the structure of an Arduino program and explain the same with an example. **(Refer Section 14.2)** **(5 Marks)**

Q. 5 List and explain different data types used in Arduino programming. **(Refer Section 14.2.2)** **(5 Marks)**

☞ **Syllabus Topic : Extended Arduino Libraries**

Q. 6. List and explain any 5 extended libraries of Arduino **(Refer Section 14.2.3)** **(10 Marks)**

☞ **Syllabus Topic : Arduino-based Internet Communication**

Q. 7 Explain how to implement "Arduino based Internet communication". **(Refer Section 14.2.4)** **(8 Marks)**

☞ **Syllabus Topic : Raspberry PI**

Q. 8 Write a short note on Raspberry Pi and various interfaces to it **(Refer Section 14.3)** **(8 marks)**

☞ **Syllabus Topic : ARM Cortex Processors**

Q. 9 Explain ARM Cortex Processors. **(Refer Section 14.4)** **(5 Marks)**

Q. 10 Compare various ARM cortex series **(Refer Section 14.5)** **(4 Marks)**

☞ **Syllabus Topic : Intel Galileo Boards**

Q. 11 Write a short note on Intel Galileo board. **(Refer Section 14.6)** **(4 Marks)**

Q. 12 Explain the steps for programming Intel Galileo. **(Refer Section 14.6.2)** **(3 Marks)**

☞ **Syllabus Topic : Sensors and Interfacing : Temperature, Pressure, Humidity**

Q. 13 Draw and explain the interfacing of a temperature and humidity sensor with Arduino. Also write a program for the same. **(Refer Section 14.7.1)** **(10 Marks)**

Q. 14 Draw and explain the interfacing of a pressure sensor with Arduino. Also write a program for the same **(Refer Section 14.7.2)** **(10 Marks)**

Chapter Ends...

□□□